L08-线程安全-AQS详解

JAVA并发编程

学习目标

- 掌握AQS实现原理
- 掌握synchronized原理

AQS详解

手写一个自己的ReentrantLock

- 用前面所学的LockSupport实现
  - 1、实现lock/unlock
  - 2、实现公平、非公平模式；
  - 3、实现可重入
  - 4、实现内存同步语义

volatile还有什么用途

■ volatile可用于限制局部代码指令重排序

1. 线程A和线程B的部分代码：

```
线程A：
content = initContent(); //(1)
isInit = true; //(2)
```

```
线程B：
if (isInit) { //(3)
        content.operation(); //(4)
}
```

2. jvm优化指令重排序后，代码的执行顺序可能如下：

```
线程A：
isInit = true; //(2)
content = initContent(); //(1)
```

3.当两个线程并发执行时，就可能出现线程B中抛空指针异常。

4.当我们在变量上加volatile修饰时，则用到该变量的块中就不会进行指令重排优化。

JMM的HB法则

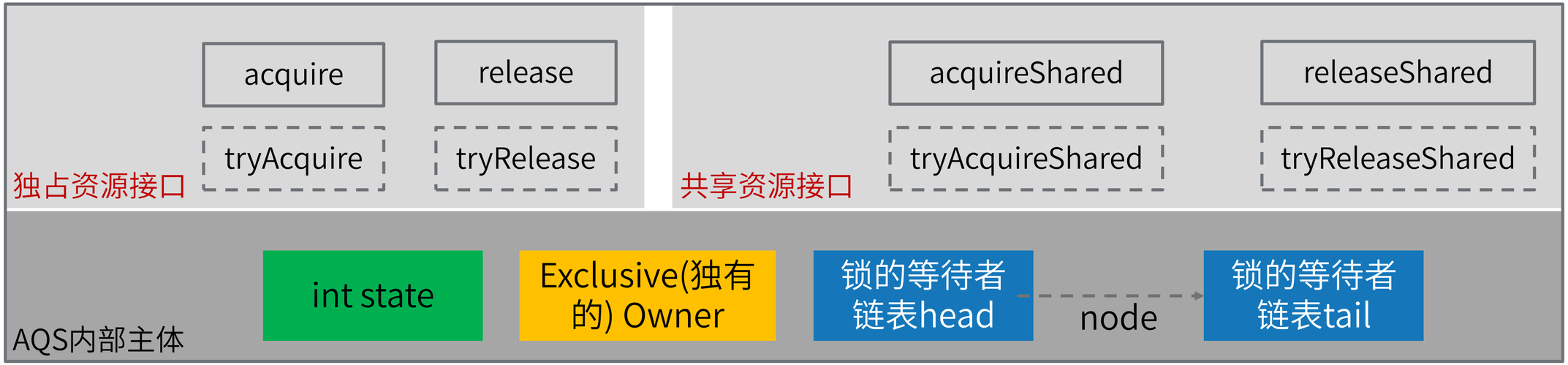
happens-before 关系用于描述两个有冲突的动作之间的顺序，如果一个action happends before 另一个action，则第一个操作被第二个操作可见，JVM需要实现如下happens-before规则：

- 程序顺序规则：每个线程中的每个操作都 happens-before 该线程中任意的后续操作。
- 监视器锁规则：一个锁的解除，happens-before与随后对这个锁的加锁。
- **volatile变量规则：对volatile域的写，happens-before于任意后续对这个volatile域的读**
- 线程启动规则：在某个线程对象上调用 start()方法 happens-before 被启动线程中的任意动作
- 线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，如在线程t1中成功执行了t2.join()，则t2中的所有操作对t1可见。
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。
- 对象终结规则：一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize方法的开始。
- 传递性：如果A happens-before于B，且B happens-before于C，那么A happens-before于C

AQS抽象队列同步器

提供了对资源占用、释放，线程的等待、唤醒等等接口和具体实现。

可以用在各种需要控制资源争用的场景中。(ReentrantLock/CountDownLatch/Semphore)



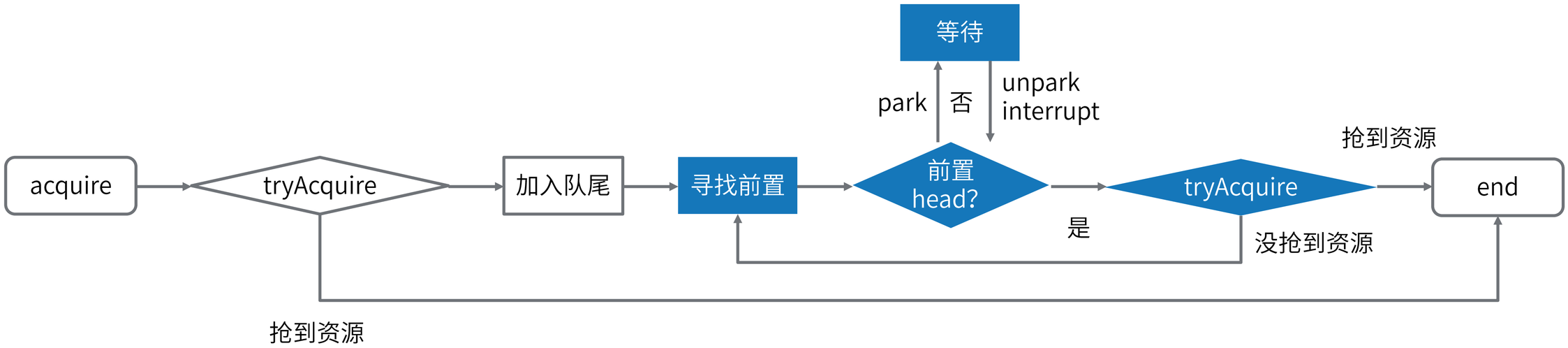
acquire、 acquireShared： 定义了资源争用的逻辑，如果没拿到，则等待。

tryAcquire、 tryAcquireShared： 实际执行占用资源的操作，如何判定获取到由使用者具体去实现。

release、 releaseShared： 定义释放资源的逻辑，释放之后，通知后续节点进行争抢。

tryRelease、 tryReleaseShared： 实际执行资源释放的操作，具体的AQS使用者去实现。

资源占用流程


acquire
tryAcquire
加入队尾
寻找前置
前置
head?
等待
park
否
unpark
interrupt
是
tryAcquire
抢到资源
没抢到资源
end
抢到资源

