二

# 14 Java虚拟机是怎么实现synchronized的?

在 Java 程序中，我们可以利用 synchronized 关键字来对程序进行加锁。它既可以用来声明一个 synchronized 代码块，也可以直接标记静态方法或者实例方法。

当声明 synchronized 代码块时，编译而成的字节码将包含 monitorenter 和 monitorexit 指令。这两种指令均会消耗操作数栈上的一个引用类型的元素（也就是 synchronized 关键字括号里的引用），作为所要加锁解锁的锁对象。

```
public void foo(Object lock) {
  synchronized (lock) {
    lock.hashCode();
  }
}
// 上面的 Java 代码将编译为下面的字节码
public void foo(java.lang.Object);
  Code:
     0: aload_1
     1: dup
     2: astore_2
     3: monitorenter
     4: aload_1
     5: invokevirtual java/lang/Object.hashCode:()I
     8: pop
     9: aload_2
    10: monitorexit
    11: goto          19
    14: astore_3
    15: aload_2
    16: monitorexit
    17: aload_3
    18: athrow
    19: return
  Exception table:
     from    to  target type
         4    11    14   any
        14    17    14   any
```

我在文稿中贴了一段包含 synchronized 代码块的 Java 代码，以及它所编译而成的字节码。你可能会留意到，上面的字节码中包含一个 monitorenter 指令以及多个 monitorexit 指令。这是因为 Java 虚拟机需要确保所获得的锁在正常执行路径，以及异常执行路径上都能

够被解锁。

你可以根据我在介绍异常处理时介绍过的知识，对照字节码和异常处理表来构造所有可能的执行路径，看看在执行了 monitorenter 指令之后，是否都有执行 monitorexit 指令。

当用 synchronized 标记方法时，你会看到字节码中方法的访问标记包括 ACC_SYNCHRONIZED。该标记表示在进入该方法时，Java 虚拟机需要进行 monitorenter 操作。而在退出该方法时，不管是正常返回，还是向调用者抛异常，Java 虚拟机均需要进行 monitorexit 操作。

```
public synchronized void foo(Object lock) {
  lock.hashCode();
}
// 上面的 Java 代码将编译为下面的字节码
public synchronized void foo(java.lang.Object);
  descriptor: (Ljava/lang/Object;)V
  flags: (0x0021) ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=1, locals=2, args_size=2
       0: aload_1
       1: invokevirtual java/lang/Object.hashCode:()I
       4: pop
       5: return
```

这里 monitorenter 和 monitorexit 操作所对应的锁对象是隐式的。对于实例方法来说，这两个操作对应的锁对象是 this；对于静态方法来说，这两个操作对应的锁对象则是所在类的 Class 实例。

关于 monitorenter 和 monitorexit 的作用，我们可以抽象地理解为每个锁对象拥有一个锁计数器和一个指向持有该锁的线程的指针。

当执行 monitorenter 时，如果目标锁对象的计数器为 0，那么说明它没有被其他线程所持有。在这个情况下，Java 虚拟机会将该锁对象的持有线程设置为当前线程，并且将其计数器加 1。

在目标锁对象的计数器不为 0 的情况下，如果锁对象的持有线程是当前线程，那么 Java 虚拟机可以将其计数器加 1，否则需要等待，直至持有线程释放该锁。

当执行 monitorexit 时，Java 虚拟机则需将锁对象的计数器减 1。当计数器减为 0 时，那便代表该锁已经被释放掉了。

之所以采用这种计数器的方式，是为了允许同一个线程重复获取同一把锁。举个例子，如果一个 Java 类中拥有多个 synchronized 方法，那么这些方法之间的相互调用，不管是直接的还是间接的，都会涉及对同一把锁的重复加锁操作。因此，我们需要设计这么一个可重入

的特性，来避免编程里的隐式约束。

说完抽象的锁算法，下面我们便来介绍 HotSpot 虚拟机中具体的锁实现。

## 重量级锁

重量级锁是 Java 虚拟机中最为基础的锁实现。在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。

Java 线程的阻塞以及唤醒，都是依靠操作系统来完成的。举例来说，对于符合 posix 接口的操作系统（如 macOS 和绝大部分的 Linux），上述操作是通过 pthread 的互斥锁（mutex）来实现的。此外，这些操作将涉及系统调用，需要从操作系统的用户态切换至内核态，其开销非常之大。

为了尽量避免昂贵的线程阻塞、唤醒操作，Java 虚拟机会在线程进入阻塞状态之前，以及被唤醒后竞争不到锁的情况下，进入自旋状态，在处理器上空跑并且轮询锁是否被释放。如果此时锁恰好被释放了，那么当前线程便无须进入阻塞状态，而是直接获得这把锁。

