# 千变万化的锁

# Lock接口

## 简介、地位、作用

### 1.作用

1. 锁是一种工具，用于控制对共享资源的访问。
2. Lock和Synchronized，这两种是最常见的锁，他们都可以达到线程安全的目的，但是在使用上和功能上又有较大的不同。
3. Lock并不是用来替代synchronized的，而是当使用synchronized不合适或不足以满足要求的时间，来提供高级功能的。
4. lock接口最常见的实现类是ReentrantLock。
5. 通常情况下，Lock只允许一个线程来访问这个共享资源，不过有的时候，一些特殊的实现也可允许并发访问，比如ReadWriteLock里面的ReadLock。

## 为什么synchronized不够用？为什么需要Lock？

1. 为什么Synchronized不够用？
   1. 效率低：锁的释放情况少、试图获得锁时不能设定超时、不能中断一个正在试图获得锁的线程
   2. 不够灵活(读写锁更灵活)：加锁和释放的时机单一，每个锁仅有单一的条件(某个对象)，可能是不够的
   3. 无法知道是否成功获取到锁

## 方法介绍

1.在Lock中声明了四个方法来获取锁

2.lock()、tryLock()、tryLock(long time,TimeUnit unit)和lockInterruptibly()

3.那么这四个方法有什么区别呢？

1.lock()

1.lock()就是最普通的获取锁。如果锁已被其他线程获取，则进行等待。

2.Lock不会像synchronized一样在异常时自动释放锁。

3.因此最佳实践是，在finally中释放锁，以保证发生异常时锁一定被释放

4.lock()方法不能被中断，这会带来很大的隐患：一旦陷入死锁，lock()就会陷入永久等待

2.tryLock()

1.tryLock()用来尝试获取锁，如果当前锁没有被其他线程占用，则获取成功。则返回true，否则返回false，代表获 取锁失败

2.相比于lock，这样的方法显然功能更强大了，我们可以根据是否能获取到锁来决定后续程序的行为。

3.该方法会立刻返回，即便在拿不到锁时不会一直在那等

3.tryLock(long time,TimeUnit unit)：超时就放弃(代码演示，代码在ThreadLock.src.lock.lock.TryLockDeadLock)。

4.lockInterruptibly()；相当于tryLock(long time,TimeUnit unit)把超时时间设置为无限。在等待死锁的过程中，线程可以被中断

5.unlock()：解锁

## 可见性保证

### 可见性

可见性是java内存模型里面最重要的一个部分，他指的是我们线程与线程之间，他们并不是随时可以看到对方最新的动态的，比如说我第一个线程修改了a =1，第二个线程可能看到还是刚才的那个值，可能刚才是0，那么他看到可能就是0。也可能是1。这就是可见性问题

### happens-before

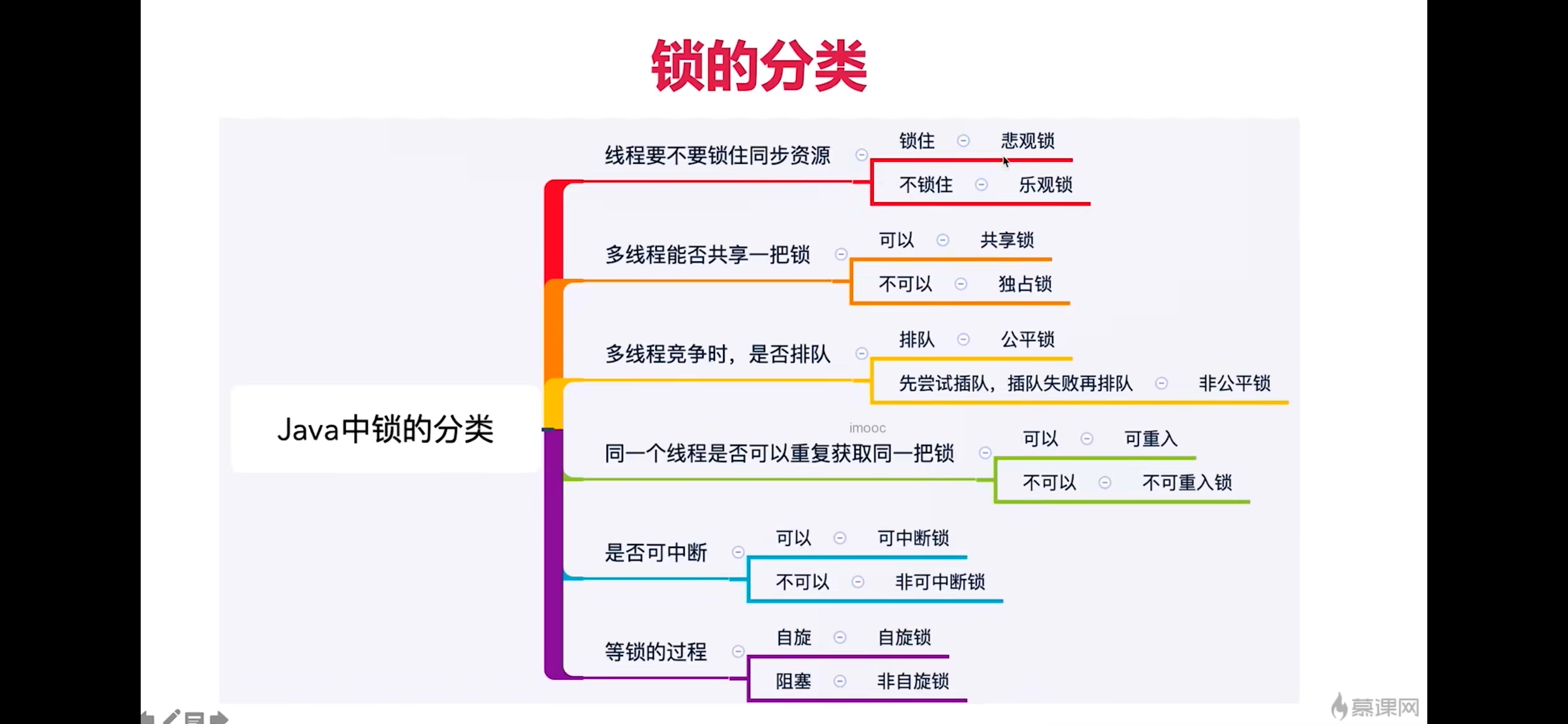
Happens-before原则指的就是说我们这件事发生了，如果其他线程一定能看到我之前所做的修改的话，就代表她们拥有happens-before。

