04 | 互斥锁（下）：如何用一把锁保护多个资源？

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。

class Account {

// 锁：保护账户余额

private final Object balLock

= new Object();

// 账户余额

private Integer balance;

// 锁：保护账户密码

private final Object pwLock

= new Object();

// 账户密码

private String password;

// 取款

void withdraw(Integer amt) {

synchronized(balLock) {

if (this.balance > amt){

this.balance -= amt;

}

}

}

// 查看余额

Integer getBalance() {

synchronized(balLock) {

return balance;

}

}

// 更改密码

void updatePassword(String pw){

synchronized(pwLock) {

this.password = pw;

}

}

// 查看密码

String getPassword() {

synchronized(pwLock) {

return password;

}

}

}

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？

class Account {

private int balance;

// 转账

void transfer(

Account target, int amt){

if (this.balance > amt) {

this.balance -= amt;

target.balance += amt;

}

}

}

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。

class Account {

private int balance;

// 转账

synchronized void transfer(

Account target, int amt){

if (this.balance > amt) {

this.balance -= amt;

target.balance += amt;

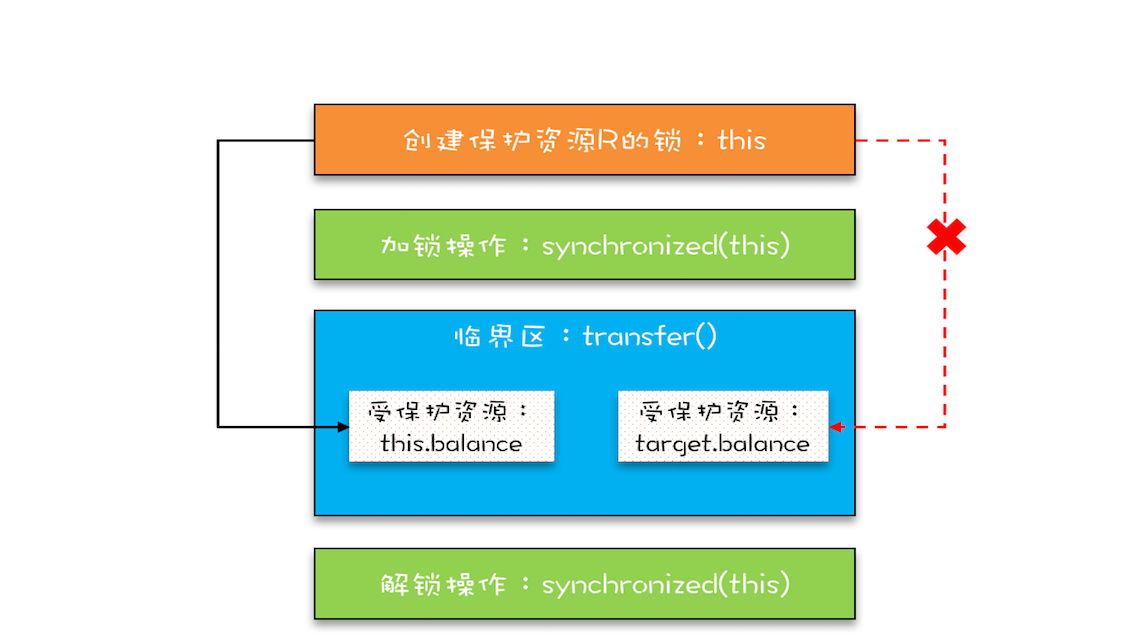
