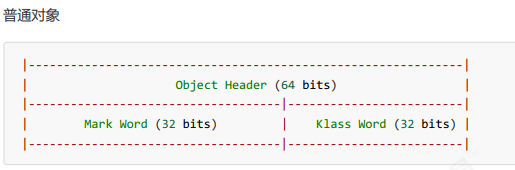
**Synchronized原理-Monitor**

1. Java对象头

Integer： 8+4 字节

int：4字节

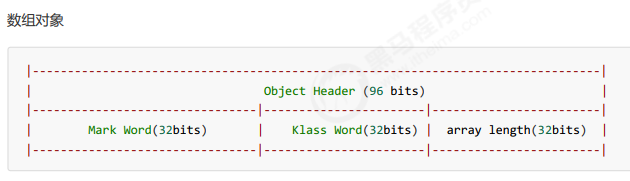
32位虚拟机下：



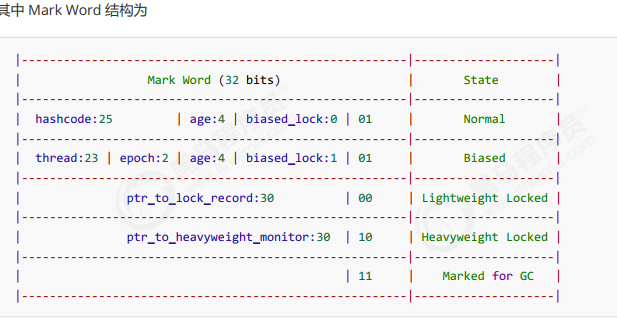
对象头在32位虚拟机下是64位（8字节）

Mark Word（32位）：

Klass Word（32位）：决定对象的类型，通过该指针可以找到类对象。



array length（32）--->数组长度



对Normal来说：

hashCode（25）--->对象的哈希值。

age（4）--->分代年龄：到达一定数值，对象会从新生代移入老年代。

biased\_lock（0）--->不是偏向锁

01--->加锁状态

对Biased（偏向）来说：

thread（23）--->记录线程ID

epoch（2）--->

age（4）--->分代年龄：到达一定数值，对象会从新生代移入老年代。

biased\_lock（1）--->是偏向锁

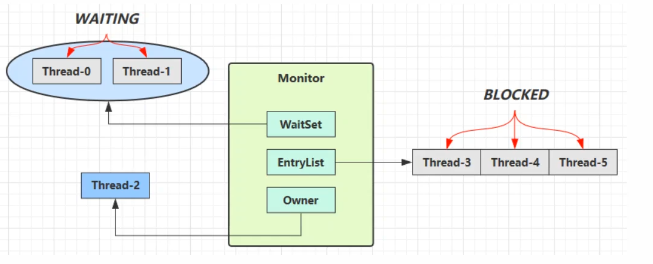
01--->加锁状态

1. Monitor结构

Monitor被翻译成监视器或管程。

每个Java对象都可以关联一个Monitor对象，如果使用synchronized给对象上锁（重量级）之后，该对象头的Mark Word就可以被设置成指向Monitor对象的指针。

Monitor结构如下：



（1）刚开始Monitor中Owner为null；

（2）当Thread-2执行synchronized (obj)就会将Monitor的所有者Owner置为Thread-2，Monitor中只有一个Owner；

（3）在Thread-2上锁过程中，如果Thread-3、Thread-4、Thread-5也来执行synchronized(obj)，就会进入EntryList BLOCKED；

（4）Thread-2执行完同步代码块中的内容，然后唤醒EntryList中等待的线程来竞争锁，竞争是非公平的。

（5）图中的WaitSet中的Thread-0，Thread-1是之前已经获得过锁，但条件不满足进入WAITING状态的线程，后面的wait-notify会分析。

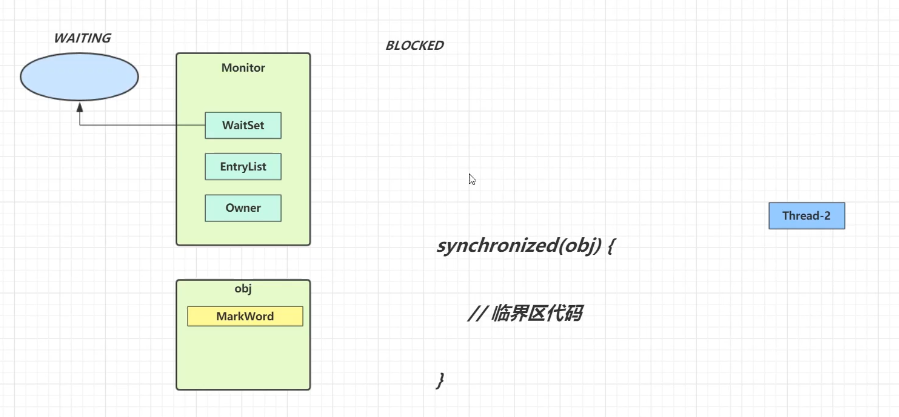
注意：

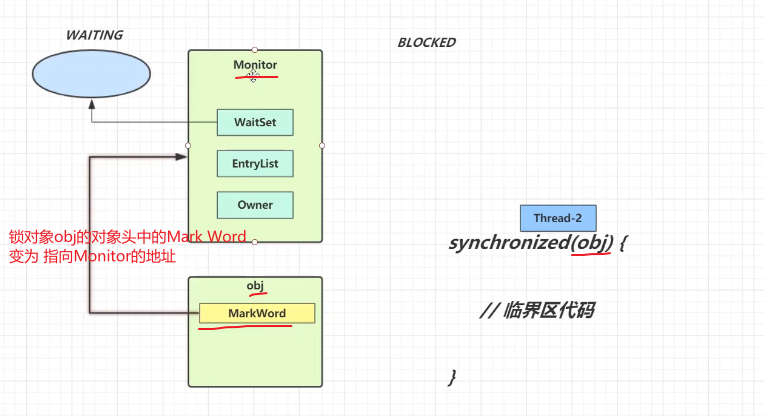
synchronized必须是进入同一个对象的Monitor才有上述效果。

不加synchronized的对象不会关联Monitor，不遵从上述规则。

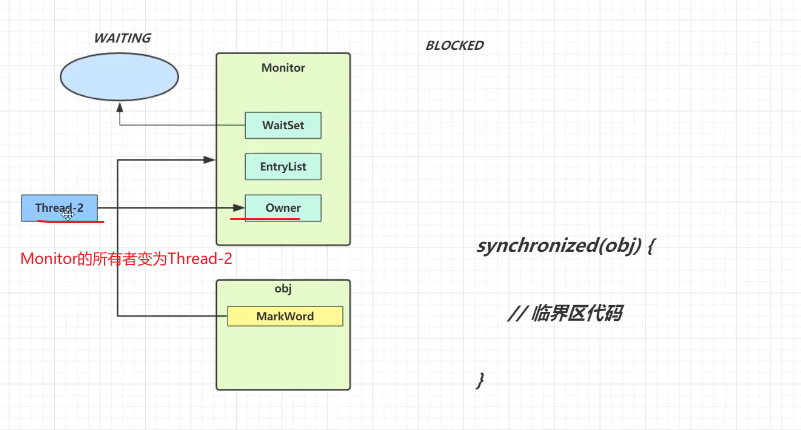
3、Monitor过程分析

【1】开始时，Thread-2竞争到锁，开始执行临界区代码。Monitor的Owner变成Thread-1，锁对象obj的MarkWord变成指向Monitor的地址。

