synchronized底层原理

synchronized作用

- 原子性：synchronized保证语句块内操作是原子的（只能有一个线程去访问）。
- 可见性：synchronized保证可见性（在执行unlock之前，必须先把此变量刷回**主内存**）。
- 有序性：synchronized保证有序性（一个变量在同一时刻只允许一条线程对其进行lock操作）。

synchronized使用

- 修饰实例方法，对当前实例对象加锁
- 修饰静态方法，多当前类的Class对象加锁
- 修饰代码块，对synchronized括号内的对象加锁

实现原理

Synchronized是通过对象内部的一个叫做监视器锁（**monitor**）来实现的，监视器锁本质又是依赖于底层的操作系统的Mutex Lock（互斥锁）来实现的。而操作系统实现线程之间的切换需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。因此，这种依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”。

jvm基于进入和退出Monitor对象来实现**方法同步**和**代码块同步**。

方法级的同步 是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。JVM可以从方法常量池中的方法表结构中的 **ACC_synchronized** 访问标志区分一个方法是否同步方法。当方法调用时，调用指令会检查方法的 ACC_synchronized 访问标志是否被设置，如果设置了，执行线程将先持有monitor（虚拟机规范中用的是管程一词）， 然后再执行方法，最后再方法完成时释放monitor。

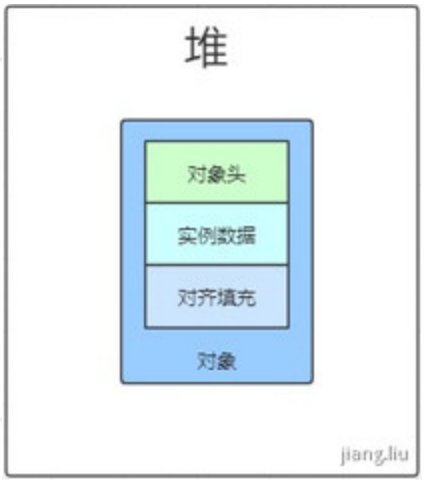
```
public synchronized void f();
  descriptor: ()V
  flags: (0x0021) ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello world
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 4: 0
      line 5: 8
```


代码块的同步 是利用 **monitorenter** 和 **monitorexit** 这两个字节码指令。它们分别位于同步代码块的开始和结束位置。当jvm执行到monitorenter指令时，当前线程试图获取monitor对象的所有权，如果未加锁或者已经被当前线程所持有，就把锁的计数器+1；当执行monitorexit指令时，锁计数器-1；当锁计数器为0时，该锁就被释放了。如果获取monitor对象失败，该线程则会进入阻塞状态，直到其他线程释放锁。

```
public void g();
  descriptor: ()V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter                      <-- 监视器入口
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Hello world
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit                       <-- 监视器出口，释放锁
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit                       <-- 发生异常，释放锁
      20: aload_2
      21: athrow
      22: return
```


理解Java对象头

在JVM中，对象在内存中的布局分为三块区域：**对象头**、**实例数据**、**对齐填充**。



通过JOL分析Java对象布局：

一个Java对象，通过ClassLayout.*parseInstance*(test).toPrintable()；打印出来的对象信息如下：

```
demo.work.Test3 object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           43 c1 00 f8 (01000011 11000001 00000000 11111000) (-134168253)
     12     4    int Test3.a                                   1
     16     4    int Test3.b                                   2
     20     4        (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

jiang.liu

实例变量：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的长度，这部分内存按4字节对齐。

填充数据：由于虚拟机要求对象起始地址必须是8个字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

> 在64位Java虚拟机上面，对象大小必须是8的倍数。

对象头：

32位和64位虚拟机表现不同，这里以主流的64位进行说明。一个对象在内存中存储必须是8字节的整数倍，其中对象头占了12字节。

对象头分为了两部分：

- 第一部分是：指向方法区元数据的**类型指针**（klass point），固定占用4字节32位；
- 第二部分是：用于存储对象hashcode、分代年龄、同步状态、线程id等信息的**mark word**，占用8字节64位。

> 对象头里面包含了 GC 状态，用 4bit 的二进制位存储，因为 4bit 最大是 1111，也就是 15，所以 GC 幸存区的复制算法是 15 次之后进入老年代。

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monitorrecord，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monitorrecord失败的线程。
- RcThis:表示blocked或waiting在该monitorrecord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GCage）。
- 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。

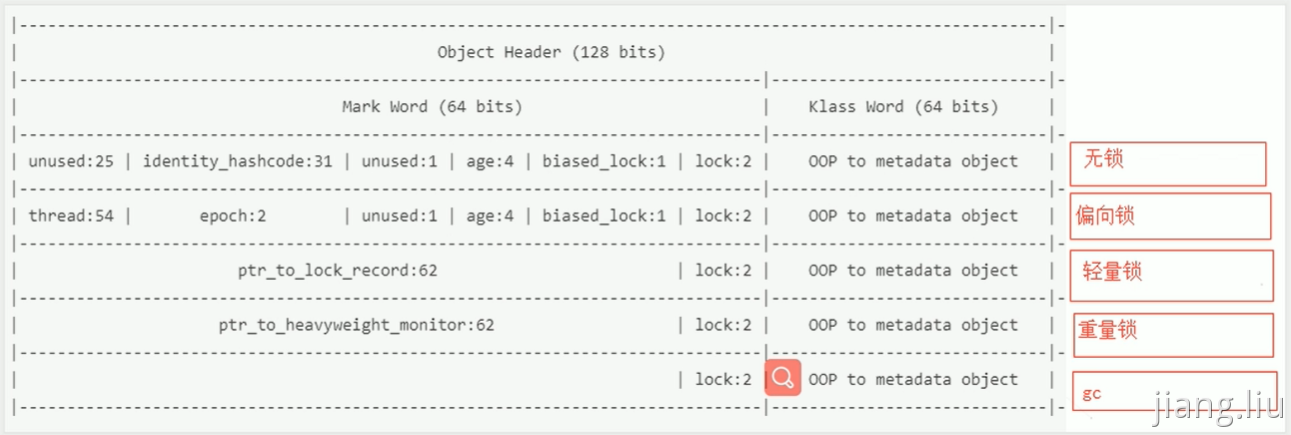
我们知道synchronized是重量级锁，效率不怎么滴，同时这个观念也一直存在我们脑海里，不过在jdk1.6中对synchronize的实现进行了各种优化，使得它显得不是那么重了，那么JVM采用了那些优化手段呢?

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。

Java对象的状态：

1. **无锁：对象刚new出来的时候**
2. **偏向锁：对象只有一个线程获取锁的时候，并且没有其他线程使用。**
3. **轻量锁：多线程交替执行。但不存在锁竞争。**
4. **重量锁：多个线程发生锁竞争。**
5. GC标记 ： 标记垃圾回收。



当锁对象第一次被线程获取的时候，虚拟机会把对象头中的标志位设置为“01”，即偏向模式。同时使用CAS操作把获取到这个锁的线程ID记录在对象的Mark Word中，如果CAS操作成功。持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作。

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，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，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JDK1.4.2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JDK1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明。

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。

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。

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。

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。