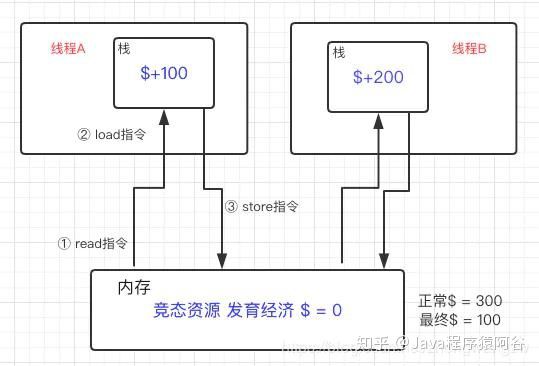
**面试官**: 那你跟我说说为什么会需要synchronized？什么场景下使用synchronized？

**安琪拉**: 这个就要说到多线程访问共享资源了，当一个资源有可能被多个线程同时访问并修改的话，需要用到**锁**，还是画个图给您看一下，请看图：



**安琪拉**: 如上图所示，比如在王者荣耀程序中，我们队有二个线程分别统计后裔和安琪拉的经济，A线程从内存中read 当前队伍总经济加载到线程的本地栈，进行 +100 操作之后，这时候B线程也从内存中取出经济值 + 200，将200写回内存，B线程前脚刚写完，后脚A线程将100 写回到内存中，就出问题了，我们队的经济应该是300， 但是内存中存的却是100，你说糟不糟心。

