

Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）

Java并发编程系列：

- Java 并发编程：核心理论
- Java并发编程：Synchronized及其实现原理
- Java并发编程：Synchronized底层优化（轻量级锁、偏向锁）
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join).
- Java 并发编程：volatile的使用及其原理

一、重量级锁

上篇文章中向大家介绍了Synchronized的用法及其实现的原理。现在我们应该知道，Synchronized是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就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”。JDK中对Synchronized做的种种优化，其核心都是为了减少这种重量级锁的使用。JDK1.6以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

二、轻量级锁

锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。JDK 1.6中默认是开启偏向锁和轻量级锁的，我们也可以通过-XX:-UseBiasedLocking来禁用偏向锁。锁的状态保存在对象的头文件中，以32位的JDK为例：

锁状态	25 bit		4bit	1bit	2bit	
	23bit	2bit		是否是偏向锁	锁标志位	
轻量级锁	指向栈中锁记录的指针				00	
重量级锁	指向互斥量（重量级锁）的指针				10	
GC标记	空				11	
偏向锁	线程ID	Epoch		对象分代年龄	1	01
无锁	对象的hashCode			对象分代年龄	0	01

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

1、轻量级锁的加锁过程

（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示。

（2）拷贝对象头中的Mark Word复制到锁记录中。

（3）拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（3），否则执行步骤（4）。

（4）如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图2.2所示。

（5）如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。

公告

昵称：liuxiaopeng
园龄：2年6个月
粉丝：249
关注：2
+加关注

搜索

找找看

谷歌搜索

最新随笔

1. Spring Boot实战：拦截器与过滤器

2. Spring Boot实战：静态资源处理

3. Spring Boot实战：集成Swagger2

4. Spring Boot实战：Restful API的构建

5. Spring Boot实战：数据库操作

6. Spring Boot实战：逐行释义HelloWorld

7. Java集合类：AbstractCollection源码解析

8. Java集合：整体结构

9. Java 并发编程：volatile的使用及其原理

10. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

我的标签

Java(15)

Spring(7)

spring boot(7)

并发编程(4)

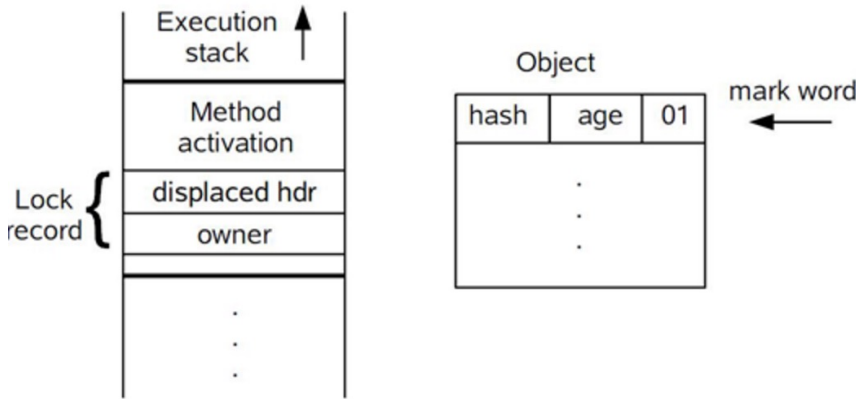


图2.1 轻量级锁CAS操作之前堆栈与对象的状态

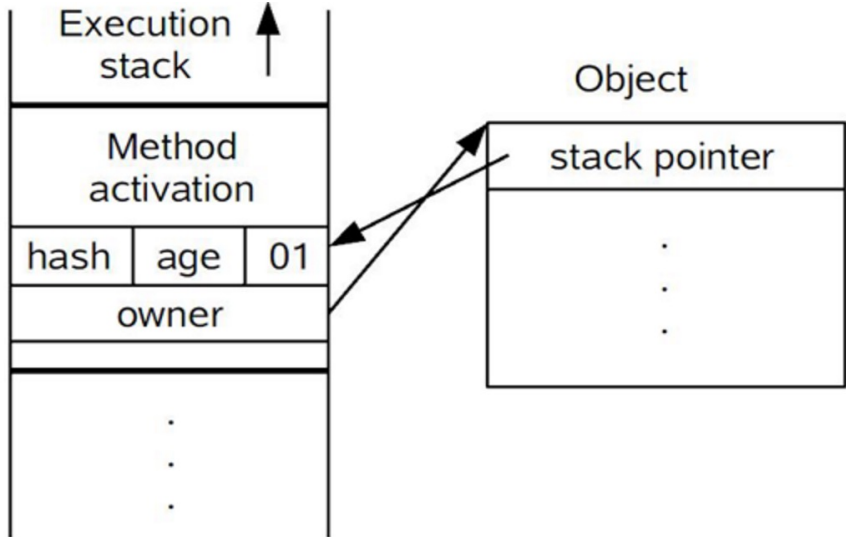


图2.2 轻量级锁CAS操作之后堆栈与对象的状态

2、轻量级锁的解锁过程：

- （1）通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word。
- （2）如果替换成功，整个同步过程就完成了。
- （3）如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

