链滴

synchronized 的原理及应用

作者：vcjmhg

原文链接：https://ld246.com/article/1622124151378

来源网站：链滴

概述

在之前的一篇文章《"Java内存模型及其原理" 》这篇文章中，曾经介绍过synchronized关键字的作，**解决的是多个线程之间访问资源的同步性**，保证被synchronized修饰的代码具有原子性。因而在本文章将会深入了解synchronized的使用方式以及原理。

使用

synchronized的使用方法主要有如下三种：

- synchronized的使用方式
  - 修饰实例方法
  - 修饰代码块
  - 修饰静态方法

修饰实例方法

作用：对当前实例对象加锁，进入同步代码前要**获取当前实例对象的锁。**

```
synchronized void method() {
  //业务代码
}
```

结合"Java内存模型及其原理"中"缓存导致可见性问题"中的实例代码:

```
public class Test {
  private static long count = 0;
  private void add10K() {
    int idx = 0;
    while(idx++ < 10000) {
```

```
      count += 1;
    }
  }
  public static long calc() throws InterruptedException {
    final Test test = new Test();
    // 创建两个线程，执行add()操作
    Thread th1 = new Thread(()->{
      test.add10K();
    });
    Thread th2 = new Thread(()->{
      test.add10K();
    });
    // 启动两个线程
    th1.start();
    th2.start();
    // 等待两个线程执行结束
    th1.join();
    th2.join();
    return count;
  }

  public static void main(String[] args) throws InterruptedException {
    System.out.println(calc());
  }
}
```

在《"Java内存模型及其原理"》这篇文章中我们讲了，上述代码最后执行的结果是处于10000到2000之间的一个数字，但通过synchronized关键字对add10K()进行修饰后：

```
package thread;

public class Test {
  private static long count = 0;
  //使用synchronized关键字修饰add10K
  private synchronized void add10K() {
    int idx = 0;
    while(idx++ < 10000) {
      count += 1;
    }
  }
  public static long calc() throws InterruptedException {
    final Test test = new Test();
    // 创建两个线程，执行add()操作
    Thread th1 = new Thread(()->{
      test.add10K();
    });
    Thread th2 = new Thread(()->{
      test.add10K();
    });
    // 启动两个线程
    th1.start();
    th2.start();
    // 等待两个线程执行结束
    th1.join();
```

```
    th2.join();
    return count;
  }

  public static void main(String[] args) throws InterruptedException {
    System.out.println(calc());
  }
}
```

