深入理解 synchronized 的实现原理

作者：jack1liu 时间: 2021-02-05 09:50:39

标签： #并发编程 java synchronized

【摘要】目录 1、实现原理 2、Java对象头 3、Monitor 4、锁优化 5、自旋锁 6、适应自旋锁 7、锁消除 8、锁粗化 9、轻量级锁 10、偏向锁 11、重量级锁 记得刚刚开始学习 Java 的时候，一遇到多线程情况就是 synchronized。对于当时的我们来说，synchronized 是如此的神奇且强大。我们赋予它一个名字“同步”，也成为我...

目录

1、实现原理

2、Java对象头

3、Monitor

4、锁优化

5、自旋锁

6、适应自旋锁

7、锁消除

8、锁粗化

9、轻量级锁

10、偏向锁

11、重量级锁

记得刚刚开始学习 Java 的时候，一遇到多线程情况就是 synchronized。对于当时的我们来说，synchronized 是如此的神奇且强大。我们赋予它一个名字“同步”，也成为我们解决多线程情况的良药，百试不爽。但是，随着学习的深入，我们知道synchronized 是一个重量级锁，相对于 Lock，它会显得那么笨重，以至于我们认为它不是那么的高效，并慢慢抛弃它。 诚然，随着Javs SE 1.6对synchronized进行各种优化后，synchronized不会显得那么重。

下面跟随 LZ 一起来探索 **synchronized 的实现机制、Java 是如何对它进行了优化、锁优化机制、锁的存储结构和升级过程。**

1、实现原理

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：

```
public class SynchronizedTest { public synchronized void test1(){ } public void test2() {
        synchronized(this){ } }
}
```

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

```
public class com.ufclub.ljs.weal.model.SynchronizedTest {
  public com.ufclub.ljs.weal.model.SynchronizedTest();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":
()V
       4: return

  public synchronized void test1();
    Code:
       0: return

  public void test2();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter          监视器进入，获取锁
       4: aload_1
       5: monitorexit           监视器退出，释放锁
       6: goto          14
       9: astore_2
      10: aload_1
      11: monitorexit
      12: aload_2
      13: athrow
      14: return
    Exception table:
       from    to  target type
           4     6     9   any
           9    12     9   any
}
```


从上面可以看出，同步代码块是使用 monitorenter 和 monitorexit 指令实现的，同步方法（在这看不出来需要看 JVM 底层实现）依靠的是方法修饰符上的 ACCSYNCHRONIZED 实现。

同步代码块

monitorenter 指令插入到同步代码块的开始位置，monitorexit 指令插入到同步代码块的结束位置，JVM 需要保证每一个monitorenter 都有一个 monitorexit 与之相对应。任何对象都有一个 monitor 与之相关联，当且一个 monitor 被持有之后，他将处于锁定状态。线程执行到 monitorenter 指令时，将会尝试获取对象所对应的 monitor 所有权，即尝试获取对象的锁；

同步方法

synchronized 方法则会被翻译成普通的方法调用和返回指令如: invokevirtual、areturn 指令，在 VM 字节码层面并没有任何特别的指令来实现被 synchronized 修饰的方法，而是在 Class 文件的方法表中将该方法的 accessflags 字段中的 synchronized 标志位置1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的 Class 在 JVM 的内部对象表示 Klass 做为锁对象。

(摘自：http://www.cnblogs.com/javaminer/p/3889023.html)

下面我们来继续分析，但是在深入之前我们需要了解两个重要的概念：**Java对象头、Monitor。**

Java 对象头、monitor：Java 对象头和 monitor 是实现 synchronized 的基础！下面就这两个概念来做详细介绍。

2、Java对象头

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 Klass Point 是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，**Mark Word 用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键。**

Mark Word

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32bit），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，但是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[illegible]	微信号锁标志位

