<https://mp.weixin.qq.com/s/pE1_gDkTpHsbJCgbptG4Qw>

# 这回，阿粉要将 synchronized 与锁的关系讲透！！！

# **synchronized 关键字**

说到锁，都会提 synchronized 。这个英文单词儿啥意思呢？翻译成中文就是「同步」的意思

一般都是使用 synchronized 这个关键字来给一段代码或者一个方法上锁，使得这段代码或者方法，在同一个时刻只能有一个线程来执行它。

synchronized 相比于 volatile 来说，用的比较灵活，你可以在方法上使用，可以在静态方法上使用，也可以在代码块上使用。

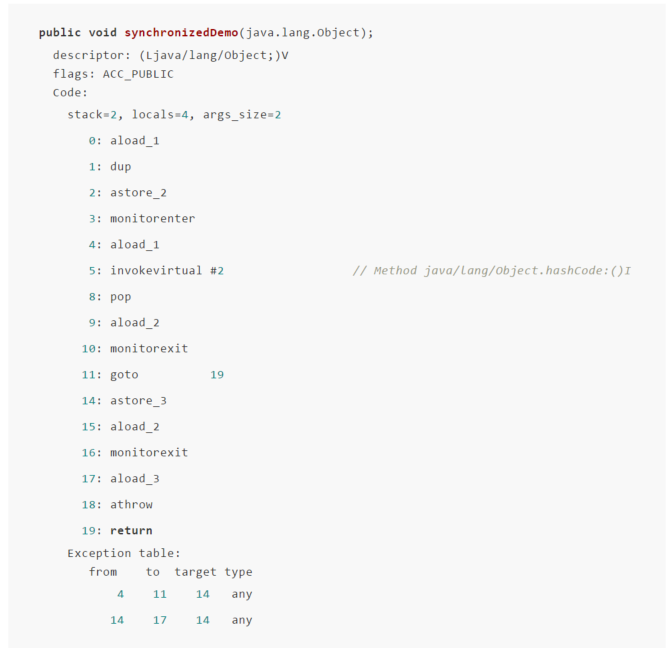
关于 synchronized 这一块大概就说到这里，阿粉今天想着重来说一下， synchronized 底层是怎么实现的

## **JVM 是如何实现 synchronized 的？**

我知道可以利用 synchronized 关键字来给程序进行加锁，但是它具体怎么实现的我不清楚呀，别急，咱们先来看个 demo :

**public** **class** **demo** {  
    **public** **void** **synchronizedDemo**(Object lock){  
  **synchronized**(lock){  
   lock.hashCode();  
  }  
 }  
}

上面是我写的一个 demo ，然后进入到 class 文件所在的目录下，使用 javap -v demo.class 来看一下编译的字节码(在这里我截取了一部分):



应该能够看到当程序声明 synchronized 代码块时，编译成的字节码会包含 monitorenter和 monitorexit 指令，这两种指令会消耗操作数栈上的一个引用类型的元素(也就是 synchronized 关键字括号里面的引用)，作为所要加锁解锁的锁对象。如果看的比较仔细的话，上面有一个 monitorenter 指令和两个 monitorexit 指令，这是 Java 虚拟机为了确保获得的锁不管是在正常执行路径，还是在异常执行路径上都能够解锁。

关于 monitorenter 和 monitorexit ，可以理解为每个锁对象拥有一个锁计数器和一个指向持有该锁的线程指针:

* 当程序执行 monitorenter 时，如果目标锁对象的计数器为 0 ，说明这个时候它没有被其他线程所占有，此时如果有线程来请求使用， Java 虚拟机就会分配给该线程，并且把计数器的值加 1
  + 目标锁对象计数器不为 0 时，如果锁对象持有的线程是当前线程， Java 虚拟机可以将其计数器加 1 ，如果不是呢？那很抱歉，就只能等待，等待持有线程释放掉
* 当执行 monitorexit 时， Java 虚拟机就将锁对象的计数器减 1 ，当计数器减到 0 时，说明这个锁就被释放掉了，此时如果有其他线程来请求，就可以请求成功