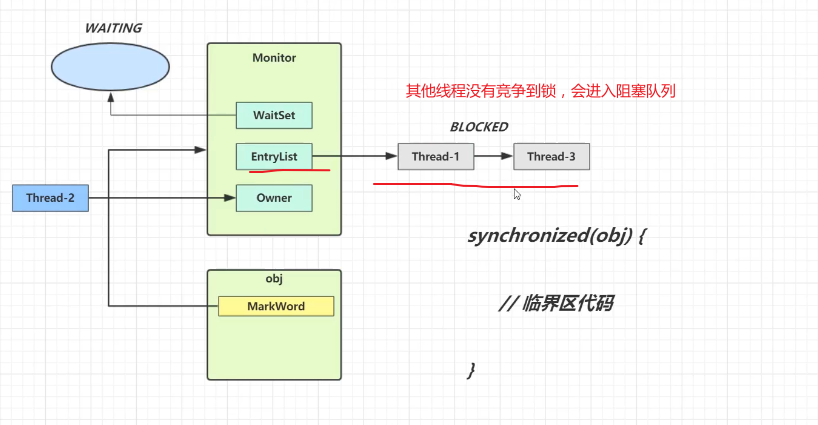




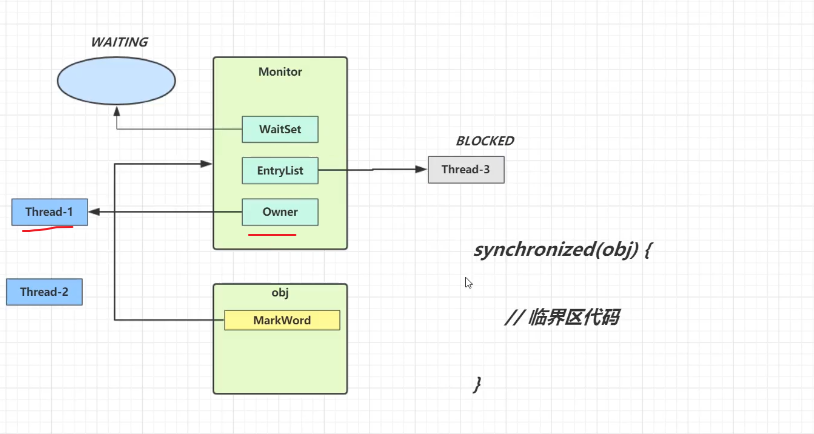




【2】其他线程Thread-1、Thread-3执行临界区代码synchronized(obj)，但是当前Monitor的所有者是Thread-2，Thread-3、Thread-4、Thread-5会进入Monitor的阻塞队列EntryList中，当执行完Thread-2后，操作系统会根据一定的策略在EntryList中选择一个线程作为新的Owner。



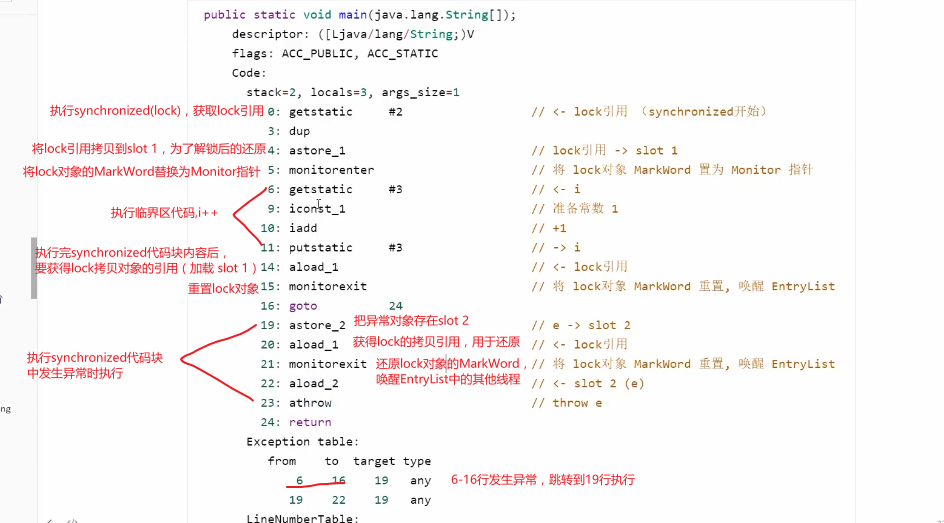
【3】当Thread-2执行完之后，会释放锁。操作系统唤醒Thread-1（不一定按照队列顺序唤醒），Monitor变成Thread-1。



4、从字节码角度理解Monitor



对应的字节码：



Synchronized原理进阶

1. 轻量级锁

轻量级锁的使用场景：

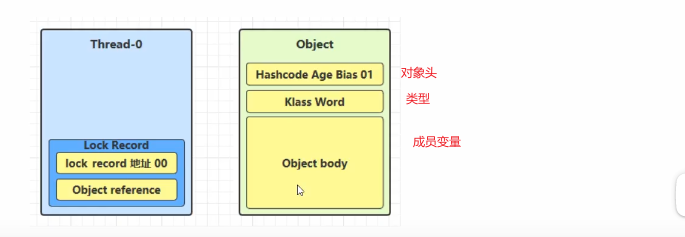
如果一个对象虽然有多个线程访问，但是多线程的访问时间是错开的（也就是没有竞争），那么可以使用轻量级锁优化。

轻量级锁对使用者是透明的，语法仍然是synchronized。

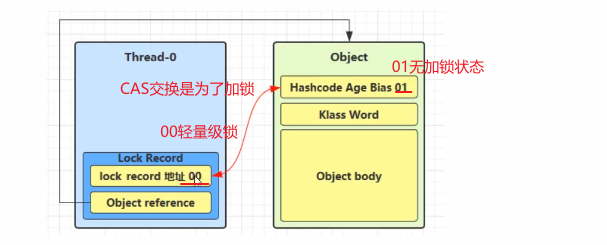
假设有两个方法同步块，利用同一个对象加锁，

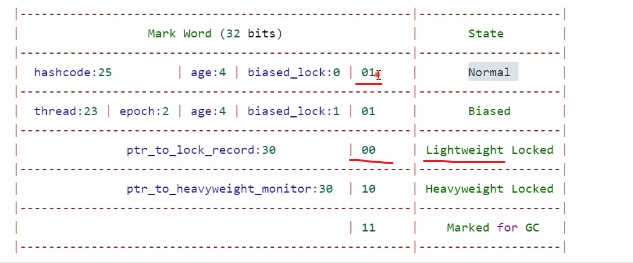
**public class** TestLightWeightLock {  
 **static final** Object ***obj*** = **new** Object();  
  
 **public static void** f1() {  
  
 **synchronized** (***obj***) {  
 *//同步块A  
 f2*();  
 }  
 }  
  
 *//锁重入* **public static void** f2() {  
 **synchronized**(***obj***){  
 *//同步块B* }  
 }  
}

【1】创建锁记录对象（Lock Record），每个线程的栈帧都会包含一个锁记录的结构，内部可以存储锁对象的MarkWord。

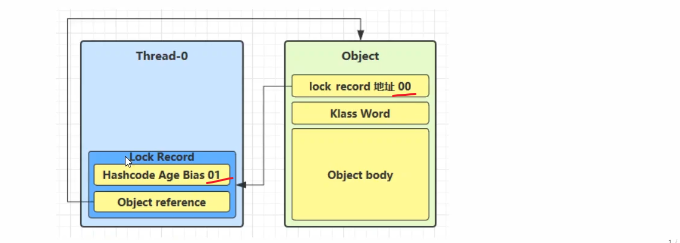


【2】让锁记录的Object reference指向锁对象，并尝试用CAS（Compare And Swap）替换Object的MarkWord，将MarkWord的值存入锁记录。





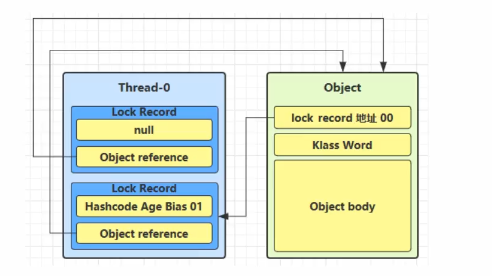
【3.1】如果CAS交换成功（对象头是01），对象头中存储了锁记录地址和状态00，表示由该线程给对象加锁，这时图示如下：