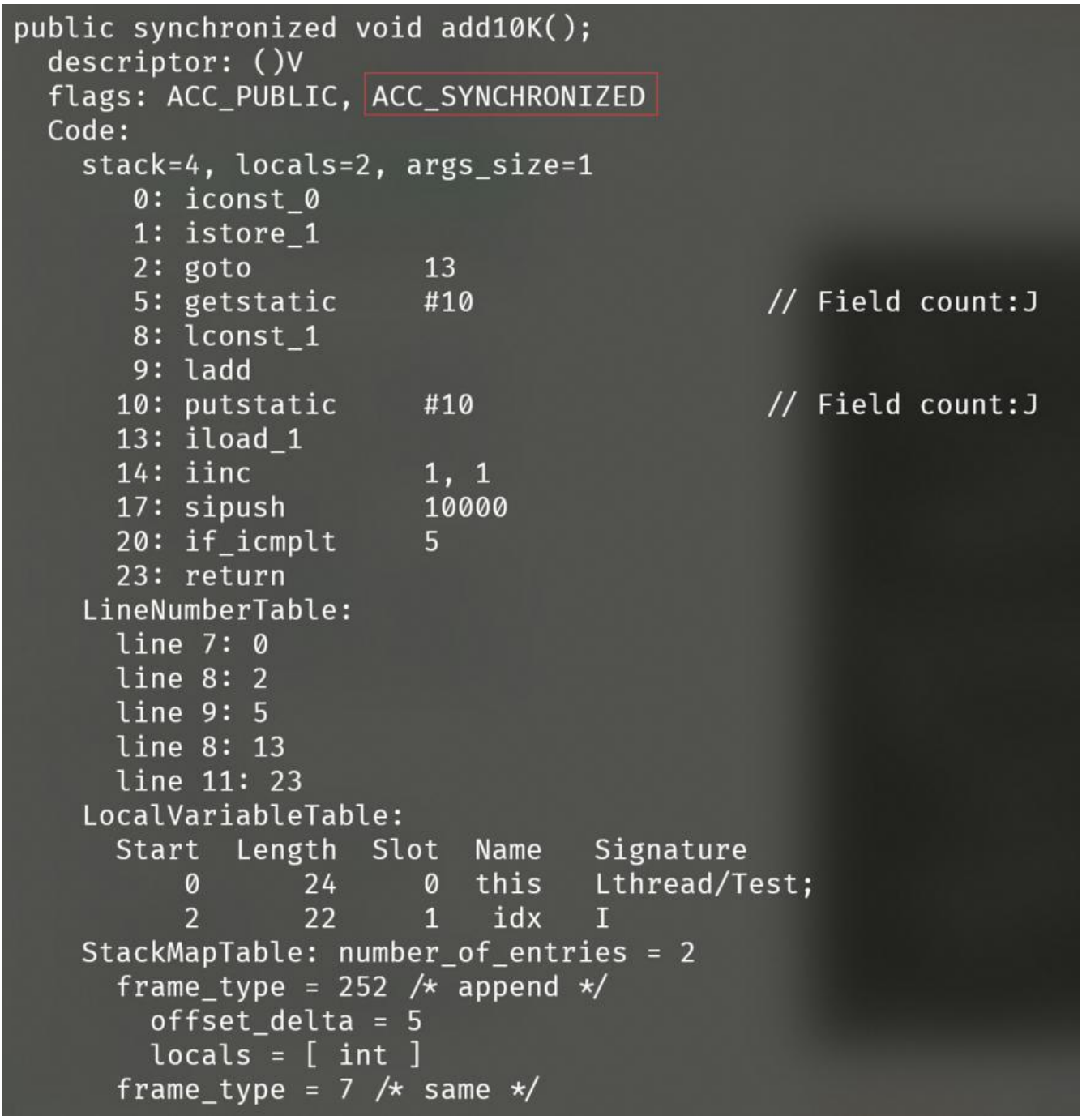
从概念角度上，我们很容易理解，在加入synchronized之后只有一个线程能够进入方法add10K()中其他方法会在该位置阻塞，因而最终执行结果一定是正确的，最终的执行结果一定是**20000**。

但从原理层面上讲，增加了synchronized到底发生了什么□question □
uestion

add10K()被synchronized 修饰后发生了什么？？

要解决这个问题，我们首先需要从JVM角度，上述代码经过synchronized修饰后发生了什么变化。

执行javap -c -s -v -l .\Test.class生成Test类对应的字节码文件如下(仅仅节选了add10K()方法部分)：



经过synchronized修饰的方法会增加增加一个ACC_SYNCHRONIZED标识，该标识会指明该方法是个同步方法。当JVM在执行有ACC_SYNCHRONIZED标识的同步方法时， 会按照同步的策略进行调。

当然除了直接修饰方法外，synchronized也可以用来修饰代码块。

修饰静态方法

作用：给当前所有类加锁，会作用于当前所有类的实例对象，进入同步代码前要获取**当前Class的锁。**

```
synchronized static void method() {
//业务代码
}
```

由于synchronized静态方法和synchronized实例方法加锁的是不同的对象，因而对于两个不同的线可以一个执行synchronized修饰的静态方法，一个执行synchronized修饰的实例方法，两者不会发互斥。

我们还是一一段简单的代码为例

```
public static synchronized int add() {
    int count = 0;
    while(count <= 10000) {
        count += 1;
    }
    return count;
 }
```

同样的add()方法在经过synchronized修饰之后，可以保证是线程安全的。

从jvm层面看，增加了synchronized到底发生了什么？？

静态add()被synchronized` 修饰后发生了什么？？

执行javap -c -s -v -l .\Test.class生成Test类对应的字节码文件如下(仅仅节选了add()方法部分)：

```
public static synchronized int add();
  descriptor: ()I
  flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=0
       0: iconst_0
       1: istore_0
       2: goto          8
       5: iinc          0, 1
       8: iload_0
       9: sipush        10000
      12: if_icmple     5
      15: iload_0
      16: ireturn
    LineNumberTable:
      line 15: 0
      line 16: 2
      line 17: 5
      line 16: 8
      line 19: 15
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          2      15     0 count   I
    StackMapTable: number_of_entries = 2
      frame_type = 252 /* append */
        offset_delta = 5
        locals = [ int ]
      frame_type = 2 /* same */
```

很明显，和前面修饰实例方法一样，经过synchronized修饰的方法会增加一个ACC_SYNCHRONIZED识，在JVM层面上，保证add()方法的同步调用。

