synchronized

1.实现原理

1.1.介绍

> Synchronization in the Java Virtual Machine is implemented by monitor entry and exit, either explicitly (by use of the monitorenter and monitorexit instructions) or implicitly (by the method invocation and return instructions). For code written in the Java programming language, perhaps the most common form of synchronization is the synchronized method. A synchronized method is not normally implemented using monitorenter and monitorexit. Rather, it is simply distinguished in the run-time constant pool by the ACC_SYNCHRONIZED flag, which is checked by the method invocation instructions (§2.11.10).

- 本段摘自 The Java® Virtual Machine Specification 3.14. Synchronization
- 在JVM中，同步的实现是通过监视器锁的进入和退出实现的，要么显示得通过monitorenter 和 monitorexit指令实现，要么隐示地通过方法调用和返回指令实现
- 对于Java代码来说，或许最常用的同步实现就是同步方法。其中同步代码块是通过使用 monitorenter 和 monitorexit 实现的，而同步方法却是使用 ACC_SYNCHRONIZED 标记符隐示的实现，原理是通过方法调用指令检查该方法在常量池中是否包含 ACC_SYNCHRONIZED 标记符

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

1. 普通同步方法，锁是当前实例对象
2. 静态同步方法，锁是当前类的 class 对象
3. 同步方法块，锁是括号里面的对象

当一个线程访问同步代码块时，它首先是需要得到锁才能执行同步代码，当退出或者抛出异常时必须要释放锁，那么它是如何来实现这个机制的呢？我们先看一段简单的代码：

1.2.查看编译结果

```
package concurrent;
public class SynchronizedDemo {
    public static synchronized void staticMethod() throws InterruptedException {
        System.out.println("静态同步方法开始");
        Thread.sleep(1000);
        System.out.println("静态同步方法结束");
    }
    public synchronized void method() throws InterruptedException {
        System.out.println("实例同步方法开始");
        Thread.sleep(1000);
        System.out.println("实例同步方法结束");
    }
```

```
    public synchronized void method2() throws InterruptedException {
        System.out.println("实例同步方法2开始");
        Thread.sleep(3000);
        System.out.println("实例同步方法2结束");
    }
    public static void main(String[] args) {
        final SynchronizedDemo synDemo = new SynchronizedDemo();
        Thread thread1 = new Thread(() -> {
            try {
               synDemo.method();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread thread2 = new Thread(() -> {
            try {
                synDemo.method2();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        thread1.start();
        thread2.start();
    }
}
```

利用 Javap 工具查看生成的 class 文件信息来分析 synchronized 的实现

- 常量池

```
  Compiled from "SynDemo.java"
public class concurrent.SynDemo
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #8.#22         // java/lang/Object."<init>":()V
   #2 = Fieldref           #23.#24        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #25            // 同步实例方法
   #4 = Methodref          #26.#27        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = String             #28            // 同步静态方法
   #6 = String             #29            // 同步代码块
   #7 = Class              #30            // concurrent/SynDemo
   #8 = Class              #31            // java/lang/Object
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               method
  #14 = Utf8               staticMethod
  #15 = Utf8               chunkMethod
  #16 = Utf8               StackMapTable
  #17 = Class              #30            // concurrent/SynDemo
  #18 = Class              #31            // java/lang/Object
  #19 = Class              #32            // java/lang/Throwable
  #20 = Utf8               SourceFile
  #21 = Utf8               SynDemo.java
  #22 = NameAndType        #9:#10         // "<init>":()V
  #23 = Class              #33            // java/lang/System
  #24 = NameAndType        #34:#35        // out:Ljava/io/PrintStream;
  #25 = Utf8               同步实例方法
  #26 = Class              #36            // java/io/PrintStream
  #27 = NameAndType        #37:#38        // println:(Ljava/lang/String;)V
  #28 = Utf8               同步静态方法
  #29 = Utf8               同步代码块
  #30 = Utf8               concurrent/SynDemo
  #31 = Utf8               java/lang/Object
  #32 = Utf8               java/lang/Throwable
  #33 = Utf8               java/lang/System
  #34 = Utf8               out
  #35 = Utf8               Ljava/io/PrintStream;
  #36 = Utf8               java/io/PrintStream
  #37 = Utf8               println
  #38 = Utf8               (Ljava/lang/String;)V
```

