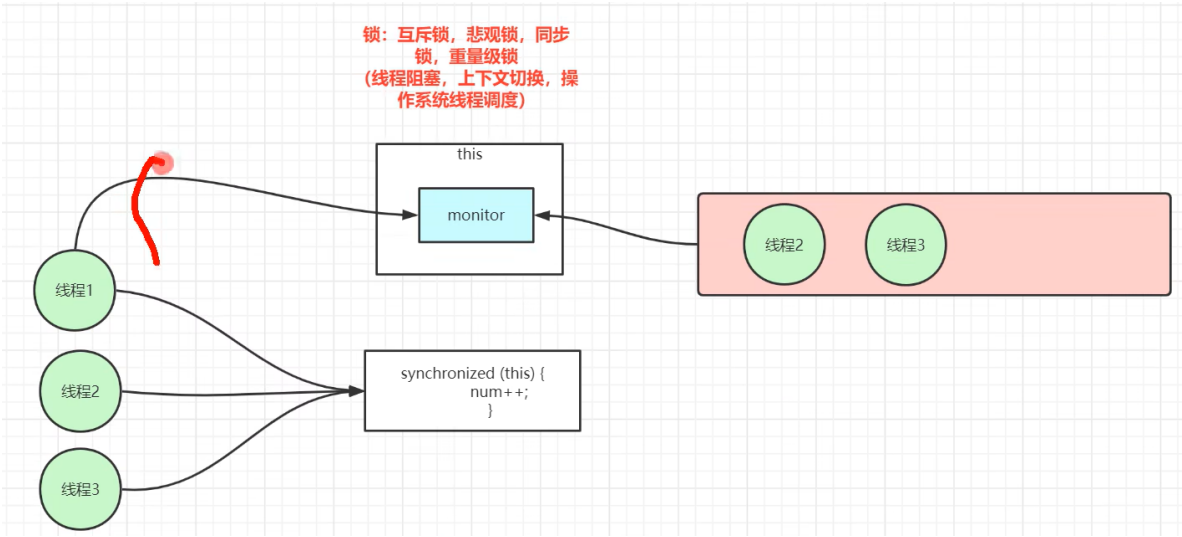
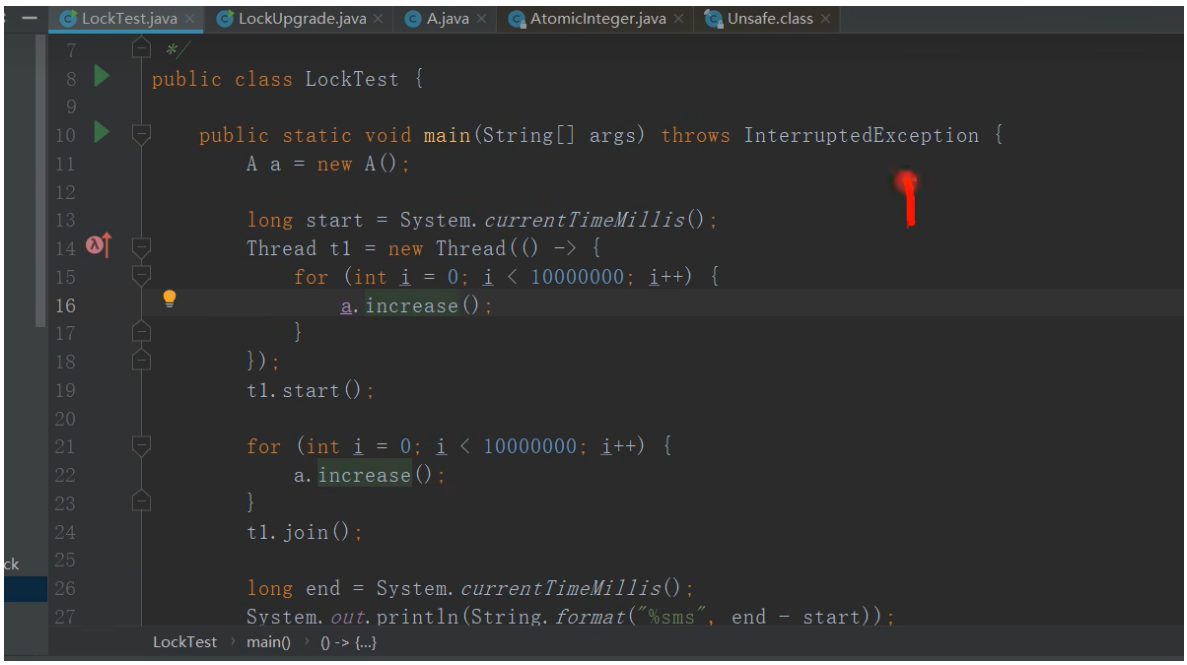
1. Synchronized锁：异步锁，同步锁，悲观锁，重量级锁