**面试官**: 那你跟我讲讲用 synchronized 怎么解决这个问题的？

**安琪拉**: 在访问竞态资源时加锁，因为多个线程会修改经济值，因此经济值就是静态资源，给您show 一下吧？下图是不加锁的代码和控制台的输出，请您过目：

二个线程，A线程让队伍经济 +1 ，B线程让经济 + 2，分别执行一千次，正确的结果应该是3000，结果得到的却是 2845。



**安琪拉**: 这个就是加锁之后的代码和控制台的输出。



**面试官**: 我看你用synchronized 锁住的是代码块，synchronized 还有别的**作用范围**吗？

**安琪拉**: 嗯嗯，synchronized 有以下三种作用范围：

1. 在静态方法上加锁；
2. 在非静态方法上加锁；
3. 在代码块上加锁； 示例代码如下

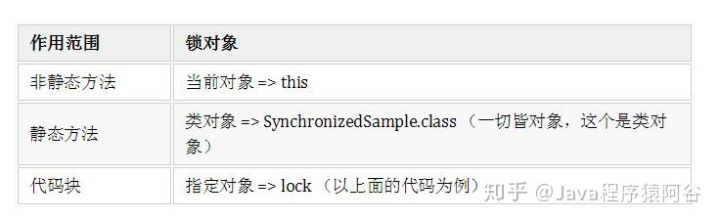


**面试官**: 那你了解 synchronized 这三种作用范围，加锁方式的区别吗？

**安琪拉**: 了解。首先要明确一点：锁是加在**对象**上面的，我们是在**对象**上加锁。

重要事情说三遍：在**对象**上加锁 ✖️ 3 （这也是为什么wait / notify 需要在锁定对象后执行，只有先拿到锁才能释放锁）

这三种作用范围的区别实际是被加锁的对象的区别，请看下表：



**面试官**: 那你清楚 JVM 是怎么通过synchronized 在对象上实现加锁，保证多线程访问竞态资源安全的吗？

**安琪拉**: 【天啦撸, 该来的还是要来】(⊙o⊙)…额，这个说起来有点复杂，我怕时间不够，要不下次再约？

**面试官**: 别下次了，今天我有的是时间，你慢慢讲，我慢慢你说。

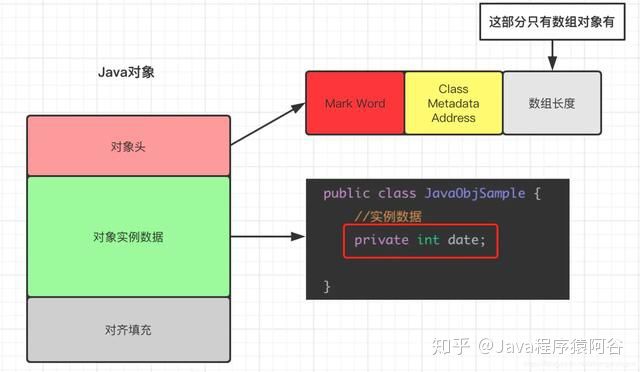
**安琪拉**: 那要跟您好好说道了。分二个时间段来跟您讨论，先说到盘古开天辟地，女娲造石补天，咳咳，不好意思扯远了。。。。。。

1. 先说在JDK6 以前，synchronized 那时还属于重量级锁，相当于关二爷手中的青龙偃月刀，每次加锁都依赖操作系统Mutex Lock实现，涉及到操作系统让线程从用户态切换到内核态，切换成本很高；
2. 到了JDK6，研究人员引入了偏向锁和轻量级锁，因为Sun 程序员发现大部分程序大多数时间都不会发生多个线程同时访问竞态资源的情况，每次线程都加锁解锁，每次这么搞都要操作系统在用户态和内核态之前来回切，太耗性能了。

**面试官**: 那你分别跟我讲讲JDK 6 以前 synchronized为什么这么重？ JDK6 之后又是 偏向锁和轻量级锁又是怎么回事？

**安琪拉**: 好的。首先要了解 synchronized 的实现原理，需要理解二个预备知识：

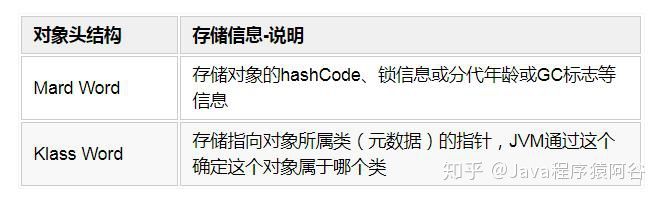
1. 第一个预备知识：需要知道 Java 对象头，锁的类型和状态和对象头的Mark Word息息相关； synchronized 锁 和 对象头息息相关。我们来看下对象的结构：



对象存储在堆中，主要分为三部分内容，对象头、对象实例数据和对齐填充（数组对象多一个区域：记录数组长度），下面简单说一下三部分内容，虽然 synchronized 只与对象头中的 Mard Word相关。

* 对象头：

对象头分为二个部分，Mard Word 和 Klass Word，列出了详细说明：

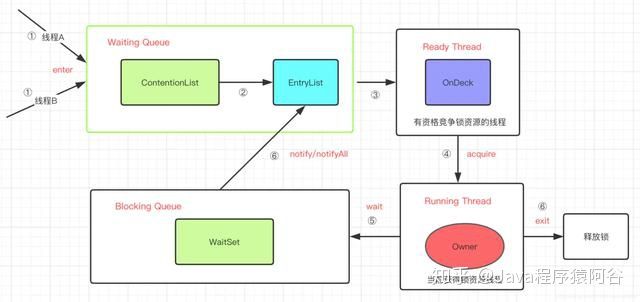


* 对象实例数据：如上图所示，类中的 成员变量data 就属于对象实例数据；
* 对齐填充：JVM要求对象占用的空间必须是8 的倍数，方便内存分配（以字节为最小单位分配），因此这部分就是用于填满不够的空间凑数用的。

2.第二个预备知识：需要了解 Monitor ，每个对象都有一个与之关联的Monitor 对象；Monitor对象属性如下所示( Hospot 1.7 代码) 。



