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？

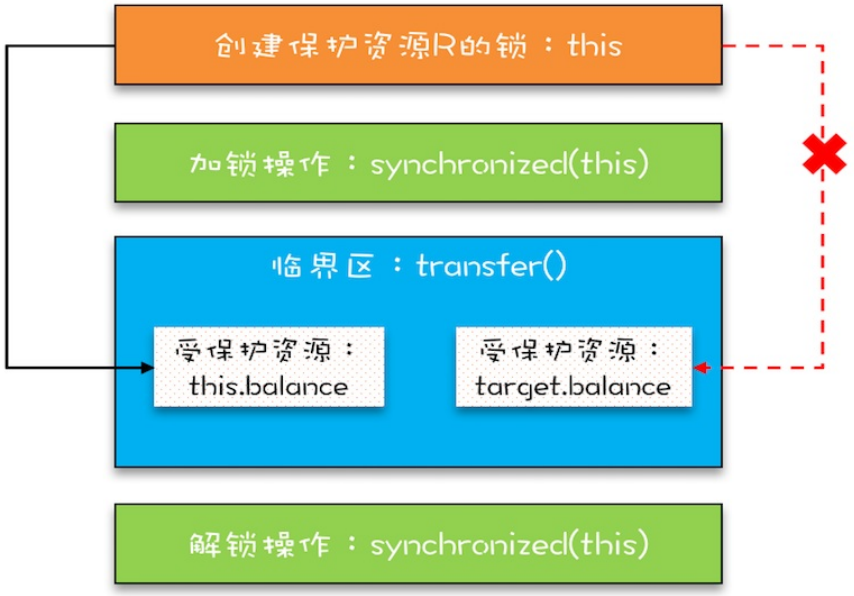
```
class Account {
  private int balance;
  // 转账
  void transfer(
      Account target, int amt){
    if (this.balance > amt) {
      this.balance -= amt;
      target.balance += amt;
    }
  }
}
```

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？

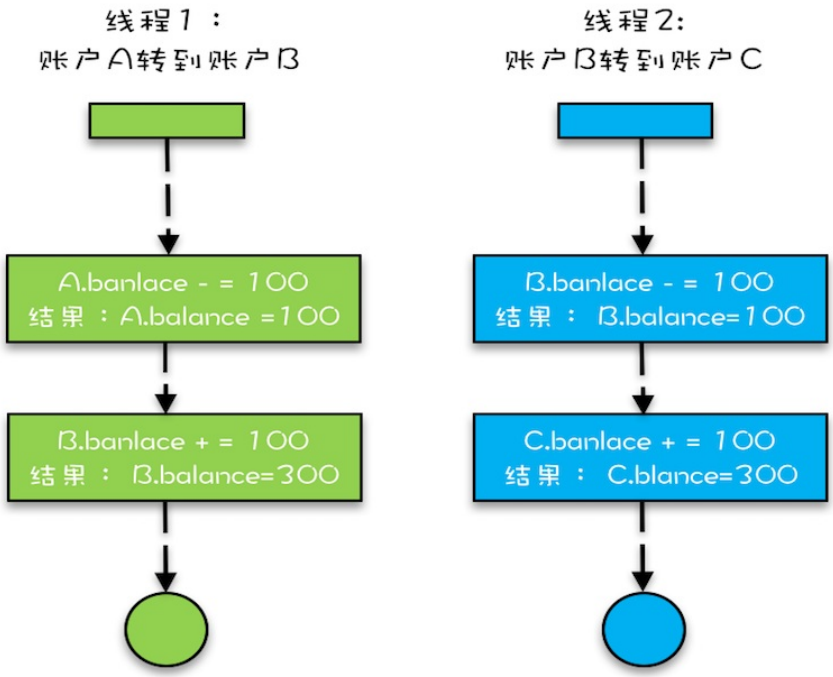
问题就出在this这把锁上，this这把锁可以保护自己的余额this.balance，却保护不了别人的余额target.balance，就像你不能用自家的锁来保护别人家的资产，也不能用自己的票来保护别人的座位一样。