### 3.Lock的加解锁和synchronized有同样的内存语义，也就是说，下一个线程加锁后可以看到所有前一个线程解锁前发生的所有操作

# 锁的分类

## 概述

1. 这些分类，是从各种不同角度出发去看的
2. 这些分类并不是互斥的，也就是多个类型可以并存：有可能一个锁，同时属于两种类型
3. 比如ReentrantLock既是互斥锁，又是可重入锁。
4. 好比是你是一个男人，同时又是一个军人，这是不互斥的。



# 乐观锁和悲观锁

## 为什么会诞生非互斥同步锁(乐观锁) --- 互斥同步锁的劣势(悲观锁)

### 互斥同步锁的劣势

* 1. 阻塞和唤醒所带来的性能劣势

因为悲观锁锁住之后，他就是独占的，其他线程如果还想获得相同的资源，那就必须等待，这带来的最大问题就是性能问题。这个性能问题主要就是发生在线程的阻塞和唤醒阶段。一些列操作会带来服务器性能的损耗。

* 1. 永久阻塞

如果持有锁的线程被永久阻塞，比如遇到了无限循环、死锁等活跃性问题，那么等待该线程释放锁的那几个悲催的线程，将永远也得不到执行。

* 1. 优先级反转

如果我们被阻塞的线程优先级比较高，而持有锁的线程优先级比较低，那么这会导致优先级反转，由于我们本身给线程设置的优先级，想让优先级搞得那个线程多运行，而让少的线程少运行，可是一旦我们这个优先级低的线程他拿到这个锁之后，假设说他不释放，或者说它释放的比较慢，那么这段时间，即便你另外的线程优先级很高，即便你能执行，你也没有用的，因为你必须要等待我们的锁释放，你才能执行，这就是导致虽然你的优先级实际很高，但是你会降到比我还低这种情况，导致优先级反转

