https://mp.weixin.qq.com/s/tSxOeJxJIcjoc-l1uxPFmg

**25 张图 | 深入浅出「偏向锁」**

在 JDK1.5 之前，面对 Java 并发问题， synchronized 是一招鲜的解决方案：

1. 普通同步方法，锁上当前实例对象
2. 静态同步方法，锁上当前类 Class 对象
3. 同步块，锁上括号里面配置的对象

monitorenter 指令是在编译后插入到同步代码块的开始位置；monitorexit是插入到方法结束和异常的位置(实际隐藏了try-finally)，每个对象都有一个 monitor 与之关联，当一个线程执行到 monitorenter 指令时，就会获得对象所对应的 monitor 的所有权，也就获得到了对象的锁

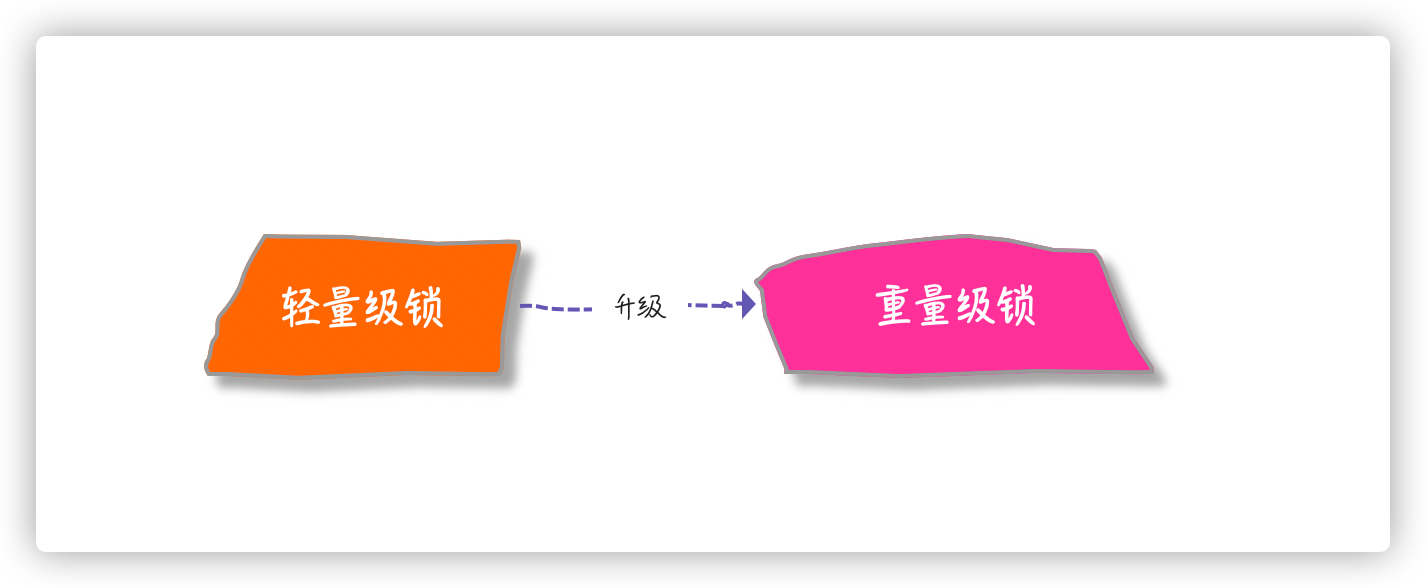
当另外一个线程执行到同步块的时候，由于它没有对应 monitor 的所有权，就会被阻塞，此时控制权只能交给操作系统，也就会从 user mode 切换到 kernel mode, 由操作系统来负责线程间的调度和线程的状态变更, 需要频繁的在这两个模式下切换（上下文转换）。这种有点竞争就找内核的行为很不好，会引起很大的开销，所以大家都叫它重量级锁，自然效率也很低，这也就给很多童鞋留下了一个根深蒂固的印象 —— synchronized关键字相比于其他同步机制性能不好。

## 锁的演变

来到 JDK1.6，要怎样优化才能让锁变的轻量级一些？答案就是：

### 轻量级锁：CPU CAS

如果 CPU 通过简单的 CAS 能处理加锁/释放锁，这样就不会有上下文的切换，较重量级锁而言自然就轻了很多。但是当竞争很激烈，CAS 尝试再多也是浪费 CPU，权衡一下，不如升级成重量级锁，阻塞线程排队竞争，也就有了轻量级锁升级成重量级锁的过程

