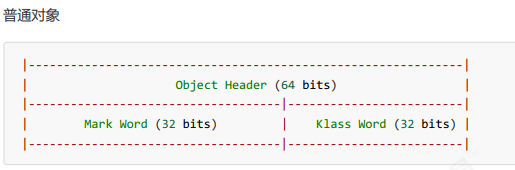
**Synchronized原理-Monitor**

1. Java对象头

Integer： 8+4 字节

int：4字节

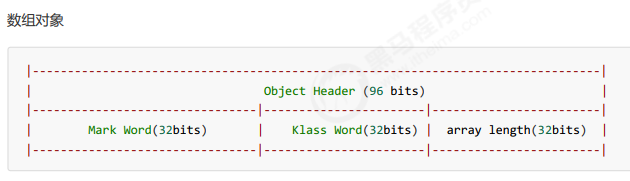
32位虚拟机下：



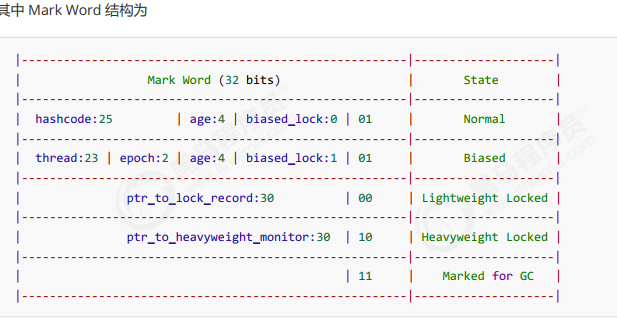
对象头在32位虚拟机下是64位（8字节）

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

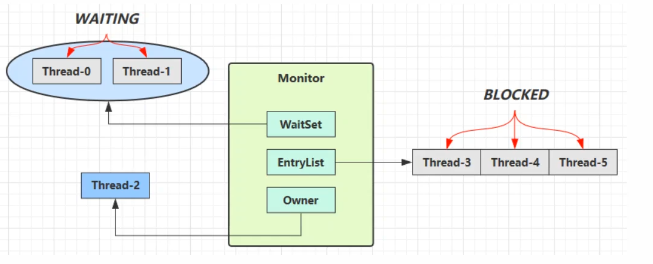
01--->加锁状态

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。

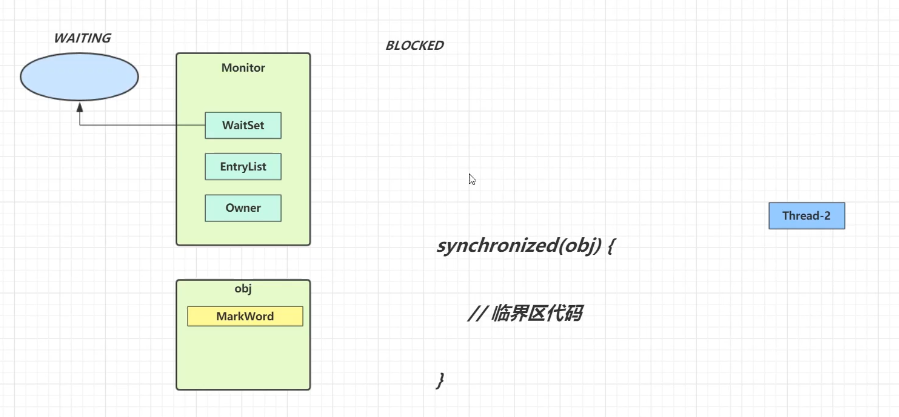
注意：

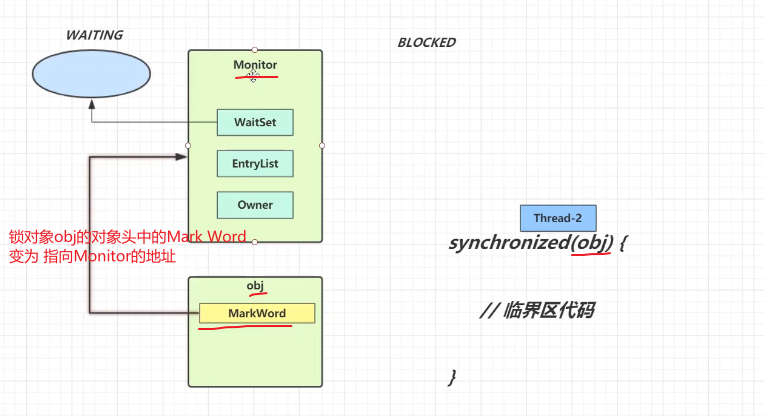
synchronized必须是进入同一个对象的Monitor才有上述效果。

不加synchronized的对象不会关联Monitor，不遵从上述规则。

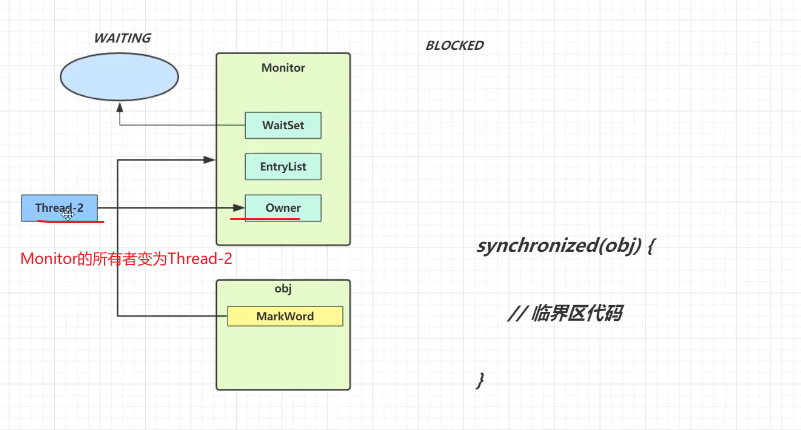
3、Monitor过程分析

【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。

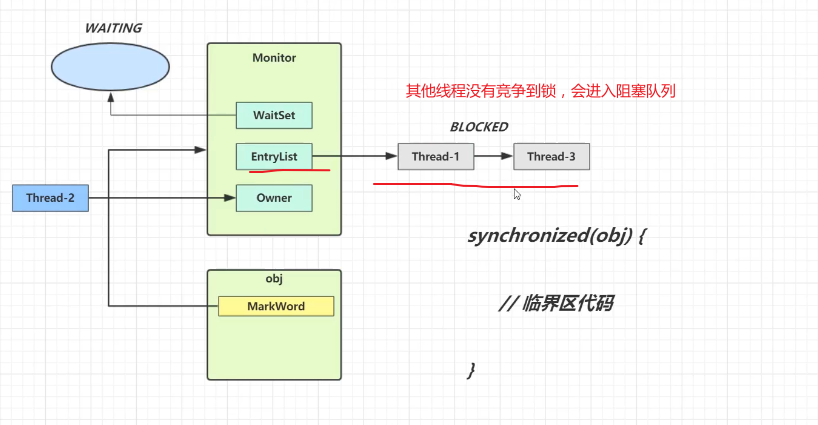




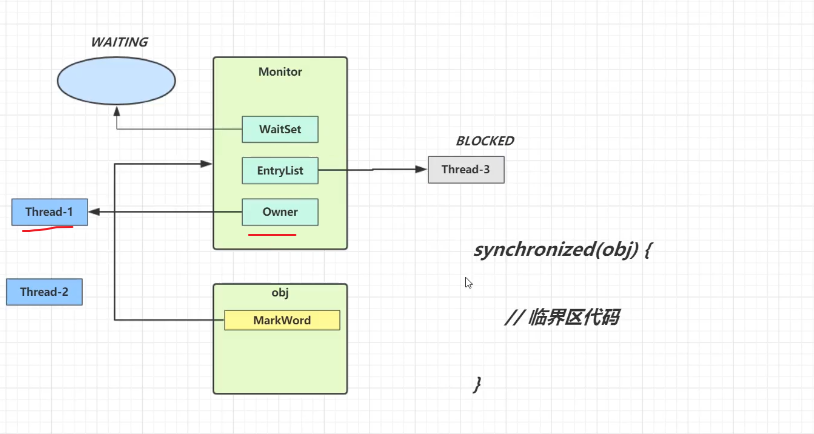




【2】其他线程Thread-1、Thread-3执行临界区代码synchronized(obj)，但是当前Monitor的所有者是Thread-2，Thread-3、Thread-4、Thread-5会进入Monitor的阻塞队列EntryList中，当执行完Thread-2后，操作系统会根据一定的策略在EntryList中选择一个线程作为新的Owner。



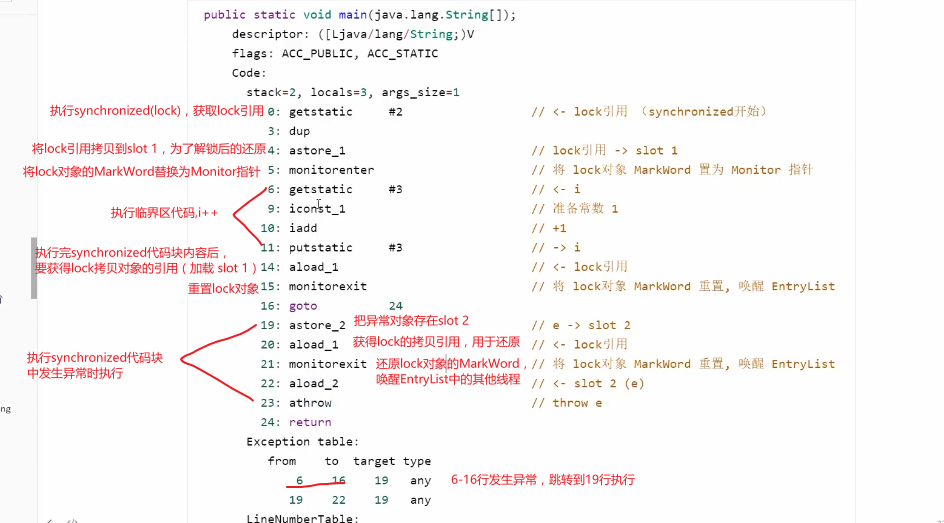
【3】当Thread-2执行完之后，会释放锁。操作系统唤醒Thread-1（不一定按照队列顺序唤醒），Monitor变成Thread-1。



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：

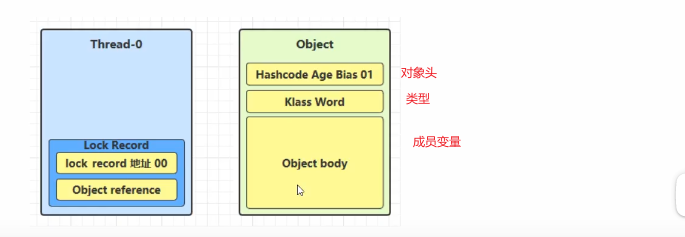
如果一个对象虽然有多个线程访问，但是多线程的访问时间是错开的（也就是没有竞争），那么可以使用轻量级锁优化。

轻量级锁对使用者是透明的，语法仍然是synchronized。

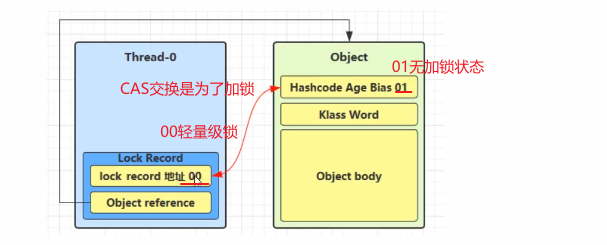
假设有两个方法同步块，利用同一个对象加锁，

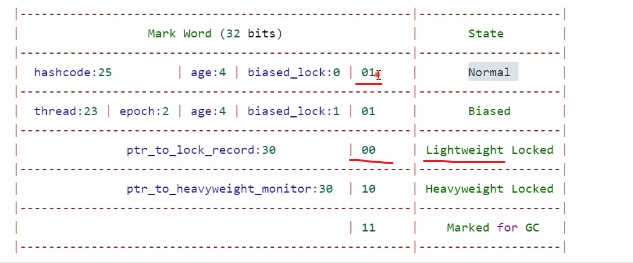
**public class** TestLightWeightLock {  
 **static final** Object ***obj*** = **new** Object();  
  
 **public static void** f1() {  
  
 **synchronized** (***obj***) {  
 *//同步块A  
 f2*();  
 }  
 }  
  
 *//锁重入* **public static void** f2() {  
 **synchronized**(***obj***){  
 *//同步块B* }  
 }  
}

【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。

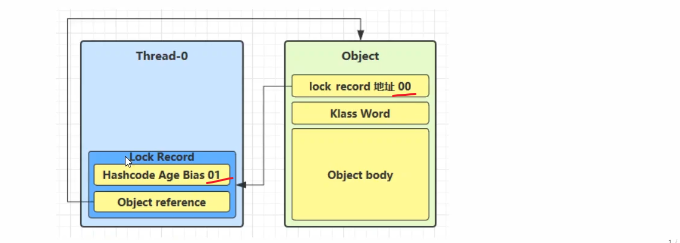


【2】让锁记录的Object reference指向锁对象，并尝试用CAS（Compare And Swap）替换Object的MarkWord，将MarkWord的值存入锁记录。





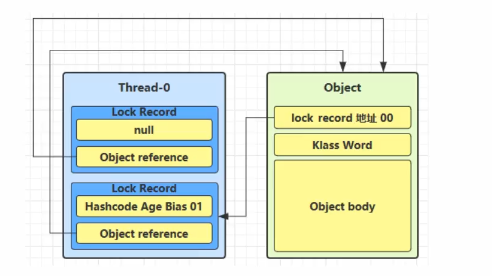
【3.1】如果CAS交换成功（对象头是01），对象头中存储了锁记录地址和状态00，表示由该线程给对象加锁，这时图示如下：