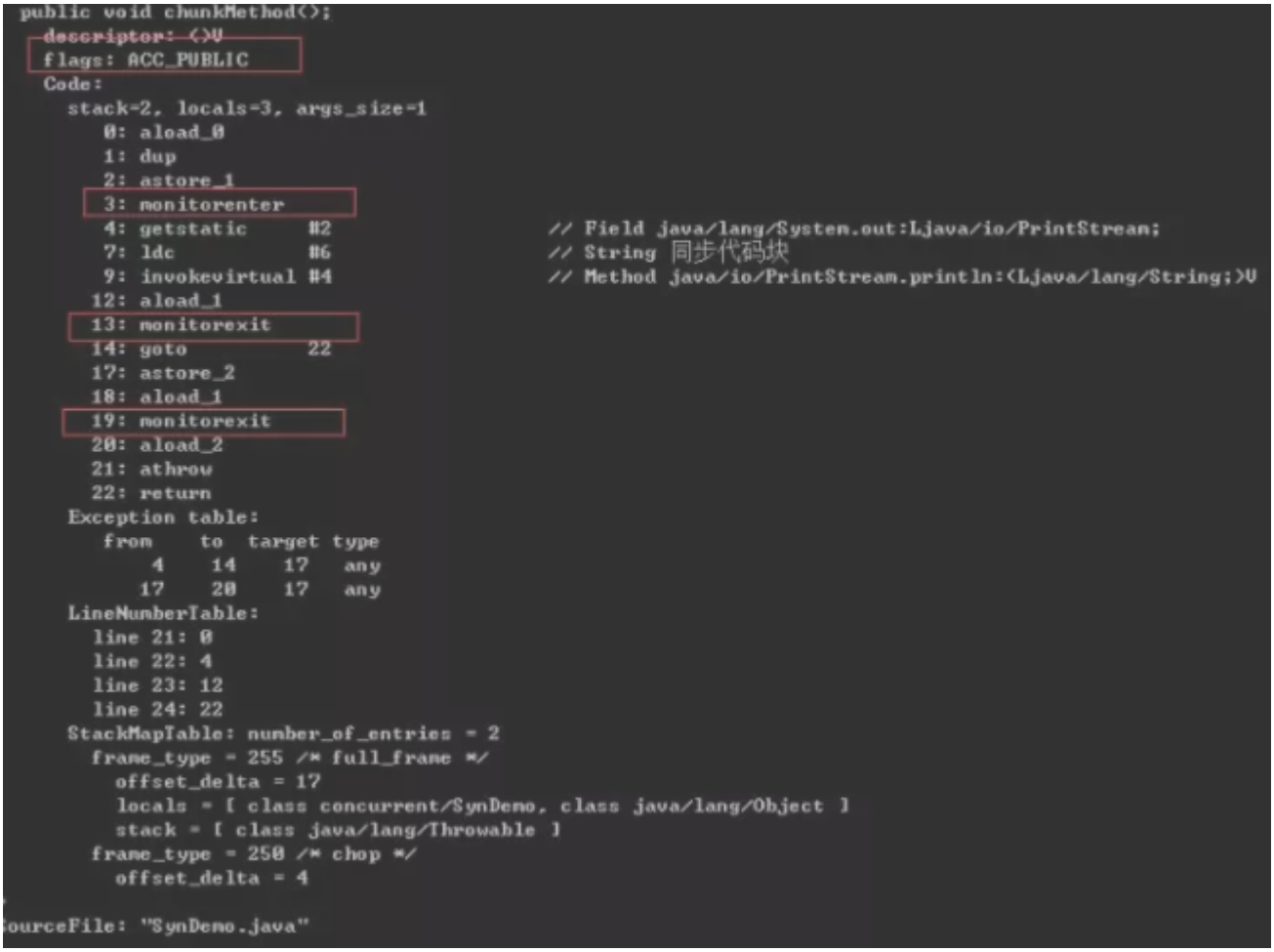
- 同步方法:同步方法会包含一个ACC_SYNCHCRONIZED标记符

```
public concurrent.SynDemo();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 6: 0

public synchronized void method();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String 同步实例方法
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 10: 0
      line 12: 8

public static synchronized void staticMethod();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=0, args_size=0
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #5                  // String 同步静态方法
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 16: 0
      line 17: 8
```

- 同步代码块:同步代码块会在代码中插入 monitorenter 和 monitorexist 指令



1.3.同步代码块原理

1.3.1.monitor监视器

- 每个对象都有一个监视器，在同步代码块中，JVM通过monitorenter和monitorexist指令实现同步锁的获取和释放功能
- 当一个线程获取同步锁时，即是通过获取monitor监视器进而等价为获取到锁
- monitor的实现类似于操作系统中的管程

1.3.2.monitor指令

> Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref, as follows: • If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor. • If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry count. • If another thread already owns the monitor associated with objectref, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

每个对象都有一个监视器。当该监视器被占用时即是锁定状态(或者说获取监视器即是获得同步锁)。线程执行monitorenter指令时会尝试获取监视器的所有权，过程如下：

- 若该监视器的进入次数为0，则该线程进入监视器并将进入次数设置为1，此时该线程即为该监视器的所有者
- 若线程已经占有该监视器并重入，则进入次数+1

- 若其他线程已经占有该监视器，则线程会被阻塞直到监视器的进入次数为0，之后线程间会竞争获取该监视器的所有权
- 只有首先获得锁的线程才能允许继续获取多个锁

1.3.3.monitorexit指令

> The thread that executes monitorexit must be the owner of the monitor associated with the instance referenced by objectref. The thread decrements the entry count of the monitor associated with objectref. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

执行monitorexit指令将遵循以下步骤：

- 执行monitorexit指令的线程必须是对象实例所对应的监视器的所有者
- 指令执行时，线程会先将进入次数-1，若-1之后进入次数变成0，则线程退出监视器(即释放锁)
- 其他阻塞在该监视器的线程可以重新竞争该监视器的所有权

1.3.4.实现原理

- 在同步代码块中，JVM通过monitorenter和monitorexist指令实现同步锁的获取和释放功能
- monitorenter指令是在编译后插入到同步代码块的开始位置
- monitorexit指令是插入到方法结束处和异常处
- JVM要保证每个monitorenter必须有对应的monitorexit与之配对
- 任何对象都有一个monitor与之关联，当且一个monitor被持有后，它将处于锁定状态
- 线程执行monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁
- 线程执行monitorexit指令时，将会将进入次数-1直到变成0时释放监视器
- 同一时刻只有一个线程能够成功，其它失败的线程会被阻塞，并放入到同步队列中，进入BLOCKED状态

1.3.5.补充

由于 wait/notify 等方法底层实现是基于监视器，因此只有在同步方法(块)中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException 的异常的原因

1.4.同步方法同步原理

区别于同步代码块的监视器实现，同步方法通过使用 ACC_SYNCHRONIZED 标记符隐示的实现

原理是通过方法调用指令检查该方法在常量池中是否包含 ACC_SYNCHRONIZED 标记符，如果有，JVM 要求线程在调用之前请求锁

2.对象头/Monitor

2.1.对象头

在JVM中，对象在内存中的布局分成三块区域：对象头、示例数据和对齐填充

- 对象头： 对象头主要存储对象的hashCode、锁信息、类型指针、数组长度(若是数组的话)等信息
- 示例数据：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组长度，这部分内存按4字节对齐
- 填充数据：由于JVM要求对象起始地址必须是8字节的整数倍，当不满足8字节时会自动填充（因此填充数据并不是必须的，仅仅是为了字节对齐）

synchronized 用的锁是存在Java对象头里的。那么什么是 Java 对象头呢？Hotspot 虚拟机的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。其中：

Mark Word 用于存储对象自身的运行时数据，如哈希码（HashCode）、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等。Java 对象头一般占有两个机器码（在 32 位虚拟机中，1 个机器码等于 4 字节，也就是 32 bits）。但是如果对象是数组类型，则需要三个机器码，因为 JVM 虚拟机可以通过 Java 对象的元数据信息确定 Java 对象的大小，无法从数组的元数据来确认数组的大小，所以用一块来记录数组长度。