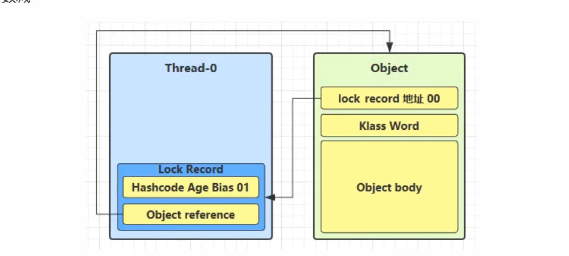
【3.2】如果CAS失败，由两种情况：

如果是其他线程已经持有了该Object的轻量级锁，这时表明有竞争，进入锁膨胀过程。

如果是自己执行了synchronized锁重入，那么再添加一条Lock Record作为重入的计数。



【4.1】当退出synchronized代码块（解锁时），如果有取值为null的锁记录，表示有重入，这时重置锁记录，表示重入计数-1。

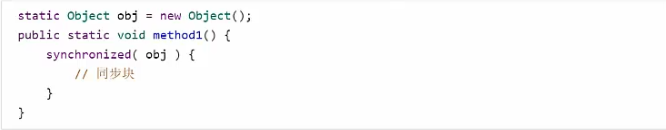


【4.2】当退出synchronized代码块（解锁时），锁记录不为null，这时使用CAS将MarkWord的值恢复给锁对象。

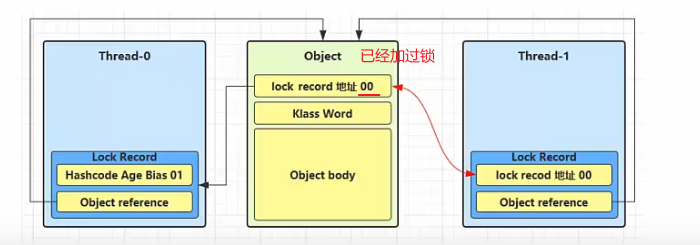
* 成功，则解锁成功。
* 失败，说明轻量级锁进行了锁膨胀或已经升级成了重量级锁，进入重量级锁的解锁流程。

1. 锁膨胀

如果在尝试加轻量级锁的过程中，CAS操作无法成功，这时一种情况是有其他线程为次对象加了轻量级（有竞争），这时需要进行锁膨胀，将轻量级锁变为重量级锁。



【1】当Thread-1进行轻量级加锁时，Thread-0已经对该对象加了轻量级锁。

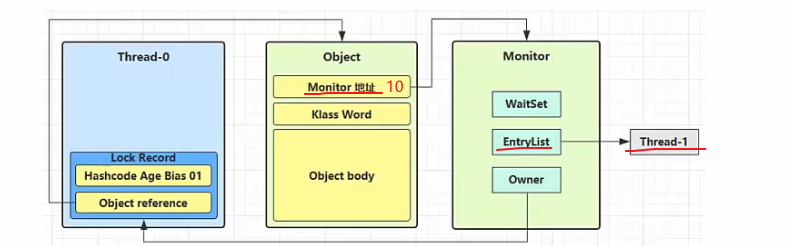


【2】这时Thread-1加轻量级锁失败，进入锁膨胀流程。

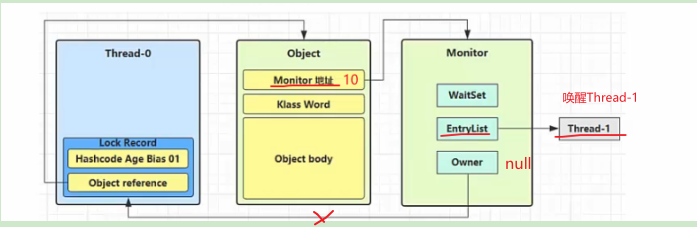
即为Object对象申请Monitor锁（重量级锁），让Object指向重量级锁地址。

Monitor对象的Owner设置为此时拥有锁对象的线程Thread-0。

然后自己进入Monitor对象的EntryList BLOCKED（阻塞队列）。



【3】当Thread-0退出同步块解锁时，使用CAS将Mark Word的值恢复给对象头，此时会失败（因为此时锁对象的MarkWord存放的是Monitor的地址），这时会进入重量级解锁流程，即按照Monitor地址找到Monitor对象，设置Owner为null，唤醒EntyList中的BLOCKED线程。

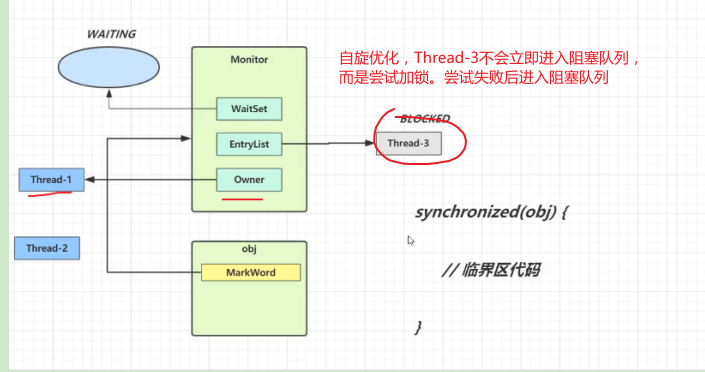


1. 自旋优化

重量级锁竞争的时候，还可以使用自旋来优化，如果当前线程自旋成功（即这个时候持锁线程已经退出了同不块，释放了锁），这时当前线程就可以避免阻塞。

阻塞要发生上下文切换，消耗资源。

适合多核CPU，单核CPU无意义。



自旋重试成功的情况：



自旋失败的情况：



在Java6之后，自旋锁是自适应的，比如对象刚刚的一次自旋操作成功过，那么认为这次自旋成功的可能性很大，会多自旋几次；反之，就少自旋甚至不自旋。

自旋会占用CPU时间，单核CPU自旋就是浪费，多核CPU自旋才能发挥优势。

Java7之后不能控制是否开启自旋功能。

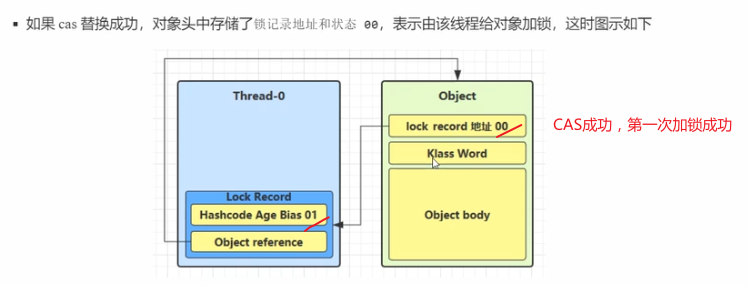
1. 偏向锁

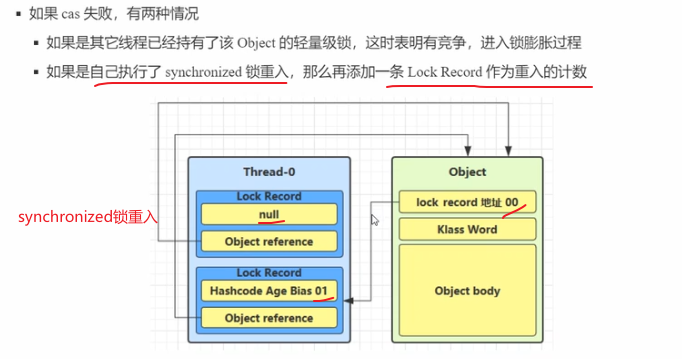
轻量级锁在没有竞争时（就自己这个线程），每次重入仍需要执行CAS操作。

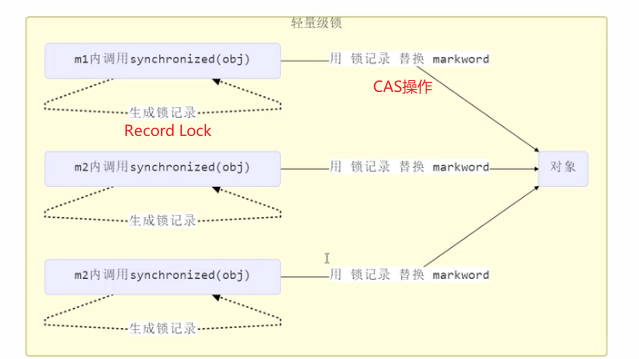
Java6中引入的偏向锁来做进一步的优化：只有第一次使用CAS将线程ID设置到对象的Mark Word头，之后发现这个线程ID是自己的就表示没有竞争，不用重新CAS。只要不发生竞争，这个对象就归该线程所有。

