# 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

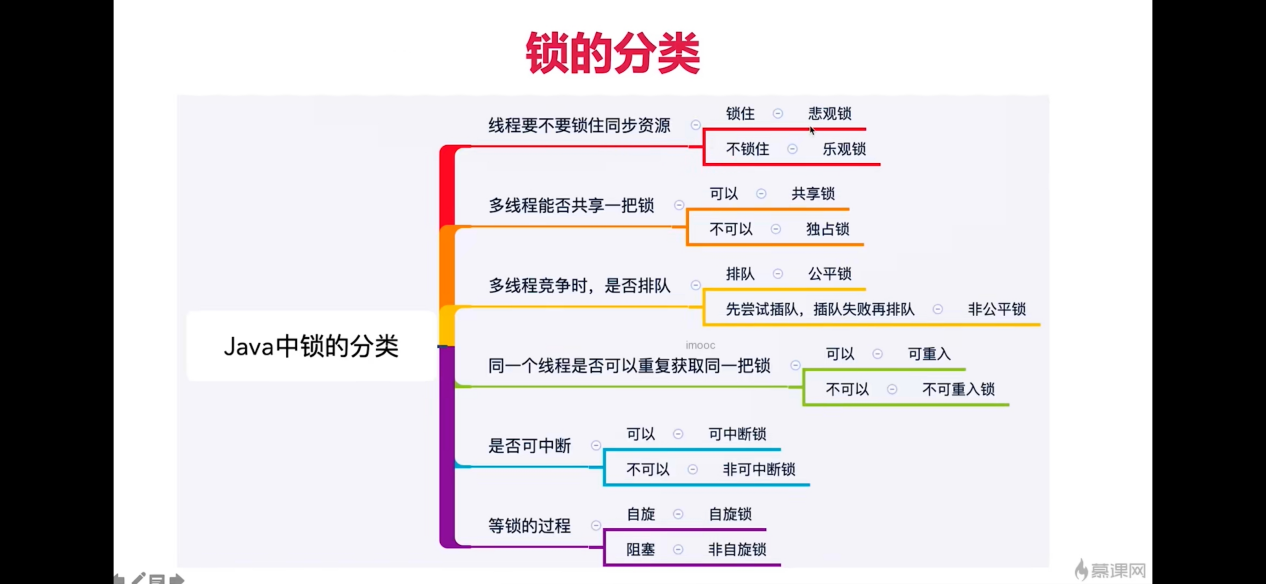
Happens-before原则指的就是说我们这件事发生了，如果其他线程一定能看到我之前所做的修改的话，就代表她们拥有happens-before。

### 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

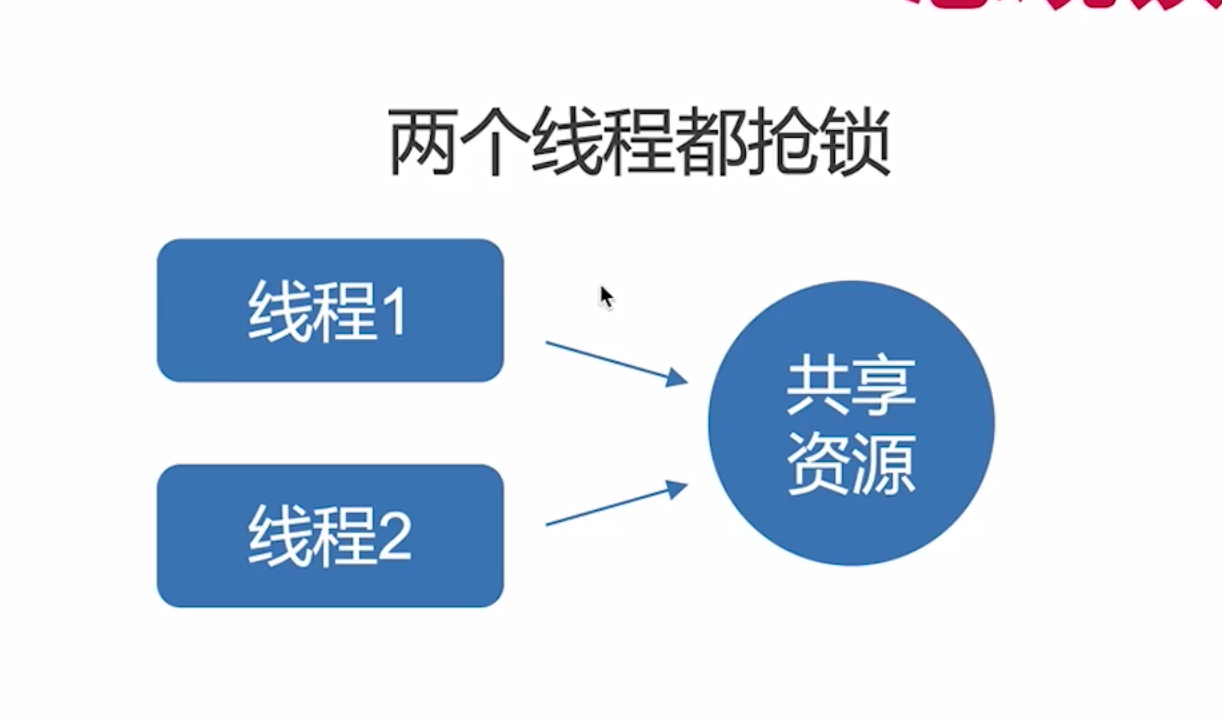
## 什么是乐观锁和悲观锁

### 人的性格

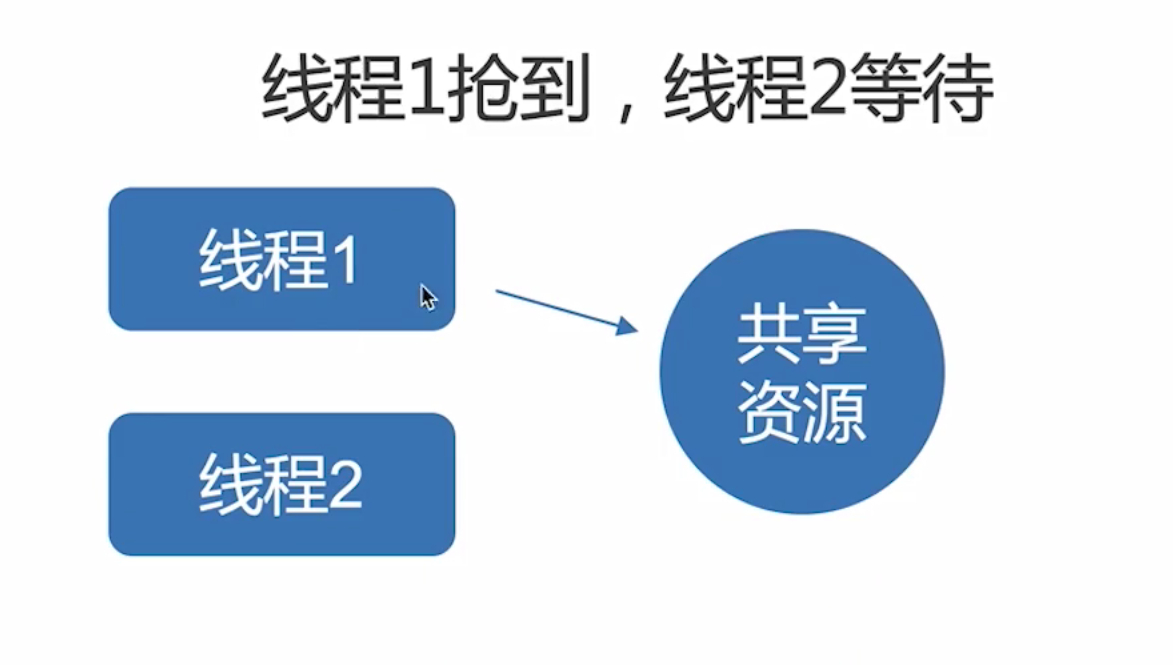
从人的性格来讲也是分为乐观和悲观的

### 从是否锁住资源的角度来分类

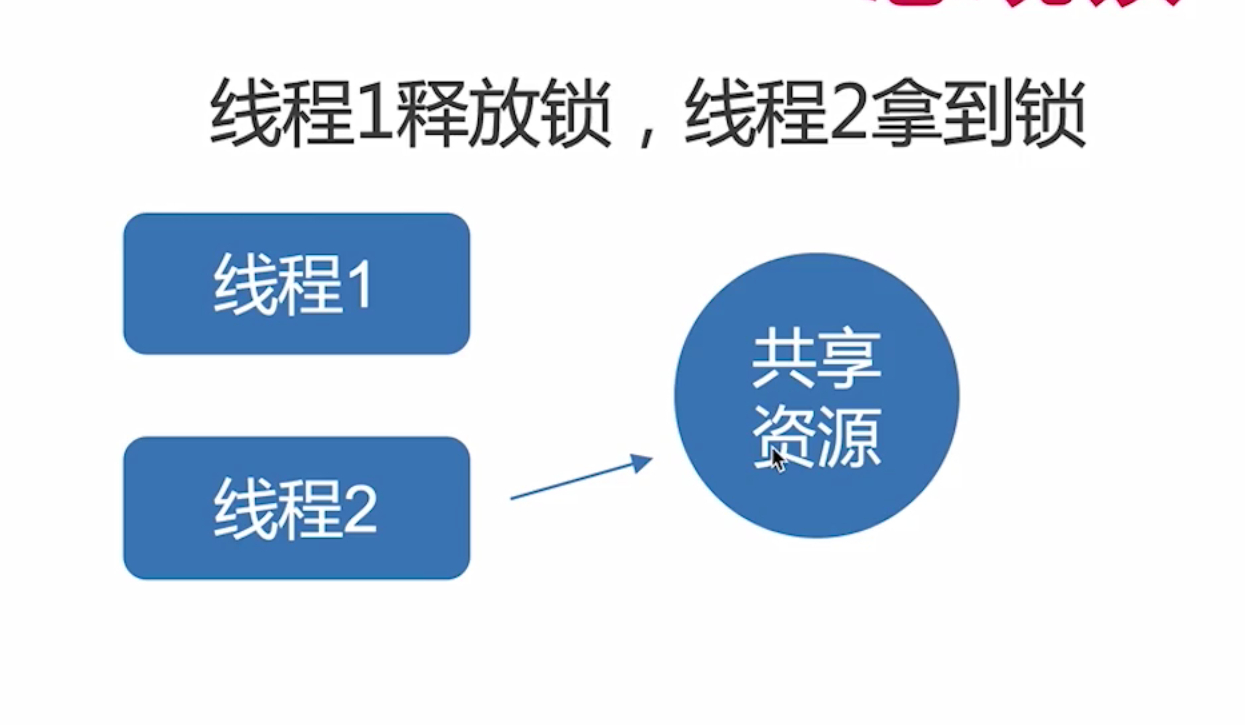
1. 悲观锁：如果我不锁住这个资源，别人就会来抢，就会造成数据结果错误，所以每次悲观锁为了确保结果的正确性，就会在每次获取并修改数据时，把数据锁住，让别人无法访问该数据，这样就可以确保数据的内容万无一失。
2. java中最典型的悲观锁就是synchronized和Lock相关类
3. 悲观锁在执行的时候是一个什么样的流程？