下图是 Java 对象头的存储结构（32位虚拟机）：

25Bit	4bit	1bit	2bit
对象的hashCode	对象的分代年龄	是否是偏向锁	锁标志位

对象头信息是与对象自身定义的数据无关的额外存储成本，但是考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据，它会根据对象的状态复用自己的存储空间，也就是说，Mark Word 会随着程序的运行发生变化，变化状态如下：

32 位虚拟机：

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象hashcode、对象分代年龄				01
轻量级锁	指向锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

64 位虚拟机：

锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit) Epoch(2bit)				1	01

2.2.Monitor

2.2.1.Monitor模式

- Monitor其实是一种同步工具，也可以说是一种同步机制，它通常被描述为一个对象，主要特点是互斥和信号机制
- 互斥： 一个Monitor锁在同一时刻只能被一个线程占用，其他线程无法占用
- 信号机制(signal)： 占用Monitor锁失败的线程会暂时放弃竞争并等待某个谓词成真（条件变量），但该条件成立后，当前线程会通过释放锁通知正在等待这个条件变量的其他线程，让其可以重新竞争锁
- Mesa派的signal机制
    - Mesa派的signal机制又称"Non-Blocking condition variable"
    - 占有Monitor锁的线程发出释放通知时，不会立即失去锁，而是让其他线程等待在队列中，重新竞争锁
    - 这种机制里，等待者拿到锁后不能确定在这个时间差里是否有别的等待者进入过Monitor，因此不能保证谓词一定为真，所以对条件的判断必须使用while

- Java中采用就是Mesa派的singal机制，即所谓的notify

在 Monitor Object 模式中，主要有四种类型的参与者：

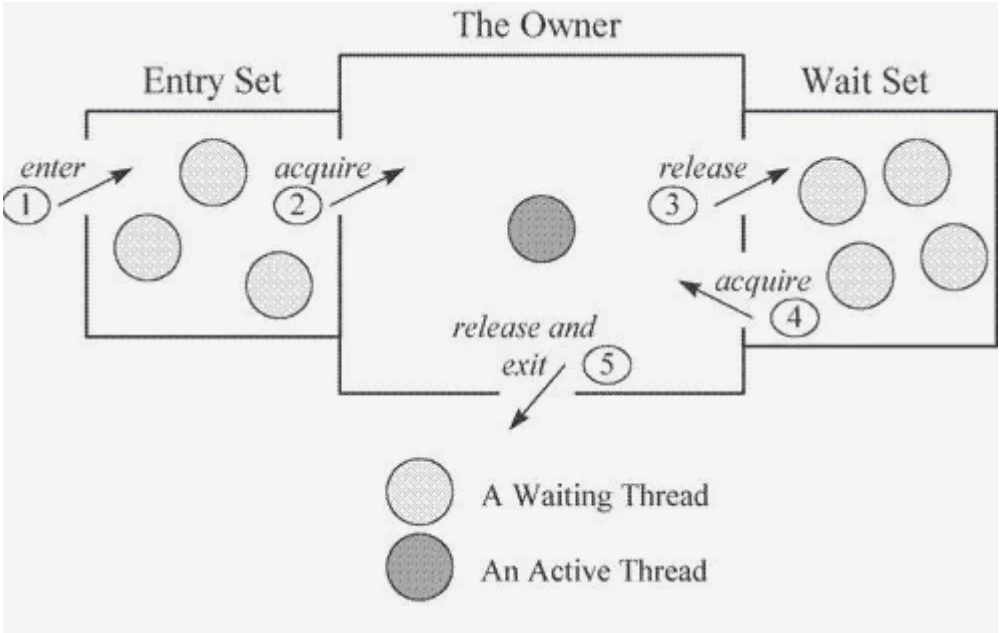
参与者	描述
监视者对象 (Monitor Object)	负责定义公共的接口方法，这些公共的接口方法会在多线程的环境下被调用执行
同步方法 (Syn Medthod)	这些方法是监视者对象所定义。为了防止竞争调价，无论是否同时有多个线程并发调用同步方法，还是监视者对象含有多个同步方法，在任意时间内只有监视者对象的一个同步方法能够被执行
监视锁 (Monitor Lock)	每个监视者对象都会拥有一把监视锁
监视条件 (Monitor Condition)	同步方法使用监视锁和监视条件来决定方法是否需要阻塞或重新执行

1. 同步方法的调用和串行化：
2. 当客户线程调用监视者对象的同步方法时，必须首先获取它的监视锁
3. 只要该监视者对象有其他同步方法正在被执行，获取操作便不会成功
4. 当监视者对象已被线程占用时(即同步方法正被执行)，客户线程将被阻塞直到它获取监视锁
5. 当客户线程成功获取监视锁后，进入临界区，执行方法实现的服务
6. 一旦同步方法完成执行，监视锁会被自动释放，目的是使其他客户线程有机会调用执行该监视者对象的同步方法
7. 同步方法线程挂起：如果调用同步方法的客户线程必须被阻塞或是有其他原因不能立刻进行，它能够在一个监视条件(Monitor Condition)上等待，这将导致该客户线程暂时释放监视锁，并被挂起在监视条件上
8. 监视条件通知：一个客户线程能够通知一个监视条件，目的是通知阻塞在该监视条件(该监视锁)的线程恢复运行
9. 同步方法线程恢复：
    1. 一旦一个早先被挂起在监视条件上的同步方法线程获取通知，它将继续在最初的等待监视条件的点上执行
    2. 在被通知线程被允许恢复执行同步方法之前，监视锁将自动被获取(线程间自动相互竞争锁)

2.2.2.MonitorRecord总数

- MonitorRecord(统一简称MR)是Java线程私有的数据结构，每一个线程都有一个可用MR列表，同时还有一个全局的可用列表
- 一个被锁住的对象都会和一个MR关联（对象头的MarkWord中的LockWord指向MR的起始地址）
- MR中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用

