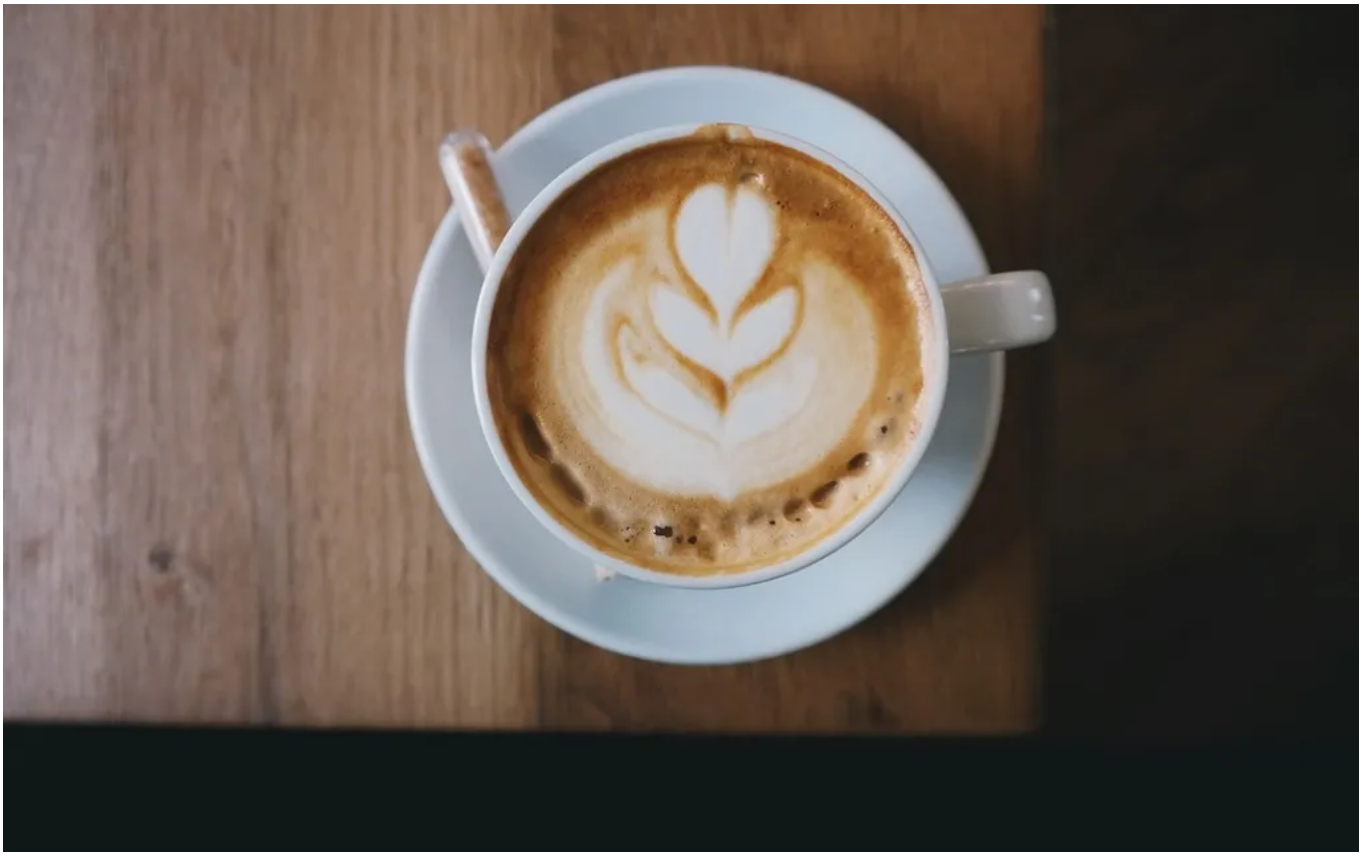
谈谈JVM内部锁升级过程

原创 洋锅 阿里技术 7月1日

收录于话题
#Java

70个



一 为什么讲这个?

总结AQS之后，对这方面顺带的复习一下。本文从以下几个高频问题出发：

- 对象在内存中的内存布局是什么样的?
- 描述synchronized和ReentrantLock的底层实现和重入的底层原理。
- 谈谈AQS，为什么AQS底层是CAS+volatile?
- 描述下锁的四种状态和锁升级过程?
- Object o = new Object() 在内存中占用多少字节?
- 自旋锁是不是一定比重量级锁效率高?
- 打开偏向锁是否效率一定会提升?
- 重量级锁到底重在哪里?
- 重量级锁什么时候比轻量级锁效率高，同样反之呢?

