二

70 有哪些解决死锁问题的策略?

本课时我们主要介绍有哪些解决死锁的策略。

线上发生死锁应该怎么办

如果线上环境发生了死锁，那么其实不良后果就已经造成了，修复死锁的**最好时机在于“防患于未然”**，而不是事后补救。就好比发生火灾时，一旦着了大火，想要不造成损失去扑灭几乎已经不可能了。死锁也是一样的，如果线上发生死锁问题，为了尽快减小损失，最好的办法是保存 JVM 信息、日志等“案发现场”的数据，然后**立刻重启服务**，来尝试修复死锁。为什么说重启服务能解决这个问题呢？因为发生死锁往往要有很多前提条件的，并且当并发度足够高的时候才有可能会发生死锁，所以**重启后再次立刻发生死锁的几率并不是很大**，当我们重启服务器之后，就可以暂时保证线上服务的可用，然后利用刚才保存过的案发现场的信息，**排查死锁、修改代码，最终重新发布**。

常见修复策略

我们有哪些常见的对于死锁的修复策略呢？下面将会介绍三种主要的修复策略，分别是：

- 避免策略
- 检测与恢复策略
- 鸵鸟策略

它们侧重各不相同，我们首先从避免策略说起。

避免策略

如何避免

避免策略最主要的思路就是，**优化代码逻辑，从根本上消除发生死锁的可能性**。通常而言，发生死锁的一个主要原因是顺序相反的去获取不同的锁。因此我们就演示如何通过**调整锁的获取顺序**来避免死锁。

转账时避免死锁

我们先来看一下转账时发生死锁的情况。这个例子是一个示意性的，是为了学习死锁所而写的例子，所以和真实的银行系统的设计有很大不同，不过没关系，因为我们主要看的是如何避免死锁，而不是转账的业务逻辑。

（1）发生了死锁

我们的转账系统为了保证线程安全，**在转账前需要首先获取到两把锁**（两个锁对象），分别是被转出的账户和被转入的账户。如果不做这一层限制，那么在某一个线程修改余额的期间，可能会有其他线程同时修改该变量，可能导致线程安全问题。所以在没有获取到这两把锁之前，是不能对余额进行操作的；只有获取到这两把锁之后，才能进行接下来真正的转账操作。当然，如果要转出的余额大于账户的余额，也不能转账，因为不允许余额变成负数。

而这期间就隐藏着发生死锁的可能，我们来看下代码：

```
public class TransferMoney implements Runnable {
    int flag;
    static Account a = new Account(500);
    static Account b = new Account(500);
    static class Account {
        public Account(int balance) {
            this.balance = balance;
        }
        int balance;
    }
    @Override
    public void run() {
        if (flag == 1) {
            transferMoney(a, b, 200);
        }
        if (flag == 0) {
            transferMoney(b, a, 200);
```

```
    }
}
public static void transferMoney(Account from, Account to, int amount) {
    //先获取两把锁，然后开始转账
    synchronized (to) {
        synchronized (from) {
            if (from.balance - amount < 0) {
                System.out.println("余额不足，转账失败。");
                return;
            }
            from.balance -= amount;
            to.balance += amount;
            System.out.println("成功转账" + amount + "元");
        }
    }
}
public static void main(String[] args) throws InterruptedException {
    TransferMoney r1 = new TransferMoney();
    TransferMoney r2 = new TransferMoney();
    r1.flag = 1;
    r2.flag = 0;
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println("a的余额" + a.balance);
```

```
        System.out.println("b的余额" + b.balance);
        }
    }
```

在代码中，首先定义了 int 类型的 flag，它是一个标记位，用于控制不同线程执行不同逻辑。然后建了两个 Account 对象 a 和 b，代表账户，它们最初都有 500 元的余额。

我们接下来看 run 方法，该方法里面会根据 flag 值，来决定传入 transferMoney 方法的参数的顺序，如果 flag 为 1，那么就代表从 a 账户转给 b 账户 200元；相反，如果 flag 为 0，那么它就从 b 账户转给 a 账户 200 元。

我们再来看一下 transferMoney 转账方法，这个方法会先尝试获取两把锁，也就是 synchronized (to) 和 synchronized (from)。当都获取成功之后，它首先会判断余额是不是足以转出本次的转账金额，如果不足的话，则直接用 return 来退出；如果余额足够，就对转出账户进行减余额，对被转入的账户加余额，最后打印出“成功转账 XX 元”的消息。