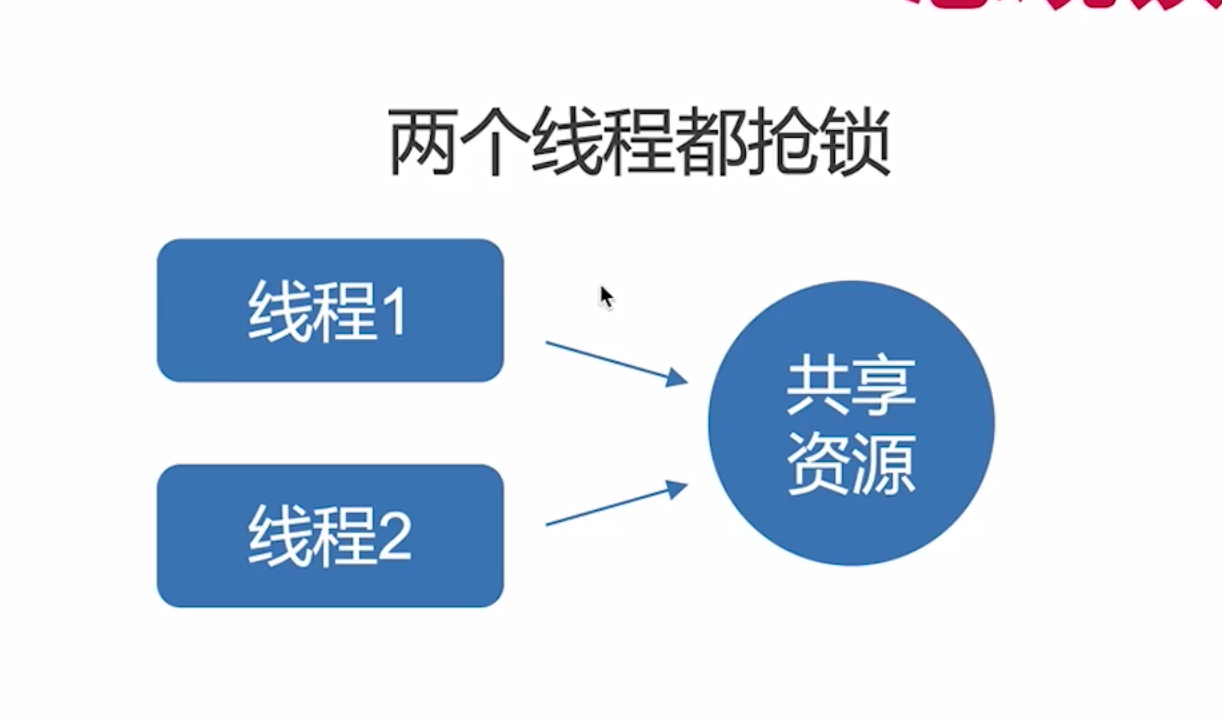
## 什么是乐观锁和悲观锁

### 人的性格

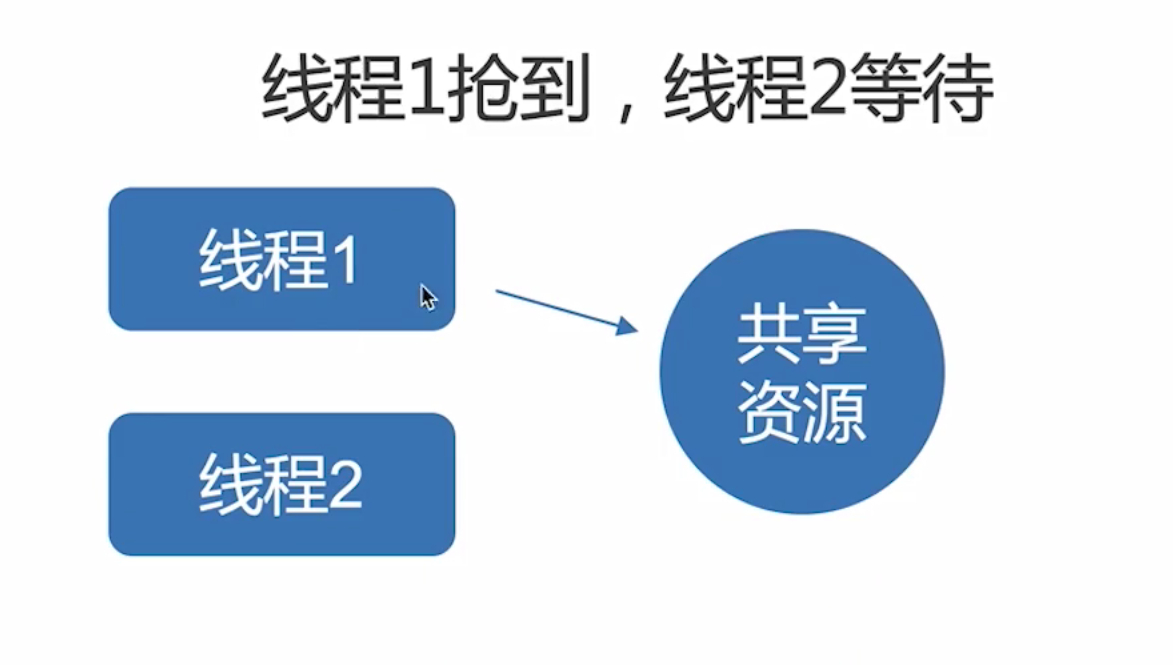
从人的性格来讲也是分为乐观和悲观的

### 从是否锁住资源的角度来分类

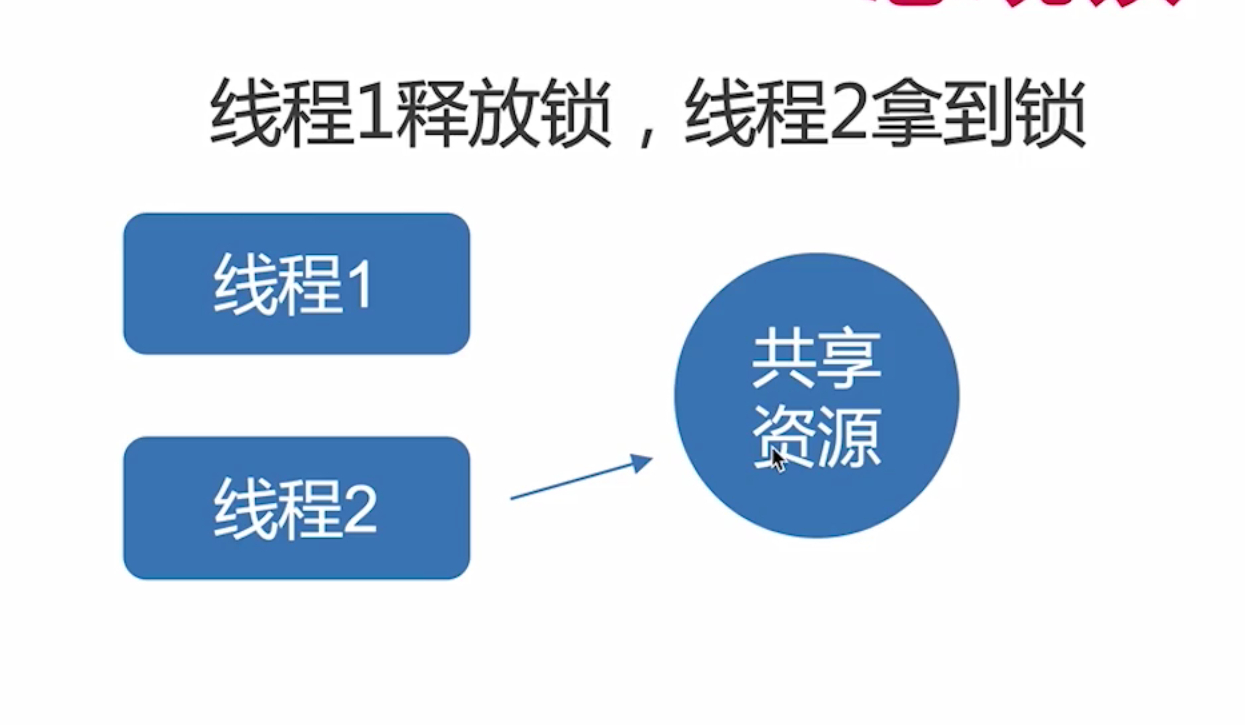
1. 悲观锁：如果我不锁住这个资源，别人就会来抢，就会造成数据结果错误，所以每次悲观锁为了确保结果的正确性，就会在每次获取并修改数据时，把数据锁住，让别人无法访问该数据，这样就可以确保数据的内容万无一失。
2. java中最典型的悲观锁就是synchronized和Lock相关类
3. 悲观锁在执行的时候是一个什么样的流程？



