

## 05 | 一不小心就死锁了，怎么办？

2019-03-09 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 12:31 大小 11.47M



在上一篇文章中，我们用 Account.class 作为互斥锁，来解决银行业务里面的转账问题，虽然这个方案不存在并发问题，但是所有账户的转账操作都是串行的，例如账户 A 转账户 B、账户 C 转账户 D 这两个转账操作现实世界里是可以并行的，但是在这个方案里却被串行化了，这样的话，性能太差。

试想互联网支付盛行的当下，8 亿网民每人每天一笔交易，每天就是 8 亿笔交易；每笔交易都对应着一次转账操作，8 亿笔交易就是 8 亿次转账操作，也就是说平均到每秒就是近 1 万次转账操作，若所有的转账操作都串行，性能完全不能接受。

那下面我们就尝试着把性能提升一下。

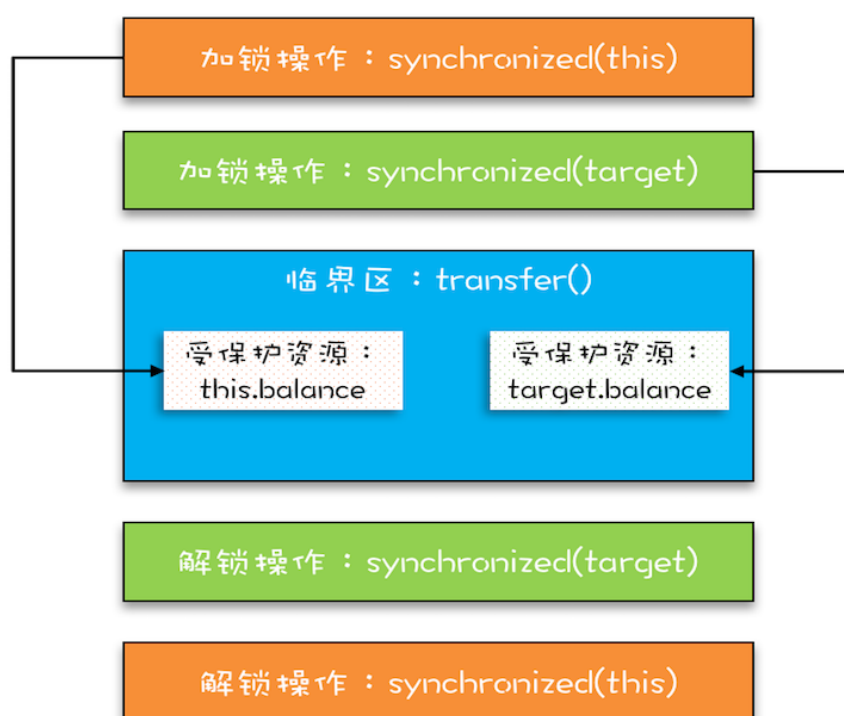
### 向现实世界要答案

现实世界里，账户转账操作是支持并发的，而且绝对是真正的并行，银行所有的窗口都可以做转账操作。只要我们能仿照现实世界做转账操作，串行的问题就解决了。

我们试想在古代，没有信息化，账户的存在形式真的就是一个账本，而且每个账户都有一个账本，这些账本都统一存放在文件架上。银行柜员在给我们做转账时，要去文件架上把转出账本和转入账本都拿到手，然后做转账。这个柜员在拿账本的时候可能遇到以下三种情况：

1. 文件架上恰好有转出账本和转入账本，那就同时拿走；
2. 如果文件架上只有转出账本和转入账本之一，那这个柜员就先把文件架上有的账本拿到手，同时等着其他柜员把另外一个账本送回来；
3. 转出账本和转入账本都没有，那这个柜员就等着两个账本都被送回来。

上面这个过程在编程的世界里怎么实现呢？其实用两把锁就实现了，转出账本一把，转入账本另一把。在 `transfer()` 方法内部，我们首先尝试锁定转出账户 `this`（先把转出账本拿到手），然后尝试锁定转入账户 `target`（再把转入账本拿到手），只有当两者都成功时，才执行转账操作。这个逻辑可以图形化为下图这个样子。



