



Java性能调优实战  
刘超  
金山软件西山居技术经理

查看详情

5880 人已学习

## 模块二 · 多线程性能调优 (10讲)

- 12 | 多线程之锁优化（上）：深入了解 Synchronized 同步锁的优化方法
- 13 | 多线程之锁优化（中）：深入了解 Lock 同步锁的优化方法
- 14 | 多线程之锁优化（下）：使用乐观锁优化并行操作
- 15 | 多线程调优（上）：哪些操作导致了上下文切换？
- 16 | 多线程调优（下）：如何优化多线程上下文切换？
- 17 | 并发容器的使用：识别不同场景下

# 12 | 多线程之锁优化（上）：深入了解 Synchronized 同步锁的优化方法

刘超 2019-06-15



00:00

讲述：李良 大小：12.10M

13:12

你好，我是刘超。从这讲开始，我们就正式进入到第三模块——多线程性能调优。

在并发编程中，多个线程访问同一个共享资源时，我们必须考虑如何维护数据的原子性。在 JDK1.5 之前，Java 是依靠 Synchronized 关键字实现锁功能来做到这点的。Synchronized 是 JVM 实现的一种内置锁，锁的获取和释放是由 JVM 隐式实现。

到了 JDK1.5 版本，并发包中新增了 Lock 接口来实现锁功能，它提供了与 Synchronized 关键字类似的同步功能，只是在使用时需要显示获取和释放锁。

Lock 同步锁是基于 Java 实现的，而 Synchronized 是基于底层操作系统的 Mutex Lock 实现的，每次获取和释放锁操作都会带来用户态和内核态的切换，从而增加系统性能开销。因此，在锁竞争激烈的情况下，Synchronized 同步锁在性能上就表现得非常糟糕，它也常被大家称为重量级锁。

特别是在单个线程重复申请锁的情况下，JDK1.5 版本的 Synchronized 锁性能要比 Lock 的性能差很多。例如，在 Dubbo 基于 Netty 实现的通信中，消费端向服务端通信之后，由于接收返回消息是异步，所以需要一个线程轮询监听返回信息。而在接收消息时，就需要用到锁来确保 request session 的原子性。如果我们这里使用 Synchronized 同步锁，那么每当同一个线程请求锁资源时，都会发生一次用户态和内核态的切换。

到了 JDK1.6 版本之后，Java 对 Synchronized 同步锁做了充分的优化，甚至在某些场景下，它的性能已经超越了 Lock 同步锁。这一讲我们就来看看 Synchronized 同步锁究竟是通过了哪些优化，实现了性能地提升。

## Synchronized 同步锁实现原理

了解 Synchronized 同步锁优化之前，我们先来看看它的底层实现原理，这样可以帮助我们更好地理解后面的内容。

通常 Synchronized 实现同步锁的方式有两种，一种是修饰方法，一种是修饰方法块。以下就是通过 Synchronized 实现的两种同步方法加锁的方式：

```
2     public synchronized void method1() {
3         // code
4     }
5
6     // 关键字在代码块上，锁为括号里面的对象
7     public void method2() {
8         Object o = new Object();
9         synchronized (o) {
10             // code
11         }
12     }
13
```

下面我们可以看下具体字节码的实现，运行以下反编译命令，就可以输出我们想要的字节码：

```
1 javac -encoding UTF-8 SyncTest.java // 先运行编译 class 文件命令
2
```

```
1 javap -v SyncTest.class // 再通过 javap 打印出字节文件
2
```

通过输出的字节码，你会发现：Synchronized 在修饰同步代码块时，是由 monitoreenter 和 monitorexit 指令来实现同步的。进入 monitoreenter 指令后，线程将持有 Monitor 对象，退出 monitoreenter 指令后，线程将释放该 Monitor 对象。

```
1 public void method2();
2     descriptor: ()V
3     flags: ACC_PUBLIC
4     Code:
5         stack=2, locals=4, args_size=1
6             0: new           #2
7             3: dup
8             4: invokespecial #1
9             7: astore_1
10            8: aload_1
11            9: dup
12            10: astore_2
13            11: monitoreenter //monitoreenter 指令
14            12: aload_2
15            13: monitorexit //monitorexit 指令
16            14: goto           22
17            17: astore_3
18            18: aload_2
19            19: monitorexit
20            20: aload_3
21            21: athrow
22            22: return
23     Exception table:
24         from   to target type
25             12    14    17  any
26             17    20    17  any
27     LineNumberTable:
28         line 18: 0
29         line 19: 8
30         line 21: 12
31         line 22: 22
32     StackMapTable: number_of_entries = 2
33         frame_type = 255 /* full_frame */
34         offset_delta = 17
35         locals = [ class com/demo/io/SyncTest, class java/lang/Object, class j
36         stack = [ class java/lang/Throwable ]
37         frame_type = 250 /* chop */
38         offset_delta = 4
39
```