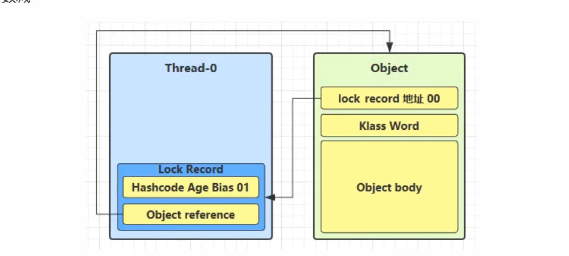
【3.2】如果CAS失败，由两种情况：

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。

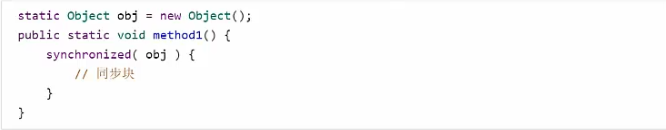


【4.2】当退出synchronized代码块（解锁时），锁记录不为null，这时使用CAS将MarkWord的值恢复给锁对象。

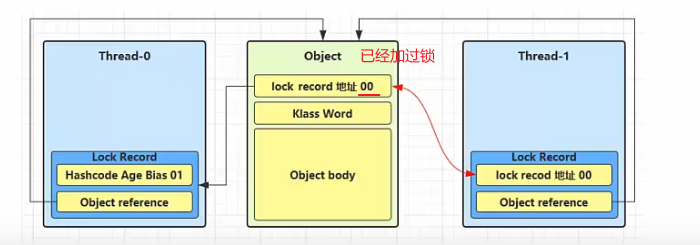
* 成功，则解锁成功。
* 失败，说明轻量级锁进行了锁膨胀或已经升级成了重量级锁，进入重量级锁的解锁流程。

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。

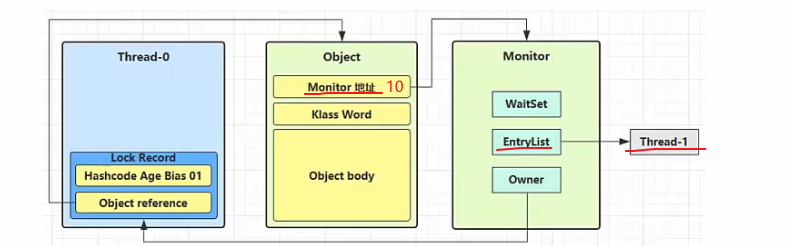


【2】这时Thread-1加轻量级锁失败，进入锁膨胀流程。

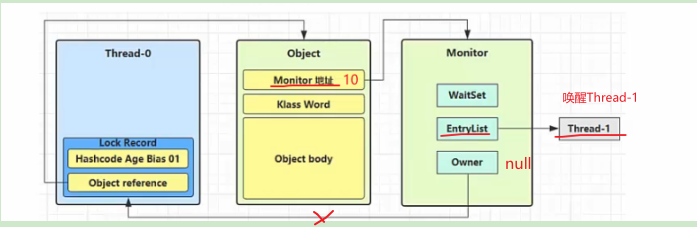
即为Object对象申请Monitor锁（重量级锁），让Object指向重量级锁地址。

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。

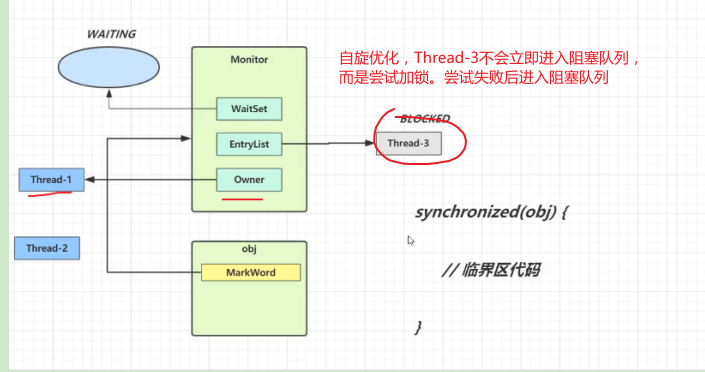


1. 自旋优化

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。

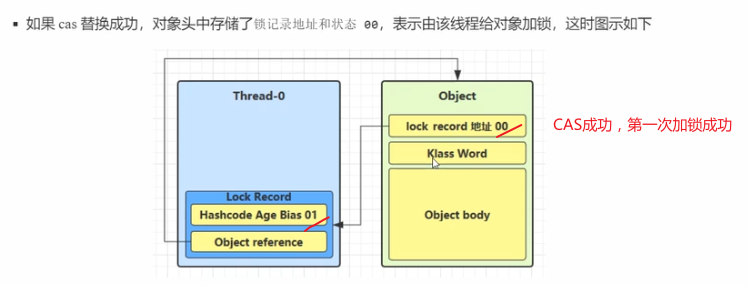
1. 偏向锁

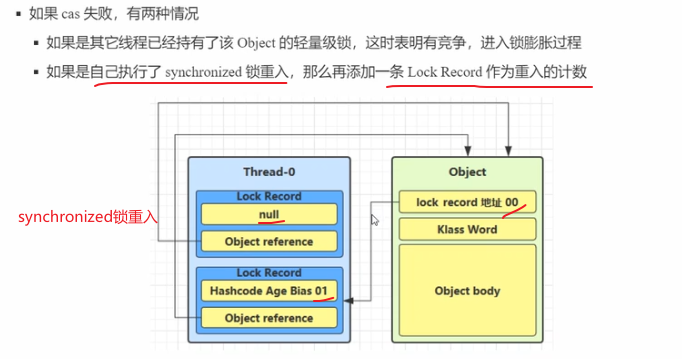
轻量级锁在没有竞争时（就自己这个线程），每次重入仍需要执行CAS操作。

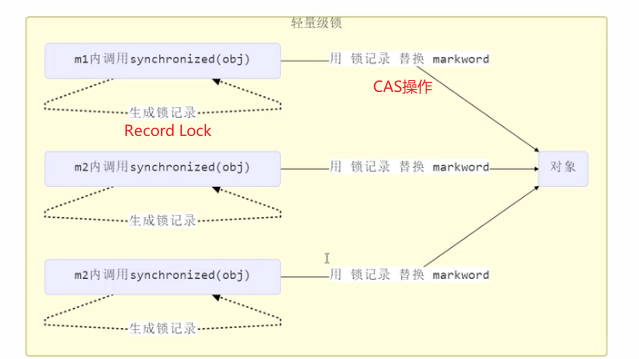
Java6中引入的偏向锁来做进一步的优化：只有第一次使用CAS将线程ID设置到对象的Mark Word头，之后发现这个线程ID是自己的就表示没有竞争，不用重新CAS。只要不发生竞争，这个对象就归该线程所有。

例如：

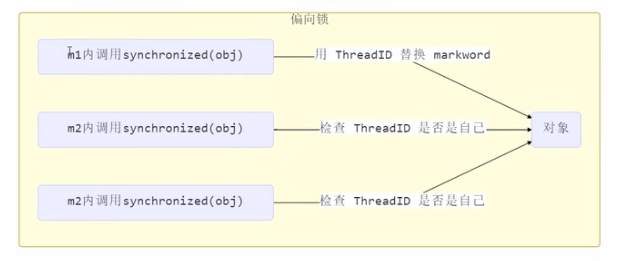
**package** com.concurrent.p2;  
  
*/\*\*  
 \* 偏向锁  
 \*/***public class** TestBiasedLock {  
 **static final** Object ***obj*** = **new** Object();  
  
 **public static void** f1() {  
  
 **synchronized** (***obj***) {  
 *//同步块A  
 f2*();  
 }  
 }  
  
 *//锁重入* **public static void** f2() {  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





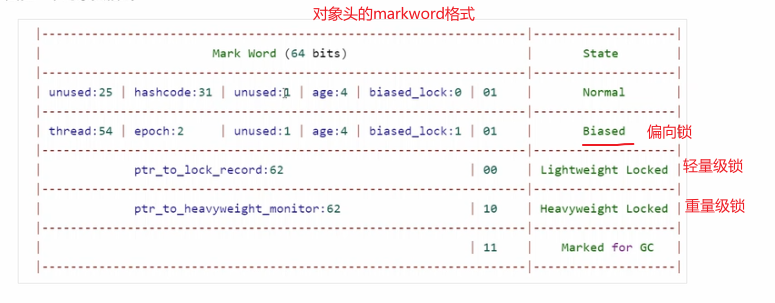


轻量级锁：第一次调用synchronized，会产生一个锁记录RecordLock，使用CAS操作将锁记录与对象的markword替换（对象由01-->00）；之后再次调用synchronized，，再次使用CAS操作尝试替换锁记录和markword，由于第一次已经进行了替换，后续替换不会成功，但是依然会产生一个锁记录，这个锁记录用来计数。



