

## 14 | Lock和Condition（上）：隐藏在并发包中的管程

2019-03-30 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 11:03 大小 10.14M



Java SDK 并发包内容很丰富，包罗万象，但是我觉得最核心的还是其对管程的实现。因为理论上利用管程，你几乎可以实现并发包里所有的工具类。在前面[《08 | 管程：并发编程的万能钥匙》](#)中提到过在并发编程领域，有两大核心问题：一个是**互斥**，即同一时刻只允许一个线程访问共享资源；另一个是**同步**，即线程之间如何通信、协作。这两大问题，管程都是能够解决的。**Java SDK 并发包通过 Lock 和 Condition 两个接口来实现管程，其中 Lock 用于解决互斥问题，Condition 用于解决同步问题。**

今天我们重点介绍 Lock 的使用，在介绍 Lock 的使用之前，有个问题需要你首先思考一下：Java 语言本身提供的 synchronized 也是管程的一种实现，既然 Java 从语言层面已经实现了管程了，那为什么还要在 SDK 里提供另外一种实现呢？难道 Java 标准委员会还能同意“重复造轮子”的方案？很显然它们之间是有巨大区别的。那区别在哪里呢？如果能深入理解这个问题，对你用好 Lock 帮助很大。下面我们就一起来剖析一下这个问题。

## 再造管程的理由

你也许曾经听到过很多这方面的传说，例如在 Java 的 1.5 版本中，synchronized 性能不如 SDK 里面的 Lock，但 1.6 版本之后，synchronized 做了很多优化，将性能追了上来，所以 1.6 之后的版本又有人推荐使用 synchronized 了。那性能是否可以成为“重复造轮子”的理由呢？显然不能。因为性能问题优化一下就可以了，完全没必要“重复造轮子”。


到这里，关于这个问题，你是否能够想出一条理由来呢？如果你细心的话，也许能想到一点。那就是我们前面在介绍[死锁问题](#)的时候，提出了一个**破坏不可抢占条件**方案，但是这个方案 synchronized 没有办法解决。原因是 synchronized 申请资源的时候，如果申请不到，线程直接进入阻塞状态了，而线程进入阻塞状态，啥都干不了，也释放不了线程已经占有的资源。但我们希望的是：

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掉了。

如果我们重新设计一把互斥锁去解决这个问题，那该怎么设计呢？我觉得有三种方案。

1. **能够响应中断**。synchronized 的问题是，持有锁 A 后，如果尝试获取锁 B 失败，那么线程就进入阻塞状态，一旦发生死锁，就没有任何机会来唤醒阻塞的线程。但如果阻塞状态的线程能够响应中断信号，也就是说当我们给阻塞的线程发送中断信号的时候，能够唤醒它，那它就有机会释放曾经持有的锁 A。这样就破坏了不可抢占条件了。
2. **支持超时**。如果线程在一段时间之内没有获取到锁，不是进入阻塞状态，而是返回一个错误，那这个线程也有机会释放曾经持有的锁。这样也能破坏不可抢占条件。
3. **非阻塞地获取锁**。如果尝试获取锁失败，并不进入阻塞状态，而是直接返回，那这个线程也有机会释放曾经持有的锁。这样也能破坏不可抢占条件。

这三种方案可以全面弥补 synchronized 的问题。到这里相信你应该也能理解了，这三个方案就是“重复造轮子”的主要原因，体现在 API 上，就是 Lock 接口的三个方法。详情如下：


 复制代码

```
1 // 支持中断的 API
2 void lockInterruptibly()
3     throws InterruptedException;
4 // 支持超时的 API
```

```
5 boolean tryLock(long time, TimeUnit unit)
6     throws InterruptedException;
7 // 支持非阻塞获取锁的 API
8 boolean tryLock();
```

## 如何保证可见性


Java SDK 里面 Lock 的使用，有一个经典的范例，就是 `try{}finally{}`，需要重点关注的是在 `finally` 里面释放锁。这个范例无需多解释，你看一下下面的代码就明白了。但是有一点需要解释一下，那就是可见性是怎么保证的。你已经知道 Java 里多线程的可见性是通过 Happens-Before 规则保证的，而 `synchronized` 之所以能够保证可见性，也是因为有一条 `synchronized` 相关的规则：`synchronized` 的解锁 Happens-Before 于后续对这个锁的加锁。那 Java SDK 里面 Lock 靠什么保证可见性呢？例如在下面的代码中，线程 T1 对 `value` 进行了 `+=1` 操作，那后续的线程 T2 能够看到 `value` 的正确结果吗？

 复制代码

```
1 class X {
2     private final Lock rtl =
3         new ReentrantLock();
4     int value;
5     public void addOne() {
6         // 获取锁
7         rtl.lock();
8         try {
9             value+=1;
10        } finally {
11            // 保证锁能释放
12            rtl.unlock();
13        }
14    }
15 }
```