二 加锁发生了什么?

无意识中用到锁的情况:

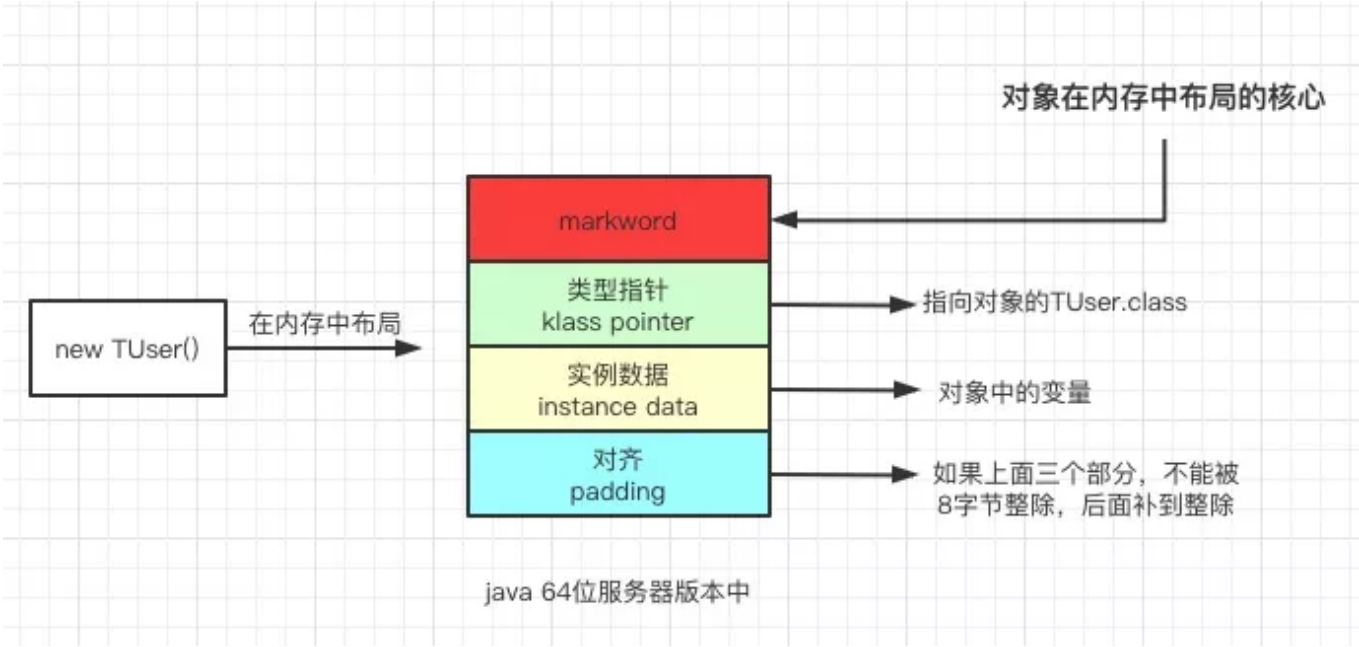
```
//System.out.println都加了锁
public void println(String x) {
  synchronized (this) {
    print(x);
    newLine();
  }
}
```

简单加锁发生了什么?

要弄清楚加锁之后到底发生了什么需要看一下对象创建之后再内存中的布局是个什么样的?

一个对象在new出来之后在内存中主要分为4个部分:

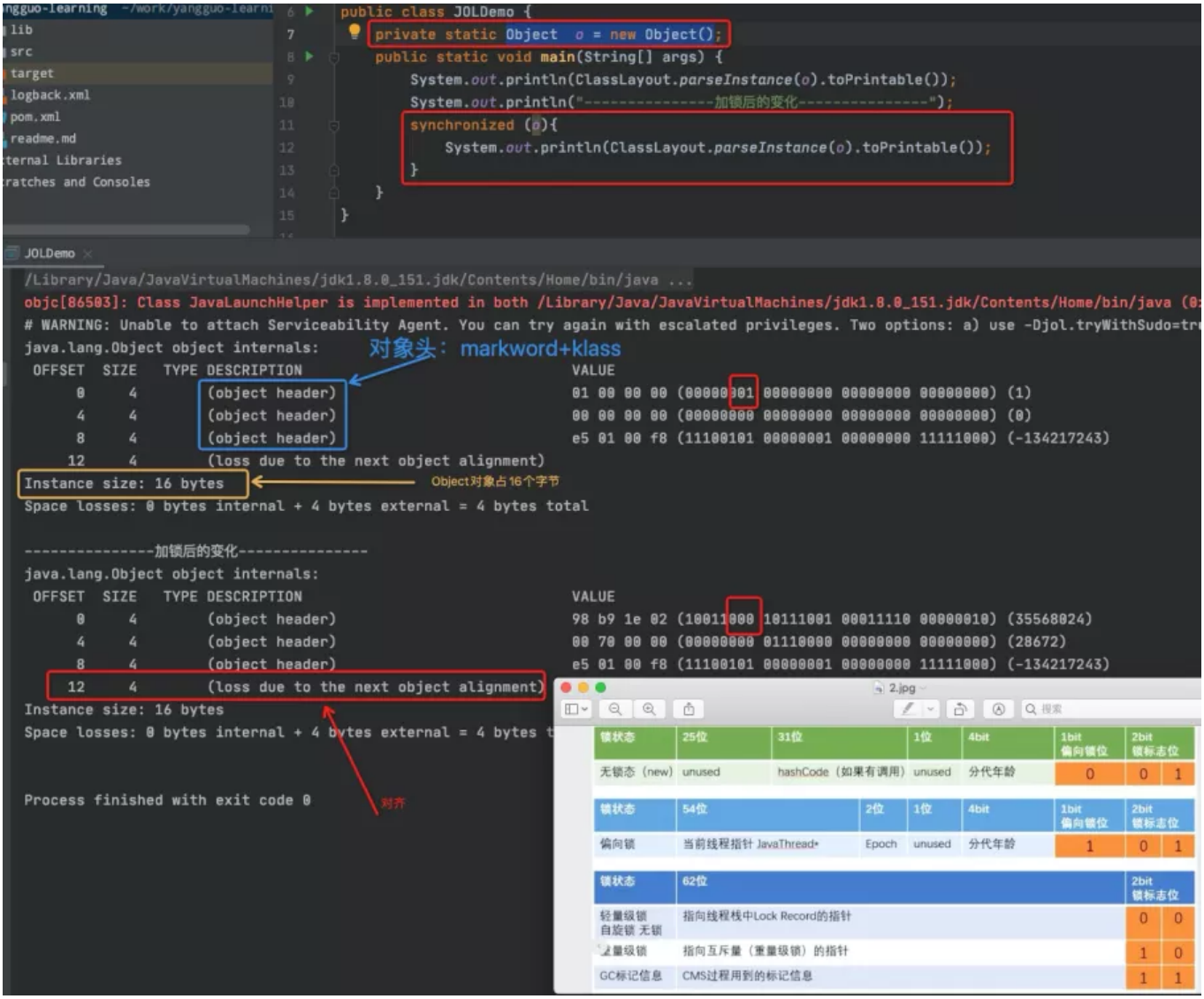
- markword这部分其实就是加锁的核心,同时还包含的对象的一些生命信息,例如是否GC、经过了几次Young GC还存活。
- klass pointer记录了指向对象的class文件指针。
- instance data记录了对象里面的变量数据。
- padding作为对齐使用,对象在64位服务器版本中,规定对象内存必须要能被8字节整除,如果不能整除,那么就靠对齐来补。举个例子:new出了一个对象,内存只占用18字节,但是规定要能被8整除,所以padding=6。