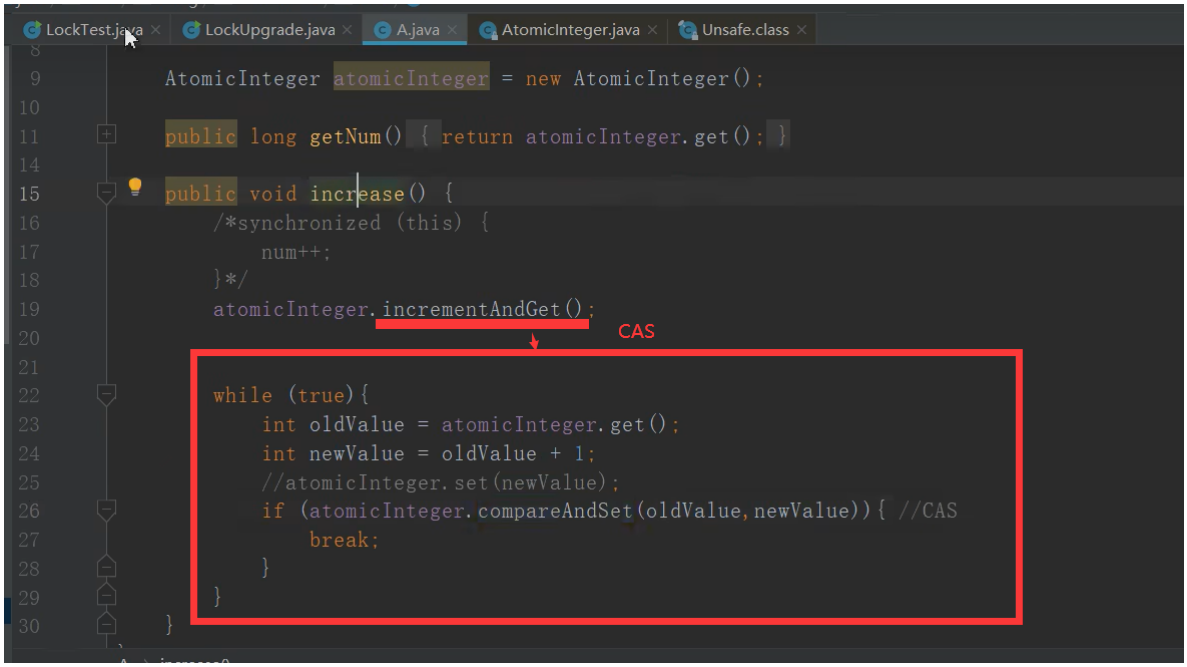
当有多个线程时，会有等待队列，就会发生线程阻塞，上下文切换，操作系统线程调度



2.CAS（比较并赋值）：无锁，自旋锁，乐观锁，轻量级锁

compare and swap | compare and set





3. ABA问题

是CAS机制中出现的一个问题，这里先说明一下CAS和原子操作：

CAS

Compare And Swap，比较并交换。是java.util.concurrent包实现的区别于synchronized同步锁的一种乐观锁。CAS有三个操作数，内存值V，预期值A，要修改的值B，当且仅当内存值V与预期值A相等时，将内存值V修改为B，否则什么也不做。

原子操作

“原子”代表最小的执行单位，该操作在执行完毕前不会被任何其他任务或者事件打断。
AtomicInteger类的compareAndSet通过原子操作实现了CAS操作，最底层基于汇编语言实现。

1.什么是ABA问题?