为什么采用这种方式呢？是为了允许同一个线程重复获取同一把锁。比如，一个 Java 类中拥有好多个 synchronized 方法，那这些方法之间的相互调用，不管是直接的还是间接的，都会涉及到对同一把锁的重复加锁操作。这样去设计的话，就可以避免这种情况。

# **锁**

在 Java 多线程中，所有的锁都是基于对象的。也就是说， Java 中的每一个对象都可以作为一个锁。你可能会有疑惑，不对呀，不是还有类锁嘛。但是 class 对象也是特殊的 Java 对象，所以呢，在 Java 中所有的锁都是基于对象的

在 Java6 之前，所有的锁都是"重量级"锁，重量级锁会带来一个问题，就是如果程序频繁获得锁释放锁，就会导致性能的极大消耗。为了优化这个问题，引入了"偏向锁"和"轻量级锁"的概念。所以在 Java6 及其以后的版本，一个对象有 4 种锁状态:无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态。

在 4 种锁状态中，无锁状态应该比较好理解，无锁就是没有锁，任何线程都可以尝试修改，所以这里就一笔带过了。

随着竞争情况的出现，锁的升级非常容易发生，但是如果想要让锁降级，条件非常苛刻，有种你想来可以，但是想走不行的赶脚。

阿粉在这里啰嗦一句:很多文章说，锁如果升级之后是不能降级的，其实在 HotSpot JVM 中，是支持锁降级的

锁降级发生在 Stop The World 期间，当 JVM 进入安全点的时候，会检查有没有闲置的锁，如果有就会尝试进行降级

看到 Stop The World 和 安全点 可能有人比较懵，我这里简单说一下，具体还需要读者自己去探索一番.(因为这是 JVM 的内容，这篇文章的重点不是 JVM )

在 Java 虚拟机里面，传统的垃圾回收算法采用的是一种简单粗暴的方式，就是 Stop-the-world ，而这个 Stop-the-world 就是通过安全点( safepoint )机制来实现的，安全点是什么意思呢？就是 Java 程序在执行本地代码时，如果这段代码不访问 Java 对象/调用 Java 方法/返回到原来的 Java 方法，那 Java 虚拟机的堆栈就不会发生改变，这就代表执行的这段本地代码可以作为一个安全点。当 Java 虚拟机收到 Stop-the-world 请求时，它会等所有的线程都到达安全点之后，才允许请求 Stop-the-world 的线程进行独占工作

接下来就介绍一下几种锁和锁升级

## **Java 对象头**

在刚开始就说了， Java 的锁都是基于对象的，那是怎么告诉程序我是个锁呢？就不得不来说， Java 对象头每个 Java 对象都有对象头，如果是非数组类型，就用 2 个字宽来存储对象头，如果是数组，就用 3 个字宽来存储对象头。在 32 位处理器中，一个字宽是 32 位;在 64 位处理器中，字宽就是 64 位咯~对象头的内容就是下面这样:

| **长度** | **内容** | **说明** |
| --- | --- | --- |
| 32/64 bit | Mark Word | 存储对象的 hashCode 或锁信息等 |
| 32/64 bit | Class Metadata Address | 存储到对象类型数据的指针 |
| 32/64 bit | Array length | 数组的长度(如果是数组) |

咱们主要来看 Mark Word 的内容:

| **锁状态** | **29 bit/61 bit** | **1 bit 是否是偏向锁** | **2 bit 锁标志位** |
| --- | --- | --- | --- |
| 无锁 |  | 0 | 01 |
| 偏向锁 | 线程 ID | 1 | 01 |
| 轻量级锁 | 指向栈中锁记录的指针 | 此时这一位不用于标识偏向锁 | 00 |
| 重量级锁 | 指向互斥量(重量级锁)的指针 | 此时这一位不用于标识偏向锁 | 10 |
| GC 标记 |  | 此时这一位不用于标识偏向锁 | 11 |

从上面表格中，应该能够看到，是偏向锁时， Mark Word 存储的是偏向锁的线程 ID ；是轻量级锁时， Mark Word 存储的是指向线程栈中 Lock Record 的指针；是重量级锁时， Mark Word 存储的是指向堆中的 monitor 对象的指针。

