synchronized与Lock锁（对比和实现细节）

synchronized

`synchronized`关键字在需要原子性、可见性和有序性这三种特性的时候都可以作为其中一种解决方案，看起来是"万能"的。的确，大部分并发控制操作都能使用synchronized来完成。

上面这段话出自《深入理解Java虚拟机》，在并发编程中，经常会用到synchronized，下面来看一下synchronized的具体细节

用法

可以同步方法或同步代码块，被修饰的部分，在同一时间只能被单线程访问

```
public class Concurrent {

    //同步方法
    public synchronized void syn1()
    {
        System.out.println("this is synchronized method !");
    }

    //同步代码块
    public void syn2()
    {
        System.out.println("this is not synchronized block !");
        synchronized (Concurrent.class)
        {
            System.out.println("this is synchronized block !");
        }
    }
}
```

实现原理

- 目的：想要保证一个共享资源在同一时间只会被一个线程访问到时
- 对于同步方法，JVM采用`ACC_SYNCHRONIZED`标记符来实现同步。
    - 同步方法是隐式同步。在该方法的常量池中会有一个`ACC_SYNCHRONIZED`标志，当被访问时，首先检查该标志；若有该标志，需要先获得监视器锁，然后开始执行方法，执行后释放锁
    - 若有线程已经获得了监视器锁，其他线程来请求执行该方法就会被阻塞
    - 方法执行过程中出现异常，抛出前会自动释放监视器锁
- 对于同步代码块，JVM采用`monitorenter`、`monitorexit`两个指令来实现同步
    - `monitorenter`可以看作加锁，`monitorexit`看作释放锁
    - 每个对象维护一个记录加锁次数的计数器
    - 同一个线程多次获得同一个对象锁时，计数器可以自增多次，释放锁时，计数器自减，直至为0，才释放锁
- 以上都是基于`Monitor`实现的，在Java虚拟机(HotSpot)中，`Monitor`是基于C++实现的，由`ObjectMonitor`实现

原子性

在Java中，为了保证原子性，提供了两个高级的字节码指令`monitorenter`和`monitorexit`

这两个字节码指令，在Java中对应的关键字就是`synchronized`

可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

保证可见性的一条规则：对一个变量解锁之前，必须先把此变量同步回主存中。这样解锁后，后续线程就可以访问到被修改后的值

这就保证了`synchronized`关键字的可见性

有序性

`synchronized`是无法禁止指令重排和处理器优化的

Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的

不管怎么重排序（编译器和处理器为了提高并行度），单线程程序的执行结果都不能被改变

因为被`synchronized`锁住的期间，只有一个线程可以进行访问，可以认为保证了有序性

锁优化

JDK1.6之后对锁进行了很多优化，出现了轻量级锁、偏向锁、锁消除、适应性自旋锁和锁粗化

- **自旋锁**
  - 通常共享数据的锁定状态只会持续很短一段时间，为了这段时间去挂起和恢复线程不值得
  - 让后面请求锁的线程稍等，但不放弃处理器的执行时间，看看持有锁的线程能否狠快释放，为了让该线程等待，只须让线程执行一个忙循环（自旋）
  - 但必须是多处理器多核心
  - 不能代替阻塞，无法预测这个等待的时间有多久
- **自适应自旋**
  - 自旋刚刚成功获得锁，且持有锁的线程正在运行，认为很有可能自旋再次成功，允许自旋等待时间更长
  - 经常自旋失败，减少等待时间
- **锁消除**
  - 虚拟机即时编译器运行时，对一些代码要求同步，但是对被检测到不可能存在共享数据竞争的锁进行消除
  - 判断依据来自逃逸分析的数据支持
  - 判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的
- **锁粗化（膨胀）**
  - 原则本来是尽量将同步块的作用范围限制的小
  - 但如果一系列连续操作都对同一个对象反复加锁解锁，甚至在循环体内加锁解锁，那么频繁的进行互斥同步操作也会导致不必要的性能损耗
  - 把加锁范围适当扩展粗化，让加锁解锁操作次数尽量少
- **轻量级锁**