修饰代码块

作用：指定加锁对象，对给定对象/类加锁。synchronized(this|object) 表示进入同步代码库前要获得**定对象的锁** 。synchronized(类.class) 表示进入同步代码前要获得 **当前 class 的锁。**

```
synchronized(this) {
  //业务代码
}
```

```
或者
//加类锁
synchronized(Test.class) {
  //业务代码
}
```

同样通过synchronized修饰代码块，我们也可以解决前边add10K()存在的线程安全问题：

```
 public void add10K() {
    int idx = 0;
    synchronized (this) {
       while(idx++ < 10000) {
          count += 1;
       }
    }
 }
```

从概念上讲，通过synchronized修饰while循环之后，整个while循环的加法操作就相当于是一个原子作，只有获取到Test对象锁的线程才能进入执行，其他线程会被阻塞，最终执行结果必然相当于执行边的加法操作，最终结果必然就是**20000**喽。

同样的，从jvm层面看，它有发生了什么呢？

被synchronized修饰的代码块发生了什么？

同样的我们执行javap -c -s -v -l .\Test.class生成Test类对应的字节码文件如下(仅仅节选了add10K()法部分)：

```
public void add10K();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=4, locals=3, args_size=1
       0: iconst_0
       1: istore_1
       2: aload_0
       3: dup
       4: astore_2
       5: monitorenter
       6: goto          17
       9: getstatic     #10                 // Field count:J
      12: lconst_1
      13: ladd
      14: putstatic     #10                 // Field count:J
      17: iload_1
      18: iinc          1, 1
      21: sipush        10000
      24: if_icmplt     9
      27: aload_2
      28: monitorexit
      29: goto          35
      32: aload_2
      33: monitorexit
      34: athrow
      35: return
    Exception table:
       from    to  target type
           6    29    32   any
          32    34    32   any
```

我们可以看到，synchronized修饰的代码块和修饰方法时，作用的机理是不同的（其实也非常容易理解，因为修饰的代码块，没法用标识符来进行标识，只能通过插入指令的形式来实现同步）：

通过synchronized修饰的代码块中会增加两条指令monitorenter指令和monitorexit指令，**其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。**（当我们例子中由于synchronized修饰的是while语句块，可能存在判断分支，因而插入了两条monitorexit来保证在每种分支条件下都能正确执行。）

那这两条指令有什么用呢?

简单来说，**当执行monitorenter时**，会尝试获取对象的锁，如果锁的计数器为 0 则表示锁可以被获，获取后将锁计数器设为 1 也就是加 1。

在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线就要阻塞等待，直到锁被另外一个线程释放为止。

扩展：Java6及以上版本对synchronized的优化

我们再深入思考下，synchronized在加锁和解锁的过程中发生了什么呢
uestion question

这个问题，比较复杂，在Java6之前，synchronized由于所用的锁都是重量级锁，效率较极低，因而了提高synchronized的效率，引入了适应性自旋、锁消除、偏向锁、轻量级锁、重量级锁等措施来对synchronized进行优化。

从对象头的Mark Word说起

我们知道，其实加锁的**本质实际上加锁的对象头写入当前线程的id。** 而实际写入的位置就是在对象的ark word部分，而实际填写的方式就是通过CAS来实现的。

Mark Word用于存储对象自身运行时的数据，如HashCode，GC分代年龄，锁状态标志，线程持有锁，偏向线程ID等等，占用的空间大小与虚拟机位长一致（32位JVM -> MarkWord是32位，64位JM->Mark Word是64位）。