例如：

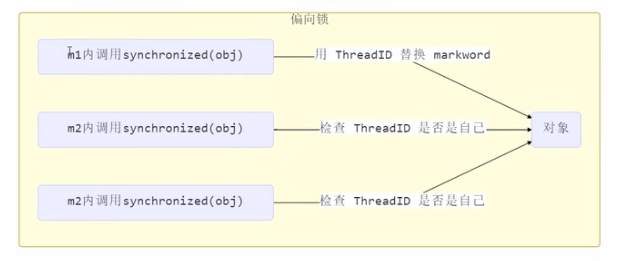
**package** com.concurrent.p2;  
  
*/\*\*  
 \* 偏向锁  
 \*/***public class** TestBiasedLock {  
 **static final** Object ***obj*** = **new** Object();  
  
 **public static void** f1() {  
  
 **synchronized** (***obj***) {  
 *//同步块A  
 f2*();  
 }  
 }  
  
 *//锁重入* **public static void** f2() {  
 **synchronized**(***obj***){  
 *//同步块B  
 f3*();  
 }  
 }  
  
 *//锁重入* **public static void** f3() {  
 **synchronized** (***obj***){  
 *//同步块C* }  
 }  
  
}





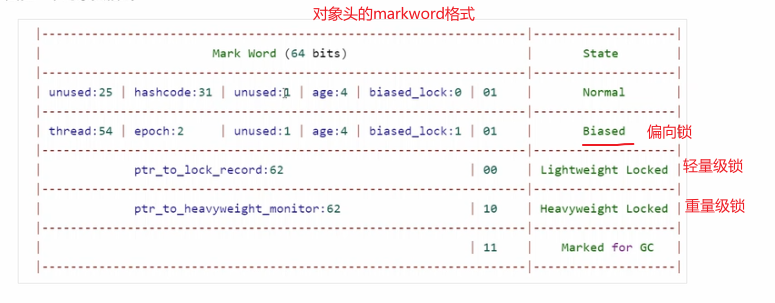


轻量级锁：第一次调用synchronized，会产生一个锁记录RecordLock，使用CAS操作将锁记录与对象的markword替换（对象由01-->00）；之后再次调用synchronized，，再次使用CAS操作尝试替换锁记录和markword，由于第一次已经进行了替换，后续替换不会成功，但是依然会产生一个锁记录，这个锁记录用来计数。



使用偏向锁优化轻量级锁，只有第一次使用CAS将线程ID设置到对象markword头，之后如果线程ID是对象自己的，就不存在竞争，也就不需要进行CAS。只要不发生竞争，这个线程就归对象所有。（线程偏向对象）

**对象头格式MarkWord**



一个对象创建时：

1）如果开启了偏向锁（默认开启），那么对象创建后，markword值为0x05，即最后3位是101，这时它的thread、epoch、age都为0。

2）偏向锁是默认、是延迟的，不会在程序启动时立即生效，如果想避免延迟，可以加JVM参数：

**-XX:BiasedLockingStartupDelay=0**

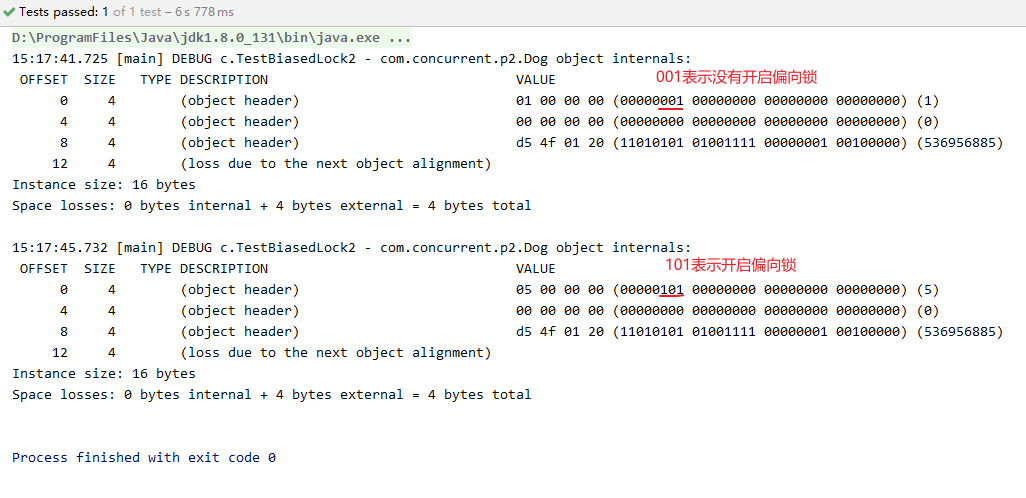
来禁用延迟。

1. 如果没有开启偏向锁，那么对象创建后，markword值为0x01（最后3位是001），这时它的hashcode、age都为0，第一次用到hashcode时才会赋值。

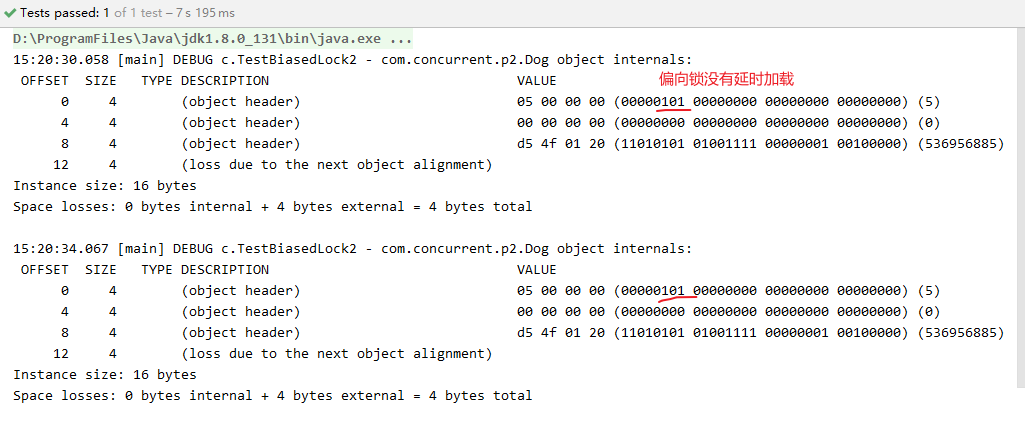
**偏向锁状态**

* 创建Dog对象，查看markword中的偏向锁

*/\*\*  
 \* 创建对象，101表示可以使用偏向锁  
 \* <p>  
 \* 如果要避免延时加载，加JVM命令  
 \* -XX:BiasedLockingStartupDelay=0  
 \*/*@Test  
**public void** t1() **throws** InterruptedException {  
 *//markword* ***log***.debug(ClassLayout.*parseInstance*(**new** Dog()).toPrintable());  
 *//延时4秒* Thread.*sleep*(4000);  
 ***log***.debug(ClassLayout.*parseInstance*(**new** Dog()).toPrintable());  
}