与线程阻塞相比，自旋状态可能会浪费大量的处理器资源。这是因为当前线程仍处于运行状况，只不过跑的是无用指令。它期望在运行无用指令的过程中，锁能够被释放出来。

我们可以用等红绿灯作为例子。Java 线程的阻塞相当于熄火停车，而自旋状态相当于怠速停车。如果红灯的等待时间非常长，那么熄火停车相对省油一些；如果红灯的等待时间非常短，比如说我们在 synchronized 代码块里只做了一个整型加法，那么在短时间内锁肯定会被释放出来，因此怠速停车更加合适。

然而，对于 Java 虚拟机来说，它并不能看到红灯的剩余时间，也就没办法根据等待时间的长短来选择自旋还是阻塞。Java 虚拟机给出的方案是自适应自旋，根据以往自旋等待时是否能够获得锁，来动态调整自旋的时间（循环数目）。

就我们的例子来说，如果之前不熄火等到了绿灯，那么这次不熄火的时间就长一点；如果之前不熄火没等到绿灯，那么这次不熄火的时间就短一点。

自旋状态还带来另外一个副作用，那便是不公平的锁机制。处于阻塞状态的线程，并没有办法立刻竞争被释放的锁。然而，处于自旋状态的线程，则很有可能优先获得这把锁。

## 轻量级锁

你可能见到过深夜的十字路口，四个方向都闪黄灯的情况。由于深夜十字路口的车辆来往可

能比较少，如果还设置红绿灯交替，那么很有可能出现四个方向仅有一辆车在等红灯的情况。

因此，红绿灯可能被设置为闪黄灯的情况，代表车辆可以自由通过，但是司机需要注意观察（个人理解，实际意义请咨询交警部门）。

Java 虚拟机也存在着类似的情形：多个线程在不同的时间段请求同一把锁，也就是说没有锁竞争。针对这种情形，Java 虚拟机采用了轻量级锁，来避免重量级锁的阻塞以及唤醒。

在介绍轻量级锁的原理之前，我们先来了解一下 Java 虚拟机是怎么区分轻量级锁和重量级锁的。

（你可以参照HotSpot Wiki里这张图阅读。）

在对象内存布局那一篇中我曾经介绍了对象头中的标记字段（mark word）。它的最后两位便被用来表示该对象的锁状态。其中，00 代表轻量级锁，01 代表无锁（或偏向锁），10 代表重量级锁，11 则跟垃圾回收算法的标记有关。

当进行加锁操作时，Java 虚拟机会判断是否已经是重量级锁。如果不是，它会在当前线程的当前栈桢中划出一块空间，作为该锁的锁记录，并且将锁对象的标记字段复制到该锁记录中。

然后，Java 虚拟机会尝试用 CAS（compare-and-swap）操作替换锁对象的标记字段。这里解释一下，CAS 是一个原子操作，它会比较目标地址的值是否和期望值相等，如果相等，则替换为一个新的值。

假设当前锁对象的标记字段为 X…XYZ，Java 虚拟机会比较该字段是否为 X…X01。如果是，则替换为刚才分配的锁记录的地址。由于内存对齐的缘故，它的最后两位为 00。此时，该线程已成功获得这把锁，可以继续执行了。

如果不是 X…X01，那么有两种可能。第一，该线程重复获取同一把锁。此时，Java 虚拟机会将锁记录清零，以代表该锁被重复获取。第二，其他线程持有该锁。此时，Java 虚拟机会将这把锁膨胀为重量级锁，并且阻塞当前线程。

当进行解锁操作时，如果当前锁记录（你可以将一个线程的所有锁记录想象成一个栈结构，每次加锁压入一条锁记录，解锁弹出一条锁记录，当前锁记录指的便是栈顶的锁记录）的值为 0，则代表重复进入同一把锁，直接返回即可。

否则，Java 虚拟机会尝试用 CAS 操作，比较锁对象的标记字段的值是否为当前锁记录的地址。如果是，则替换为锁记录中的值，也就是锁对象原本的标记字段。此时，该线程已经成功释放这把锁。

如果不是，则意味着这把锁已经被膨胀为重量级锁。此时，Java 虚拟机会进入重量级锁的释放过程，唤醒因竞争该锁而被阻塞了的线程。

## 偏向锁

如果说轻量级锁针对的情况很乐观，那么接下来的偏向锁针对的情况则更加乐观：从始至终只有一个线程请求某一把锁。

这就好比你在私家庄园里装了个红绿灯，并且庄园里只有你在开车。偏向锁的做法便是在红绿灯处识别来车的车牌号。如果匹配到你的车牌号，那么直接亮绿灯。

具体来说，在线程进行加锁时，如果该锁对象支持偏向锁，那么 Java 虚拟机会通过 CAS 操作，将当前线程的地址记录在锁对象的标记字段之中，并且将标记字段的最后三位设置为 101。

在接下来的运行过程中，每当有线程请求这把锁，Java 虚拟机只需判断锁对象标记字段中：最后三位是否为 101，是否包含当前线程的地址，以及 epoch 值是否和锁对象的类的 epoch 值相同。如果都满足，那么当前线程持有该偏向锁，可以直接返回。

这里的 epoch 值是一个什么概念呢?

