

27 | 并发工具类模块热点问题答疑

2019-04-30 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 09:34 大小 8.78M



前面我们用 13 篇文章的内容介绍了 Java SDK 提供的并发工具类，这些工具类都是久经考验的，所以学好用好它们对于解决并发问题非常重要。我们在介绍这些工具类的时候，重点介绍了这些工具类的产生背景、应用场景以及实现原理，目的就是让你在面对并发问题的时候，有思路，有办法。只有思路、办法有了，才谈得上开始动手解决问题。

当然了，只有思路和办法还不足以把问题解决，最终还是要动手实践的，我觉得在实践中有两方面的问题需要重点关注：**细节问题与最佳实践**。千里之堤毁于蚁穴，细节虽然不能保证成功，但是可以导致失败，所以我们一直都强调要关注细节。而最佳实践是前人的经验总结，可以帮助我们不要阴沟里翻船，所以没有十足的理由，一定要遵守。


为了让你学完即学即用，我在每篇文章的最后都给你留了道思考题。这 13 篇文章的 13 个思考题，基本上都是相关工具类在使用中需要特别注意的一些细节问题，工作中容易碰到且

费神费力，所以咱们今天就来一一分析。

1. while(true) 总不让人省心

《14 | Lock&Condition (上)：隐藏在并发包中的管程》的思考题，本意是通过破坏不可抢占条件来避免死锁问题，但是它的实现中有一个致命的问题，那就是：while(true) 没有 break 条件，从而导致了死循环。除此之外，这个实现虽然不存在死锁问题，但还是存在活锁问题的，解决活锁问题很简单，只需要随机等待一小段时间就可以了。

修复后的代码如下所示，我仅仅修改了两个地方，一处是转账成功之后 break，另一处是在 while 循环体结束前增加了 Thread.sleep(随机时间)。

 复制代码

```
1 class Account {
2     private int balance;
3     private final Lock lock
4         = new ReentrantLock();
5     // 转账
6     void transfer(Account tar, int amt){
7         while (true) {
8             if(this.lock.tryLock()) {
9                 try {
10                     if (tar.lock.tryLock()) {
11                         try {
12                             this.balance -= amt;
13                             tar.balance += amt;
14                             // 新增：退出循环
15                             break;
16                         } finally {
17                             tar.lock.unlock();
18                         }
19                     }//if
20                 } finally {
21                     this.lock.unlock();
22                 }
23             }//if
24             // 新增：sleep 一个随机时间避免活锁
25             Thread.sleep(随机时间);
26         }//while
27     }//transfer
28 }
```

这个思考题里面的 while(true) 问题还是比较容易看出来的，**但不是所有的 while(true) 问题都这么显而易见的**，很多都隐藏得比较深。


例如，[《21 | 原子类：无锁工具类的典范》](#)的思考题本质上也是一个 while(true)，不过它隐藏得就比较深了。看上去 while(!rf.compareAndSet(or, nr)) 是有终止条件的，而且跑单线程测试一直都没有问题。实际上却存在严重的并发问题，问题就出在对 or 的赋值在 while 循环之外，这样每次循环 or 的值都不会发生变化，所以一旦有一次循环 rf.compareAndSet(or, nr) 的值等于 false，那之后无论循环多少次，都会等于 false。也就是说在特定场景下，变成了 while(true) 问题。既然找到了原因，修改就很简单了，只要把对 or 的赋值移到 while 循环之内就可以了，修改后的代码如下所示：

复制代码

```
1 public class SafeWM {
2     class WMRRange{
3         final int upper;
4         final int lower;
5         WMRRange(int upper,int lower){
6             // 省略构造函数实现
7         }
8     }
9     final AtomicReference<WMRange>
10     rf = new AtomicReference<>()
11         new WMRRange(0,0)
12     );
13     // 设置库存上限
14     void setUpper(int v){
15         WMRRange nr;
16         WMRRange or;
17         // 原代码在这里
18         //WMRange or=rf.get();
19         do{
20             // 移动到此处
21             // 每个回合都需要重新获取旧值
22             or = rf.get();
23             // 检查参数合法性
24             if(v < or.lower){
25                 throw new IllegalArgumentException();
26             }
27             nr = new
28                 WMRRange(v, or.lower);
29         }while(!rf.compareAndSet(or, nr));
30     }
31 }
```

2. signalAll() 总让人省心

《15 | Lock&Condition (下) : Dubbo 如何用管程实现异步转同步? 》的思考题是关于 signal() 和 signalAll() 的, Dubbo 最近已经把 signal() 改成 signalAll() 了, 我觉得用 signal() 也不能说错, 但的确是用 **signalAll() 会更安全**。我个人也倾向于使用 signalAll(), 因为我们写程序, 不是做数学题, 而是在搞工程, 工程中会有很多不稳定的因素, 更有很多你预料不到的情况发生, 所以不要让你的代码铤而走险, 尽量使用更稳妥的方案和设计。Dubbo 修改后的相关代码如下所示:

 复制代码

```
1 // RPC 结果返回时调用该方法
2 private void doReceived(Response res) {
3     lock.lock();
4     try {
5         response = res;
6         done.signalAll();
7     } finally {
8         lock.unlock();
9     }
10 }
```