两个转账操作并行示意图

而至于详细的代码实现，如下所示。经过这样的优化后，账户 A 转账户 B 和账户 C 转账户 D 这两个转账操作就可以并行了。

```
1 class Account {
2     private int balance;
3     // 转账
4     void transfer(Account target, int amt){
5         // 锁定转出账户
6         synchronized(this) {
7             // 锁定转入账户
8             synchronized(target) {
9                 if (this.balance > amt) {
10                     this.balance -= amt;
11                     target.balance += amt;
12                 }
13             }
14         }
15     }
16 }
```

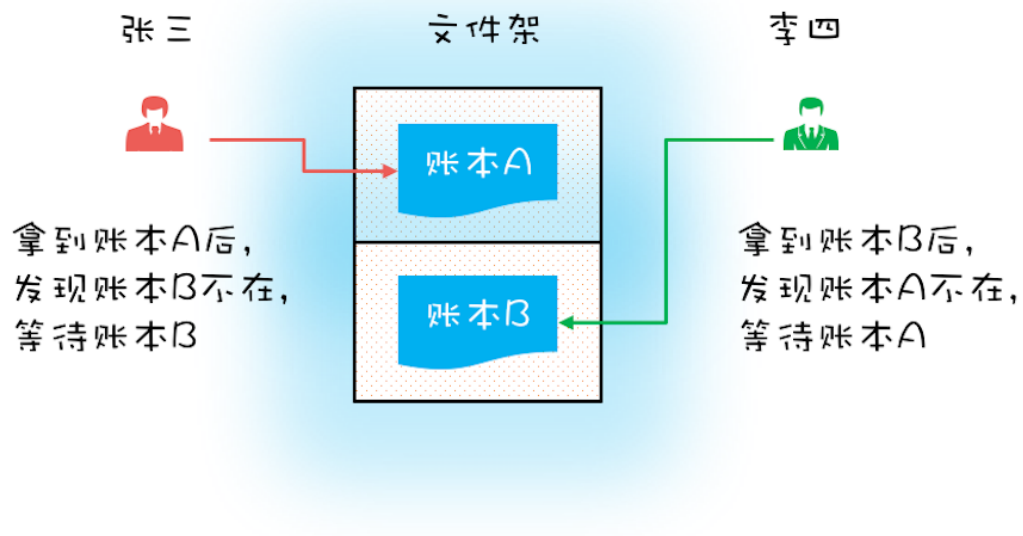
## 没有免费的午餐

上面的实现看上去很完美，并且也算是将锁用得出神入化了。相对于用 Account.class 作为互斥锁，锁定的范围太大，而我们锁定两个账户范围就小多了，这样的锁，上一章我们介绍过，叫**细粒度锁**。使用细粒度锁可以提高并行度，是性能优化的一个重要手段。

这个时候可能你已经开始警觉了，使用细粒度锁这么简单，有这样的好事，是不是也要付出点什么代价啊？编写并发程序就需要这样时时刻刻保持谨慎。

**的确，使用细粒度锁是有代价的，这个代价就是可能会导致死锁。**

在详细介绍死锁之前，我们先看看现实世界里的一种特殊场景。如果有客户找柜员张三做个转账业务：账户 A 转账户 B 100 元，此时另一个客户找柜员李四也做个转账业务：账户 B 转账户 A 100 元，于是张三和李四同时都去文件架上拿账本，这时候有可能凑巧张三拿到了账本 A，李四拿到了账本 B。张三拿到账本 A 后就等着账本 B（账本 B 已经被李四拿走），而李四拿到账本 B 后就等着账本 A（账本 A 已经被张三拿走），他们要等多久呢？他们会永远等待下去...因为张三不会把账本 A 送回去，李四也不会把账本 B 送回去。我们姑且称为死等吧。



### 转账业务中的“死等”

现实世界里的死等，就是编程领域的死锁了。**死锁**的一个比较专业的定义是：**一组互相竞争资源的线程因互相等待，导致“永久”阻塞的现象。**

上面转账的代码是怎么发生死锁的呢？我们假设线程 T1 执行账户 A 转账户 B 的操作，账户 A.transfer(账户 B)；同时线程 T2 执行账户 B 转账户 A 的操作，账户 B.transfer(账户 A)。当 T1 和 T2 同时执行完①处的代码时，T1 获得了账户 A 的锁（对于 T1，this 是账户 A），而 T2 获得了账户 B 的锁（对于 T2，this 是账户 B）。之后 T1 和 T2 在执行②处的代码时，T1 试图获取账户 B 的锁时，发现账户 B 已经被锁定（被 T2 锁定），所以 T1 开始等待；T2 则试图获取账户 A 的锁时，发现账户 A 已经被锁定（被 T1 锁定），所以 T2 也开始等待。于是 T1 和 T2 会无限地等待下去，也就是我们所说的死锁了。