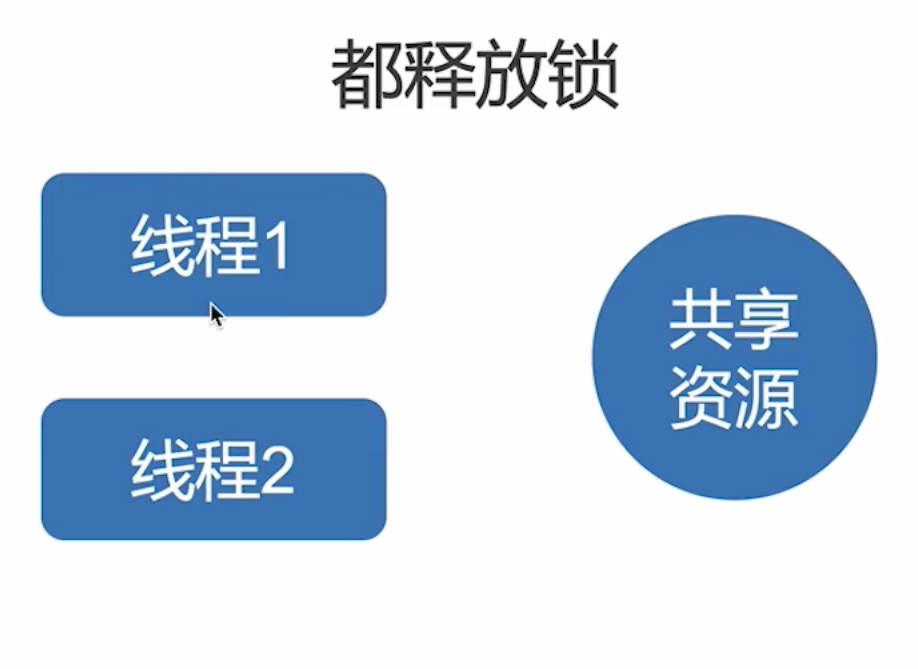
首先假设两个线程都去抢这把锁，只有一个想爱你成能抢到。



我们假设线程1抢到，那么线程2就开始等待。他就开始反复的等待

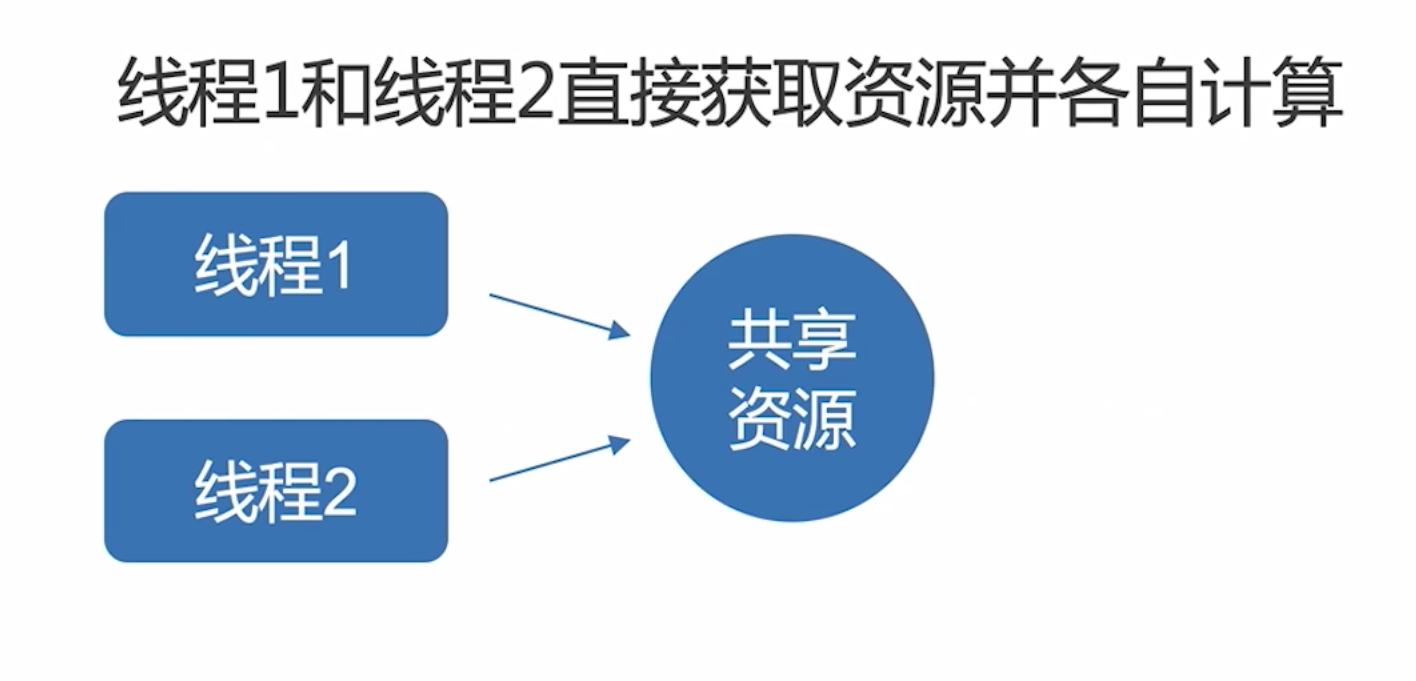


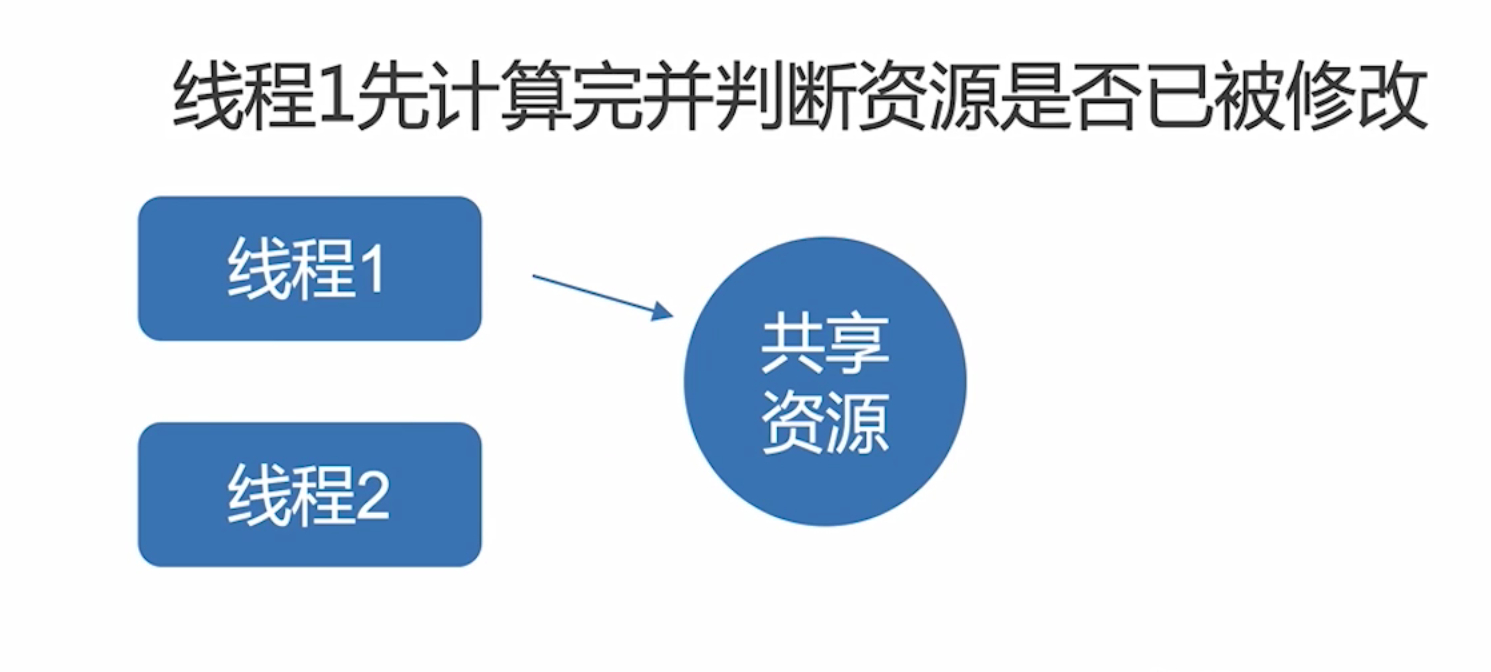
直到线程1释放了之后线程2才能拿到，然后呢线程2拿到之后去执行完自己的逻辑。

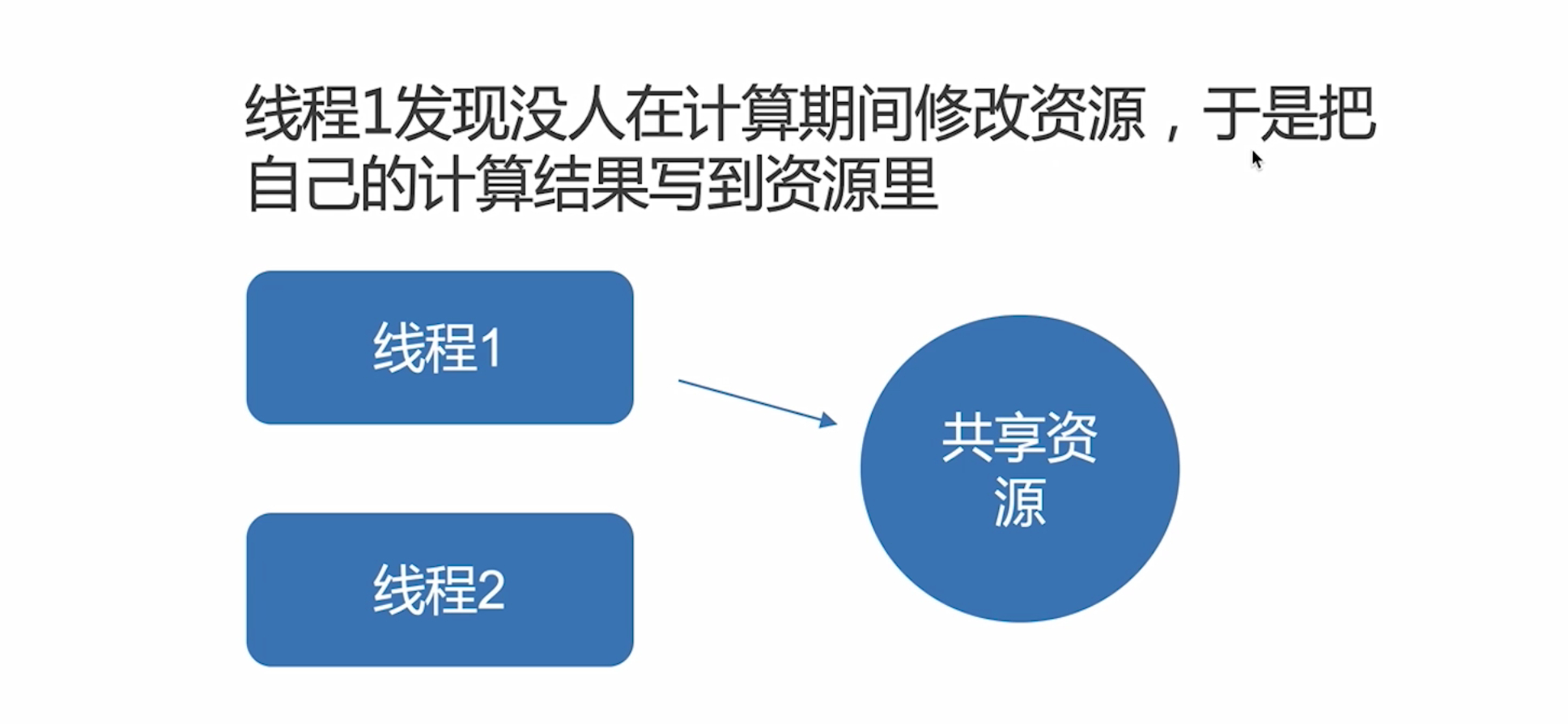


最终把他给释放掉。

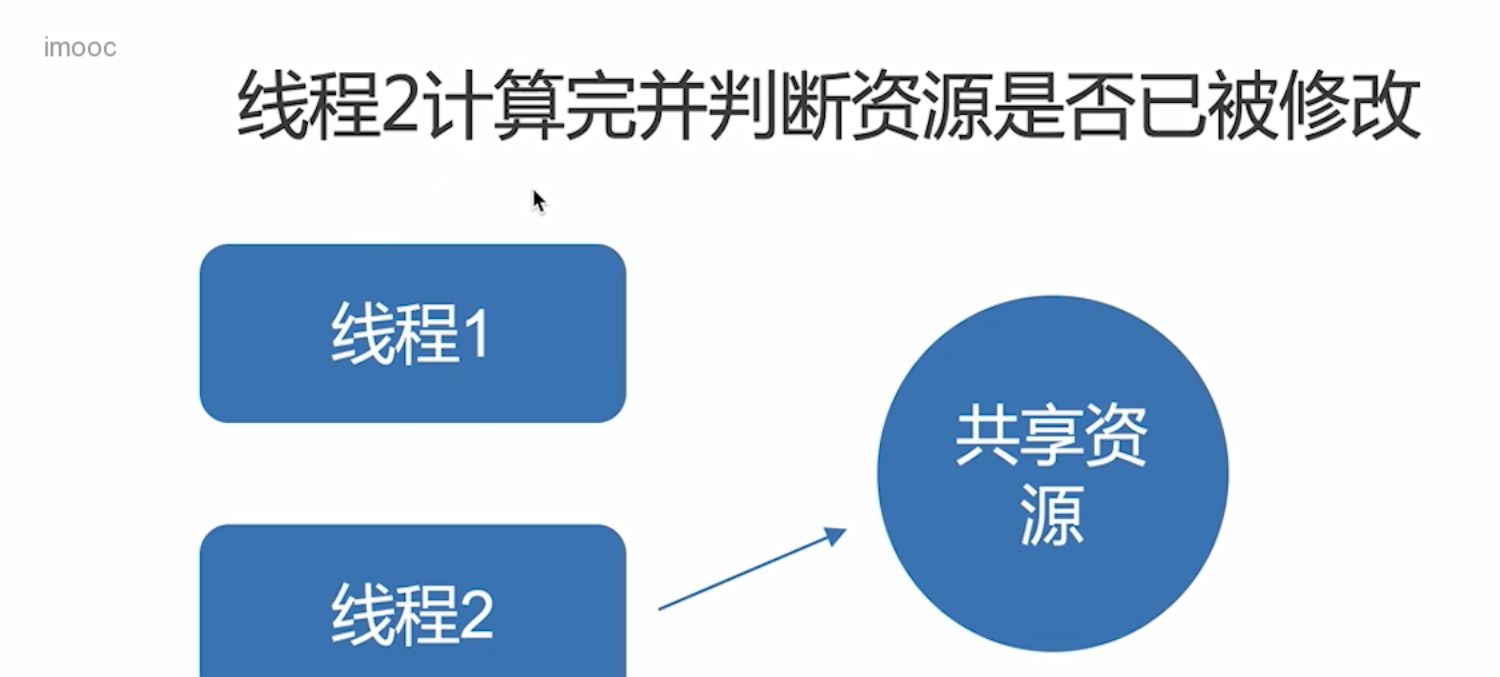
1. 乐观锁；认为自己在处理操作的时候不会有其它线程来干扰，所以并不会锁住被操作对象，