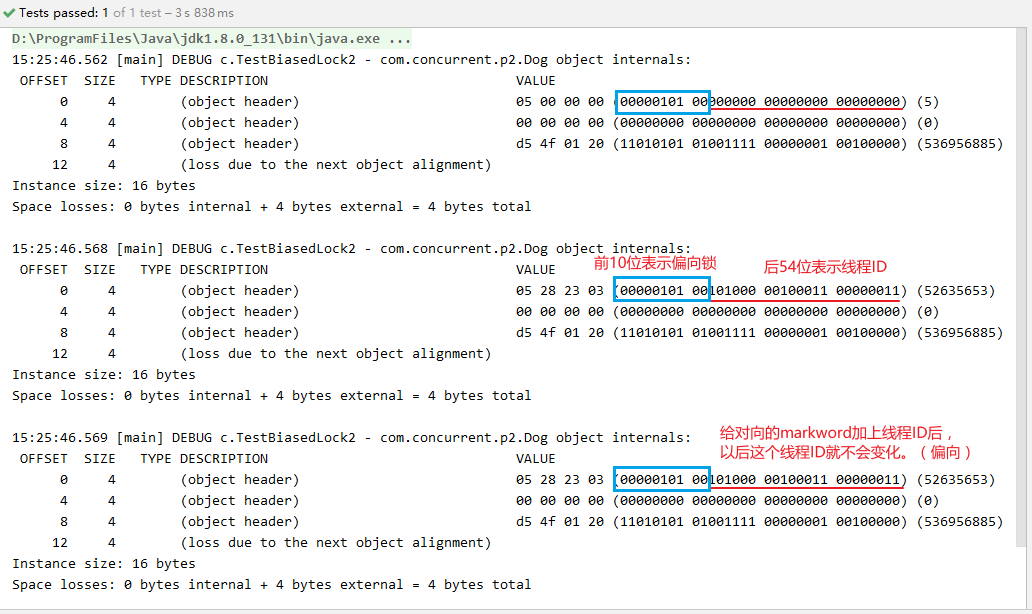


加了JVM参数之后：



* 给对象加上偏向锁

*/\*\*  
 \* 加上偏向锁  
 \*/*@Test  
**public void** t2() {  
 Dog d = **new** Dog();  
 *//加锁前* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 *//加锁中* **synchronized** (d) {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 *//加锁后* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
}



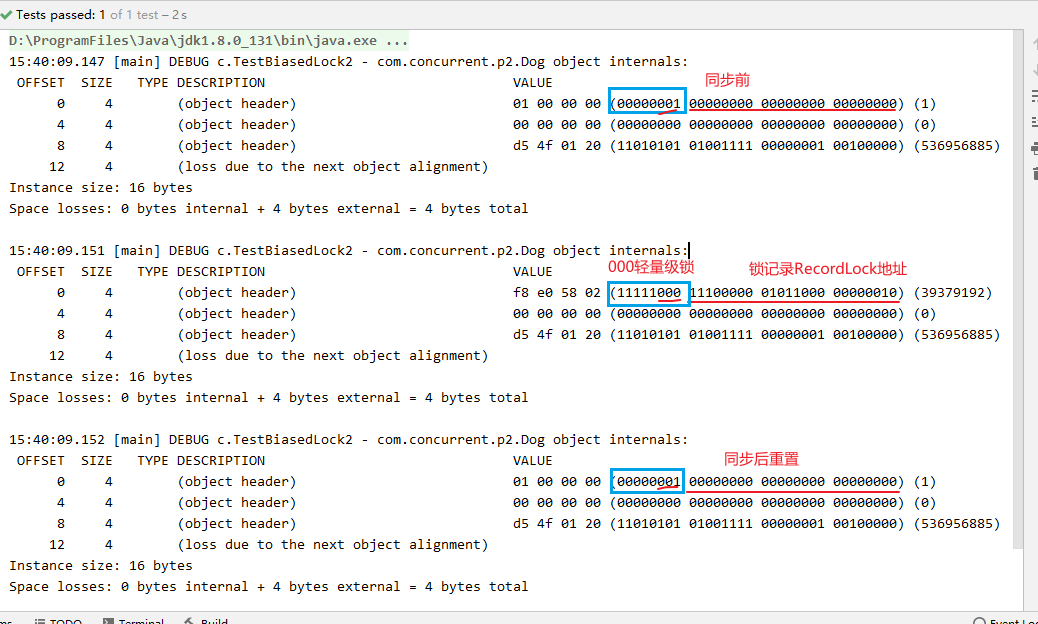
加了偏向锁之后，对象的markword中存放的始终是主线程的ID。

* 禁用偏向锁之后，使用轻量级锁

JVM参数：

**-XX:-UseBiasedLocking**

*/\*\*  
 \* 禁用偏向锁JVM参数  
 \* -XX:-UseBiasedLocking  
 \*/*@Test  
**public void** t3() {  
 Dog d = **new** Dog();  
 *//加锁前* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 *//加锁中* **synchronized** (d) {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 *//加锁后* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
}



**撤销-调用对象的hashCode方法**

调用了对象的hashCode，但偏向锁的对象MarkWord中存储的是线程id，如果调用hashCode会导致偏向锁被撤销。

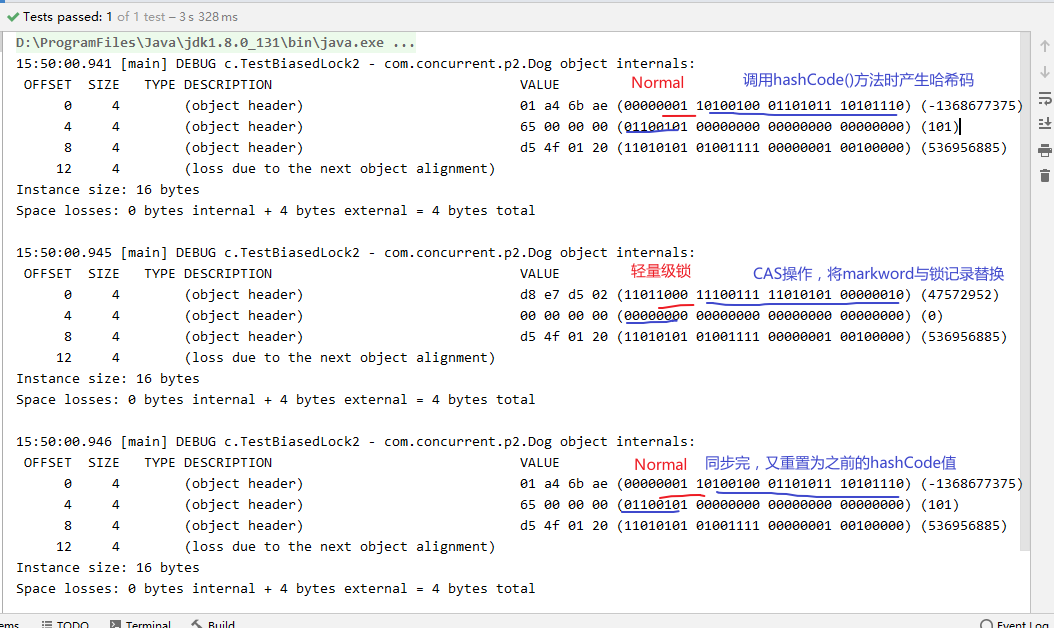
轻量级锁会在锁记录中记录hashCode；

重量级锁会在Monitor中记录hashCode；

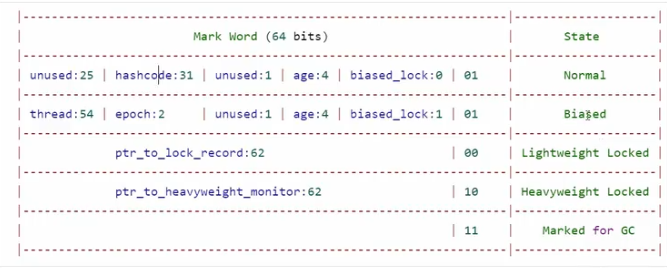
在调用hashCode后使用偏向锁，记得去掉 **-XX:-UseBiasedLocking** （禁用偏向锁的JVM命令）

* 调用对象的hashCode()方法，会禁用偏向锁

*/\*\*  
 \* 同步前调用对象的 hashCode()方法  
 \* <p>  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/*@Test  
**public void** t4() {  
 Dog d = **new** Dog();  
 d.hashCode(); *//会禁用对象的偏向锁  
 //加锁前* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 *//加锁中* **synchronized** (d) {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 *//加锁后* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
}