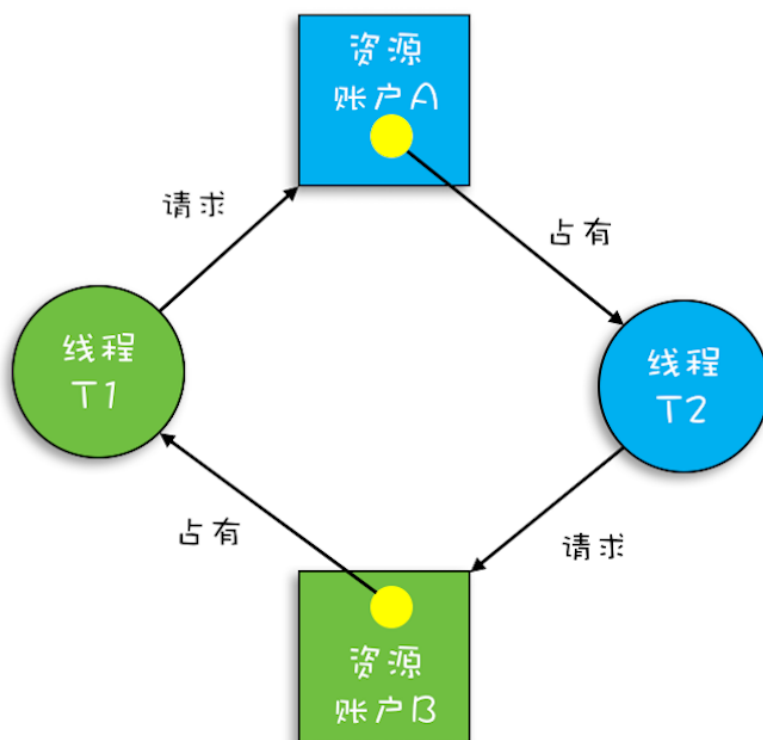
复制代码

```
1 class Account {
2     private int balance;
3     // 转账
4     void transfer(Account target, int amt){
5         // 锁定转出账户
6         synchronized(this){    ①
7             // 锁定转入账户
8             synchronized(target){ ②
9                 if (this.balance > amt) {
10                     this.balance -= amt;
11                     target.balance += amt;
```



```
12     }
13     }
14 }
15 }
16 }
```

关于这种现象，我们还可以借助资源分配图来可视化锁的占用情况（资源分配图是个有向图，它可以描述资源和线程的状态）。其中，资源用方形节点表示，线程用圆形节点表示；资源中的点指向线程的边表示线程已经获得该资源，线程指向资源的边则表示线程请求资源，但尚未得到。转账发生死锁时的资源分配图就如下图所示，一个“各据山头死等”的尴尬局面。



转账发生死锁时的资源分配图

## 如何预防死锁

并发程序一旦死锁，一般没有特别好的方法，很多时候我们只能重启应用。因此，解决死锁问题最好的办法还是规避死锁。

那如何避免死锁呢？要避免死锁就需要分析死锁发生的条件，有个叫 Coffman 的牛人早就总结过了，只有以下这四个条件都发生时才会出现死锁：

1. 互斥，共享资源 X 和 Y 只能被一个线程占用；
2. 占有且等待，线程 T1 已经取得共享资源 X，在等待共享资源 Y 的时候，不释放共享资源 X；
3. 不可抢占，其他线程不能强行抢占线程 T1 占有的资源；
4. 循环等待，线程 T1 等待线程 T2 占有的资源，线程 T2 等待线程 T1 占有的资源，就是循环等待。