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**

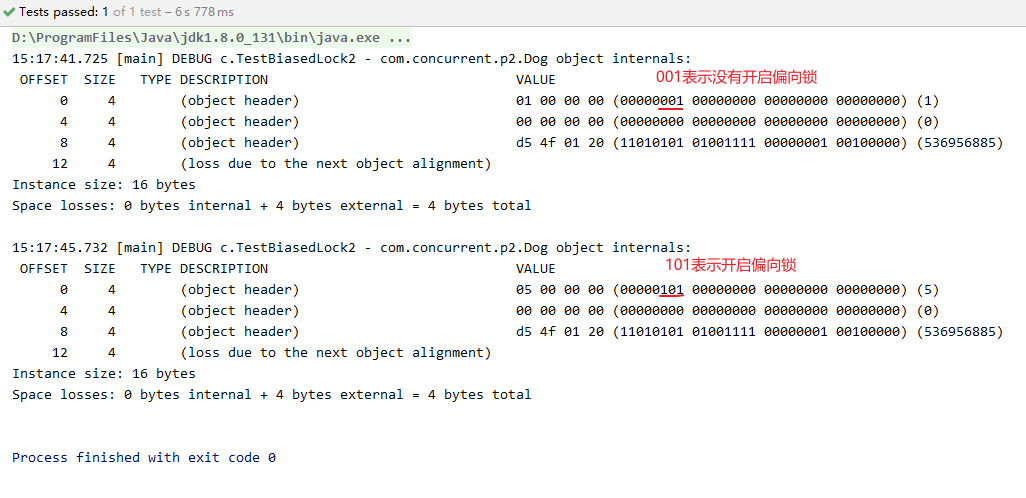
来禁用延迟。

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。

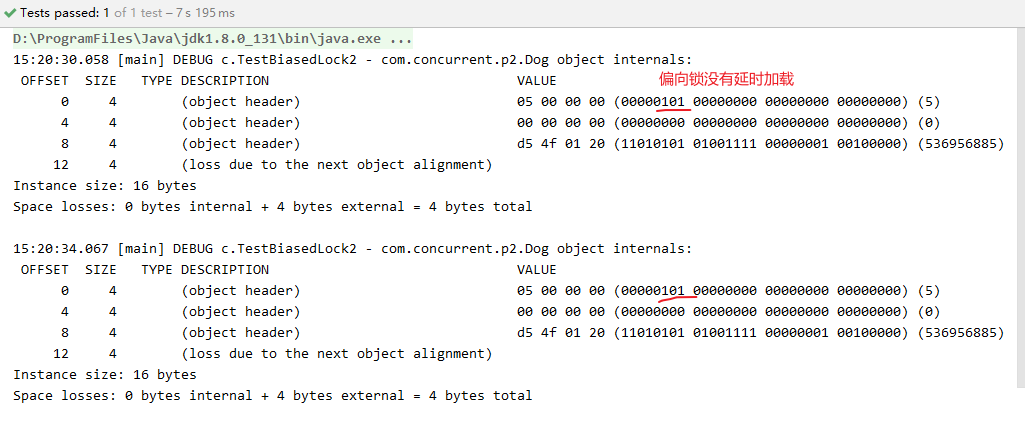
**偏向锁状态**

* 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());  
}

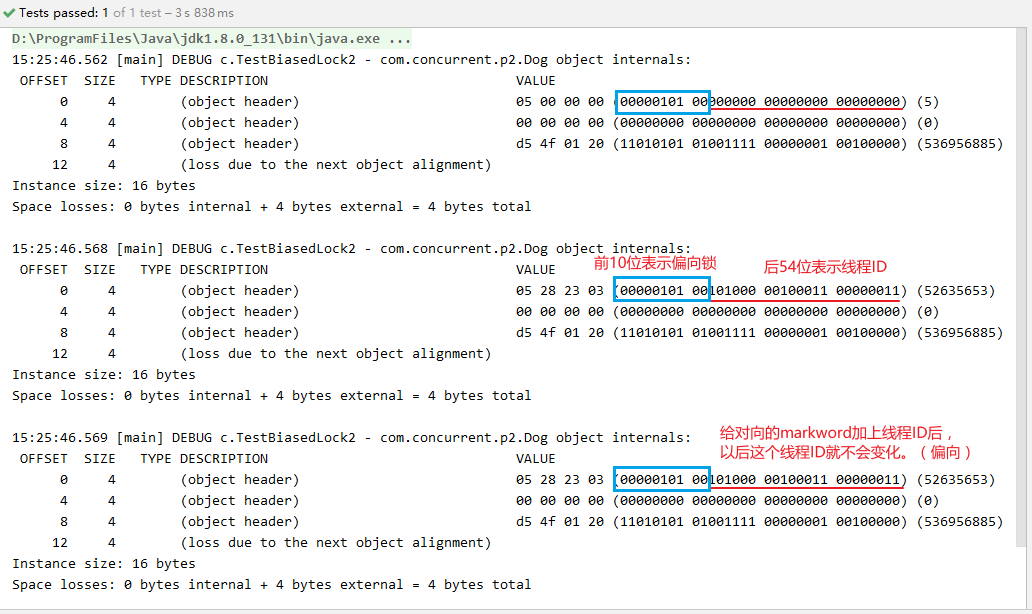


加了JVM参数之后：



* 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。

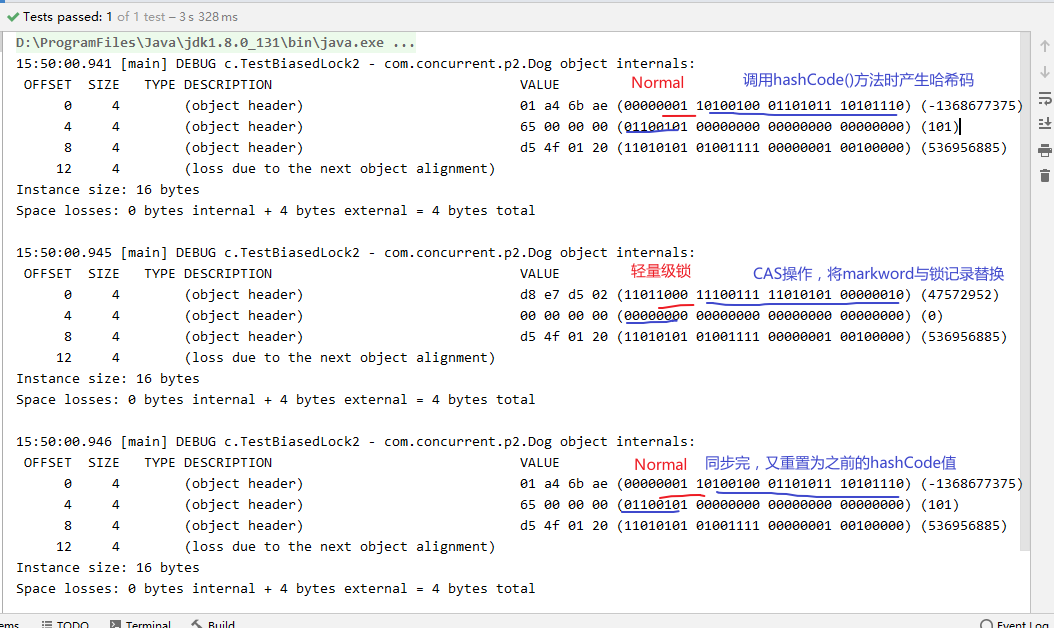
轻量级锁会在锁记录中记录hashCode；

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。

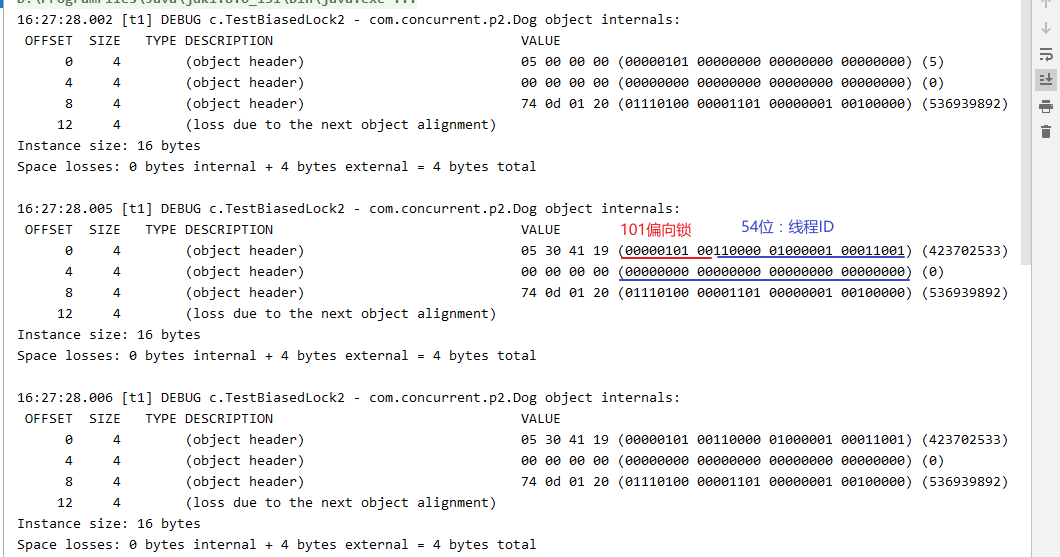
在重量级锁中，对象的hashCode会存放在Monitor对象中，解锁之后可以还原。

**撤销-其他线程使用对象**

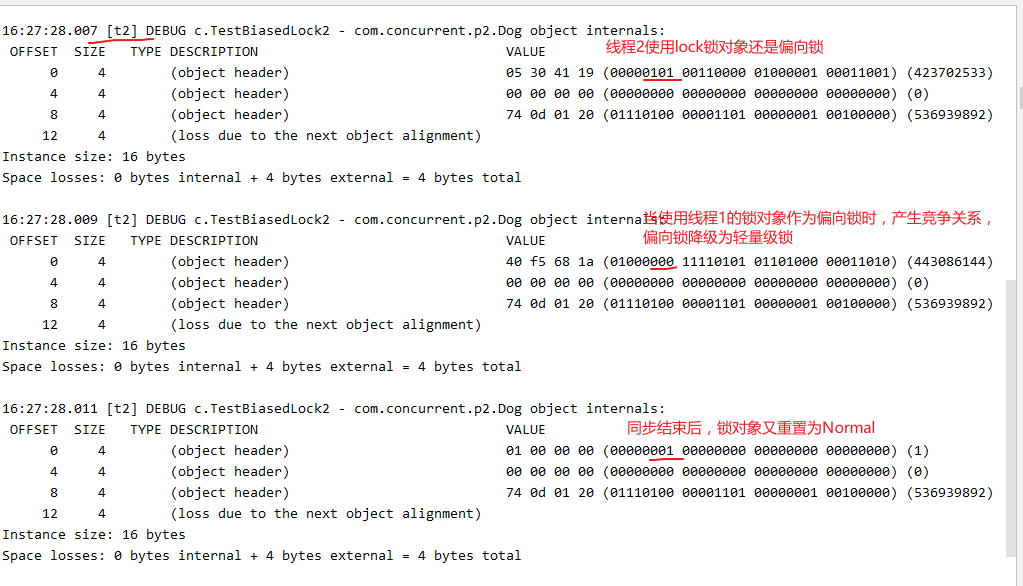
当有其他线程使用偏向锁对象时，会将偏向锁降级为轻量级锁。

*/\*\*  
 \* 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。

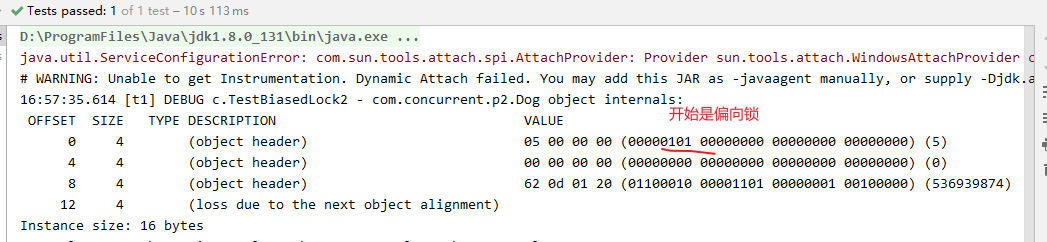


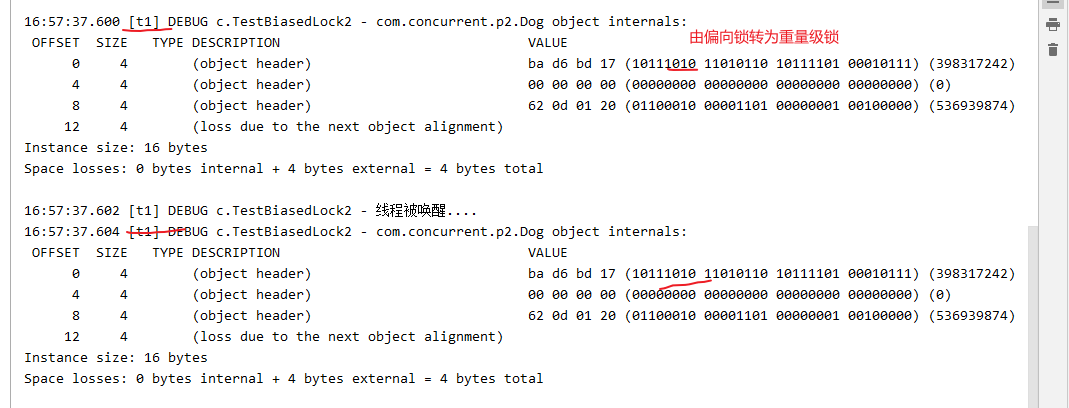
**撤销-调用wait/notify**

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。

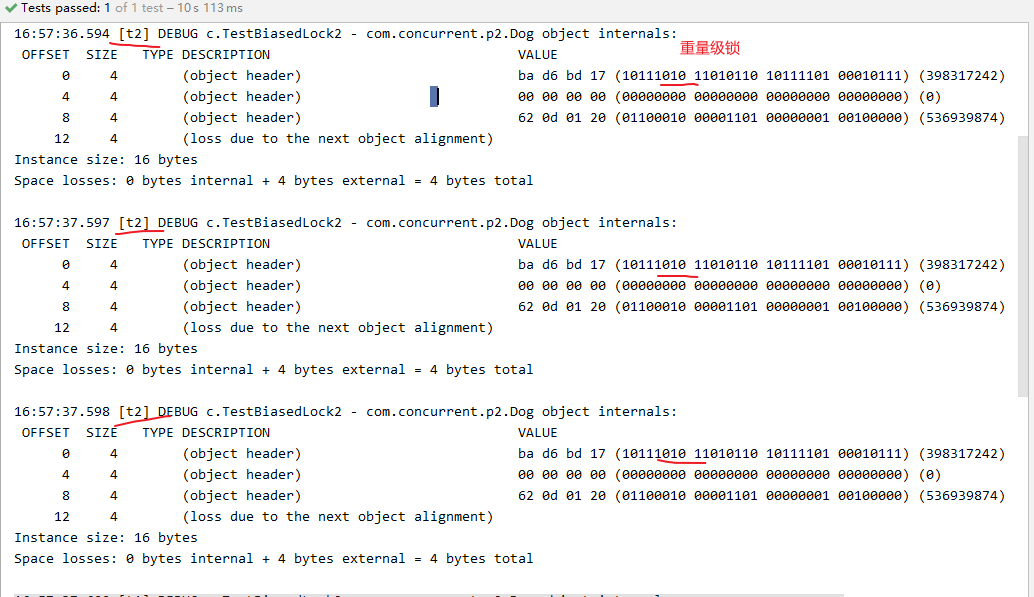
*/\*\*  
 \* 撤销-wait/notify  
 \* 偏向锁转为重量级锁  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/*@Test  
**public void** t6() **throws** InterruptedException {  
 Dog d = **new** Dog();  
 *//线程1* **new** Thread(() -> {  
 ***log***.debug(ClassLayout.*parseInstance*(d).toPrintable());  
 **synchronized** (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：



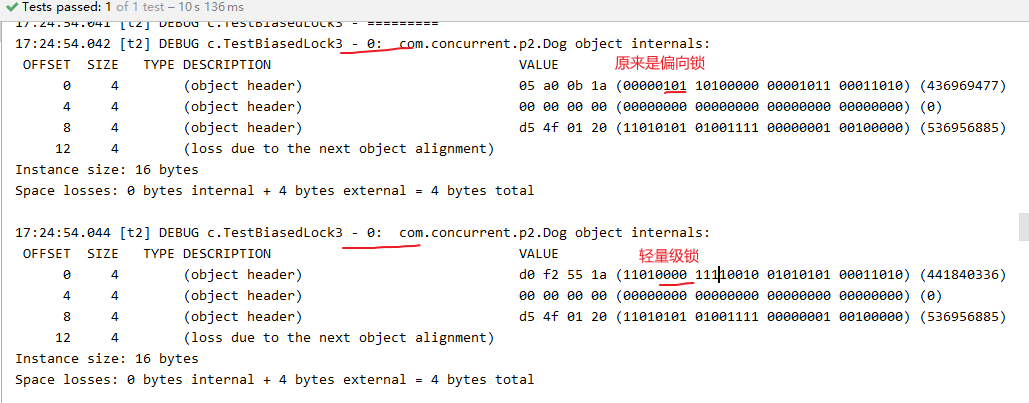
**批量重偏向**

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。

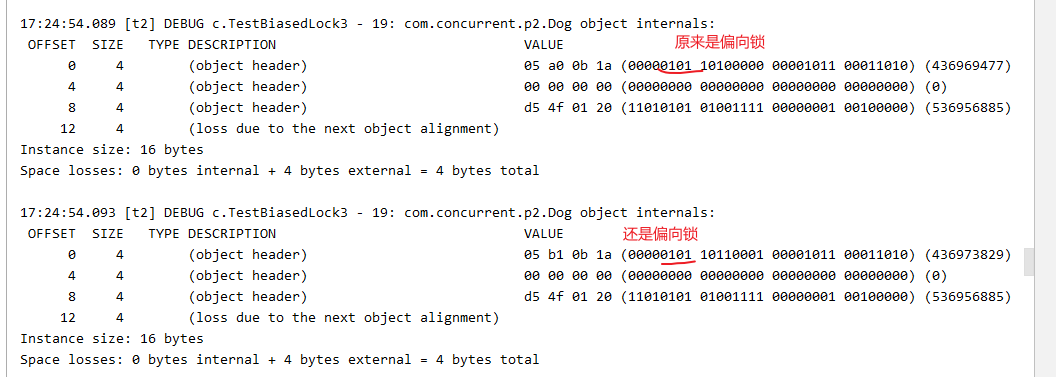
当撤销偏向锁阈值超过20次后，jvm会认为是不是偏向错了，于是会在给这些对象加锁时重新偏向至加锁线程。