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。

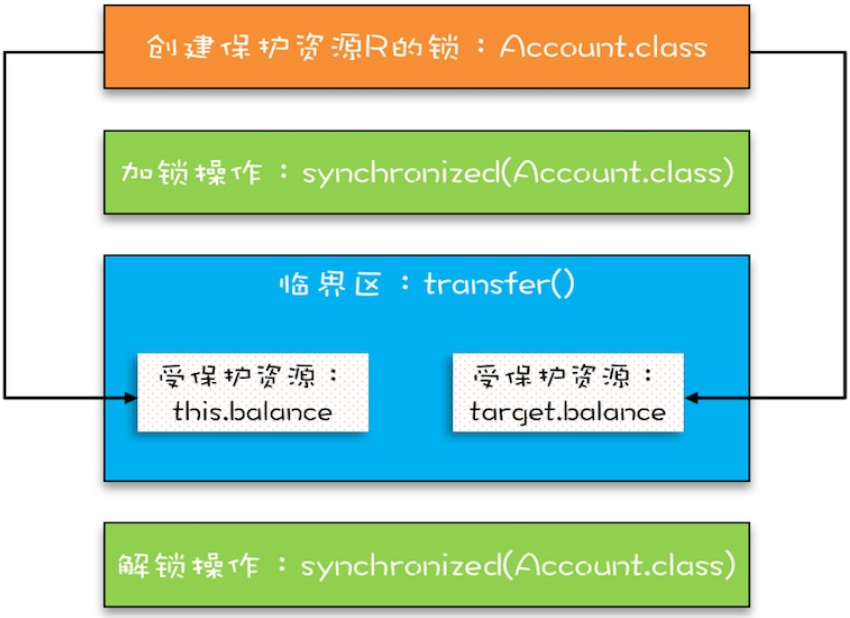
```
class Account {
  private Object lock;
  private int balance;
  private Account();
  // 创建Account时传入同一个lock对象
  public Account(Object lock) {
    this.lock = lock;
  }
  // 转账
  void transfer(Account target, int amt){
    // 此处检查所有对象共享的锁
    synchronized(lock) {
      if (this.balance > amt) {
        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？

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。

猜你喜欢

精选留言：

- 树森 2019-03-07 09:25:12
  有个疑问，使用Account.class获得锁，那所有转账操作不是都成串行了，这里实践中可行吗？ [39赞]

  作者回复2019-03-07 20:03:26
  不可行，下一期讲优化

- 少主江衫 2019-03-07 08:42:07
  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。老师所讲的例子中的两个Object无论多次访问过程中都未发生变化？
  请老师指正。 [34赞]

  作者回复2019-03-07 20:08:00
  正确，不能用可变对象做锁

- senekis 2019-03-07 05:11:34
  解决原子性问题，是要保证中间状态对外不见

太精辟了！ [23赞]

- 夜空中最亮的星（华仔）2019-03-07 16:20:42
  我是一名普通的运维工程师，我是真看不懂java代码，我是来听思想的 。 [9赞]

  作者回复2019-03-07 20:32:14
  那我就放心了

- 老杨同志 2019-03-07 16:20:34
  思考题：
  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 [8赞]

  作者回复2019-03-07 20:32:54
  是的

- 强哥 2019-03-07 12:48:00
  文章里第二个例子根本无法用到实践中，锁力度太大，可以用乐观关锁解决，另外分布式的情况下，应该如何分析也应该讲讲？至于原子性其实跟数据库的原子性还是有差异的，例如虚拟机异常退出时，synchinzed也无法操作原子操作的。 [7赞]

  作者回复2019-03-07 19:47:26
  分布式的不讲了，分支太多不好。下一期会讲优化

- yuc 2019-03-09 11:36:12
  是否可以在Account中添加一个静态object，通过锁这个object来实现一个锁保护多个资源，如下：
  class Account {
  private static Object lock = new Object();
  private int balance;
  // 转账
  void transfer(Account target, int amt){
  synchronized(lock) {
  if (this.balance > amt) {
  this.balance -= amt;
  target.balance += amt;
  }
  }
  }
  }
  [6赞]

  作者回复2019-03-09 13:57:22
  这种方式比锁class更安全，因为这个缺是私有的。有些最佳实践要求必须这样做。👍

- Zach_ 2019-03-07 11:57:33
  王老师， 您在第二讲中贴出的英文链接的地址很棒，看着您写过的专栏，再去看它，有种恍然大悟地感觉~！ 恳请您还是在后续地专栏里，继续保持这种死磕并发基础地原汁原味地链接啊~！ 您地专栏是您多年地理解与实战的营养，加上您亲自地朗读，当然也是原汁原味。但是我的意思是，我们应该有一批人很少看英文类的文档，所以才会有这种恳请~！ 谢谢老师~! [5赞]

作者回复2019-03-07 19:46:09
感谢盛赞，我会继续保持的

- 峰 2019-03-07 08:11:40
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。 [4赞]

作者回复2019-03-07 19:27:21
下一期会讲这个

- 别皱眉 2019-03-13 23:54:52
老师，很感谢有这个专栏，让我能够更加系统的学习并发知识。
对于思考题,之所以不可行是因为每次修改balance和password时都会使锁发生变化。

以下只是我的猜想
比如有线程A、B、C
线程A首先拿到balance1锁，线程B这个时候也过来，发现锁被拿走了，线程B被放入一个地方进行等待。
当A修改掉变量balance的值后，锁由balance1变为balance2.
线程B也拿到那个balance1锁，这时候刚好有线程C过来，拿到了balance2锁。
由于B和C持有的锁不同，所以可以同时执行这个方法来修改balance的值,这个时候就有可能是线程B修改的值会覆盖掉线程C修改的值?