反过来分析，**也就是说只要我们破坏其中一个，就可以成功避免死锁的发生。**

其中，互斥这个条件我们没有办法破坏，因为我们用锁为的就是互斥。不过其他三个条件都是有办法破坏掉的，到底如何做呢？

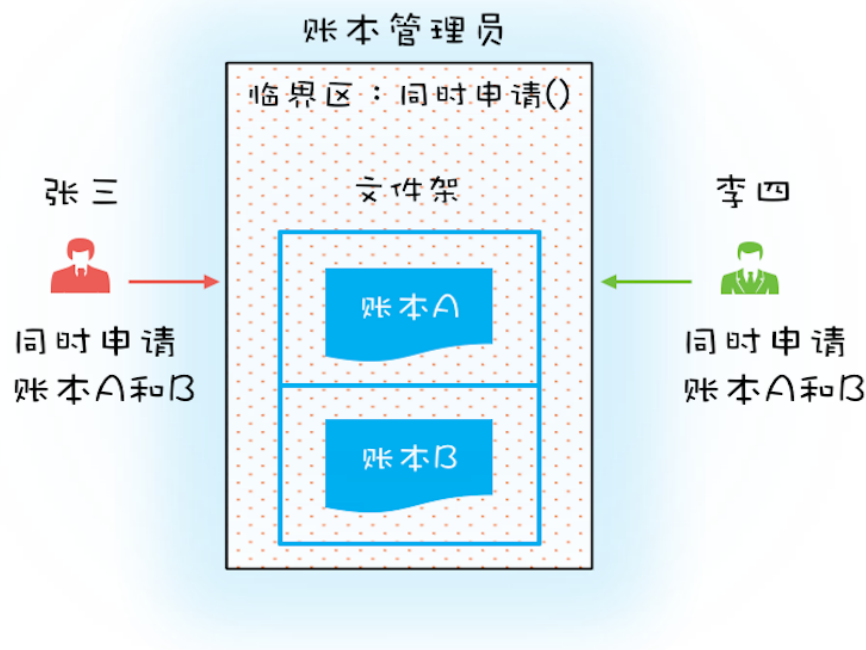
1. 对于“占用且等待”这个条件，我们可以一次性申请所有的资源，这样就不存在等待了。
2. 对于“不可抢占”这个条件，占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可抢占这个条件就破坏掉了。
3. 对于“循环等待”这个条件，可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后自然就不存在循环了。

我们已经从理论上解决了如何预防死锁，那具体如何体现在代码上呢？下面我们就来尝试用代码实践一下这些理论。

## 1. 破坏占用且等待条件


从理论上讲，要破坏这个条件，可以一次性申请所有资源。在现实世界里，就拿前面我们提到的转账操作来讲，它需要的资源有两个，一个是转出账户，另一个是转入账户，当这两个账户同时被申请时，我们该怎么解决这个问题呢？

可以增加一个账本管理员，然后只允许账本管理员从文件架上拿账本，也就是说柜员不能直接在文件架上拿账本，必须通过账本管理员才能拿到想要的账本。例如，张三同时申请账本 A 和 B，账本管理员如果发现文件架上只有账本 A，这个时候账本管理员是不会把账本 A 拿下来给张三的，只有账本 A 和 B 都在的时候才会给张三。这样就保证了“一次性申请所有资源”。



通过账本管理员拿账本

对应到编程领域，“同时申请”这个操作是一个临界区，我们也需要一个角色（Java 里面的类）来管理这个临界区，我们就把这个角色定为 Allocator。它有两个重要功能，分别是：同时申请资源 apply() 和同时释放资源 free()。账户 Account 类里面持有一个 Allocator 的单例（必须是单例，只能由一个人来分配资源）。当账户 Account 在执行转账操作的时候，首先向 Allocator 同时申请转出账户和转入账户这两个资源，成功后再锁定这两个资源；当转账操作执行完，释放锁之后，我们需通知 Allocator 同时释放转出账户和转入账户这两个资源。具体的代码实现如下。

 复制代码

```
1 class Allocator {
2     private List<Object> als =
3         new ArrayList<>();
4     // 一次性申请所有资源
5     synchronized boolean apply(
6         Object from, Object to){
7         if(als.contains(from) ||
8             als.contains(to)){
9             return false;
10        } else {
11            als.add(from);
12            als.add(to);
13        }
14        return true;
15    }
```

```
16 // 归还资源
17 synchronized void free(
18     Object from, Object to){
19     als.remove(from);
20     als.remove(to);
21 }
22 }
23
24 class Account {
25     // actr 应该为单例
26     private Allocator actr;
27     private int balance;
28     // 转账
29     void transfer(Account target, int amt){
30         // 一次性申请转出账户和转入账户，直到成功
31         while(!actr.apply(this, target))
32             ;
33         try{
34             // 锁定转出账户
35             synchronized(this){
36                 // 锁定转入账户
37                 synchronized(target){
38                     if (this.balance > amt){
39                         this.balance -= amt;
40                         target.balance += amt;
41                     }
42                 }
43             }
44         } finally {
45             actr.free(this, target)
46         }
47     }
48 }
```