MonitorRecord	描述
Owner	1.当该值为NULL是表示没有任何线程拥有该monitor，初始值为NULL 2.当线程成功拥有该锁(monitor record)后保存线程唯一标识，当锁被释放时又设置为NULL
EntryQ	关联一个系统互斥锁（semaphore），阻塞所有竞争该锁(monitor record)失败的线程
RcThis	阻塞或等待在该锁(monitor record)的线程个数 -- 被阻塞/等待的线程被存入同步/等待队列中
Nest	记录重入次数
HashCode	保存从对象头拷贝过来的HashCode值（可能还包含GC age）
Candidate	1.用来避免不必要的阻塞或等待线程唤醒 2.只有两种可能的值，0表示没有需要唤醒的线程，1表示要唤醒一个继任线程来竞争锁 原因：因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降



- 线程如果获得监视锁成功，将成为该监视锁对象的拥有者
- 在任一时刻，监视器对象只属于一个活动线程(Owner)
- 拥有者可以调用wait方法自动释放监视锁，进入等待状态

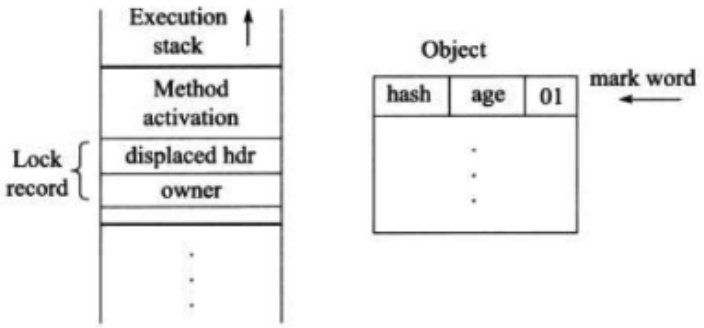


图 13-3 轻量级锁 CAS 操作之前堆栈与对象的状态[⊖]

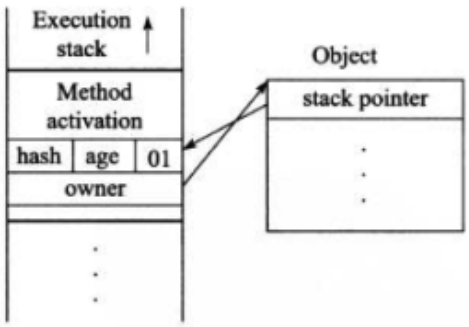


图 13-4 轻量级锁 CAS 操作之后堆栈与对象的状态

3.锁优化

3.1.自旋锁

- 痛点：由于线程的阻塞/唤醒需要CPU在用户态和内核态间切换，频繁的转换对CPU负担很重，进而对并发性能带来很大的影响
- 现象：通过大量分析发现，对象锁的锁状态通常只会持续很短一段时间，没必要频繁地阻塞和唤醒线程
- 原理：通过执行一段无意义的空循环让线程等待一段时间，不会被立即挂起，看持有锁的线程是否很快释放锁，如果锁很快被释放，那当前线程就有机会不用阻塞就能拿到锁了，从而减少切换，提高性能
- 隐患：若锁能很快被释放，那么自旋效率就很好(真正执行的自旋次数越少效率越好，等待时间就少)；但若是锁被一直占用，那自旋其实没有做任何有意义的事但又白白占用和浪费了CPU资源，反而造成资源浪费
- 注意：自旋次数必须有个限度(或者说自旋时间)，如果超过自旋次数(时间)还没获得锁，就要被阻塞挂起
- 使用： JDK1.6以上默认开启-XX:+UseSpinning，自旋次数可通过-XX:PreBlockSpin调整，默认10次

由来

线程的阻塞和唤醒，需要 CPU 从用户态转为核心态。频繁的阻塞和唤醒对 CPU 来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时，我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间。为了这一段很短的时间，频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁。

定义

所谓自旋锁，就是让该线程**等待**一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。

怎么等待呢？**执行一段无意义的循环即可**（自旋）。

自旋等待不能替代阻塞，先不说对处理器数量的要求（多核，貌似现在没有单核的处理器了），**虽然它可以避免线程切换带来的开销，但是它占用了处理器的时间。**如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好，反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，典型的占着茅坑不拉屎，这样反而会带来性能上的浪费。

所以说，自旋等待的时间（自旋的次数）必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

自旋锁在 JDK 1.4.2 中引入，默认关闭，但是可以使用 -XX:+UseSpinning 开开启。 在 JDK1.6 中默认开启。同时自旋的默认次数为 10 次，可以通过参数 -XX:PreBlockSpin 来调整。

如果通过参数 -XX:PreBlockSpin 来调整自旋锁的自旋次数，会带来诸多不便。假如我将参数调整为 10 ，但是系统很多线程都是等你刚刚退出的时候，就释放了锁（假如你多自旋一两次就可以获取锁），你是不是很尴尬。于是 JDK 1.6 引入自适应的自旋锁，让虚拟机会变得越来越聪明。

3.2.自适应自旋锁

- 痛点：由于自旋锁只能指定固定的自旋次数，但由于任务的差异，导致每次的最佳自旋次数有差异
- 原理：通过引入"智能学习"的概念，由前一次在同一个锁上的自旋时间和锁的持有者的状态来决定自旋的次数，换句话说就是自旋的次数不是固定的，而是可以通过分析上次得出下次，更加智能
- 实现：若当前线程针对某锁自旋成功，那下次自旋此时可能增加(因为JVM认为这次成功是下次成功的基础)，增加的话成功几率可能更大；反正，若自旋很少成功，那么自旋次数会减少(减少空转浪费)甚至直接省略自旋过程，直接阻塞(因为自旋完全没有意义，还不如直接阻塞)
- 补充：有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，JVM对锁的状况预测会越来越准确，JVM会变得越来越智能

JDK 1.6 引入了更加聪明的自旋锁，即自适应自旋锁。

所谓自适应就意味着自旋的次数不再是固定的，**它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。**它怎么做呢？

- 线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。
- 反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。