一个线程把数据A变成了B，然后又重新变成了A，此时另一个线程读取该数据的时候，发现A没有变化，就误认为是原来的那个A，但是此时A的一些属性或状态已经发生过变化。
下面一段代码对ABA问题进行重现：

package com.lpl;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class ABA {

```
private static AtomicInteger index = new AtomicInteger(10);

public static void main(String[] args) {
    //张三线程去修改index的值
    new Thread(() -> {
```

```
        //通过CAS自旋算法锁修改index的值
        index.compareAndSet(10, 11);
        index.compareAndSet(11, 10);
        System.out.println(Thread.currentThread().getName() + "  10 -> 11 ->
10");
    }, "张三").start();

    //李四线程去读取内存值并设置新值
    new Thread(() -> {
        try{
            //线程休眠2秒
            TimeUnit.SECONDS.sleep(2);
            //判断是否修改成功
            boolean isSuccess = index.compareAndSet(10, 12);
            System.out.println(Thread.currentThread().getName() + "  index是否是
预期的值：" + isSuccess + "，设置的新值是：" + index.get());
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "李四").start();
}
```

}
程序运行的结果为：

这里正常情况下在张三对index进行了操作后虽然index的值没有发生变化，但是李四再次拿到并进行操作的数据已经不是原来最初的数据了，这就产生了ABA问题。

2.解决方案

要解决ABA问题，可以增加一个版本号，当内存位置V的值每次被修改后，版本号都加1。

（1）第一种方式，使用AtomicStampedReference对象

AtomicStampedReference内部维护了对象值和版本号，在创建AtomicStampedReference对象时，需要传入初始值和初始版本号，当AtomicStampedReference设置对象值时，对象值以及状态戳都必须满足期望值，写入才会成功。
对上面程序进行修改：

```
package com.lpl;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicStampedReference;

public class ABA {private static AtomicStampedReference<Integer>
atomicStampedReference = new AtomicStampedReference<>(10, 1);

    public static void main(String[] args) {
        //张三线程去修改参考对象的值
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "  拿到的当前时
间戳版本号为：" + atomicStampedReference.getStamp());
```

```
            //休眠1秒，为了让李四线程也拿到同样的初始版本号
            try{
                TimeUnit.SECONDS.sleep(1);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
            //通过CAS自旋算法锁修改index的值
            atomicStampedReference.compareAndSet(10, 11,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);
            atomicStampedReference.compareAndSet(11, 10,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName() + "  10 -> 11 ->
10");
        }, "张三").start();

        //李四线程去读取内存值并设置新值
        new Thread(() -> {
            try{
                int stamp = atomicStampedReference.getStamp();
                System.out.println(Thread.currentThread().getName() + "  拿到的当
前时间戳版本号为：" + stamp);
                //线程休眠2秒，为了让张三线程完成ABA操作
                TimeUnit.SECONDS.sleep(2);
                //判断是否修改成功
                boolean isSuccess = atomicStampedReference.compareAndSet(10, 12,
stamp, atomicStampedReference.getStamp() + 1);
                System.out.println(Thread.currentThread().getName() + "  最新版本
号：" + atomicStampedReference.getStamp() + "，是否修改成功：" + isSuccess + "，当前
值是：" + atomicStampedReference.getReference());
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "李四").start();
    }
}
```

程序运行结果：

线程张三完成CAS操作，最新版本号已经变成3，与线程李四之前拿到的版本号1不相等，所以操作失败。
有时候，我们并不关心引用变量更改了几次，只是单纯的关心是否更改过，所以就有了AtomicMarkableReference对象，可以标识引用变量是否被更改过。