不知道到底是不是这样?老师可以详细讲下这个过程吗?谢谢 [3赞]

作者回复2019-03-14 12:14:09
你分析的很仔细了，就是这样的，bc锁的不是一个对象。不能保证互斥性

- 水如天 2019-03-07 22:48:57
以前碰到一个坑，线程上下文的类加载器改变了，导致前后加载的类不一致 [3赞]

作者回复2019-03-08 12:16:02
osgi？

- walkingonair 2019-03-07 20:22:48
使用Account.class获得锁有很明显的性能问题，而如何解决这个性能问题恰恰是我想知道的。我的一个想法是利用String对象的intern方法生成转账相关的字符串，利用这个字符串作为锁，这个方案不知道在实践中是否可行？
另外，狂战俄洛伊同学提出的锁两个对象的示例，我看着像是经典的死锁案例，因加锁顺序不一致导致的死锁，当A->B，A等待B的锁，而当B->A，B等待A的锁，产生死锁，不知理解的是否有问题，欢迎老师和各位同学指教。 [3赞]

作者回复2019-03-07 21:58:50
下一期介绍优化

- 忠艾一生 2019-03-07 19:50:59
这两把锁是会变的，所以无法保证互斥性。在第一个线程执行完之后，this.balance与this.password这两个对象锁都与第一个线程的对象锁是不一样的。 所以是不正确的。
[3赞]

作者回复2019-03-07 20:33:21
回答正确

- zhaozp 2019-03-07 20:27:16
可变对象不能作为锁 [2赞]

作者回复2019-03-07 21:59:11
总结的到位

- 狂战俄洛伊 2019-03-07 16:27:49
首先，对于文中的课后思考，这个问题肯定是不行的。
synchronized加锁的对象是根据地址来的，只要地址变了，锁也就变了。
当balance或者password发生变化的时候（地址变化），其他线程还是能获得锁，然后执行转账的操作。
所以这里加锁的对象一定要是地址不能变的资源，例如文中的Account.class这个对象地址就不会变化，所以可以用来作为锁。
但是问题来了，如果用Account.class作为加锁对象的话，那所有的转账功能将会是串行。实际上4个不同账户之间的转账是可以并行执行的，例如A给B转账，C给D转账，这两个动作是可以一起执行的。
为了解决这个问题，我对转账代码做了些调整，目的是为了让不同账户之间的转账可以并发运行。下面是我对转账改进后的代码：
```
// 转账
void transfer( Account target, int amt){
synchronized(this) {
synchronized (target) {
if (this.balance > amt) {
this.balance -= amt;
target.balance += amt;
}
}
}
}
```

首先这段代码是线程安全的。

但是我实际测试下来性能却反而不如直接对Account.class加锁高，请问老师这是为什么呢？ [2赞]

作者回复2019-03-07 20:38:20
转账里面sleep一段时间就有效果了

- zyl 2019-03-07 13:34:48
请问这个画图软件是什么？谢谢 [2赞]

作者回复2019-03-09 14:28:48
我已经确认过了，是PPT

- Nevermore 2019-03-07 09:48:52
Integer i = 6 -->Integer i = 7 ，这个过程经历了引用重新指向的问题 ，就相当于new Integer(6) --> new Integer(7)，但是对于Integer和string ，jvm对他们都有优化处理，integer是-127-127内的数值都会放入缓存，而对于本例子的string也会放进常量池，我的理解是，假如A,B线程同时修改余额，当A修改余额成功之前，A,B线程看到的都是同一个余额，也就是同一个对象，当A修改成功后，改变了对象，原先的锁就不是原来的锁了，对于线程A来说，就相当于释放了锁，但是，当余额的范围超过了127之后，以上的逻辑就失效了。而密码的修改不会存在这个问题，能完全的锁住。

以上是我的理解，望老师指正！ [2赞]

作者回复2019-03-07 19:59:21
密码的修改也锁不住

- 落叶1 2019-03-07 09:06:17
思考题:受保护资源作为锁保护自己，跟没加锁一样。修改金额后，锁对象都变了，加锁不成功 [2赞]

作者回复2019-03-07 19:22:21
是的

- wang 2019-03-07 08:48:21
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。 [2赞]

作者回复2019-03-07 19:09:29
深刻！👍

- linqw 2019-03-09 22:06:00
老师，有个疑问，锁是为了保证在多个线程操作，每个线程的操作之前多能先读取到其余线程的操作结果，保证数据的一致性，防止数据被覆盖，但是为什么锁不能使用可变的对象作为锁，Integer和String都是不可变对象，操作都会生成新对象，但是这个会影响到多个线程对数据的操作结果么？比如取款，虽然取款完锁对象改变，但是在并发的时候，不管是其他线程在前一个线程取款前获取到的锁还是取款后获取到的锁，对数据都不会覆盖呀？老师帮忙解答下哦 [1赞]

作者回复2019-03-09 22:56:23
多把锁保护一个资源，起不到互斥作用，就可能同时取款，这样数据就被覆盖了。