在主函数中我们新建了两个 TransferMoney 对象，并且把它们的 flag 分别设置为 1 和 0，然后分别传入两个线程中，并把它们都启动起来，最后，打印出各自的余额。

执行结果如下：

```
成功转账200元
成功转账200元
a的余额500
b的余额500
```

代码是可以正常执行的，打印结果也是符合逻辑的。此时并没有发生死锁，因为**每个锁的持有时间很短，同时释放也很快**，所以在低并发的情况下，不容易发生死锁的现象。那我们对代码做一些小调整，让它发生死锁。

如果我们在两个 synchronized 之间加上一个 Thread.sleep(500)，来模拟银行**网络迟延**等情况，那么 transferMoney 方法就变为：

```
public static void transferMoney(Account from, Account to, int amount) {
    //先获取两把锁，然后开始转账
    synchronized (to) {
        try {
```

```
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (from) {
                if (from.balance - amount < 0) {
                    System.out.println("余额不足，转账失败。");
                    return;
                }
                from.balance -= amount;
                to.balance += amount;
                System.out.println("成功转账" + amount + "元");
            }
        }
    }
```

可以看到 transferMoney 的变化就在于，在两个 synchronized 之间，也就是获取到第一把锁后、获取到第二把锁前，我们加了睡眠 500 毫秒的语句。此时再运行程序，会有很大的概率发生死锁，从而导致**控制台中不打印任何语句，而且程序也不会停止**。

我们分析一下它为什么会发生死锁，最主要原因就是，两个不同的线程**获取两个锁的顺序是相反的**（第一个线程获取的这两个账户和第二个线程获取的这两个账户顺序恰好相反，**第一个线程的“转出账户”正是第二个线程的“转入账户”**），所以我们就可以从这个“相反顺序”的角度出发，来解决死锁问题。

（2）实际上不在乎获取锁的顺序

经过思考，我们可以发现，其实转账时，并不在乎两把锁的相对获取顺序。转账的时候，我们无论先获取到转出账户锁对象，还是先获取到转入账户锁对象，只要最终能拿到两把锁，就能进行安全的操作。所以我们来调整一下获取锁的顺序，使得先获取的账户和该账户是“转入”或“转出”无关，而是**使用 HashCode 的值来决定顺序**，从而保证线程安全。

修复之后的 transferMoney 方法如下：

```
public static void transferMoney(Account from, Account to, int amount) {
    int fromHash = System.identityHashCode(from);
    int toHash = System.identityHashCode(to);
    if (fromHash < toHash) {
        synchronized (from) {
            synchronized (to) {
                if (from.balance - amount < 0) {
                    System.out.println("余额不足，转账失败。");
                    return;
                }
                from.balance -= amount;
                to.balance += amount;
                System.out.println("成功转账" + amount + "元");
            }
        }
    } else if (fromHash > toHash) {
        synchronized (to) {
            synchronized (from) {
                if (from.balance - amount < 0) {
                    System.out.println("余额不足，转账失败。");
                    return;
                }
                from.balance -= amount;
                to.balance += amount;
                System.out.println("成功转账" + amount + "元");
            }
        }
    }
```

```
}
```

可以看到，我们会分别计算出这两个 Account 的 HashCode，然后根据 HashCode 的大小来决定获取锁的顺序。这样一来，不论是哪个线程先执行，不论是转出还是被转入，它获取锁的顺序都会严格根据 HashCode 的值来决定，那么**大家获取锁的顺序就一样了，就不会出现获取锁顺序相反的情况**，也就避免了死锁。

（3）有主键就更安全、方便

下面我们看一下用主键决定锁获取顺序的方式，它会更加的安全方便。刚才我们使用了 HashCode 作为排序的标准，因为 HashCode 比较通用，每个对象都有，不过这依然有极小的概率会发生 HashCode 相同的情况。在实际生产中，需要排序的往往是一个实体类，而一个实体类一般都会有一个主键 ID，**主键 ID 具有唯一、不重复的特点**，所以如果我们这个类包含主键属性的话就方便多了，我们也没必要去计算 HashCode，直接使用它的主键 ID 来进行排序，由主键 ID 大小来决定获取锁的顺序，就可以确保避免死锁。