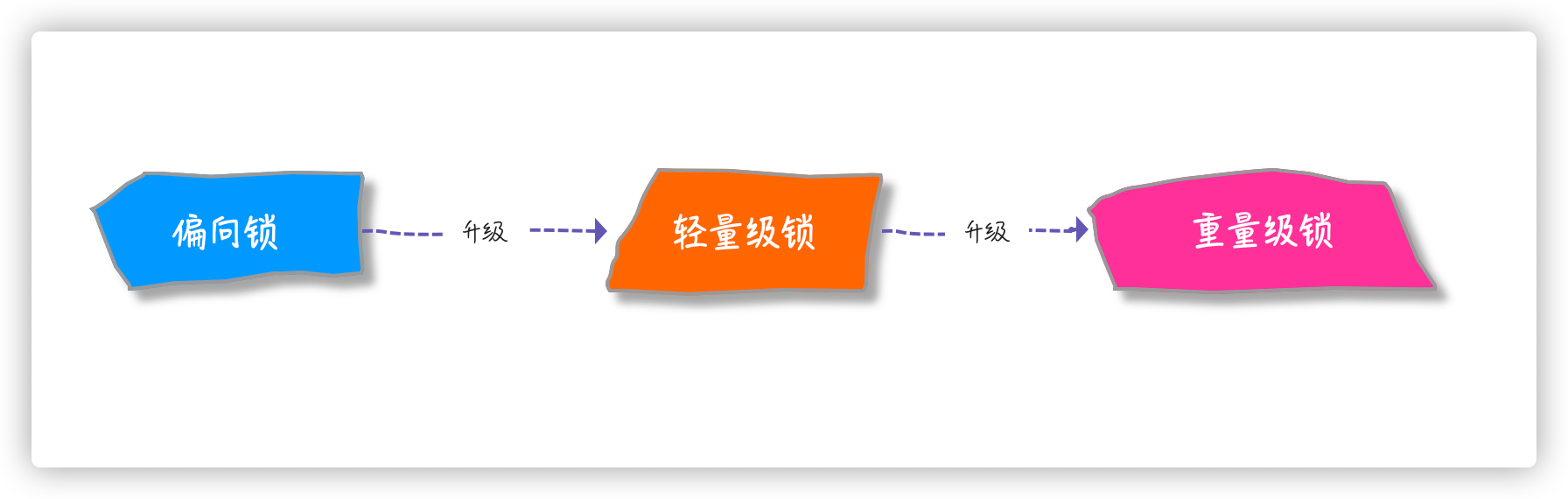


程序员在追求极致的道路上是永无止境的，HotSpot 的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一个线程多次获得，同一个线程反复获取锁，如果还按照轻量级锁的方式获取锁（CAS），也是有一定代价的，如何让这个代价更小一些呢？

偏向锁

偏向锁实际就是锁对象潜意识「偏心」同一个线程来访问，让锁对象记住线程 ID，当线程再次获取锁时，亮出身份，如果同一个 ID 直接就获取锁就好了，是一种 load-and-test 的过程，相较 CAS 自然又轻量级了一些

可是多线程环境，也不可能只是同一个线程一直获取这个锁，其他线程也是要干活的，如果出现多个线程竞争的情况，也就有了偏向锁升级的过程。



这里可以先思考一下：偏向锁可以绕过轻量级锁，直接升级到重量级锁吗？

都是同一个锁对象，却有多种锁状态，其目的显而易见：

占用的资源越少，程序执行的速度越快

偏向锁，轻量锁，它俩都不会调用系统互斥量（Mutex Lock），只是为了提升性能，多出的两种锁的状态，这样可以在不同场景下采取最合适的策略，所以可以总结性的说：

* 偏向锁：无竞争的情况下，只有一个线程进入临界区，采用偏向锁
* 轻量级锁：多个线程可以交替进入临界区，采用轻量级锁
* 重量级锁：多线程同时进入临界区，交给操作系统互斥量来处理

