05、一不小心就死锁了怎么办?

如何避免死锁

只要以下四个条件都发生才会出现死锁:

- 互斥:共享资源X和Y只能被一个线程占用
- 占有且等待,线程T1已经获得共享资源X,在等待共享资源Y的时候,不释放共享资源Y
- 不可抢占,其他线程不能强行抢占线程T1占有的资源
- 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 **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,因为我们用锁的目的就是互斥,不过其他三个条件都是有办法破坏的。

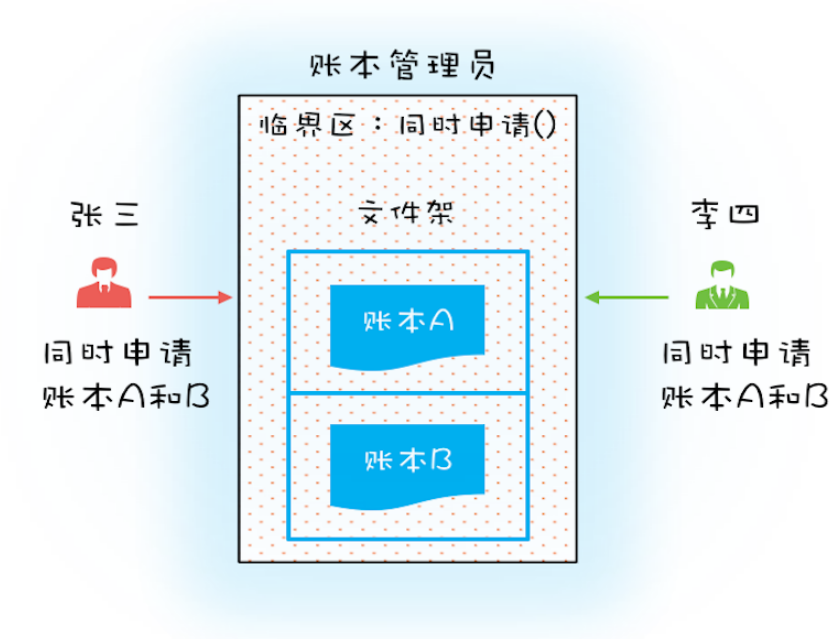
1. 对于"占用切且等待"这个条件,我们可以一次申请所有资源,这样就不存在等待了。
2. 对于"不可抢占"这个条件,占用部分资源的线程进一步申请其他资源时,如果申请不到,可以主动释放它占有的资源,这样不可抢占这个条件就破坏了。
3. 对于"循环等待"这个条件,可以考按序申请来预防,所谓按序申请,是指资源是有线性顺序的,申请的时候可以先申请资源序号小的,再申请资源序号大的,这样线性化后自然就不存在循环了。

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

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"。



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。

```
class Allocator {
  private List<Object> als =
    new ArrayList<>();
  // 一次性申请所有资源
  synchronized boolean apply(
    Object from, Object to){
    if(als.contains(from) ||
         als.contains(to)){
      return false;
    } else {
      als.add(from);
      als.add(to);
    }
    return true;
  }
  // 归还资源
  synchronized void free(
    Object from, Object to){
    als.remove(from);
    als.remove(to);
  }
}

class Account {
  // actr 应该为单例
  private Allocator actr;
  private int balance;
  // 转账
  void transfer(Account target, int amt){
    // 一次性申请转出账户和转入账户，直到成功
    while(!actr.apply(this, target))
      ;
    try{
      // 锁定转出账户
      synchronized(this){
        // 锁定转入账户
        synchronized(target){
          if (this.balance > amt){
            this.balance -= amt;
            target.balance += amt;
          }
        }
      }
    } finally {
      actr.free(this, target)
    }
  }
}
```

2、破坏不可抢占

破坏不可抢占条件看上去简单，核心是要能够主动释放它占有的资源，这一点synchonized是做不到的，原因是synchoronized申请资源的时候，如果申请不到，线程就直接进入阻塞状态了，线程进入阻塞状态，啥都不干了，也释放不聊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。

```
class Account {
  private int id;
  private int balance;
  // 转账
  void transfer(Account target, int amt){
    Account left = this        ①
    Account right = target;    ②
    if (this.id > target.id) { ③
      left = target;           ④
      right = this;            ⑤
    }                          ⑥
    // 锁定序号小的账户
    synchronized(left){
      // 锁定序号大的账户
      synchronized(right){
        if (this.balance > amt){
          this.balance -= amt;
          target.balance += amt;
        }
      }
    }
  }
}
```

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，也更能够看清楚问题的本质。

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。