再来看以下同步方法的字节码，你会发现：当 Synchronized 修饰同步方法时，并没有发现 monitoreenter 和 monitorexit 指令，而是出现了一个 ACC\_SYNCHRONIZED 标志。

这是因为 JVM 使用了 ACC\_SYNCHRONIZED 访问标志来区分一个方法是否是同步方法。当方法调用时，调用指令将会检查该方法是否被设置 ACC\_SYNCHRONIZED 访问标志。如果设置了该标志，执行线程将先持有 Monitor 对象，然后再执行方法。在该方法运行期间，其它线程将无法获取到该 Monitor 对象，当方法执行完成后，再释放该 Monitor 对象。

```
1 public synchronized void method1();
2     descriptor: ()V
3     flags: ACC_PUBLIC, ACC_SYNCHRONIZED // ACC_SYNCHRONIZED 标志
4     Code:
5         stack=0, locals=1, args_size=1
6             0: return
7             LineNumberTable:
8                 line 8: 0
9
10
```

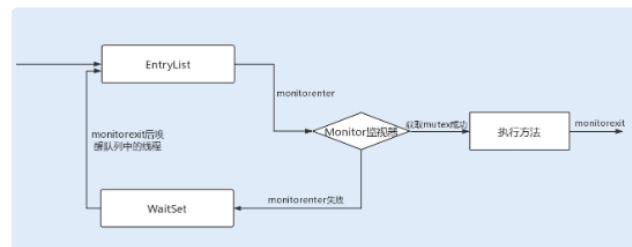
通过以上的源码，我们再来看看 Synchronized 修饰方法是怎么实现锁原理的。

JVM 中的同步是基于进入和退出管程（Monitor）对象实现的。每个对象实例都会有一个 Monitor，Monitor 可以和对象一起创建、销毁。Monitor 是由 ObjectMonitor 实现，而 ObjectMonitor 是由 C++ 的 ObjectMonitor.hpp 文件实现，如下所示：

```
1 ObjectMonitor() {
2     _header = NULL;
3     _count = 0; // 记录个数
4     _waiters = 0;
5     _recursions = 0;
6     _object = NULL;
7     _owner = NULL;
8     _WaitSet = NULL; // 处于 wait 状态的线程，会被加入到 _WaitSet
9     _WaitSetLock = 0 ;
10    _Responsible = NULL ;
11    _succ = NULL ;
12    _cxq = NULL ;
13    FreeNext = NULL ;
14    _EntryList = NULL ; // 处于等待锁 block 状态的线程，会被加入到该列表
15    _SpinFreq = 0 ;
16    _SpinClock = 0 ;
17    OwnerIsThread = 0 ;
18 }
19
```

当多个线程同时访问一段同步代码时，多个线程会先被存放在 EntryList 集合中，处于 block 状态的线程，都会被加入到该列表。接下来当线程获取到对象的 Monitor 时，Monitor 是依靠底层操作系统的 Mutex Lock 来实现互斥的，线程申请 Mutex 成功，则持有该 Mutex，其它线程将无法获取到该 Mutex。

如果线程调用 wait() 方法，就会释放当前持有的 Mutex，并且该线程会进入 WaitSet 集合中，等待下一次被唤醒。如果当前线程顺利执行完方法，也将释放 Mutex。



看完上面的讲解，相信你对同步锁的实现原理已经有个深入的了解了。总结来说就是，同步锁在这种实现方式中，因 Monitor 是依赖于底层的操作系统实现，存在用户态与内核态之间的切换，所以增加了性能开销。

## 锁升级优化

为了提升性能，JDK1.6 引入了偏向锁、轻量级锁、重量级锁概念，来减少锁竞争带来的上下文

切换，而正是新增的 Java 对象头实现了锁升级功能。

当 Java 对象被 Synchronized 关键字修饰成为同步锁后，围绕这个锁的一系列升级操作都将和 Java 对象头有关。

## Java 对象头

在 JDK1.6 JVM 中，对象实例在堆内存中被分为了三个部分：对象头、实例数据和对齐填充。其中 Java 对象头由 Mark Word、指向类的指针以及数组长度三部分组成。