三、偏向锁

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

1、偏向锁获取过程：

- （1）访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——确认为可偏向状态。
- （2）如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤（5），否则进入步骤（3）。
- （3）如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。
- （4）如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。
- （5）执行同步代码。

2、偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

3、重量级锁、轻量级锁和偏向锁之间转换

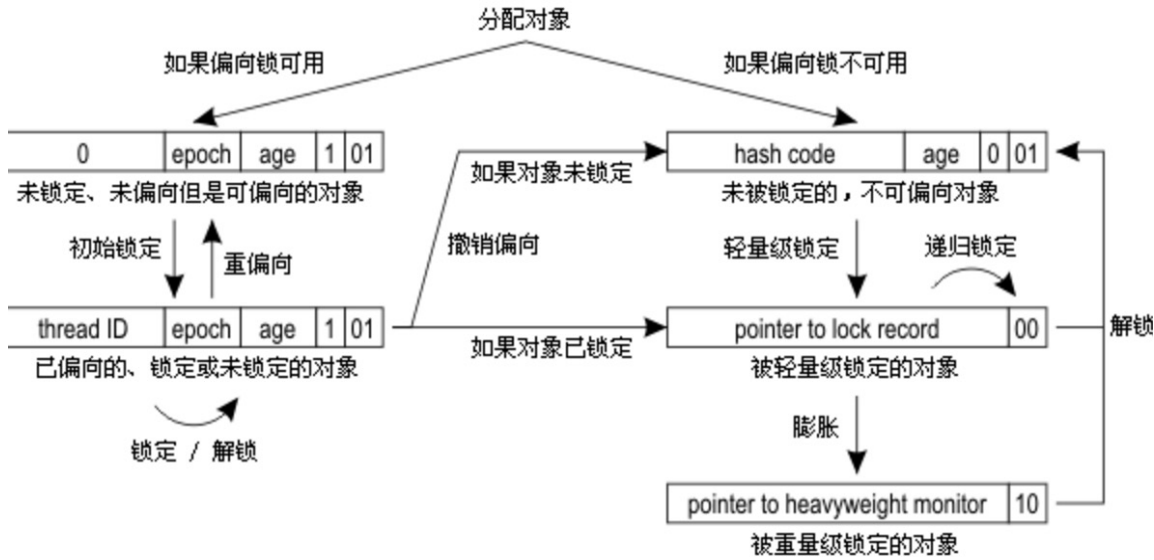


图 2.3三者的转换图

该图主要是对上述内容的总结，如果对上述内容有较好的了解的话，该图应该很容易看懂。

四、其他优化

Rest(2)
Restful(1)
过滤器(1)
集合框架(1)
静态资源(1)
拦截器(1)
更多

随笔档案
2018年1月 (5)
2017年12月 (1)
2016年6月 (1)
2016年5月 (3)
2016年4月 (4)
2016年3月 (3)

积分与排名
积分 - 40237
排名 - 9732

最新评论
1. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）
博主写得很棒
--还好可以改名字

2. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）
@就这个名引用在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。针对这句话有个疑问，如果我能保.....
--还好可以改名字
3. Re:Java8内存模型—永久代(PermGen)和元空间(Metaspace)
感谢分享，写的很好！
--我不将就


4. Re:Spring Boot实战：集成Swagger 2
学习了 菜鸟飘过~~~
--祎祎家斌斌

1、**适应性自旋（Adaptive Spinning）**：从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

2、**锁粗化（Lock Coarsening）**：锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。举个例子：




```
1 package com.paddx.test.string;
2
3 public class StringBufferTest {
4     StringBuffer stringBuffer = new StringBuffer();
5
6     public void append(){
7         stringBuffer.append("a");
8         stringBuffer.append("b");
9         stringBuffer.append("c");
10    }
11 }
```




这里每次调用stringBuffer.append方法都需要加锁和解锁，如果虚拟机检测到有一系列连串的对同一个对象加锁和解锁操作，就会将其合并成一次范围更大的加锁和解锁操作，即在第一次append方法时进行加锁，最后一次append方法结束后进行解锁。