*/\*\*  
 \* 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：



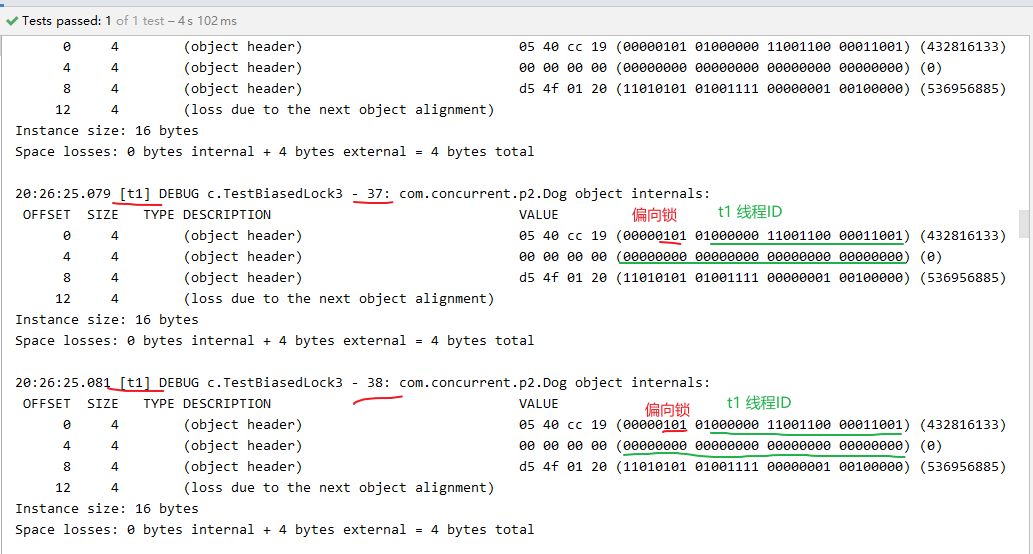
**批量撤销**

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。

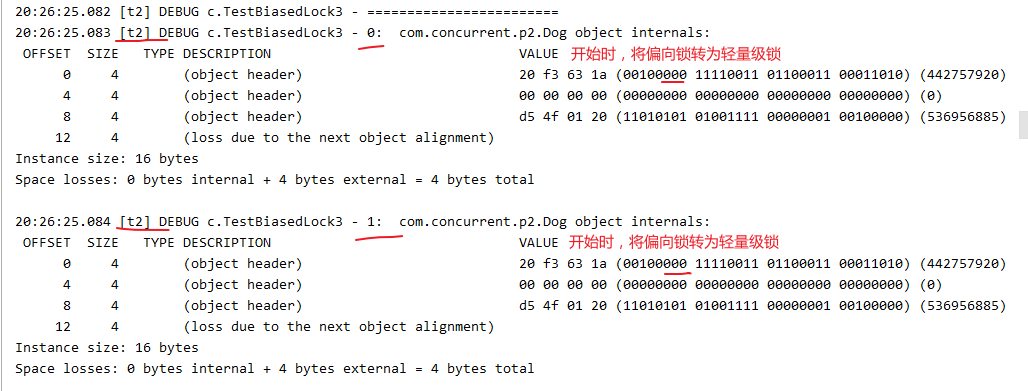
*/\*\*  
 \* 批量撤销  
 \* 关闭延时加载 -XX:BiasedLockingStartupDelay=0  
 \*/***static** Thread *t1*, *t2*, *t3*;  
  
@Test  
**public void** t2() {  
 Vector<Dog> list = **new** Vector<>();  
 **long** loopNumber = 39;  
 *//线程1  
 t1* = **new** Thread(() -> {  
 **for** (**int** i = 0; i < loopNumber; i++) {  
 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());

}

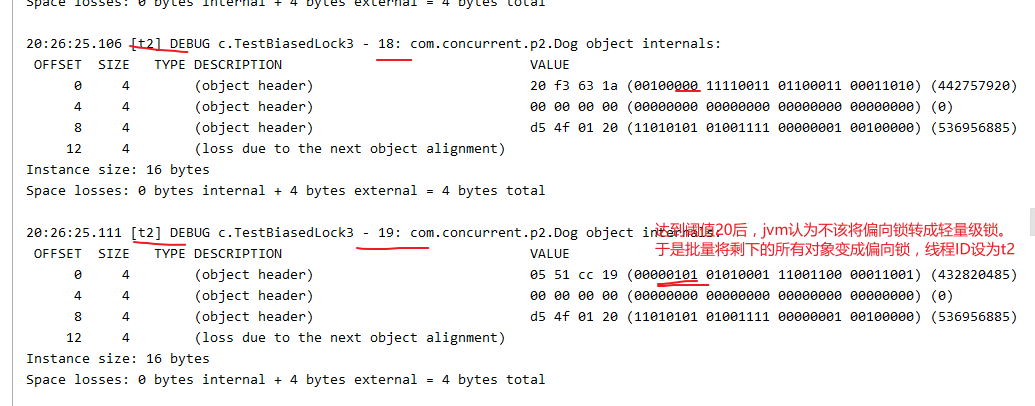
t1线程：锁对象偏向t1线程。



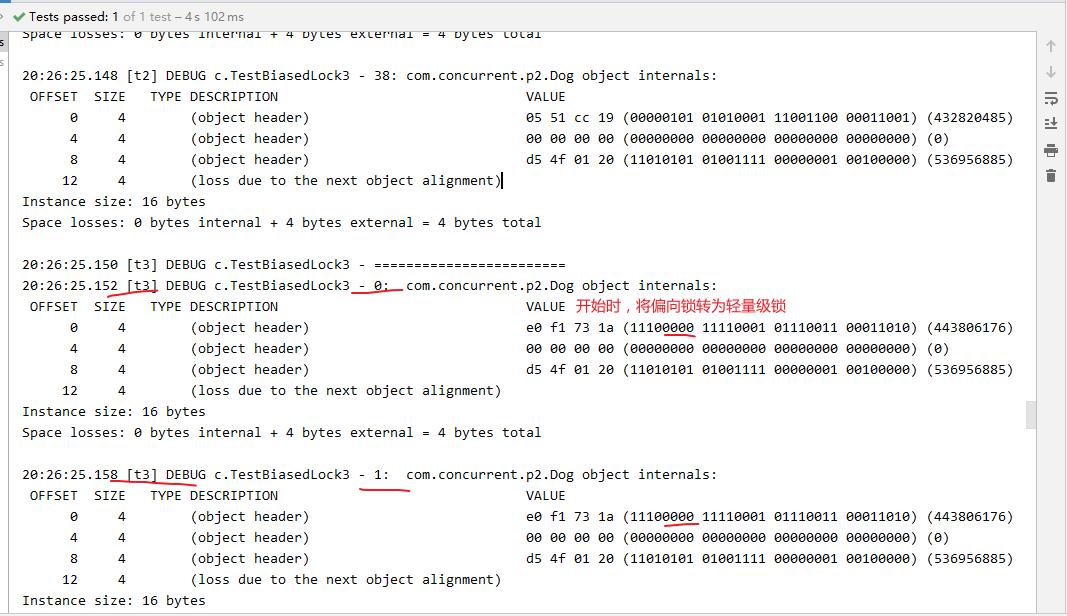
t2线程：一开始一个一个撤销，将锁对象变成轻量级锁。



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）。

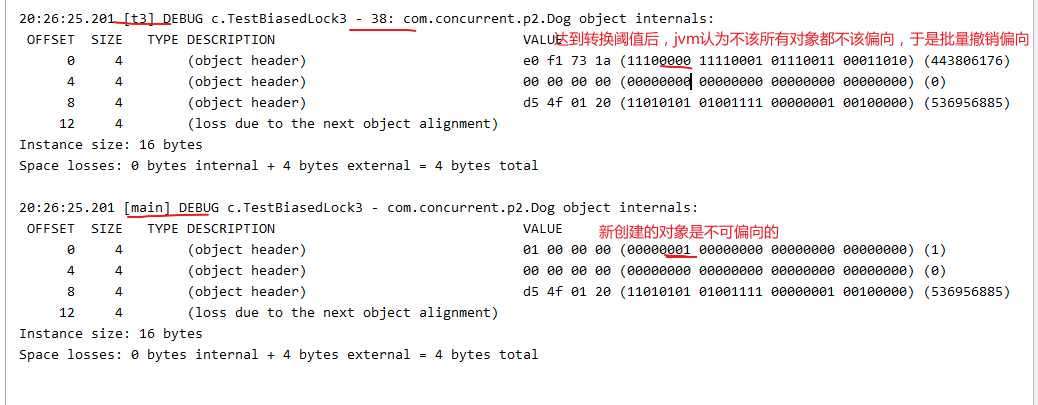


t3线程：前20个都是轻量级锁，不可偏向的。



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

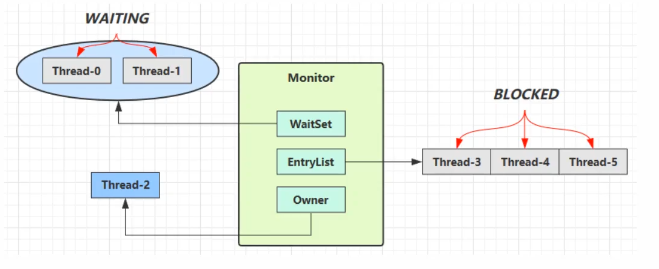
没有加锁消除优化，会导致运行效率低。





wait/notify

1. wait/notify工作原理



* Owner线程发现条件不满足，调用wait()方法，即可进入WaitSet变成WAITING状态。
* BLOCKED和WAITING的线程都处于阻塞状态，不占用CPU时间片。
* BLOCKED线程会在Owner线程释放锁时唤醒。
* WAITING线程会在Owner线程调用notify或者notifyAll时唤醒，但唤醒后并不意味者立即获得锁，仍然需要进入EntryList重新竞争。

1. wait/notify API

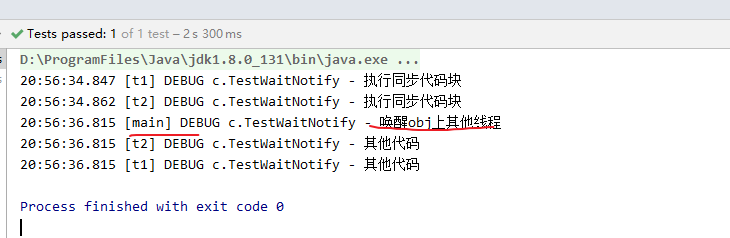
* obj.wait() 让进入 object 监视器的线程到 waitSet 等待。
* obj.notify() 在 object 上正在 waitSet 等待的线程中挑一个唤醒。
* obj.notifyAll() 让 object 上正在 waitSet 等待的线程全部唤醒。

它们都是线程之间进行协作的手段，都属于 Object 对象的方法。必须获得此对象的锁，才能调用这几个方法。

* 无参数wait()，无限制等待下去

*//定义锁对象***static final** Object ***obj*** = **new** Object();  
  
@Test  
**public void** t1() **throws** InterruptedException {  
 *//线程1* Thread t1 = **new** Thread(() -> {  
 **synchronized** (***obj***) {  
 ***log***.debug(**"执行同步代码块"**);  
 **try** {  
 ***obj***.wait(); *//让线程一直在obj上等待下去* } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"其他代码"**);  
 }  
 }, **"t1"**);  
 t1.start();  
 *//线程2* Thread t2 = **new** Thread(() -> {  
 **synchronized** (***obj***) {  
 ***log***.debug(**"执行同步代码块"**);  
 **try** {  
 ***obj***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"其他代码"**);  
 }  
 }, **"t2"**);  
 t2.start();  
 *//主线程2秒后执行* Thread.*sleep*(2000);  
 ***log***.debug(**"唤醒obj上其他线程"**);  
 **synchronized** (***obj***) {  
 *//obj.notify(); //唤醒obj上的一个线程* ***obj***.notifyAll(); *//唤醒obj上所有的线程* }  
 *//等待t1、t2执行完成* t1.join();  
 t2.join();  
}

运行结果：

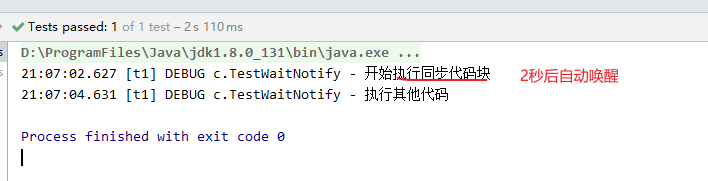


* 有参数wait(long time)

*//定义锁对象***static final** Object ***obj*** = **new** Object();

@Test  
**public void** t2() {  
 Thread t1 = **new** Thread(() -> {  
 **synchronized** (***obj***) {  
 ***log***.debug(**"开始执行同步代码块"**);  
 **try** {  
 *//1秒后自动唤醒* ***obj***.wait(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"执行其他代码"**);  
 }  
 }, **"t1"**);  
 t1.start();  
 *//等待t1执行完* **try** {  
 t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

运行结果：

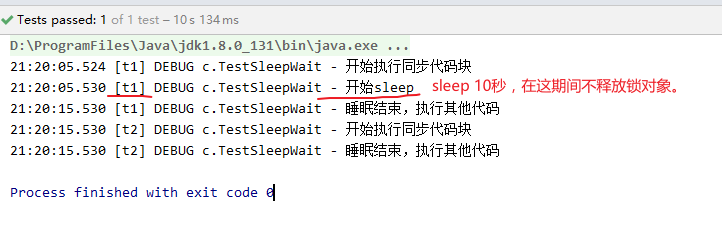


1. sleep(long n)和wait(long n)的区别
2. sleep 是 Thread 方法，而 wait 是 Object 的方法。
3. sleep 不需要强制和 synchronized 配合使用，但 wait 需要和 synchronized 一起用。
4. sleep 在睡眠的同时，不会释放对象锁的，但 wait 在等待的时候会释放对象锁。
5. 共同点：它们状态是 TIMED\_WAITING。

