

04 | 互斥锁（下）：如何用一把锁保护多个资源？

2019-03-07 王宝令



讲述：王宝令

时长 09:30 大小 8.71M



在上一篇文章中，我们提到**受保护资源和锁之间合理的关联关系应该是 N:1 的关系**，也就是说可以用一把锁来保护多个资源，但是不能用多把锁来保护一个资源，并且结合文中示例，我们也重点强调了“不能用多把锁来保护一个资源”这个问题。而至于如何保护多个资源，我们今天就来聊聊。

当我们要保护多个资源时，首先要区分这些资源是否存在关联关系。


保护没有关联关系的多个资源

在现实世界里，球场的座位和电影院的座位就是没有关联关系的，这种场景非常容易解决，那就是球赛有球赛的门票，电影院有电影院的门票，各自管理各自的。

同样这对应到编程领域，也很容易解决。例如，银行业务中有针对账户余额（余额是一种资源）的取款操作，也有针对账户密码（密码也是一种资源）的更改操作，**我们可以为账**

户余额和账户密码分配不同的锁来解决并发问题，这个还是很简单的。

相关的示例代码如下，账户类 Account 有两个成员变量，分别是账户余额 balance 和账户密码 password。取款 withdraw() 和查看余额 getBalance() 操作会访问账户余额 balance，我们创建一个 final 对象 balLock 作为锁（类比球赛门票）；而更改密码 updatePassword() 和查看密码 getPassword() 操作会修改账户密码 password，我们创建一个 final 对象 pwLock 作为锁（类比电影票）。不同的资源用不同的锁保护，各自管各自的，很简单。

 复制代码

```
1 class Account {
2     // 锁：保护账户余额
3     private final Object balLock
4         = new Object();
5     // 账户余额
6     private Integer balance;
7     // 锁：保护账户密码
8     private final Object pwLock
9         = new Object();
10    // 账户密码
11    private String password;
12
13    // 取款
14    void withdraw(Integer amt) {
15        synchronized(balLock) {
16            if (this.balance > amt){
17                this.balance -= amt;
18            }
19        }
20    }
21    // 查看余额
22    Integer getBalance() {
23        synchronized(balLock) {
24            return balance;
25        }
26    }
27
28    // 更改密码
29    void updatePassword(String pw){
30        synchronized(pwLock) {
31            this.password = pw;
32        }
33    }
34    // 查看密码
35    String getPassword() {
36        synchronized(pwLock) {
37            return password;
```


```
38     }
39 }
40 }
```

当然，**我们也可以用一把互斥锁来保护多个资源**，**例如我们可以用 this 这一把锁来管理账户类里所有的资源：账户余额和用户密码。**具体实现很简单，示例程序中所有的方法都增加同步关键字 `synchronized` 就可以了，这里我就不一一展示了。

但是用一把锁有个问题，就是性能太差，会导致取款、查看余额、修改密码、查看密码这四个操作都是串行的。而我们用两把锁，取款和修改密码是可以并行的。**用不同的锁对受保护资源进行精细化管理，能够提升性能。**这种锁还有个名字，叫**细粒度锁**。


保护有关联关系的多个资源

如果多个资源是有关联关系的，那这个问题就有点复杂了。例如银行业务里面的转账操作，账户 A 减少 100 元，账户 B 增加 100 元。这两个账户就是有关联关系的。那对于像转账这种有关联关系的操作，我们应该怎么去解决呢？先把这个问题代码化。我们声明了一个账户类：`Account`，该类有一个成员变量余额：`balance`，还有一个用于转账的方法：`transfer()`，然后怎么保证转账操作 `transfer()` 没有并发问题呢？

 复制代码

```
1 class Account {
2     private int balance;
3     // 转账
4     void transfer(
5         Account target, int amt){
6         if (this.balance > amt) {
7             this.balance -= amt;
8             target.balance += amt;
9         }
10    }
11 }
```

相信你的直觉会告诉你这样的解决方案：用户 `synchronized` 关键字修饰一下 `transfer()` 方法就可以了，于是你很快就完成了相关的代码，如下所示。