3. Semaphore 需要锁中锁

《16 | Semaphore: 如何快速实现一个限流器? 》的思考题是对象池的例子中 Vector 能否换成 ArrayList, 答案是不可以的。Semaphore 可以允许多个线程访问一个临界区, 那就意味着可能存在多个线程同时访问 ArrayList, 而 ArrayList 不是线程安全的, 所以对象池的例子中是不能够将 Vector 换成 ArrayList 的。**Semaphore 允许多个线程访问一个临界区, 这也是一把双刃剑**, 当多个线程进入临界区时, 如果需要访问共享变量就会存在并发问题, 所以必须**加锁**, 也就是说 Semaphore 需要锁中锁。

4. 锁的申请和释放要成对出现

《18 | StampedLock: 有没有比读写锁更快的锁? 》思考题的 Bug 出在没有正确地释放锁。锁的申请和释放要成对出现, 对此我们有一个最佳实践, 就是使用 **try{}finally{}** , 但是 try{}finally{} 并不能解决所有锁的释放问题。比如示例代码中, 锁的升级会生成新的 stamp, 而 finally 中释放锁用的是锁升级前的 stamp, 本质上这也属于锁的申请和释放没有成对出现, 只是它隐藏得有点深。解决这个问题倒也很简单, 只需要对 stamp 重新赋值就可以了, 修复后的代码如下所示:


```
1 private double x, y;
2 final StampedLock sl = new StampedLock();
3 // 存在问题的方法
4 void moveIfAtOrigin(double newX, double newY){
5     long stamp = sl.readLock();
6     try {
7         while(x == 0.0 && y == 0.0){
8             long ws = sl.tryConvertToWriteLock(stamp);
9             if (ws != 0L) {
10                 // 问题出在没有对 stamp 重新赋值
11                 // 新增下面一行
12                 stamp = ws;
13                 x = newX;
14                 y = newY;
15                 break;
16             } else {
17                 sl.unlockRead(stamp);
18                 stamp = sl.writeLock();
19             }
20         }
21     } finally {
22         // 此处 unlock 的是 stamp
23         sl.unlock(stamp);
24     }
```

5. 回调总要关心执行线程是谁

《19 | CountdownLatch 和 CyclicBarrier: 如何让多线程步调一致?》的思考题是: CyclicBarrier 的回调函数使用了一个固定大小为 1 的线程池, 是否合理? 我觉得是合理的, 可以从以下两个方面来分析。

第一个是线程池大小是 1, 只有 1 个线程, 主要原因是 check() 方法的耗时比 getPOrders() 和 getDOrders() 都要短, 所以没必要用多个线程, 同时单线程能保证访问的数据不存在并发问题。

第二个是使用了线程池, 如果不使用, 直接在回调函数里调用 check() 方法是否可以呢? 绝对不可以。为什么呢? 这个要分析一下回调函数和唤醒等待线程之间的关系。下面是 CyclicBarrier 相关的源码, 通过源码你会发现 CyclicBarrier 是同步调用回调函数之后才唤醒等待的线程, 如果我们在回调函数里直接调用 check() 方法, 那就意味着在执行 check() 的时候, 是不能同时执行 getPOrders() 和 getDOrders() 的, 这样就起不到提升性能的作用。

```

1 try {
2     //barrierCommand 是回调函数
3     final Runnable command = barrierCommand;
4     // 调用回调函数
5     if (command != null)
6         command.run();
7     ranAction = true;
8     // 唤醒等待的线程
9     nextGeneration();
10    return 0;
11 } finally {
12     if (!ranAction)
13         breakBarrier();
14 }

```

所以，当遇到回调函数的时候，你应该本能地问自己：执行回调函数的线程是哪一个？这个在多线程场景下非常重要。因为不同线程 ThreadLocal 里的数据是不同的，有些框架比如 Spring 就用 ThreadLocal 来管理事务，如果不清楚回调函数用的是哪个线程，很可能会导致错误的事务管理，并最终导致数据不一致。

CyclicBarrier 的回调函数究竟是哪个线程执行的呢？如果你分析源码，你会发现执行回调函数的线程是将 CyclicBarrier 内部计数器减到 0 的那个线程。所以我们前面讲执行 check() 的时候，是不能同时执行 getPOrders() 和 getDOrders()，因为执行这两个方法的线程一个在等待，一个正在忙着执行 check()。

再次强调一下：**当看到回调函数的时候，一定问一问执行回调函数的线程是谁。**

6. 共享线程池：有福同享就要有难同当

《24 | CompletableFuture：异步编程没那么难》的思考题是下列代码是否有问题。很多同学都发现这段代码的问题了，例如没有异常处理、逻辑不严谨等等，不过我更想让你关注的是：findRuleByJdbc() 这个方法隐藏着一个阻塞式 I/O，这意味着会阻塞调用线程。默认情况下所有的 CompletableFuture 共享一个 ForkJoinPool，当有阻塞式 I/O 时，可能导致所有的 ForkJoinPool 线程都阻塞，进而影响整个系统的性能。

```

1 // 采购订单
2 PurchersOrder po;

