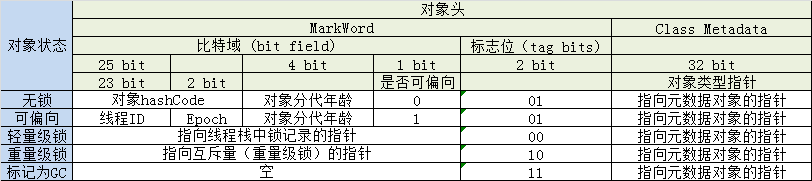
# **一些知识：**

## 1、CAS指令

该指令是是原子性的， 也就是说 CPU 执行该指令时， 是不会被中断执行其他指令的

该指令概念上存在 3 个参数， 第一个参数【目标地址】， 第二个参数【值1】， 第三个参数【值2】， 指令会比较【目标地址存储的内容】和 【值1】 是否一致， 如果一致， 则将【值 2】 填写到【目标地址】

## 2、对象头的Mark Word

****

3、

# **Synchronized关键字**

## 一、作用

1、确保线程互斥的访问同步代码

2、保证共享变量的修改能够及时可见

3、有效解决重排序问题

## 二、用法

1、修饰普通方法：对同一个对象（的方法）进行同步（可与其他对象并行）。但是其他线程还是可以访问该实例对象的其他非synchronized方法

2、修饰静态方法：对同一个类（的静态方法）进行同步。对静态方法的同步本质上是对类的同步，因为静态方法本质上是属于类的方法，而不是对象上的方法。（可与其他类并行）

3、修饰代码块：对同一个对象或类（的代码块）进行同步

## 三、原理

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.（可重入）

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

注意：

1、代码块（monitorenter和monitorexit）

2、方法级的同步是隐式，即无需通过字节码指令来控制的。ACC\_SYNCHRONIZED：该标识指明了方法是否为一个同步方法

## 四、低效的原因（重量级锁）

监视器锁本质是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间。

## 五、等待唤醒机制

指notify/notifyAll和wait方法，在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

## 六、锁的四种状态（锁的升级是单向的，只能从低到高升级）

### 1、无锁状态

### 2、偏向锁

场景：在只有一个线程执行同步块时进一步提高性能。只需要在置换ThreadID的时候依赖一次CAS原子指令

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，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。