 复制代码

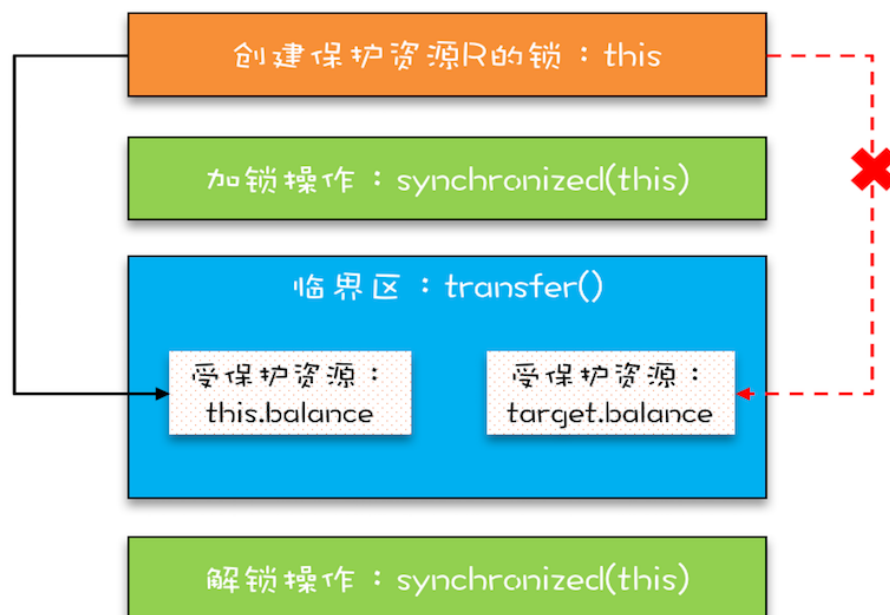
```

1 class Account {
2     private int balance;
3     // 转账
4     synchronized void transfer(
5         Account target, int amt){
6         if (this.balance > amt) {
7             this.balance -= amt;
8             target.balance += amt;
9         }
10    }
11 }

```

在这段代码中，临界区内有两个资源，分别是转出账户的余额 `this.balance` 和转入账户的余额 `target.balance`，并且用的是一把锁 `this`，符合我们前面提到的，多个资源可以用一把锁来保护，这看上去完全正确呀。真的是这样吗？可惜，这个方案仅仅是看似正确，为什么呢？

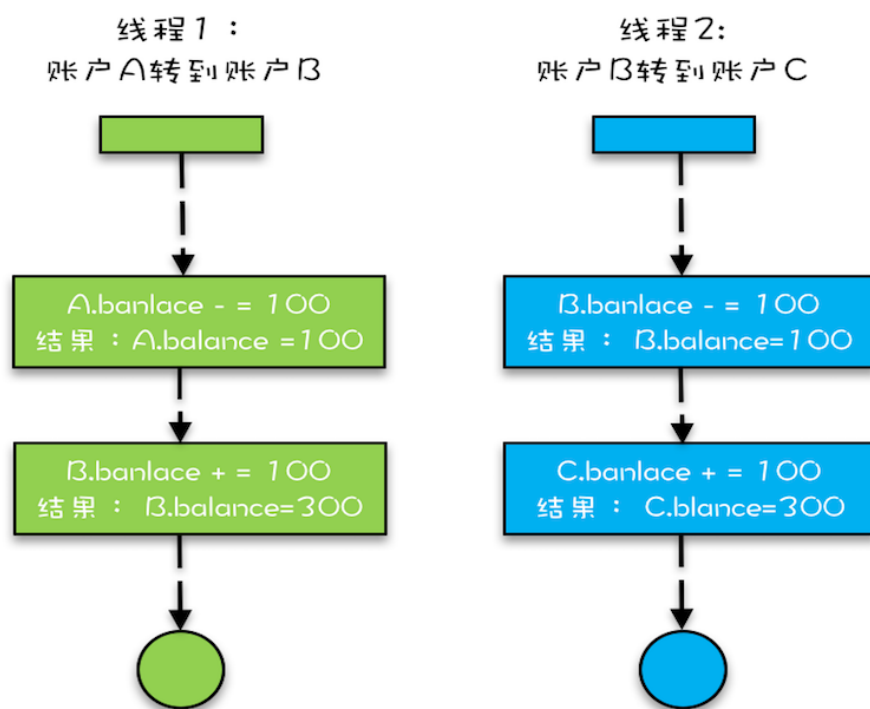
问题就出在 `this` 这把锁上，`this` 这把锁可以保护自己的余额 `this.balance`，却保护不了别人的余额 `target.balance`，就像你不能用自家的锁来保护别人家的资产，也不能用自己的票来保护别人的座位一样。