知道了这4个部分之后，我们来验证一下底层。借助于第三方包 JOL = Java Object Layout java内存布局去看看。很简单的几行代码就可以看到内存布局的样式：

```
public class JOLDemo {
    private static Object  o;
    public static void main(String[] args) {
        o = new Object();
        synchronized (o){
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

将结果打印出来：

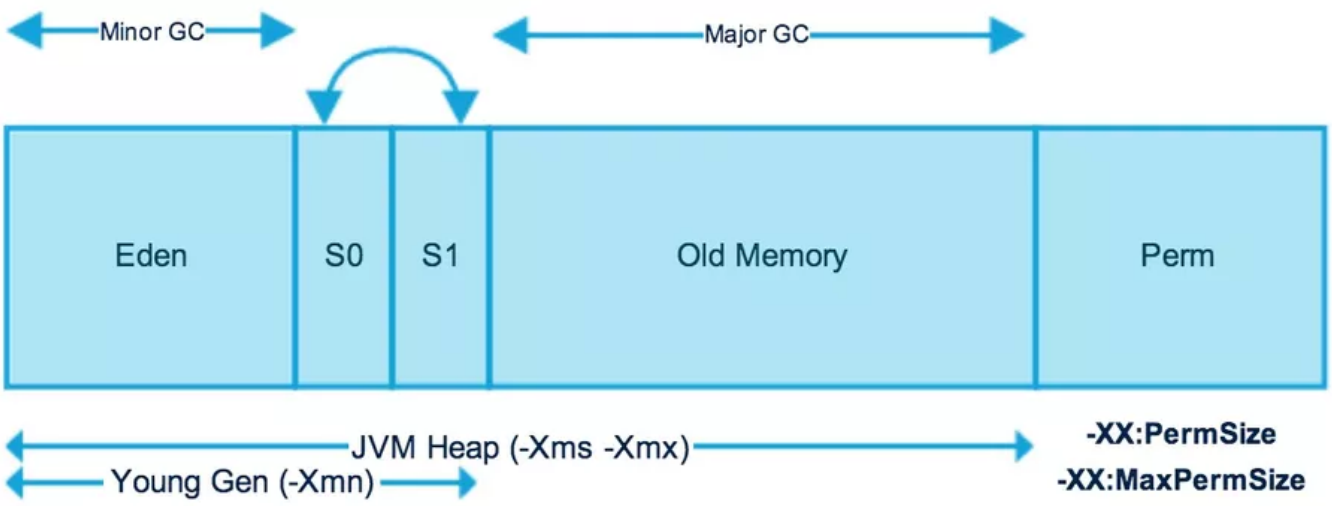


从输出结果看：

1）对象头包含了12个字节分为3行，其中前2行其实就是markword，第三行就是klass指针。值得注意的是在加锁前后输出从001变成了000。Markword用处：8字节(64bit)的头记录一些信息，锁就是修改了markword的内容8字节(64bit)的头记录一些信息，锁就是修改了markword的内容字节(64bit)的头记录一些信息。从001无锁状态，变成了00轻量级锁状态。

2）New出一个object对象，占用16个字节。对象头占用12字节，由于Object中没有额外的变量，所以instance = 0，考虑要对象内存大小要被8字节整除，那么padding=4，最后new Object() 内存大小为16字节。

拓展：什么样的对象会进入老年代？很多场景例如对象太大了可以直接进入，但是这里想探讨的是为什么从Young GC的对象最多经历15次Young GC还存活就会进入Old区（年龄是可以调的，默认是15）。上图中hotspots的markword的图中，用了4个bit去表示分代年龄，那么能表示的最大范围就是0-15。所以这也就是为什么设置新生代的年龄不能超过15，工作中可以通过-XX:MaxTenuringThreshold去调整，但是一般我们不会动。



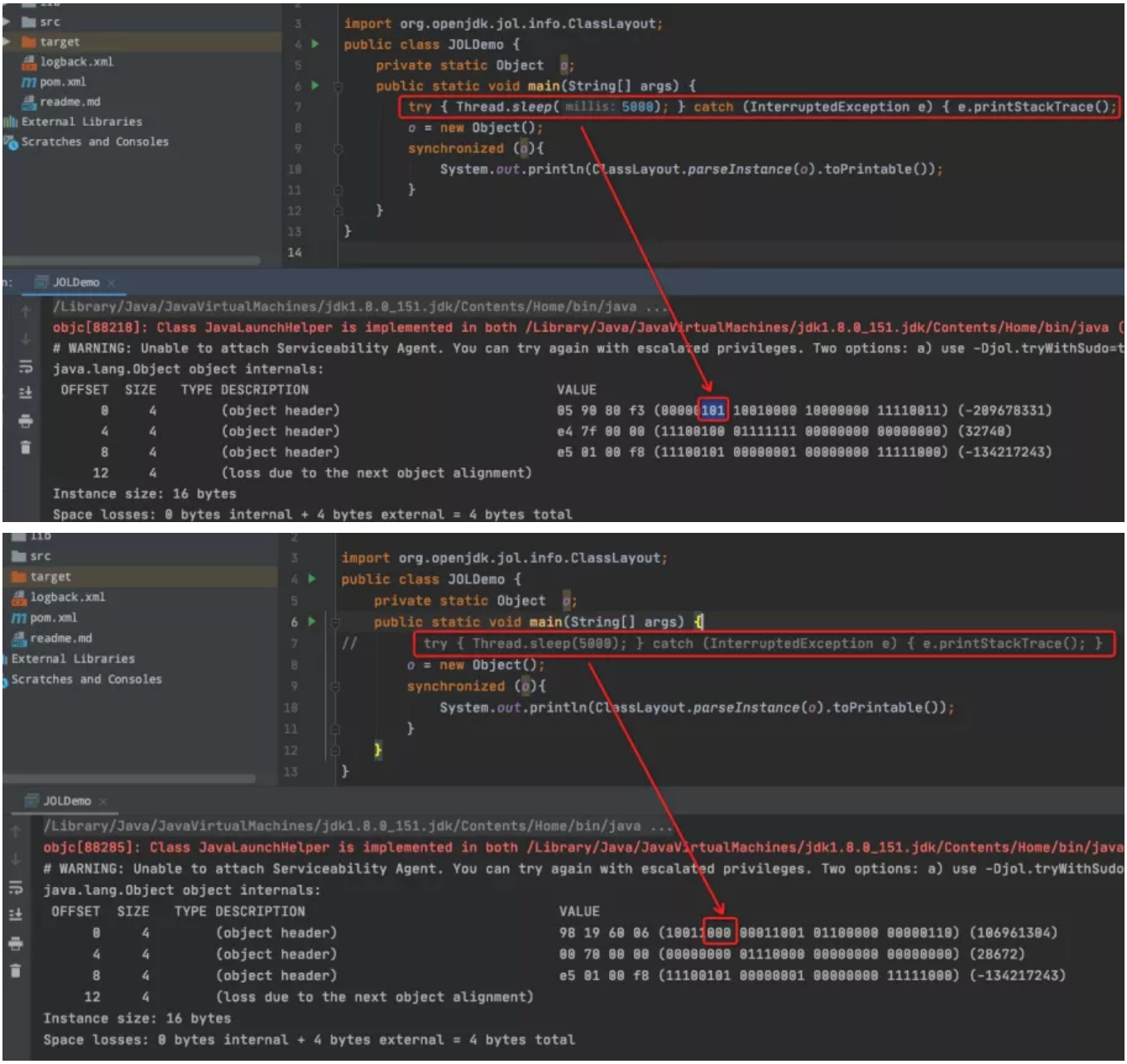
三 锁的升级过程

1 锁的升级验证

探讨锁的升级之前，先做个实验。两份代码，不同之处在于一个中途让它睡了5秒，一个没睡。看看是否有区别。