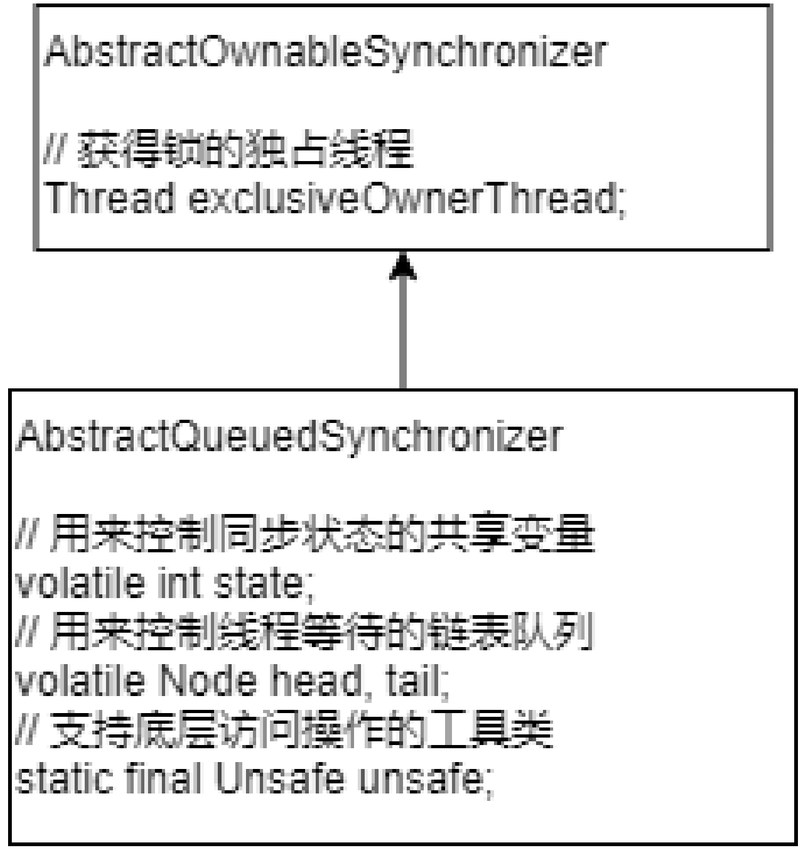
AQS详解

AQS实现锁语义的逻辑

写state值，volatile特性会刷新主内存。

读state值，volatile特性会使得线程从主内存读取，其他线程的修改变得可见。

锁的获取和释放都需要使得状态的变化在线程间同步



AQS提供了3个方法来访问修改同步状态

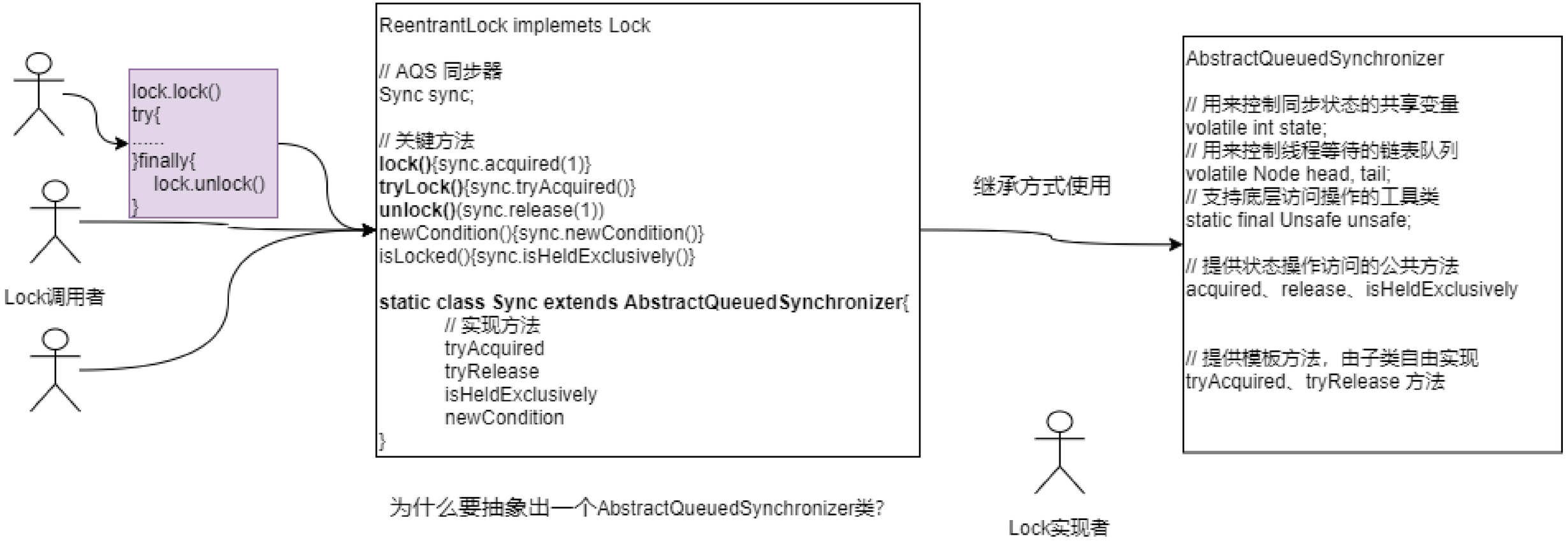
getState()，获取当前同步状态

setState(newState)，设置当前同步状态

compareAndSetState(expect,update)，使用CAS设置当前状态，保证状态设置的原子性

AQS详解

设计思想


lock.lock()
try{
.....
}finally{
lock.unlock()
}
Lock调用者
ReentrantLock implemets Lock
// AQS 同步器
Sync sync;
// 关键方法
lock(){sync.acquired(1)}
tryLock(){sync.tryAcquired()}
unlock(){sync.release(1))
newCondition(){sync.newCondition()}
isLocked(){sync.isHeldExclusively()}