有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明。

3.3.阻塞锁

3.3.1.阻塞锁

- 加锁成功：当出现锁竞争时，只有获得锁的线程能够继续执行
- 加锁失败：竞争失败的线程会由running状态进入blocking状态，并被放置到与目标锁相关的一个等待队列中
- 解锁：当持有锁的线程退出临界区，释放锁后，会将等待队列中的一个阻塞线程唤醒，令其重新参与到锁竞争中

3.3.2.公平锁

公平锁就是获得锁的顺序按照先到先得的原则，从实现上说，要求当一个线程竞争某个对象锁时，只要这个锁的等待队列非空，就必须把这个线程阻塞并塞入队尾（插入队尾一般通过一个CAS操作保持插入过程中没有锁释放）

3.3.3.非公平锁

相对的，非公平锁场景下，每个线程都先要竞争锁，在竞争失败或当前已被加锁的前提下才会被塞入等待队列，在这种实现下，后到的线程有可能无需进入等待队列直接竞争到锁（随机性）

3.4.锁消除

- 痛点：根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加
- 锁原理： JVM在编译时通过对运行上下文的描述，去除不可能存在共享资源竞争的锁，通过这种方式消除无用锁，即删除不必要的加锁操作，从而节省开销
- 使用： 逃逸分析和锁消除分别可以使用参数-XX:+DoEscapeAnalysis和-XX:+EliminateLocks(锁消除必须在-server模式下)开启
- 补充：在JDK内置的API中，例如StringBuffer、Vector、HashTable都会存在隐性加锁操作，可消除

由来

为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制。但是，在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除。如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。

定义

锁消除的依据是**逃逸分析的数据支持**。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于我们程序员来说这还不清楚么？我们会在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？我们虽然没有显示使用锁，但是我们在使用一些 JDK 的内置 API 时，如 StringBuffer、Vector、HashTable 等，这个时候会**存在隐性的加锁操作**。比如 StringBuffer 的 #append(..)方法，Vector 的 add(...) 方法：

```
public void vectorTest(){
    Vector<String> vector = new Vector<String>();
    for (int i = 0 ; i < 10 ; i++){
        vector.add(i + "");
    }
    System.out.println(vector);
}
```

在运行这段代码时，JVM 可以明显检测到变量 vector 没有逃逸出方法 #vectorTest() 之外，所以 JVM 可以大胆地将 vector 内部的加锁操作消除。

3.3.锁粗化

- 痛点：多次连接在一起的加锁、解锁操作会造成原理：将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的
- 锁使用：将多个彼此靠近的同步块合同在一个同步块 或 把多个同步方法合并为一个方法
- 补充：在JDK内置的API中，例如StringBuffer、Vector、HashTable都会存在隐性加锁操作，可合并

由来

我们知道在使用同步锁的时候，需要让同步块的作用范围尽可能小：仅在共享数据的实际作用域中才进行同步。这样做的目的，是为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

在大多数的情况下，上述观点是正确的，LZ 也一直坚持着这个观点。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。

定义

锁粗话概念比较好理解，就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。

如上面实例：vector 每次 add 的时候都需要加锁操作，JVM 检测到对同一个对象（vector）连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到 for 循环之外。

3.4.锁升级

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。它们会随着竞争的激烈而逐渐升级。注意，锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

3.4.1.重量级锁

重量级锁通过对象内部的监视器（Monitor）实现。

其中，Monitor 的**本质**是，依赖于底层操作系统的 Mutex Lock http://dreamrunner.org/blog/2014/06/29/qian-tan-mutex-lock/实现。操作系统实现线程之间的切换，需要从用户态到内核态的切换，切换成本非常高。

MutexLock最核心的理念就是 尝试获取锁.若可得到就占有.若不能,就进入睡眠等待

3.4.2.轻量级锁

引入轻量级锁的主要目的，是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

- 痛点：由于线程的阻塞/唤醒需要CPU在用户态和内核态间切换，频繁的转换对CPU负担很重，进而对并发性能带来很大的影响主要

- 目的： 在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗升级
- 时机： 当关闭偏向锁功能或多线程竞争偏向锁会导致偏向锁升级为轻量级锁
- 原理： 在只有一个线程执行同步块时进一步提高性能
- 数据结构： 包括指向栈中锁记录的指针、锁标志位
- 隐患：对于轻量级锁有个使用前提是"没有多线程竞争环境"，一旦越过这个前提，除了互斥开销外，还会增加额外的CAS操作的开销，在多线程竞争环境下，轻量级锁甚至比重量级锁还要慢

当关闭偏向锁功能或者多个线程竞争偏向锁，**导致偏向锁升级为轻量级锁**，则会尝试获取轻量级锁，其步骤如下：

获取锁

1. 判断当前对象是否处于无锁状态？若是，则 JVM 首先将在当前线程的栈帧中，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的 拷贝（官方把这份拷贝加了一个 Displaced 前缀，即 Displaced Mark Word）；否则，执行步骤（3）；
2. JVM 利用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指正。如果成功，表示竞争到锁，则将锁标志位变成 00（表示此对象处于轻量级锁状态），执行同步操作；如果失败，则执行步骤（3）；
3. **判断当前对象的 Mark Word 是否指向当前线程的栈帧？如果是，则表示当前线程已经持有当前对象的锁，则直接执行同步代码块；否则，只能说明该锁对象已经被其他线程抢占了，当前线程便尝试使用自旋来获取锁。若自旋后没有获得锁，此时轻量级锁会升级为重量级锁，锁标志位变成 10，当前线程会被阻塞。**

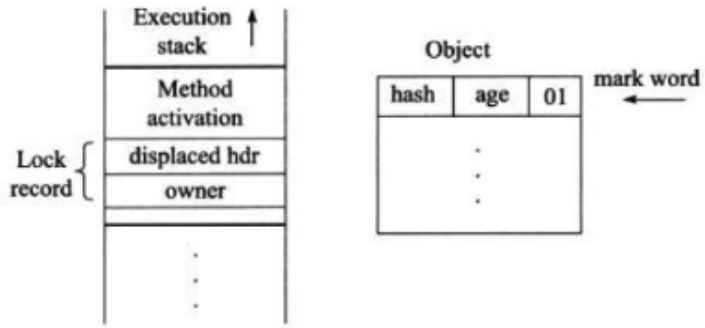


图 13-3 轻量级锁 CAS 操作之前堆栈与对象的状态[⊖]