  设计初衷：没有多线程竞争的前提下，减少传统重量级锁使用操作系统互斥量产生的性能消耗

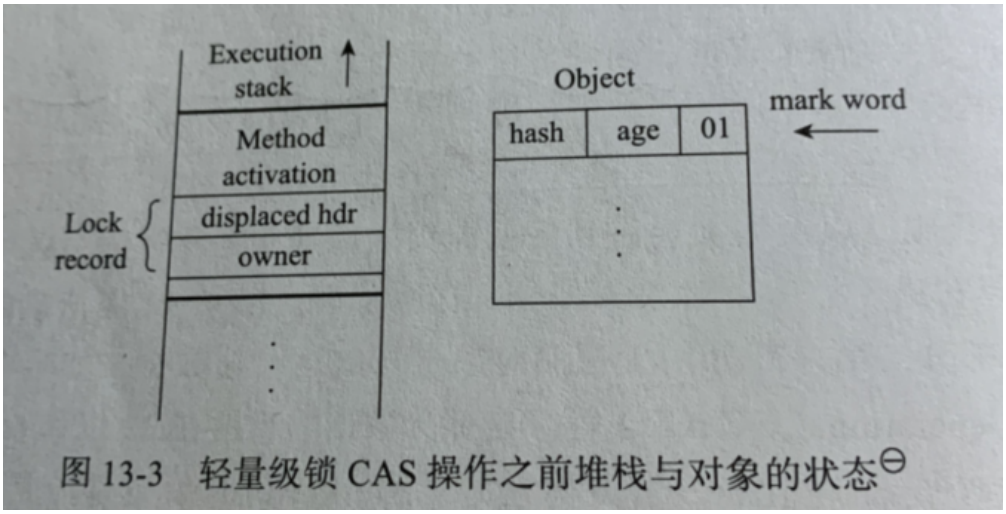
- HotSpot虚拟机对象头两部分
  - 存储对象自身的运行时数据，如哈希码，GC分代年龄等（Mark Word）
  - 指向方法区对象类型数据的指针，数组对象还有额外部分存储数组长度
- 轻量级锁工作过程：
  - 代码即将进入同步块时，如果此同步对象没有被锁定（锁标志为01），虚拟机先在当前线程栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word拷贝
  - 

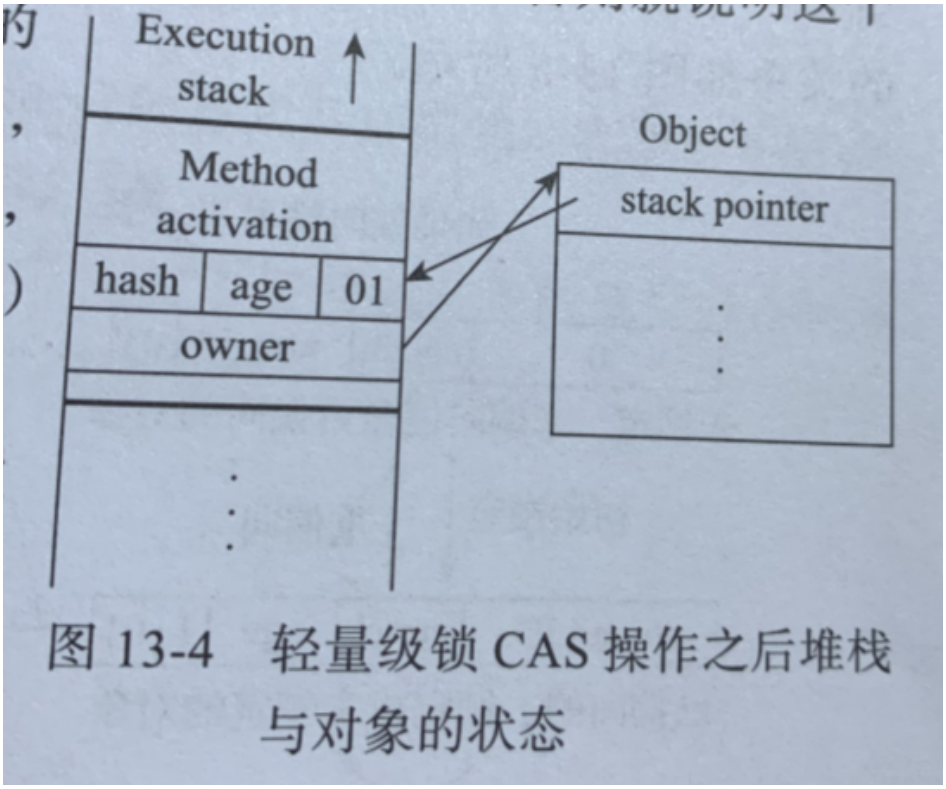
图 13-3 轻量级锁 CAS 操作之前堆栈与对象的状态
  - 虚拟机使用CAS操作尝试把对象的Mark Word更新为指向Lock Record的指针；
    - 更新成功，代表该线程拥有了这个对象的锁，且对象的Mark Word标志位转变为00，表示此对象属于轻量级锁定状态
    - 