到这里，大家应该理解了全局大框，但仍然会有很多疑问：

1. 锁对象是在哪存储线程 ID 才可以识别同一个线程的？
2. 整个升级过程是如何过渡的？

想理解这些问题，需要先知道 Java 对象头的结构

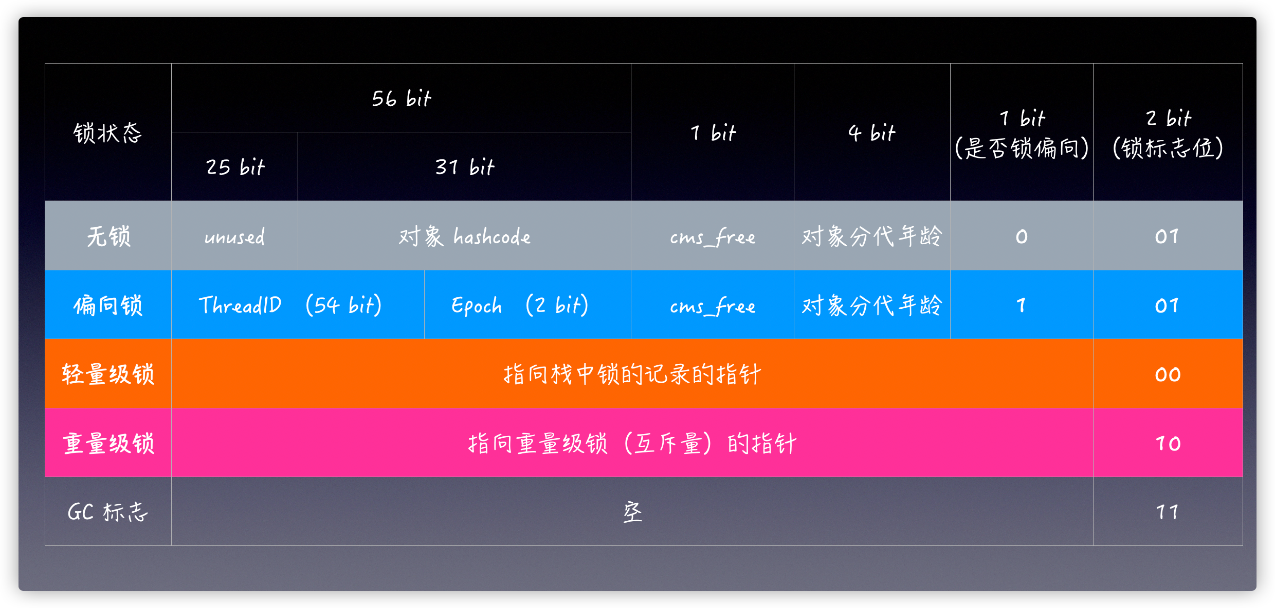
## 认识 Java 对象头

按照常规理解，识别线程 ID 需要一组 mapping 映射关系来搞定，如果单独维护这个 mapping 关系又要考虑线程安全的问题。奥卡姆剃刀原理，Java 万物皆是对象，对象皆可用作锁，与其单独维护一个 mapping 关系，不如中心化将锁的信息维护在 Java 对象本身上。