用锁 `this` 保护 `this.balance` 和 `target.balance` 的示意图

下面我们具体分析一下，假设有 A、B、C 三个账户，余额都是 200 元，我们用两个线程分别执行两个转账操作：账户 A 转给账户 B 100 元，账户 B 转给账户 C 100 元，最后我们期望的结果应该是账户 A 的余额是 100 元，账户 B 的余额是 200 元，账户 C 的余额是 300 元。

我们假设线程 1 执行账户 A 转账到账户 B 的操作，线程 2 执行账户 B 转账到账户 C 的操作。这两个线程分别在两颗 CPU 上同时执行，那它们是互斥的吗？我们期望是，但实际上并不是。因为线程 1 锁定的是账户 A 的实例（A.this），而线程 2 锁定的是账户 B 的实例（B.this），所以这两个线程可以同时进入临界区 transfer()。同时进入临界区的结果是什么呢？线程 1 和线程 2 都会读到账户 B 的余额为 200，导致最终账户 B 的余额可能是 300（线程 1 后于线程 2 写 B.balance，线程 2 写的 B.balance 值被线程 1 覆盖），可能是 100（线程 1 先于线程 2 写 B.balance，线程 1 写的 B.balance 值被线程 2 覆盖），就是不可能是 200。




并发转账示意图

使用锁的正确姿势

在上一篇文章中，我们提到用同一把锁来保护多个资源，也就是现实世界的“包场”，那在编程领域应该怎么“包场”呢？很简单，只要我们的锁能覆盖所有受保护资源就可以了。在上面的例子中，this 是对象级别的锁，所以 A 对象和 B 对象都有自己的锁，如何让 A 对象和 B 对象共享一把锁呢？


稍微开动脑筋，你会发现其实方案还挺多的，比如可以让所有对象都持有一个唯一性的对象，这个对象在创建 Account 时传入。方案有了，完成代码就简单了。示例代码如下，我们把 Account 默认构造函数变为 private，同时增加一个带 Object lock 参数的构造函数，创建 Account 对象时，传入相同的 lock，这样所有的 Account 对象都会共享这个 lock 了。

 复制代码

```
1 class Account {
2     private Object lock;
3     private int balance;
4     private Account();
5     // 创建 Account 时传入同一个 lock 对象
6     public Account(Object lock) {
7         this.lock = lock;
8     }
9     // 转账
10    void transfer(Account target, int amt){
11        // 此处检查所有对象共享的锁
12        synchronized(lock) {
13            if (this.balance > amt) {
14                this.balance -= amt;
15                target.balance += amt;
16            }
17        }
18    }
19 }
```

这个办法确实能解决问题，但是有点小瑕疵，它要求在创建 Account 对象的时候必须传入同一个对象，如果创建 Account 对象时，传入的 lock 不是同一个对象，那可就惨了，会出现锁自家门来保护他家资产的荒唐事。在真实的项目场景中，创建 Account 对象的代码很可能分散在多个工程中，传入共享的 lock 真的很难。