为什么调用hashCode()方法会禁用偏向锁？



因为在偏向锁的MarkWord中，没有存放31位hashCode值的位置。偏向锁的大部分空间都用来存放线程ID。

在轻量级锁中，对象的hashCode会存放在线程栈帧的锁记录中。

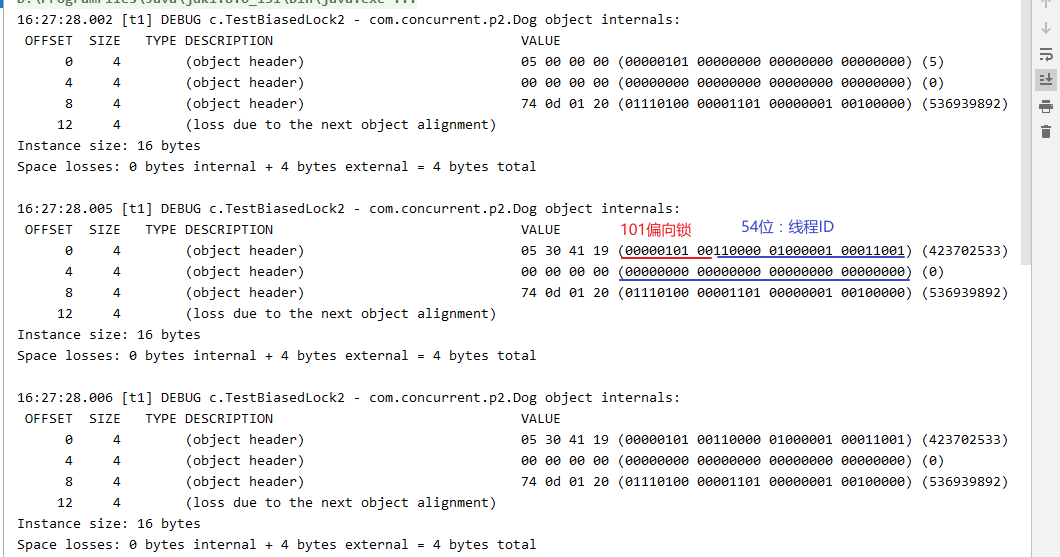
在重量级锁中，对象的hashCode会存放在Monitor对象中，解锁之后可以还原。

**撤销-其他线程使用对象**

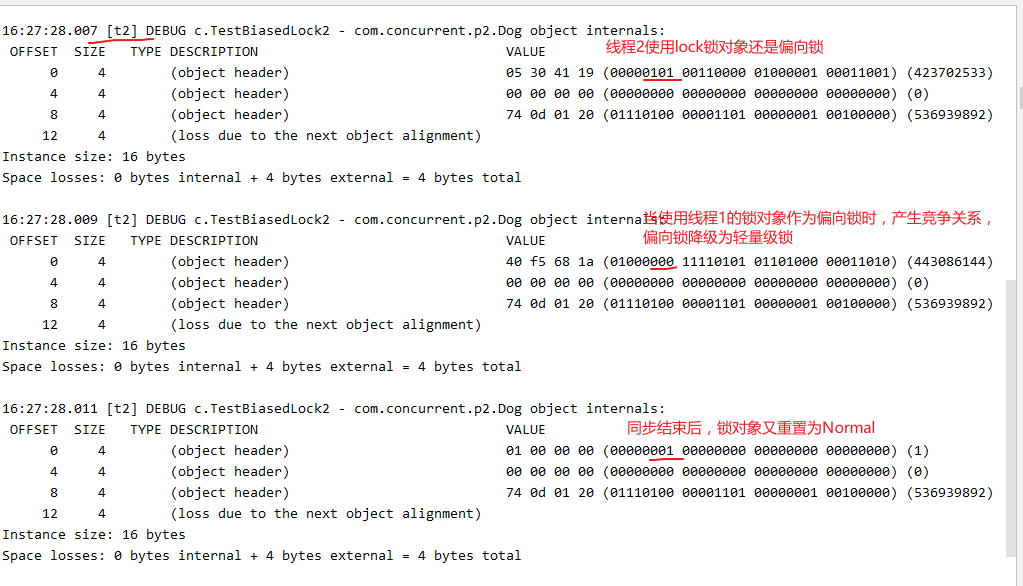
当有其他线程使用偏向锁对象时，会将偏向锁降级为轻量级锁。

*/\*\*  
 \* 撤销-其他线程使用偏向锁对象  
 \* <p>  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/* **static final** Object ***lock*** = **new** Object();  
  
 @Test  
 **public void** t5() **throws** InterruptedException {  
 Dog d = **new** Dog();  
 **new** Thread(() -> {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (d) {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (***lock***) {  
 ***lock***.notify();  
 }  
 }, **"t1"**).start();  
 **new** Thread(() -> {  
 **synchronized** (***lock***) { *//把两个线程分开* **try** {  
 ***lock***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (d) {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }, **"t2"**).start();  
  