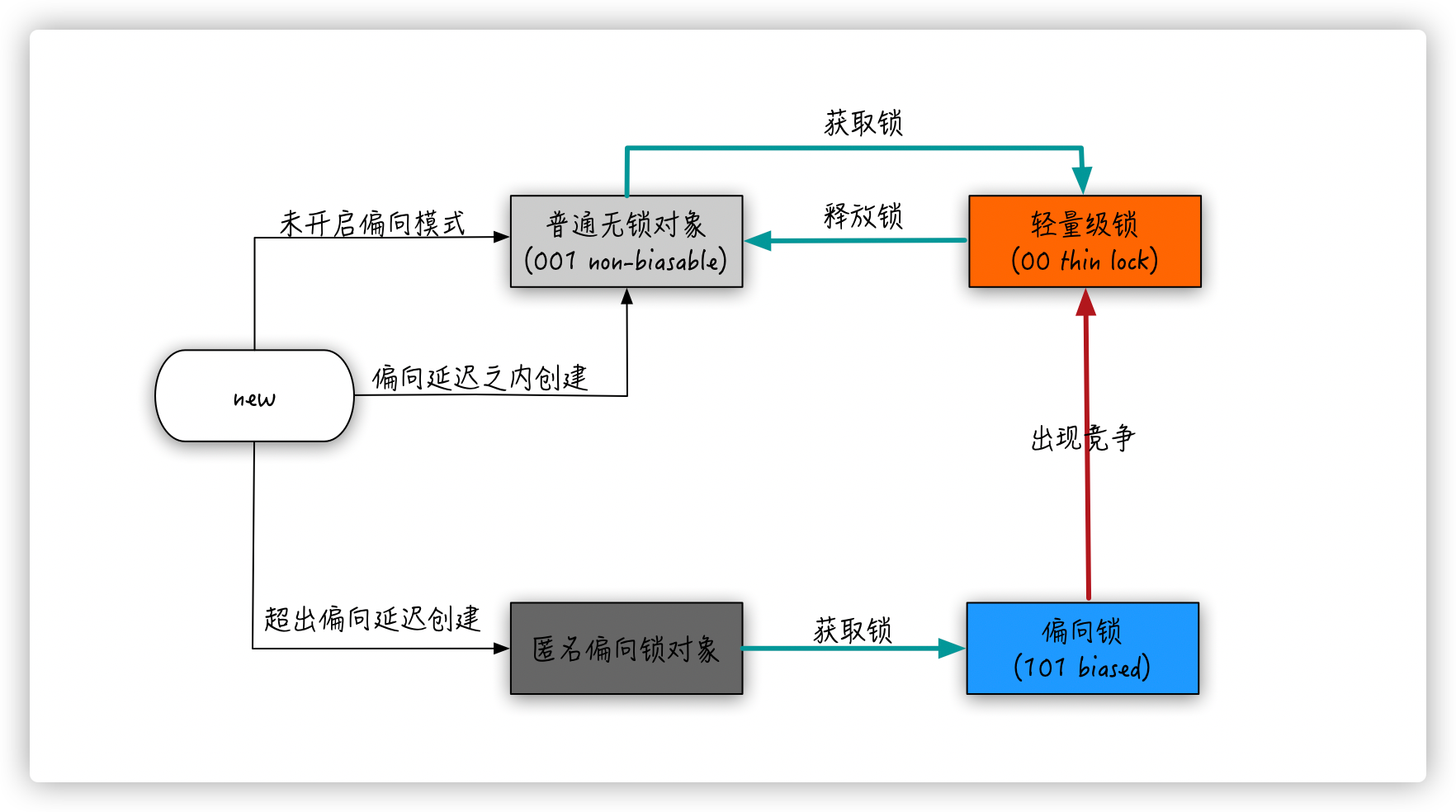
Java 对象头最多由三部分构成：

* MarkWord
* ClassMetadata Address
* Array Length （如果对象是数组才会有这部分）

其中 Markword 是保存锁状态的关键，对象锁状态可以从偏向锁升级到轻量级锁，再升级到重量级锁，加上初始的无锁状态，可以理解为有 4 种状态。想在一个对象中表示这么多信息自然就要用位存储，在 64 位操作系统中，是这样存储的（注意颜色标记），想看具体注释的可以看 hotspot(1.8) 源码文件 path/hotspot/src/share/vm/oops/markOop.hpp 第 30 行。



虽然默认开启了偏向锁，但是开启有延迟，大概 4s。原因是 JVM 内部的代码有很多地方用到了synchronized，如果直接开启偏向，产生竞争就要有锁升级，会带来额外的性能损耗，所以就有了延迟策略。



从这样的运行结果上来看，偏向锁像是“一锤子买卖”，只要偏向了某个线程，后续其他线程尝试获取锁，都会变为轻量级锁，这样的偏向非常有局限性。事实上并不是这样，如果你仔细看标记2（已偏向状态），还有个 epoch 我们没有提及，这个值就是打破这种局限性的关键，在了解 epoch 之前，我们还要了解一个概念——偏向撤销。

### 偏向撤销

在真正讲解偏向撤销之前，需要和大家明确一个概念——偏向锁撤销和偏向锁释放是两码事

1. 撤销：笼统的说就是多个线程竞争导致不能再使用偏向模式的时候，主要是告知这个锁对象不能再用偏向模式
2. 释放：和你的常规理解一样，对应的就是 synchronized 方法的退出或 synchronized 块的结束

