**题目：Lock和Condition(上)：隐藏在并发包中的管程**[](" \l "lockcondition" \o "Permanent link)

**再造管程理由**[](" \l "_1" \o "Permanent link)

**如何保证可见性**[](" \l "_2" \o "Permanent link)

**什么是可重入锁**[](" \l "_3" \o "Permanent link)

**公平锁与非公平锁**[](" \l "_4" \o "Permanent link)

**用锁的最佳实践**[](" \l "_5" \o "Permanent link)

**总结**[](" \l "_6" \o "Permanent link)

在编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。这两个问题，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是用来解决互斥问题，Condition是用于解决同步问题。**

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的区别的。那么区别在哪呢？如果能够深入理解这个问题，对你用好Lock帮助很大的。

**再造管程的理由**[](" \l "_7" \o "Permanent link)

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”。

如果细心的话，也许能想到一点。那就是：我们分析如何解决死锁问题时提出了一个 **破坏不可抢占条件** 方案，但是这个方案synchronized没有办法解决。原因就是Synchronized申请资源的时候，如果申请不到，线程直接进入了阻塞状态了，耳线城进入了阻塞状态，啥都管不了了，也释放不了线程已经占有的资源，当我们希望的是：

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？有如下三种方案：

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，那么它就有机会释放曾经持有的锁A,这样就破坏了不可抢占的条件了。
2. **支持超时。**如果一个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，那么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，专业那个也能破坏不可抢占条件。

这三种方案可以完全弥补synchronized的问题。到这里相信你也能理解了，这三个方案就是”重复造轮子”的主要原因，体现在API，就是Lock接口的三个方法。详情如下：

*//支持中断的API*

**void** **lockInterruptibly();**

*//支持超时的API*

**boolean** **tryLock(long** time**,** TimeUnit unit**);**

*//支持非阻塞获取锁的API*

**boolean** **tryLock();**

**如何保证可见性**[](" \l "_8" \o "Permanent link)

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finally里面释放锁。这个范例就无需多解释了，看代码。

**class** **X** **{**

**private** **final** Lock rtl **=**

**new** ReentrantLock**();**

**int** value**;**

**public** **void** **addOne()** **{**

*// 获取锁*

rtl**.**lock**();**

**try** **{**

value**+=**1**;**

**}** **finally** **{**

*// 保证锁能释放*

rtl**.**unlock**();**

**}**

**}**

**}**

但是有一点需要解释以下，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syncrhonized之所以能够保证可见性，也是因为有一条synchronized相关规则：synchronized的解锁Happens-Before于后续对这个锁的加锁。

那Java SDK里面的Lock靠什么保证可见性呢？例如在上面的代码中，线程T1对value进行了+=1的操作，那后续的线程T2能够看到value的正确结果吗？

答案是肯定的。它是利用volatile相关的Happens-Before规则。Java SDK里面的ReentrantLock，内部有一个抽象类Sync，这个类继承AQS，而在AQS这个类内部有一个成员变量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，简化的后的代码如下。也就是说，在执行value+=1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变量的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T的value+=1 Happens-Before线程T2的lock()操作。

**class** **SampleLock** **{**

**volatile** **int** state**;**

*// 加锁*

lock**()** **{**

*// 省略代码无数*

state **=** 1**;**

**}**

*// 解锁*

unlock**()** **{**

*// 省略代码无数*

state **=** 0**;**

**}**

**}**

**public** **ReentrantLock()** **{**

sync **=** **new** NonfairSync**();**

**}**

**static** **final** **class** **NonfairSync** **extends** Sync **{**

**private** **static** **final** **long** serialVersionUID **=** 7316153563782823691L**;**

**final** **void** **lock()** **{**

**if** **(**compareAndSetState**(**0**,** 1**))**

setExclusiveOwnerThread**(**Thread**.**currentThread**());**

**else**

acquire**(**1**);**

**}**

**protected** **final** **boolean** **tryAcquire(int** acquires**)** **{**

**return** nonfairTryAcquire**(**acquires**);**

**}**

**}**

*//在Reentrant类中有一个抽象类Sync，它继承了AQS（抽象队列同步器）*

**abstract** **static** **class** **Sync** **extends** AbstractQueuedSynchronizer **{}**

**什么是可重入锁**[](" \l "_9" \o "Permanent link)

所谓可重入锁，顾名思义，是指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。

**class** **X** **{**

**private** **final** Lock rtl **=**

**new** ReentrantLock**();**

**int** value**;**

**public** **int** **get()** **{**

*// 获取锁*

rtl**.**lock**();** ②

**try** **{**

**return** value**;**

**}** **finally** **{**

*// 保证锁能释放*

rtl**.**unlock**();**

**}**

**}**

**public** **void** **addOne()** **{**

*// 获取锁*

rtl**.**lock**();**

**try** **{**

value **=** 1 **+** get**();** ①

**}** **finally** **{**

*// 保证锁能释放*

rtl**.**unlock**();**

**}**

**}**

**}**

**公平锁与非公平锁**[](" \l "_10" \o "Permanent link)

在使用ReentrantLock的时候，你会发现ReentrantLock这个类有两个构造函数，一个是五参构造函数，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。

*// 无参构造函数：默认非公平锁*

**public** **ReentrantLock()** **{**

sync **=** **new** NonfairSync**();**

**}**

*// 根据公平策略参数创建锁*

**public** **ReentrantLock(boolean** fair**){**

sync **=** fair **?** **new** FairSync**()**

**:** **new** NonfairSync**();**

**}**

我们前面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，当由线程释放锁的时候，就需要从等待队列中唤醒一个等代线程。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：

永远只在更新对象的成员变量时加锁 永远只在访问可变的成员变量时加锁 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**[](#_11)

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。