## 2. 破坏不可抢占条件


破坏不可抢占条件看上去很简单，核心是要能够主动释放它占有的资源，这一点 `synchronized` 是做不到的。原因是 `synchronized` 申请资源的时候，如果申请不到，线程直接进入阻塞状态了，而线程进入阻塞状态，啥都干不了，也释放不了线程已经占有的资源。

你可能会质疑，“Java 作为排行榜第一的语言，这都解决不了？”你的怀疑很有道理，Java 在语言层次确实没有解决这个问题，不过在 SDK 层面还是解决了的，`java.util.concurrent` 这个包下面提供的 `Lock` 是可以轻松解决这个问题的。关于这个话题，咱们后面会详细讲。



### 3. 破坏循环等待条件

破坏这个条件，需要对资源进行排序，然后按序申请资源。这个实现非常简单，我们假设每个账户都有不同的属性 id，这个 id 可以作为排序字段，申请的时候，我们可以按照从小到大的顺序来申请。比如下面代码中，①~⑥处的代码对转出账户（this）和转入账户（target）排序，然后按照序号从小到大的顺序锁定账户。这样就不存在“循环”等待了。

 复制代码

```
1 class Account {
2     private int id;
3     private int balance;
4     // 转账
5     void transfer(Account target, int amt){
6         Account left = this           ①
7         Account right = target;       ②
8         if (this.id > target.id) {    ③
9             left = target;           ④
10            right = this;             ⑤
11        }                             ⑥
12        // 锁定序号小的账户
13        synchronized(left){
14            // 锁定序号大的账户
15            synchronized(right){
16                if (this.balance > amt){
17                    this.balance -= amt;
18                    target.balance += amt;
19                }
20            }
21        }
22    }
23 }
```

## 总结

当我们在编程世界里遇到问题时，应不局限于当下，可以换个思路，向现实世界要答案，**利用现实世界的模型来构思解决方案**，这样往往能够让我们的方案更容易理解，也更能够看清问题的本质。

但是现实世界的模型有些细节往往会被我们忽视。因为在现实世界里，人太智能了，以致有些细节实在是显得太不重要了。在转账的模型中，我们为什么会忽视死锁问题呢？原因主要是在现实世界，我们会交流，并且会很智能地交流。而编程世界里，两个线程是不会智能地

交流的。所以在利用现实模型建模的时候，我们还要仔细对比现实世界和编程世界里的各角色之间的差异。

我们今天这一篇文章主要讲了**用细粒度锁来锁定多个资源时，要注意死锁的问题**。这个就需要你能把它强化为一个思维定势，遇到这种场景，马上想到可能存在死锁问题。当你知道风险之后，才有机会谈如何预防和避免，因此，**识别出风险很重要**。

预防死锁主要是破坏三个条件中的一个，有了这个思路后，实现就简单了。但仍需注意的是，有时候预防死锁成本也是很高的。例如上面转账那个例子，我们破坏占用且等待条件的成本就比破坏循环等待条件的成本高，破坏占用且等待条件，我们也是锁了所有的账户，而且还是用了死循环 `while(!actr.apply(this, target));` 方法，不过好在 `apply()` 这个方法基本不耗时。在转账这个例子中，破坏循环等待条件就是成本最低的一个方案。

所以我们在选择具体方案的时候，还需要**评估一下操作成本，从中选择一个成本最低的方案**。

## 课后思考

我们上面提到：破坏占用且等待条件，我们也是锁了所有的账户，而且还是用了死循环 `while(!actr.apply(this, target));` 这个方法，那它比 `synchronized(Account.class)` 有没有性能优势呢？

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 04 | 互斥锁（下）：如何用一把锁保护多个资源？

下一篇 06 | 用“等待-通知”机制优化循环等待

## 精选留言 (102)

写留言



捞鱼的搬砖...