图 13-4 轻量级锁 CAS 操作之后堆栈与对象的状态
    - 如果这个更新操作失败了，那就意味着至少存在一条线程与当前线程竞争获取该对象的锁
    - 虚拟机会检查对象的Mark Word是否指向当前线程的栈帧，如果是，说明当前线程已经拥有了这个对象的锁，直接进入同步块继续执行即可，否则就说明这个锁对象已经被其他线程抢占了
    - 两条以上的线程争用同一个锁，轻量级锁必须膨胀为重量级锁（互斥量）
  - 提升性能的依据
    - 绝大部分锁，整个同步周期内都是不存在竞争的

        - 如果竞争过多，还不如重量级锁
- 偏向锁
    - 消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能
    - **偏向锁的目标是，减少无竞争且只有一个线程使用锁的情况下，使用轻量级锁产生的性能消耗**
    - 偏向锁假定只有将来第一个申请锁的线程会使用锁，后来的都不使用
        - 因此，只需要在Mark Word中用CAS算法记录owner，若记录成功，则偏向锁获取成功，记录锁状态为偏向锁，只要当前线程为owner就可以零成本获取锁
        - 否则，就说明有其他线程进行竞争，膨胀为轻量级锁

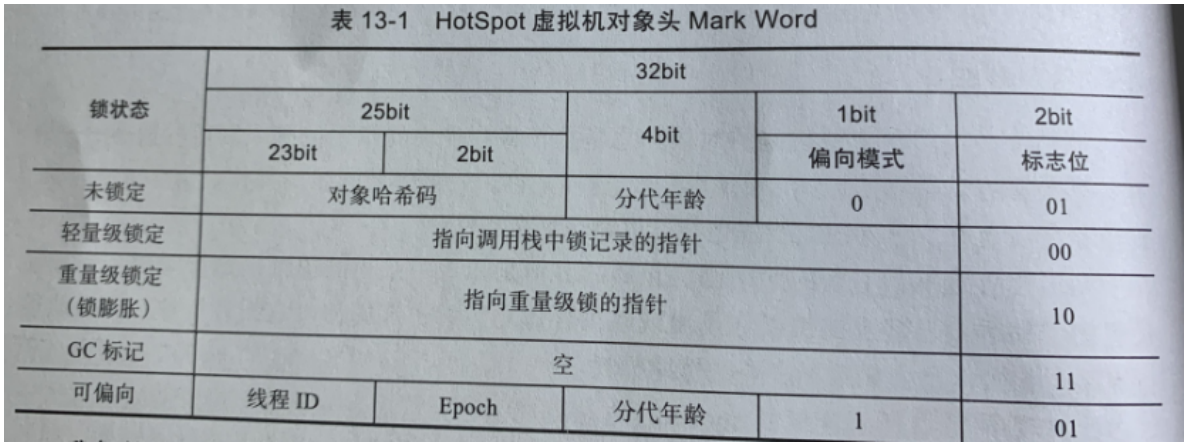


表 13-1 HotSpot 虚拟机对象头 Mark Word

锁状态	32bit				
	25bit		4bit	1bit	2bit
	23bit	2bit		偏向模式	标志位
未锁定	对象哈希码		分代年龄	0	01
轻量级锁定	指向调用栈中锁记录的指针				00
重量级锁定（锁膨胀）	指向重量级锁的指针				10
GC 标记	空				11
可偏向	线程 ID	Epoch	分代年龄	1	01