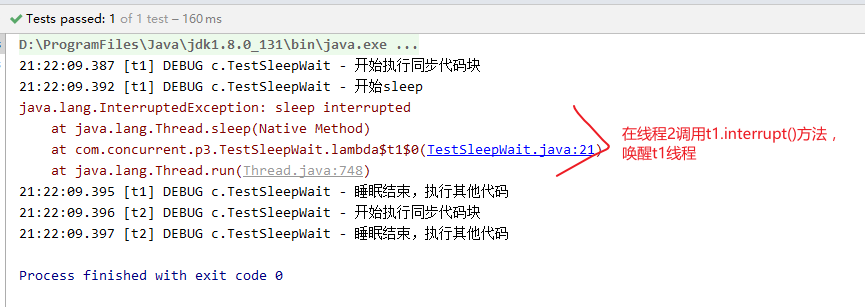
* Thread.sleep(long time);不会释放锁

**static final** Object ***obj*** = **new** Object();  
  
@Test  
**public void** t1() {  
 *//线程1* Thread t1 = **new** Thread(() -> {  
 **synchronized** (***obj***) {  
 ***log***.debug(**"开始执行同步代码块"**);  
 ***log***.debug(**"开始sleep"**);  
 **try** {  
 *//sleep时，线程不会释放锁* Thread.*sleep*(1000 \* 10);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"睡眠结束，执行其他代码"**);  
 }  
 }, **"t1"**);  
 t1.start();  
  
 *//线程2* Thread t2 = **new** Thread(() -> {  
 *//唤醒t1的sleep* t1.interrupt();  
 **synchronized** (***obj***) {  
 ***log***.debug(**"开始执行同步代码块"**);  
 ***log***.debug(**"睡眠结束，执行其他代码"**);  
 }  
 }, **"t2"**);  
 t2.start();  
  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

如果不在其他线程中interrupt，



在其他线程中interrupt，

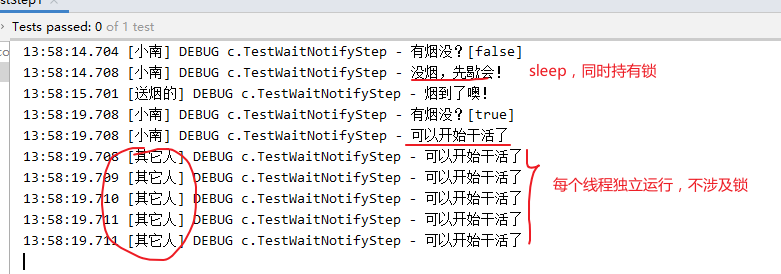


1. 代码优化

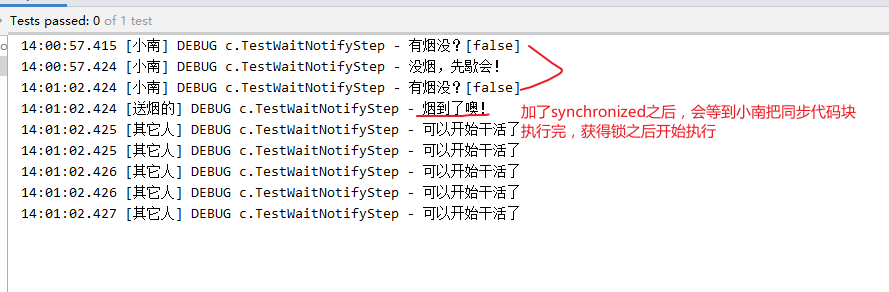
【1】使用sleep

**static final** Object ***room*** = **new** Object();  
**static boolean** *hasCigarette* = **false**;  
**static boolean** *hasTakeout* = **false**;  
  
@Test  
**public void** testStep1() **throws** InterruptedException {  
 *//线程1* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (!*hasCigarette*) {  
 ***log***.debug(**"没烟，先歇会！"**);  
 **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (*hasCigarette*) {  
 ***log***.debug(**"可以开始干活了"**);  
 }  
 }  
 }, **"小南"**).start();  
  
 *//线程2-6* **for** (**int** i = 0; i < 5; i++) {  
 **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"可以开始干活了"**);  
 }  
 }, **"其它人"**).start();  
 }  
  
 *//线程7* Thread.*sleep*(1000);  
 **new** Thread(() -> {  
 *// 这里能不能加 synchronized (room)？* *hasCigarette* = **true**;  
 ***log***.debug(**"烟到了噢！"**);  
 }, **"送烟的"**).start();  
  
 Thread.*sleep*(100000);  
}

运行结果（送烟的没加synchronized ）：



运行结果（送烟的加了synchronized）：



问题：

当小南持有锁sleep的时候，不会释放锁，导致其他人不能获得锁，也就一直无法运行。（效率低）

其他干活的线程都会一直阻塞，效率太低。

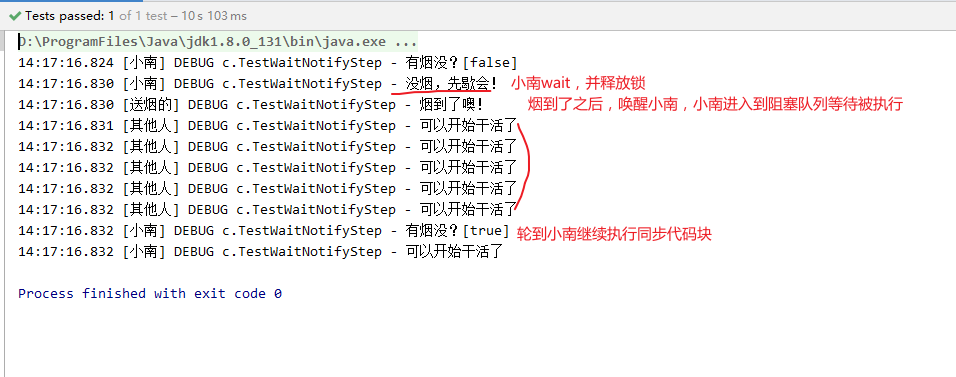
小南线程必须睡足5秒才能醒来，就算是烟提前送到也无法醒来。

加了synchronized(room)后，就好比小南在房间里面反锁，烟根本无法送进门，main没加synchronized就好像是翻窗户进来的。

【2】只有1个线程wai()，使用notify()

@Test  
**public void** testStep2() **throws** InterruptedException {  
 *//使用wait-notify优化  
 //小南线程* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (!*hasCigarette*) {  
 ***log***.debug(**"没烟，先歇会！"**);  
 *//没有烟就等待，此时可以释放锁* **try** {  
 ***room***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (*hasCigarette*) {  
 ***log***.debug(**"可以开始干活了"**);  
 }  
 }  
 }, **"小南"**).start();  
  
 *//其他人线程* **for** (**int** i = 0; i < 5; i++) {  
 **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"可以开始干活了"**);  
 }  
 }, **"其他人"**).start();  
 }  
  
 *//送烟的* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 *hasCigarette* = **true**;  
 ***log***.debug(**"烟到了噢！"**);  
 *//唤醒正在等待的线程* ***room***.notify();  
 }  
 }, **"送烟的"**).start();  
  
 Thread.*sleep*(10000);  
}

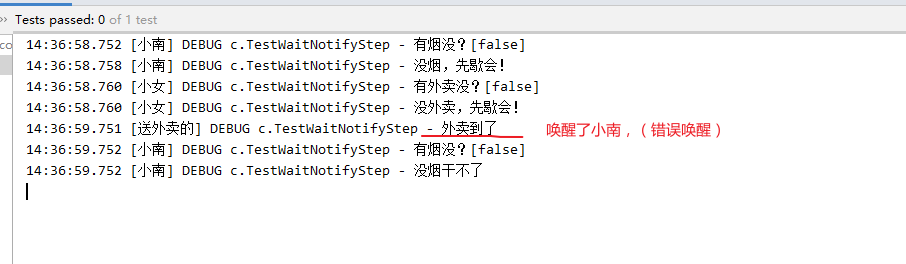
运行结果：



【3】多个wait()线程，使用notify()

@Test  
**public void** testStep3() **throws** InterruptedException {  
 *//小南线程* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (!*hasCigarette*) {  
 ***log***.debug(**"没烟，先歇会！"**);  
 *//没有烟就等待，此时可以释放锁* **try** {  
 ***room***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (*hasCigarette*) {  
 ***log***.debug(**"可以开始干活了"**);  
 } **else** {  
 ***log***.debug(**"没烟干不了"**);  
 }  
 }  
 }, **"小南"**).start();  
  
 *//小女线程* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有外卖没？[{}]"**, *hasTakeout*);  
 **if** (!*hasTakeout*) {  
 ***log***.debug(**"没外卖，先歇会！"**);  
 *//没有外卖就等待，此时可以释放锁* **try** {  
 ***room***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有外卖没？[{}]"**, *hasTakeout*);  
 **if** (*hasTakeout*) {  
 ***log***.debug(**"可以开始干活了"**);  
 } **else** {  
 ***log***.debug(**"没外卖干不了"**);  
 }  
 }  
 }, **"小女"**).start();  
  
 *//送外卖的* **new** Thread(() -> {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **synchronized** (***room***) {  
 *hasTakeout* = **true**;  
 ***log***.debug(**"外卖到了"**);  
 *//room.notify();* ***room***.notify();  
 }  
 }, **"送外卖的"**).start();  
  
 Thread.*sleep*(10000);  
}

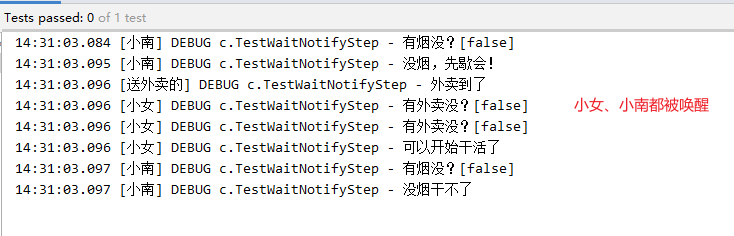
运行结果：



问题：notify()只能随机唤醒一个wait()的线程，（错误唤醒）

【4】notifyAll();

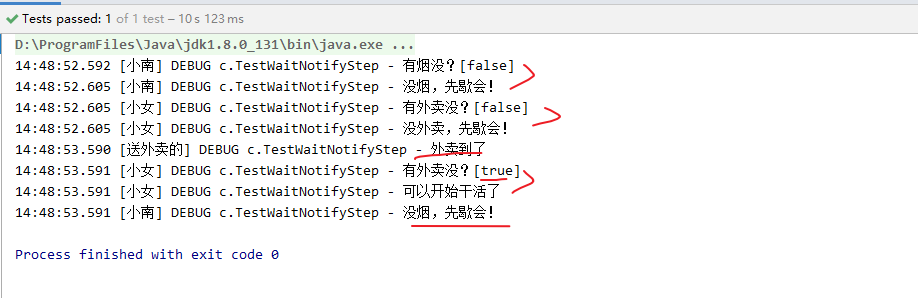
*//送外卖的***new** Thread(() -> {  
 **synchronized** (***room***) {  
 *hasTakeout* = **true**;  
 ***log***.debug(**"外卖到了"**);  
 *//room.notify();* ***room***.notifyAll();  
 }  
}, **"送外卖的"**).start();



【5】将if单次判断改为while多次判断（解决虚假唤醒）\*\*\*

@Test  
**public void** testStep5() **throws** InterruptedException {  
 *//小南线程* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 *//将if改为while多次判断* **while**(!*hasCigarette*) {  
 ***log***.debug(**"没烟，先歇会！"**);  
 *//没有烟就等待，此时可以释放锁* **try** {  
 ***room***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有烟没？[{}]"**, *hasCigarette*);  
 **if** (*hasCigarette*) {  
 ***log***.debug(**"可以开始干活了"**);  
 } **else** {  
 ***log***.debug(**"没烟干不了"**);  
 }  
 }  
 }, **"小南"**).start();  
  
 *//小女线程* **new** Thread(() -> {  
 **synchronized** (***room***) {  
 ***log***.debug(**"有外卖没？[{}]"**, *hasTakeout*);  
 **while**(!*hasTakeout*) {  
 ***log***.debug(**"没外卖，先歇会！"**);  
 *//没有外卖就等待，此时可以释放锁* **try** {  
 ***room***.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"有外卖没？[{}]"**, *hasTakeout*);  
 **if** (*hasTakeout*) {  
 ***log***.debug(**"可以开始干活了"**);  
 } **else** {  
 ***log***.debug(**"没外卖干不了"**);  
 }  
 }  
 }, **"小女"**).start();  
  
 *//送外卖的* **new** Thread(() -> {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **synchronized** (***room***) {  
 *hasTakeout* = **true**;  
 ***log***.debug(**"外卖到了"**);  
 *//room.notify();* ***room***.notifyAll();  
 }  
 }, **"送外卖的"**).start();  
  
 Thread.*sleep*(10000);  
}

运行结果：



1. wait()/notify() 虚假唤醒解决方法

使用notifyAll()。

使用while多次判断，不要使用if单次判断。

***.//线程A***

***synchronized(lock){***

***while(条件不成立){***

***lock.wait()***

***}***

***}***

***//线程B***

***synchronized(lock){***

***while(条件不成立){***

***...***

***}***

***}***

***//唤醒线程***

***synchronized(lock){***

***lock.notifyAll();***

***}***

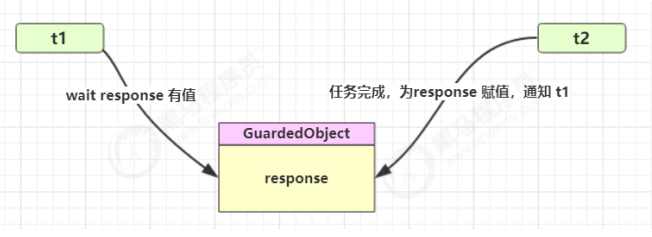
同步模式-保护性暂停

1、概念

即 Guarded Suspension，用在一个线程等待另一个线程的执行结果。

要点：

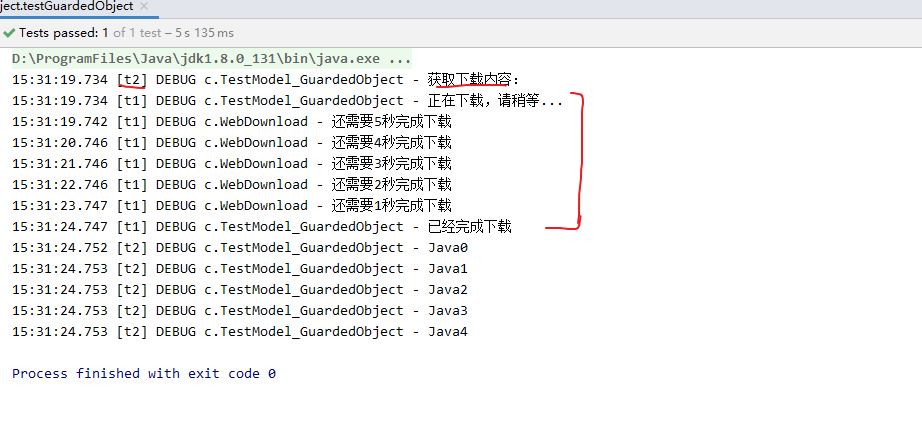
* 有一个结果需要从一个线程传递到另一个线程，让他们关联同一个 GuardedObject
* 如果有结果不断从一个线程到另一个线程那么可以使用消息队列（见生产者/消费者）
* JDK 中，join 的实现、Future 的实现，采用的就是此模式
* 因为要等待另一方的结果，因此归类到同步模式