Mark Word会用不同的标记来标识不同的对象状态（比如未被加锁、轻量级锁、重量级锁等状态）：

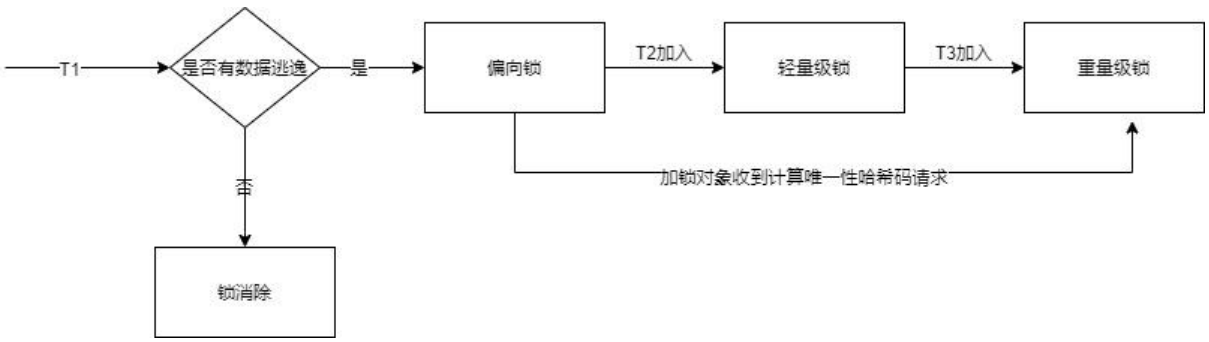
锁状态	32bit				
	25bit		4bit	1bit	2bit
	23bit	2bit		偏向模式	标志位
未锁定	对象哈希码		分代年龄	0	1
轻量级锁定	指向调用栈中锁记录的指针				0
重量级锁定 (锁膨胀)	指向重量级锁的指针				10
GC 标记	空				11
可偏向	线程 ID	Epoch	分代年龄	1	01

此时可能会有小伙伴会有疑问了：不同级别的锁时如何转换的呢?

这就涉及到了锁的升级。

锁的升级

为了便于理解，我们设想一种场景来模拟锁的升级过程，整个过程如下图所示：



原文链接：synchronized 的原理及应用

该开始的时候只有一个线程T1运行，此时T1进入同步代码块时：

● 首先会进行 **逃逸分析**，如果堆上的数据不会逃逸被其他线程访问到，那么就可以认为它是栈上的据，是线程安全的，就会转而进行锁消除操作，忽略所有同步措施直接运行。

如果有数据逃逸的情况发生，此时就会进入**偏向锁**状态，当有其他线程如T2竞争同步代码时，此时**偏锁**会失效，转而升级为**轻量级锁。**当然在处于偏向锁状态的对象，在收到计算唯一性哈希码请求（执hashCode()方法）时，会直接膨胀为重量级锁。

当然**轻量级锁**也不是最终的锁状态，如果有**超过两个线程**比如又加入第三个线程T3来竞争同步资源，就会膨胀，最终就会变成**重量级锁。**

上边的整个过程，就是锁的升级过程，上面提到了好几种类型的锁，直接看有些迷糊，因此下边，我按照不同锁的类型，来解决不同锁的加锁解锁的过程，已经应用的时机。

首先，我们先讲解一下，锁消除的过程。

锁消除

什么叫做锁消除呢?

首先用一下专业术语来解释一下：

锁消除就是指虚拟机即时编译器在运行时，对一些代码要求同步，但对被检测到的不可能存在共享数竞争的锁进行消除。

用大白话解释，锁消除就是说把“锁去掉”，但我们不可能见到一个锁就把它给“去掉”，这加锁就有任何意义了，因而了解锁消除的场景和时机就极为重要了，我们只能在**运行期间**，把那些不可能存在**享数据竞争的锁**进行消除。

如何判断线程是否具有共享数据竞争呢？？

这就需要逃逸分析技术做支持喽，在进行逃逸分析后如果发现，如果堆上的数据都不会逃逸被其他线访问到，那么就可以把它们当做栈上数据对待，认为它们是线程**私有的**，同步加锁自然无需再进行。

当然，如果进行逃逸分析之后，发现确实存在数据逃逸问题，此时就需要加锁。

偏向锁

什么是偏向锁呢?

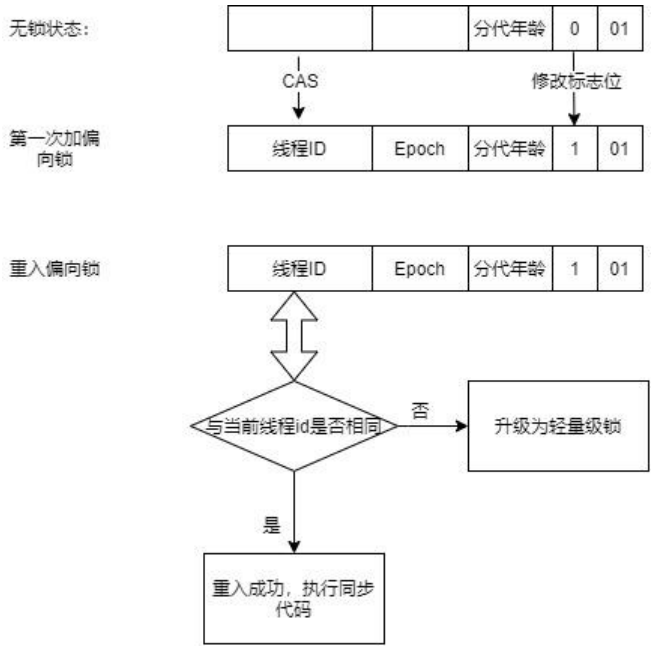
简单来说，这个锁会偏向第一个获取它的线程，如果在接下来的执行过程，该锁一直没有被其他锁获，则持有偏向锁的线程将永远不需要再进行同步，也就是说偏向锁不需要解锁，因而偏向锁的效率极。