static class Sync extends AbstractQueuedSynchronizer{
// 实现方法
tryAcquired
tryRelease
isHeldExclusively
newCondition
}
为什么要抽象出一个AbstractQueuedSynchronizer类?
继承方式使用
AbstractQueuedSynchronizer
// 用来控制同步状态的共享变量
volatile int state;
// 用来控制线程等待的链表队列
volatile Node head, tail;
// 支持底层访问操作的工具类
static final Unsafe unsafe;
// 提供状态操作访问的公共方法
acquired、release、isHeldExclusively
// 提供模板方法，由子类自由实现
tryAcquired、tryRelease 方法
Lock实现者

Synchronized原理详解

思考

```
synchronized(this){
    i++;
}
```

synchronized提供了锁住this对象的语义，加锁的状态是如何记录的?

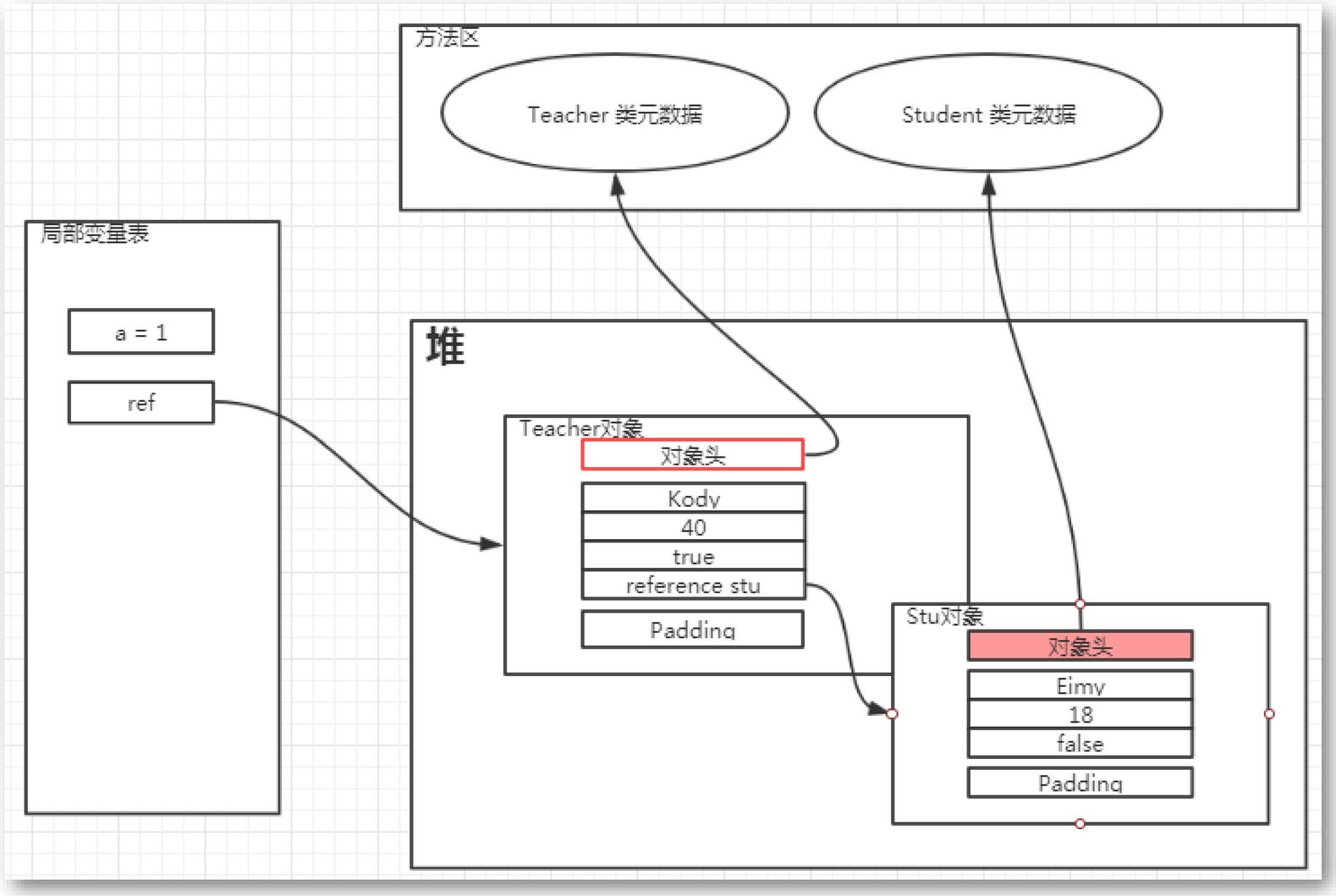
状态会被记录到this对象中吗?