2、GuardedObject代码和测试代码

**package** com.concurrent.p3;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
*/\*\*  
 \* 同步模式-保护性暂停  
 \*/*@Slf4j(topic = **"c.TestModel\_GuardedObject"**)  
**public class** TestModel\_GuardedObject {  
 @Test  
 **public void** testGuardedObject() {  
 *//定义共享对象* GuardedObject guardedObject = **new** GuardedObject();  
  
 *//线程1执行下载* Thread t1 = **new** Thread(() -> {  
 ***log***.debug(**"正在下载，请稍等..."**);  
 List<String> list = (ArrayList<String>) WebDownload.*download*();  
 guardedObject.complete(list);  
 ***log***.debug(**"已经完成下载"**);  
 }, **"t1"**);  
 t1.start();  
  
 *//线程2获取下载结果* Thread t2 = **new** Thread(() -> {  
 ***log***.debug(**"获取下载内容..."**);  
 List<String> list = (ArrayList<String>) guardedObject.get();  
 list.forEach((s) -> {  
 ***log***.debug(s);  
 });  
 }, **"t2"**);  
 t2.start();  
  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
*/\*\*  
 \* 线程同步对象：GuardedObject  
 \*/***class** GuardedObject {  
 *//同步对象* **private** Object **resp**;  
  
 *//获取结果* **public** Object get() {  
 **synchronized** (**this**) {  
 *//循环判断，如果当前结果为空则等待；不为空则返回结果* **while** (**this**.**resp**== **null**) {  
 **try** {  
 **this**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **return resp**;  
 }  
 }  
  
 *//产生结果* **public void** complete(Object resp) {  
 **synchronized** (**this**) {  
 **this**.**resp**= resp;  
 *//唤醒所有等待线程* **this**.notifyAll();  
 }  
 }  
}  
  
*/\*\*  
 \* 模拟网页下载器：WebDownload  
 \*/*@Slf4j(topic = **"c.WebDownload"**)  
**class** WebDownload {  
 **public static** Object download() {  
 List<String> list = **new** ArrayList<>();  
 *//模拟5秒内完成下载* **for** (**int** i = 0; i < 5; i++) {  
 ***log***.debug(**"还需要{}秒完成下载"**, (5 - i));  
 **try** {  
 Thread.*sleep*(1000);  
 list.add(**"Java"** + i);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **return** list;  
 }  
}

运行结果：



保护性暂停模式的有点：

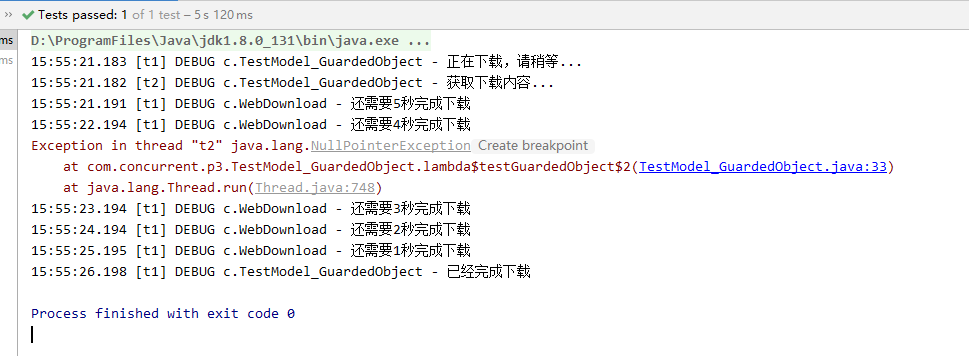
与join()相比，不用等到另一个线程结束再执行别的代码。

等待结果的变量能够设置成局部的，不用设置成全局的。

1. 扩展1

*//获取结果扩展-添加获取超时参数  
//timeout 最大等待时间***public** Object get(**long** timeout) {  
 **synchronized** (**this**) {  
 **long** passedTime = 0;  
 **long** beginTime = System.*currentTimeMillis*();  
 *//循环判断，如果当前结果为空则等待；不为空则返回结果* **while** (**this**.**resp** == **null**) {  
 **long** waitTime = timeout - passedTime;  
 *//如果经历时间超过超时时间，返回* **if** (waitTime <= 0) {  
 **break**;  
 }  
 **try** {  
 **this**.wait(waitTime); *//避免虚假唤醒15:00:01* } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 passedTime = System.*currentTimeMillis*() - beginTime;  
 }  
 **return resp**;  
 }  
}

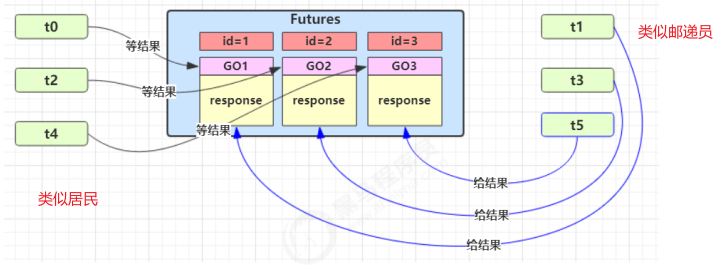
运行结果：



1. 扩展2-多任务版GuardedObject

图中 Futures 就好比居民楼一层的信箱（每个信箱有房间编号），左侧的 t0，t2，t4 就好比等待邮件的居民，右侧的 t1，t3，t5 就好比邮递员。

如果需要在多个类之间使用 GuardedObject 对象，作为参数传递不是很方便，因此设计一个用来解耦的中间类，这样不仅能够解耦【结果等待者】和【结果生产者】，还能够同时支持多个任务的管理。



新增 id 用来标识 Guarded Object。

* 同步类MailGuardedObject：

**package** com.concurrent.p3.multiGuardedObject;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
@Slf4j(topic = **"c.MailGuardedObject"**)  
**public class** MailGuardedObject {  
 *//标识 GuardedObject resp* **private** Integer **id**;  
  
 **private** Object **resp**;  
  
 **public** MailGuardedObject(Integer id) {  
 **this**.**id** = id;  
 }  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
  
 *//产生结果* **public void** complete(Object obj) {  
 **synchronized** (**this**) {  
 **this**.**resp** = obj;  
 **this**.notifyAll();  
 }  
 }  
  
 *//获取结果* **public** Object get() {  
 **synchronized** (**this**) {  
 **while** (**resp** == **null**) {  
 **try** {  
 **this**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **return resp**;  
 }  
 }  
  
 *//超时获取结果* **public** Object get(**long** timeout) {  
 **synchronized** (**this**) {  
 **long** begin = System.*currentTimeMillis*();  
 **long** passTime = 0; *//经历时间* **while** (**resp** == **null**) {  
 **long** waitTime = timeout - passTime;  
 **if** (waitTime <= 0) {  
 **break**;  
 }  
 **try** {  
 **this**.wait(waitTime);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 passTime = System.*currentTimeMillis*() - begin;  
 }  
 **return resp**;  
 }  
 }  
  
}

* 中间解耦类 MailBox

**package** com.concurrent.p3.multiGuardedObject;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
**import** java.util.Hashtable;  
**import** java.util.Map;  
**import** java.util.Set;  
  
*/\*\*  
 \* 中间解耦类  
 \* 解耦结果的产生者和结果获取者  
 \* RPC框架常用  
 \*/*@Slf4j(topic = **"c.MailBox"**)  
**public class** MailBox {  
 *//HashTable是线程安全的* **private static** Map<Integer, MailGuardedObject> *mailBoxs* = **new** Hashtable<>();  
  
 *//唯一标识* **private static int** *id* = 1;  
  
 *//id递增* **private synchronized static int** generateId() {  
 **return** *id*++;  
 }  
  
 *//产生MailGuardedObject* **public static** MailGuardedObject createMailGuardedObject() {  
 *//创建MailGuardedObject对象，id自增* MailGuardedObject mailGuardedObject = **new** MailGuardedObject(*generateId*());  
 *mailBoxs*.put(mailGuardedObject.getId(), mailGuardedObject);  
 **return** mailGuardedObject;  
 }  
  
 *//根据id获取MailGuardedObject* **public static** MailGuardedObject getMailGuardedObject(**int** id) {  
 **return** *mailBoxs*.remove(id);  
 }  
  
 *//返回编id集合* **public static** Set<Integer> getIds() {  
 **return** *mailBoxs*.keySet();  
 }  
}

* 收信人类 User

**package** com.concurrent.p3.multiGuardedObject;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
*/\*\*  
 \* 用户线程，获取邮件  
 \*/*@Slf4j(topic = **"c.User"**)  
**public class** User **extends** Thread {  
 @Override  
 **public void** run() {  
 *//收信* MailGuardedObject mailGuardedObject = MailBox.*createMailGuardedObject*();  
 ***log***.debug(**"开始收信 id:{}"**, mailGuardedObject.getId());  
 Object mail = mailGuardedObject.get(5000);  
 ***log***.debug(**"收信 id:{},内容:{}"**, mailGuardedObject.getId(), mail);  
 }  
}

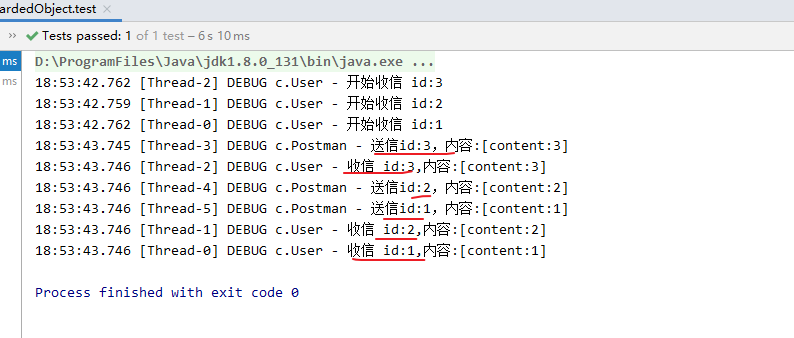
* 邮递员类 Postman

**package** com.concurrent.p3.multiGuardedObject;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
@Slf4j(topic = **"c.Postman"**)  
**public class** Postman **extends** Thread {  
  
 **private int id**;  
 **private** String **mail**;  
  
 **public** Postman(**int** id, String mail) {  
 **this**.**id** = id;  
 **this**.**mail** = mail;  
 }  
  
 @Override  
 **public void** run() {  
 MailGuardedObject mailGuardedObject = MailBox.*getMailGuardedObject*(**id**);  
 ***log***.debug(**"送信id:{}，内容:{}"**, mailGuardedObject.getId(), **mail**);  
 mailGuardedObject.complete(**mail**);  
 }  
}

* 测试代码

**package** com.concurrent.p3.multiGuardedObject;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
@Slf4j(topic = **"c.TestModel\_MultiGuardedObject"**)  
**public class** TestModel\_MultiGuardedObject {  
  
 @Test  
 **public void** test() **throws** InterruptedException {  
 **for** (**int** i = 0; i < 3; i++) {  
 **new** User().start();  
 }  
 Thread.*sleep*(1000);  
 **for** (Integer id : MailBox.*getIds*()) {  
 **new** Postman(id, **"[content:"** + id + **"]"**).start();  
 }  
  
 Thread.*sleep*(5000);  
 }  
}

运行结果：



join原理

join()应用了保护性暂停模式。

源码分析：

*/\*\*  
 \* Waits for this thread to die.  
 \*  
 \* <p> An invocation of this method behaves in exactly the same  
 \* way as the invocation  
 \*  
 \* <blockquote>  
 \* {****@linkplain*** *#join(long) join}{****@code*** *(0)}  
 \* </blockquote>  
 \*  
 \** ***@throws*** *InterruptedException  
 \* if any thread has interrupted the current thread. The  
 \* <i>interrupted status</i> of the current thread is  
 \* cleared when this exception is thrown.  
 \*/***public final void** join() **throws** InterruptedException {  
 join(0);   
}

*/\*\*  
 \* Waits at most {****@code*** *millis} milliseconds for this thread to  
 \* die. A timeout of {****@code*** *0} means to wait forever.  
 \*  
 \* <p> This implementation uses a loop of {****@code*** *this.wait} calls  
 \* conditioned on {****@code*** *this.isAlive}. As a thread terminates the  
 \* {****@code*** *this.notifyAll} method is invoked. It is recommended that  
 \* applications not use {****@code*** *wait}, {****@code*** *notify}, or  
 \* {****@code*** *notifyAll} on {****@code*** *Thread} instances.  
 \*  
 \** ***@param millis*** *\* the time to wait in milliseconds  
 \*  
 \** ***@throws*** *IllegalArgumentException  
 \* if the value of {****@code*** *millis} is negative  
 \*  
 \** ***@throws*** *InterruptedException  
 \* if any thread has interrupted the current thread. The  
 \* <i>interrupted status</i> of the current thread is  
 \* cleared when this exception is thrown.  
 \*/***public final synchronized void** join(**long** millis)  
**throws** InterruptedException {  
  **long base = System.*currentTimeMillis*(); //开始时间  
 long now = 0; //记录经历的时间**  
 **if** (millis < 0) { //等待时间小于0，不合法  
 **throw new** IllegalArgumentException(**"timeout value is negative"**);  
 }  
  