Mark Word 记录了对象和锁有关的信息。Mark Word 在 64 位 JVM 中的长度是 64bit，我们可以一起看下 64 位 JVM 的存储结构是怎么样的。如下图所示：

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

锁升级功能主要依赖于 Mark Word 中的锁标志位和释放偏向锁标志位，**Synchronized 同步锁就是从偏向锁开始的，随着竞争越来越激烈，偏向锁升级到轻量级锁，最终升级到重量级锁**。下面我们就沿着这条优化路径去看下具体的内容。

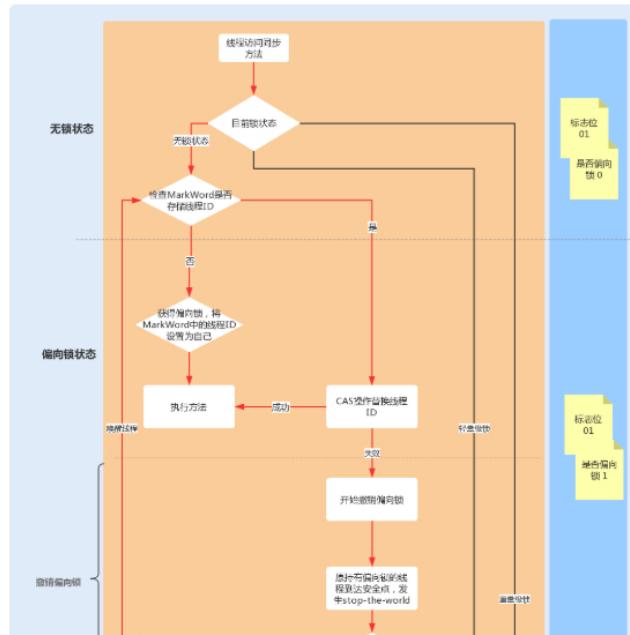
### 1. 偏向锁

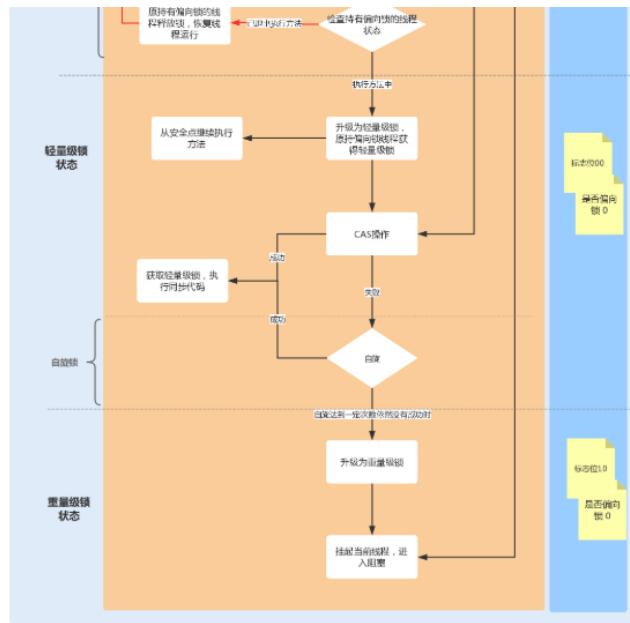
偏向锁主要用来优化同一线程多次申请同一个锁的竞争。在某些情况下，大部分时间是同一个线程竞争锁资源，例如，在创建一个线程并在线程中执行循环监听的场景下，或单线程操作一个线程安全集合时，同一线程每次都需要获取和释放锁，每次操作都会发生用户态与内核态的切换。

偏向锁的作用就是，当一个线程再次访问这个同步代码或方法时，该线程只需去对象头的 Mark Word 中去判断一下是否有偏向锁指向它的 ID，无需再进入 Monitor 去竞争对象了。**当对象被当做同步锁并有一个线程抢到了锁时，锁标志位还是 01，“是否偏向锁”标志位设置为 1，并且记录抢到锁的线程 ID，表示进入偏向锁状态。**

一旦出现其它线程竞争锁资源时，偏向锁就会被撤销。偏向锁的撤销需要等待全局安全点，暂停持有该锁的线程，同时检查该线程是否还在执行该方法，如果是，则升级锁，反之则被其它线程抢占。