对象关联的 ObjectMonitor 对象有一个线程内部竞争锁的机制，如下图所示：



**面试官**: 预备的二个知识我大体看了，后面给我讲讲 JDK 6 以前 synchronized具体实现逻辑吧。

**安琪拉**: 好的。【开始我的表演】

1. 当有二个线程A、线程B都要开始给我们队的经济 money变量 + 钱，要进行操作的时候 ，发现方法上加了synchronized锁，这时线程调度到A线程执行，A线程就抢先拿到了锁。拿到锁的步骤为： - 1.1 将 MonitorObject 中的 \_owner设置成 A线程； - 1.2 将 mark word 设置为 Monitor 对象地址，锁标志位改为10； - 1.3 将B线程阻塞放到 ContentionList 队列；
2. JVM 每次从Waiting Queue 的尾部取出一个线程放到OnDeck作为候选者，但是如果并发比较高，Waiting Queue会被大量线程执行CAS操作，为了降低对尾部元素的竞争，将Waiting Queue 拆分成ContentionList 和 EntryList 二个队列, JVM将一部分线程移到EntryList 作为准备进OnDeck的预备线程。另外说明几点：

* 所有请求锁的线程首先被放在ContentionList这个竞争队列中;
* Contention List 中那些有资格成为候选资源的线程被移动到 Entry List 中;
* 任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck;
* 当前已经获取到所资源的线程被称为 Owner;
* 处于 ContentionList、EntryList、WaitSet 中的线程都处于阻塞状态，该阻塞是由操作系统来完成的(Linux 内核下采用 pthread\_mutex\_lock 内核函数实现的);

3.作为Owner 的A 线程执行过程中，可能调用wait 释放锁，这个时候A线程进入 Wait Set , 等待被唤醒。

以上就是我想说的 synchronized 在 JDK 6之前的实现原理。

**面试官**: 那你知道 synchronized 是公平锁还是非公平锁吗？

**安琪拉**: 非公平的。主要有以下二点原因：

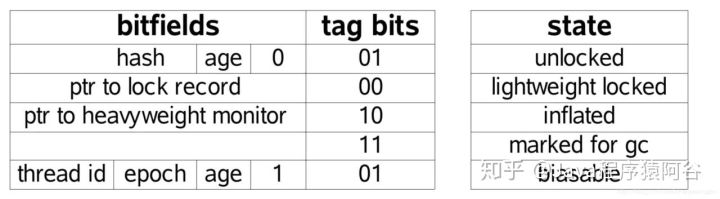
* Synchronized 在线程竞争锁时，首先做的不是直接进ContentionList 队列排队，而是尝试自旋获取锁（可能ContentionList 有别的线程在等锁），如果获取不到才进入 ContentionList，这明显对于已经进入队列的线程是不公平的；
* 另一个不公平的是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁资源。

**面试官**: 你前面说到 JDK 6 之后synchronized 做了优化，跟我讲讲？

**安琪拉**: 不要着急！ 容我点个治疗，再跟你掰扯掰扯。前面说了锁跟对象头的 Mark Word 密切相关，我们把目光放到对象头的 Mark Word 上, Mark Word 存储结构如下图和源代码注释（以32位JVM为例，后面的讨论都基于32位JVM的背景，64位会特殊说明）。 Mard Word会在不同的锁状态下，32位指定区域都有不同的含义，这个是为了节省存储空间，用4 字节就表达了完整的状态信息，当然，对象某一时刻只会是下面5 种状态种的某一种。