 **if** (millis == 0) { //等待时间等于0，会一直等待下去。wait()  
 **while** (isAlive()) {  
 wait(0);  
 }  
 } **else** {  
 **while** (isAlive()) {

// 防止错误唤醒  
  **long delay = millis - now;**   
 **if** (delay <= 0) {   
 **break**;  
 }  
 wait(delay);  
  **now = System.*currentTimeMillis*() - base;** }  
 }  
}

异步模式-生产者/消费者

1、概念

* 与前面的保护性暂停中的 GuardObject 不同，不需要产生结果和消费结果的线程一一对应；
* 消费队列可以用来平衡生产和消费的线程资源；
* 生产者仅负责产生结果数据，不关心数据该如何处理，而消费者专心处理结果数据；
* 消息队列是有容量限制的，满时不会再加入数据，空时不会再消耗数据；
* JDK 中各种阻塞队列，采用的就是这种模式；



消息队列有延时，不会被立刻消费，异步的。

2、生产者-消费者模式代码

【1】消息JavaBean

**package** com.concurrent.p3.message\_queue;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
*/\*\*  
 \* 消息的JavaBean  
 \*/*@Slf4j(topic = **"c.Message"**)  
**public class** Message {  
 **private static** Integer *id* = 1;  
 **private** String **message**;  
  
 **private static** Integer generateId() {  
 **return** *id*++;  
 }  
  
 **public** Message(String message) {  
 *generateId*();  
 **this**.**message** = message;  
 }  
  
 **public** Integer getId() {  
 **return** *id*;  
 }  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Message{"** +  
 **"id="** + *id* + **"\t"** +  
 **"message='"** + **message** + **'\''** +  
 **'}'**;  
 }  
}

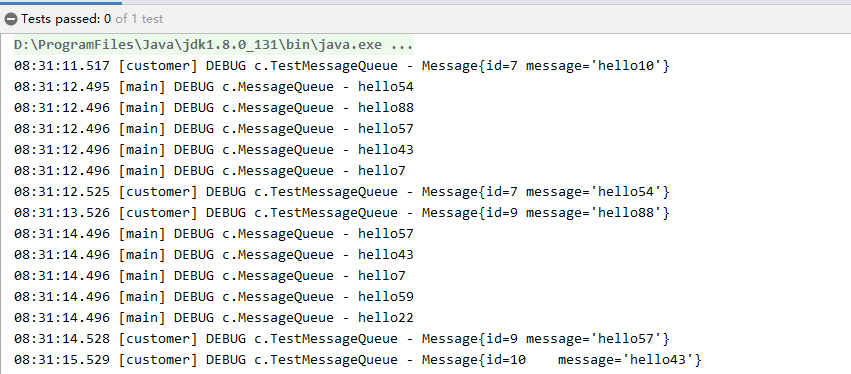
【2】消息队列

**package** com.concurrent.p3.message\_queue;  
  
**import** lombok.extern.slf4j.Slf4j;  
  
**import** java.util.LinkedList;  
**import** java.util.Random;  
  
@Slf4j(topic = **"c.MessageQueue"**)  
**public class** MessageQueue {  
 *//消息队列* **private** LinkedList<Message> **list** = **new** LinkedList<>();  
 *//队列容量* **private** Integer **capacity**;  
  
 *//构造方法，初始化容量* **public** MessageQueue(Integer capacity) {  
 **this**.**capacity** = capacity;  
 }  
  
 *//获取消息* **public** Message get() {  
 **synchronized** (**this**) {  
 *//如果队列为空，消费者就一直等待* **while** (**list**.size() == 0) {  
 **try** {  
 **this**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//如果队列不为空，则取出队头消息，并唤醒生产者* Message message = **list**.removeFirst();  
 **this**.notifyAll();  
 **return** message;  
 }  
 }  
  
 *//产生消息* **public void** complete() {  
 **synchronized** (**this**) {  
 *//如果队列满，生产者就等待* **while** (**list**.size() == **capacity**) {  
 **try** {  
 **this**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//如果队列不满，生产者就一直生产消息* Message message = **new** Message(**"hello"** + (**new** Random().nextInt(100) + 1));  
 *//将产生的消息放到队列尾* **list**.addLast(message);  
 *//唤醒消费者* **this**.notifyAll();  
 }  
 }  
  
 *//查看当前队列的情况* **public void** showList() {  
 **synchronized** (**this**) {  
 **list**.forEach((s) -> {  
 ***log***.debug(s.getMessage());  
 });  
 ***log***.debug(**"================="**);  
 }  
 }  
}

【3】测试代码，1个生成者，1个消费者

**package** com.concurrent.p3.message\_queue;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
@Slf4j(topic = **"c.TestMessageQueue"**)  
**public class** TestMessageQueue {  
  
 @Test  
 **public void** testMessageQueue() {  
 MessageQueue mq = **new** MessageQueue(5);  
 *//生产者线程* Thread procedure = **new** Thread(() -> {  
 **while** (**true**) {  
 mq.complete();  
 }  
 }, **"procedure"**);  
 procedure.start();  
  
 *//消费者线程* Thread customer = **new** Thread(() -> {  
 **while** (**true**) {  
 *//每个1秒，消费1次* **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 Message message = mq.get();  
 ***log***.debug(message.toString());  
 }  
 }, **"customer"**);  
 customer.start();  
  
 *//在主线程查看队列的情况* **while** (**true**) {  
 *//每个2秒查看消息队列的情况* **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 mq.showList();  
 }  
 }  
}

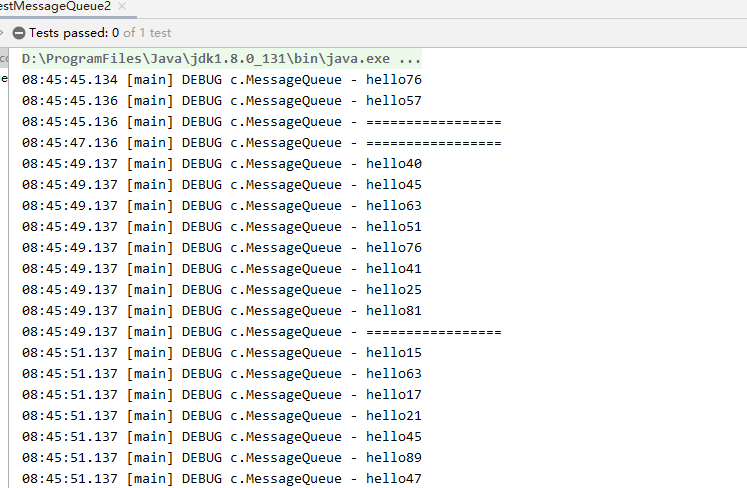
运行结果：



【4】测试代码，3个生产者，1个消费者

**package** com.concurrent.p3.message\_queue;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
@Slf4j(topic = **"c.TestMessageQueue"**)  
**public class** TestMessageQueue {  
  
 */\*\*  
 \* 1个生产者，1个消费者  
 \*/* @Test  
 **public void** testMessageQueue1() {  
 MessageQueue mq = **new** MessageQueue(5);  
 *//生产者线程* Thread procedure = **new** Thread(() -> {  
 **while** (**true**) {  
 mq.complete();  
 }  
 }, **"procedure"**);  
 procedure.start();  
  
 *//消费者线程* Thread customer = **new** Thread(() -> {  
 **while** (**true**) {  
 *//每个1秒，消费1次* **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 Message message = mq.get();  
 ***log***.debug(message.toString());  
 }  
 }, **"customer"**);  
 customer.start();  
  
 *//在主线程查看队列的情况* **while** (**true**) {  
 *//每个2秒查看消息队列的情况* **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 mq.showList();  
 }  
 }  
  
 */\*\*  
 \* 3个生产者，1个消费者  
 \*/* @Test  
 **public void** testMessageQueue2() {  
 *//3个生产者线程* List<Thread> procedureThreadList = **new** ArrayList<>();  
 *//消息队列* MessageQueue mq = **new** MessageQueue(10);  
 **for** (**int** i = 0; i < 3; i++) {  
 Thread p = **new** Thread(() -> {  
 **while** (**true**) {  
 mq.complete();  
 }  
 }, **"p"** + i + 1);  
 procedureThreadList.add(p);  
 }  
 **for** (Thread i : procedureThreadList) {  
 i.start();  
 }  
  
 *//1个消费者* Thread c1 = **new** Thread(()->{  
 **while** (**true**) {  
 Message message = mq.get();  
 *//log.debug(message.toString());* }  
 },**"c1"**);  
 c1.start();  
  
 *//在主线程查看队列的情况* **while** (**true**) {  
 *//每个2秒查看消息队列的情况* **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 mq.showList();  
 }  
 }  
}

运行结果：



park和unpark

1、基本使用

它们是 LockSupport 类中的方法

// 暂停当前线程

LockSupport.park();

// 恢复某个线程的运行

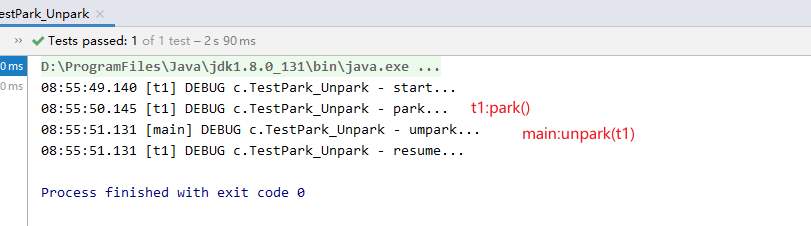
LockSupport.unpark(暂停线程对象)

先 park 再 unpark

2、代码

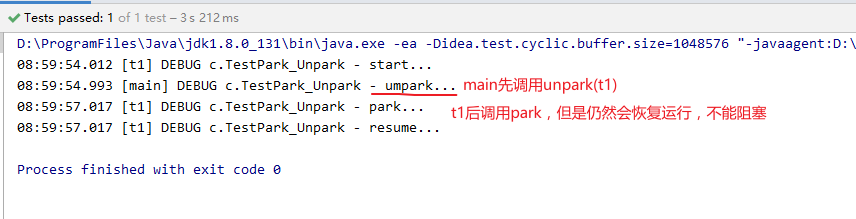
*/\*\*  
 \* 暂停当前线程  
 \* LockSupport.park();  
 \* 恢复某个线程的运行  
 \* LockSupport.unpark(暂停线程对象)  
 \*/*@Test  
**public void** testPark\_Unpark() {  
 *//线程1* Thread t1 = **new** Thread(() -> {  
 ***log***.debug(**"start..."**);  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"park..."**);  
 *//park* LockSupport.*park*();  
 ***log***.debug(**"resume..."**);  
 }, **"t1"**);  
 t1.start();  
  
 *//主线程中unpark* **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 ***log***.debug(**"umpark..."**);  
 LockSupport.*unpark*(t1);  
  
 *//等待t1执行完毕* **try** {  
 t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

运行结果：



unpark可以在park之前，之后调用。在park之前调用，可以恢复线程运行。



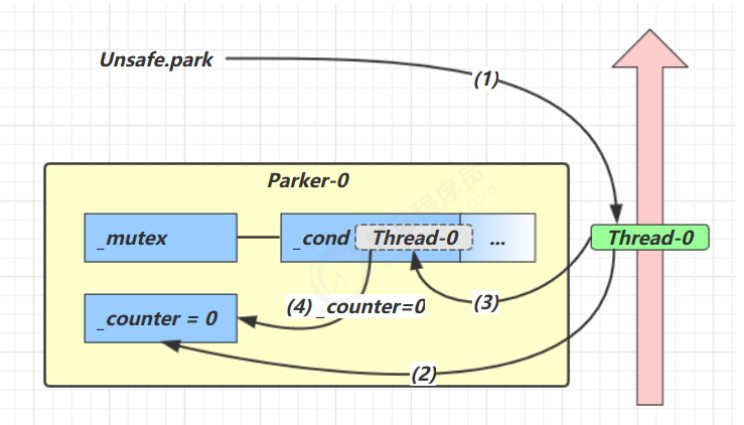


3、特点

与 Object 的 wait & notify 相比：

* wait，notify 和 notifyAll 必须配合 Object Monitor （重量级锁）一起使用，而 park，unpark 不必；
* park & unpark 是以线程为单位来【阻塞】和【唤醒】线程，而 notify 只能随机唤醒一个等待线程，notifyAll是唤醒所有等待线程，就不那么【精确】；
* park & unpark 可以先 unpark，而 wait & notify 不能先 notify；

4、park()/unpark()底层原理



每个线程都有自己的一个 Parker 对象，由三部分组成 \_counter ， \_cond 和 \_mutex 。

打个比喻：

* 线程就像一个旅人，Parker就像他随身携带的背包，\_cond条件变量就好比背包中的帐篷。\_counter 就好比背包中的备用干粮（0 为耗尽，1 为充足）。
* 调用 park 就是要看需不需要停下来歇息：

如果备用干粮耗尽（\_counter=0），那么钻进帐篷歇息;

如果备用干粮充足（\_counter=1），那么不需停留，继续前进;

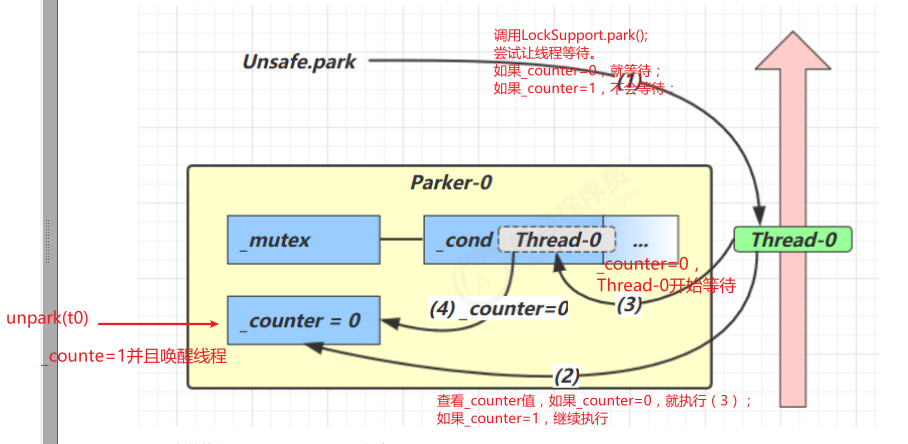
* 调用 unpark，就好比令干粮充足（令\_counter=1）：

如果这时线程还在帐篷，就唤醒让他继续前进；

如果这时线程还在运行，那么下次他调用 park 时，仅是消耗掉备用干粮，不需停留继续前进

因为背包空间有限，多次调用 unpark 仅会补充一份备用干粮。

【情况1】先park()