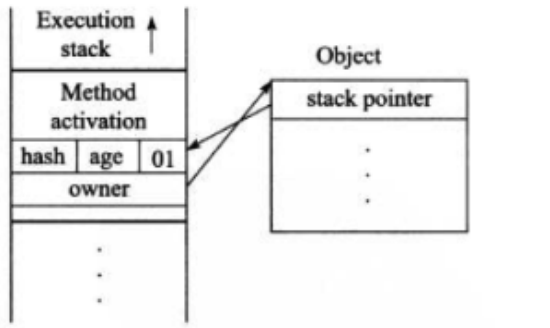
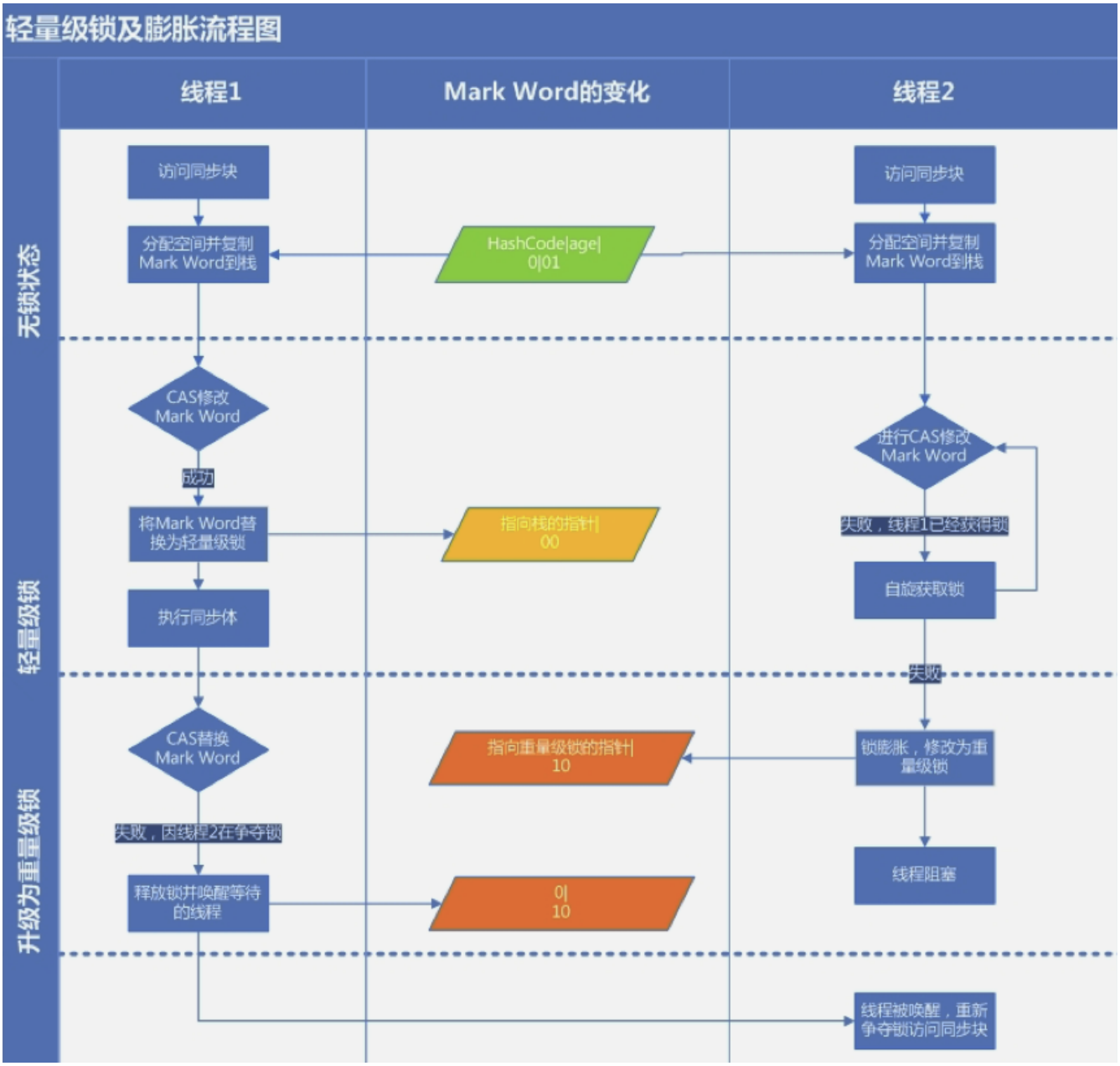


图 13-4 轻量级锁 CAS 操作之后堆栈与对象的状态

释放锁

轻量级锁的释放也是通过 CAS 操作来进行的，主要步骤如下：

1. 取出在获取轻量级锁保存在 Displaced Mark Word 中 数据。
2. 使用 CAS 操作将取出的数据替换当前对象的 Mark Word 中。如果成功，则说明释放锁成功；否则，执行（3）。



对于轻量级锁，其性能提升的依据是：“对于绝大部分的锁，在整个生命周期内都是不会存在竞争的”。如果打破这个依据则除了互斥的开销外，还有额外的 CAS 操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢。

3.4.3.偏向锁

- 痛点： Hotspot作者发现在大多数情况下不存在多线程竞争的情况，而是同一个线程多次获取到同一个锁，为了让线程获得锁代价更低，因此设计了偏向锁 (这个跟业务使用有很大关系)
- 主要目的： 为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径
- 原理： 在只有一个线程执行同步块时通过增加标记检查而减少CAS操作进一步提高性能
- 数据结构： 包括占有锁的线程id，是否是偏向锁，epoch(偏向锁的时间戳)，对象分代年龄、锁标志位
- 优势：偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令，其余时刻不需要CAS指令(相比其他锁)
- 隐患：由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗（这个通常只能通过大量压测才可知）
- 对比：轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