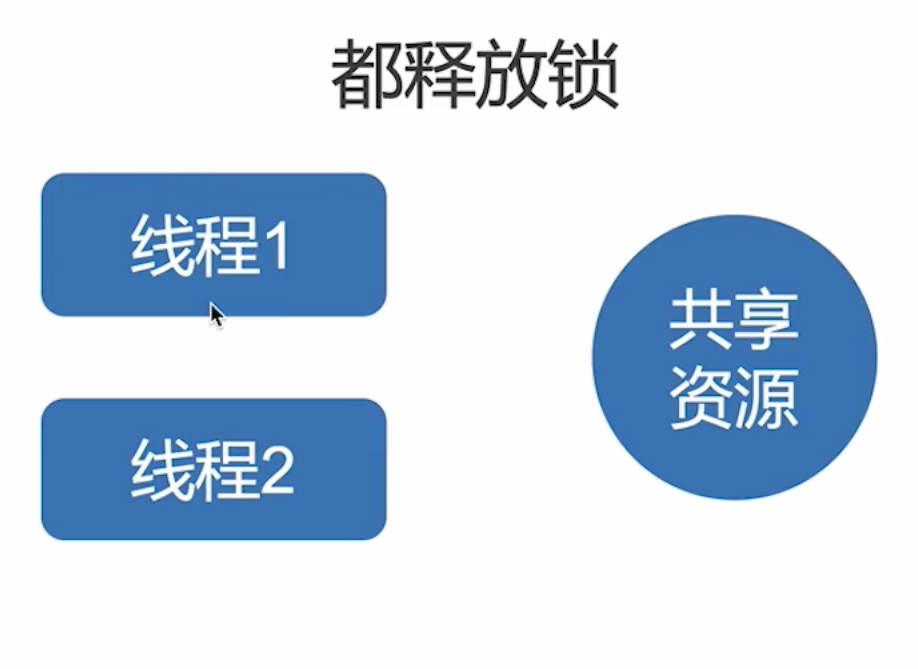
首先假设两个线程都去抢这把锁，只有一个想爱你成能抢到。



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

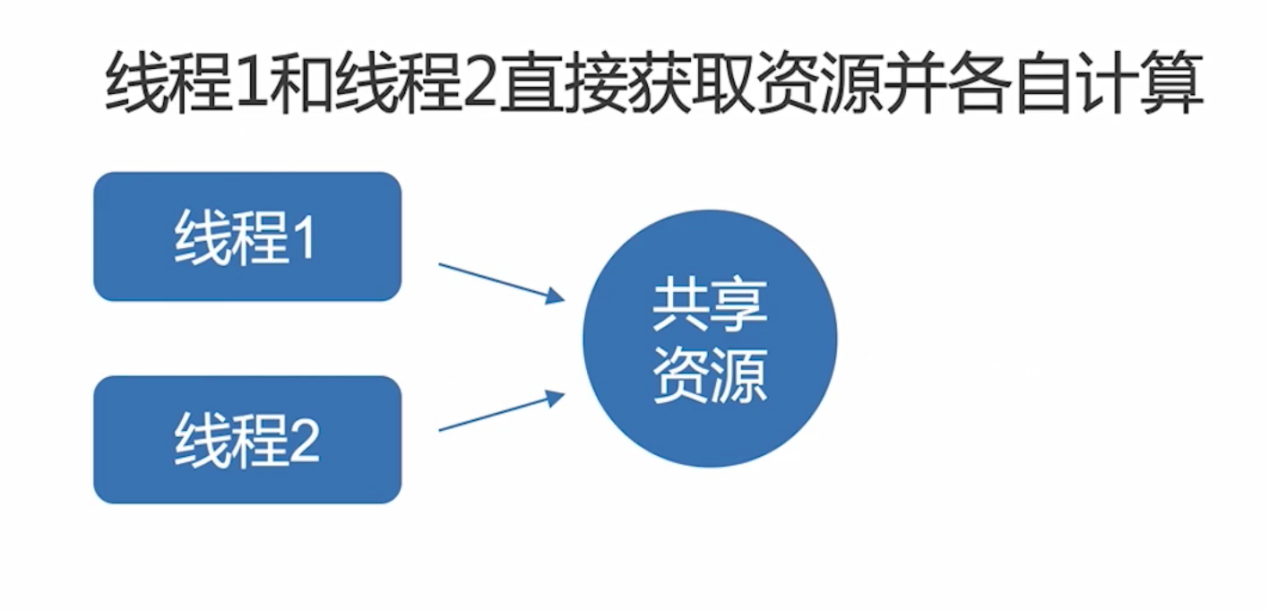


直到线程1释放了之后线程2才能拿到，然后呢线程2拿到之后去执行完自己的逻辑。

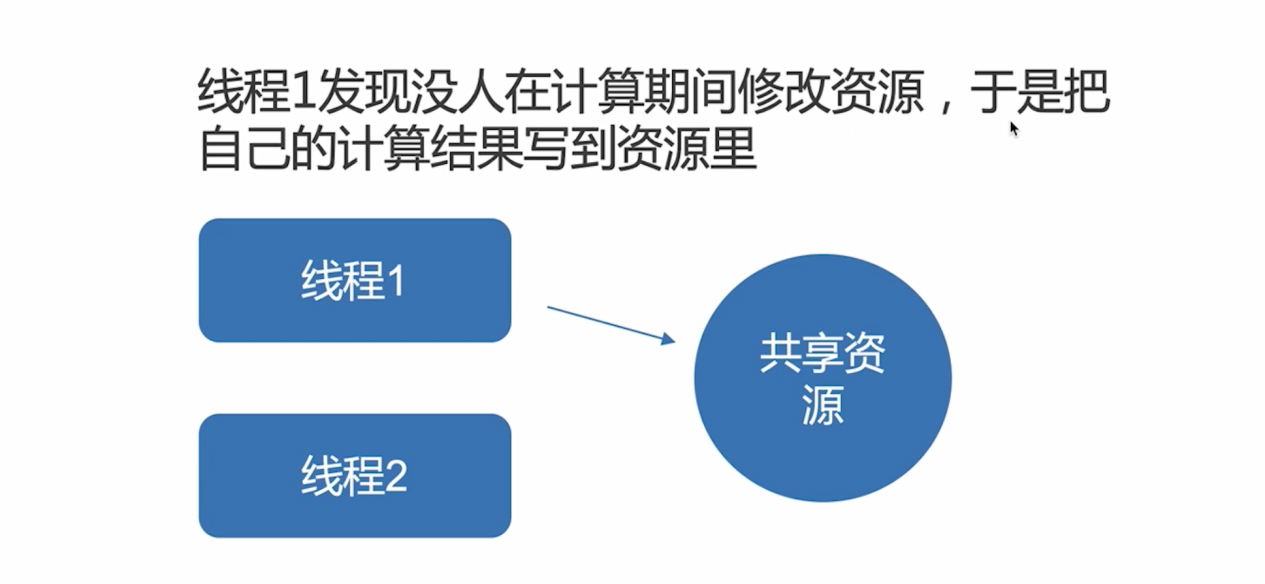


最终把他给释放掉。

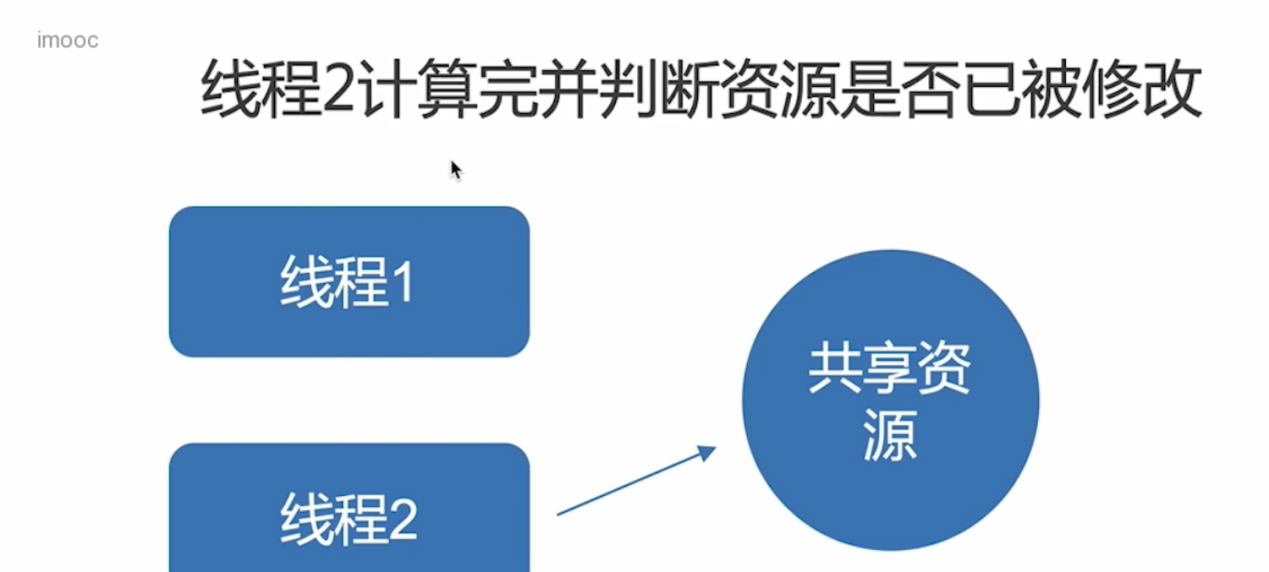
1. 乐观锁；认为自己在处理操作的时候不会有其它线程来干扰，所以并不会锁住被操作对象，
2. 乐观锁：在更新的时候，去对比在我修改的期间数据有没有被其他人改变过：如果没被改变过，就说明真的是只有我自己在操作，那我就正常去修改数据
3. 乐观锁：如果数据和我一开始拿到的不一样了，说明其他人在这段时间诶修改过数据，那我就不能继续刚才的更新数据过程了，我会选择放弃、报错、重试等策略。。
4. 乐观锁：乐观锁的实现一般都是利用CAS算法来实现的，CAS的核心思想就是我可以在一个原子操作内，把你这个数据对比并且交换。那么在此期间是不会有人能打断我的。
5. 乐观锁是是如何操作一个数据的？