2019-03-09

25

synchronized(Account.class) 锁了Account类相关的所有操作。相当于文中说的包场了，只要与Account有关联，通通需要等待当前线程操作完成。while死循环的方式只锁定了当前操作的两个相关的对象。两种影响到的范围不同。

展开

作者回复：还真是这样啊！



Tony Du

22

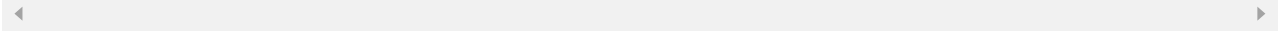
2019-03-09

while循环是不是应该有个timeout，避免一直阻塞下去？

展开 ∨

作者回复: 你考虑的很周到! 📖

加超时在实际项目中非常重要!



**DemonLee**

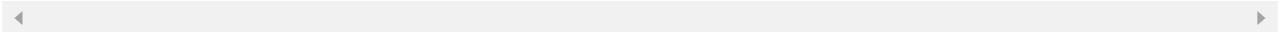
2019-03-09

👍 16

while(actr.apply(this, target)); --> while(!actr.apply(this, target));

我感觉应该是这样，老师，我理解错了？

作者回复: 你发现了个大bug!感谢感谢!!! 我这就修改一下啊



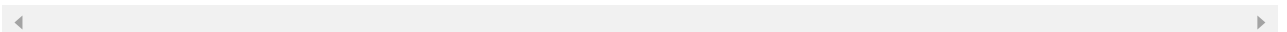
**邈邈的流浪...**

2019-03-09

👍 11

思考题的话希望老师能够过后给出一个比较标准的答案，毕竟大家的留言中说法各不相同  
很难去判断答案的对错

作者回复: 这一部分的最后一章，要不就给答案吧



**几字凉了秋...**

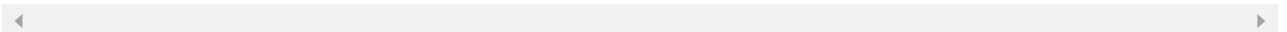
2019-03-10

👍 8

老师，请问一下，在实际的开发中，account对象应该是从数据库中查询出来的吧，假如A  
转B，C转B一起执行，那B的account对象如何保证是同一个对象，不太理解。。。

展开 ∨

作者回复: 实际开发中都是用数据库事务+乐观锁的方式解决的。这个就是个例子，为了说明死锁  
是怎么回事，以及死锁问题怎么解决。





别皱眉

2019-03-14

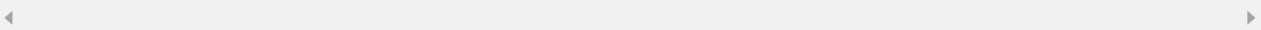
👍 7

@阿官 我来回答下你的问题

以下是阿官的问题

-----  
老师，在破坏占用且等待的案例中，为何申请完两个账户的资源后还需要再分别锁定this...  
展开 ▾

作者回复: 你说到我心里了 😊😊😊



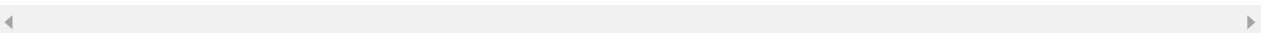
张立华

2019-03-12

👍 7

之前遇到死锁，我就是用资源id的从小到大的顺序去申请锁解决的  
展开 ▾

作者回复: 这个方案最简单



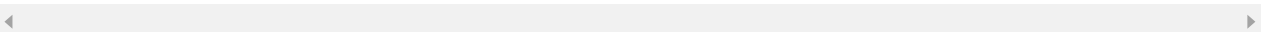
Howie

2019-03-09

👍 7

while 循环就是一个自旋锁机制吧，自旋锁的话要关注它的循环时间，不能一直循环下去，不然会浪费 cpu 资源。

作者回复: 自旋锁在JVM里是一种特殊的锁机制，自诩不会阻塞线程的。咱们这个其实还是会阻塞线程的。不过原理都一样，你这样理解也没问题。



轻歌赋

2019-03-09

👍 6

存在性能差距，虽然申请的时候加锁导致单线程访问，但是hash判断和赋值时间复杂度低，而在锁中执行业务代码时间长很多。  
申请的时候单线程，但是执行的时候就可以多线程了，这里性能提升比较明显