## **偏向锁**

HotSpot 的作者经过大量的研究发现，在大多数情况下,**锁不仅不存在多线程竞争，而且总是由同一线程多次获得**

基于此，就引入了偏向锁的概念

所以啥是偏向锁呢？用大白话说就是，我现在给锁设置一个变量，当一个线程请求的时候，发现这个锁是 true ，也就是说这个时候没有所谓的资源竞争，那也不用走什么加锁/解锁的流程了，直接拿来用就行。但是如果这个锁是 false 的话，说明存在其他线程竞争资源，那咱们再走正规的流程

### **看一下具体的实现原理:**

当一个线程第一次进入同步块时，会在对象头和栈帧中的锁记录中存储锁偏向的线程 ID 。当下次该线程进入这个同步块时，会检查锁的 Mark Word 里面存放的是不是自己的线程 ID。如果是，说明线程已经获得了锁，那么这个线程在进入和退出同步块时，都不需要花费 CAS 操作来加锁和解锁；如果不是，说明有另外一个线程来竞争这个偏向锁，这时就会尝试使用 CAS 来替换 Mark Word 里面的线程 ID 为新线程的 ID 。此时会有两种情况:

替换成功，说明之前的线程不存在了，那么 Mark Word 里面的线程 ID 为新线程的 ID ，锁不会升级，此时仍然为偏向锁

替换失败，说明之前的线程仍然存在，那就暂停之前的线程，设置偏向锁标识为 0 ，并设置锁标志位为 00 ，升级为轻量级锁，按照轻量级锁的方式进行竞争锁

### **撤销偏向锁**

偏向锁使用了一种等到竞争出现时才释放锁的机制。也就说，如果没有人来和我竞争锁的时候，那么这个锁就是我独有的，当其他线程尝试和我竞争偏向锁时，我会释放这个锁。

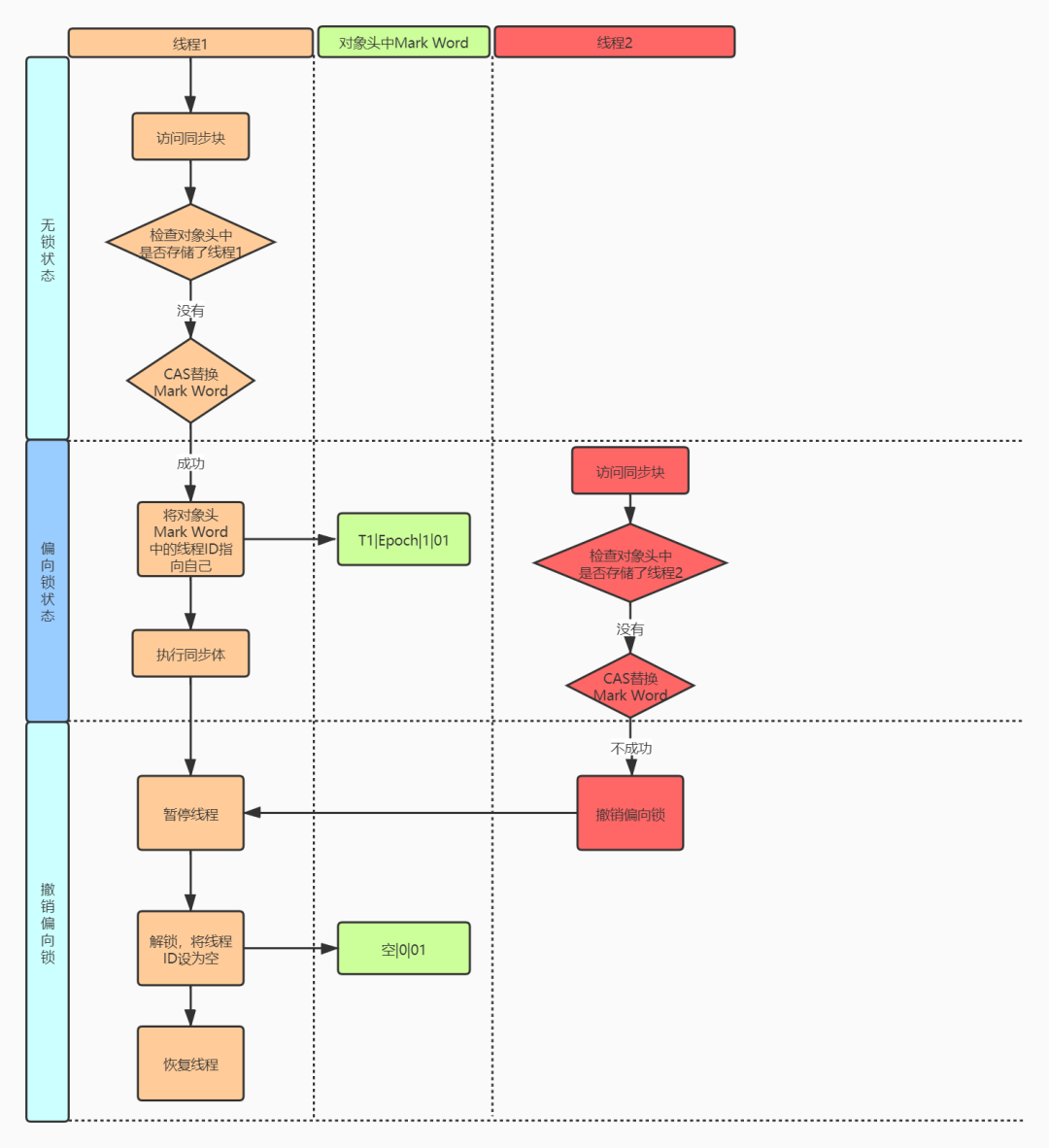
在偏向锁向轻量级锁升级时，首先会暂停拥有偏向锁的线程，重置偏向锁标识，看起来这个过程挺简单的，但是开销是很大的，因为:

* 首先需要在一个安全点停止拥有锁的线程
* 然后遍历线程栈，如果存在锁记录的话，就需要修复锁记录和 Mark Word ，变成无锁状态
* 最后唤醒被停止的线程，把偏向锁升级成轻量级锁

你以为就是升级一个轻量级锁？too young too simple

偏向锁向轻量级锁升级的过程中，是非常耗费资源的，如果应用程序中所有的锁通常都处于竞争状态，偏向锁此时就是一个累赘，此时就可以通过 JVM 参数关闭偏向锁: -XX:-UseBiasedLocking=false ，那么程序默认会进入轻量级锁状态

最后，来张图吧~



## **轻量级锁**

如果多个线程在不同时段获取同一把锁，也就是不存在锁竞争的情况，那么 JVM 就会使用轻量级锁来避免线程的阻塞与唤醒

### **轻量级锁加锁**

JVM 会为每个线程在当前线程的栈帧中创建用于存储锁记录的空间，称之为 Displaced Mark Word 。如果一个线程获得锁的时候发现是轻量级锁，就会将锁的 Mark Word 复制到自己的 Displaced Mark Word 中。之后线程会尝试用 CAS 将锁的 Mark Word 替换为指向锁记录的指针。

如果替换成功，当前线程获得锁，那么整个状态还是 轻量级锁 状态

如果替换失败了呢？说明 Mark Word 被替换成了其他线程的锁记录，那就尝试使用自旋来获取锁.(自旋是说，线程不断地去尝试获取锁，一般都是用循环来实现的)