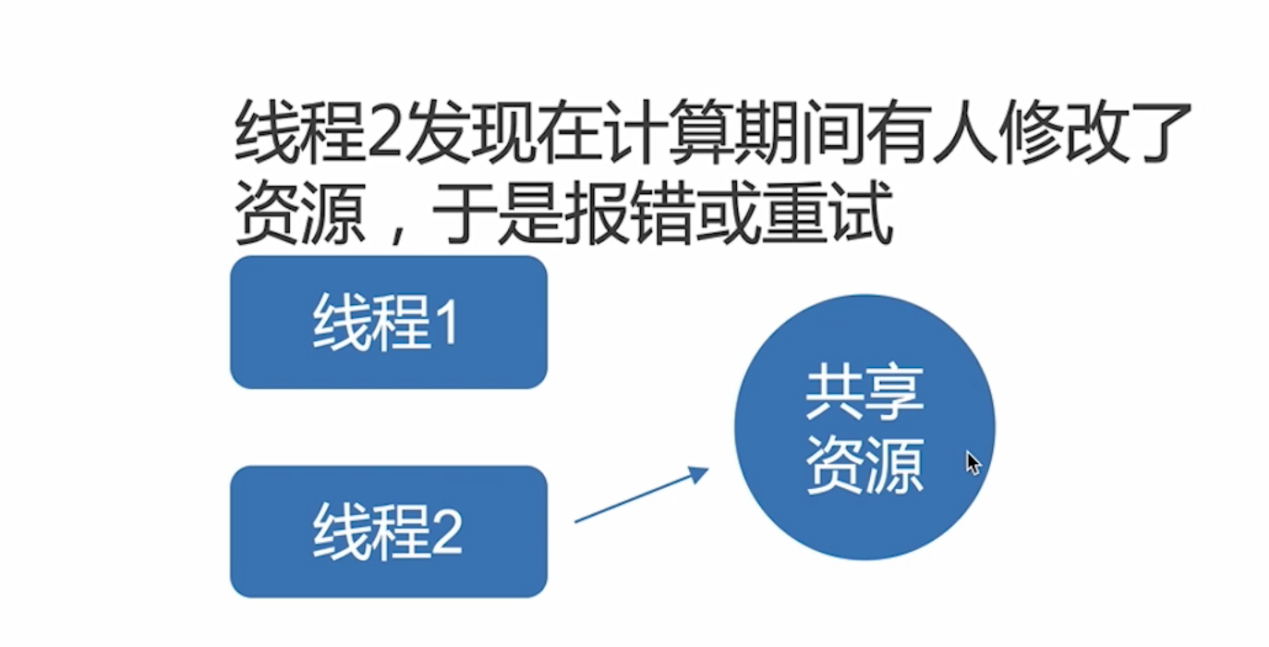
假设我们线程1和线程2相同时获得到这个资源，然后呢他们没有人回去锁它  
他们实际上是并行开始计算了，假设我们线程1先计算完，这个时候在计算完之后他不会立刻的会写进去，因为他计算完了他会去判断一下在此期间有没有人把它修改过。



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。

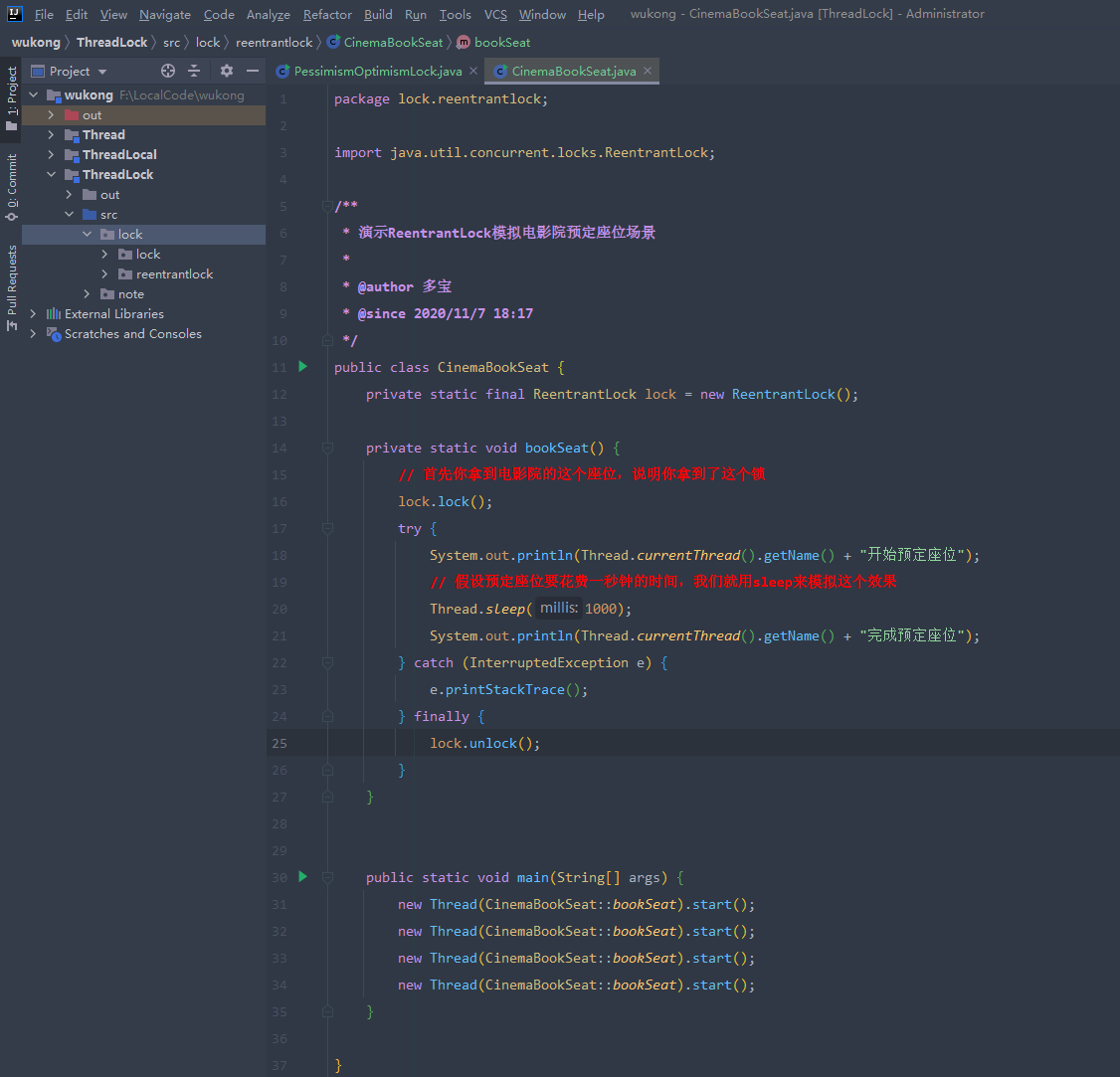
# 可重入锁和非可重入锁，以ReentrantLock为例(重点)

## 使用案例

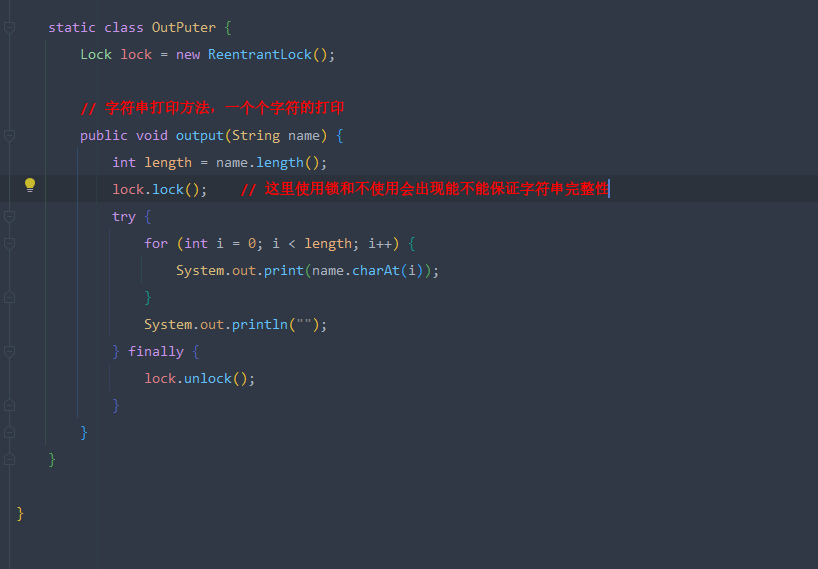
### 预定电影院座位

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)

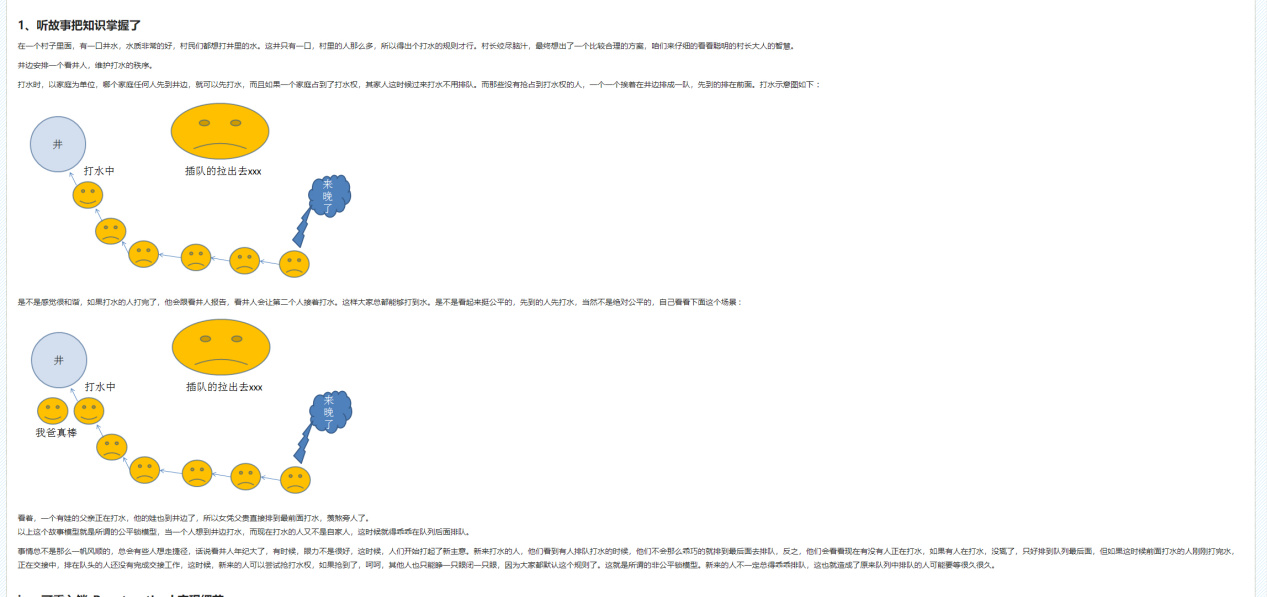


### 打印字符串



## 可重入性质

### 什么是可重入？



### 好处？

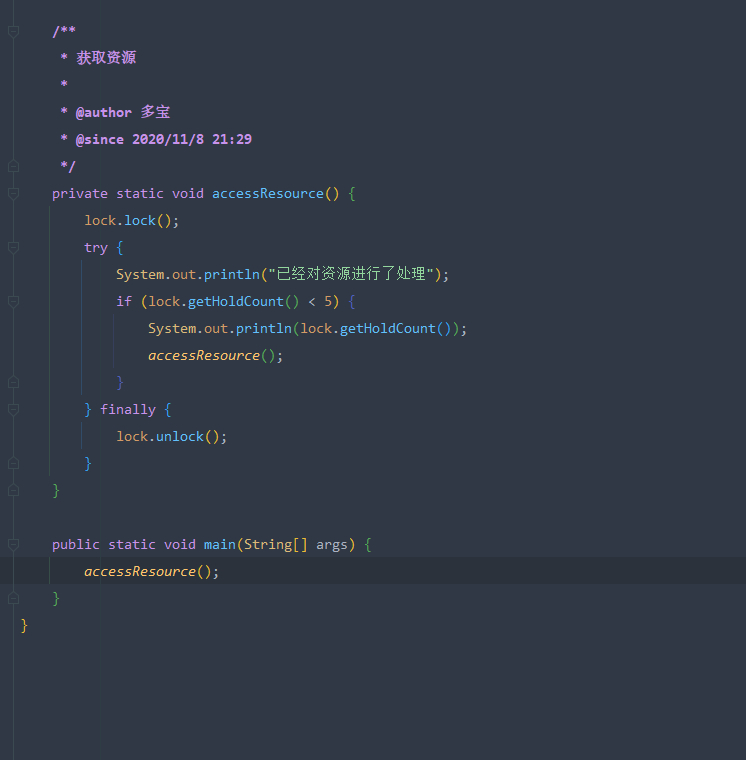
1. 、避免死锁

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，假设我们不具备可重入性，那么这个时候再去获取那把锁，你是获取不到的，因为这把锁你必须要先释放才能再次获取，你如果不具备可重入性的话，这个时候就发生死锁了，相当于我手上拿着这把锁，我还想获取这把锁，你是获取不到的，有了可重入性之后呢，我们就不会发生这种现象了，可以避免死锁的发生，