Lock锁

以ReentrantLock为例，实现过程可以归纳为：

- 锁标志计数值 `state`
- 双向链表
- CAS操作
- 自旋

用法

```
import java.util.concurrent.locks.ReentrantLock;

public class Concurrent {

    public static void main(String[] args) throws InterruptedException {
        final int[] counter = {0};

        ReentrantLock lock = new ReentrantLock();

        for(int i = 0;i<50;i++)
        {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    lock.lock();
                    try{
                        int a = counter[0];
                        counter[0] = a+1;
```

```
                        System.out.println(
Thread.currentThread().getName()+".counter is:"+counter[0]);
                    }
                    finally {
                        lock.unlock();
                    }
                }
            }).start();
        }
        Thread.sleep(5000);
        System.out.println(counter[0]);
    }
}
```

开50个线程同时更新counter，输出结果如下：

```
Thread-0.counter is:1
Thread-4.counter is:2
Thread-5.counter is:3
Thread-1.counter is:4
Thread-3.counter is:5
Thread-15.counter is:6
Thread-13.counter is:7
Thread-8.counter is:8
Thread-9.counter is:9
Thread-11.counter is:10
Thread-12.counter is:11
Thread-14.counter is:12
Thread-16.counter is:13
Thread-17.counter is:14
Thread-19.counter is:15
Thread-18.counter is:16
Thread-20.counter is:17
Thread-21.counter is:18
Thread-22.counter is:19
Thread-24.counter is:20
Thread-23.counter is:21
Thread-26.counter is:22
Thread-25.counter is:23
Thread-27.counter is:24
Thread-28.counter is:25
Thread-29.counter is:26
Thread-30.counter is:27
Thread-31.counter is:28
Thread-32.counter is:29
Thread-33.counter is:30
Thread-36.counter is:31
Thread-34.counter is:32
Thread-35.counter is:33
Thread-37.counter is:34
Thread-10.counter is:35
Thread-39.counter is:36
Thread-40.counter is:37
Thread-41.counter is:38
Thread-42.counter is:39
Thread-44.counter is:40
Thread-43.counter is:41
```

```
Thread-46.counter is:42
Thread-45.counter is:43
Thread-48.counter is:44
Thread-47.counter is:45
Thread-49.counter is:46
Thread-2.counter is:47
Thread-38.counter is:48
Thread-6.counter is:49
Thread-7.counter is:50
50
```

实现原理

构造函数：

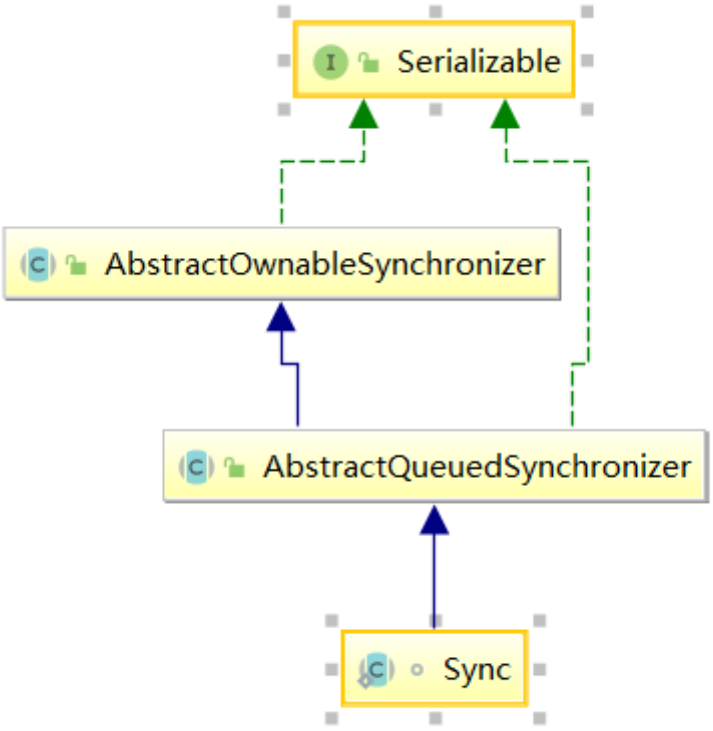
```
    /**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     */
    public ReentrantLock() {
        sync = new NonfairSync();
    }
```