若锁占用，线程挂起，

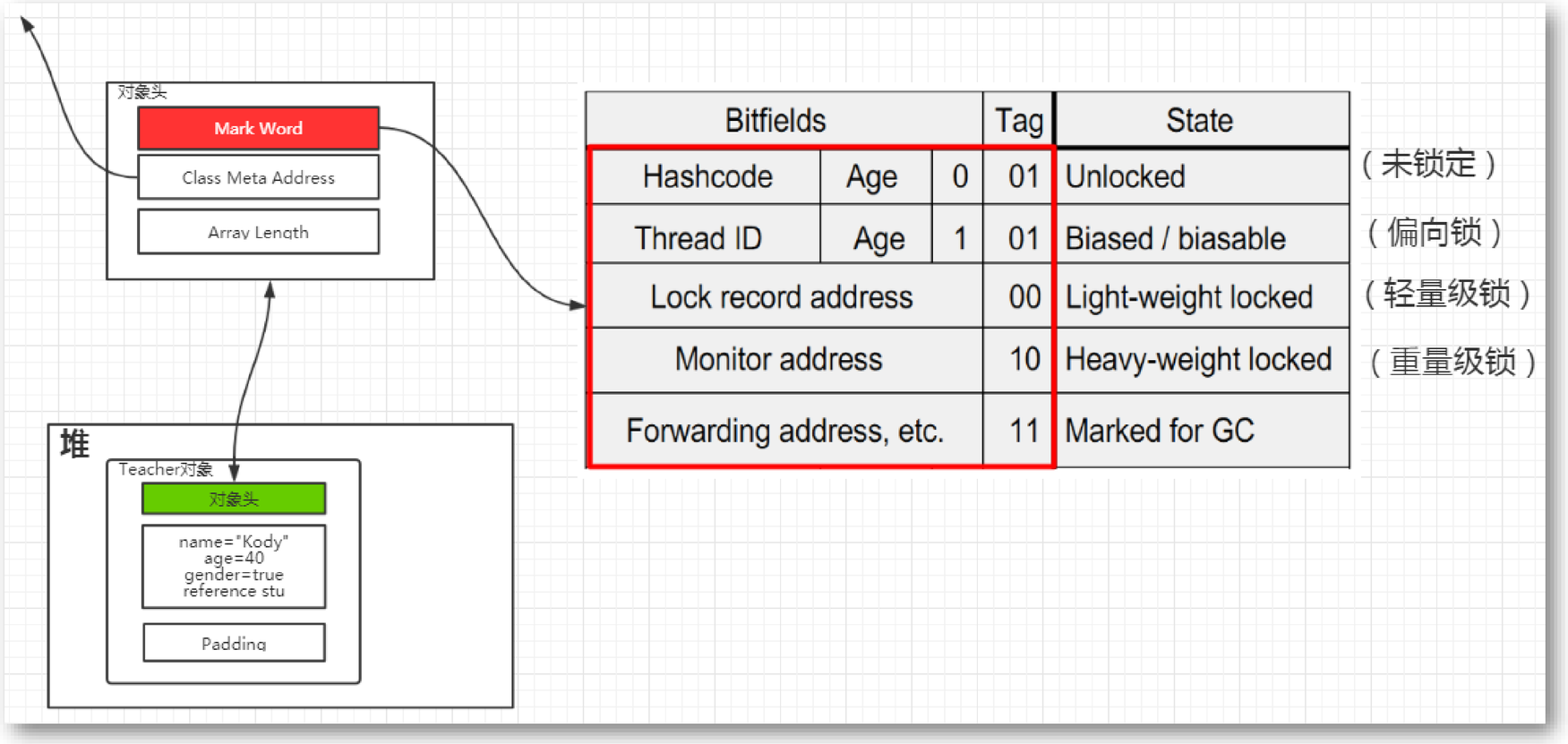
释放锁时，唤醒挂起的线程，是如何做到的?

堆内存中的Java对象

```java
public class Demo5_main {
    public static void main(String args[]){
        int a = 1;

        Teacher kody = new Teacher();
        kody.stu = new Student();
    }
}

class Teacher{
    String name = "Kody";
    int age = 40;
    boolean gender = true;

    Student stu;
}

class Student{
    String name = "Emily";
    int age = 18;
    boolean gender = false;
}
```



Mark Word



Bitfields			Tag	State	
Hashcode	Age	0	01	Unlocked	（未锁定）
Thread ID	Age	1	01	Biased / biasable	（偏向锁）
Lock record address			00	Light-weight locked	（轻量级锁）
Monitor address			10	Heavy-weight locked	（重量级锁）
Forwarding address, etc.			11	Marked for GC	