引入偏向锁主要目的是：为了在无多线程竞争的情况下，尽量减少不必要的轻量级锁执行路径。

在上文，可以看到偏向锁时，Mark Word 的数据结构为：线程 ID、Epoch(偏向锁的时间戳)、对象分带年龄、是否是偏向锁(1)、锁标识位(01)。

只需要检查是否为偏向锁、锁标识为以及 ThreadID 即可，处理流程如下：

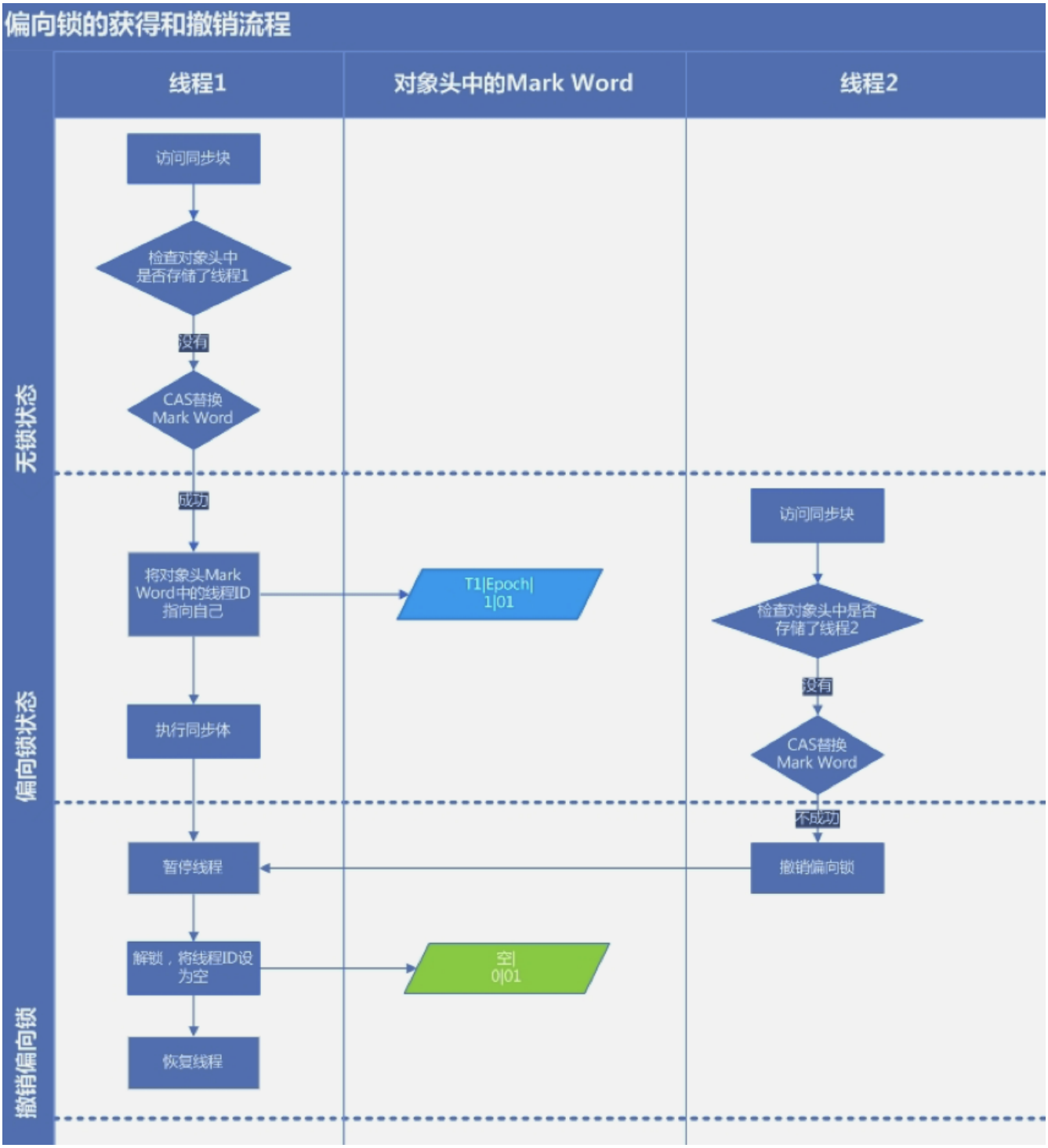
获取偏向锁

1. 检测 Mark Word是 否为可偏向状态，即是否为偏向锁的标识位为 1 ，锁标识位为 01 。
2. 若为可偏向状态，则测试线程 ID 是否为当前线程 ID ？ 如果是，则执行步骤（5）；否则，执行步骤（3）。
3. 如果线程 ID 不为当前线程 ID ，则通过 CAS 操作竞争锁。竞争成功，则将 Mark Word 的线程 ID 替换为当前线程 ID ，则执行步骤（5）；否则，执行线程（4）。
4. 通过 CAS 竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块。
5. 执行同步代码块