自旋是耗费 CPU 的，如果一直获取不到锁，线程就会一直自旋， CPU 那么宝贵的资源就这么被白白浪费了

解决这个问题最简单的办法就是指定自旋的次数，比如如果没有替换成功，那就循环 10 次，还没有获取到，那就进入阻塞状态

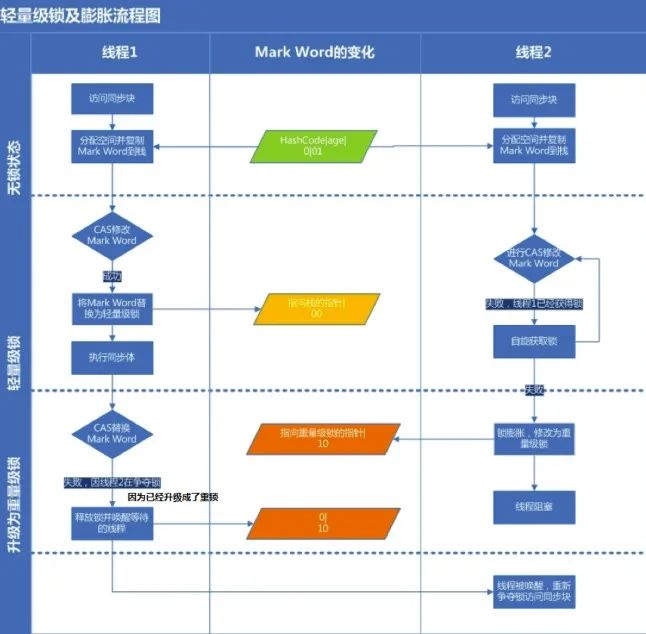
但是 JDK 采用了一个更加巧妙的方法---适应性自旋。就是说，如果这次线程自旋成功了，那我下次自旋次数更多一些，因为我这次自旋成功，说明我成功的概率还是挺大的，下次自旋次数就更多一些，那么如果自旋失败了，下次我自旋次数就减少一些，就比如，已经看到了失败的前兆，那我就先溜，而不是非要“不撞南墙不回头”

自旋失败之后，线程就会阻塞，同时锁会升级成**重量级锁**

### **轻量级锁释放:**

在释放锁时，当前线程会使用 CAS 操作将 Displaced Mark Word 中的内容复制到锁的 Mark Word 里面。如果没有发生竞争，这个复制的操作就会成功;如果有其他线程因为自旋多次导致轻量级锁升级成了重量级锁， CAS 操作就会失败，此时会释放锁同时唤醒被阻塞的过程

同样，来一张图吧:



## **重量级锁**

重量级锁依赖于操作系统的互斥量( mutex )来实现。但是操作系统中线程间状态的转换需要相对比较长的时间(因为操作系统需要从用户态切换到内核态，这个切换成本很高),所以重量级锁效率很低，但是有一点就是，被阻塞的线程是不会消耗 CPU 的

每一个对象都可以当做一个锁，那么当多个线程同时请求某个对象锁时，它会怎么处理呢？

对象锁会设置集中状态来区分请求的线程:

Contention List:所有请求锁的线程将被首先放置到该竞争队列

Entry List: Contention List 中那些有资格成为候选人的线程被移到 Entry List 中

Wait Set:调用 wait 方法被阻塞的线程会被放置到 Wait Set 中

OnDeck:任何时刻最多只能有一个线程正在竞争锁，该线程称为 OnDeck

Owner:获得锁的线程称为 Owner

!Owner:释放锁的线程

当一个线程尝试获得锁时，如果这个锁被占用，就会把该线程封装成一个 ObjectWaiter对象插入到 Contention List 队列的队首，然后调用 park 函数挂起当前线程

当线程释放锁时，会从 Contention List 或者 Entry List 中挑选一个线程进行唤醒

如果线程在获得锁之后，调用了 Object.wait 方法，就会将该线程放入到 WaitSet 中，当被 Object.notify 唤醒后，会将线程从 WaitSet 移动到 Contention List 或者 Entry List 中。

但是，当调用一个锁对象的 wait 或 notify 方法时,**如果当前锁的状态是偏向锁或轻量级锁，则会先膨胀成重量级锁**

## **总结:**

synchronized 关键字是通过 monitorenter 和 monitorexit 两种指令来保证锁的

当一个线程准备获取共享资源时:

首先检查 MarkWord 里面放的是不是自己的 ThreadID ，如果是，说明当前线程处于 "偏向锁"

如果不是，锁升级，这时使用 CAS 操作来执行切换，新的线程根据 MarkWord 里面现有的 ThreadID 来通知之前的线程暂停，将 MarkWord 的内容置为空

然后，两个线程都将锁对象 HashCode 复制到自己新建的用于存储锁的记录空间中，接着开始通过 CAS 操作，把锁对象的 MarkWord 的内容修改为自己新建的记录空间地址，以这种方式竞争 MarkWord ，成功执行 CAS 的线程获得资源，失败的则进入自旋

自旋的线程在自旋过程中，如果成功获得资源（也就是之前获得资源的线程执行完毕，释放了共享资源），那么整个状态依然是 轻量级锁 的状态

如果没有获得资源，就进入 重量级锁 的状态，此时，自旋的线程进行阻塞，等待之前线程执行完成并且唤醒自己