以上我们介绍了死锁的避免策略。

检测与恢复策略

下面我们再来看第二个策略，那就是检测与恢复策略。

什么是死锁检测算法

它和之前避免死锁的策略不一样，避免死锁是通过逻辑让死锁不发生，而这里的检测与恢复策略，是**先允许系统发生死锁，然后再解除。**例如系统可以在每次调用锁的时候，都记录下来调用信息，形成一个“锁的调用链路图”，然后隔一段时间就用死锁检测算法来检测一下，搜索这个图中是否存在环路，一旦发生死锁，就可以用死锁恢复机制，比如剥夺某一个资源，来解开死锁，进行恢复。所以它的思路和之前的死锁避免策略是有很大不同的。

在检测到死锁发生后，如何解开死锁呢？

方法1——线程终止

第一种解开死锁的方法是线程（或进程，下同）终止，在这里，系统会逐个去终止已经陷入死锁的线程，线程被终止，同时释放资源，这样死锁就会被解开。

当然这个终止是需要讲究顺序的，一般有以下几个考量指标。

（1）优先级

一般来说，终止时会考虑到线程或者进程的优先级，先终止优先级低的线程。例如，前台线程会涉及界面显示，这对用户而言是很重要的，所以前台线程的优先级往往高于后台线程。

（2）已占用资源、还需要的资源

同时也会考虑到某个线程占有的资源有多少，还需要的资源有多少？如果某线程已经占有了一大堆资源，只需要最后一点点资源就可以顺利完成任务，那么系统可能就不会优先选择终止这样的线程，会选择终止别的线程来优先促成该线程的完成。

（3）已经运行时间

另外还可以考虑的一个因素就是已经运行的时间，比如当前这个线程已经运行了很多个小时，甚至很多天了，很快就能完成任务了，那么终止这个线程可能不是一个明智的选择，我们可以让那些刚刚开始运行的线程终止，并在之后把它们重新启动起来，这样成本更低。

这里会有各种各样的算法和策略，我们根据实际业务去进行调整就可以了。

方法2——资源抢占

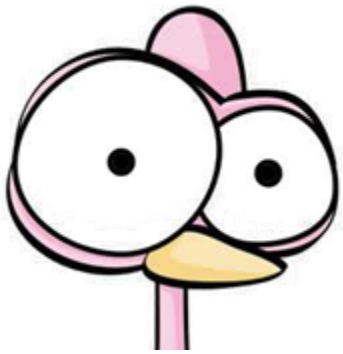
第二个解开死锁的方法就是资源抢占。其实，我们不需要把整个的线程终止，而是只需要把它已经获得的资源进行剥夺，比如让线程回退几步、 释放资源，这样一来就不用终止掉整个线程了，这样造成的后果会比刚才终止整个线程的后果更小一些，**成本更低**。

当然这种方式也有一个缺点，那就是如果算法不好的话，我们抢占的那个线程可能一直是同一个线程，就会造成**线程饥饿**。也就是说，这个线程一直被剥夺它已经得到的资源，那么它就长期得不到运行。

以上就是死锁的检测与恢复策略。

鸵鸟策略

下面我们再来看一下鸵鸟策略，鸵鸟策略以鸵鸟命名，因为鸵鸟有一个特点，就是遇到危险的时候，它会把头埋到沙子里，这样一来它就看不到危险了。



鸵鸟策略的意思就是，如果我们的系统发生死锁的概率不高，并且一旦发生其后果不是特别严重的话，我们就可以选择先忽略它。直到死锁发生的时候，我们再人工修复，比如重启服务，这并不是不可以的。如果我们的系统用的人比较少，比如是**内部的系统，那么在并发量极低的情况下，它可能几年都不会发生死锁**。对此我们考虑到投入产出比，自然也没有必要去对死锁问题进行特殊的处理，这是需要根据我们的业务场景进行合理选择的。

总结

本课时我们主要介绍了有哪些解决死锁的策略。首先介绍了在线上发生死锁的时候，应该在保存了重要数据后，优先恢复线上服务；然后介绍了三种具体的修复策略：一是避免策略，其主要思路就是去改变锁的获取顺序，防止相反顺序获取锁这种情况的发生；二是检测与恢复策略，它是允许死锁发生，但是一旦发生之后它有解决方案；三是鸵鸟策略。

上一页 下一页