

讲堂 > Java核心技术36讲 > 文章详情

## 第16讲 | synchronized底层如何实现？什么是锁的升级、降级？

2018-06-12 杨晓峰



### 第16讲 | synchronized底层如何实现？什么是锁的升级、降级？

朗读者：黄洲君 11'02" | 5.06M

我在[上一讲](#)对比和分析了 `synchronized` 和 `ReentrantLock`，算是专栏进入并发编程阶段的热身，相信你已经对线程安全，以及如何使用基本的同步机制有了基础，今天我们将深入了解 `synchronize` 底层机制，分析其他锁实现和应用场景。

今天我要问你的问题是，**synchronized 底层如何实现？什么是锁的升级、降级？**

### 典型回答

在回答这个问题前，先简单复习一下上一讲的知识点。`synchronized` 代码块是由一对 `monitorenter/monitorexit` 指令实现的，`Monitor` 对象是同步的基本实现单元。

在 Java 6 之前，`Monitor` 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作。

现代的 ( Oracle ) `JDK` 中，`JVM` 对此进行了大刀阔斧地改进，提供了三种不同的 `Monitor` 实现，也就是常说的三种不同的锁：偏斜锁 ( `Biased Locking` )、轻量级锁和重量级锁，大大改

进了其性能。

所谓锁的升级、降级，就是 JVM 优化 `synchronized` 运行的机制，当 JVM 检测到不同的竞争状况时，会自动切换到适合的锁实现，这种切换就是锁的升级、降级。

当没有竞争出现时，默认会使用偏斜锁。JVM 会利用 CAS 操作（[compare and swap](#)），在对象头上的 Mark Word 部分设置线程 ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁。这样做的假设是基于在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

如果有另外的线程试图锁定某个已经被偏斜过的对象，JVM 就需要撤销（`revoke`）偏斜锁，并切换到轻量级锁实现。轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁。

我注意到有的观点认为 Java 不会进行锁降级。实际上据我所知，锁降级确实是会发生的，当 JVM 进入安全点（[SafePoint](#)）的时候，会检查是否有闲置的 Monitor，然后试图进行降级。

## 考点分析

今天的问题主要是考察你对 Java 内置锁实现的掌握，也是并发的经典题目。我在前面给出的典型回答，涵盖了一些基本概念。如果基础不牢，有些概念理解起来就比较晦涩，我建议还是尽量理解和掌握，即使有不懂的也不用担心，在后续学习中还会逐步加深认识。

我个人认为，能够基础性地理解这些概念和机制，其实对于大多数并发编程已经足够了，毕竟大部分工程师未必会进行更底层、更基础的研发，很多时候解决的是知道与否，真正的提高还要靠实践踩坑。

后面我会进一步分析：

- 从源码层面，稍微展开一些 `synchronized` 的底层实现，并补充一些上面答案中欠缺的细节，有同学反馈这部分容易被问到。如果你对 Java 底层源码有兴趣，但还没有找到入手点，这里可以成为一个切入点。
- 理解并发包中 `java.util.concurrent.lock` 提供的其他锁实现，毕竟 Java 可不是只有 `ReentrantLock` 一种显式的锁类型，我会结合代码分析其使用。

## 知识扩展

我在[上一讲](#)提到过 `synchronized` 是 JVM 内部的 Intrinsic Lock，所以偏斜锁、轻量级锁、重量级锁的代码实现，并不在核心类库部分，而是在 JVM 的代码中。

Java 代码运行可能是解释模式也可能是编译模式（如果不记得，请复习[专栏第 1 讲](#)），所以对应的同步逻辑实现，也会分散在不同模块下，比如，解释器版本就是：

## [src/hotspot/share/interpreter/interpreterRuntime.cpp](#)

为了简化便于理解，我这里会专注于通用的基类实现：

## [src/hotspot/share/runtime/](#)

另外请注意，链接指向的是最新 JDK 代码库，所以可能某些实现与历史版本有所不同。

首先，synchronized 的行为是 JVM runtime 的一部分，所以我们需要先找到 Runtime 相关的功能实现。通过在代码中查询类似 “monitor\_enter” 或 “Monitor Enter”，很直观的就可以定位到：

- [sharedRuntime.cpp](#)/hpp，它是解释器和编译器运行时的基类。
- [synchronizer.cpp](#)/hpp，JVM 同步相关的各种基础逻辑。

在 sharedRuntime.cpp 中，下面代码体现了 synchronized 的主要逻辑。

```
1 Handle h_obj(THREAD, obj);
2   if (UseBiasedLocking) {
3       // Retry fast entry if bias is revoked to avoid unnecessary inflation
4       ObjectSynchronizer::fast_enter(h_obj, lock, true, CHECK);
5   } else {
6       ObjectSynchronizer::slow_enter(h_obj, lock, CHECK);
7   }
```