为什么要这样做呢? 因为**经验表明**, 其实大部分情况下, 都会是**同一个线程进入同一块同步代码块的，也是为什么会有偏向锁出现的原因。**

在Jdk1.6中, 偏向锁的开关是默认开启的, **适用于只有一个线程访问同步块的场景。**

加锁:

对象从**无锁**到获取到**偏向锁**再到**重入锁**整个过程，如下图所示：



当一个对象处于无锁状态时，它的偏向锁标志为0，标志位为01，**hashcode为空**（计算过hashCode对象无法进入偏向模式）。

当锁对象第一次被线程获取的时候，虚拟机会把对象头的偏向模式设置为1，表示进入偏向模式，同会通过CAS操作获取到这个锁的线程ID并记录到Mark Word中。如果操作成功，持有偏向锁的线程后进入这个锁相关的同步块时，只需要比较偏向锁中的线程ID是否和当前ID一致即可，**如果相同的话就不再需要任何同步操作，直接执行同步代码。**

偏向锁的撤销时机：

偏向锁使用了一种等到竞争出现才释放锁的机制, 所以**当其他线程尝试竞争偏向锁时,** 持有偏向锁的线才会释放锁。

另一点，由于当时使用偏向锁时，Mark Word的大部分空间都用来存储持有锁的线程ID了，**因而当个对象处于偏向锁模式**，又收到需要计算其一致性哈希码请求时，偏向锁会立即撤销，**并膨胀为重量锁。同时已经计算过一致性哈希码的对象无法进入偏向模式。**

具体撤销过程如下：

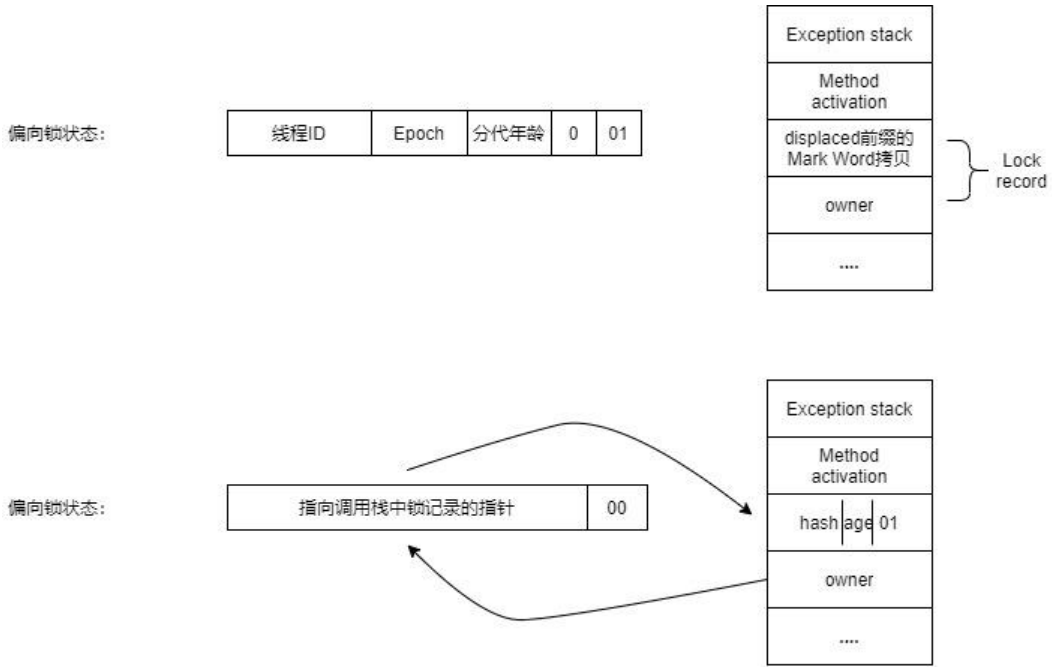
偏向锁的撤销需要等到全局安全点(在这个时间点上没有正在执行的字节码). **首先会暂停持有偏向锁的程**, 然后检查持有偏向锁的线程是否存活, 如果线程不处于活动状态, 则将锁对象的对象头设置为无锁态; 如果线程仍然活着, 则锁对象的对象头中的MarkWord和栈中的锁记录要么重新偏向于其它线程要恢复到无锁状态, 最后唤醒暂停的线程(释放偏向锁的线程)。