下面是简化后的 Mark Word



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

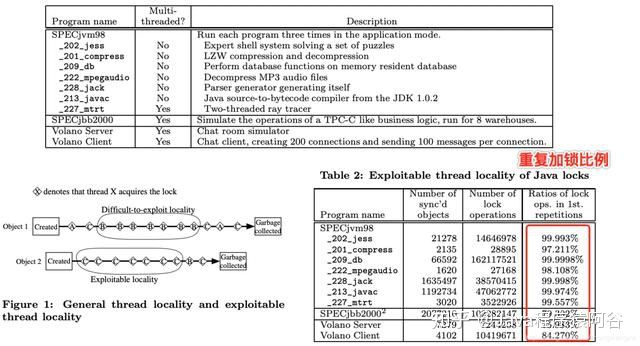
lock： 锁状态标识位

JavaThread\*： 保存持有偏向锁的线程ID

epoch： 保存偏向时间戳

**安琪拉**: 由于 synchronized 重量级锁有以下二个问题, 因此JDK 6 之后做了改进，引入了偏向锁和轻量级锁：

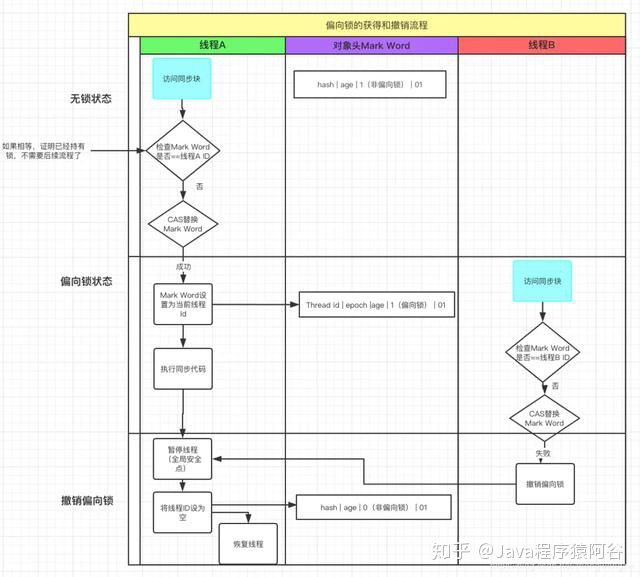
* 依赖底层操作系统的 mutex 相关指令实现，加锁解锁需要在用户态和内核态之间切换，性能损耗非常明显。
* 研究人员发现，大多数对象的加锁和解锁都是在特定的线程中完成。也就是出现线程竞争锁的情况概率比较低。他们做了一个实验，找了一些典型的软件，测试**同一个线程**加锁解锁的重复率，如下图所示，可以看到重复加锁比例非常高。早期JVM 有 19% 的执行时间浪费在锁上。



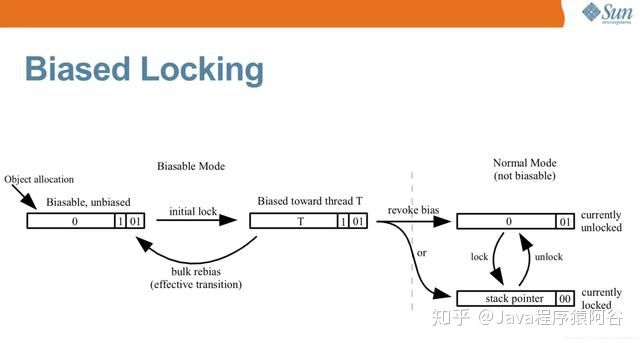
Thin locks are a lot cheaper than inflated locks, but their performance suffers from the fact that every compare-and-swap operation must be executed atomically on multi-processor machines, although most objects are locked and unlocked only by one particular thread.  
It was reported that 19% of the total execution time was wasted by thread synchronization in an early version of Java virtual machine。