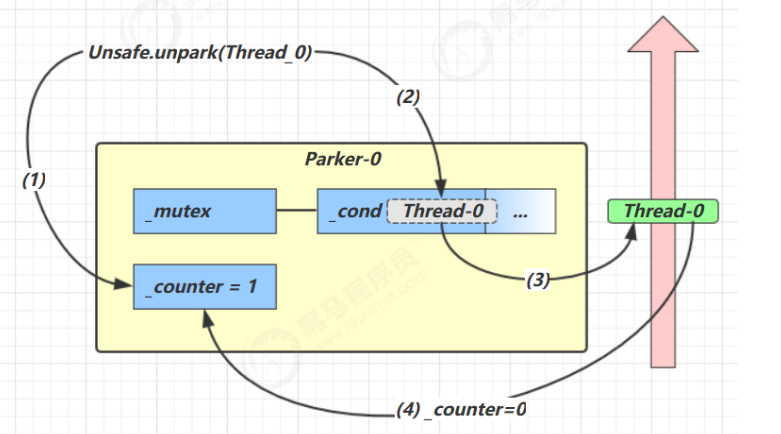
1. 当前线程调用 Unsafe.park() 方法

2. 检查 \_counter ，本情况为 0，这时，获得 \_mutex 互斥锁

3. 线程进入 \_cond 条件变量阻塞

4. 设置 \_counter = 0

【情况2】后调用unpark()



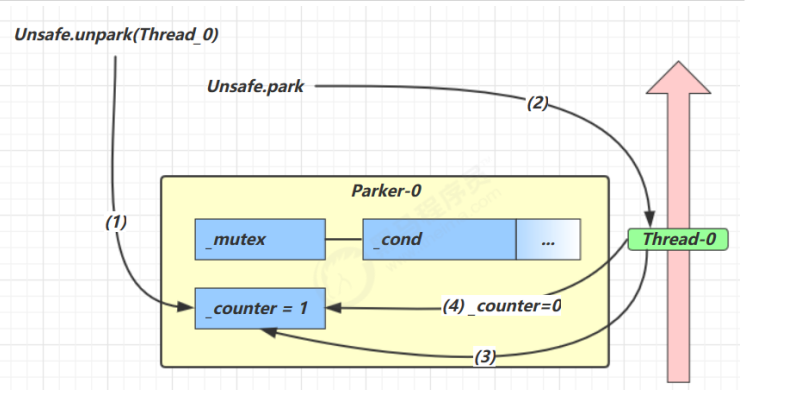
1. 调用 Unsafe.unpark(Thread\_0) 方法，设置 \_counter 为 1

2. 唤醒 \_cond 条件变量中的 Thread\_0

3. Thread\_0 恢复运行

4. 设置 \_counter 为 0

【情况3】先调用unpark()，再调用park()



1. 调用 Unsafe.unpark(Thread\_0) 方法，设置 \_counter 为 1

2. 当前线程调用 Unsafe.park() 方法

3. 检查 \_counter ，本情况为 1，这时线程无需阻塞，继续运行

4. 设置 \_counter 为 0

5、源码

park()

**public static void** park(Object blocker) {  
 Thread t = Thread.*currentThread*(); //获取当前线程  
 *setBlocker*(t, blocker);   
 ***UNSAFE*.park(false, 0L); //Unsafe对象调用本地park方法（C实现的）** *setBlocker*(t, **null**);  
}

setBolcker()

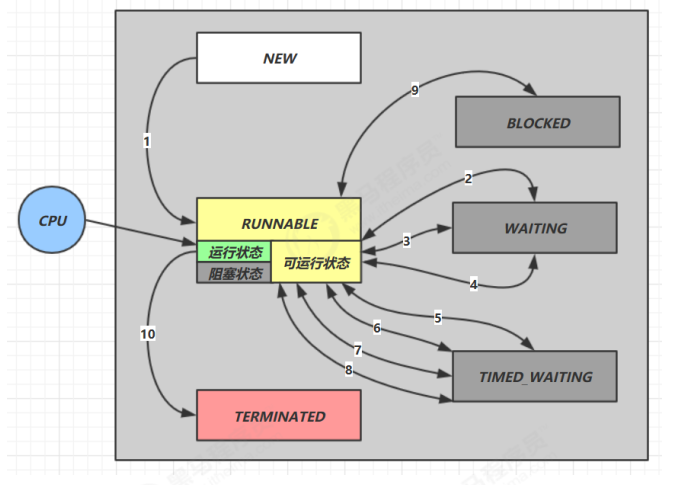
**private static void** setBlocker(Thread t, Object arg) {  
 *// Even though volatile, hotspot doesn't need a write barrier here.* ***UNSAFE*.putObject(t, *parkBlockerOffset*, arg); //本地方法**  
}

unpark()

**public static void** unpark(Thread thread) {  
 **if** (thread != **null**)  
 ***UNSAFE*.unpark(thread); //调用本地unpark方法**  
}

**public native void** unpark(Object var1);

重新理解线程状态转换



Thread t ;

情况1：NEW-->RUNNABLE

创建线程对象时，线程t的状态是NEW。

当调用t.start()时，线程状态由NEW-->RUNNABLE。

情况2：RUNNABLE<-->WAITING

t线程调用synchronized(obj)后，

调用 obj.wait() 时，t线程从RUNNABLE-->WAITING

调用 obj.notify()，obj.notifyAll()，t.interrupt()时：

如果竞争成功，t线程从WAITING-->RUNNABLE

如果竞争失败，t线程从WAITING-->BLOCKED（进入阻塞队列EntryList），等待调度器。

情况3：RUNNABLE<-->WAITING

当前线程调用t.join()后，当前线程从RUNNABLE-->WAITING。

注意：当前线程在t线程对象的监视器上等待。

t线程结束，或者调用了当前线程的interrupt()时，当前线程从WAITING-->RUNNABLE。

情况4：RUNNABLE<-->WAITING

当前线程调用LockSupport.park()，会让当前线程从RUNNABLE-->WAITING

调用了线程的LockSupport.unpark(目标线程)，或者调用了目标线程的interrupt()方法，会让目标线程从WAITING-->RUNNABLE。

情况5：RUNNABLE<-->TIMED\_WAITING

t线程调用synchronized(obj)获取了对象锁后：

调用了obj.wait(long n)后，t线程从RUNNABLE-->TIMED-WAITING。

t线程等待了n毫秒后，或者调用了obj.notify()，obj.notifyAll()，t.interrupt()时：

如果竞争成功，t线程从TIMED\_WAITING-->RUNNABLE

如果竞争失败，t线程从TIMED\_WAITING-->BLOCKED（进入阻塞队列EntryList），等待调度器。

情况6：RUNNABLE<-->TIMED\_WAITING

当前线程调用 t.join(long n)，当前线程从RUNNABLE<-->TIMED\_WAITING

注意：当前线程在t线程的监视器上等待。

当前线程等待了n毫秒或者t线程结束，或者调用了当前线程的interrupt()时，当前线程从TIMED\_WAITING-->RUNNABLE。

情况7：RUNNABLE<-->TIMED\_WAITING

当前线程调用了Threads.sleep(long n); 当前线程从RUNNABLE-->TIMED\_WAITING

当前线程等待时间超过了n毫秒，当前线程从TIMED\_WAITING-->RUNNABLE

情况8：RUNNABLE<-->TIMED\_WAITING

当前线程调用 LockSupport.parkNanos(long nanos) 或 LockSupport.parkUntil(long millis) 时，当前线

程从 RUNNABLE --> TIMED\_WAITING

调用 LockSupport.unpark(目标线程) 或调用了线程 的 interrupt() ，或是等待超时，会让目标线程从

TIMED\_WAITING--> RUNNABLE

情况9：RUNNABLE<-->BLOCKED

t 线程用 synchronized(obj) 获取了对象锁时如果竞争失败，从 RUNNABLE --> BLOCKED

持 obj 锁线程的同步代码块执行完毕，会唤醒该对象上所有 BLOCKED 的线程重新竞争，如果其中 t 线程竞争成功，从 BLOCKED --> RUNNABLE ，其它失败的线程仍然 BLOCKED

情况10：RUNNABLE <--> TERMINATED

当前线程所有代码运行完毕，进入 TERMINATED。

多把锁

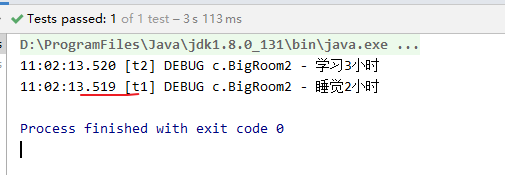
锁住大对象，会造成低并发程度。

*/\*\*  
 \* 直接锁大对象  
 \*/*@Slf4j(topic = **"c.BigRoom"**)  
**class** BigRoom {  
 **public void** sleep() {  
 **synchronized** (**this**) {  
 ***log***.debug(**"睡觉2小时"**);  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 **public void** study() {  
 **synchronized** (**this**) {  
 ***log***.debug(**"学习3小时"**);  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

锁住小对象，并发程度高。

*/\*\*  
 \* 优化  
 \* 加2个不同的锁  
 \*/*@Slf4j(topic = **"c.BigRoom2"**)  
**class** BigRoom2 {  
 **private static final** Object ***studyRoom*** = **new** Object();  
 **private static final** Object ***sleepRoom*** = **new** Object();  
  
 **public void** sleep() {  
 **synchronized** (***studyRoom***) {  
 ***log***.debug(**"睡觉2小时"**);  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 **public void** study() {  
 **synchronized** (***sleepRoom***) {  
 ***log***.debug(**"学习3小时"**);  
 **try** {  
 Thread.*sleep*(3000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

运行结果：



将锁的粒度细分

* 好处，是可以增强并发度
* 坏处，如果一个线程需要同时获得多把锁，就容易发生死锁

活跃性

1、死锁概念

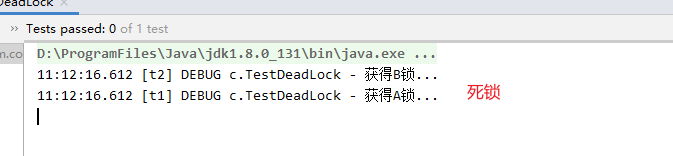
有这样的情况：一个线程需要同时获取多把锁，这时就容易发生死锁。

t1线程获得A对象锁，接下来想获取B对象的锁， t2线程获得B对象锁，接下来想获取A对象的锁。

2、死锁代码

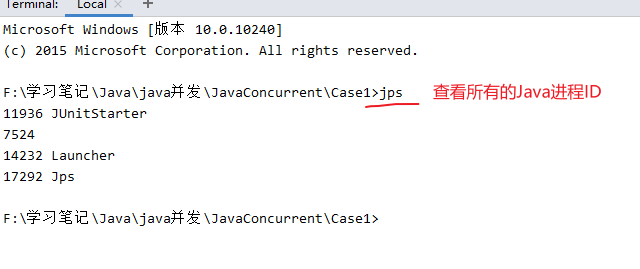
**static final** Object ***A*** = **new** Object();  
**static final** Object ***B*** = **new** Object();  
  
@Test  
**public void** testDeadLock() {  
 *//线程1获得A锁，又想获得B锁* Thread t1 = **new** Thread(() -> {  
 **synchronized** (***A***) {  
 ***log***.debug(**"获得A锁..."**);  
 **synchronized** (***B***) {  
 ***log***.debug(**"获得B锁..."**);  
 }  
 }  
 }, **"t1"**);  
 t1.start();  
 *//线程2获得B锁，又想获得A锁* Thread t2 = **new** Thread(() -> {  
 **synchronized** (***B***) {  
 ***log***.debug(**"获得B锁..."**);  
 **synchronized** (***A***) {  
 ***log***.debug(**"获得A锁..."**);  
 }  
 }  
 }, **"t2"**);  
 t2.start();  
 **try** {  
 t1.join();  
 t2.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

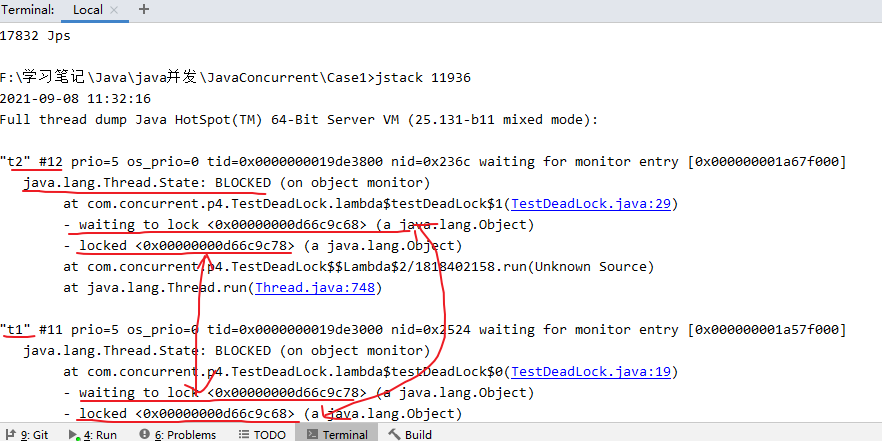
运行结果：



1. 定位死锁

检测死锁可以使用 jconsole工具，或者使用 jps 定位进程 id，再用 jstack 定位死锁：







1. 哲学家进餐问题