2. 乐观锁：在更新的时候，去对比在我修改的期间数据有没有被其他人改变过：如果没被改变过，就说明真的是只有我自己在操作，那我就正常去修改数据
3. 乐观锁：如果数据和我一开始拿到的不一样了，说明其他人在这段时间诶修改过数据，那我就不能继续刚才的更新数据过程了，我会选择放弃、报错、重试等策略。。
4. 乐观锁：乐观锁的实现一般都是利用CAS算法来实现的，CAS的核心思想就是我可以在一个原子操作内，把你这个数据对比并且交换。那么在此期间是不会有人能打断我的。
5. 乐观锁是是如何操作一个数据的？



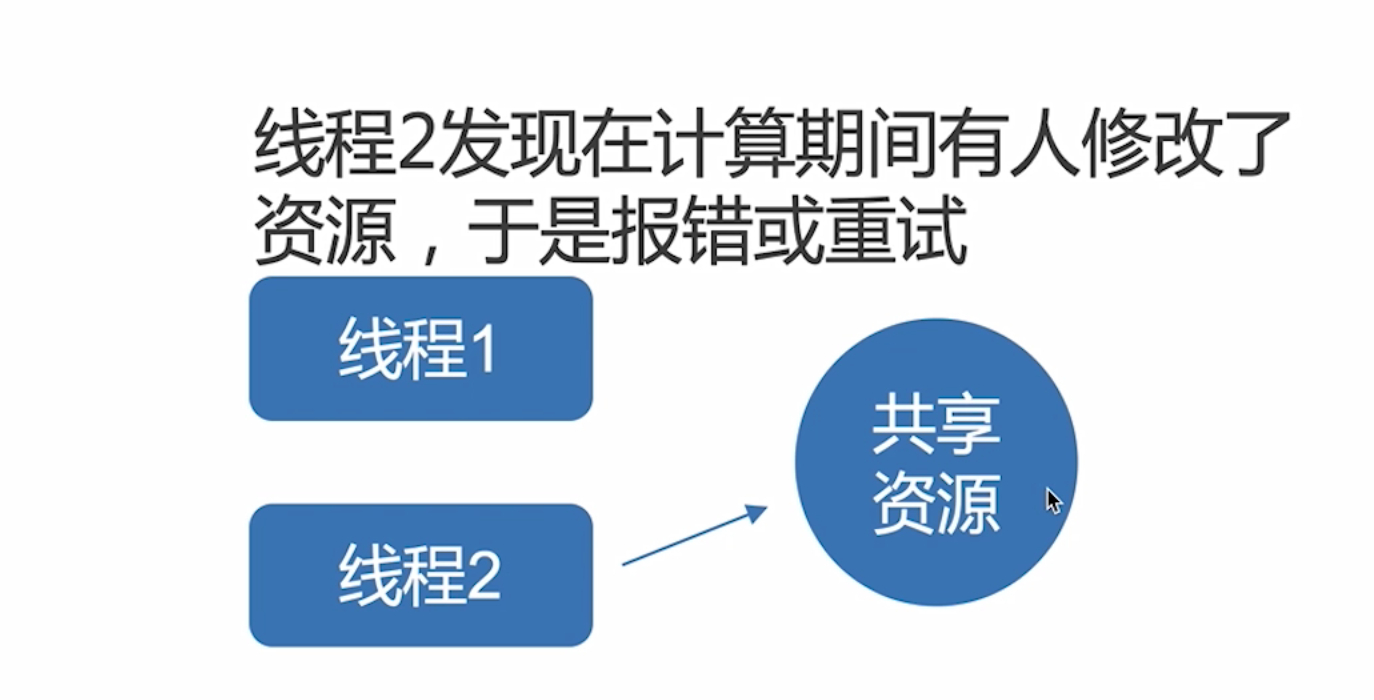
假设我们线程1和线程2相同时获得到这个资源，然后呢他们没有人回去锁它  
他们实际上是并行开始计算了，假设我们线程1先计算完，这个时候在计算完之后他不会立刻的会写进去，因为他计算完了他会去判断一下在此期间有没有人把它修改过。



如果线程1发现在此期间没有被人修改过，他就把结果去写进去



那么这个时候线程2他也就算完了，他也会去判断共享资源有没有被人修改过，实际上已经是被人修改过了，因为刚才已经被线程1修改过了，于是呢他就不把自己的结果给放进去了



线程2刚才计算的结果可以丢弃，或者重试或者报错，都可以，那么现成这个修改是失败的。

## 典型例子和应用

1. 典型例子

\* 悲观锁：synchronized和Lock接口

\* 乐观锁的典型例子就是原子类、并发容器等(原子类，并发容器他们的底层实现很多都用到了乐观锁的实现)