```

```
3 CompletableFuture<Boolean> cf =
4     CompletableFuture.supplyAsync(()->{
5         // 在数据库中查询规则
6         return findRuleByJdbc();
7     }).thenApply(r -> {
8         // 规则校验
9         return check(po, r);
10    });
11 Boolean isOk = cf.join();
```

利用共享，往往能让我们快速实现功能，所谓是有福同享，但是代价就是有难要同当。在强调高可用的今天，大多数人更倾向于使用隔离的方案。

7. 线上问题定位的利器：线程栈 dump

[《17 | ReadWriteLock：如何快速实现一个完备的缓存？》](#)和[《20 | 并发容器：都有哪些“坑”需要我们填？》](#)的思考题，本质上都是定位线上并发问题，方案很简单，就是通过查看线程栈来定位问题。重点是查看线程状态，分析线程进入该状态的原因是否合理，你可以参考[《09 | Java 线程（上）：Java 线程的生命周期》](#)来加深理解。

为了便于分析定位线程问题，你需要给线程赋予一个有意义的名字，对于线程池可以通过自定义 ThreadFactory 来给线程池中的线程赋予有意义的名字，也可以在执行 run() 方法时通过 Thread.currentThread().setName(); 来给线程赋予一个更贴近业务的名字。

总结

Java 并发工具类到今天为止，就告一段落了，由于篇幅原因，不能每个工具类都详细介绍。Java 并发工具类内容繁杂，熟练使用是需要一个过程的，而且需要多加实践。希望你学完这个模块之后，遇到并发问题时最起码能知道用哪些工具可以解决。至于工具使用的细节和最佳实践，我总结的也只是我认为重要的。由于每个人的思维方式和编码习惯不同，也许我认为不重要的，恰恰是你的短板，所以这部分内容更多地还是需要你去实践，在实践中养成良好的编码习惯，不断纠正错误的思维方式。

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。

© 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 26 | Fork/Join：单机版的MapReduce

下一篇 28 | Immutability模式：如何利用不变性解决并发问题？

精选留言 (9)

写留言



Sunqc

2019-04-30

7

听老师讲课是一种享受，很舒服，从文字叙述就感觉很和蔼可亲，不像有的老师，虽然技术也很牛，但是话里话外透漏着自己多牛多牛的感觉

作者回复：感谢感谢😊



遇见阳光

2019-04-30

2

老师，我有一个疑问，如果说每个不同的业务都需要不同的线程池去处理，那这样线程池

不是越来越多，这种应该如何解决



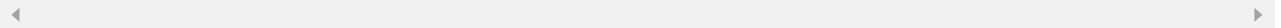
邱

2019-04-30

👍 1

王老师你好，我想问您一个问题:在实际的项目中使用线程池并行执行任务的时候，是不是和数据库的交互都不要放在线程池当中

作者回复: 这个还是要看实际场景，主要是考虑数据库事务，还有线程池是不是隔离的



Zach_

2019-05-12

👍

天呐，我一直以为执行check()的是 fixedPool中的的那唯一——一个线程!

展开 ▾



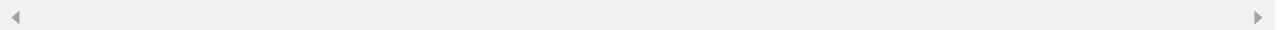
郑晨Cc

2019-05-05

👍

老师 第一个while(true)的例子 怎么在释放锁之前就 break退出循环了？难道break不该在释放锁之后吗？

作者回复: finally都会执行



捞鱼的搬砖...

2019-05-01

👍

老师能不能在上面提到的原文出错的代码边写上正确的做，并用注释说明

展开 ▾



ban

2019-05-01

👍

老师，你好。

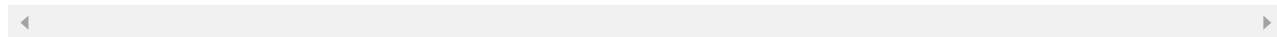
第五题的问题里面：通过源码你会发现 CyclicBarrier 是同步调用回调函数之后才唤醒等待的线程，如果我们在回调函数里直接调用 check() 方法，那就意味着在执行 check() 的时

候，是不能同时执行 `getPOrders()` 和 `getDOrders()` 的。

...

展开 ▾

作者回复: 执行 `check()` 的时候，是不能同时执行 `getPOrders()` 和 `getDOrders()`，因为执行这两个方法的线程一个在等待，一个正在忙着执行 `check()`。



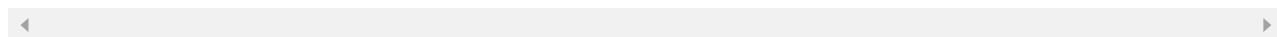
苏志辉

2019-04-30



第五个问题，我觉得应该先同步取完前两个节点再异步调用`check`逻辑，否则极端情况，取到的两个节点不是匹配的

作者回复: 如果`check`就一个线程执行，应该不会



张三

2019-04-30



打卡，虽然没有深入了解每个工具类，但确实了解更多了。

展开 ▾