```
public class JOLDemo {
    private static Object  o;
    public static void main(String[] args) {
        o = new Object();
        synchronized (o){
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
--------------------------------------------------------------------------------
public class JOLDemo {
    private static Object  o;
    public static void main(String[] args) {
      try { Thread.sleep(5000); } catch (InterruptedException e) { e.printStac
        o = new Object();
        synchronized (o){
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

这两份代码会不会有什么区别？运行之后看看结果：



有点意思的是，让主线程睡了5s之后输出的内存布局跟没睡的输出结果居然不一样。

Syn锁升级之后，jdk1.8版本的一个底层默认设置4s之后偏向锁开启。也就是说在4s内是没有开启偏向锁的，加了锁就直接升级为轻量级锁了。

那么这里就有几个问题了？

- 为什么要进行锁升级，以前不是默认syn就是重量级锁么？要么不用要么就用别的不行么？
- 既然4s内如果加了锁就直接到轻量级，那么能不能不要偏向锁，为什么要有偏向锁？
- 为什么要设置4s之后开始偏向锁？

问题1：为什么要进行锁升级？锁了就锁了，不就要加锁么？

首先明确早起jdk1.2效率非常低。那时候syn就是重量级锁，申请锁必须要经过操作系统老大kernel进行系统调用，入队进行排序操作，操作完之后再返回给用户态。

内核态：用户态如果要做一些比较危险的操作直接访问硬件，很容易把硬件搞死（格式化，访问网卡，访问内存干掉、）操作系统为了系统安全分成两层，用户态和内核态 。申请锁资源的时候用户态要向操作系统老大内核态申请。Jdk1.2的时候用户需要跟内核态申请锁，然后内核态还会给用户态。这个过程是非常消耗时间的，导致早期效率特别低。有些jvm就可以处理的为什么还交给操作系统做去呢？能不能把jvm就可以完成的锁操作拉取出来提升效率，所以也就有了锁优化。