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

简单介绍了Java对象头，我们下面再看Monitor。

3、Monitor

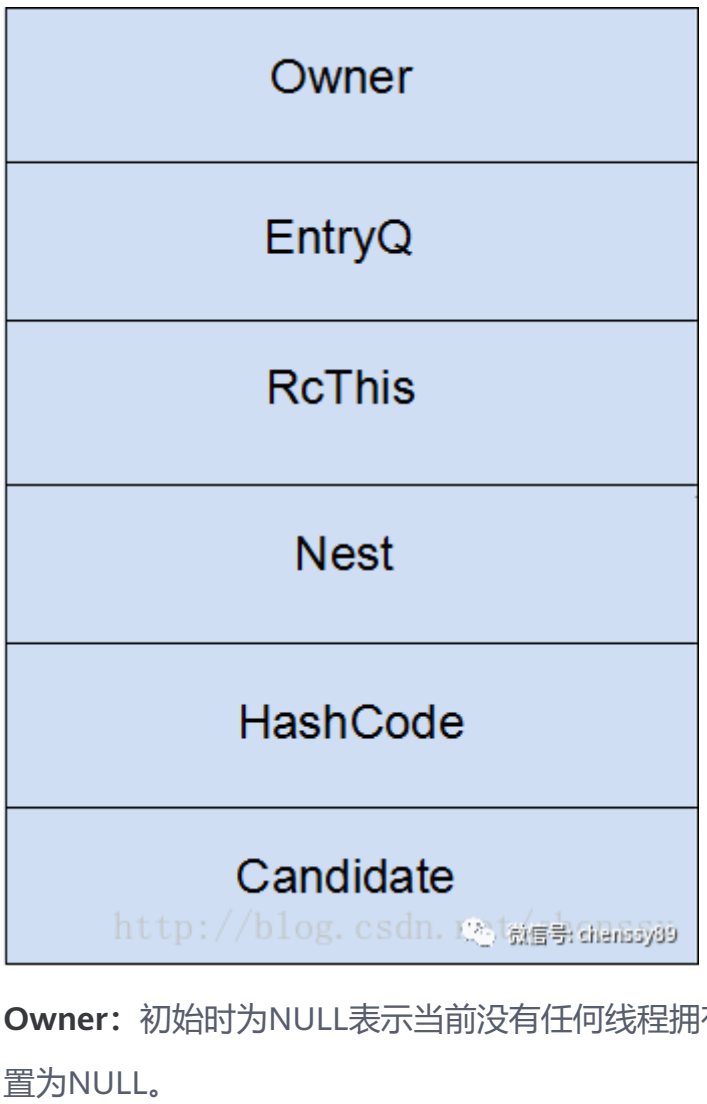
什么是 Monitor?

我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象。

与一切皆对象一样，所有的 Java 对象是天生的 Monitor，每一个 Java 对象都有成为 Monitor 的潜质，因为在 Java 的设计中 ，每一个 Java 对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者 Monitor 锁。

Monitor 是线程私有的数据结构，每一个线程都有一个可用 monitor record 列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个 monitor 关联（对象头的 MarkWord 中的 LockWord 指向 monitor 的起始地址），同时 monitor 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。

其结构如下：



Owner：初始时为NULL表示当前没有任何线程拥有该 monitor record，当线程成功拥有该锁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），阻塞所有试图锁住 monitor record失败的线程。

RcThis：表示 blocked 或 waiting在该 monitor record上的所有线程的个数。

Nest：用来实现重入锁的计数。HashCode:保存从对象头拷贝过来的HashCode值（可能还包含GC age）。

Candidate：用来避免不必要的阻塞或等待线程唤醒，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。

Candidate 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。

摘自：*Java*中*synchronized*的实现原理与应用

我们知道 synchronized 是重量级锁，效率不怎么滴，同时这个观念也一直存在我们脑海里，不过在 **JDK 1.6中对 synchronize的实现进行了各种优化，使得它显得不是那么重了，那么 JVM 采用了那些优化手段呢?**

4、锁优化

JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。他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。

5、自旋锁

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。 **何谓自旋锁？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 JDK1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明。

6、适应自旋锁

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。**

7、锁消除

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。锁消除的依据是逃逸分析的数据支持。

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。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，这个时候会存在隐形的加锁操作。