**面试官**: 你跟我讲讲 JDK 6 以来 synchronized 锁状态怎么从无锁状态到偏向锁的吗？

**安琪拉**: OK的啦！，我们来看下图对象从无锁到偏向锁转化的过程（JVM -XX:+UseBiasedLocking 开启偏向锁）：

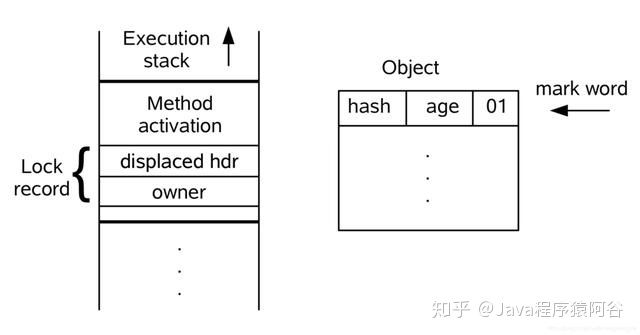


1. 首先A 线程访问同步代码块，使用CAS 操作将 Thread ID 放到 Mark Word 当中；
2. 如果CAS 成功，此时线程A 就获取了锁
3. 如果线程CAS 失败，证明有别的线程持有锁，例如上图的线程B 来CAS 就失败的，这个时候启动偏向锁撤销 （revoke bias）；
4. 锁撤销流程： - 让 A线程在全局安全点阻塞（类似于GC前线程在安全点阻塞） - 遍历线程栈，查看是否有被锁对象的锁记录（ Lock Record），如果有Lock Record，需要修复锁记录和Markword，使其变成无锁状态。 - 恢复A线程 - 将是否为偏向锁状态置为 0 ，开始进行轻量级加锁流程 （后面讲述） 下图说明了 Mark Word 在这个过程中的转化

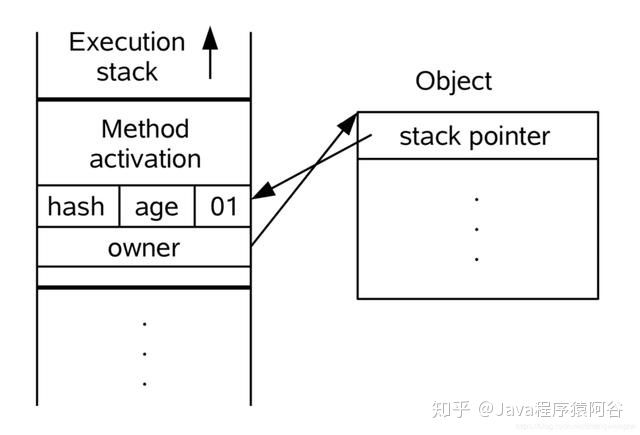


**面试官**: 不错，那你跟我讲讲偏向锁撤销怎么到轻量级锁的？ 还有轻量级锁什么时候会变成重量级锁？ **安琪拉**: 继续上面的流程，锁撤销之后（偏向锁状态为0），现在无论是A线程还是B线程执行到同步代码块进行加锁，流程如下：

* 线程在自己的栈桢中创建锁记录 LockRecord。
* 线程A 将 Mark Word 拷贝到线程栈的 Lock Record中，这个位置叫 displayced hdr，如下图所示：