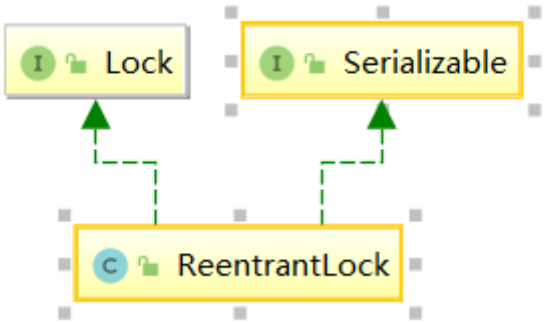
```
/** Synchronizer providing all implementation mechanics */
private final Sync sync;
```

```
abstract static class Sync extends AbstractQueuedSynchronizer
```

```
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable
```

继承关系





默认不带参数的构造函数使用非公平锁

`This is equivalent to using {@code ReentrantLock(false)}.`

AbstractQueuedSynchronizer

观察源码发现这个类就是一个 **双向链表**

数据结构主要包括：一个 int类型状态 + 双向链表

lock.lock()获取锁

```
public void lock() {
    sync.acquire(1);
}
```

```
/**
 * Acquires in exclusive mode, ignoring interrupts.  Implemented
 * by invoking at least once {@link #tryAcquire},
 * returning on success.  Otherwise the thread is queued, possibly
 * repeatedly blocking and unblocking, invoking {@link
 * #tryAcquire} until success.  This method can be used
```

```
     * to implement method {@link Lock#lock}.
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     */
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

- tryAcquire：会尝试再次通过CAS获取一次锁。
- addWaiter：将当前线程加入上面锁的双向链表（等待队列）中
- acquireQueued：通过自旋，判断当前队列节点是否可以获取锁

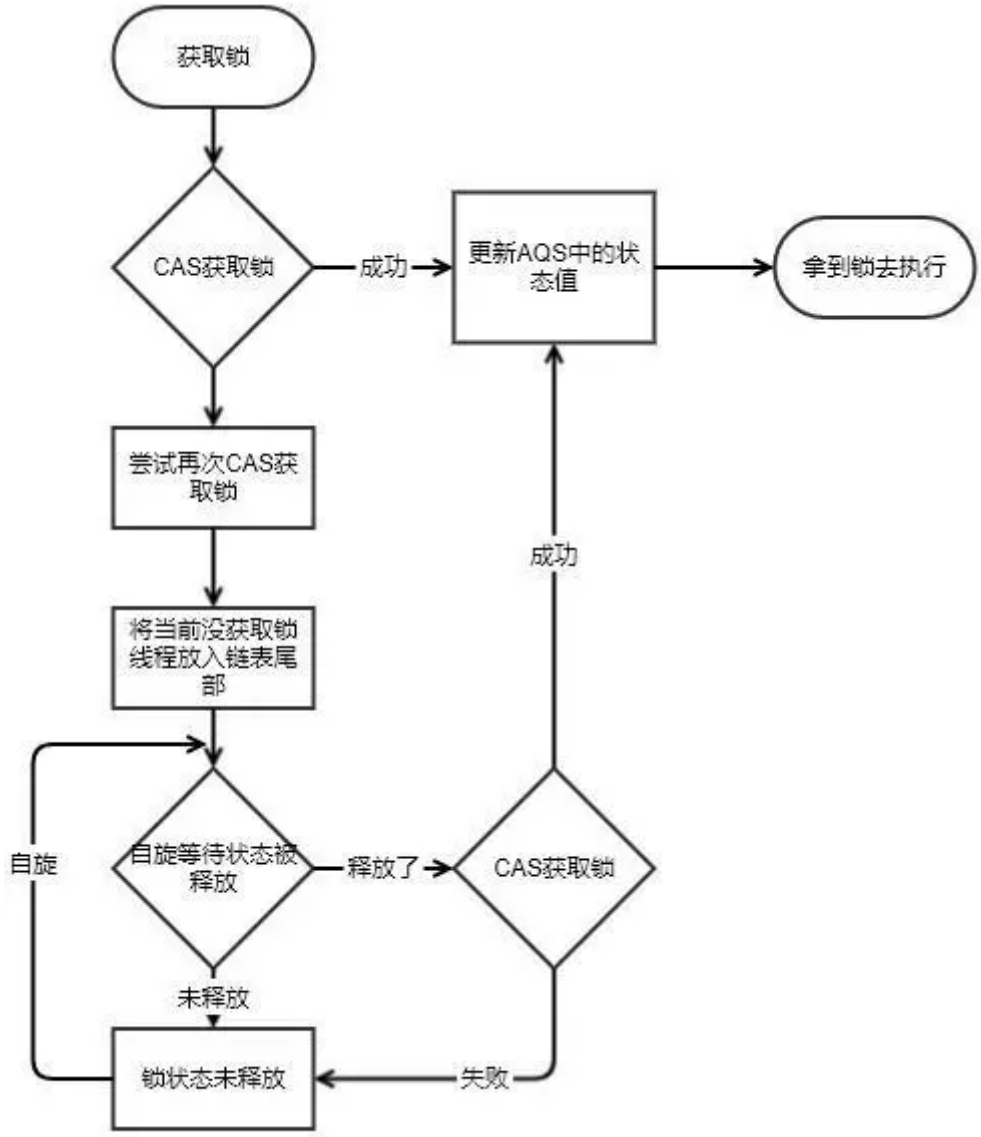
```
private Node addWaiter(Node mode) {
    Node node = new Node(mode);