默认情况下JVM锁会经历：未锁定->偏向锁 -> 轻量级锁 -> 重量级锁　这四个状态

参考来源：

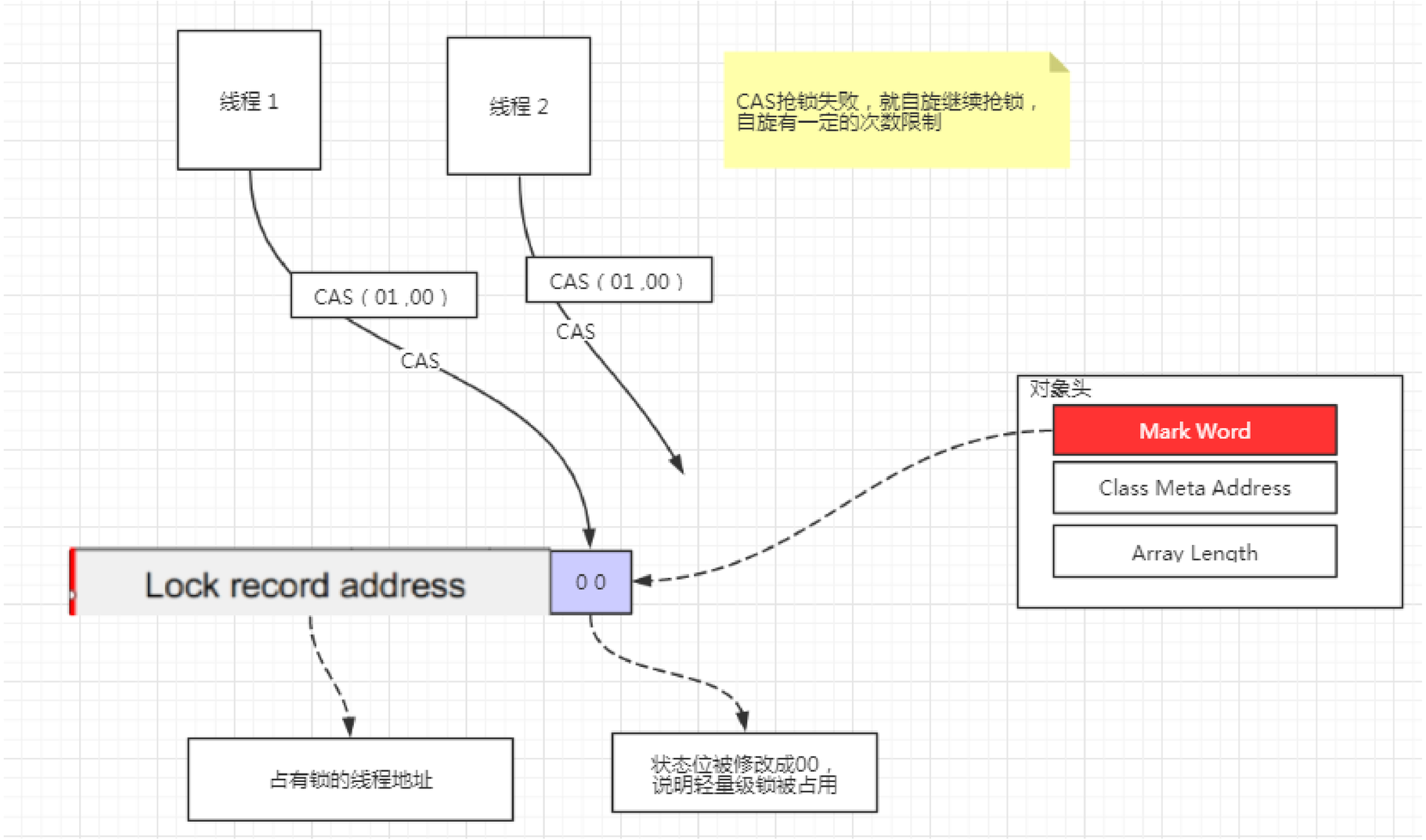
https://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf

https://wiki.openjdk.java.net/display/HotSpot/Synchronization

《深入理解Java虚拟机》

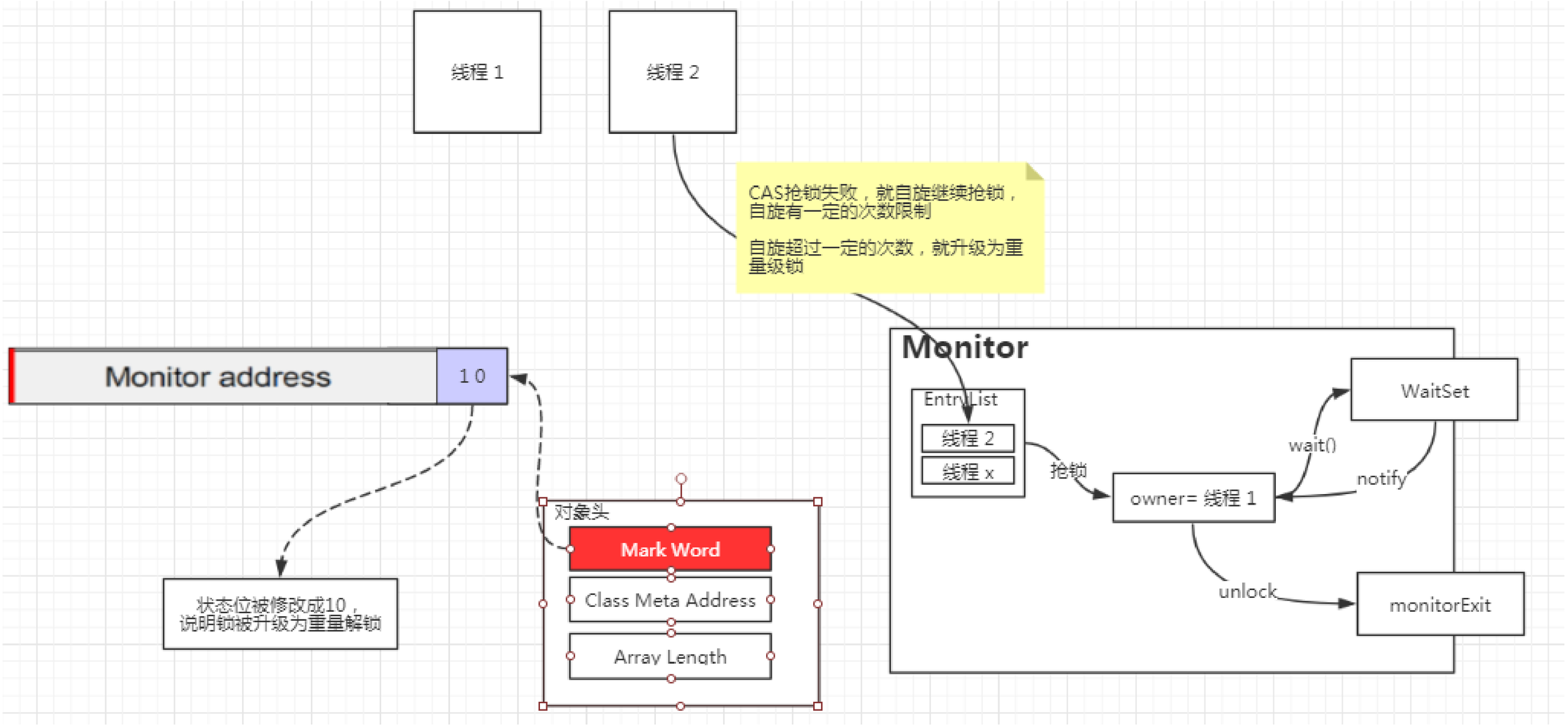
轻量级锁

在未锁定的状态下，可以通过CAS来抢锁，抢到的是轻量级锁



重量级锁

轻量级锁中的自旋有一定的次数限制，超过了次数限制，轻量级锁升级为重量级锁。



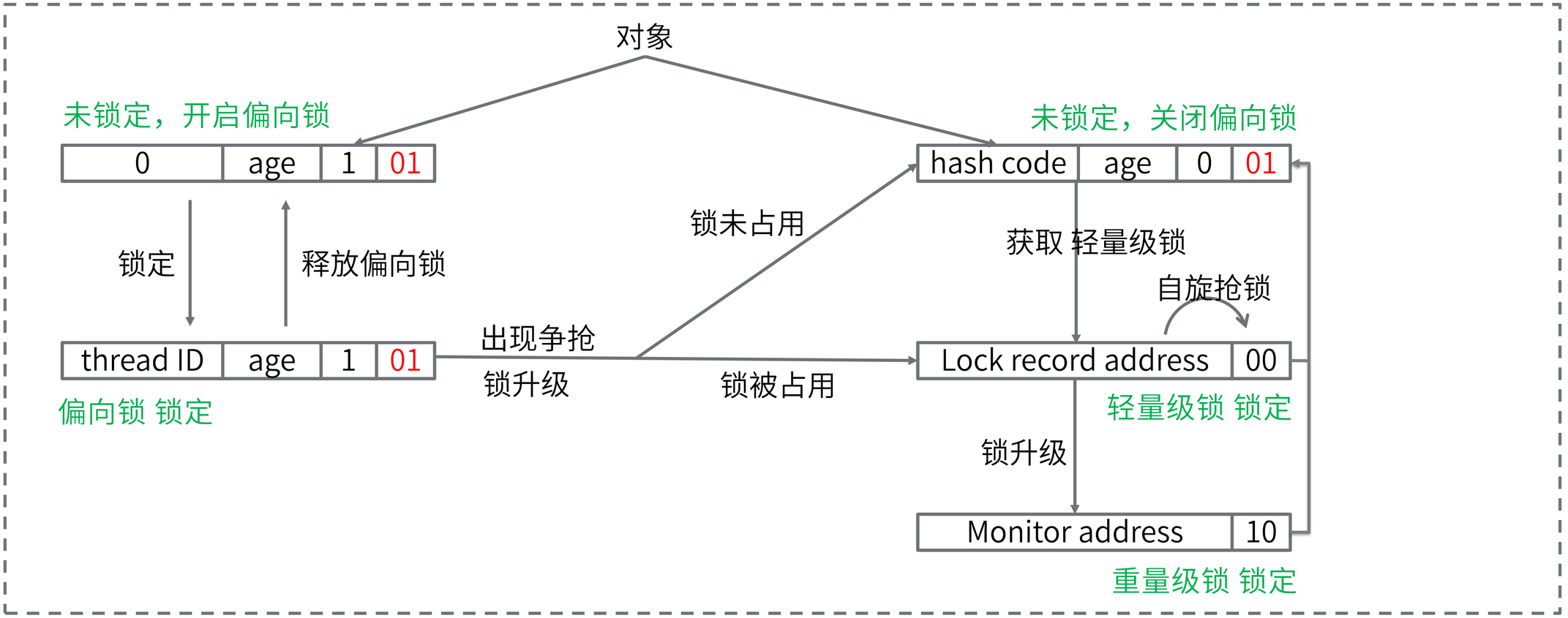
偏向锁

在JDK6 以后,默认已经开启了偏向锁这个优化，通过JVM 参数 -XX:-UseBiasedLocking 来禁用偏向锁

若偏向锁开启，只有一个线程抢锁，可获取到偏向锁

Bitfields			Tag	State	
Hashcode	Age	0	01	Unlocked	（未锁定，关闭偏向锁）
	Age	1	01	Biased / biasable	（未锁定，开启偏向锁）
Thread ID	Age	1	01	Biased / biasable	（锁定 偏向锁）

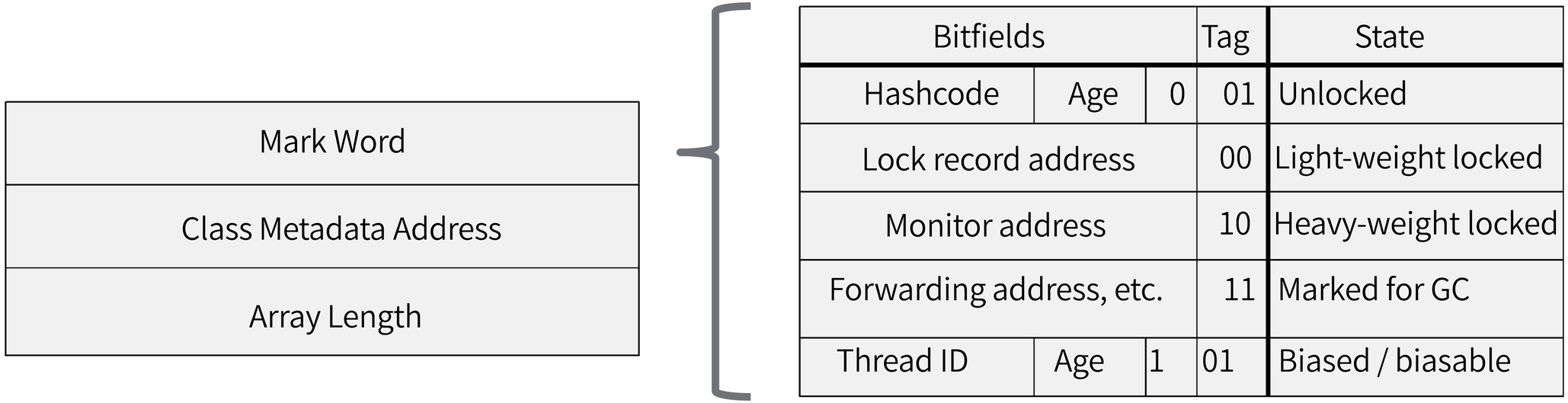
锁的升级过程



偏向标记第一次有用，出现过争用后就没用了。 -XX：-UseBiasedLocking 禁用使用偏置锁定，

偏向锁，本质就是无锁，如果没有发生过任何多线程争抢锁的情况，JVM认为就是单线程，无需做同步（jvm为了少干活：同步在JVM底层是有很多操作来实现的，如果是没有争用，就不需要去做同步操作）