何为偏向撤销？

从偏向状态撤回原有的状态，也就是将 MarkWord 的第 3 位（是否偏向撤销）的值，从 1 变回 0

如果只是一个线程获取锁，再加上「偏心」的机制，是没有理由撤销偏向的，所以**偏向的撤销只能发生在有竞争的情况下**

想要撤销偏向锁，还不能对持有偏向锁的线程有影响，所以就要等待持有偏向锁的线程到达一个 safepoint 安全点 (这里的安全点是 JVM 为了保证在垃圾回收的过程中引用关系不会发生变化设置的一种安全状态，在这个状态上会暂停所有线程工作)， 在这个安全点会挂起获得偏向锁的线程

在这个安全点，线程可能还是处在不同状态的，先说结论（因为源码就是这么写的，可能有疑惑的地方会在后面解释）

线程不存活或者活着的线程但退出了同步块，很简单，直接撤销偏向就好了

活着的线程但仍在同步块之内，那就要升级成轻量级锁

这个和 epoch 貌似还是没啥关系，因为这还不是全部场景。偏向锁是特定场景下提升程序效率的方案，可并不代表程序员写的程序都满足这些特定场景，比如这些场景（在开启偏向锁的前提下）：

1. 一个线程创建了大量对象并执行了初始的同步操作，之后在另一个线程中将这些对象作为锁进行之后的操作。这种case下，会导致大量的偏向锁撤销操作
2. 明知有多线程竞争（生产者/消费者队列），还要使用偏向锁，也会导致各种撤销

很显然，这两种场景肯定会导致偏向撤销的，一个偏向撤销的成本无所谓，大量偏向撤销的成本是不能忽视的。那怎么办？既不想禁用偏向锁，还不想忍受大量撤销偏向增加的成本，这种方案就是设计一个有阶梯的底线。

#### 批量重偏向（bulk rebias）

这是第一种场景的快速解决方案，以 class 为单位，为每个 class 维护一个偏向锁撤销计数器，每一次该class的对象发生偏向撤销操作时，该计数器 +1，当这个值达到重偏向阈值（默认20）时：

BiasedLockingBulkRebiasThreshold = 20

JVM 就认为该class的偏向锁有问题，因此会进行批量重偏向, 它的实现方式就用到了我们上面说的 epoch

Epoch，如其含义「纪元」一样，就是一个时间戳。每个 class 对象会有一个对应的epoch字段，每个处于偏向锁状态对象的mark word 中也有该字段，其初始值为创建该对象时 class 中的epoch的值（此时二者是相等的）。每次发生批量重偏向时，就将该值加1，同时遍历JVM中所有线程的栈

找到该 class 所有正处于加锁状态的偏向锁对象，将其epoch字段改为新值

class 中不处于加锁状态的偏向锁对象（没被任何线程持有，但之前是被线程持有过的，这种锁对象的 markword 肯定也是有偏向的），保持 epoch 字段值不变

这样下次获得锁时，发现当前对象的epoch值和class的epoch，本着今朝不问前朝事 的原则（上一个纪元），那就算当前已经偏向了其他线程，也不会执行撤销操作，而是直接通过 CAS 操作将其mark word的线程 ID 改成当前线程 ID，这也算是一定程度的优化，毕竟没升级锁；

如果 epoch 都一样，说明没有发生过批量重偏向, 如果 markword 有线程ID，还有其他锁来竞争，那锁自然是要升级的(如同前面举的例子 epoch=0)。