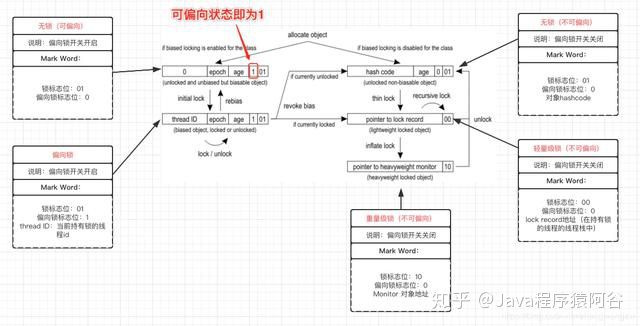


* 将锁记录中的Owner指针指向加锁的对象（存放对象地址）。
* 将锁对象的对象头的MarkWord替换为指向锁记录的指针。这二步如下图所示



5. 这时锁标志位变成 00 ，表示轻量级锁

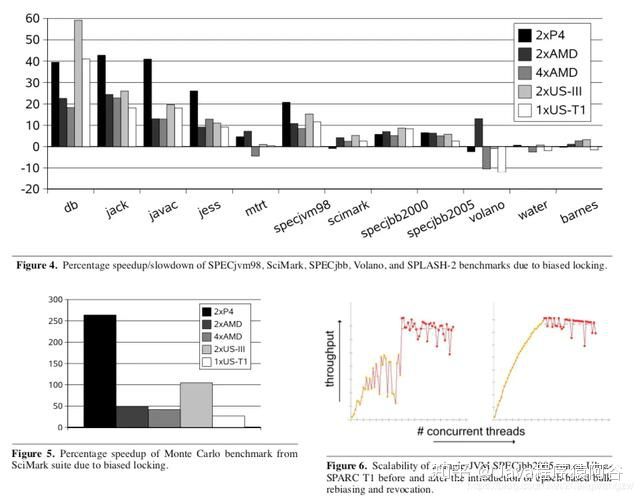
**面试官**: 看来对synchronized 很有研究嘛。我钟馗不信难不倒你，那轻量级锁什么时候会升级为重量级锁, 请回答？ **安琪拉**: 当锁升级为轻量级锁之后，如果依然有新线程过来竞争锁，首先新线程会自旋尝试获取锁，尝试到一定次数（默认10次）依然没有拿到，锁就会升级成重量级锁。 **面试官**: 为什么这么设计？ **安琪拉**: 一般来说，同步代码块内的代码应该很快就执行结束，这时候线程B 自旋一段时间是很容易拿到锁的，但是如果不巧，没拿到，自旋其实就是死循环，很耗CPU的，因此就直接转成重量级锁咯，这样就不用了线程一直自旋了。 这就是锁膨胀的过程，下图是Mark Word 和锁状态的转化图



主要图我标注出来的，锁当前为可偏向状态，偏向锁状态位置就是1，看到很多网上的文章都写错了，把这里写成只有锁发生偏向才会置为1，一定要注意。

**面试官**: 既然偏向锁有撤销，还会膨胀，性能损耗这么大，还需要用他们呢？

**安琪拉**: 如果确定竞态资源会被高并发的访问，建议通过-XX:-UseBiasedLocking 参数关闭偏向锁，偏向锁的好处是并发度很低的情况下，同一个线程获取锁不需要内存拷贝的操作，免去了轻量级锁的在线程栈中建Lock Record，拷贝Mark Down的内容，也免了重量级锁的底层操作系统用户态到内核态的切换，因为前面说了，需要使用系统指令。另外Hotspot 也做了另一项优化，基于锁对象的epoch 批量偏移和批量撤销偏移，这样大大降低了偏向锁的CAS和锁撤销带来的损耗，图是研究人员做的压测：



**安琪拉**: 他们在几款典型软件上做了测试，发现基于epoch 批量撤销偏向锁和批量加偏向锁能大幅提升吞吐量，但是并发量特别大的时候性能就没有什么特别的提升了。

**面试官**：可以可以，那你看过synchronized 底层实现源码没有？

**安琪拉**: 那当然啦，源码是我的二技能，高爆发的伤害能不能打出来就看它了，我们一步一步来。 我们把文章开头的示例代码编译成class 文件，然后通过javap -v SynchronizedSample.class 来看下synchronized 到底在源码层面如何实现的？ 如下图所示：



**安琪拉**: synchronized 在代码块上是通过 monitorenter 和 monitorexit指令实现，在静态方法和 方法上加锁是在方法的flags 中加入 ACC\_SYNCHRONIZED 。JVM 运行方法时检查方法的flags，遇到同步标识开始启动前面的加锁流程，在方法内部遇到monitorenter指令开始加锁。

monitorenter 指令函数源代码在 InterpreterRuntime::monitorenter中

IRT\_ENTRY\_NO\_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread\* thread, BasicObjectLock\* elem))

#ifdef ASSERT

thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);

#endif

if (PrintBiasedLockingStatistics) {

Atomic::inc(BiasedLocking::slow\_path\_entry\_count\_addr());

}

Handle h\_obj(thread, elem->obj());

assert(Universe::heap()->is\_in\_reserved\_or\_null(h\_obj()),

"must be NULL or an object");

//是否开启了偏向锁

if (UseBiasedLocking) {

// 尝试偏向锁

ObjectSynchronizer::fast\_enter(h\_obj, elem->lock(), true, CHECK);

} else {

// 轻量锁逻辑

ObjectSynchronizer::slow\_enter(h\_obj, elem->lock(), CHECK);

}

assert(Universe::heap()->is\_in\_reserved\_or\_null(elem->obj()),

"must be NULL or an object");