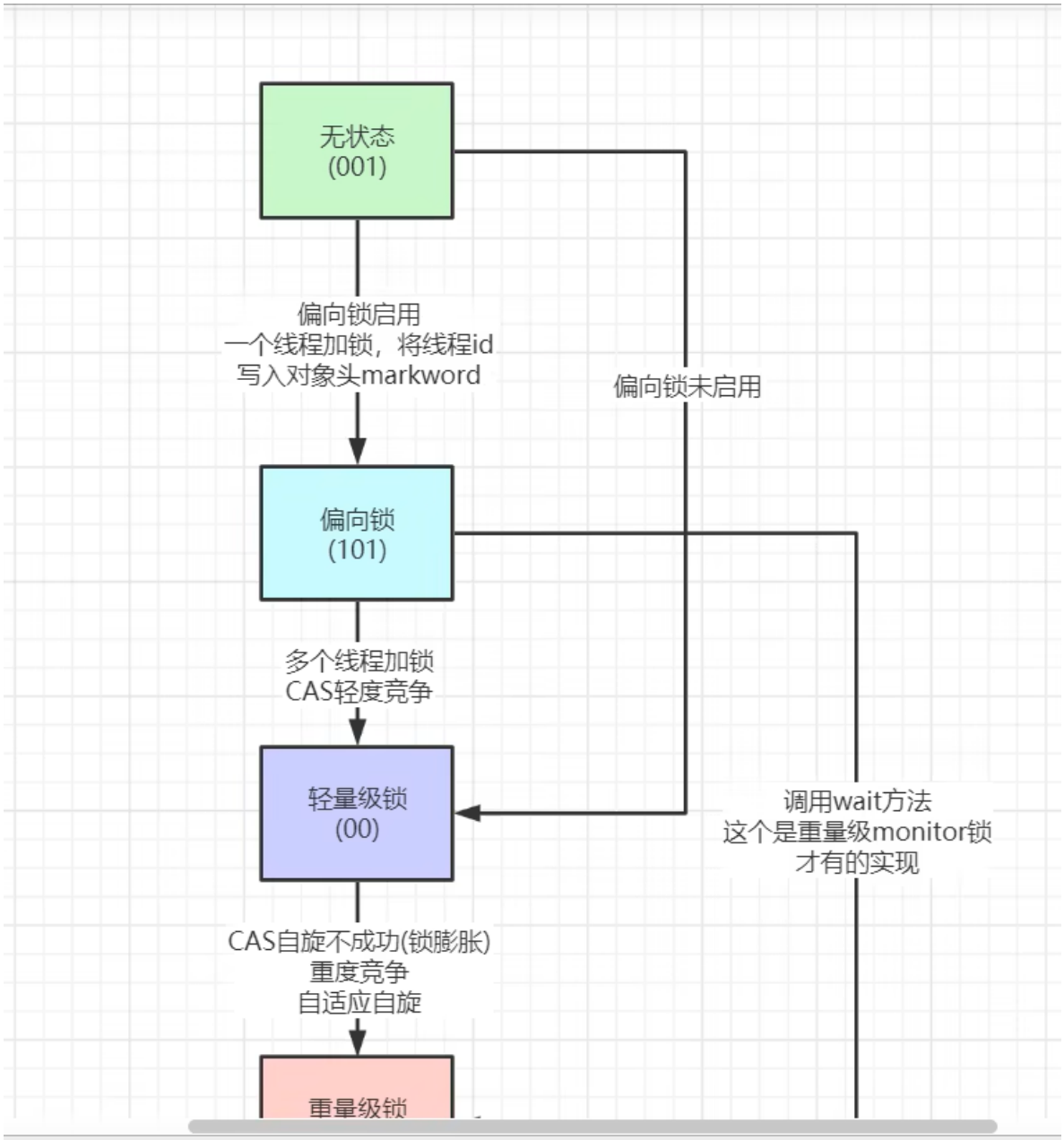
（2）第二种方式，使用AtomicMarkableReference对象

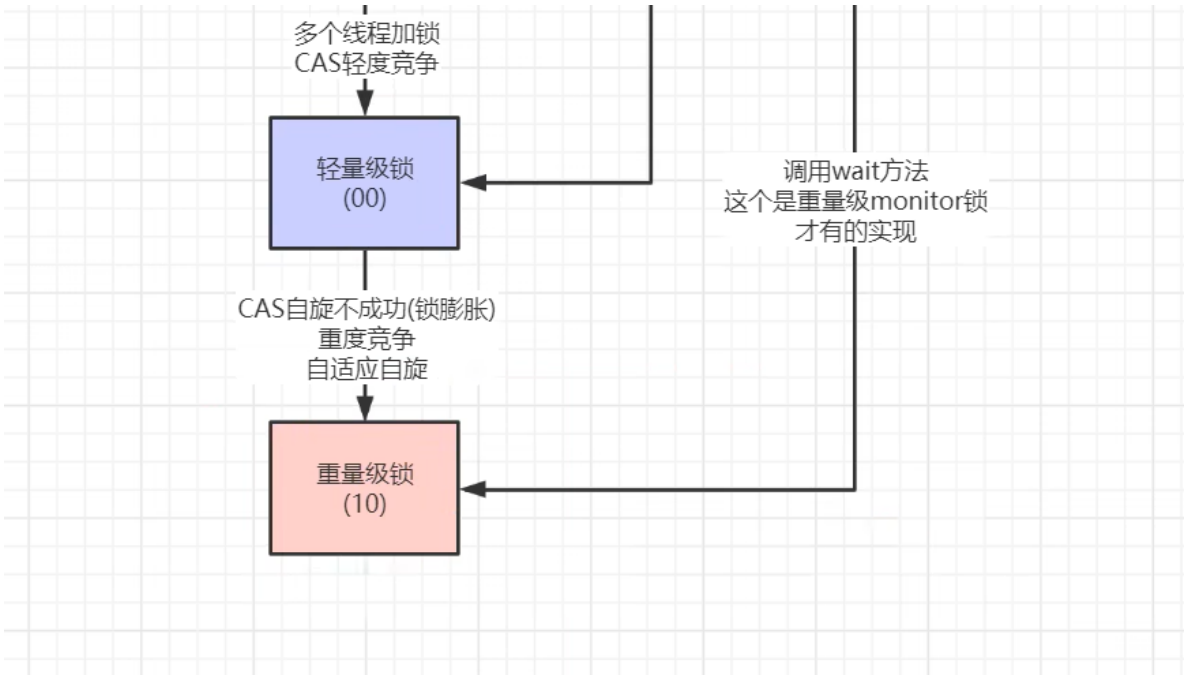
修改代码为：

```
package com.lpl;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicMarkableReference;

public class ABA {
    private static AtomicMarkableReference<Integer> atomicMarkableReference =
new AtomicMarkableReference<Integer>(10, false);
```

```
    public static void main(String[] args) {
        //张三线程去修改参考对象的值
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "  当前参考对象是否被修改：" + atomicMarkableReference.isMarked());

            //休眠1秒，为了让李四线程也拿到是否被修改的标识
            try{
                TimeUnit.SECONDS.sleep(1);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
            //通过CAS自旋算法锁修改index的值
            atomicMarkableReference.compareAndSet(10, 11, atomicMarkableReference.isMarked(), true);
            atomicMarkableReference.compareAndSet(11, 10, atomicMarkableReference.isMarked(), true);
            System.out.println(Thread.currentThread().getName() + "  10 -> 11 -> 10");
        }, "张三").start();

        //李四线程去读取内存值并设置新值
        new Thread(() -> {
            try{
                boolean isMarked = atomicMarkableReference.isMarked();
                System.out.println(Thread.currentThread().getName() + "  当前参考对象是否被修改：" + isMarked);
                //线程休眠2秒，为了让张三线程完成ABA操作
                TimeUnit.SECONDS.sleep(2);
                //判断是否修改成功
                boolean isSuccess = atomicMarkableReference.compareAndSet(10, 12, isMarked, true);
                System.out.println(Thread.currentThread().getName() + "  当前修改状态为：" + atomicMarkableReference.isMarked() + "，是否修改成功：" + isSuccess + "，当前值是：" + atomicMarkableReference.getReference());
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "李四").start();
    }
}
```