下图中红线流程部分为偏向锁获取和撤销流程：





因此，在高并发场景下，当大量线程同时竞争同一个锁资源时，偏向锁就会被撤销，发生 stop the word 后，开启偏向锁无疑会带来更大的性能开销，这时我们可以通过添加 JVM 参数关闭偏向锁来调优系统性能，示例代码如下：

```
1 -XX:-UseBiasedLocking // 关闭偏向锁（默认打开）
2
```

[复制代码](#)

或

```
1 -XX:+UseHeavyMonitors // 设置重量级锁
2
```

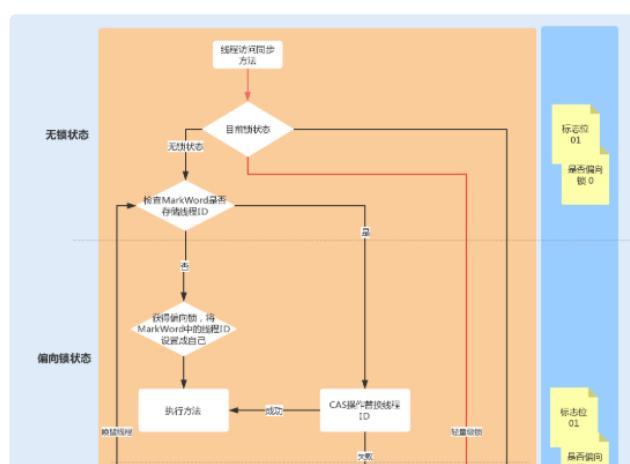
[复制代码](#)

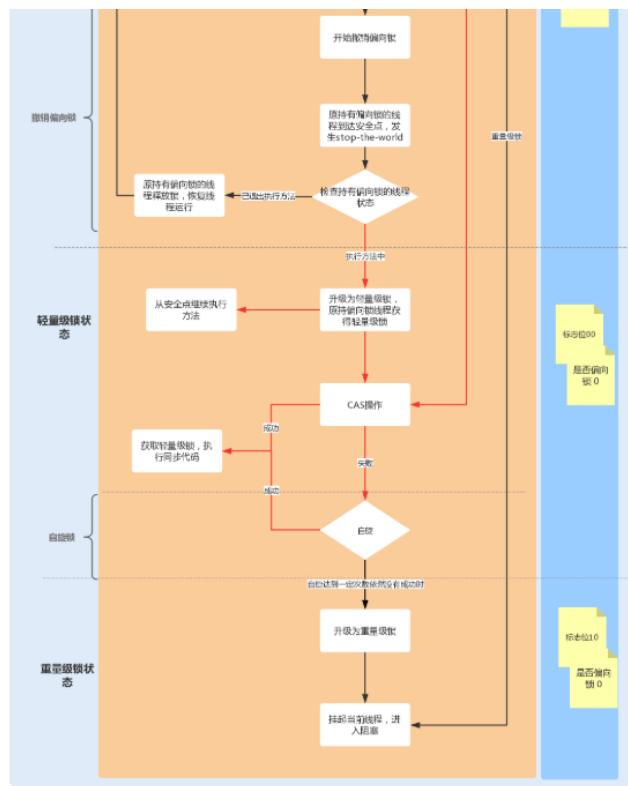
## 2. 轻量级锁

当有另外一个线程竞争获取这个锁时，由于该锁已经是偏向锁，当发现对象头 Mark Word 中的线程 ID 不是自己的线程 ID，就会进行 CAS 操作获取锁，如果获取成功，直接替换 Mark Word 中的线程 ID 为自己的 ID，该锁会保持偏向锁状态；如果获取锁失败，代表当前锁有一定的竞争，偏向锁将升级为重量级锁。