[复制代码](#)

其实现可以简单进行分解：

- UseBiasedLocking 是一个检查，因为，在 JVM 启动时，我们可以指定是否开启偏斜锁。

偏斜锁并不适合所有应用场景，撤销操作（revoke）是比较重的行为，只有当存在较多不会真正竞争的 synchronized 块儿时，才能体现出明显改善。实践中对于偏斜锁的一直是有争议的，有人甚至认为，当你需要大量使用并发类库时，往往意味着你不需要偏斜锁。从具体选择来看，我还是建议需要在实践中进行测试，根据结果再决定是否使用。


还有一方面是，偏斜锁会延缓 JIT 预热的进程，所以很多性能测试中会显式地关闭偏斜锁，命令如下：

```
1 -XX:-UseBiasedLocking
2
```

[复制代码](#)

- fast\_enter 是我们熟悉的完整锁获取路径，slow\_enter 则是绕过偏斜锁，直接进入轻量级锁获取逻辑。

那么 `fast_enter` 是如何实现的呢？同样是通过在代码库搜索，我们可以定位到 `synchronizer.cpp`。类似 `fast_enter` 这种实现，解释器或者动态编译器，都是拷贝这段基础逻辑，所以如果我们修改这部分逻辑，要保证一致性。这部分代码是非常敏感的，微小的问题都可能导致死锁或者正确性问题。

 复制代码

```
1 void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock,
2                                     bool attempt_rebias, TRAPS) {
3     if (UseBiasedLocking) {
4         if (!SafepointSynchronize::is_at_safepoint()) {
5             BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREE
6             if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
7                 return;
8             }
9             } else {
10                assert(!attempt_rebias, "can not rebias toward VM thread");
11                BiasedLocking::revoke_at_safepoint(obj);
12            }
13            assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
14        }
15    }
16    slow_enter(obj, lock, THREAD);
17 }
18
```

我来分析下这段逻辑实现：

- [biasedLocking](#)定义了偏斜锁相关操作，`revoke_and_rebias` 是获取偏斜锁的入口方法，`revoke_at_safepoint` 则定义了当检测到安全点时的处理逻辑。
- 如果获取偏斜锁失败，则进入 `slow_enter`。
- 这个方法里面同样检查是否开启了偏斜锁，但是从代码路径来看，其实如果关闭了偏斜锁，是不会进入这个方法的，所以算是个额外的保障性检查吧。

另外，如果你仔细查看[synchronizer.cpp](#)里，会发现不仅仅是 `synchronized` 的逻辑，包括从本地代码，也就是 JNI，触发的 Monitor 动作，全都可以在里面找到（`jni_enter/jni_exit`）。

关于[biasedLocking](#)的更多细节我就不展开了，明白它是通过 CAS 设置 Mark Word 就完全够用了，对象头中 Mark Word 的结构，可以参考下图：

普通对象

Unused(25)	Hash(31)	Unused(1)	Age(4)	Biased lock(1)	lock(2)
------------	----------	-----------	--------	----------------	---------

被偏斜的对象

<u>Thread pointer(54)</u>	Epoch(2)	Unused(1)	Age(4)	Biased lock(1)	Lock(2)
---------------------------	----------	-----------	--------	----------------	---------

顺着锁升降级的过程分析下去，偏斜锁到轻量级锁的过程是如何实现的呢？

我们来看看 `slow_enter` 到底做了什么。

```

1 void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
2     markOop mark = obj->mark();
3     if (mark->is_neutral()) {
4         // 将目前的 Mark Word 复制到 Displaced Header 上
5         lock->set_displaced_header(mark);
6         // 利用 CAS 设置对象的 Mark Word
7         if (mark == obj()->cas_set_mark((markOop) lock, mark)) {
8             TEVENT(slow_enter: release stacklock);
9             return;
10        }
11        // 检查存在竞争
12    } else if (mark->has_locker() &&
13              THREAD->is_lock_owned((address)mark->locker())) {
14        // 清除
15        lock->set_displaced_header(NULL);
16        return;
17    }
18
19    // 重置 Displaced Header
20    lock->set_displaced_header(markOopDesc::unused_mark());
21    ObjectSynchronizer::inflate(THREAD,
22                               obj(),
23                               inflate_cause_monitor_enter->enter(THREAD));
24 }
25

```

[复制代码](#)