 Thread.*sleep*(10000);  
 }  
}

线程1：synchronized(b)，



线程2：由于锁对象已经将线程1的ID记录下来，再一次调用synchronized(b)时，线程2和线程1产生了竞争关系，偏向锁降级为轻量级锁。

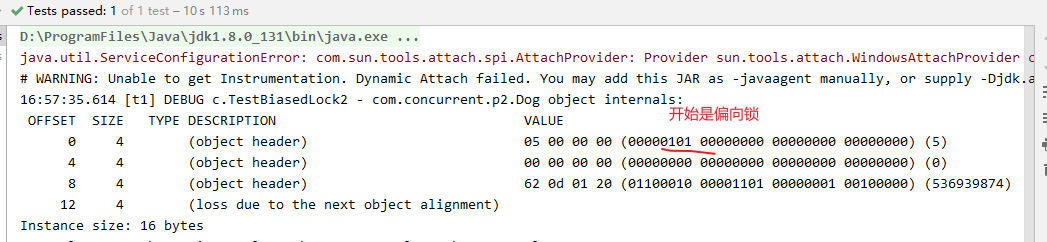


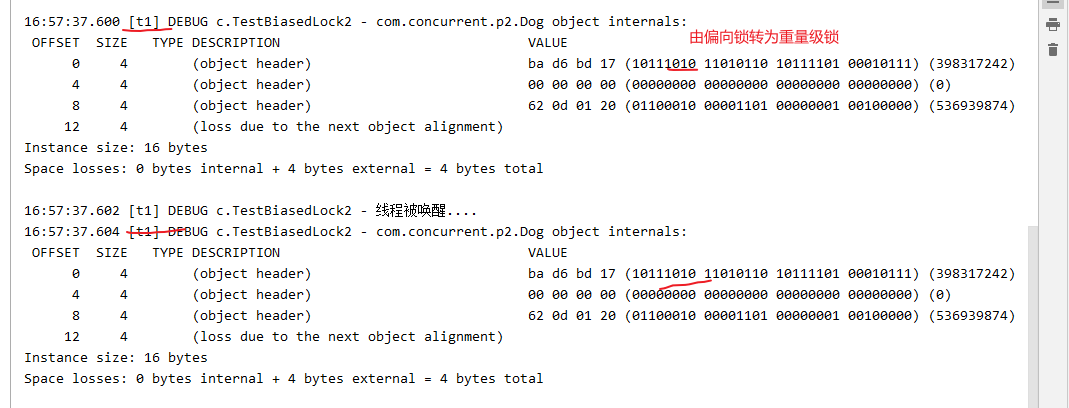
**撤销-调用wait/notify**

只有重量级锁有wai/notify，当调用这些方法时，由偏向锁转为重量级锁。

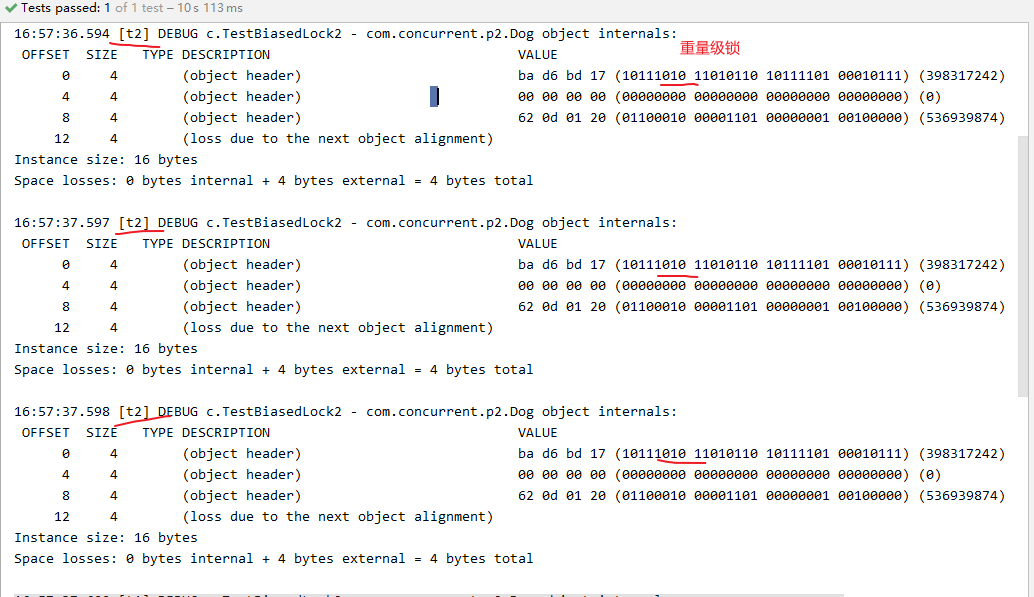
*/\*\*  
 \* 撤销-wait/notify  
 \* 偏向锁转为重量级锁  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/*@Test  
**public void** t6() **throws** InterruptedException {  
 Dog d = **new** Dog();  
 *//线程1* **new** Thread(() -> {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (d) {  
 **try** {  
 d.wait(); *//线程t1等待* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 ***log***.debug(**"线程被唤醒...."**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }, **"t1"**).start();  
 *//线程2* **new** Thread(() -> {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (d) {  
 **try** {  
 Thread.*sleep*(1000); *//1000秒后唤醒线程t1* ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 d.notify();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 }, **"t2"**).start();  
  
 Thread.*sleep*(10000);  
}

线程t1，由偏向锁转为重量级锁：





线程t2，重量级锁：



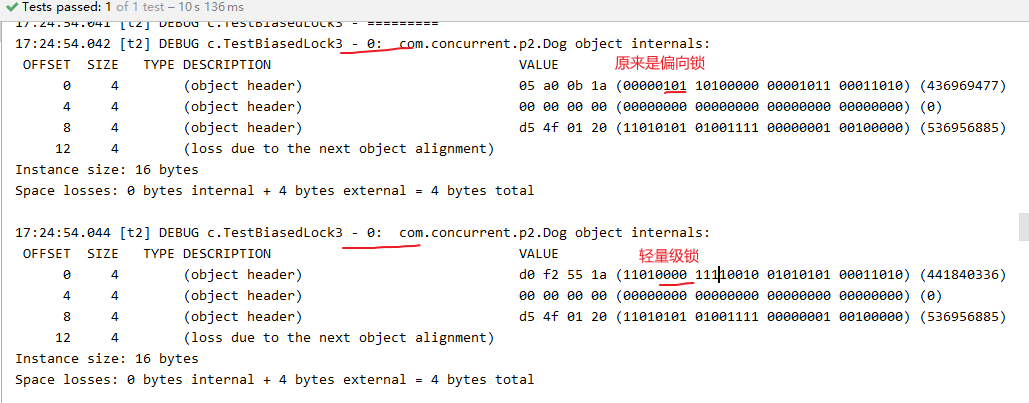
**批量重偏向**

如果对象虽然被多个线程访问，但没有竞争，这时偏向了线程T1的对象仍有可能偏向线程T2，重偏向会重置对象的ThreadID。

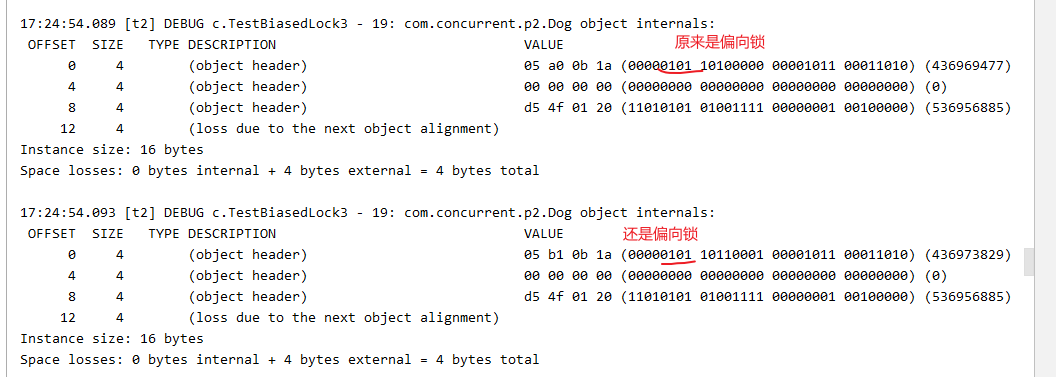
当撤销偏向锁阈值超过20次后，jvm会认为是不是偏向错了，于是会在给这些对象加锁时重新偏向至加锁线程。

*/\*\*  
 \* 批量重偏向  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/*@Test  
**public void** t1() **throws** InterruptedException {  
 Vector<Dog> list = **new** Vector<>();  
 Thread t1 = **new** Thread(() -> {  
 **for** (**int** i = 0; i < 30; i++) {  
 Dog d = **new** Dog();  
 list.add(d);  
 **synchronized** (d) {  
 ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 }  
 **synchronized** (list) {  
 list.notifyAll();  
 }  
 }, **"t1"**);  
 t1.start();  
  
 Thread t2 = **new** Thread(() -> {  
 **synchronized** (list) {  
 **try** {  
 list.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"========="**);  
 **for** (**int** i = 0; i < 30; i++) {  
 Dog d = list.get(i);  
 ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (d) { *//批量重偏向* ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 }  
 }, **"t2"**);  
 t2.start();  
  
 Thread.*sleep*(10000);  
}

i = 0，偏向锁的线程ID为t1：



当到了 i = 19后，并且偏向锁的线程ID变为了t2：



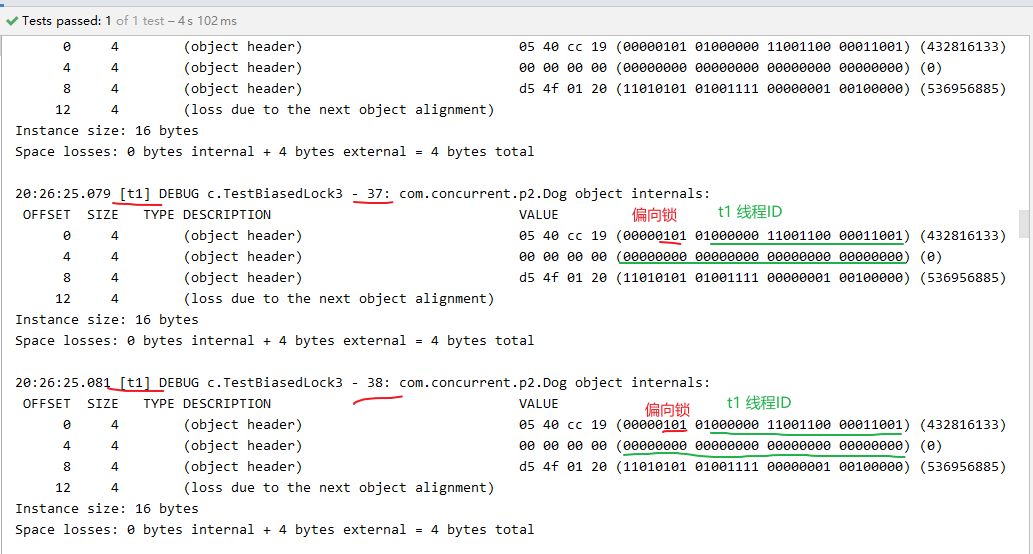
**批量撤销**

当撤销偏向锁阈值超过40次后，jvm会认为自己确实偏向错了，根本不该偏向，于是整个类的所有对象都会变成不可偏向的，新建的对象也是不可偏向的。

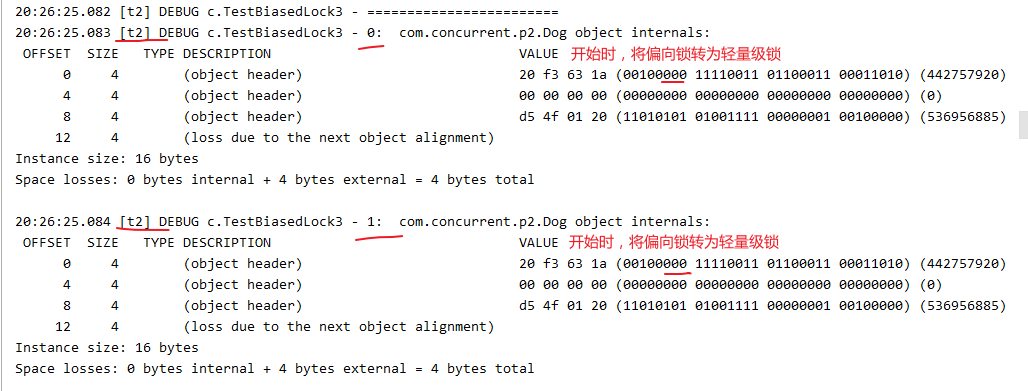
*/\*\*  
 \* 批量撤销  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/***static** Thread *t1*, *t2*, *t3*;  
  