1. 代码演示
2. Git：Git就是乐观锁的典型例子，当我们往远程仓库push代码的时候，git就会检查远端仓库的版本是不是领先于我们现在的版本，如果远程仓库的版本号和本地的不一样，就表示有其他人修改了远端代码了，我们的这次提交就失败；如果远端和本地版本号一致，我们就可以顺利提交版本到远程仓库。
3. Git不适合用悲观锁，否则公司倒闭

就是说我在写代码的过程中，你是不能提交的，因为我把整个的远端仓库都给锁住了，这段时间内只能我来提交，那么可能就会造成今天我一整天都在写代码，一致锁住，那么其他人就没有办法去协作，其他人写的也都发布不上去，那么效率就会非常低下。

1. 数据库
2. 、select for update就是悲观锁。

当我们用了select for update语句执行数据库，那么他就会把库给锁住，然后呢你再去更新，更新的期间其他人不能修改。

1. 、用version控制数据库就是乐观锁

\* 添加一个字段lock\_version，这个是专门用来记录版本号的

\* 先查询这个更新语句的version：select \* from table，先把版本号给查出来

\* 然后update set num = 2,version = version + 1 where version = 1 and id = 5

Where version = 1(其实上面的语句查询出来的版本号)其实就是在检查在更新过程中其他人有没有修改过该语句。

如果version被更新了等于2，不一样就会更新出错，这就是乐观锁的原理。

## 开销对比

1. 悲观锁的原始开销要高于乐观锁，但是特点是一劳永逸，临界区持锁时间就算越来越差，也不会对互斥锁的开销造成影响。
2. 相反，虽然乐观锁一开始的开销比悲观锁小，但是如果自旋时间很长或者不停重试，那么消耗的资源也会越来越多。

## 两种锁各自的使用场景

1. 悲观锁：是和并发写入多的情况，适用于临界区持锁时间比较长的情况，悲观锁可以避免大量的无用自旋等消耗，典型情况：

\* 临界区有IO操作：持有锁时间会比较长

\* 临界区代码复杂或者循环量大

\* 临界区竞争非常激烈

2、乐观锁：适合并发写入，小大部分是读取的场景，不加锁的能让读取性能大大提高。

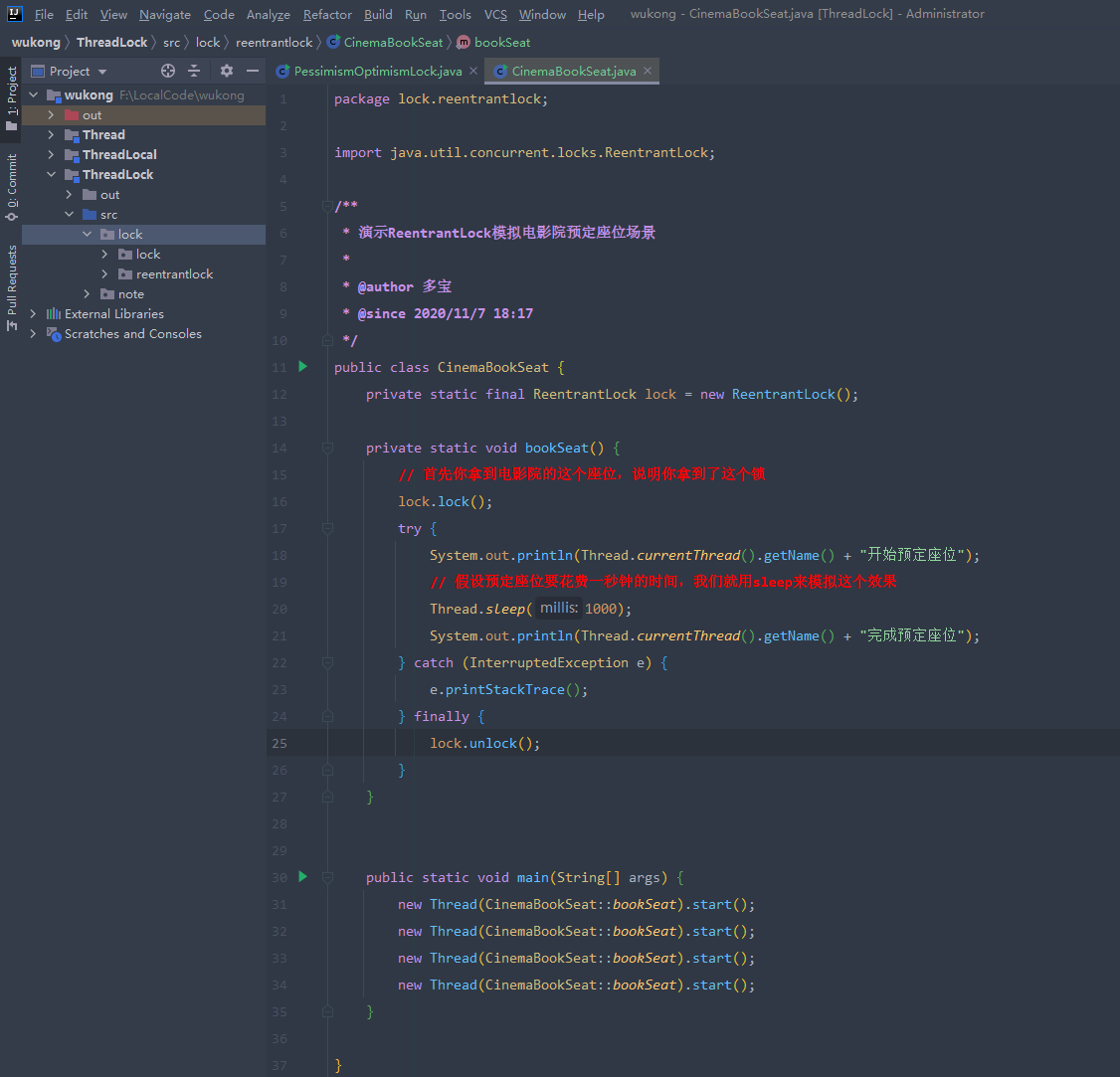
# 可重入锁和非可重入锁，以ReentrantLock为例(重点)