1. 、提升封装性

这样一来我们的加锁、解锁就没有那么麻烦，避免一次一次的加锁又解锁，降低了我们编程的难度

### 代码演示



### 已获得的锁得数量

Lock.getHoldCount()

### 源码对比：可重入锁ReentrantLock以及非可重入锁ThreadPoolExecutor的Worker类



实际上ReentrantLock他背后上锁的过程的核心是AQS，AQS是一个框架，可以说是一种工具，AQS是我们很多并发工具的一个核心，

### ReentrantLock的其他方法介绍。

1. isHeldByCurrentThread可以看出锁是否被当前线程持有
2. getQueueLength可以返回当前正在等待这把锁的队列有多长，一般这两个方法是开发和调试时候使用，上线后用到的不多

### 使用案例和回顾

1. 去电影院看电影的时候想去抢锁。
2. 打印字符串的方式看下锁的使用。

会发现如果我们不使用锁，多个线程之间确实是会顺序错乱。

1. 可重入性质(重点)

如果你多辆车其中一个摇到号，就可以为你其他的车同时上牌照，这就是可重入的性质。

可重入好处：

\* 避免死锁

\* 提升封装性

# 公平锁和非公平锁

## 什么是公平和非公平？

### 公平是指按照线程请求的顺序，来分配锁；非公平指的是，不完全按照请求的顺序，在一定情况下，可以插队。

### 注意：非公平也同样不提倡”插队”行为，这里的非公平，指的是”在合适的时机”插队，而不是盲目插队。

### 什么是合适的实际呢？

### 火车票被插队的例子

假设以前还没有12306网上APP的时候，大家还是会说排队去火车站买，那个时候是这样的，其实12306也没几年呢，那么我们那个时候买火车票，尤其是春运的时候，那个可是很难抢票的，这个时候一个插队那直接是影响到我能不能买到票，所以是非常关键的。这个时候，假设有这么一个情况，我们是排在队伍的第二位，然后再我们前面呢，有一个人，它是先于我们排队，所以它自然是先于我们买票。然后让买完了票，走了，下一个本来是我，可是因为经过了彻夜的排队，其实那个时候我的脑袋还是我那个嗡嗡作响，还不是特别清醒，确实该轮到我了，可是呢这个时候呢，我也没有一下子缓过神来，在那愣住了，这个时候第一个人本来已经走了，他突然回来，又问了一下乘务员，说，我就问一句，很快的，请问那个火车几点发车？就这样问一句，那么你说这个叫不叫插队？这个实际上就是完全模拟了我们在线程中插队的情况。

我们来想一下，这种情况下，主要是体现了什么呢？体现了第一：由于我从哪个呆萌的状态到缓过神来去执行，这就对应到我们线程从阻塞状态 -- 被唤醒，这个是需要一个长时间切换的。而刚才那个人，它呢，是很清醒的。他直接来问，问好之后，他就走，其实并没有影响到我们什么东西，因为那个时候，就算他不来问，我也是脑子不清楚，也没办法买票，这个也就是反映了我们这边非公平的意思。

## 为什么要有非公平锁？

### 看到这里，小伙伴们肯定会很生气了，联想到自己买票被插队的情况，更是怒火中烧，凭什么默认策略是非公平，我难道前面的线程白白排队了吗？JAVA的设计者是不是没有素质？

1. 实际情况并不是这样的，Java设计者这样设计的目的，是为了提高效率。
2. 避免了唤醒带来的空档期

我们为什么不希望锁都是公平的呢？毕竟公平是一种很好的现象，不公平是不好的，但是啊，如果我们始终是公平的话，那么他在把那个已经挂起的线程恢复过来的这段时间，是有开销的，而这段时间如果你是公平的话，你要求必须排队的话，那么这段时间谁都拿不到锁，谁都没办法处理，但是我们假设啊，我们是可以允许非公平的。

我们假设我们现在有三个线程，第一个线程A持有这把锁，线程B请求这把锁，由于这个锁已经A持有了，那么B自然而然要去休息，假设A这个时候释放了，那么B呢就要被唤醒，就要去拿到这把锁，假设与此同时突然C来请求这个锁，那么由于C这个线程啊，他本身由于是一直处于唤醒状态，他也没有休息，它是可以立刻执行的，那么C很有可能在B被完全唤醒之前就已经获得了并且是使用完了并且又释放掉了这把锁，这就形成了一种双赢的局面。

为什么叫双赢呢？第一个，C肯定是赢了，C没有排队，她拿到锁了，并且是用完了，释放了。

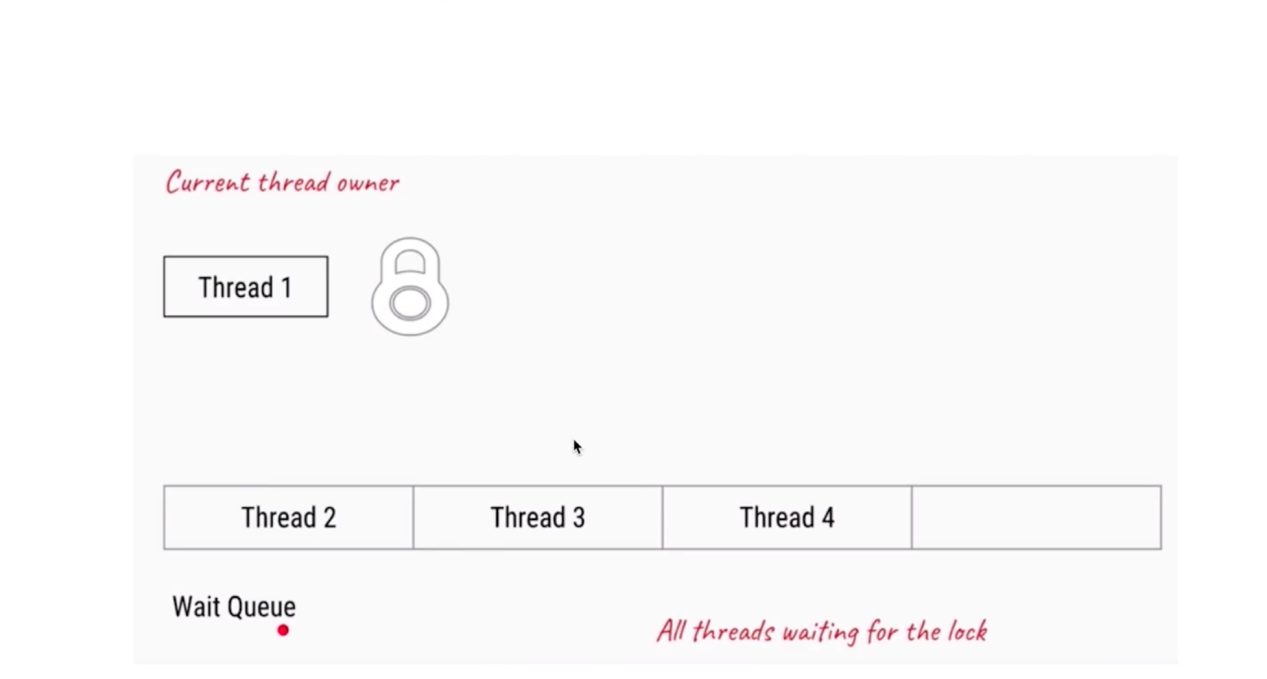
第二个其实B也没有输，为什么呢？B本身这段时间，他知道说A已经释放了，然后呢唤醒B，唤醒B的这个过程是耗时的，那么这段时间呢？本身这段时间，我既然耗时，我也拿不到锁，不如就让给别人，所以说，对于B而言，他拿到锁的时间并没有推迟，所以这是一种双赢的局面，这种插队呢，是可以带来吞吐量的提升的。这就是为什么会出现非公平锁的原因，主要就是因为啊，在我们大多数的情况下，由于这个唤醒的过程，这个开销，其实是比较大的，子啊这个期间，他为了增加我们的吞吐量，来给这个期间呢来给利用出去，这就是我们非公平设计的最根本的原因。

说到唤醒，举一个例子，以前呢女神，他换男朋友，换男朋友有空档期，在这个空档期的过程中，备胎来补上，为什么备胎补上啊？因为备胎，随叫随到，没有很高的唤醒成本，而男朋友呢，就像是里面切换的成本一样，也很慎重，阿么唤醒呢也是有很大成本的，所以这样一来就提高了效率。

## 公平的情况(以ReentrantLock为例)

### 如果在创建ReentrantLock对象时，参数填写为true，那么这就是个公平锁：

### 假设线程1234是按照顺序调用lock的



首先是第一个线程拿到，那么234线程就开始在队列里面排队，等第一个执行完事之后，他就释放了，释放完之后，第二个线程拿到了锁。



为什么锁给到了第二个，因为第二个是最先进来排队的，那它等待的时间最长，于是呢就给了第二个，剩下的呢就毅然在队列里面进行排队，之后就开始执行线程3、4。

## 不公平的情况(以ReentrantLock为例)

### 对接上面不公平的情况，现在演示不公平的情况

1. 如果在线程1释放锁的时候，线程5恰好去执行lock，
2. 由于ReentrantLock发现此时并没有线程持有lock这把锁(线程2还没来得及获得到，因为获取需要时间)
3. 线程5可以插队，直接拿到这把锁，也就是ReentrantLock默认的公平策略，也就是”不公平”。