}

}

}

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？

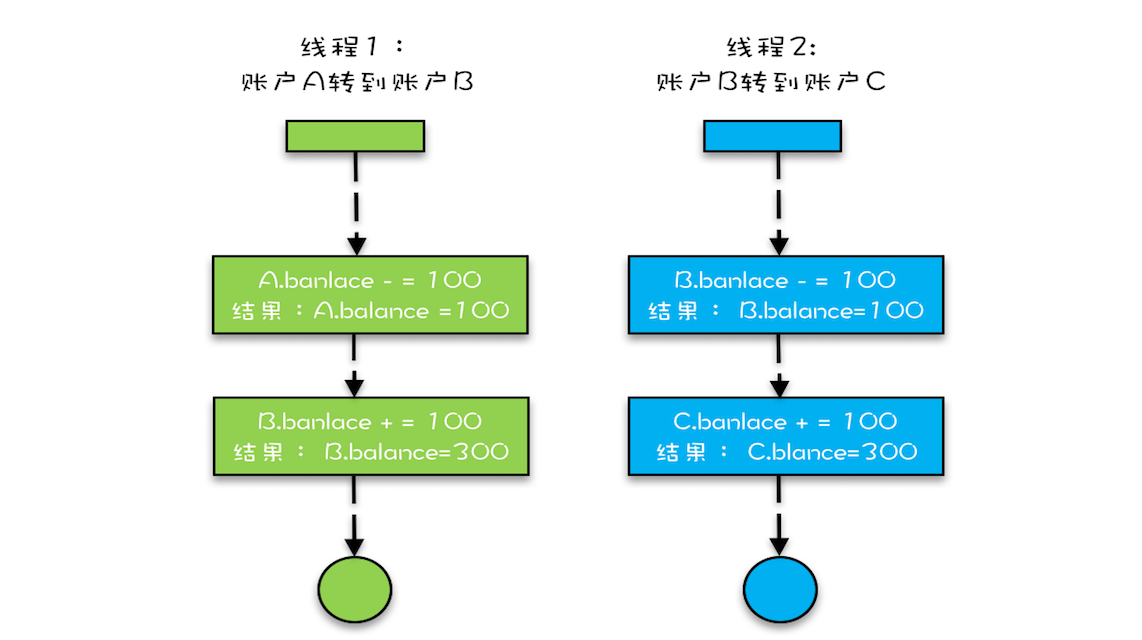
问题就出在 this 这把锁上，this 这把锁可以保护自己的余额 this.balance，却保护不了别人的余额 target.balance，就像你不能用自家的锁来保护别人家的资产，也不能用自己的票来保护别人的座位一样。



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 转账户 B 的操作，线程 2 执行账户 B 转账户 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。

class Account {

private Object lock；

private int balance;

private Account();

// 创建Account时传入同一个lock对象

public Account(Object lock) {

this.lock = lock;

}

// 转账

void transfer(Account target, int amt){

// 此处检查所有对象共享的锁

synchronized(lock) {

if (this.balance > amt) {

this.balance -= amt;

target.balance += amt;

}

}

}

}

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。

class Account {

private int balance;

// 转账

void transfer(Account target, int amt){

synchronized(Account.class) {

if (this.balance > amt) {

this.balance -= amt;

target.balance += amt;

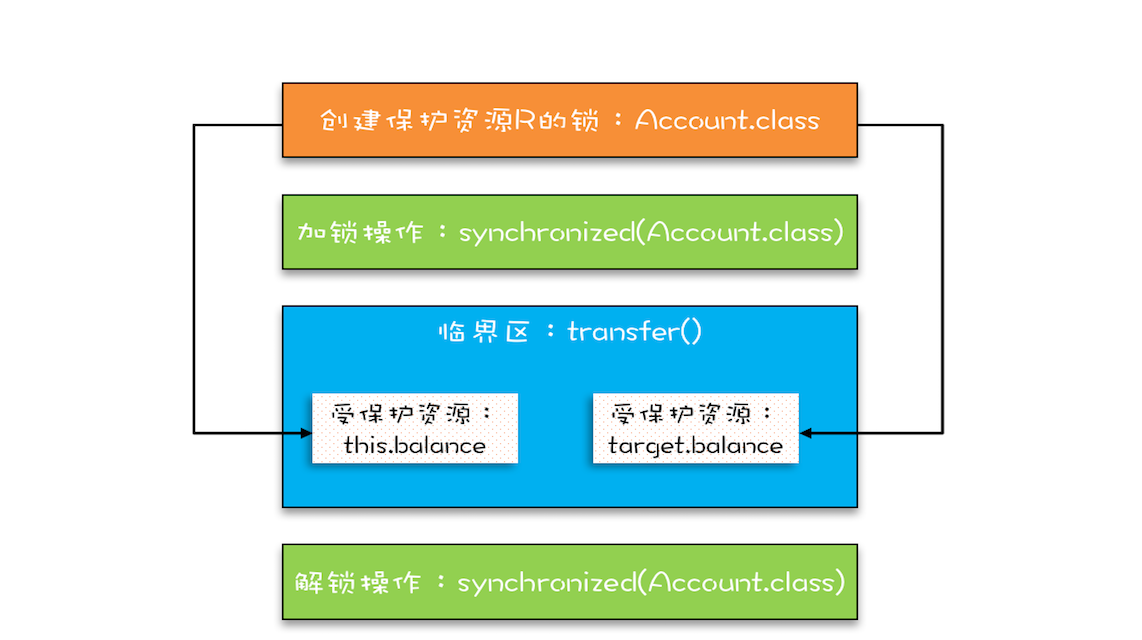
}

}

}

}

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？

我觉得不能用balance和password做为锁对象。这两个对象balance是Integer，password是String都是不可变变对象，一但对他们进行赋值就会变成新的对象，加的锁就失效了