失败的情况：存在其他线程申请锁（锁竞争）

### 3、轻量级锁00

场景：线程交替执行同步块。目的：减少无实际竞争情况下，使用重量级锁产生的性能消耗。

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

失败的情况：存在锁竞争激烈

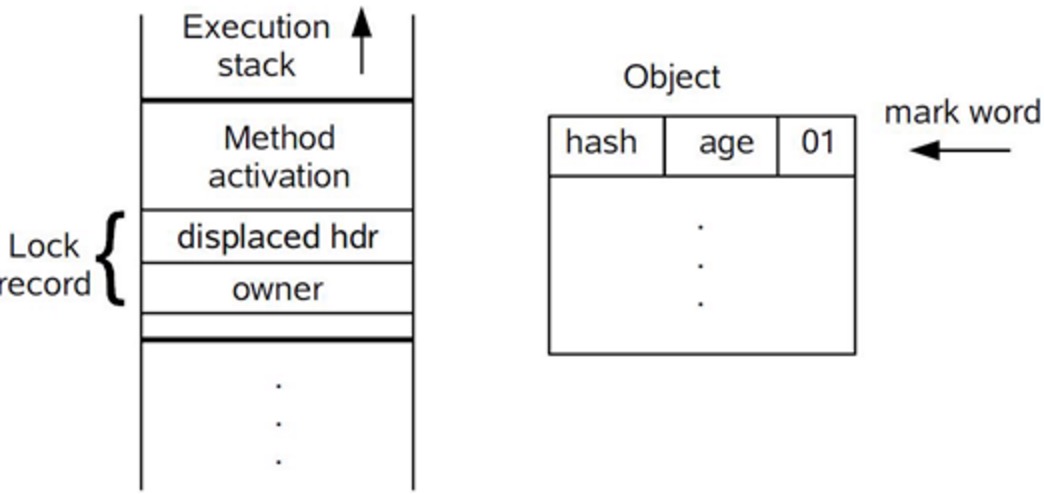
### 4、重量级锁10

当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。

## 七、加锁过程

### 1、轻量级锁的加锁过程

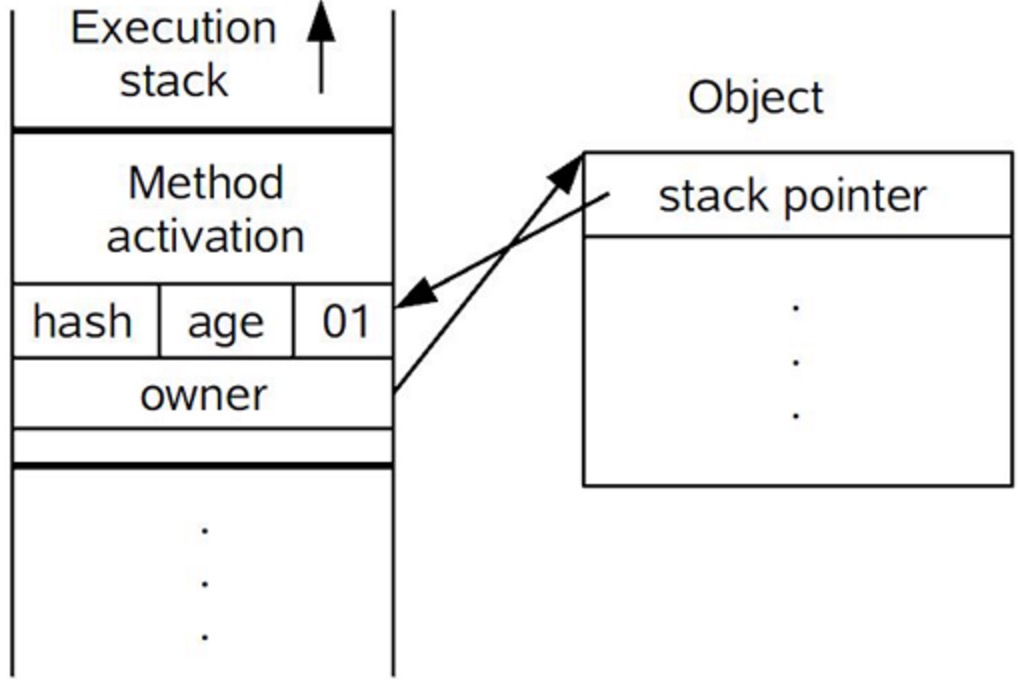
（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位：01，是否为偏向锁：0），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为Displaced Mark Word。这时候线程堆栈与对象头状态如下图：



（2）拷贝对象头中的Mark Word到锁记录中

（3）拷贝成功后，虚拟机将使用CAS操作将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指向object mark word。如果更新成功，则执行步骤4，否则执行步骤5。

（4）如果这个更新操作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为00，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如下图：



(5)如果这个更新操作失败，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志变为10，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞而采取循环获取锁的过程。

### 2、轻量级锁的解锁过程

（1）通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换成当前的Mark Word

（2）如果成功，整个同步过程就完成了

（3）如果失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

### 3、偏向锁的加锁过程

（1）访问Mark Word中偏向锁的标志是否设置成1，锁标志位是否为01（确认可偏向状态）

（2）如果为可偏向状态

* 如果CAS操作成功，则认为已经获取到对象的偏向锁，执行同步代码

补充：一个线程在执行完同步代码块以后，并不会尝试将Mark Word中的Thread ID赎回原值。这样做的好处是：如果该线程需要再次对这个对象加锁，而这个对象之前一直没有被其他线程尝试获取过锁，依旧停留在可偏向的状态下，即可以不修改对象头的情况下，直接认为偏向成功。