同步关键字加锁原理



Bitfields			Tag	State
Hashcode	Age	0	01	Unlocked
Lock record address			00	Light-weight locked
Monitor address			10	Heavy-weight locked
Forwarding address, etc.			11	Marked for GC
Thread ID	Age	1	01	Biased / biasable

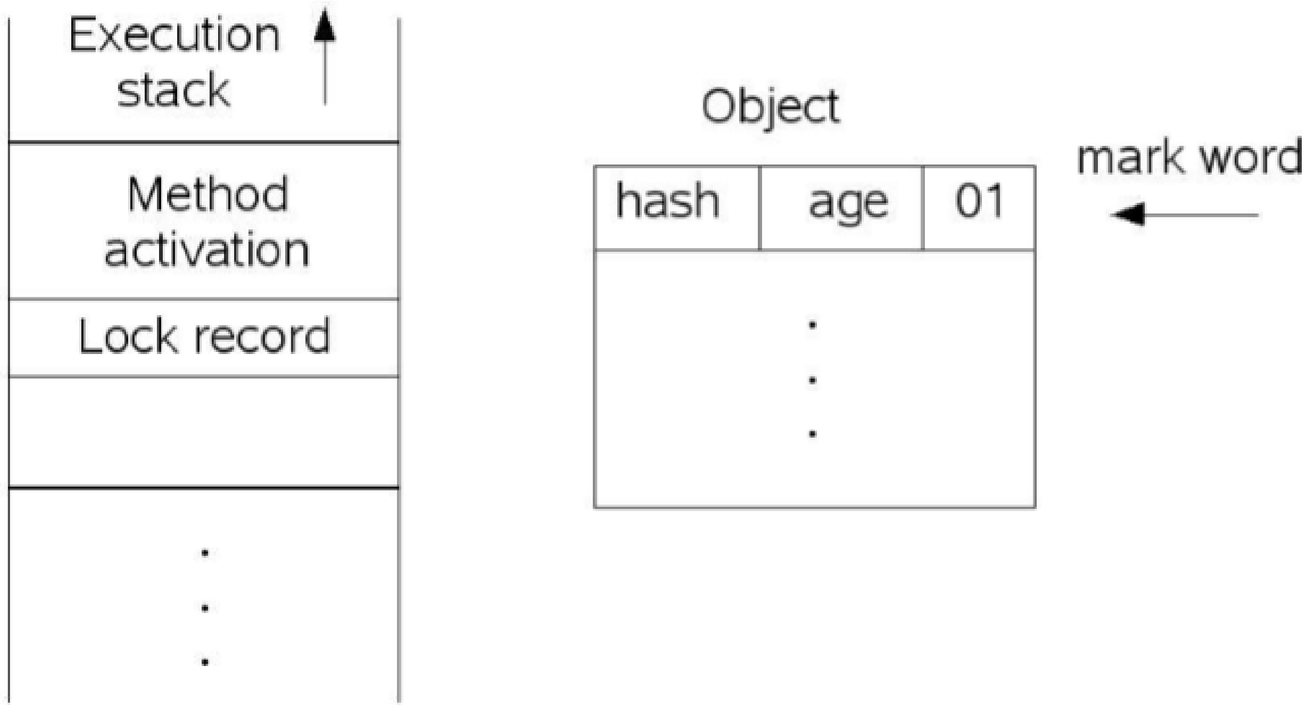
HotSpot中，对象前面会有一个类指针和标题，储标识哈希码的标题字以及用于分代垃圾收集的年龄和标记位

