一：java多线程互斥，和java多线程引入偏向锁和轻量级锁的原因？

--->synchronized的重量级别的锁，就是在线程运行到该代码块的时候，让程序的运行级别从用户态切换到内核态，把所有的线程挂起，让cpu通过操作系统指令，去调度多线程之间，谁执行代码块，谁进入阻塞状态。这样会频繁出现程序运行状态的切换，线程的挂起和唤醒，这样就会大量消耗资源，程序运行的效率低下。

为了提高效率，jvm的开发人员，引入了偏向锁，和轻量级锁，尽量让多线程访问公共资源的时候，不进行程序运行状态的切换。

--->jvm规范中可以看到synchronized在jvm里实现原理，jvm基于进入和退出Monitor对象来实现方法同步和代码块同步的。在代码同步的开始位置织入monitorenter,在结束同步的位置（正常结束和异常结束处）织入monitorexit指令实现。线程执行到monitorenter处，将会获取锁对象对应的monitor的所有权，即尝试获得对象的锁。（任意对象都有一个monitor与之关联，当且一个monitor被持有后，他处于锁定状态）

--->java的多线程安全是基于lock机制实现的，而lock的性能往往不如人意。原因是，monitorenter与monitorexit这两个控制多线程同步的bytecode原语，是jvm依赖操作系统互斥（mutex）来实现的。

--->互斥是一种会导致线程挂起，并在较短时间内又需要重新调度回原线程的，较为消耗资源的操作。

--->为了优化java的Lock机制，从java6开始引入轻量级锁的概念。轻量级锁本意是为了减少多线程进入互斥的几率，并不是要替代互斥。它利用了cpu原语**Compare-And-Swap（cas,汇编指令CMPXCHG）,尝试进入互斥前，进行补救。**

二：为什么要自旋或者自适应自旋？

--->前面我们讨论互斥同步的时候，提到了互斥同步对性能最大的影响是阻塞的实现，**挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性能 带来了很大的压力。**同时，虚拟机的开发团队也注意到在许多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如 果物理机器有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一会”，但不放弃处理器的执行时间，看看持有 锁的线程是否很快就会释放锁。为了让**线程等待，我们只须让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。**

--->自旋锁在JDK 1.4.2中就已经引入，只不过默认是关闭的，可以使用-XX:+UseSpinning参数来开启，在JDK 1.6中就已经改为默认开启了。**自旋等待不能代替阻塞**，且先不说对处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但它是要占用处理器时间的， 所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作， 反而会带来性能的浪费。因此自旋等待的时间必须要有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了。自旋次 数的默认值是10次，用户可以使用参数-XX:PreBlockSpin来更改。

--->在JDK 1.6中引入了**自适应的自旋锁**。自适应意味着自旋的时间不再固定了，而是由**前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定**。如果在同一个锁对象 上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间， 比如100个循环。另一方面，如果对于某个锁，**自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程**，以避免浪费处理器资源。有了自适应自 旋，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测就会越来越准确，虚拟机就会变得越来越“聪明”了。

三：锁削除

--->锁削除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是**被检测到不可能存在共享数据竞争的锁进行削除。**锁削除的主要判定依据来源于逃逸分析的数 据支持（第11章已经讲解过逃逸分析技术），如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当作栈上数据对待， 认为它们是线程私有的，同步加锁自然就无须进行。

--->也许读者会有疑问，变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是程序员自己应该是很清楚的，怎么会在明知道不存在数据争用的 情况下要求同步呢？答案是有许多同步措施并不是程序员自己加入的，同步的代码在Java程序中的普遍程度也许超过了大部分读者的想象。比如：（只是说明概念，但实际情况并不一定如例子）**在线程安全的环境中使用stringBuffer进行字符串拼加。则会在java文件编译的时候，进行锁销除。**

四：锁粗化

--->原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待锁的线程也能尽快地拿到锁。

--->大部分情况下，上面的原则都是正确的，但是**如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。**

--->如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将**会把加锁同步的范围扩展（锁粗化）到整个操作序列的外部。**