4.JDK1.6后的锁优化


无状态
(001)
偏向锁启用
一个线程加锁，将线程id
写入对象头markword
偏向锁未启用
偏向锁
(101)
多个线程加锁
CAS轻度竞争
轻量级锁
(00)
调用wait方法
这个是重量级monitor锁
才有的实现
CAS自旋不成功(锁膨胀)
重度竞争
自适应自旋
重量级锁


多个线程加锁
CAS轻度竞争
轻量级锁
(00)
调用wait方法
这个是重量级monitor锁
才有的实现
CAS自旋不成功(锁膨胀)
重度竞争
自适应自旋
重量级锁
(10)

轻量级锁一定比重量级锁性能高吗

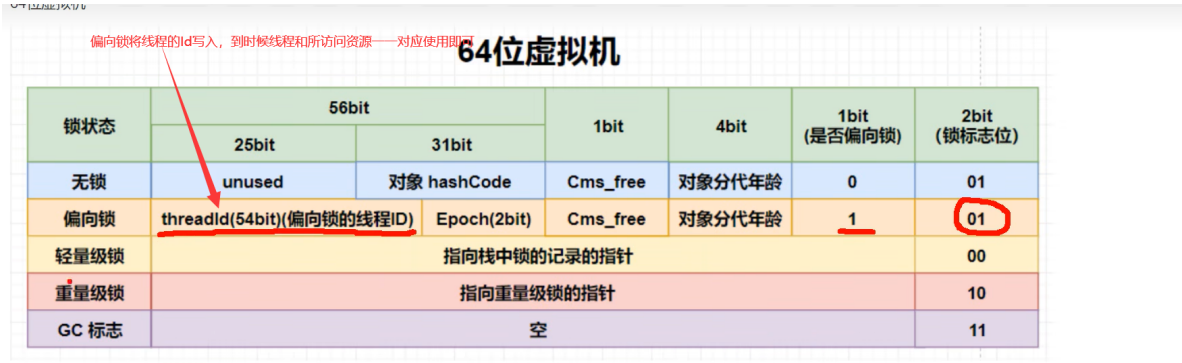
答：不一定，看自旋时间（while循环进行CompareAndSet自旋，如果线程多的话，会很多做无用功）和唤醒的代价谁高，线程超过10个，直接重量级锁

5.从hostpot底层对象结构来理解锁膨胀的升级过程

5.1 一个对象的组成成分

对象的组成有3个部分：**对象头**；**实例数据**；**对齐填充字节。**其中对象头包含3个部分：**Mark Word**；**指向类的指针**；**数组长度（如果当前对象不是数组则没有此部分）**

5.1.1 Mark Word



锁状态	56bit		1bit	4bit	1bit (是否偏向锁)	2bit (锁标志位)
	25bit	31bit				
无锁	unused	对象 hashCode	Cms_free	对象分代年龄	0	01
偏向锁	threadId(54bit)(偏向锁的线程ID)	Epoch(2bit)	Cms_free	对象分代年龄	1	01
轻量级锁	指向栈中锁的记录的指针					00
重量级锁	指向重量级锁的指针					10
GC 标志	空					11

ps：从上图可以看到，Mark Work存储的是对象自身运行时的数据，比如：锁标志位；是否偏向锁；GC分代年龄；对象的hashCode；获取到该锁的线程的线程ID；偏向时间戳（Epoch）等等。

5.2 无锁到偏向锁：

jvm默认延时4秒自动开启偏向锁