想问问老师，如何判断多线程的阻塞导致的问题呢？有什么工具吗



展开 ∨

作者回复: 可以用top命令查看Java线程的cpu利用率, 用jstack来dump线程。开发环境可以用java visualvm查看线程执行情况

◀ ▶



winter

2019-03-09

👍 4

我的想法是, 如果Account对象中只有转账业务的话, while(ctr.apply(this, target))和对对象锁synchronized(Account.class)的性能优势几乎看不出来, synchronized(Account.class)的性能甚至更差; 但是如果Account对象中如果还有其它业务, 比如查看余额等功能也加了synchronized(Account.class)修饰, 那么把单独的转账业务剥离出来, 性能的提升可能就比较明显了。

展开 ∨

作者回复: 是的, 有时候性能更差, 毕竟要synchronized三次。但是有些场景会更好, 例如转账操作很慢, 而apply很快, 这个时候允许a->b,c->d并行就有优势了。

◀ ▶



aguan(^·...

2019-03-14

👍 3

老师, 在破坏占用且等待的案例中, 为何申请完两个账户的资源后还需要再分别锁定this和target账户呢?

作者回复: 为了保险而已, 单纯这个例子是不需要的, 如果还有取款操作就需要了

◀ ▶



λ

2019-03-11

👍 3

单例导致操作也是串行的吧

展开 ∨

作者回复: 是串行, 但是允许A转B, C转D

◀ ▶



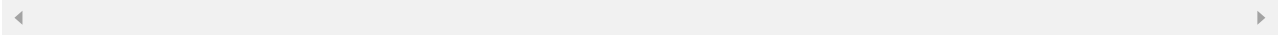
**gogo**

2019-03-10

👍 3

看了老师的讲解学到了很多，联想了下实际转账业务，应该是数据库来实现的，假如有账户表account，利用mysql的悲观锁select ...for update对a, b两条数据锁定，这时也有可能发生死锁，按照您讲到的第三种破坏循环等待的方式，按照id的大小顺序依次锁定。我这样理解的对吗？

作者回复: 是的，就是id的次序。



**李可威**

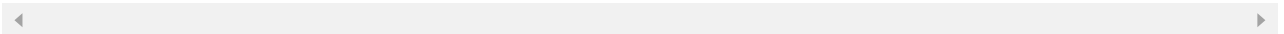
2019-03-17

👍 2

老师为什么按序申请资源就可以破坏循环等待条件呢？这点没有看懂求解答  
展开 ▾

作者回复: 循环等待，一定是A->B->C->...->N->A形成环状。

如果按需申请，是不允许N->A出现的，只能N->P。没有环状，也就不会死锁了。



**陈华**

2019-03-14

👍 2

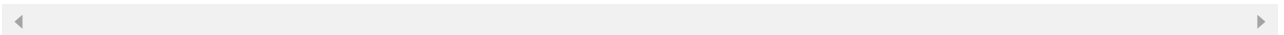
对于第三点，按资源顺序来锁就能打破循环等待有疑问。

例如：账户 1 向 账户 3 转账

同时 账户 3 向 账户 5 转账

即使按资源顺序来锁，也是起不了啥作用吧！？，

作者回复: 能起作用，这俩操作不会死锁



**GP**

2019-03-13

👍 2

问下，上节最后说到，不能用可变对象做锁，这里为何又synchronized (left)？

作者回复: 保护的是对象里面的成员，这俩对象变也只能是里面成员变，相对于里面的成员来说，这俩对象是永远不会变的。你可以这样理解。不是绝对不能用于可变对象，只是一条最佳实践。





QQ怪

2019-03-09

👍 2

死循环只是锁的是两个对象，而Account锁的是所有，串行化了

展开 ▾

作者回复: 死循环里其实也还锁了一个全局对象



新世界

2019-03-09

👍 2

没有性能优势，alloctor的操作也是获取alloctor对象的锁，和获取account的对象锁本质没有区别

作者回复: 锁的范围变了，所以场景不同性能也会有差异，并发量小的话，性能还会变差



西西弗与卡...

2019-03-09

👍 2

性能优势还是有的，毕竟后者是对这个类的所有访问都有锁的动作

展开 ▾

作者回复: 是的，锁的范围是个大问题。允许A->B 和 C->D可以并行还是很重要的



嘟嘟科技研...

2019-03-15

👍 1

```
class Allocator {  
    private List<Object> als =  
        new ArrayList<>();  
    // 一次性申请所有资源  
    synchronized boolean apply(...
```

展开 ▾

作者回复: 释放2个资源是在转账完成之后, 而不是while执行完。线程2获得资源一定是线程1 执行完转账了