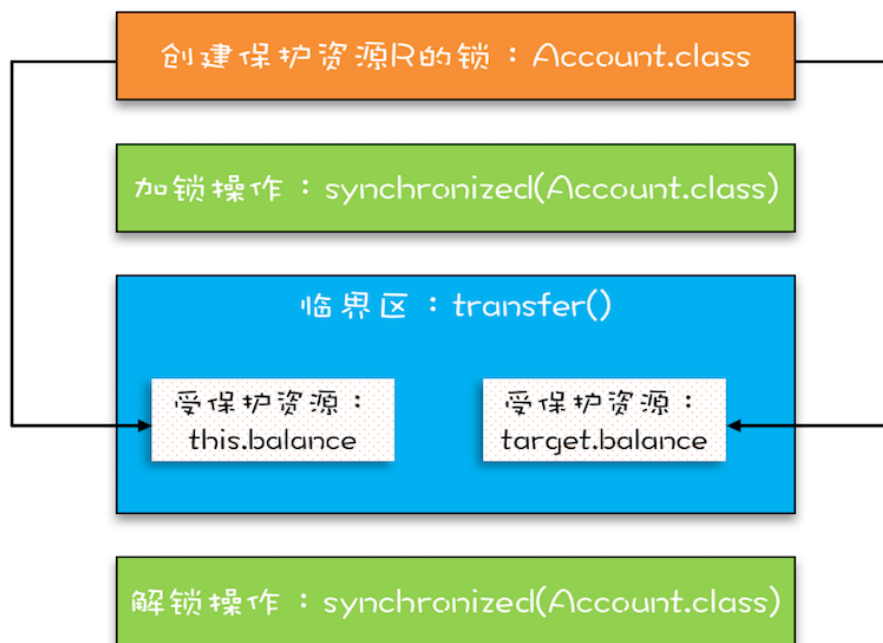
所以，上面的方案缺乏实践的可行性，我们需要更好的方案。还真有，就是用 **Account.class 作为共享的锁**。Account.class 是所有 Account 对象共享的，而且这个对象是 Java 虚拟机在加载 Account 类的时候创建的，所以我们不用担心它的唯一性。使用 Account.class 作为共享的锁，我们就无需在创建 Account 对象时传入了，代码更简单。

 复制代码

```
1 class Account {
2     private int balance;
```

```
3 // 转账
4 void transfer(Account target, int amt){
5     synchronized(Account.class) {
6         if (this.balance > amt) {
7             this.balance -= amt;
8             target.balance += amt;
9         }
10    }
11 }
12 }
```

下面这幅图很直观地展示了我们是如何使用共享的锁 `Account.class` 来保护不同对象的临界区的。



总结

相信你看完这篇文章后，对如何保护多个资源已经很有心得了，关键是要分析多个资源之间的关系。如果资源之间没有关系，很好处理，每个资源一把锁就可以了。如果资源之间有关联关系，就要选择一个粒度更大的锁，这个锁应该能够覆盖所有相关的资源。除此之外，还要梳理出有哪些访问路径，所有的访问路径都要设置合适的锁，这个过程可以类比一下门票管理。

我们再引申一下上面提到的关联关系，关联关系如果用更具体、更专业的语言来描述的话，其实是一种“原子性”特征，在前面的文章中，我们提到的原子性，主要是面向 CPU 指令的，转账操作的原子性则是属于是面向高级语言的，不过它们本质上是一样的。

“原子性”的本质是什么？其实不是不可分割，不可分割只是外在表现，其本质是多个资源间有一致性的要求，**操作的中间状态对外不可见**。例如，在 32 位的机器上写 long 型变量有中间状态（只写了 64 位中的 32 位），在银行转账的操作中也有中间状态（账户 A 减少了 100，账户 B 还没来得及发生变化）。所以**解决原子性问题，是要保证中间状态对外不可见**。

课后思考

在第一个示例程序里，我们用了两把不同的锁来分别保护账户余额、账户密码，创建锁的时候，我们用的是：`private final Object xxxLock = new Object();`，如果账户余额用 `this.balance` 作为互斥锁，账户密码用 `this.password` 作为互斥锁，**你觉得是否可以呢？**

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」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (21)

写留言



不靠谱的琴...

2019-03-07

3

思考题：使用修改的对象作为互斥锁对象可行，上面有提到锁的粒度问题，这个属于最小粒度。



Geek_a53d6...