我们先从偏向锁的撤销讲起。当请求加锁的线程和锁对象标记字段保持的线程地址不匹配时（而且 epoch 值相等，如若不等，那么当前线程可以将该锁重偏向至自己），Java 虚拟机需要撤销该偏向锁。这个撤销过程非常麻烦，它要求持有偏向锁的线程到达安全点，再将偏向锁替换成轻量级锁。

如果某一类锁对象的总撤销数超过了一个阈值（对应 Java 虚拟机参数 -XX:BiasedLockingBulkRebiasThreshold，默认为 20），那么 Java 虚拟机会宣布这个类的偏向锁失效。

具体的做法便是在每个类中维护一个 epoch 值，你可以理解为第几代偏向锁。当设置偏向锁时，Java 虚拟机需要将该 epoch 值复制到锁对象的标记字段中。

在宣布某个类的偏向锁失效时，Java 虚拟机实则将该类的 epoch 值加 1，表示之前那一代的偏向锁已经失效。而新设置的偏向锁则需要复制新的 epoch 值。

为了保证当前持有偏向锁并且已加锁的线程不至于因此丢锁，Java 虚拟机需要遍历所有线程的 Java 栈，找出该类已加锁的实例，并且将它们标记字段中的 epoch 值加 1。该操作需要所有线程处于安全点状态。

如果总撤销数超过另一个阈值（对应 Java 虚拟机参数 -XX:BiasedLockingBulkRevokeThreshold，默认值为 40），那么 Java 虚拟机会认为这个类已经不再适合偏向锁。此时，Java 虚拟机会撤销该类实例的偏向锁，并且在之后的加锁过程中直接为该类实例设置轻量级锁。

## 总结与实践

今天我介绍了 Java 虚拟机中 synchronized 关键字的实现，按照代价由高至低可分为重量级锁、轻量级锁和偏向锁三种。

重量级锁会阻塞、唤醒请求加锁的线程。它针对的是多个线程同时竞争同一把锁的情况。Java 虚拟机采取了自适应自旋，来避免线程在面对非常小的 synchronized 代码块时，仍会被阻塞、唤醒的情况。

轻量级锁采用 CAS 操作，将锁对象的标记字段替换为一个指针，指向当前线程栈上的一块空间，存储着锁对象原本的标记字段。它针对的是多个线程在不同时间段申请同一把锁的情况。

偏向锁只会在第一次请求时采用 CAS 操作，在锁对象的标记字段中记录下当前线程的地址。在之后的运行过程中，持有该偏向锁的线程的加锁操作将直接返回。它针对的是锁仅会被同一线程持有的情况。

今天的实践环节，我们来验证一个坊间传闻：调用 Object.hashCode() 会关闭该对象的偏向锁 [1]。

你可以采用参数 -XX:+PrintBiasedLockingStatistics 来打印各类锁的个数。由于 C2 使用的是另外一个参数 -XX:+PrintPreciseBiasedLockingStatistics，因此你可以限制 Java 虚拟机仅使用 C1 来即时编译（对应参数 -XX:TieredStopAtLevel=1）。

1. 通过参数 -XX:+UseBiasedLocking，比较开关偏向锁时的输出结果。
2. 在 main 方法的循环前添加 lock.hashCode 调用，并查看输出结果。
3. 在 Lock 类中复写 hashCode 方法，并查看输出结果。
4. 在 main 方法的循环前添加 System.identityHashCode 调用，并查看输出结果。

```
// Run with -XX:+UnlockDiagnosticVMOptions -XX:+PrintBiasedLockingStatistics -XX:Ti
public class SynchronizedTest {

  static Lock lock = new Lock();
  static int counter = 0;

  public static void foo() {
```

```
    synchronized (lock) {
      counter++;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    // lock.hashCode(); // Step 2
    // System.identityHashCode(lock); // Step 4
    for (int i = 0; i < 1_000_000; i++) {
      foo();
    }
  }

  static class Lock {
    // @Override public int hashCode() { return 0; } // Step 3
  }
}
```

[1] https://blogs.oracle.com/dave/biased-locking-in-hotspot

上一页 下一页