撤销偏向锁

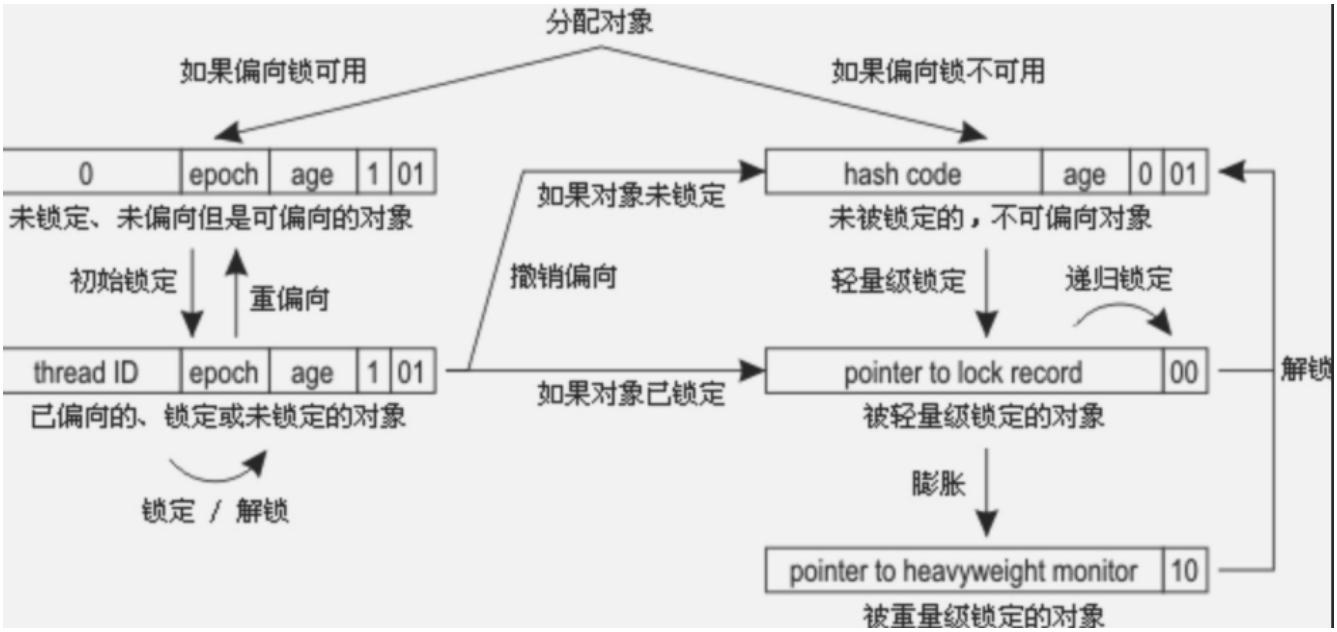
1. 偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。
2. 偏向锁的撤销需要等待全局安全点（这个时间点是上没有正在执行的代码）。其步骤如下：
    1. 暂停拥有偏向锁的线程，判断锁对象是否还处于被锁定状态。
    2. 撤销偏向锁，恢复到无锁状态（ 01 ）或者轻量级锁的状态。
3. 要么重新偏向于其他线程(即将偏向锁交给其他线程，相当于当前线程"被"释放了锁)
4. 要么恢复到无锁或者标记锁对象不适合作为偏向锁(此时锁会被升级为轻量级锁)

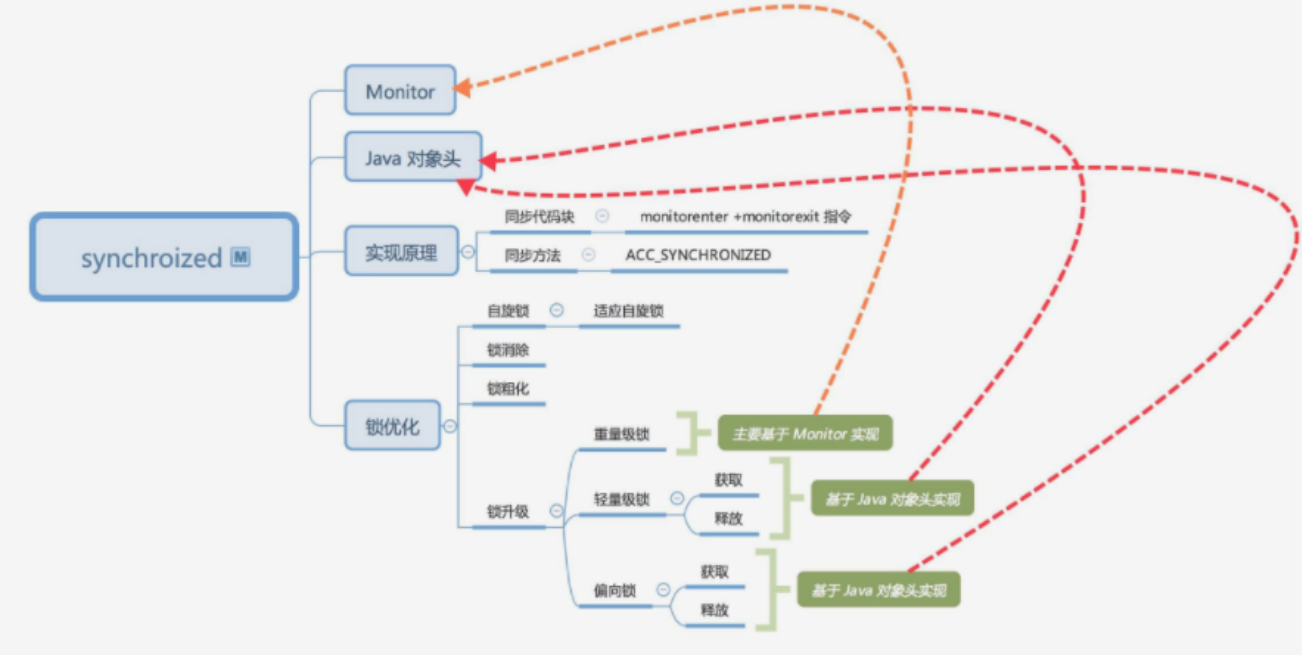
最后唤醒暂停的线程，被阻塞在安全点的线程继续往下执行同步代码块


偏向锁的获得和撤销流程
线程1
对象头中的Mark Word
线程2
无锁状态
偏向锁状态
撤销偏向锁
访问同步块
检查对象头中
是否存储了线程1
没有
CAS替换
Mark Word
成功
将对象头Mark
Word中的线程ID
指向自己
T1|Epoch|
1|01
执行同步体
访问同步块
检查对象头中是否
存储了线程2
没有
CAS替换
Mark Word
不成功
暂停线程
撤销偏向锁
解锁，将线程ID设
为空
空|
0|01
恢复线程

3.4.4.对比和转换

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗CPU。	追求响应时间。 同步块执行速度非常快。
重量级锁	线程竞争不使用自旋，不会消耗CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。 同步块执行速度较长。




synchroized M
Monitor
Java 对象头
实现原理
同步代码块
monitorenter +monitorexit 指令
同步方法
ACC_SYNCHRONIZED
锁优化
自旋锁
适应自旋锁
锁消除
锁粗化
锁升级
重量级锁
主要基于 Monitor 实现
轻量级锁
获取
释放
基于 Java 对象头实现
偏向锁
获取
释放
基于 Java 对象头实现