轻量级锁适用于线程交替执行同步块的场景，绝大部分的锁在整个同步周期内都不存在长时间的竞争。

下图中红线部分为升级轻量级锁及操作流程：





### 3. 自旋锁与重量级锁

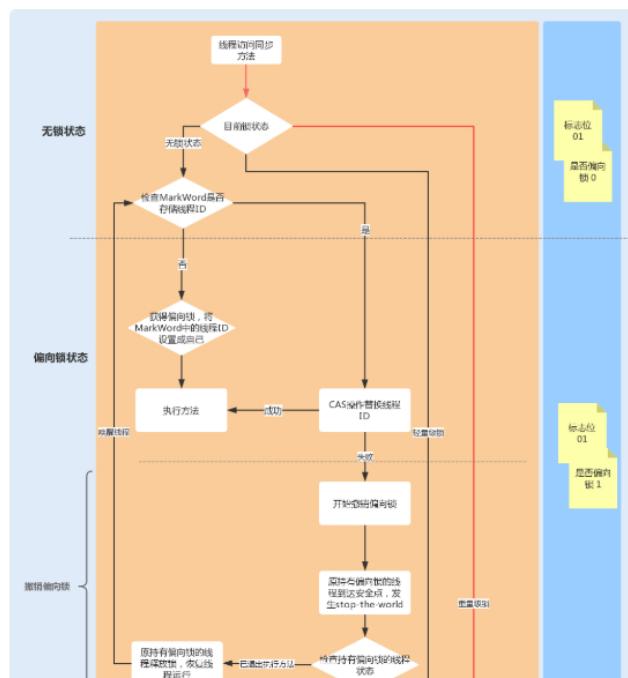
轻量级锁 CAS 抢锁失败，线程将会被挂起进入阻塞状态。如果正在持有锁的线程在很短的时间内释放资源，那么进入阻塞状态的线程无疑又要申请锁资源。

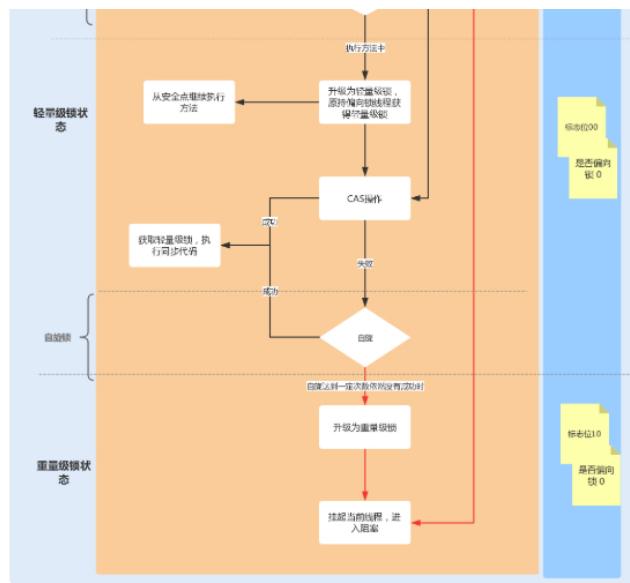
JVM 提供了一种自旋锁，可以通过自旋方式不断尝试获取锁，从而避免线程被挂起阻塞。这是基于大多数情况下，线程持有锁的时间都不会太长，毕竟线程被挂起阻塞可能会得不偿失。

从 JDK1.7 开始，自旋锁默认启用，自旋次数由 JVM 设置决定，这里我不建议设置的重试次数过多，因为 CAS 重试操作意味着长时间地占用 CPU。

自旋锁重试之后如果抢锁依然失败，同步锁就会升级至重量级锁，锁标志位改为 10。在这个状态下，未抢到锁的线程都会进入 Monitor，之后会被阻塞在 \_WaitSet 队列中。

下图中红线流程部分为自旋后升级为重量级锁的流程：





在锁竞争不激烈且锁占用时间非常短的场景下，**自旋锁可以提高系统性能**。一旦锁竞争激烈或锁占用的时间过长，自旋锁将会导致大量的线程一直处于 CAS 重试状态，占用 CPU 资源，反而会增加系统性能开销。所以自旋锁和重量级锁的使用都要结合实际场景。

在高负载、高并发的场景下，我们可以通过设置 JVM 参数来关闭自旋锁，优化系统性能，示例代码如下：

```

1 -XX:-UseSpinning // 参数关闭自旋锁优化（默认打开）
2 -XX:PreBlockSpin // 参数修改默认的自旋次数。JDK1.7 后，去掉此参数，由 jvm 控制
3

```

## 动态编译实现锁消除 / 锁粗化

除了锁升级优化，Java 还使用了编译器对锁进行优化。JIT 编译器在动态编译同步块的时候，借助了一种被称为逃逸分析的技术，来判断同步块使用的锁对象是否只能够被一个线程访问，而没有被发布到其它线程。

确认是的话，那么 JIT 编译器在编译这个同步块的时候不会生成 synchronized 所表示的锁的申请与释放的机器码，即消除了锁的使用。在 Java7 之后的版本就不需要手动配置了，该操作可以自动实现。