答案必须是肯定的。**Java SDK 里面锁**的实现非常复杂，这里我就不展开细说了，但是原理还是需要简单介绍一下：它是**利用了 volatile 相关的 Happens-Before 规则**。Java SDK 里面的 `ReentrantLock`，内部持有一个 `volatile` 的成员变量 `state`，获取锁的时候，会读写 `state` 的值；解锁的时候，也会读写 `state` 的值（简化后的代码如下面所示）。也就是说，在执行 `value+=1` 之前，程序先读写了一次 `volatile` 变量 `state`，在执行 `value+=1` 之后，又读写了一次 `volatile` 变量 `state`。根据相关的 Happens-Before 规则：

1. **顺序性规则**：对于线程 T1，`value+=1` Happens-Before 释放锁的操作 `unlock()`；
2. **volatile 变量规则**：由于 `state = 1` 会先读取 `state`，所以线程 T1 的 `unlock()` 操作 Happens-Before 线程 T2 的 `lock()` 操作；
3. **传递性规则**：线程 T1 的 `value+=1` Happens-Before 线程 T2 的 `lock()` 操作。

 复制代码

```
1 class SampleLock {
2     volatile int state;
3     // 加锁
4     lock() {
5         // 省略代码无数
6         state = 1;
7     }
8     // 解锁
9     unlock() {
10        // 省略代码无数
11        state = 0;
12    }
13 }
```

所以说，后续线程 T2 能够看到 `value` 的正确结果。如果你觉得理解起来还有点困难，建议你重温一下前面我们讲过的 [《02 | Java 内存模型：看 Java 如何解决可见性和有序性问题》](#) 里面的相关内容。

## 什么是可重入锁

如果你细心观察，会发现我们创建的锁的具体类名是 `ReentrantLock`，这个翻译过来叫**可重入锁**，这个概念前面我们一直没有介绍过。**所谓可重入锁，顾名思义，指的是线程可以重复获取同一把锁。**例如下面代码中，当线程 T1 执行到 ① 处时，已经获取到了锁 `rtl`，当在 ① 处调用 `get()` 方法时，会在 ② 再次对锁 `rtl` 执行加锁操作。此时，如果锁 `rtl` 是可重入的，那么线程 T1 可以再次加锁成功；如果锁 `rtl` 是不可重入的，那么线程 T1 此时会被阻塞。

除了可重入锁，可能你还听说过可重入函数，可重入函数怎么理解呢？指的是线程可以重复调用？显然不是，所谓**可重入函数，指的是多个线程可以同时调用该函数**，每个线程都能得到正确结果；同时在一个线程内支持线程切换，无论被切换多少次，结果都是正确的。多线程可以同时执行，还支持线程切换，这意味着什么呢？线程安全啊。所以，可重入函数是线程安全的。

```

1 class X {
2     private final Lock rtl =
3         new ReentrantLock();
4     int value;
5     public int get() {
6         // 获取锁
7         rtl.lock();           ②
8         try {
9             return value;
10        } finally {
11            // 保证锁能释放
12            rtl.unlock();
13        }
14    }
15    public void addOne() {
16        // 获取锁
17        rtl.lock();
18        try {
19            value = 1 + get(); ①
20        } finally {
21            // 保证锁能释放
22            rtl.unlock();
23        }
24    }
25 }

```

## 公平锁与非公平锁

在使用 `ReentrantLock` 的时候，你会发现 `ReentrantLock` 这个类有两个构造函数，一个是无参构造函数，一个是传入 `fair` 参数的构造函数。`fair` 参数代表的是锁的公平策略，如果传入 `true` 就表示需要构造一个公平锁，反之则表示要构造一个非公平锁。

```

1 // 无参构造函数：默认非公平锁
2 public ReentrantLock() {
3     sync = new NonfairSync();
4 }
5 // 根据公平策略参数创建锁
6 public ReentrantLock(boolean fair){
7     sync = fair ? new FairSync()
8             : new NonfairSync();
9 }

```



在前面[《08 | 管程：并发编程的万能钥匙》](#)中，我们介绍过入口等待队列，锁都对应着一个等待队列，如果一个线程没有获得锁，就会进入等待队列，当有线程释放锁的时候，就需要从等待队列中唤醒一个等待的线程。如果是公平锁，唤醒的策略就是谁等待的时间长，就唤醒谁，很公平；如果是非公平锁，则不提供这个公平保证，有可能等待时间短的线程反而先被唤醒。