轻量级锁

当出现有两个线程来竞争锁的话, 那么偏向锁就失效了, 此时锁就会膨胀, 升级为轻量级锁。

加锁：

从偏向锁或者无锁状态转成**轻量级锁**的过程如下：



在代码进入同步块时，如果此同步对象没有被锁定，虚拟机首先将在当前线程的栈帧中建立起一个名为记录（Lock record）的空间，用于存储当前Mark Word的拷贝（以Displaced为前缀），然后虚拟将尝试通过CAS操作把对象的Mark Word转变为00，表示此对象处于轻量级锁定状态中。如果更新败，则说明存在一条线程与当前线程竞争，此时虚拟机会首先检查对象的Mark Word是否指向当前程的栈帧，如果是，说明当前线程已经拥有了这个对象的锁，直接重入，进入同步代码块直接执行；则，说明这个锁对象已经被其他线程抢占了。

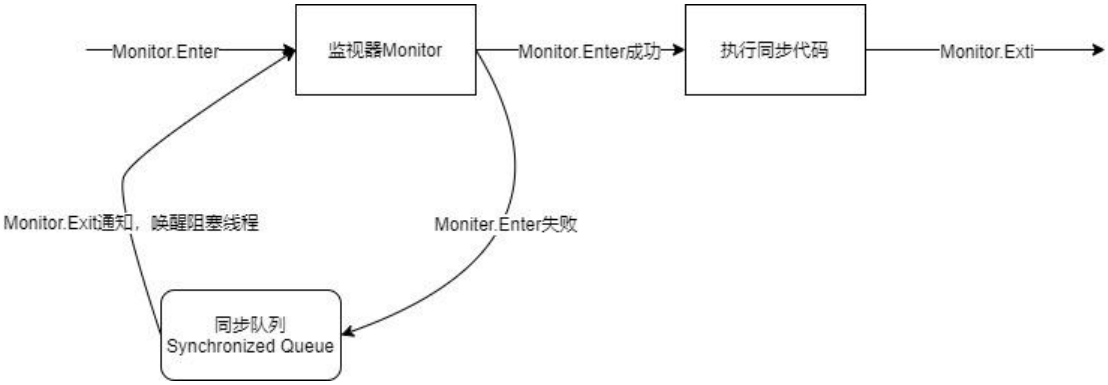
当有出现两条以上的线程争用同一个锁的情况，那轻量级锁就不再有效，会膨胀为重量级锁。

解锁：

轻量级锁解锁时，会使用原子的CAS操作将当前线程的锁记录替换回到对象头，如果成功，表示没有竞发生；**如果失败**，表示当前锁存在竞争，锁就会膨胀成重量级锁。

重量级锁

重量级锁的实现就比较复杂，需要借助前边介绍的monitorenter和monitorexit指令，当加锁时，线会尝试获取加锁对象的monitor（每一个对象都内置了一个monitor对象，由C++实现），获取成功开始执行同步代码，获取失败进入同步队列，当执行到monitorexit指令时，释放对象锁，并唤醒同队列中的阻塞线程。整个执行逻辑如下图所示：



锁的小结

原文链接：synchronized 的原理及应用

最后，简单总结一下锁优缺点：

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, 和执行非同步代码方法的性能相差无.	如果线程间存在锁竞争, 会带来额外的锁撤销的消耗.	用于只有一个线程访问的同步场景
轻量级锁	竞争的线程不会阻塞, 提高了程序的响应速度	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, 同步快执速度非常快
重量级锁	线程竞争不适用自旋, 不会消耗CPU	线堵塞, 响应时间缓慢	追求吞吐量, 同步快执行时间速度较长

总结

本篇文章主要讲了synchronized的三个应用方式：修饰实例方法、修饰静态方法、修饰代码块，并且对每种方式都从JVM角度讲解了其实现的底层原理。在扩展部分则详细讲解了Java6对synchronized优化措施，以及底层的原理。

参考

1. 《深入理解JVM虚拟机》
2. 《Java并发实战》
3. 《Java并发编程实战》
4. Java6及以上版本对synchronized的优化