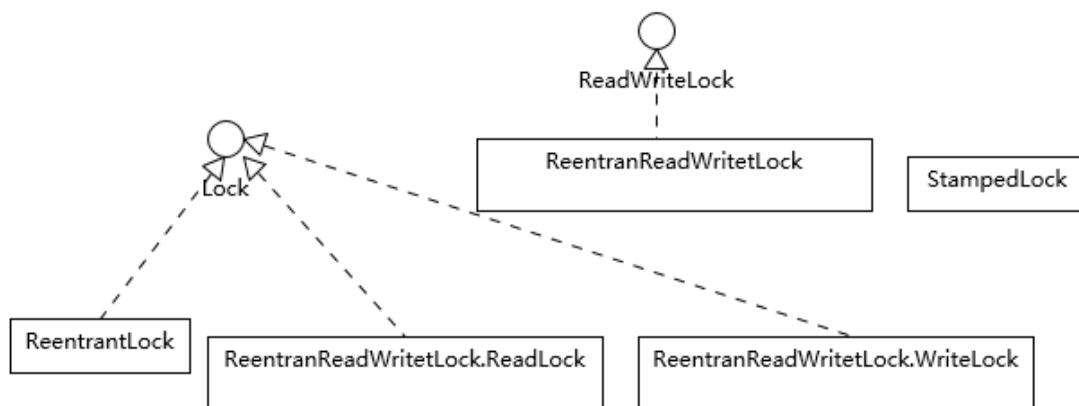
请结合我在代码中添加的注释，来理解如何从试图获取轻量级锁，逐步进入锁膨胀的过程。你可以发现这个处理逻辑，和我在这一讲最初介绍的过程是十分吻合的。

- 设置 Displaced Header，然后利用 `cas_set_mark` 设置对象 Mark Word，如果成功就成功获取轻量级锁。
- 否则 Displaced Header，然后进入锁膨胀阶段，具体实现在 `inflate` 方法中。

今天就不介绍膨胀的细节了，我这里提供了源代码分析的思路和样例，考虑到应用实践，再进一步增加源代码解读意义不大，有兴趣的同学可以参考我提供的[synchronizer.cpp](#)链接，例如：

- `deflate_idle_monitors` 是分析锁降级逻辑的入口，这部分行为还在进行持续改进，因为其逻辑是在安全点内运行，处理不当可能拖长 JVM 停顿（STW，stop-the-world）的时间。
- `fast_exit` 或者 `slow_exit` 是对应的锁释放逻辑。

前面分析了 `synchronized` 的底层实现，理解起来有一定难度，下面我们来看一些相对轻松的内容。我在上一讲对比了 `synchronized` 和 `ReentrantLock`，Java 核心类库中还有其他一些特别的锁类型，具体请参考下面的图。



你可能注意到了，这些锁竟然不都是实现了 `Lock` 接口，`ReadWriteLock` 是一个单独的接口，它通常是代表了一对儿锁，分别对应只读和写操作，标准类库中提供了再入版本的读写锁实现（`ReentrantReadWriteLock`），对应的语义和 `ReentrantLock` 比较相似。

`StampedLock` 竟然也是个单独的类型，从类图结构可以看出它是不支持再入性的语义的，也就是它不是以持有锁的线程为单位。


为什么我们需要读写锁（`ReadWriteLock`）等其他锁呢？

这是因为，虽然 `ReentrantLock` 和 `synchronized` 简单实用，但是行为上有一定局限性，通俗点说就是“太霸道”，要么不占，要么独占。实际应用场景中，有的时候不需要大量竞争的写操作，而是以并发读取为主，如何进一步优化并发操作的粒度呢？

Java 并发包提供的读写锁等扩展了锁的能力，它所基于的原理是多个读操作是不需要互斥的，因为读操作并不会更改数据，所以不存在互相干扰。而写操作则会导致并发一致性的问题，所以写线程之间、读写线程之间，需要精心设计的互斥逻辑。



下面是一个基于读写锁实现的数据结构，当数据量较大，并发读多、并发写少的时候，能够比纯同步版本凸显出优势。


 复制代码

```
1 public class RWSample {
2     private final Map<String, String> m = new TreeMap<>();
3     private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
4     private final Lock r = rwl.readLock();
5     private final Lock w = rwl.writeLock();
6     public String get(String key) {
7         r.lock();
8         System.out.println(" 读锁锁定! ");
9         try {
10             return m.get(key);
11         } finally {
12             r.unlock();
13         }
14     }
15
16     public String put(String key, String entry) {
17         w.lock();
18         System.out.println(" 写锁锁定! ");
19         try {
20             return m.put(key, entry);
21         } finally {
22             w.unlock();
23         }
24     }
25     // ...
26 }
27
```