## 用锁的最佳实践

你已经知道，用锁虽然能解决很多并发问题，但是风险也是挺高的。可能会导致死锁，也可能影响性能。这方面有是否有相关的最佳实践呢？有，还很多。但是我觉得最值得推荐的是并发大师 Doug Lea 《Java 并发编程：设计原则与模式》一书中，推荐的三个用锁的最佳实践，它们分别是：

1. 永远只在更新对象的成员变量时加锁
2. 永远只在访问可变的成员变量时加锁
3. 永远不在调用其他对象的方法时加锁

这三条规则，前两条估计你一定会认同，最后一条你可能会觉得过于严苛。但是我还是倾向于你去遵守，因为调用其他对象的方法，实在是太不安全了，也许“其他”方法里面有线程 `sleep()` 的调用，也可能会有奇慢无比的 I/O 操作，这些都会严重影响性能。更可怕的是，“其他”类的方法可能也会加锁，然后双重加锁就可能导致死锁。

**并发问题，本来就难以诊断，所以你一定要让你的代码尽量安全，尽量简单，哪怕有一点可能会出问题，都要努力避免。**


## 总结

Java SDK 并发包里的 `Lock` 接口里面的每个方法，你可以感受到，都是经过深思熟虑的。除了支持类似 `synchronized` 隐式加锁的 `lock()` 方法外，还支持超时、非阻塞、可中断的方式获取锁，这三种方式为我们编写更加安全、健壮的并发程序提供了很大的便利。希望你以后在使用锁的时候，一定要仔细斟酌。

除了并发大师 Doug Lea 推荐的三个最佳实践外，你也可以参考一些诸如：减少锁的持有时间、减小锁的粒度等业界广为人知的规则，其实本质上它们都是相通的，不过是在该加锁的地方加锁而已。你可以自己体会，自己总结，最终总结出自己的一套最佳实践来。

## 课后思考

你已经知道 tryLock() 支持非阻塞方式获取锁，下面这段关于转账的程序就使用到了 tryLock()，你来看看，它是否存在死锁问题呢？

 复制代码

```
1 class Account {
2     private int balance;
3     private final Lock lock
4         = new ReentrantLock();
5     // 转账
6     void transfer(Account tar, int amt){
7         while (true) {
8             if(this.lock.tryLock()) {
9                 try {
10                     if (tar.lock.tryLock()) {
11                         try {
12                             this.balance -= amt;
13                             tar.balance += amt;
14                         } finally {
15                             tar.lock.unlock();
16                         }
17                     } //if
18                 } finally {
19                     this.lock.unlock();
20                 }
21             } //if
22         } //while
23     } //transfer
24 }
```

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」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 理论基础模块热点问题答疑

下一篇 15 | Lock和Condition（下）：Dubbo如何用管程实现异步转同步？

## 精选留言 (44)

 写留言




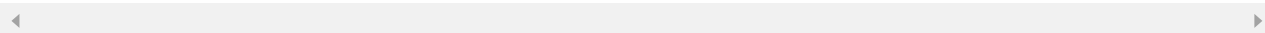
2019-03-30

 21

我觉得:不会出现死锁，但会出现活锁

展开 ▾

作者回复: 





xiyi

2019-03-30

 15

存在活锁。这个例子可以稍微改下，成功转账后应该跳出循环。加个随机重试时间避免活



锁

作者回复: 🍵🍵🍵



bing

2019-03-30

🍵 13

文中说的公平锁和非公平锁，是不按照排队的顺序被唤醒，我记得非公平锁的场景应该是线程释放锁之后，如果来了一个线程获取锁，他不必去排队直接获取到，应该不会入队吧。获取不到才进吧

展开 ∨

作者回复: 是的，高手 🍵🍵🍵



刘晓林

2019-03-30

🍵 8

- 1.这个是个死循环啊，有锁没群，都出不来。
- 2.如果抛开死循环，也会造成活锁，状态不稳定。当然这个也看场景，假如冲突窗口很小，又在单机多核的话，活锁的可能性还是很小的，可以接受

展开 ∨

作者回复: 🍵🍵🍵



Q宝的宝

2019-03-30

🍵 7

老师，本文在讲述如何保证可见性时，分析示例--“线程 T1 对 value 进行了 +=1 操作后，后续的线程 T2 能否看到 value 的正确结果？”时，提到三条Happen-Before规则，这里在解释第2条和第3条规则时，似乎说反了，正确的应该是，根据volatile变量规则，线程T1的unlock()操作Happen-Before于线程T2的lock()操作，所以，根据传递性规则，线程 T1 的 value +=1操作Happen-Before于线程T2的lock()操作。请老师指正。

展开 ∨

作者回复: 火眼金睛 🍵🍵🍵🍵，这就改过来



小华

2019-03-30

👍 6

有可能活锁，A，B两账户相互转账，各自持有自己lock的锁，都一直在尝试获取对方的锁，形成了活锁

作者回复: 📬



羊三@XCoin...