锁粗化同理，就是在 JIT 编译器动态编译时，如果发现几个相邻的同步块使用的是同一个锁实例，那么 JIT 编译器将会把这几个同步块合并为一个大的同步块，从而避免一个线程“反复申请、释放同一个锁”所带来的性能开销。

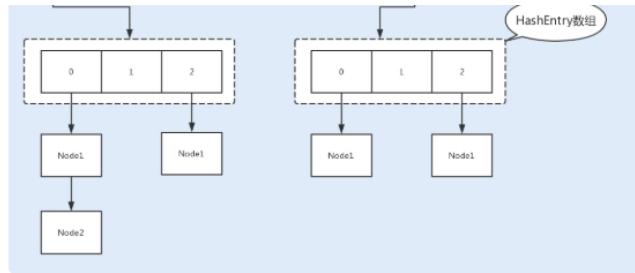
## 减小锁粒度

除了锁内部优化和编译器优化之外，我们还可以通过代码层来实现锁优化，减小锁粒度就是一种惯用的方法。

当我们的锁对象是一个数组或队列时，集中竞争一个对象的话会非常激烈，锁也会升级为重量级锁。**我们可以考虑将一个数组和队列对象拆成多个小对象，来降低锁竞争，提升并行度。**

最经典的减小锁粒度的案例就是 JDK1.8 之前实现的 ConcurrentHashMap 版本。我们知道，HashTable 是基于一个数组 + 链表实现的，所以在并发读写操作集合时，存在激烈的锁资源竞争，也因此性能会存在瓶颈。而 ConcurrentHashMap 就很巧妙地使用了分段锁 Segment 来降低锁资源竞争，如下图所示：





## 总结

JVM 在 JDK1.6 中引入了分级锁机制来优化 Synchronized，当一个线程获取锁时，首先对象锁将成为一个偏向锁，这样做是为了优化同一线程重复获取导致的用户态与内核态的切换问题；其次如果有多个线程竞争锁资源，锁将会升级为轻量级锁，它适用于在短时间内持有锁，且分锁有交替切换的场景；轻量级锁还使用了自旋锁来避免线程用户态与内核态的频繁切换，大大地提高了系统性能；但如果锁竞争太激烈了，那么同步锁将会升级为重量级锁。

**减少锁竞争，是优化 Synchronized 同步锁的关键。**我们应该尽量使 Synchronized 同步锁处于轻量级锁或偏向锁，这样才能提高 Synchronized 同步锁的性能；通过减小锁粒度来降低锁竞争也是一种最常用的优化方法；另外我们还可以通过减少锁的持有时间来提高 Synchronized 同步锁在自旋时获取锁资源的成功率，避免 Synchronized 同步锁升级为重量级锁。

这一讲我们重点了解了 Synchronized 同步锁优化，这里由于字数限制，也为了你能更好地理解内容，目录中 12 讲的内容我拆成了两讲，在下一讲中，我会重点讲解 Lock 同步锁的优化方法。

## 思考题

请问以下 Synchronized 同步锁对普通方法和静态方法的修饰有什么区别？

```

1 // 修饰普通方法
2     public synchronized void method1() {
3         // code
4     }
5
6     // 修饰静态方法
7     public synchronized static void method2() {
8         // code
9     }
10

```

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。

极客时间

# Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超  
金山软件西山居技术经理

新版升级：点击「请朋友读」，20位好友免费读，邀请订阅更有现金奖励。

胡晓

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Command + Enter 发表

0/2000字

提交留言

精选留言(41)

bro.