@Test  
**public void** t2() {  
 Vector<Dog> list = **new** Vector<>();  
 **long** loopNumber = 39;  
 *//线程1  
 t1* = **new** Thread(() -> {  
 **for** (**int** i = 0; i < loopNumber; i++) {  
 Dog d = **new** Dog();  
 **synchronized** (d) {  
 ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 list.add(d);  
 }  
 LockSupport.*unpark*(*t2*);  
 }, **"t1"**);  
 *t1*.start();  
 *//线程2  
 t2* = **new** Thread(() -> {  
 LockSupport.*park*();  
 ***log***.debug(**"========================"**);  
 **for** (**int** i = 0; i < loopNumber; i++) {  
 Dog d = list.get(i);  
 **synchronized** (d) {  
 ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 }  
 LockSupport.*unpark*(*t3*);  
 }, **"t2"**);  
 *t2*.start();  
 *//线程3  
 t3* = **new** Thread(() -> {  
 LockSupport.*park*();  
 ***log***.debug(**"========================"**);  
 **for** (**int** i = 0; i < loopNumber; i++) {  
 Dog d = list.get(i);  
 **synchronized** (d) {  
 ***log***.debug(i + **":\t"** + ClassLayout.*parseInstance*(d).toPrintable());  
 }  
 }  
 }, **"t3"**);  
 *t3*.start();  
 **try** {  
 *t3*.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//创建新对象，jvm认为竞争激烈，将整个类的对象都设置成不可偏向* ***log***.debug(ClassLayout.*parseInstance*(**new** Dog()).toPrintable());

}

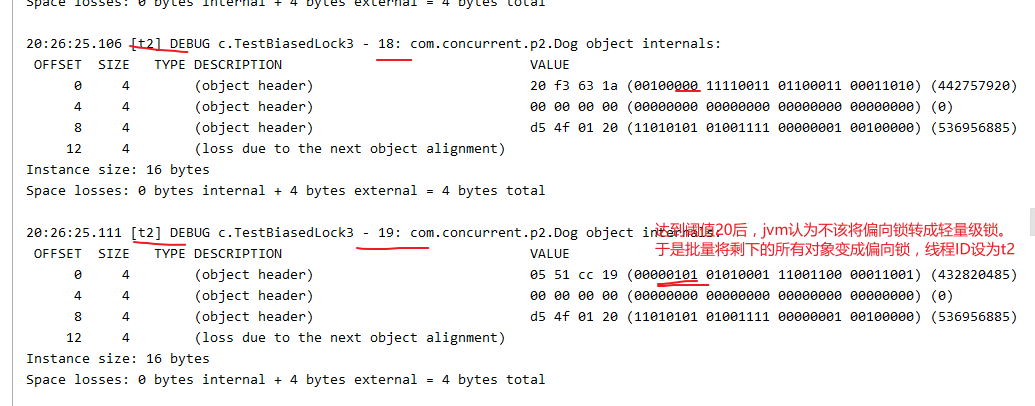
t1线程：锁对象偏向t1线程。



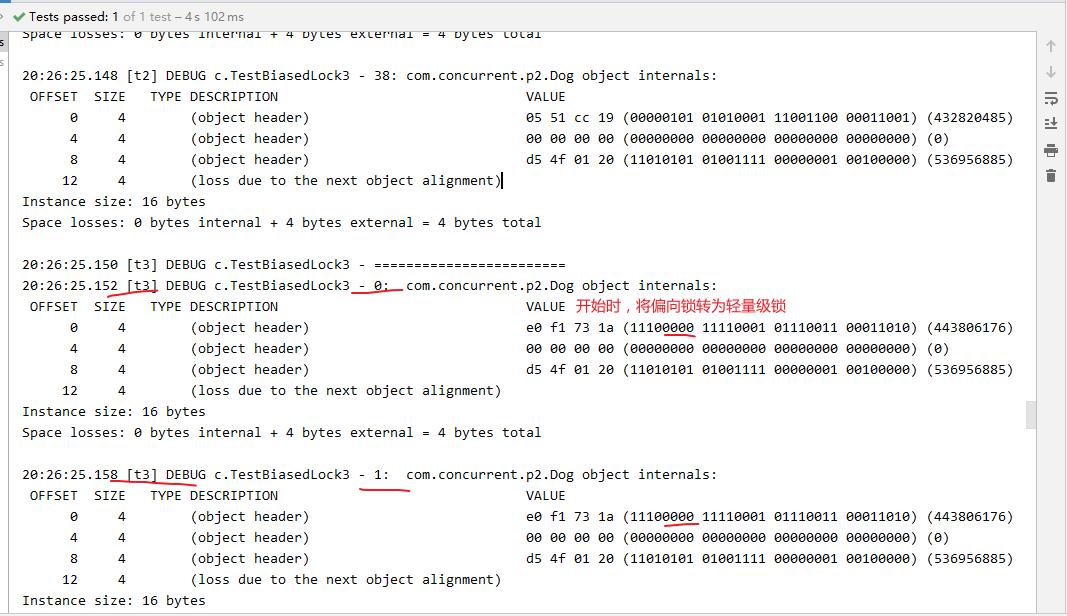
t2线程：一开始一个一个撤销，将锁对象变成轻量级锁。



t2线程：达到20阈值后，开始批量重偏向，锁对象开始重新偏向t2线程（线程ID发生变化）。

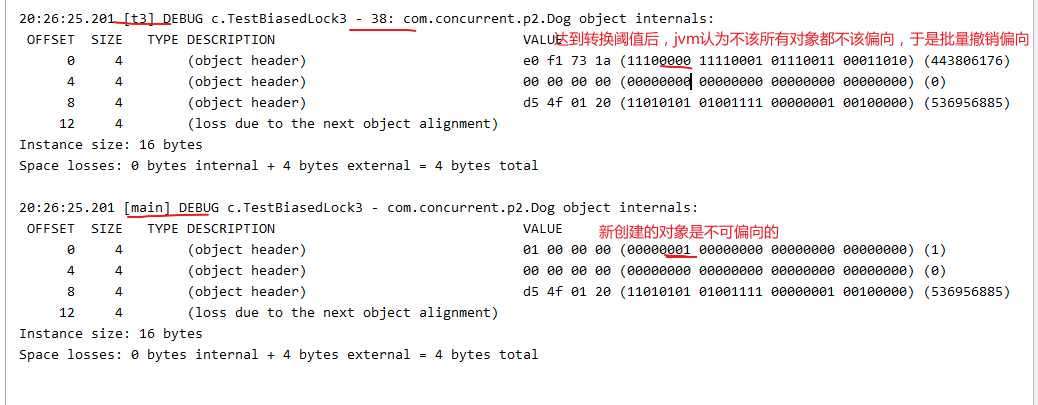


t3线程：前20个都是轻量级锁，不可偏向的。



从第19个开始到38，原来是偏向t2线程的，从t2线程切换到t3线程，变为轻量级锁。解锁之后编程01不可偏向的状态。

从第40个对象开始，就变成了不可偏向的状态。



1. 锁消除

没有加锁消除优化，会导致运行效率低。