图的意思解释为，现在线程2还在排队，1一旦解锁，与此同时5来加锁，5直接就可以拿到这把锁。

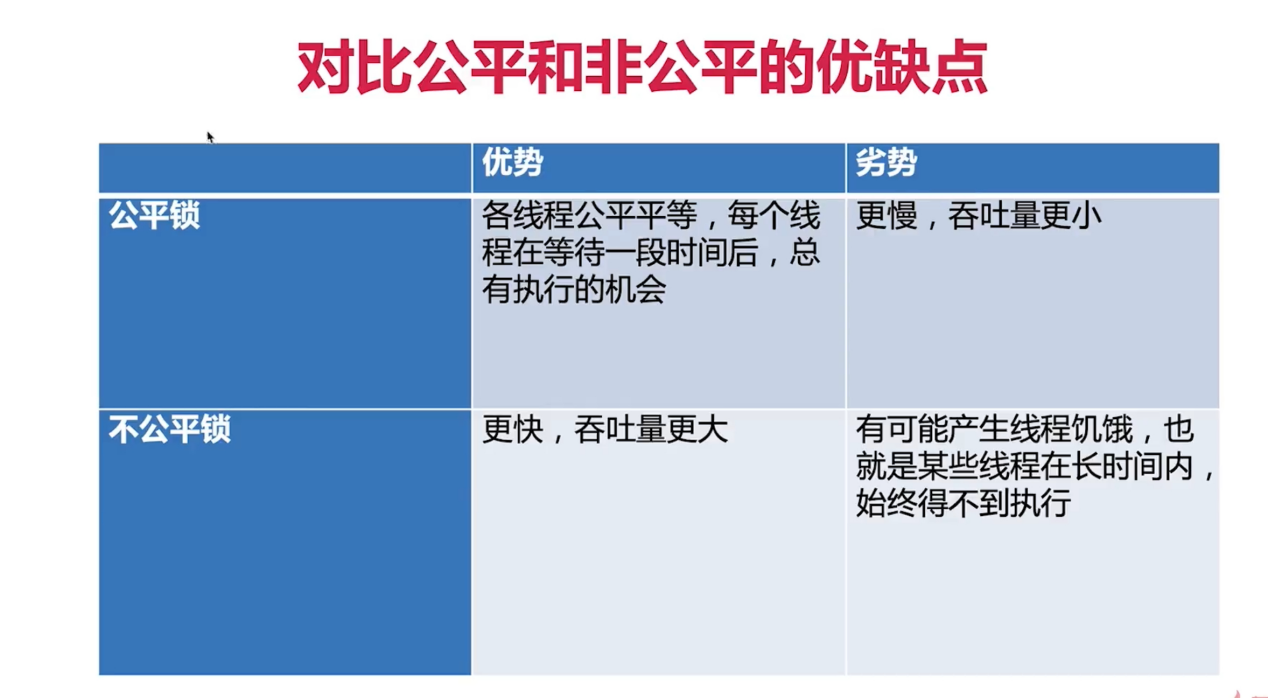
## 代码案例：演示公平和非公平的效果

## 特例

### 针对tryLock()方法，它是很猛的，他不遵守设定的公平的规则

### 例如：当有线程执行tryLock()的时候，一旦有线程释放了锁，那么这个正在tryLock()的线程就会获取到锁，即使在他之前已经有其它现在在等待队列里了。

## 对比公平和非公平的优缺点



公平锁在等待线程的过程中，时间被浪费了，所以速度会慢一些。

对于非公平锁而言，虽然是更快，吞吐量更大，但是会带来线程饥饿的问题，也就是说，每次我释放，总有人来抢，也就是某些线程在很长的一段时间内得不到执行。这就叫做是饥饿。

## 源码分析



公平锁：他会去判断是不是有人已经在队列中已经开始排队了(公平锁红色着重不分)，如果是没有的话，我再去获取，

非公平锁根本就不会判断是队列里面有没有人拿到锁

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

## 什么是共享锁和排它锁？

1. 排它锁又称为独占锁、独享锁
2. 共享锁，又称为读锁，获得共享锁之后，可以查看但无法共享和删除数据，其他线程此时也可以获取到共享锁，也可以查看但无法修改和删除数据。
3. 3.共享锁和排它锁的典型是读写锁ReentrantReadWriteLock，其中读锁是共享锁、写锁是独享锁

## 读写锁的作用

1. 在没有读写锁之前，我们假设使用ReentrantLock，那么虽然我们保证了线程安全，但是也浪费了一定的资源：多个读操作同时进行，并没有线程安全问题。
2. 在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的，提高了程序的执行效率。

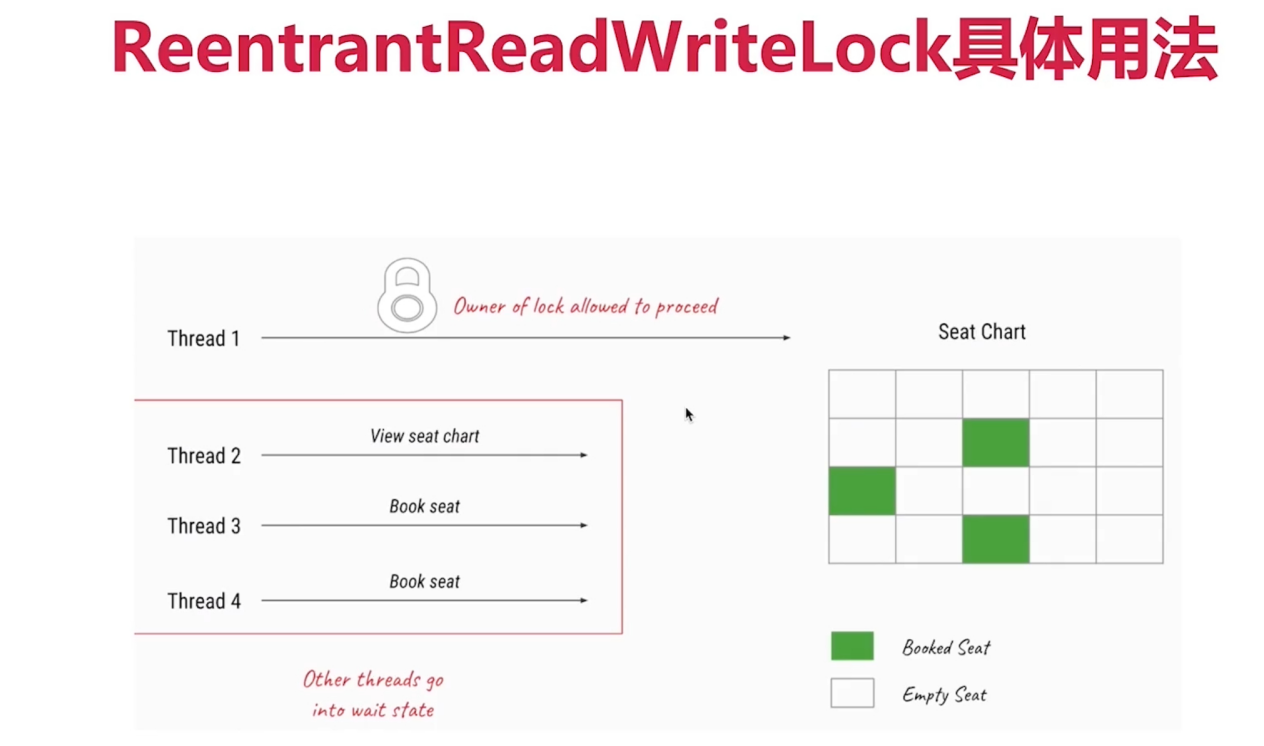
## 读写锁的规则

1. 多个线程只申请读锁，都可以申请到
2. 如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。
3. 如果有一个线程已经占用了写锁，则此时如果其他线程申请写锁或者读锁，则申请的线程会一直等待释放写锁。
4. 一句话总结：要么是一个或多个线程同时有读锁，要么是一个线程有写锁，但是两者不会同时出现(要么多读，要么一写)
5. 换一种思路更容易理解：读写锁只是一把锁，可以通过两种方式锁定：读锁定和写锁定。读写锁可以同时被一个或多个线程读锁定，也可以被单一线程写锁定。但是永远不能同时是这把锁进行读锁定和写锁定。
6. 这里是把”获取写锁”理解为”要读写锁进行写锁定”，相当于是换了一种思路，不过原则是不变的，就是要么是一个或多个线程同时有读锁(同时读锁定)，要么就是一个线程同时有写锁(进行写锁定)，但是两者不会同时出现

## ReentrantReadWriteLock具体用法(代码演示)

### 场景

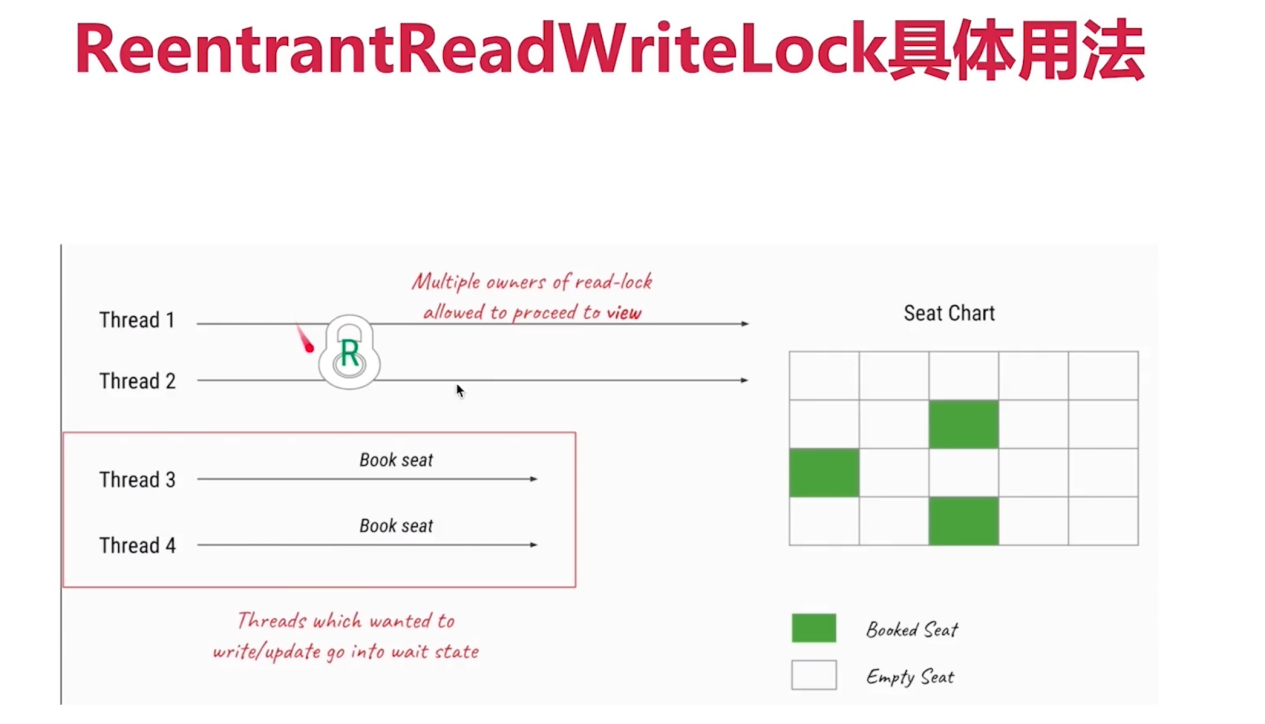
1. 电影院升级



在这种情况下，现在有四个人，他们不是说谁哦度想买票，有的人不想买，只想看一看。只想看一看呢，实际上就是锁他想读取。那么，你读取的情况下，没必要阻塞你。

上面这张图是表示之前我们使用了ReentrantLock

假设第一个线程和第二个线程他们现在是只想查看，第三个和第四个是想修改，但是由于我们是ReentrantLock，不是读写锁，那么这个时候还是得一个一个来，第一个先看，第二个再看，假设我们升级了，升级为读写锁了。



这个R代表是read，在这种情况下，我们是允许多个线程同时读的，所以说，在这种情况下，我们看到线程1和2同时都去查看现在状态是怎么样的，你现在座位卖掉了多少，他们是有这个能力的。

所以说，现在使用读写锁提高了我们得效率。

那么假设现在1和2都已经读完了，就轮到3了，这时间3和4就可以写入了。



到那时我们的写入呢只能是有一个线程持锁进行写入，那么这个时候，由3先去写，W代表write，线程4等待，因为我们不允许两个写锁同时存在，所以呢先现在就是3先来，最后才是4。这就是我们用了这么一个读写锁之后呢么的一个流程。

## 读锁和写锁的交互方式

### 选择规则

选择规则意思就是，在等待队列里面选哪一个线程来执行，比如说我们的现在的一个线程已经是持有读锁了，他正在读取呢，那么后面的队列中，可能有想读的，可能有想写的，有想写的那肯定是要排队，但是假设这个时候有想读的，他能不能插队？