五：偏向锁，轻量级锁，重量级锁对比

|  |  |  |  |
| --- | --- | --- | --- |
| 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，和执行非同步方法相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 | **追求响应速度，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，同步块执行速度较慢** |

对象头的存储内容（monitor）**（有锁的对象。）**

|  |  |  |
| --- | --- | --- |
| 长度 | 内容 | 说明 |
| 32/64bit | Mark Word | **存储对象的hashcode或锁信息** |
| 32/64bit | 类对象的地址 | 存储到对象类型数据的指针 |
| 32/64bit | Array length | 数组的长度（如果当前对象是数组） |

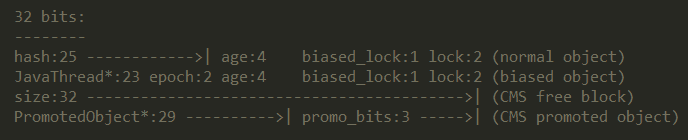
其中Mark Word存储内容如下



**markOop实现**

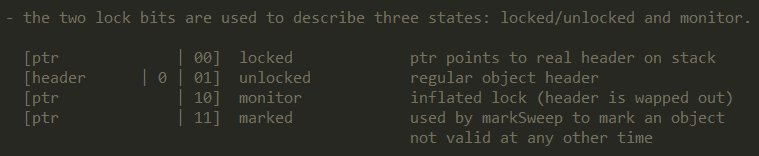
HotSpot通过markOop类型实现Mark Word，具体实现位于markOop.hpp文件中。

由于对象需要存储的运行时数据很多，考虑到虚拟机的内存使用，markOop被设计成一个非固定的数据结构，以便在极小的空间存储尽量多的数据，根据对象的状态复用自己的存储空间，32位虚拟机的markOop实现如下：



* hash： 保存对象的哈希码
* age： 保存对象的分代年龄
* biased\_lock： 偏向锁标识位
* lock： 锁状态标识位
* JavaThread\*： 保存持有偏向锁的线程ID
* epoch： 保存偏向时间戳

markOop中不同的锁标识位，代表着不同的锁状态：



六：锁的状态

--->锁一共有四种状态（由低到高的次序）：**无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态**

--->锁的等级只可以升级，不可以降级。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。

七：偏向锁（第一次对象用CAS操作，之后都不需要。直到有线程竞争，升级）

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，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在**对象头和栈帧中的锁记录里存储锁偏向的线程ID**，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，**只需简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。(即偏向锁的线程ID是当前线程的)**

**如果测试失败，则需要再测试一下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁）：**

**如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程。**

**如果没有设置，则使用CAS竞争锁（则证明此时已经不是偏向锁了，那么就用CAS去竞争锁，此时已经是轻量级以上的锁了）；**

--->a线程获得锁，会在**a线程的的栈帧里**创建lockRecord，在lockRecord里和锁对象的MarkWord里存储线程a的线程id.以后该线程的进入，就不需要cas操作，只需要判断是否是当前线程。

--->a线程获取锁，不会释放锁。直到b线程也要竞争该锁时，a线程才会释放锁。

--->偏向锁的**释放**，**（其实就是线程b要操作的时候，看是否可以释放掉a线程的偏向锁）**需要等待全局安全点（在这个时间点上没有正在执行的字节码），它会首先暂停拥有偏向锁的线程**(达到安全点再暂停阿~)**，然后**检查持有偏向锁的线程是否还活着**，如果线程不处于活动状态，则将锁对象的MarkWord设置成无锁状态，**再指向b线程**。**如果线程仍然活着，拥有偏向锁的栈会被执行。线程a不需要用到该偏向锁了，则恢复到无锁，如果还要用，则和b产生竞争，标记对象不适合作为偏向锁。最后唤醒暂停的线程。**