问题2：为什么要有偏向锁？

其实这本质上归根于一个概率问题，统计表示，在我们日常用的syn锁过程中70%-80%的情况下，一般都只有一个线程去拿锁，例如我们常使用的System.out.println、StringBuffer，虽然底层加了syn锁，但是基本没有多线程竞争的情况。那么这种情况下，没有必要升级到轻量级锁级别了。偏向的意义在于：第一个线程拿到锁，将自己的线程信息标记在锁上，下次进来就不需要在拿去拿锁验证了。如果超过1个线程去抢锁，那么偏向锁就会撤销，升级为轻量级锁，其实我认为严格意义上来讲偏向锁并不算一把真正的锁，因为只有一个线程去访问共享资源的时候才会有偏向锁这个情况。

无意使用到锁的场景：

```
/***StringBuffer内部同步***/
public synchronized int length() {
  return count;
}

//System.out.println 无意识的使用锁
public void println(String x) {
   synchronized (this) {
     print(x);
     newLine();
   }
 }
```

问题3：为什么jdk8要在4s后开启偏向锁?

其实这是一个妥协，明确知道在刚开始执行代码时，一定有好多线程来抢锁，如果开了偏向锁效率反而降低，所以上面程序在睡了5s之后偏向锁才开放。为什么加偏向锁效率会降低，因为中途多了几个额外的过程，上了偏向锁之后多个线程争抢共享资源的时候要进行锁升级到轻量级锁，这个过程还的把偏向锁进行撤销在进行升级，所以导致效率会降低。为什么是4s？这是一个统计的时间值。

当然我们是可以禁止偏向锁的，通过配置参数-XX:-UseBiasedLocking = false来禁用偏向锁。jdk15之后默认已经禁用了偏向锁。本文是在jdk8的环境下做的锁升级验证。

2 锁的升级流程

上面已经验证了对象从创建出来之后进内存从无锁状态->偏向锁（如果开启了）->轻量级锁的过程。对于锁升级的流程继续往下，轻量级锁之后就会变成重量级锁。首先我们先理解什么叫做轻量级锁，从一个线程抢占资源（偏向锁）到多线程抢占资源升级为轻量级锁，线程如果没那么多的话，其实这里就可以理解为CAS，也就是我们说的Compare and Swap，比较并交换值。在并发编程中最简单的一个例子就是并发包下面的原子操作类AtomicInteger。在进行类似++操作的时候，底层其实就是CAS锁。

```
public final int getAndIncrement() {
  return unsafe.getAndAddInt(this, valueOffset, 1);
}

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

问题4：什么情况下轻量级锁要升级为重量级锁呢?

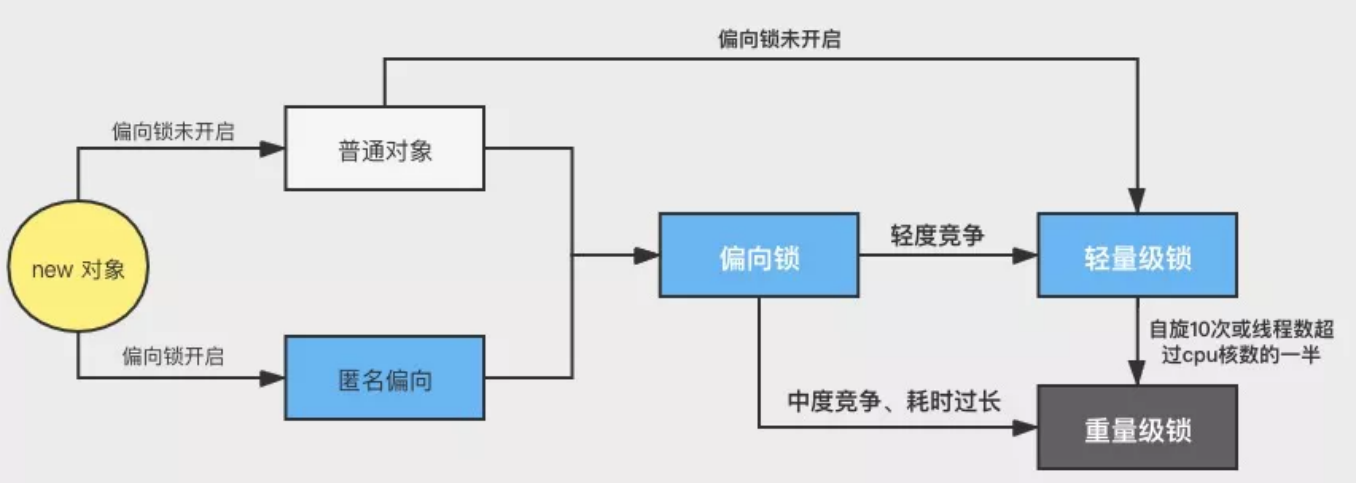
首先我们可以思考的是多个线程的时候先开启轻量级锁，如果它carry不了的情况下才会升级为重量级。那么什么情况下轻量级锁会carry不住。1、如果线程数太多，比如上来就是10000个，那么这里CAS要转多久才可能交换值，同时CPU光在这10000个活着的线程中来回切换中就耗费了巨大的资源，这种情况下自然就升级为重量级锁，直接叫给操作系统入队管理，那么就算10000个线程那也是处理休眠的情况等待排队唤醒。2、CAS如果自旋10次依然没有获取到锁，那么也会升级为重量级。

总的来说2种情况会从轻量级升级为重量级，10次自旋或等待cpu调度的线程数超过cpu核数的一半，自动升级为重量级锁。看服务器CPU的核数怎么看，输入top指令，然后按1就可以看到。

问题5：都说syn为重量级锁，那么到底重在哪里?

JVM偷懒把任何跟线程有关的操作全部交给操作系统去做，例如调度锁的同步直接交给操作系统去执行，而在操作系统中要执行先要入队，另外操作系统启动一个线程时需要消耗很多资源，消耗资源比较重，重就重在这里。

整个锁升级过程如图所示：



四 synchronized的底层实现

上面我们对对象的内存布局有了一些了解之后，知道锁的状态主要存放在markword里面。这里我们看看底层实现。