    for (;;) {
        Node oldTail = tail;
        if (oldTail != null) {
            node.setPrevRelaxed(oldTail);
            //cas操作，在线程安全的前提下，将当前线程加入到链表尾部
            if (compareAndSetTail(oldTail, node)) {
                oldTail.next = node;
                return node;
            }
        } else {
            initializeSyncQueue();
        }
    }
}
```

```
final boolean acquireQueued(final Node node, int arg) {
    boolean interrupted = false;
    try {
        for (;;) {
            final Node p = node.predecessor();
            //当前线程到达头部，尝试CAS更新锁状态，更新成功则该等待线程可以从头部移除
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node))
                interrupted |= parkAndCheckInterrupt();
        }
    } catch (Throwable t) {
        cancelAcquire(node);
        if (interrupted)
            selfInterrupt();
        throw t;
    }
```

```
}
```

流程图



lock.unlock() 释放锁

```
/**
 * Attempts to release this lock.
 *
 * <p>If the current thread is the holder of this lock then the hold
 * count is decremented.  If the hold count is now zero then the lock
 * is released.  If the current thread is not the holder of this
 * lock then {@link IllegalMonitorStateException} is thrown.
 *
 * @throws IllegalMonitorStateException if the current thread does not
 *         hold this lock
 */
public void unlock() {
    sync.release(1);
}
```

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

```
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
```

最后通过调用`NonfairSync.tryRelease()`实现

释放锁就是对AQS中状态值`state`进行修改，同时更新下一个链表中的线程等待节点

lock锁带有大量的CAS+自旋，在低锁冲突下使用，如果锁冲突很多，自旋操作很多，会明显降低效率，还不如直接加`synchronized`重量级锁

两种方式对比

1.前者是非公平锁，后者可以实现公平锁

2.sync是通过操作Mark Word来对字节码进行添加修改指令实现的，是原生语法，需要JVM的支持。lock是通过双向链表结合状态量+cas操作来实现的实现，是API层面的

3.sync锁方便简单，由编译器来保障加锁和释放；lock要手动声明加锁和释放，要记得在try/catch结构中的finally中释放锁

4.sync锁不够灵活，按块锁住粒度很粗；lock灵活度高，由编码人员控制，锁的粒度很细

5.lock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。