比如 StringBuffer 的 append() 方法，Vector 的 add() 方法:

```
public void vectorTest(){
    Vector vector = new Vector(); 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。

8、锁粗化

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。 **那什么是锁粗化？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。

9、轻量级锁

引入轻量级锁的主要目的是在都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。

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。

判断当前对象是否处于无锁状态（hashcode、0、01），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）；

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）；

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，这时轻量级锁需要膨胀为重量级锁，锁标志位变成10，后面等待的线程将会进入阻塞状态；

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：

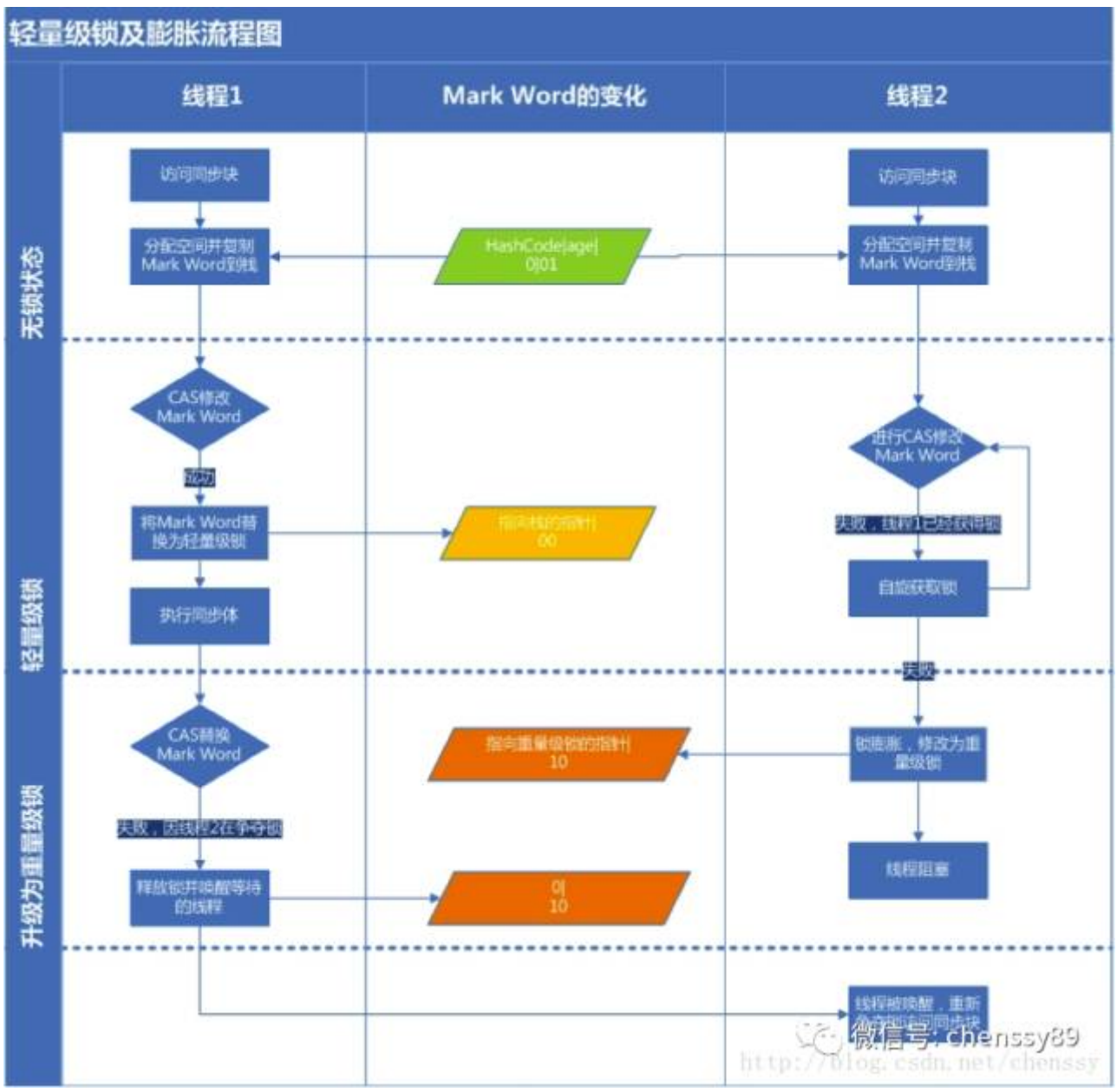
取出在获取轻量级锁保存在 Displaced Mark Word 中的数据；

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）；

如果CAS操作替换失败，说明有其他线程尝试获取该锁，则需要在释放锁的同时需要唤醒被挂起的线程。