--->关闭偏向锁，通过jvm的参数-XX:UseBiasedLocking=false,则默认会进入轻量级锁。

偏向锁升级：一个对象刚开始实例化的时候，没有任何线程来访问它的时候。它是可偏向的，意味着，**它现在认为只可能有一个线程来访问它，所以当第一个线程来访问它的时候，它会偏向这个线程，此时，对象持有偏向锁。**偏向第一个线程，这个线程在修改对象头MarkWord成为偏向锁的时候使用CAS操作，并将对象头中的ThreadID改成自己的ID，之后再次访问这个对象时，只需要对比ID，不需要再使用CAS在进行操作。

一旦有第二个线程访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象时偏向状态，这时表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活。如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程，如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁（偏向锁就是这个时候升级为轻量级锁的）。

如果不存在使用了，则可以将对象回复成无锁状态，然后重新偏向。

再来一次：

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为**轻量级锁的获取及释放依赖多次CAS原子指令**，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，**轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。**

1、偏向锁获取过程：

　　（1）访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——**确认为可偏向状态**。

　　（2）如果为可偏向状态，则**测试线程ID是否指向当前线程**，如果是，进入步骤（5），否则进入步骤（3）。

　　（3）如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。

　　（4）如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，**偏向锁升级为轻量级锁**，然后被阻塞在安全点的线程继续往下执行同步代码。

**（5）执行同步代码。**

2、偏向锁的释放：

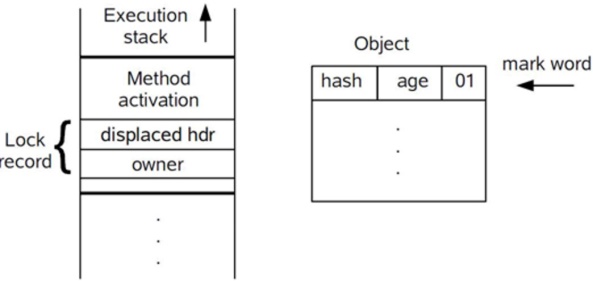
偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

这个得看拥有该偏向锁是否还有需要用，如果该线程已经死了或者没用了，则恢复未锁定，再重新偏向即可，否则，则升级，并且偏向状态为0，此时已经不是偏向锁了。~

**八：轻量级锁**

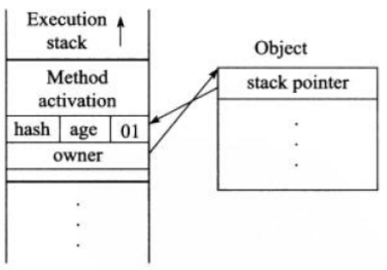
1、轻量级锁的加锁过程(争取线程都可以成功的拿到对象的锁，则不用升级为重量锁)

（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**（锁标志位为“01”状态，是否为偏向锁为“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的状态如图所示。



（2）拷贝对象头中的Mark Word复制到锁记录中。

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



　　（4）如果这个更新动作成功了，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.

（5）如果这个更新操作失败了，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，锁标志的状态值变为“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，而采用循环去获取锁的过程。

### 3.2 解锁

  轻量级锁解锁时，把复制的对象头替换回去（cas）如果替换成功（就是要把无所的状态放回去给对象头，之后锁继续被拿还是轻量级锁，但是如果锁已经是重量级锁了，那么就失败，之后锁就是重量级的锁了），锁结束，之后别的线程来拿还是轻量级锁，如果失败，说明已有竞争，释放锁，此时把对象头设为重量级锁，并notify 唤醒其他等待线程。

**九：重量级锁**

就是让**争抢锁的线程从用户态转换成内核态。**让cpu借助操作系统进行线程协调。

**synchronized的对象锁**，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有**存在多种实现方式**，如**monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成。**

在代码同步的开始位置织入monitorenter,在结束同步的位置（正常结束和异常结束处）织入monitorexit指令实现。线程执行到monitorenter处，将会获取锁对象对应的monitor的所有权，即尝试获得对象的锁。（任意对象都有一个monitor与之关联，当且一个monitor被持有后，他处于锁定状态）

轻量级锁认为竞争存在，但是竞争的程度很轻，一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（自旋），另一个线程就会释放锁。 但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁，重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。