#ifdef ASSERT

thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);

#endif

IRT\_END

偏向锁代码

// -----------------------------------------------------------------------------

// Fast Monitor Enter/Exit

// This the fast monitor enter. The interpreter and compiler use

// some assembly copies of this code. Make sure update those code

// if the following function is changed. The implementation is

// extremely sensitive to race condition. Be careful.

void ObjectSynchronizer::fast\_enter(Handle obj, BasicLock\* lock, bool attempt\_rebias, TRAPS) {

//是否使用偏向锁

if (UseBiasedLocking) {

// 如果不在全局安全点

if (!SafepointSynchronize::is\_at\_safepoint()) {

// 获取偏向锁

BiasedLocking::Condition cond = BiasedLocking::revoke\_and\_rebias(obj, attempt\_rebias, THREAD);

if (cond == BiasedLocking::BIAS\_REVOKED\_AND\_REBIASED) {

return;

}

} else {

assert(!attempt\_rebias, "can not rebias toward VM thread");

// 在全局安全点，撤销偏向锁

BiasedLocking::revoke\_at\_safepoint(obj);

}

assert(!obj->mark()->has\_bias\_pattern(), "biases should be revoked by now");

}

// 进轻量级锁流程

slow\_enter (obj, lock, THREAD) ;

}

偏向锁的实现具体代码在 BiasedLocking::revoke\_and\_rebias 中，因为函数非常长，就不贴出来，有兴趣的可以在**Hotspot 1.8-biasedLocking.cpp[2]**去看。 轻量级锁代码流程

void ObjectSynchronizer::slow\_enter(Handle obj, BasicLock\* lock, TRAPS) {

//获取对象的markOop数据mark

markOop mark = obj->mark();

assert(!mark->has\_bias\_pattern(), "should not see bias pattern here");

//判断mark是否为无锁状态 & 不可偏向（锁标识为01，偏向锁标志位为0）

if (mark->is\_neutral()) {

// Anticipate successful CAS -- the ST of the displaced mark must

// be visible <= the ST performed by the CAS.

// 保存Mark 到 线程栈 Lock Record 的displaced\_header中

lock->set\_displaced\_header(mark);

// CAS 将 Mark Down 更新为 指向 lock 对象的指针，成功则获取到锁

if (mark == (markOop) Atomic::cmpxchg\_ptr(lock, obj()->mark\_addr(), mark)) {

TEVENT (slow\_enter: release stacklock) ;

return ;

}

// Fall through to inflate() ...

} else

// 根据对象mark 判断已经有锁 & mark 中指针指的当前线程的Lock Record（当前线程已经获取到了，不必重试获取）

if (mark->has\_locker() && THREAD->is\_lock\_owned((address)mark->locker())) {

assert(lock != mark->locker(), "must not re-lock the same lock");

assert(lock != (BasicLock\*)obj->mark(), "don't relock with same BasicLock");

lock->set\_displaced\_header(NULL);

return;

}

lock->set\_displaced\_header(markOopDesc::unused\_mark());

// 锁膨胀

ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);

做个假设，现在线程A 和B 同时执行到临界区if (mark->is\_neutral())： 1、线程A和B都把Mark Word复制到各自的\_displaced\_header字段，该数据保存在线程的栈帧上，是线程私有的； 2、Atomic::cmpxchg\_ptr 属于原子操作，保障了只有一个线程可以把Mark Word中替换成指向自己线程栈 displaced\_header中的，假设A线程执行成功，相当于A获取到了锁，开始继续执行同步代码块； 3、线程B执行失败，退出临界区，通过ObjectSynchronizer::inflate方法开始膨胀锁；

**面试官**： synchronized 源码这部分可以了，不下去了。你跟我讲讲Java中除了synchronized 还有别的锁吗？

**安琪拉**: 还有ReentrantLock也可以实现加锁。

**面试官**： 那写段代码实现之前加经济的同样效果。

**安琪拉**: coding 如图：