批量重偏向是第一阶梯底线，还有第二阶梯底线。

#### 批量撤销（bulk revoke）

当达到重偏向阈值后，假设该 class 计数器继续增长，当其达到批量撤销的阈值后（默认40）时，

BiasedLockingBulkRevokeThreshold = 40

JVM就认为该 class 的使用场景存在多线程竞争，会标记该 class 为不可偏向。之后对于该 class 的锁，直接走轻量级锁的逻辑

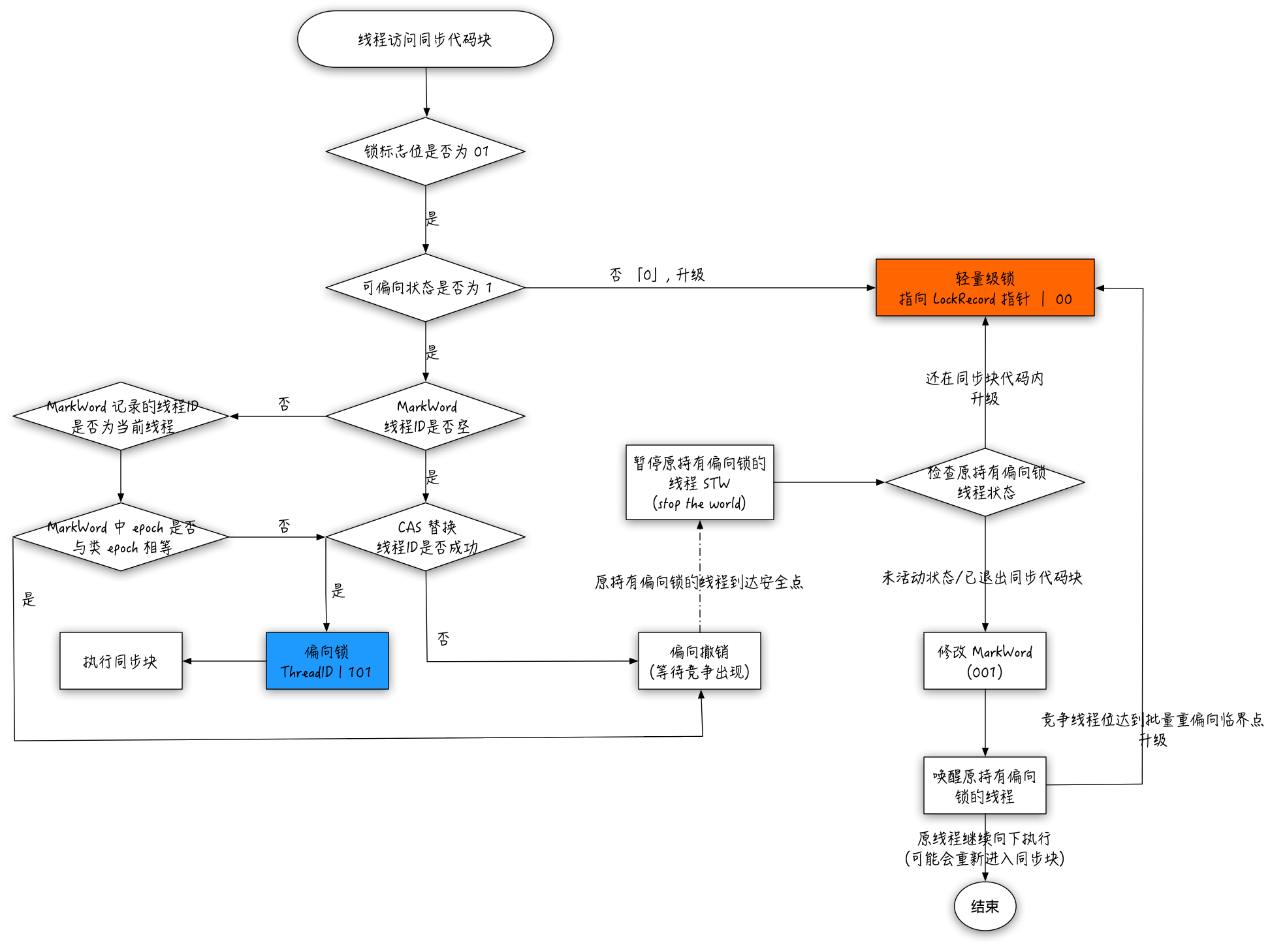
这就是第二阶梯底线，但是在第一阶梯到第二阶梯的过渡过程中，也就是在彻底禁用偏向锁之前，还给一次改过自新的机会，那就是另外一个计时器：

BiasedLockingDecayTime = 25000

1. 如果在距离上次批量重偏向发生的 25 秒之内，并且累计撤销计数达到40，就会发生批量撤销（偏向锁彻底 game over）
2. 如果在距离上次批量重偏向发生超过 25 秒之外，那么就会重置在 [20, 40) 内的计数, 再给次机会