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；

下图是轻量级锁的获取和释放过程：



10、偏向锁

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。上面提到了轻量级锁的加锁解锁操作是需要依赖多次CAS原子指令的。那么偏向锁是如何来减少不必要的CAS操作呢？我们可以查看 Mark work 的结构就明白了。

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：获取锁。

检测Mark Word是否为可偏向状态，即是否为偏向锁1，锁标识位为01；

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）；

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，否则执行线程（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；

执行同步代码块。

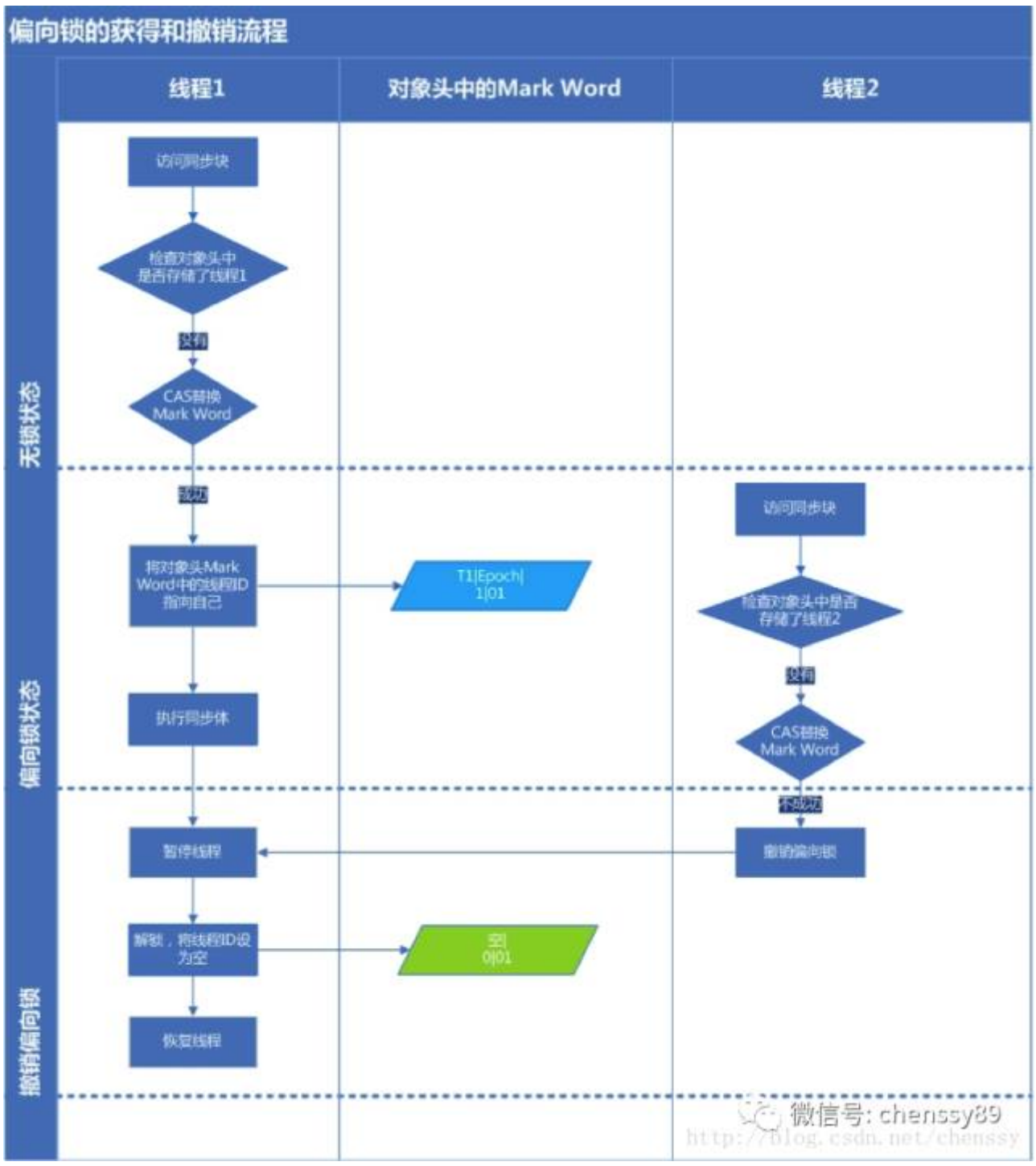
释放锁偏向锁的释放采用了一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要等待全局安全点（这个时间点是上没有正在执行的代码）。

其步骤如下：

暂停拥有偏向锁的线程，判断锁对象石是否还处于被锁定状态；

撤销偏向苏，恢复到无锁状态（01）或者轻量级锁的状态。

下图是偏向锁的获取和释放流程：



11、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 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。

参考资料：

周志明：《深入理解Java虚拟机》

方腾飞：《Java并发编程的艺术》

Java 中 synchronized 的实现原理与应用）