### 读线程插队(比喻：男女共用厕所，男生可以插队吗？)

那么为什么是想要他插队呢？因为是我们现在一个人正在读，那么第二个人你等在后面，但我知道你也想读，那么一起读不就好了吗。这个是不影响的，因为我们的读是多个线程可以同时读的，所以就算来了三个五个，你们一起读，那么最好是你们都同时的读取结束，这不就是提高了效率了吗？所以这种角度去考虑的话，确实是提高了效率。

这就好比是我们男生和女生共用厕所，有的那个厕所是男女混用的，那么在这种情况下，通常情况下，男生至少说从平均角度来讲，可能是比女生上洗手间的时间会快一些，那么是不是就允许你男生就插队呢？其实在景区的时候啊，确实，如果允许的话，可能整体效率可能是提高了一些，但是对女生来讲，其实不公平，人家不想你插队，因为如果你们老是有人来插队的话，其实最终还是对我有影响的。还是拖延了我的时间，

### 升降级

读锁和写锁他不是平等的，实际上写锁我们看出来他更加厉害，读锁相对没有那么厉害。所以我们就考虑，是不是允许他升降级。比如说，他现在持有写锁，能不能让他在不释放写锁的情况下，直接去获取这个读锁呢？可不可以呢？或者说这个是降级是吧，升级可不可以？升级指的是我现在持有读锁，我想升级成为写锁，可不可以呢？这些都是我们在设计读写锁的时候啊，需要考虑的。各种各样的细节。

## ReentrantLock的实现

### 插队

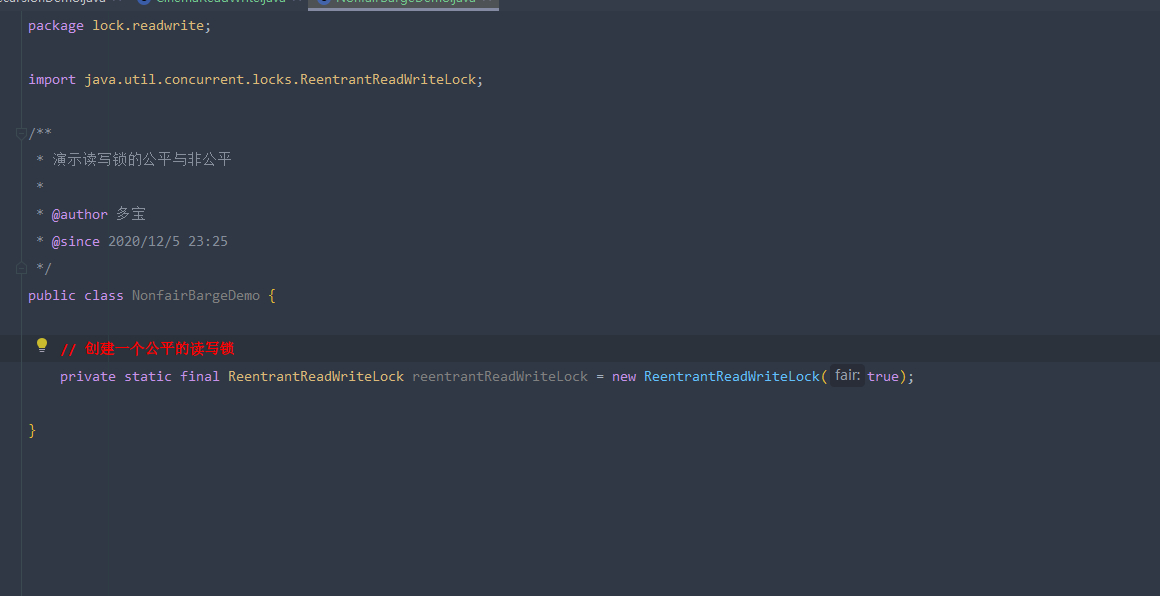
他在插队这个方面是不允许读锁插队的。

1. 读锁插队策略

首先一个，对于公平锁而言，如果他是公平的，那么你就不要想插队了。我们之前说ReentrantLock的时候，我们提到过，他不公平的情况下，那么可能可以插队，那么公平的情况下呢？如果队列里已经有人在排队了，你也是一样，你要去排队。

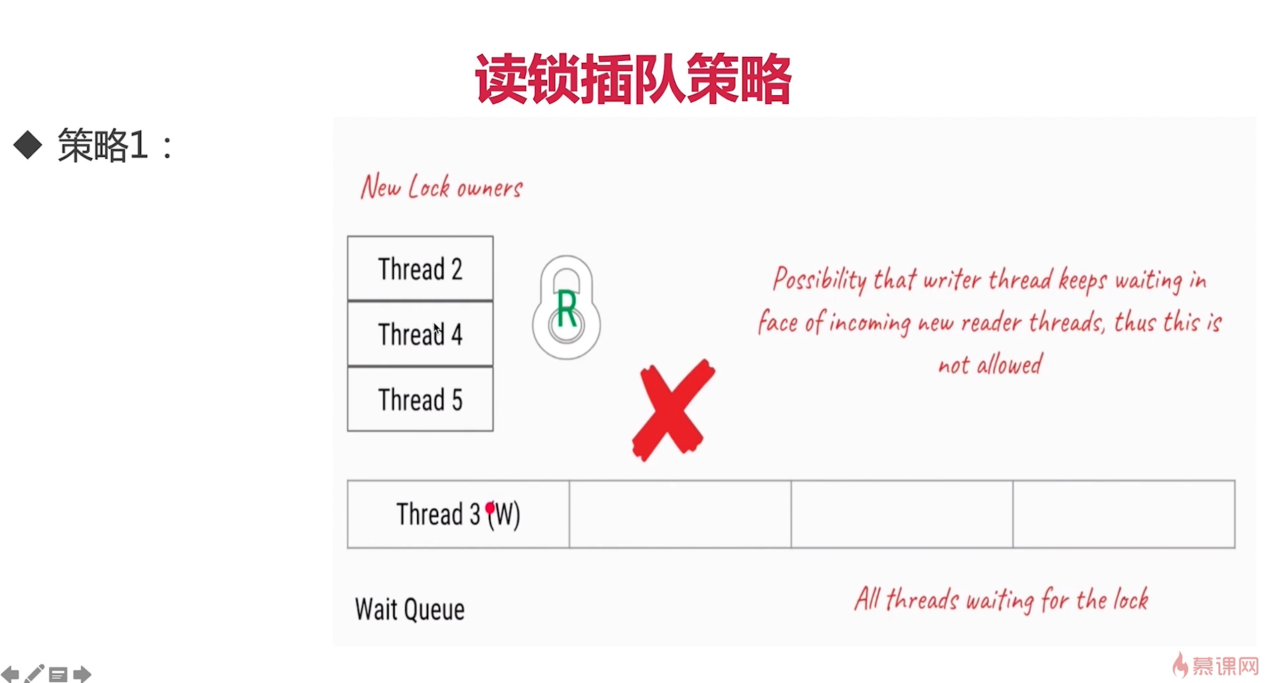
在读写锁这里呢，也是一样的。

1. 如果你是一个公平的读写锁



如果是公平的读写锁，那么就不要想着插队了，如果队列里面有线程在等待的话，那你只能一个一个排队。

1. 非公平：假设线程2和线程4正在同时读取，线程3想要写入，拿不到锁，于是进入等待队列，线程5不在队列里，现在过来想要读取，那么就直接让线程5进行插队读取，是可以进行操作的。但是此时有两种策略。
2. 两种策略
3. 第一种策略



第一种策略是我们想着让5插队，这个时间让5插队的话看下图上展示的情况，2和4原来在读取，5进来了，虽然他看到3在队列里，但是5直接过来插入，这么一种情况。

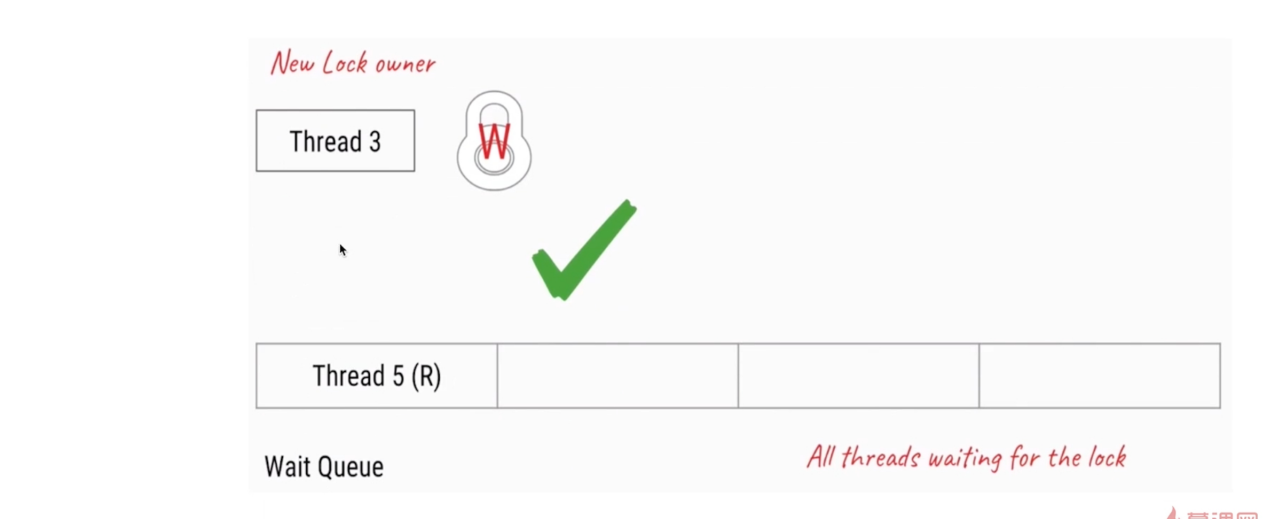
这种策略是有好处的，那么好处在于读可以插队，效率高。因为是线程5不会影响到2和4，你也是没有必要去等3.因为你3呢是写入，其实我是读取，读取可以共存，比如说这个时候，我在多几个也不怕嘛。我再多个线程6/7/8，我一起读，一起读可能就只是废一秒钟，然后全都读完了，这样确实是从整体上提高了效率，要比我等你啊，效率要更高，可是呢，这是一个优点，他也同时是优缺点的。

缺点就在于，容易造成饥饿。为什么呢？我们想一下，如果我们允许5这样干，那么后面还会有6/7/8，6/7/8进来呢，他没有进到队列，就是过来直接就像插队嘛，6/7/89/10线程，一直到100、到1000，他们，他们都想读取，都想插队，如果按照我们这样的政策，5可以插队，后面的6/7/8，一直到1000，都可以插队，那么我们的线程2可能过一会他就读取完毕了，就直接是撤退了，那么这个锁就被线程4和5持有，过一会呢，4也撤退了，这个时候5就变到第一位了，后面来了6/7，然后6/7撤退呢，后面又来8/9，也就是说始终会有人持有着这个读锁，虽然每个线程他持有这个读锁也就一两秒钟，但是你架不住这个线程特别多，也就意味着，假设你允许这个政策是存在的，那么后面来了一千个线程，全是读操作，全部都插队，那么这个时候问题就发生了，我们看看在队列里面的这个线程3，他会特别的孤独寂寞冷，反正就是他这个人很不开心，为什么很不开心？这其实用咱们专业术语来讲，这叫饥饿，因为这个线程3始终得不到运行，因为他已经是排队很久了，本该轮到她的时候，老是被后面的人插队，所以这种政策的最大弊端就是容易让一些想要获得写锁的线程造成饥饿的问题

1. 第二种策略：避免饥饿



现在是2和4正在读取，3在队列中，5过来了，我不让它来插队，直接把它塞到队列里，现在让线程在队列里面等，如果5也去等，等到2和4线程都读取完毕，轮到了3



轮到3之后呢，5就在队列中跑到了第一位置，5的R代表它是想读取，W呢是想写入，线程3是一个写锁，线程3拿到写锁，线程5呢就等待，那么一会呢线程3释放写锁了，于是呢线程5可以拿到读锁，它是这样的一个顺序，也就是说，在这种策略下，我们虽然插队可以提高整体的效率，这里的插队特指我们的读插队可以提高整体效率，但是我们为了防止我们这种饥饿现象的发生，我们不允许他插队

这就是我们ReentrantReadWriteLock的实现。

1. 策略选择演示

①、策略的选择取决于具体锁的实现，ReentrantReadWriteLock的实现是选择了策略2，是很明智的.