大家有兴趣可以写代码测试一下临界点，观察锁对象 markword 的变化

至此，整个偏向锁的工作流程可以用一张图表示：



### HashCode 哪去了

上面场景一，无锁状态，对象头中没有 hashcode；偏向锁状态，对象头还是没有 hashcode，那我们的 hashcode 哪去了？

首先要知道，hashcode 不是创建对象就帮我们写到对象头中的，而是要经过第一次调用 Object::hashCode() 或者System::identityHashCode(Object) 才会存储在对象头中的。第一次生成的 hashcode后，该值应该是一直保持不变的，但偏向锁又是来回更改锁对象的 markword，必定会对 hashcode 的生成有影响，那怎么办呢？，我们来用代码验证：

结论就是：即便初始化为可偏向状态的对象，一旦调用 Object::hashCode() 或者System::identityHashCode(Object) ，进入同步块就会直接使用轻量级锁。

#### 场景二

假如已偏向某一个线程，然后生成 hashcode，然后同一个线程又进入同步块，会发生什么呢？来看代码：

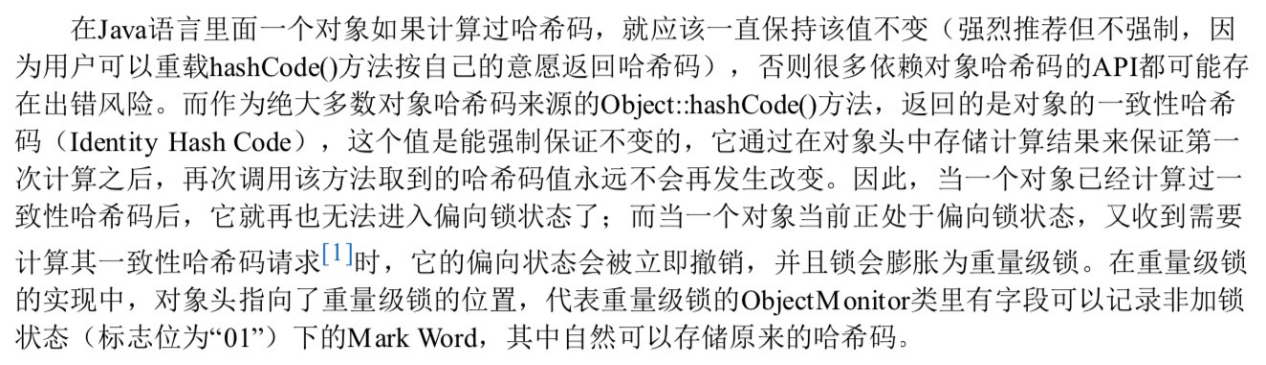
结论就是：同场景一，会直接使用轻量级锁。

#### 场景三

那假如对象处于已偏向状态，在同步块中调用了那两个方法会发生什么呢？继续代码验证：

结论就是：如果对象处在已偏向状态，生成 hashcode 后，就会直接升级成重量级锁。

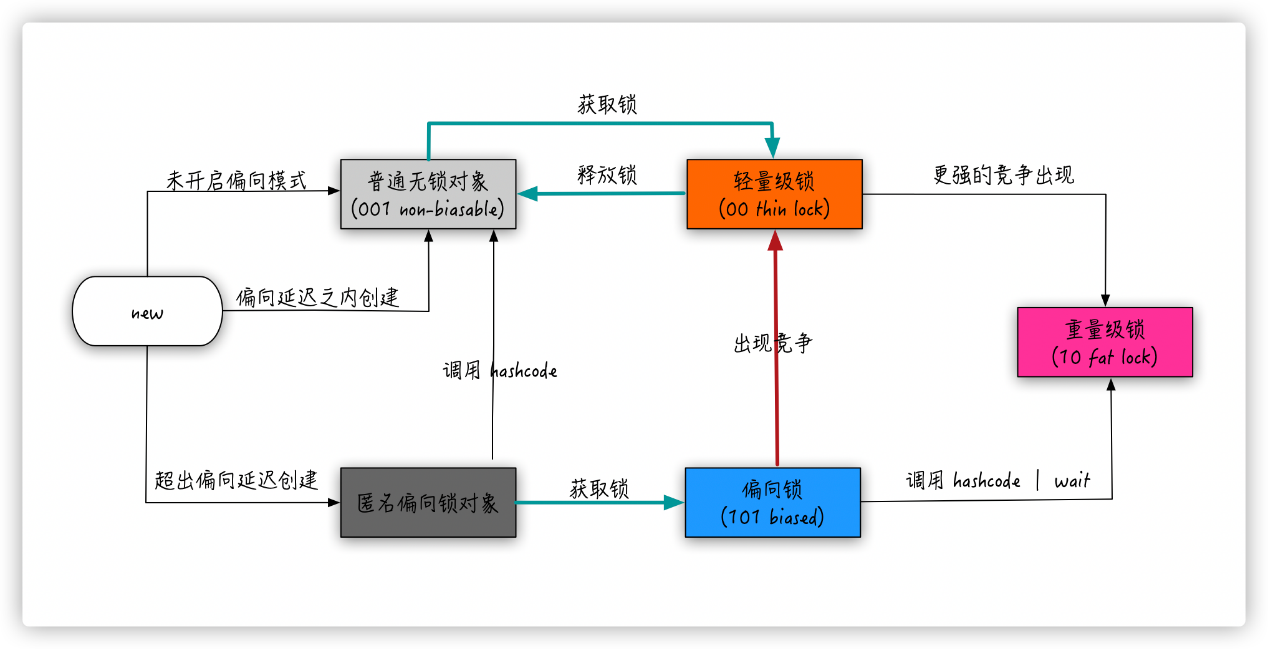
最后用书中的一段话来描述 锁和hashcode 之前的关系：