在运行过程中，如果读锁试图锁定时，写锁是被某个线程持有，读锁将无法获得，而只好等待对方操作结束，这样就可以自动保证不会读取到有争议的数据。

读写锁看起来比 `synchronized` 的粒度似乎细一些，但在实际应用中，其表现也并不尽如人意，主要还是因为相对比较大的开销。

所以，JDK 在后期引入了 `StampedLock`，在提供类似读写锁的同时，还支持优化读模式。优化读基于假设，大多数情况下读操作并不会和写操作冲突，其逻辑是先试着修改，然后通过 `validate` 方法确认是否进入了写模式，如果没有进入，就成功避免了开销；如果进入，则尝试获取读锁。请参考我下面的样例代码。

 复制代码

```
1 public class StampedSample {
2     private final StampedLock sl = new StampedLock();
3
4     void mutate() {
```

```
5      long stamp = sl.writeLock();
6      try {
7          write();
8      } finally {
9          sl.unlockWrite(stamp);
10     }
11 }
12
13 Data access() {
14     long stamp = sl.tryOptimisticRead();
15     Data data = read();
16     if (!sl.validate(stamp)) {
17         stamp = sl.readLock();
18         try {
19             data = read();
20         } finally {
21             sl.unlockRead(stamp);
22         }
23     }
24     return data;
25 }
26 // ...
27 }
28
```

注意，这里的 `writeLock` 和 `unlockWrite` 一定要保证成对调用。

你可能很好奇这些显式锁的实现机制，Java 并发包内的各种同步工具，不仅仅是各种 `Lock`，其他的如[Semaphore](#)、[CountDownLatch](#)，甚至是早期的[FutureTask](#)等，都是基于一种[AQS](#)框架。

今天，我全面分析了 `synchronized` 相关实现和内部运行机制，简单介绍了并发包中提供的其他显式锁，并结合样例代码介绍了其使用方法，希望对你有所帮助。

## 一课一练

关于今天我们讨论的你做到心中有数了吗？思考一个问题，你知道“自旋锁”是做什么的吗？它的使用场景是什么？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。





# Java核心技术36讲

—— 前 Oracle 首席工程师  
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



版权归极客邦科技所有，未经许可不得转载

写留言

## 精选留言



公号-Java大后端

19

自旋锁:竞争锁的失败的线程，并不会真实的在操作系统层面挂起等待，而是JVM会让线程做几个空循环(基于预测在不久的将来就能获得)，在经过若干次循环后，如果可以获得锁，那么进入临界区，如果还不能获得锁，才会真实的将线程在操作系统层面进行挂起。

适用场景:自旋锁可以减少线程的阻塞，这对于锁竞争不激烈，且占用锁时间非常短的代码块来说，有较大的性能提升，因为自旋的消耗会小于线程阻塞挂起操作的消耗。

如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，线程自旋的消耗大于线程阻塞挂起操作的消耗，造成cpu的浪费。

2018-06-12

作者回复

不错，自旋是种乐观情况的优化

2018-06-12



yearning

15

这次原理真的看了很久，一直鼓劲自己，看不懂就是说明自己有突破。

下面看了并发编程对于自旋锁的了解，同时更深刻理解同步锁的性能。

自旋锁采用让当前线程不停循环体内执行实现，当循环条件被其他线程改变时，才能进入临界区。

由于自旋锁只是将当前线程不停执行循环体，不进行线程状态的改变，所以响应会更快。但当线程不停增加时，性能下降明显。

线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁。

为什么会提出自旋锁，因为互斥锁，在线程的睡眠和唤醒都是复杂而昂贵的操作，需要大量的CPU指令。如果互斥仅仅被锁住是一小段时间，用来进行线程休眠和唤醒的操作时间比睡眠时间还长，更有可能比不上不断自旋锁上轮询的时间长。

当然自旋锁被持有的时间更长，其他尝试获取自旋锁的线程会一直轮询自旋锁的状态。这将十分浪费CPU。

在单核CPU上，自旋锁是无用，因为当自旋锁尝试获取锁不成功会一直尝试，这会一直占用CPU，其他线程不可能运行，同时由于其他线程无法运行，所以当前线程无法释放锁。

混合型互斥锁，在多核系统上起初表现的像自旋锁一样，如果一个线程不能获取互斥锁，它不会马上被切换为休眠状态，在一段时间依然无法获取锁，进行睡眠状态。

混合型自旋锁，起初表现的和正常自旋锁一样，如果无法获取互斥锁，它也许会放弃该线程的执行，并允许其他线程执行。

切记，自旋锁只有在多核CPU上有效果，单核毫无效果，只是浪费时间。

以上基本参考来源于：

[http://ifeve.com/java\\_lock\\_see1/](http://ifeve.com/java_lock_see1/)