默认情况下JVM锁会经历：偏向锁 -> 轻量级锁 -> 重量级锁这四个状态

参考来源：https://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf

https://wiki.openjdk.java.net/display/HotSpot/Synchronization

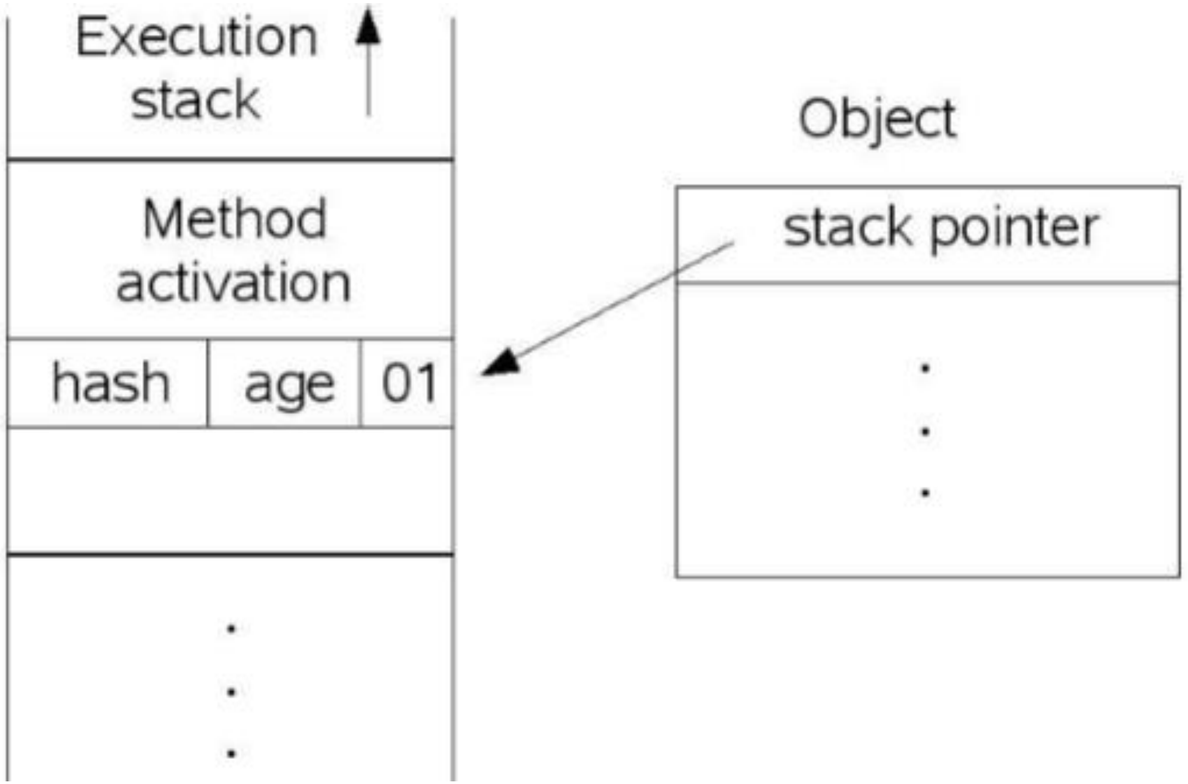
同步关键字加锁原理 – 轻量级锁



线程栈开辟一个空间，存储当前锁定对象的Mark Word信息。

Lock record可以存储多个锁定的对象信息。

同步关键字加锁原理– 轻量级锁



使用CAS修改mark word完毕，加锁成功。则mark word中的tag进入 00 状态。

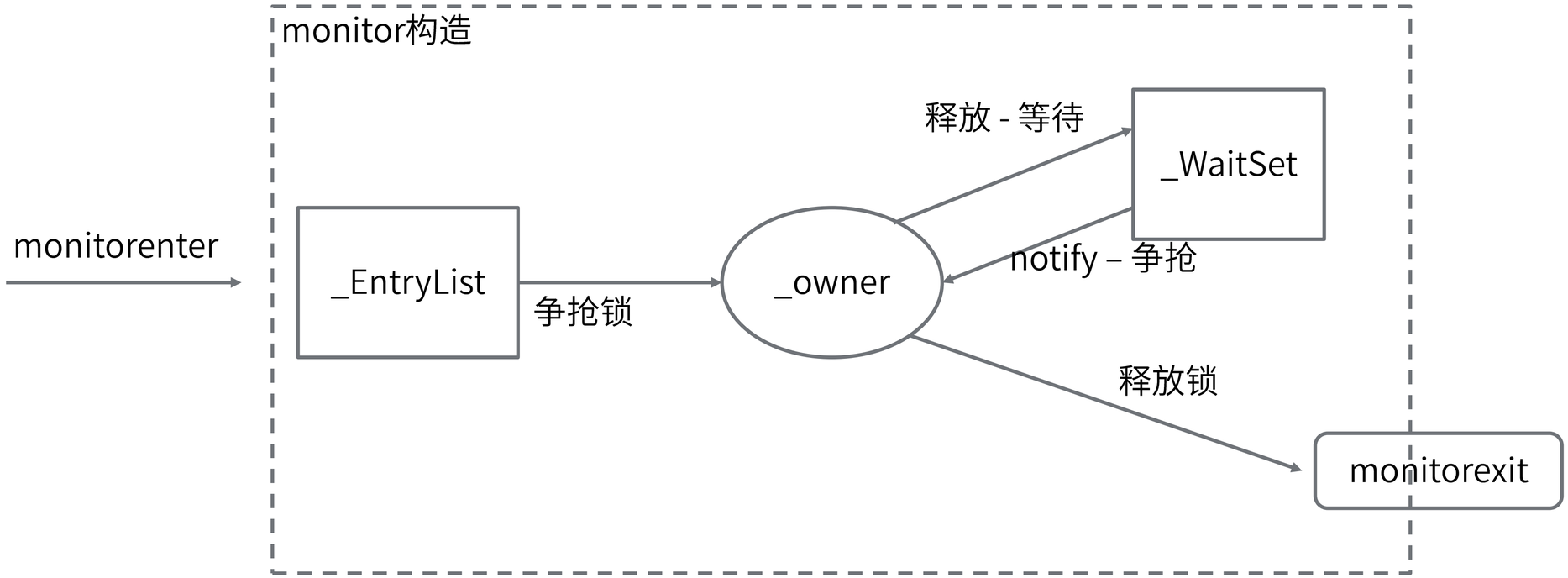
解锁的过程，则是一个逆向恢复mark word的过程

重量级锁 – 监视器(monitor)

修改mark word如果失败，会自旋CAS一定次数，该次数可以通过参数配置：

超过次数，仍未抢到锁，则锁升级为重量级锁，进入阻塞。

monitor也叫做管程，计算机操作系统原理中有提及类似概念。一个对象会有一个对应的monitor。



Synchronized的剖析

- JVM实现

  Monitorenter、Monitorexit

- 处理器实现，汇编指令

  lock cmpxchg %r15, 0x16(%r10) 和 lock cmpxchg %r10, (%r11)

理解synchronized的可见性和原子性

前面了解到，cmpxchg是CAS的汇编指令，上面汇编指令的含义是

- 先用lock指令对总线和缓存上锁，
- 然后用cmpxchg CAS操作设置对象头中的synchronized标志位
- CAS完成后释放锁，把缓存刷新到主内存。

synchronized的底层操作含义是

- 先对对象头的锁标志位用lock cmpxchg的方式设置成“锁住“状态
- 释放锁时，再用lock cmpxchg的方式修改对象头的锁标志位为”释放“状态，写操作都立刻写回主内存。
- JVM会进一步对synchronized时CAS失败的那些线程进行阻塞操作，这部分的逻辑没有体现在lock cmpxchg指令上。

lock cmpxchg指令前者保证了可见性和防止重排序，后者保证了操作的原子性。

动手实践，下节课见