### 升降级

在升降级的方面，它是允许降级，不允许升级

### 总结

1. 公平锁：不允许插队
2. 非公平锁：
   1. 写锁可以随时插队

因为本身写锁插队就不容易，就写锁



我们假设是这种情况，这个时间假设又来了一个写锁，thread6我们假设它是写锁，他过来了，现在还是没进入到队列，想尝试插队，那这个时候，他根本就抢不到R这把锁，他没办法插队成功，因为根据我们读写锁的策略呢，不能说同时有读又有写，人家已经在读了，所以说这个时候线程6就会发现，那我没办法了，抢不到锁，插不了队，他就会进来排队，所以这个时候他跟我们的ReentrantLock插队策略是完全一致的。就是说尝试去抢，抢不到就排队，写锁就是这样的。

读锁之所以特殊，就是因为他多个线程可以同时可以去持有这个读锁，所以说他插队很容易，如果这个时候来了一个thread6，他想去插读锁，他一下子就插入成功了，因为读锁可以同时被很多线程持有，所以，正是因为他这个能力太强了吗，我们不得不限制它一下，要是不限制的话，排队的写锁就永远得不到。所以这就是这样设计的一个初衷。

* 1. 读锁仅在等待队列头结点不是想获取写锁的线程的时候可以插队。

也就是说，

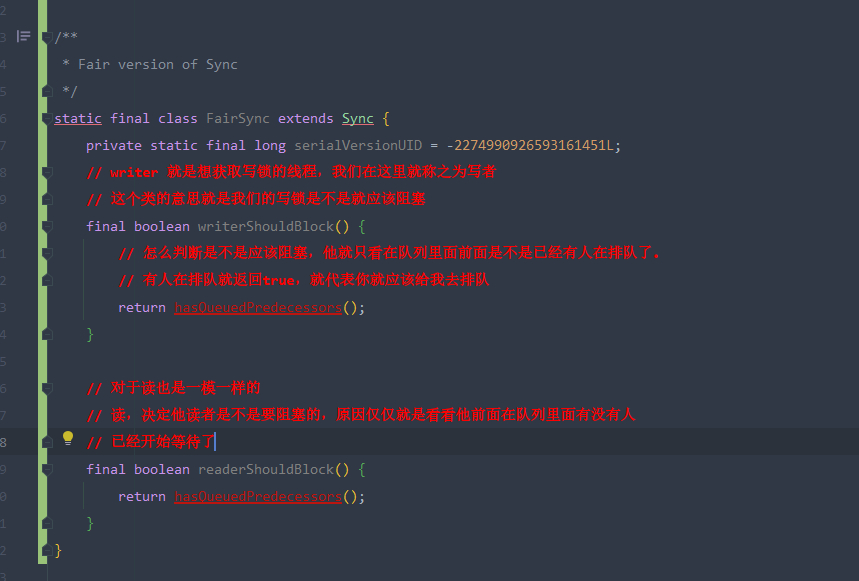


在等待队列里，他发现线程3头结点是W，是写锁，如果是写的话，读锁到了这里，它拿锁之前先判断，第一个是不是写锁，是写锁的话，他就不插队了，直接就进来了，5就是这个情况，刚才5还没进到队列之前，那虽然这个锁是不公平的，虽然允许你插队，但是呢我们5在读取这个读锁之前，他回去看头结点，还不是写锁，如果是写锁，直接就进到等待队列里面了，这个是一个非常完整的表述，

* 1. 源码分析

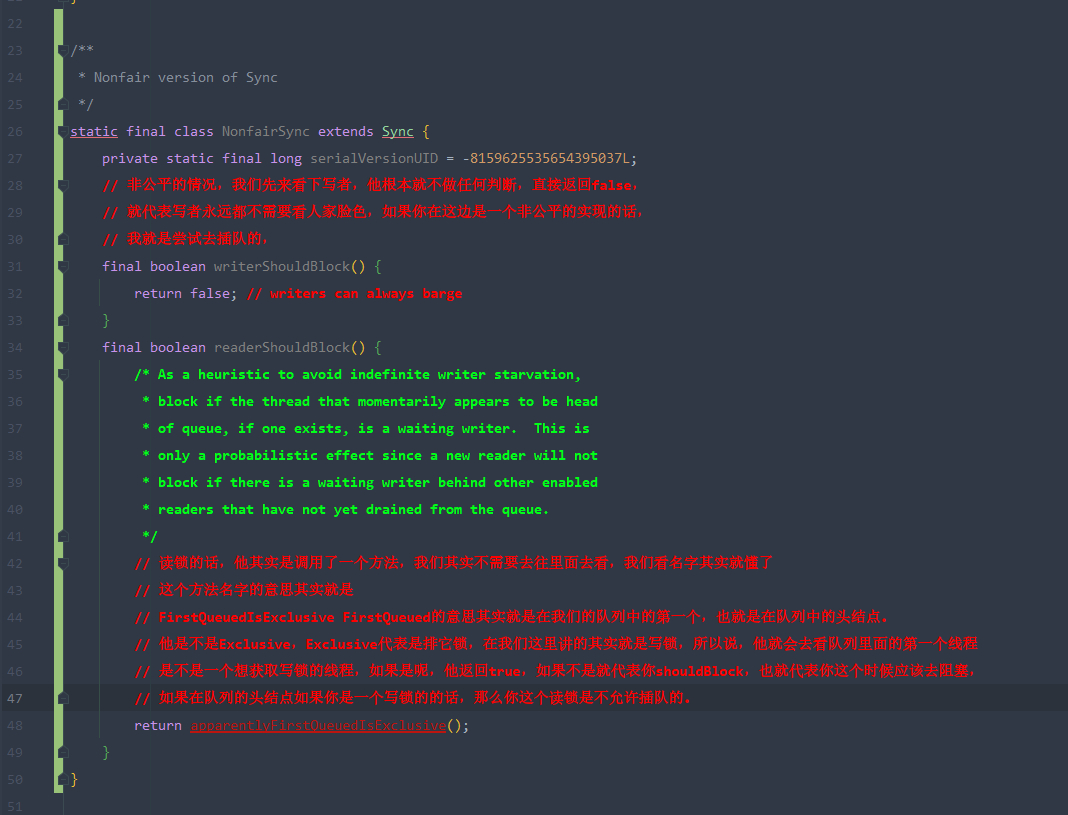


我们先来看公平的情况

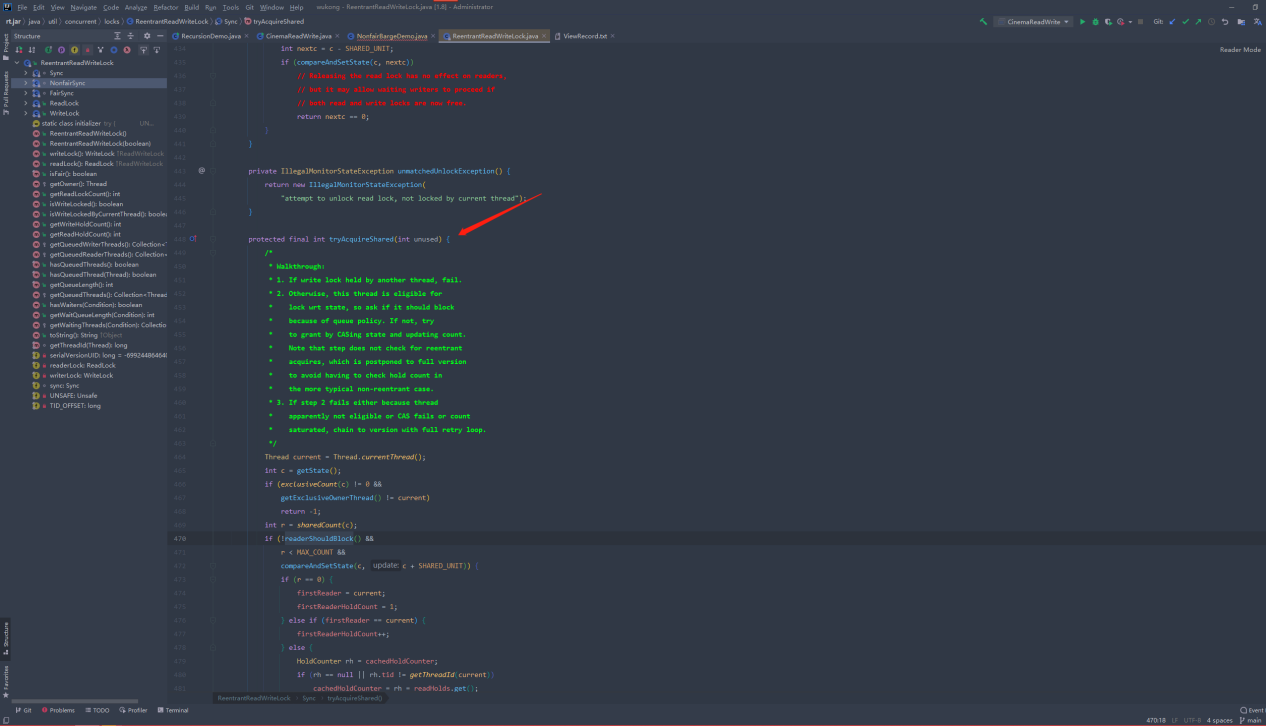


所以这和我们按刚才所研究的是一样的，锁在公平的情况下，你无论谁来，只要队列里已经有等待的线程了，就去等待。

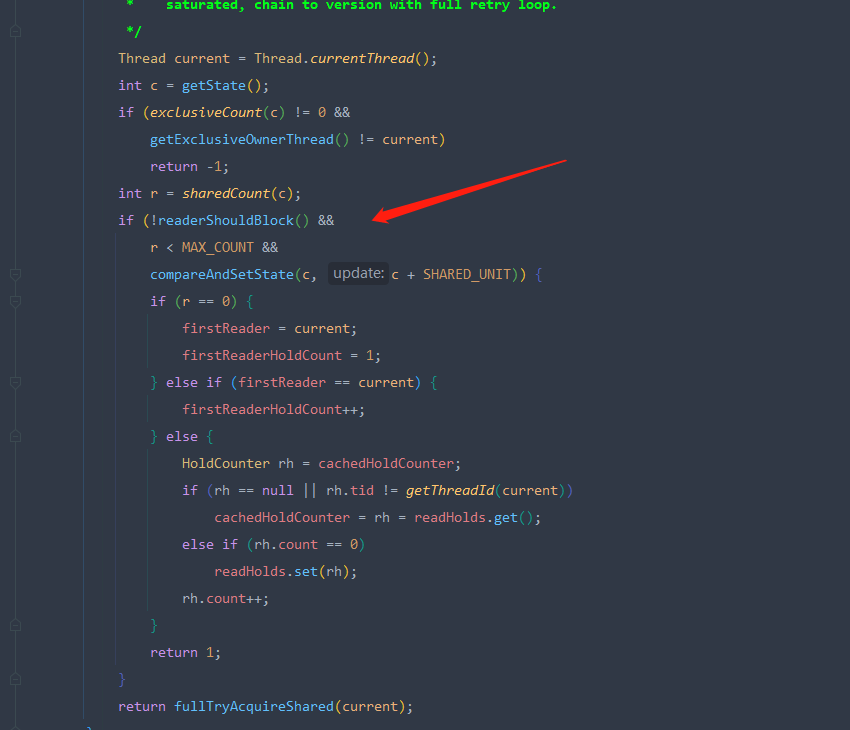
我们再来看下非公平的情况



我们看下readerShouldBlock()是被哪里调用的。



它是被这个方法锁调用的，tryAcquireShared(int unused)这个方法实际上指的就是我们的读锁在获取的时候，读锁一旦像获取，他就会调用这个方法，那么调用之后呢，