2019-03-07

1

用this.balance 和this.password 都不行。在同一个账户多线程访问时候，A线程取款进行this.balance-=amt;时候此时this.balance对应的值已经发生变换，线程B再次取款时拿到的balance对应的值并不是A线程中的，也就是说不能把可变的对象当成一把锁。this.password 虽然说是String修饰但也会改变，所以也不行。老师所讲例子中的两个Object无论多次访问过程中都未发生变化？...

展开



Solitary_...

2019-03-07

1

不行，每个实例的this不同，锁住锁住的不是同一个对象

展开



wang

2019-03-07

1

不可以。因为balance为integer对象，当值被修改相当于换锁，还有integer有缓存-128到127，相当于同一个对象。



往事随风，...

2019-03-07

1

虽然对象值会更改，但是对象不会变，还是同一个对象，是还可以作为互斥对象的

展开 ▾



yang

2019-03-07



banlance可以，因为用的是堆中对象的地址，当其值改变时，引用地址不变；password不可以，因为string是存在堆中的常量池中，如果值变了，引用关系也变了，所以就是不同的锁对象了。

个人理解，敬请指正



新世界

2019-03-07



不可行，java的integer和string代码有采用享元模式，比如Integer.valueOf创建两个都是1的对象，其实本质是一个对象吧，如果在这个对象上加锁，粒度太粗了



rayjun

2019-03-07



可以的，因为 this.balance 和 this.password 相互之间是没有关系的，可以分别用做保护自己的锁



冰激凌的眼...

2019-03-07



string一旦修改，就成了其他的对象，可能无效吧。

小整数是不是有个缓存问题，导致要么被共享锁粒度过大，要么被换成其他对象导致失效。



密码123456

2019-03-07



不可行，假设余额是100。只是锁了100。如果值更改成200。其他线程发现锁的是100，而当前值并不是100，也可以进去临界区



无庸

2019-03-07



不可以的。不同对象实例的this 是不同的对象。

展开 ∨



沙漠里的骆驼...

2019-03-07



个人感觉不可行；

因为Integer对象是不可变的，每次更新时是new了一个新对象。所以在A线程对balance做变更的时候，导致this.balance对象发生变化，而B线程此时是可以进入临界区的(锁已经是新锁)，而进入临界区的时候对象甚至可能未初始化。



Geek_69358...

2019-03-07



考虑密码和余额这两个资源没有关联关系，所以这两个资源使用各自的锁来控制自身的操作，是完全没有问题的。



Zach_

2019-03-07



在第一个示例程序里，我们用了两把不同的锁来分别保护账户余额、账户密码，创建锁的时候，我们用的是：`private final Object xxxLock = new Object();`，如果账户余额用this.balance 作为互斥锁，账户密码用 this.password 作为互斥锁，你觉得是否可以呢？

用this.balance this.password 相当于私有锁 可以作为业务与其他Account没有关联关...

展开 ∨



峰

2019-03-07



思考题，我的答案是不行，因为对象可变，所以导致加锁对象不一样。

然后感觉加锁的所有用户用同一个锁的粒度太大了，但如果每次转账操作，是不是可以同时加两个用户的锁，如果有先后顺序又可能有死锁问题。



Kid☺

2019-03-07



定义一个静态对象也可以作为互斥锁管理多个关联的资源

思考题:可以的 每个账户只访问自身的资源



苦行僧

2019-03-07



对象锁 类锁

展开 ∨



王昊

2019-03-07



账户转账这个是一个特别好的例子，但感觉老师讲的不透彻。锁Account.class性能太差了，细粒度锁又可能死锁，这块希望能好好讲讲



孙悟空

2019-03-07



思考题，不行。

Java里的对象变量仅仅是一个引用，每次写入这两个成员变量后，他们指向的对象都会发生变化，所以对应的锁不止一个。多线程写入时，会出现多个锁保护同一个资源的情况。就会出现不止一个线程可以进去临界区的问题。



Geek_c4250...

2019-03-07



不可以吧，锁在运行期间不能改变锁值吧，值改变了怎么锁住？

展开 ∨