Synchronized锁升级步骤

1. 偏向锁:JDK6中引入的一项锁优化,它的目的是消除数据在无竞争情况下的同步原语,进一步提高程序的运行性能 ,
2. 偏向锁会偏向于第一个获得它的线程,如果在接下来的执行过程中,该锁没有被其他的线程获取,则持有偏向锁的线程将永远不需要同步。大多数情况下,锁不仅不存在多线程竞争,而且总是由同一线程多次获得,为了让线程获得锁的代价更低而引入了偏向锁
3. 当锁对象第一次被线程获取的时候,线程使用CAS操作把这个锁的线程ID记录再对象Mark Word之中,同时置偏向标志位1,以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁,只需要简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功,表示线程已经获得了锁。
4. 如果线程使用CAS操作时失败则表示该锁对象上存在竞争并且这个时候另外一个线程获得偏向锁的所有权。当到达全局安全点 (safepoint, 这个时间点上没有正在执行的字节码) 时获得偏向锁的线程被挂起,膨胀为轻量级锁 (涉及Monitor Record, Lock Record相关操作, 这里不展开), 同时被撤销偏向锁的线程继续往下执行同步代码。
5. 当有另外一个线程去尝试获取这个锁时, 偏向模式就宣告结束
6. 线程在执行同步块之前, JVM会先在当前线程的栈帧中创建用于存储锁记录(Lock Record)的空间,并将对象头中的Mark Word复制到锁记录中,官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功,当前线程获得锁,如果失败,表示其他线程竞争锁,当前线程便尝试使用自旋来获取锁。如果自旋失败则锁会膨胀为重量级锁。如果自旋成功则依然处于轻量级锁的状态
7. 轻量级锁的解锁过程也是通过CAS操作来进行的,如果对象的Mark Word仍然指向线程的锁记录,那就用CAS操作把对象当前的Mark Word和线程中赋值的Displaced Mark Word替换回来,如果替换成功,整个同步过程就完成了,如果替换失败,就说明有其他线程尝试过获取该锁,那就要在释放锁的同时,唤醒被挂起的线程
8. 轻量级锁提升程序同步性能的依据是:对于绝大部分的锁,在整个同步周期内都是不存在竞争的(区别于偏向锁)。这是一个经验数据。如果没有竞争,轻量级锁使用CAS操作避免了使用互斥量的开销,但如果存在锁竞争,除了互斥量的开销外,还额外发生了CAS操作,因此在有竞争的情况下,轻量级锁比传统的重量级锁更慢

简单概括为:

1. 检测Mark Word里面是不是当前线程ID,如果是,表示当前线程处于偏向锁
2. 如果不是,则使用CAS将当前线程ID替换到Mark Word,如果成功则表示当前线程获得偏向锁,设置偏向标志位1
3. 如果失败,则说明发生了竞争,撤销偏向锁,升级为轻量级锁
4. 当前线程使用CAS将对象头的mark Word锁标记位替换为锁记录指针,如果成功,当前线程获得锁
5. 如果失败,表示其他线程竞争锁,当前线程尝试通过自旋获取锁 for(;;)
6. 如果自旋成功则依然处于轻量级状态
7. 如果自旋失败,升级为重量级锁

- 索指针:在当前线程的栈帧中划出一块空间,作为该锁的锁记录,并且将锁对象的标记字段复制到改锁记录中!

作者回复: 赞

2019-06-18

1 7

-W.LI-

老师好!获取偏斜锁和轻量级锁的时候使用的CAS操作预期值传的是null(希望锁已释放), 替换后值是当前线程什么?

作者回复: 老师没有看懂你问的具体问题,麻烦再描述一下你的问题。

2019-06-18

3



晓杰

感觉讲得有点晦涩啊，不知道其他人什么感觉

作者回复: 如果哪里不懂的, 可以多提问, 希望我能帮助到你。

2019-06-16

...  
3



苏志辉

entrylist和waitset那个地方不太理解, monitorenter失败后会进入entrylist吧, 只有调用wait方法才会进入waitset吧, 还请老师指点下

作者回复: 在获取到参与锁资源竞争的线程会进入entrylist, 线程monitorenter失败后会进入到waitset, 此时说明已经有线程获取到锁了, 所以需要进入等待。调用wait方法也会进入到waitset。

2019-06-16

...  
3



nightmare

加在普通方法锁对象是当前对象, 其ObjectMonitor就是对象的, 而静态方法上, 锁对象就是字节码对象, 静态方法是所有对象共享的, 锁粒度比较大

2019-06-15

...  
3



Jxin

1.8后, 可以尽量采用并发包中的无锁或则称乐观锁来实现。读写极端场景可以看情况选用读写锁或票据锁。

课后题, 前者锁实例, 后者锁类的字节码对象。后者力度太大应该结合业务场景尽量规避。

2019-06-15

...  
2



陆离

非静态方法是对象锁, 静态方法是类锁

2019-06-15

...  
2



任鹏斌

普通方法中锁的是当前对象, 静态方法锁的是静态类

作者回复: 对的, 普通方法中的锁时锁对象, 而修饰静态方法是类锁

2019-07-03

...  
1



不靠谱~

1.课后作业: 实际对象锁和类对象锁的区别, 锁对象不一样。  
2.1.8后CurrentHashMap已经不用segment策略了, 想请教一下老师1.8后是怎样保证性能的呢?  
3.对锁升级不太了解的同学可以看一下《Java并发编程的艺术》。里面有很详细的介绍, 不过也是比较难理解, 多看几遍。

作者回复: JDK1.8之后ConcurrentHashMap就放弃了分段锁策略, 而是直接使用CAS+Synchronized方式保证性能, 这里的锁是指锁table的首个Node节点。在添加数据的时候, 如果Node数组没有值的情况, 则会使用CAS添加数据, CAS成功则添加成功, 失败则进入锁代码块执行插入链表或红黑树或转红黑树操作。

2019-06-20

...  
1



bro.