#### 调用 Object.wait() 方法会发生什么？

Object 除了提供了上述 hashcode 方法，还有 wait() 方法，这也是我们在同步块中常用的，那这会对锁产生哪些影响呢？来看代码：

结论就是，wait 方法是互斥量（重量级锁）独有的，一旦调用该方法，就会升级成重量级锁（这个是面试可以说出的亮点内容哦）。

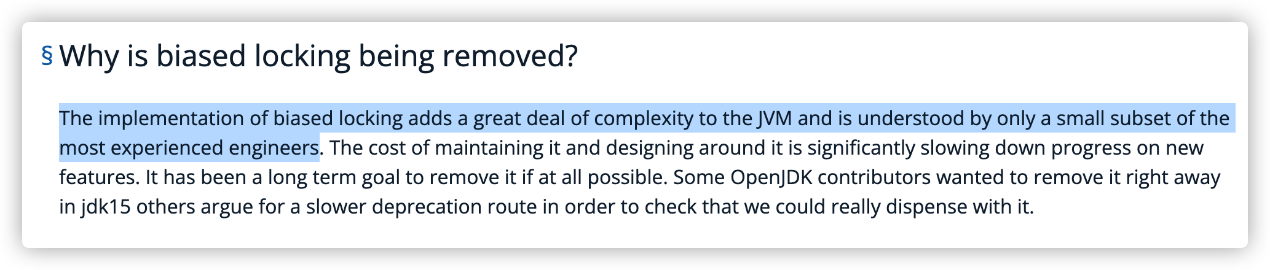


## 告别偏向锁

看到这个标题你应该是有些慌，为啥要告别偏向锁，因为维护成本有些高了，来看 Open JDK 官方声明，JEP 374: Deprecate and Disable Biased Locking，相信你看上面的文字说明也深有体会，为了一个现在少有的场景付出了巨大的代码实现。

最终就是，JDK 15 之前，偏向锁默认是 enabled，从 15 开始，默认就是 disabled，除非显示的通过 UseBiasedLocking 开启

其中在 quarkus 上的一篇文章说明的更加直接



偏向锁给 JVM 增加了巨大的复杂性，只有少数非常有经验的程序员才能理解整个过程，维护成本很高，大大阻碍了开发新特性的进程（换个角度理解，你掌握了，是不是就是那少数有经验的程序员了呢？哈哈）

## 总结

偏向锁可能就这样的走完了它的一生，有些同学可能直接发问，都被 deprecated 了，JDK都 17 了，还讲这么多干什么？

1. java 任它发，我用 Java8，这是很多主流的状态，至少你用的版本没有被 deprecated
2. 面试还是会被经常问到
3. 万一哪天有更好的设计方案，“偏向锁”又以新的形式回来了呢，了解变化才能更好理解背后设计
4. 奥卡姆剃刀原理，我们现实中的优化也一样，如果没有必要不要增加实体，如果增加的内容带来很大的成本，不如大胆的废除掉，接受一点落差