* 如果CAS操作失败，则说明有另外一个线程Thread B抢先获取了偏向锁。这种状态说明该对象的竞争比较激烈，此时需要撤销Thread B获得的偏向锁，将Thread B持有的锁升级为轻量级锁。该操作需要等待全局安全点JVM safepoint（此时间点，没有线程在执行字节码）

（3）如果是已偏向状态，则检测Mark Word中存储的Thread ID是否等于当前Thread ID

* 如果相等，则证明本线程已经获取到偏向锁，可以直接继续执行同步代码块
* 如果不等，则证明该对象目前偏向于其他线程，需要撤销偏向锁

### 4、偏向锁的撤销

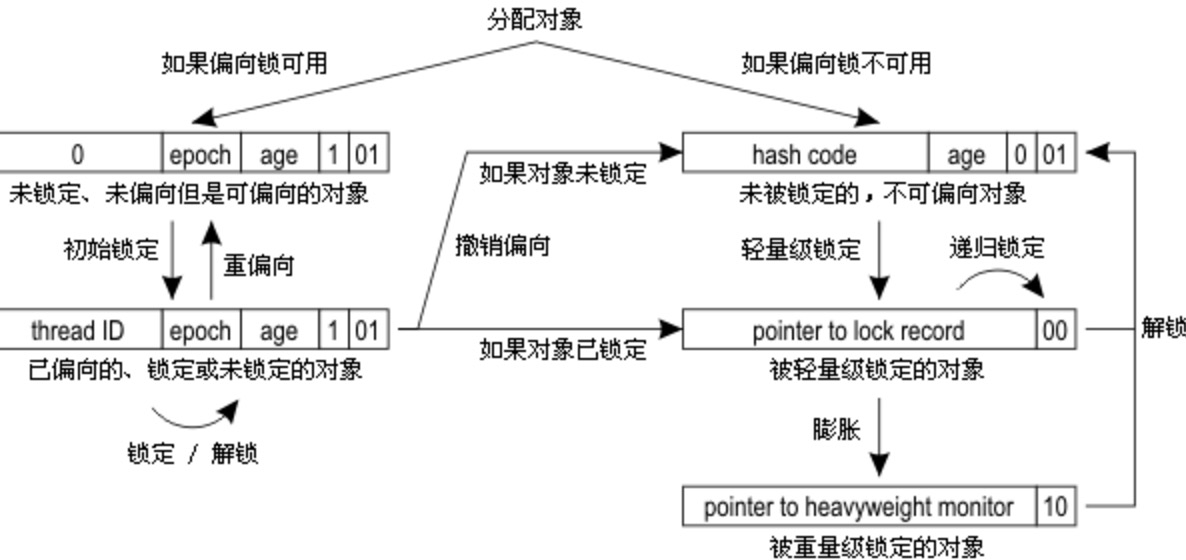
偏向锁的撤销并不是将对象恢复到无锁可偏向的状态，而是在偏向锁的获取过程中，发现竞争时，直接将一个被偏向的对象升级到被加了轻量级锁的状态

* 在偏向锁CAS更新操作失败以后，等待到达全局安全点

● 通过Mark Word中已经存在的Thread ID找到成功获取了偏向锁的那个进程，然后在该线程的栈帧中补充上轻量级加锁时，会保存的锁记录，然后将被获取了偏向锁的对象的Mark Word更新为指向这条锁记录的指针。

● 至此，锁撤销操作完成，阻塞在安全点的线程可以继续执行。

### 5、锁的相互转换



## 八、其他优化

### 1、自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

缺点：自旋需要消耗CPU（单处理器情况下，完全是浪费），最好指定自旋次数或使用jdk的适应性自旋：线程如果成功了，就增加下次的自旋次数，失败则减少自旋次数。

### 2、锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

### 3、锁粗化

将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。

### 九、总结

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗。 | 适用于只有一个线程访问同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU。 | 追求响应时间。  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | 线程阻塞，响应时间缓慢。 | 追求吞吐量。  同步块执行速度较长。 |

java.util.concurrent.Lock 通过 Java 代码搭配sun.misc.Unsafe 中的本地调用实现的