<http://ifeve.com/practice-of-using-spinlock-instead-of-mutex/>

2018-06-12

作者回复

很不错总结

2018-06-12



sunlight001

12

自旋锁是尝试获取锁的线程不会立即阻塞，采用循环的方式去获取锁，好处是减少了上下文切换，缺点是消耗cpu

2018-06-12

作者回复

不错

2018-06-12



黑子

6

自旋锁 for(;;)结合cas确保线程获取取锁

2018-06-12

| 作者回复

差不多

2018-06-12



jacy

👍 5

看了大家对自旋锁的评论，我的收获如下：

1. 基于乐观情况下推荐使用，即锁竞争不强，锁等待时间不长的情况下推荐使用
2. 单cpu无效，因为基于cas的轮询会占用cpu, 导致无法做线程切换
3. 轮询不产生上下文切换，如果可估计到睡眠的时间很长，用互斥锁更好

2018-06-19

| 作者回复

不错

2018-06-19



灰飞灰猪不会灰飞.烟灭

👍 3

老师 AQS就不涉及用户态和内核态的切换了 对吧？

2018-06-12

| 作者回复

我理解是，cas是基于特定指令

2018-06-12



刘杰

👍 2

偏斜锁和轻量级锁的区别不是很清晰

2018-07-12



Miaoze

👍 2

杨老师，看到有回复说自旋锁在单核CPU上是无用，感觉这个理论不准确，因为Java多线程在很早时候单核CPC的PC上就能运行，计算机原理中也介绍，控制器会轮巡各个进程或线程。而且多线程是运行在JVM上，跟物理机没有很直接的关系吧？

2018-06-13

| 作者回复

已回复，我也认为单核无用

2018-06-13



(亮亮)

👍 1

自旋锁失败后，不是进入同步等待队列吗？

2018-08-22



gesanri

👍 1

我有一个疑问，最后这个stampedlock的例子，access方法中，调用读乐观锁之后直接就进行read操作，但这个时候不知道validate的结果，如果validate为false，还要再read一次，为什么不先判断validate为true再read呢？是因为read这个操作太轻量级了吗？

2018-08-04



苍天大树

👍 1

第一次回答问题哈，自旋锁就是当获取锁失败后，自己定时循环去获取锁，不进入休眠状态。这样的好处就是快，坏处就是消耗cpu

2018-08-04



齐帆

👍 1

老师后面会详细讲 AQS 吗

2018-06-12

| 作者回复

有的

2018-06-12



肖一林

👍 1

重量级锁还是互斥锁吗？自旋锁应该是线程拿不到锁的时候，采取重试的办法，适合重试次数不多的场景，如果重试次数过多还是会被系统挂起，这种情况下还不如没有自旋锁。

2018-06-12

| 作者回复

是的

2018-06-12



大熊

👍 0

老师，请问下为什么要有读锁？读不会改变数据为什么还要加锁呢

2018-10-16

| 作者回复

针对不同场景，例如，并发读比写多，比较适合readwritelock；readlock不是排他的（exclusive），保证看到的data是更新过的

2018-10-24



stephen chow

👍 0

StampLock是先试着读吧？你写的先试着修改。。

2018-10-01

| 作者回复

嗯，是有点写跑偏了，看上下文倒也能理解，谢谢指出

2018-10-24



一个坏人

👍 0

老师好，请教一下：“自动内存管理系统为什么要求对象的大小必须是8字节的整数倍？”，即内存对齐的根本原因在于？

2018-08-25

| 作者回复

这个...还真没想到，“对齐”就是原因吧，更高效的访问，处理器操作方便

2018-09-04



小飞哥 超級會員

0

太底层了，在应用过程中如何解决这些种类的锁呢？

2018-08-18



July

0

这一节看的好晕 得反复多看几遍

2018-08-16



四阿哥

0

这里说的普通轻量级锁，没明白具体是什么

2018-08-14



clz1341521

0

自旋锁是一种乐观优化

自旋锁:竞争锁的失败的线程，并不会真实的在操作系统层面挂起等待，而是JVM会让线程做几个空循环(基于预测在不久的将来就能获得)，在经过若干次循环后，如果可以获得锁，那么进入临界区，如果还不能获得锁，才会真实的将线程在操作系统层面进行挂起。

适用场景:自旋锁可以减少线程的阻塞，这对于锁竞争不激烈，且占用锁时间非常短的代码块来说，有较大的性能提升，因为自旋的消耗会小于线程阻塞挂起操作的消耗。

2018-08-05