2019-03-30

👍 6

用非阻塞的方式去获取锁，破坏了第五章所说的产生死锁的四个条件之一的“不可抢占”。所以不会产生死锁。

用锁的最佳实践，第三个“永远不在调用其他对象的方法时加锁”，我理解其实是在工程规范上避免可能出现的锁相关问题。

展开 ▾

作者回复: 是的



姜戈

2019-03-30

👍 5

我也觉得是存在活锁，而非死锁。存在这种可能性：互相持有各自的锁，发现需要的对方的锁都被对方持有，就会释放当前持有的锁，导致大家都在不停持锁，释放锁，但事情还没做。当然还是会存在转账成功的情景，不过效率低下。我觉得此时需要引入Condition，协调两者同步处理转账！

作者回复: 用condition会更复杂




Liam

2019-03-30

👍 2

- 1 不会出现死锁，因为不存在阻塞的情况
- 2 线程较多的情况会导致部分线程始终无法获取到锁，导致活锁

作者回复: 



海鸿

2019-03-30

 2

突然有个问题：

cpu层面的原子性是单条cpu指令。

java层面的互斥（管程）保证了原子性。

这两个原子性意义应该不一样吧？

我的理解是cpu的原子性是不受线程调度影响，指令要不执行了，要么没执行。而java层...

展开 

作者回复: 对



朱小豪

2019-03-30

 2

应该是少了个break跳出循环，然后这个例子是会产生死锁的，因为满足了死锁产生的条件。

作者回复: 加了break，也会有活锁问题，不加的话我觉得也是活锁，因为锁都会释放



朱小豪

2019-03-30

 2

本文最后的例子，不明白为什么要用while true而且没有跳出循环，这不是死循环吗



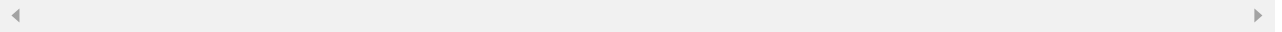
tdytaylo...

2019-05-15

 1

老师，关于这个问题，我思考之后觉得不会出现死锁，但是没看出为什么会出现活锁

作者回复: 想想对面相遇的两个人互相谦让的例子看看



尹圣

2019-04-11

👍 1

```
public class Main {
```

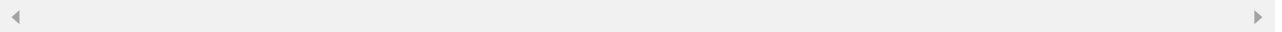
```
    static volatile int state = 0;
```

```
    public static long account = 0;...
```

展开 ▾

作者回复: 不知道你说的不安全是指哪里, state 你只是写了, 没有读, 而且account++也不是互斥的操作。

java并发包里用volatile保证可见性, 还用aqs实现了互斥。保证线程安全不是这么简单的。



linqw

2019-04-07

👍 1

```
class Account {
```

```
    private int balance;
```

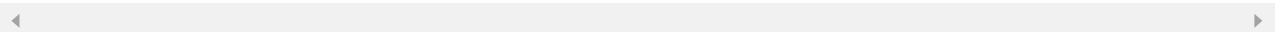
```
    private final Lock lock
```

```
        = new ReentrantLock();
```

```
    // 转账...
```

展开 ▾

作者回复: 点个大大的赞! 不过还可以再优化一下, 如果阻塞在tar.lock.tryLock上一段时间, this.lock是不能释放的。



右耳听海

2019-03-31

👍 1

请问state=1先读取是怎么得出来的, 还有lock和unlock的方法对state都是写操作, 怎么用到valiate规则的, valiate规则不是读取操作先与写操作吗, 这个地方两个都是写操作

展开 ▾

作者回复: =1之前有一段代码会查看状态是否为0, 显然不能三七二十一直接设置



JackJin

2019-03-31

👍 1

老师您好:

那在解决这个活锁问题时, 是在获取其他对象锁前面 (tar.lock.tryLock()) 加个随机线程睡眠时间? 还是《java编程: 设计原则与模式》中的第三条, 永远不在调用其他对象时加锁; 去掉 (tar.lock.tryLock()) 这个锁来解决活锁呢?

作者回复: 加个随机线程睡眠时间就可以了



alias cd=...

2019-03-31

👍 1

不会死锁因为, 打破了不释放的原则。

展开 ∨



JGOS

2019-06-03

👍

老师 如果线程t2, 不加锁直接读value,是不是会读到旧数据?

展开 ∨



假装自己不...

2019-05-30

👍

```
if(this.lock.tryLock()) {  
    try {  
        this.balance -= amt;  
        if (tar.lock.tryLock()) {  
            try {...
```

展开 ∨