## 使用案例

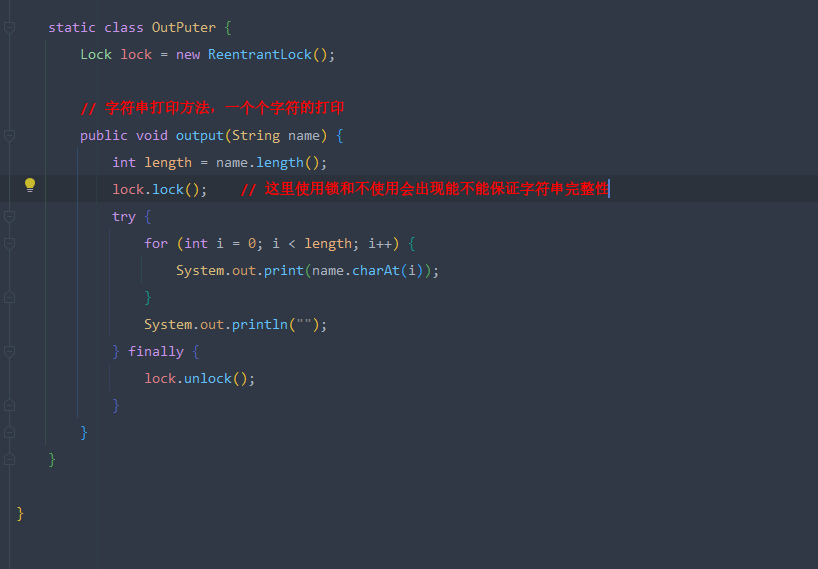
### 预定电影院座位

来到电影院之前需要预定票和座位，因为我们在预定座位的时间很可能我们的座位是并发预定的，选完座位之后APP会显示该座位为你锁定三分钟，请您付款。其实这个过程就是上锁了。

### 代码演示一下这个类似于电影院抢座位的过程。(ReentrantLock的用法)

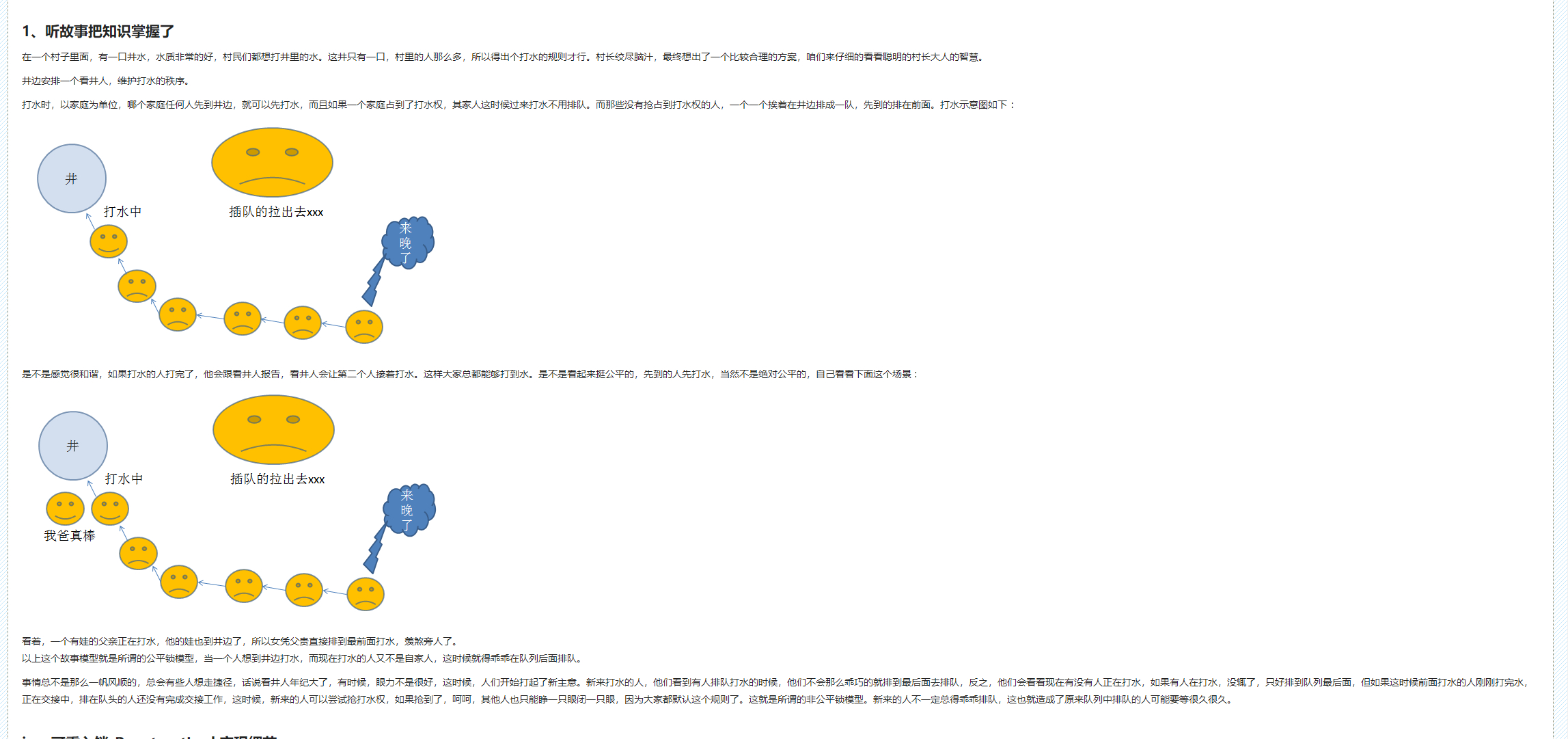


### 打印字符串



## 可重入性质

### 什么是可重入？



### 好处？

1. 、避免死锁

假设我们两个方法都被synchronized修饰了，或者是被同一个锁给锁住了，这个时候线程A运行第一个方法，她拿到锁了，可是这个时候，他如果是想执行第二个方法，这个方法被同样的锁锁住了。

# 公平锁和非公平锁

# 共享锁和排它锁：以ReentrantReadWriteLock读写锁为例(重点)

# 自旋锁和阻塞锁

# 可中断锁：顾名思义，就是可以响应中断的锁

# 9.锁优化