修饰普通方法是改类的对象,比如class A 创建了两个对象 class A1 跟class A2,对于method1来说A1,A2直接不是互斥的,但是对于静态方法或者synchronized(A.class)表示加锁对象为.class文件,一个项目只有唯一一个class文件,所以是互斥的

2019-06-18

··· 1



黑崽

锁升级的图中显示markword是否存储线程ID的图中, 两个路径是与不是, 是不是画反了? 是的话, 要as替换成本己。不是, 那么就直接获取偏向锁

作者回复: 是的, 感谢黑崽同学的提醒。

2019-06-17

··· 1



chris~jiang

老师, 您好, synchronized锁只会升级, 不会降级吧? 如果系统只在某段时间高并发, 升级到了重量级锁, 然后系统变成低并发了, 就一直是重量级锁了吗? 请老师解惑, 谢谢⚠

2019-07-02

···



拉可里啦

在高并发的场景下, 只能使用重量锁了吧 因为锁最终会升级到重量级锁, 那么就需要提前关闭偏向锁和自旋锁

作者回复: 对的

2019-06-30

···



拉可里啦

老师你好, 在偏向锁中, 如果另一个线程通过cas获取到了偏向锁, 那么之前获取到的偏向锁的线程还在执行中, 是否被中断了, 还是执行完

作者回复: 这个时候是无法获取到偏向锁的

2019-06-29

···



拉可里啦

在使用偏向锁中, 进行cas替换失败的原因是什么? 我认为有两点: 1.被替换的线程目前还在执行中 2.被替换的线程已执行完毕, 但是有其它线程同时获取到了偏向锁。不知我理解的是否合理, 还请老师指点。

作者回复: 对的, 理解正确

2019-06-28

···



拉可里啦

有个偏向锁的疑问: 偏向锁认为自始至终只有一个线程访问, 那么是怎么确定请求过来的是同一个线程呢? 因为每次请求线程都是不同的线程ID啊

作者回复: 同一个线程, 线程ID是一样的, 是根据线程ID区别的。



2019-06-28



Geek\_ebda96

因此，在高并发场景下，当大量线程同时竞争同一个锁资源时，偏向锁就会被撤销，发生 stop the word 后，开启偏向锁无疑会带来更大的性能开销。老师，这句话的意思是说撤销偏向锁的过程，发生stop the word非常耗时影响性能吗？偏向锁不适合高并发场景，但低并发场景，用偏向锁的意义在哪里呢？低并发关闭偏向锁，直接获取轻量级的锁，这个也没啥问题吧。

作者回复：是的，理解没问题。偏向锁适合当个永久线程下或有嵌套锁的资源锁操作。

2019-06-26



Better me

老师能否解释下CurrentHashMap分段锁的那张图，不是很了解分段锁的实现

作者回复：最开始CurrentHashMap只有一个table[]数组，如果要操作该CurrentHashMap，则需要锁住table[]对象，这个时候所有的操作都来竞争一个table[]对象锁。

而分段锁则是将table[]数组分解称为了一个Segment[]数组，每个Segment[]的元素里面包含了  
一个HashEntry<K,V>[]数组，在第一次写入Segment[]数组时是CAS操作，如果CAS失败或  
Segment[]数组中的值不为null，则通过tryLock()将该Segment[]的这个元素锁住。而不是锁住整个Segment[]数组。

2019-06-24



VIC

40

怎么让synchronized使用偏向锁，轻量级锁呢？

作者回复：JVM默认情况下会使用偏向锁和轻量级锁，只有在竞争锁资源非常激烈的情况下，才会升级到重量级锁。

所以我们可以减少锁竞争来保持synchronized使用偏向锁，轻量级锁。

2019-06-24



左瞳

还没看完，看到了图我反手一个赞

2019-06-23



汤小高

王一 谢赫 大家好 一直关注着大神的专栏，学习了很多知识，感谢大神的分享！