他在真正的获取之前，他一定会去检查这个时候是不是应该被阻塞，也就是说即便你想让它去获取我们的读锁，然会呢，他会根据不同的实现来决定它是不是该阻塞，如果是公平的，他就会去检查队列里是不是前面已经有了。如果有就排队，如果是非公平的，他同样会检查，但检查的内容不一样，他会去检查第一个在队列中的节点，是不是写，如果是写，他同样去排队。这个就是我们具体的一个逻辑。

* 1. 代码演示：1、读不插队 2、读实际上可以插队。

### 4.锁的升降级

1、为什么需要升降级？

比如说我们现在有一个任务，他上来可能是先写日志，记录我来参与这次行动了，写完之后，他去读取一些其他文件，或者是怎么样，那么这个任务，他特点就是这样，阿姨开始会有一些的写入，但后面试一些读取，那么这个时候我肯定不希望我整个方法在执行的过程中被打断，所以呢在我认为完全执行完毕之前，我肯定不会主动去释放我的锁，但是，由于我只是一开始我想写，后面呢我只是读，如果我一直持有的都是写锁，那浪费资源啊，那么这个时候，我支持降级的话，那么我直接从我的写锁拿到我的读锁，我在把我的读锁释放掉，这个时候，人家就可以跟我一起来读取了，所以我就提高了整体的效率，因为我后期只需要读取，我不需要写入。我拿着写锁是浪费资源的，但是与此同时我又不想把我的锁释放掉，因为我一旦释放掉，下次轮到我还不知道什么时候呢，我只想在持有的同时想去降级，就是这样的一个需求，所以呢，我们的锁支不支持升降级也是需要我们考虑的。

1. 支持锁的降级，不支持升级：代码演示

对于我们的锁来说，降级毕竟来说还是容易，升级难嘛，代码来演示一下。

1. 为什么不支持所得升级？死锁？

降级的好处就在于它可以提高效率，但是为什么锁是不可以升级的呢？

我们来想一下为什么升级容易造成死锁？我们知道读写锁的特点呢是如果大家都申请读，那么可以多个线程同时读，但如果是写呢，只有一个线程可以写，并且不能同时有读同时有写。

正是因为我们不能同时有读又有写，所以说如果我们是想升级的话，那必须要等到所有的读锁都释放我们才能升级，如果所有的读锁你都释放完毕了，那么这个时候，我升级为写锁，那就没问题了。

我们假设一种情况，我们现在是有三个线程，有A/B/C。他们都是已经获得了读锁了，并且他们都在读取。那么其中A他想升级为写锁，如果A想升级为写，那么这要求我们的B、C必须放弃读锁，因为如果持有读锁的情况下，由于我们不能有读又有写，所以呢，如果B、C还正在读，A是不能升级的。我们假设随着时间的推移，B和C终究会逐渐放弃他们的读锁，他读取完毕他就放弃嘛。那么这个时候A他等到了这个时间，现在轮到A进行升级锁，于是A升级为写锁，这看上去是成功了，这时候，我们再考虑另外一种情况，假设只有两个线程A和B，他们都在读，然后呢，他们同时都想升级，因为假如说你给了人家升级的能力。他们就有可能同时都想升级。

假设A和B本身他们两个同时都在读，现在A想升级，那么A要是想升级要求除了A之外都要释放读锁，那么在这种情况下呢就是B要释放读锁，可是B也想升级啊，B升级要求A释放读锁，那么他们都在等待对方去释放他们的锁。这就陷入了死锁，但是呢，我们说锁的升级啊。并不是不可能的。可以有实现的方案。比如说我们只要能保证我们只能有一个线程升级，其他的线程都必须释放，如果我们能用各种各样的代码来保证的话，那么我们也是可以去实现我们自己的锁的升级的。这个取决于各自锁不同的实现。

但是我们今天的ReentranReadWriteLock它不支持升降级。那么他也是我们所需要主要掌握的锁，所以呢我们要知道它不支持。在这一点上，它只支持降级。不支持升级。目的就是避免死锁。

### 5.共享锁和排它锁总结

1、ReentrantReadWriteLock实现了ReadWriteLock接口，最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁

2、锁申请和释放的策略。

a)、多个线程只申请读锁，都可以申请到。

b)、如果有一个线程已经占用了读锁，则此时其他线程想要申请写锁，则申请写锁的线程会一直等待释放读锁。

c)、如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

d)、要么是一个或多个线程同时有读锁，要么是一个线程有写锁，但是两者不会同时出现。(要么多读，要么一写)

3、插队策略：

为了防止饥饿，读锁不能插队。

1. 升降级策略

只能降级，不能升级。降级的好处，将既可以提高效率，但是呢我们也知道升级的风险，升级可能会带来死锁。

1. 使用场景：

相对于ReentrantLock适用于一般场景，ReentrantReadWriteLock适用于读多写少的情况，合理使用可以进一步提高并发效率。

# 自旋锁和阻塞锁

## 自旋锁

### 概念

1、阻塞或唤醒一个java线程需要操作系统切换CPU状态来完成，这种状态转换需要消耗处理器时间

2、如果同步代码块中的内容过于简单，状态切换消耗的时间有可能比用户代码执行的时间还要长。

3、在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复的花费可能会让系统得不偿失。

4、如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。

5、而为了让当前线程”稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。

### 缺点

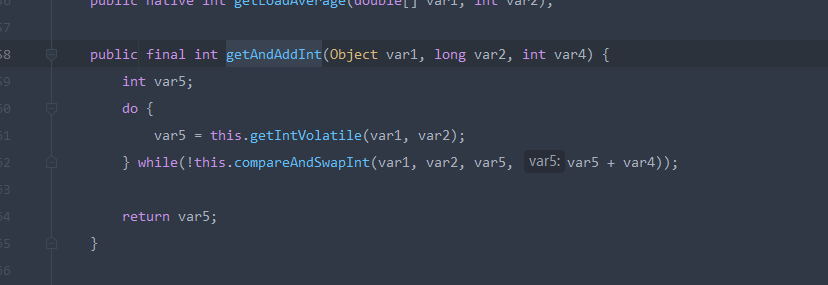
1. 如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源

为什么呢？因为前面那个人人家不想释放。轮到你，你本该去阻塞，你又不阻塞，你老是站着CPU来问我，你问我我就会告诉你。我现在占着的这个人他就说了，别问，问就是不释放，你别来问我，你问我我也不释放，那如果是这种情况的话，我们自旋锁它的效率就不高了。

1. 在自旋的过程中，一直消耗CPU，所以虽然自旋锁的起始开销低于悲观锁。但是随着自旋时间的增长，开销也是线性增长的。

### 原理和源码分析

1. 在java1.5版本及以上的并发框架java.util.concurrent的atomic包下的类基本上都是自旋锁的实现。
2. AtomicInteger的实现，自旋锁的实现原理是CAS。AtomicInteger中调用unsafe进行自增操作的源码中的do-while循环就是一个自旋操作，如果修改过程中遇到其他线程竞争导致没修改成功，就在while里死循环，直至修改成功。



我们看下AtomicInteger里面的getAndIncrement()，他里面就是一个do-while操作。

3.自己写一个简单的自旋锁

### 适用场景

1. 自旋锁一般适用于多核的服务器，在并发度不是特别高的情况下，比阻塞锁的效率高
2. 另外，自旋锁适用于临界区比较短小的情况，否则如果临界区很大(线程一旦拿到锁，很久以后才会释放)，那也是不合适的

## 阻塞锁

### 概念

1、阻塞锁与自旋锁相反，阻塞锁如果遇到没拿到锁的情况，会直接把线程阻塞，直到被唤醒

# 可中断锁：顾名思义，就是可以响应中断的锁

## 不可中断锁

1. 在java中，synchronized就是不可中断锁，而Lock是可中断锁，因为tryLock(time)和lockInterruptibly都能响应中断。
2. 如果某一个线程A正在执行锁中的代码，另一个线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以中断它，这种就是可中断锁。

# 锁优化

## java虚拟机对锁的优化。

### 自旋锁和自适应锁。

这实际上是一种优化，在此之前我们说过自旋锁，还指出呢他有一些不好的地方，但是你不能只看到人家不好的地方，这是一种重要的发明。在有些情况下，比如说我们锁比较短小的情况下，那么我们使用自旋锁，它是可以很有效的提高我们的效率的。并且呢，他还出现了一种自适应的概念。

自适应指的就是我们在尝试自旋的时候，不是说一直就是盲目到底，尝试各一百万遍他还是自己继续进行尝试，他是说你尝试个十次二十次，哎，你发现尝试不到，于是呢他就转为阻塞锁。这是非常聪明的一种体现，这就是叫做自适应，并且这个转为的次数呢也是可以不断完善的。就是说这次他可能发现自旋成功了，于是下一次呢，他还等待一百个循环尝试。那么如果这一次自旋失败了，那么后面呢他可能连自旋都不要了，直接就进入到了阻塞状态，这一系列的预测呢都代表我们的虚拟机变得越来越聪明了。我们也可以在JVM的配置中区设置默认的自旋参数

### 2.锁消除

就是说有一些场景下我们不必要加锁。比如说这段代码啊他就是在我们方法内部的，并且呢所有同步的东西都是在我方法内部的。那么在这种情况下根本就不可能有外人来访问到我里面的这个东西。那么在这种情况下呢虚拟机会分析出来，认为他们是私有的，无需加锁，于是呢就把这个锁直接就给消除了。这就是锁消除的情况

### 3.锁粗化

我们之前有一个原则就是希望让锁，也就是同步块的范围越小越好，这样一来呢，我们就可以在真正需要同步的地方才同步，如果是有竞争呢也可以尽快拿到并且释放锁。在大部分情况下呢，上面这些原则都是对的。但是在有些情况下，一系列的操作都是在对一个对象繁复的加锁解锁，那么这种情况下，这种加锁解锁开销是比较大的 。那么不如啊，就把这一系列的加锁解锁合为一个，在最开始弄一个加锁，在最后弄一个解锁。这么一来呢，就消除了这么多加锁解锁的过程。提高了效率，我们的JVM啊，他会动态监测。如果发现前后相邻的synchronized代码块使用的是同一个锁对象，那么他就把这几个合为一个较大的，这样一来在执行的时候就无须反复的申请和释放锁了。只要申请和释放锁一次就能执行完全部的代码块，也提高了性能。