```
1 public class RnEnterLockDemo {
2       public void method() {
```

```
3           synchronized (this) {
4               System.out.println("start");
5           }
6       }
7   }
```

对这段简单代码进行反解析看看什么情况。javap -c RnEnterLockDemo.class

```
yuanyang@B-R1ZFLVDL-1905 learning % clear
yuanyang@B-R1ZFLVDL-1905 learning % javap -c ReEnterLockDemo.class
Compiled from "ReEnterLockDemo.java"
public class com.yangguotest.learning.ReEnterLockDemo {
  public com.yangguotest.learning.ReEnterLockDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void m1();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String =====外层
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_0
      13: invokevirtual #5                  // Method m2:()V
      16: aload_1
      17: monitorexit
      18: goto          26
      21: astore_2
      22: aload_1
      23: monitorexit
      24: aload_2
      25: athrow
```

首先我们能确定的是syn肯定是还有加锁的操作，看到的信息中出现了monitorenter和monitorexit，主观上就可以猜到这是跟加锁和解锁相关的指令。有意思的是1个monitorenter和2个monitorexit。为什么呢？正常来说应该就是一个加锁和一个释放锁啊。其实这里也体现了syn和lock的区别。syn是JVM层面的锁，如果异常了不用自己释放，jvm会自动帮助释放，这一步就取决于多出来的那个monitorexit。而lock异常需要我们手动补获并释放的。

关于这两条指令的作用，我们直接参考JVM规范中描述：

> monitorenter：

> 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

翻译一下：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

- 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
- 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
- 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

> monitorexit：
>
> 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.

翻译一下：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor的所有权。

通过这段话的描述，很清楚的看出Synchronized的实现原理，Synchronized底层通过一个monitor的对象来完成，wait/notify等方法其实也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常。

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，如果目标对象的计数器为零，那么说明它没有被其他线程所持有，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i。在目标锁对象的计数器不为零的情况下，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。计数器为零代表锁已被释放。

总结

以往的经验中，只要用到synchronized就以为它已经成为了重量级锁。在jdk1.2之前确实如此，后来发现太重了，消耗了太多操作系统资源，所以对synchronized进行了优化。以后可以直接用，至于锁的力度如何，JVM底层已经做好了我们直接用就行。

最后再看看开头的几个问题，是不是都理解了呢。带着问题去研究，往往会更加清晰。希望对大家有所帮助。

收录于话题 #Java·70个

上一篇
重温设计模式之 Factory

下一篇
从操作系统层面分析Java IO演进之路