3、**锁消除（Lock Elimination）**：锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。看下面这段程序：



```
1 package com.paddx.test.concurrent;
2
3 public class SynchronizedTest02 {
4
5     public static void main(String[] args) {
6         SynchronizedTest02 test02 = new SynchronizedTest02();
7         //启动预热
8         for (int i = 0; i < 10000; i++) {
9             i++;
10        }
11        long start = System.currentTimeMillis();
12        for (int i = 0; i < 100000000; i++) {
13            test02.append("abc", "def");
14        }
15        System.out.println("Time=" + (System.currentTimeMillis() - start));
16    }
17
18    public void append(String str1, String str2) {
19        StringBuffer sb = new StringBuffer();
20        sb.append(str1).append(str2);
21    }
22 }
```



虽然StringBuffer的append是一个同步方法，但是这段程序中的StringBuffer属于一个局部变量，并且不会从该方法中逃逸出去，所以其实这过程是线程安全的，可以将锁消除。下面是我本地执行的结果：

```
ogon:classes liuxp$
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=5680
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=5754
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=5636
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=5729
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=5523
ogon:classes liuxp$
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=2482
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=2379
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=2500
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=2502
ogon:classes liuxp$ java -server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -XX:-UseBiasedLocking com.paddx.test.concurrent.SynchronizedTest02
Time=2563
```

为了尽量减少其他因素的影响，这里禁用了偏向锁（-XX:-UseBiasedLocking）。通过上面程序，可以看出消除锁以后性能还是有比较大提升的。

注：可能JDK各个版本之间执行的结果不尽相同，我这里采用的JDK版本为1.6。

五、总结

本文重点介绍了JDK中采用轻量级锁和偏向锁等对Synchronized的优化，但是这两种锁也不是完全没缺点的，比如竞争比较激烈的时候，不但无法提升效率，反而会降低效率，因为多了一个锁升级的过程，这个时候就需要通过-XX:-UseBiasedLocking来禁用偏向锁。下面是这几种锁的对比：

锁	优点	缺点	适用场景
---	----	----	------

5. Re:Spring Boot实战：拦截器与过滤器
这两个使用场景有啥区别？
--四度空间的平面

阅读排行榜
1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(52638)
2. Java并发编程：Synchronized及其实现原理(43890)
3. Java 并发编程：volatile的使用及其原理(24424)
4. Java 并发编程：核心理论(22461)
5. Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）(21912)

评论排行榜
1. Java并发编程：Synchronized及其实现原理(21)
2. Java 并发编程：volatile的使用及其原理(16)
3. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(13)
4. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(12)
5. 通过反编译深入理解Java String及intern(10)

推荐排行榜
1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(36)
2. Java并发编程：Synchronized及其实现原理(33)
3. 从字节码层面看“HelloWorld”(26)
4. Java 并发编程：核心理论(21)
5. Spring Boot实战：逐行释义HelloWorld(15)

偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如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。 同步块执行速度非常快。
重量级锁	线程竞争不使用自旋，不会消耗CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。 同步块执行速度较长。

参考文献：

<http://www.iteye.com/topic/1018932>

<http://www.infoq.com/cn/articles/java-se-16-synchronized>

<http://frank1234.iteye.com/blog/2163142>

<https://www.artima.com/insidejvm/ed2/threadsynch3.html>

<http://www.tuicool.com/articles/2aeAZn>

作者：liuxiaopeng

博客地址：http://www.cnblogs.com/paddix/

声明：转载请在文章页面明显位置给出原文连接。

标签: Java, 并发编程

好文要顶

关注我

收藏该文



liuxiaopeng

关注 - 2

粉丝 - 249

+加关注

« 上一篇：Java并发编程：Synchronized及其实现原理

» 下一篇：Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

posted @ 2016-04-25 07:56 liuxiaopeng 阅读(21913) 评论(6) 编辑 收藏

评论列表

#1楼 2016-08-19 10:09 dracularking

为什么一旦出现多线程竞争的情况就必须撤销偏向锁呢？

支持(0) 反对(0)

#2楼 2017-04-18 18:45 只会一点java

这一篇是不是写的仓促了些，很多来龙去脉没讲通啊！

支持(0) 反对(0)

#3楼 2017-06-08 10:39 我这人脸盲

@ dracularking

当多个线程并发的时候实际会发现总是其中的一个线程获取锁，当一个线程访问同步代码的时候，会在对象头和锁的记录中存储线程的ID，下次这个线程再访问同步块的时候只需要检查对象头中的id就可以进入同步代码中，而无需释放锁。减少了释放锁和获得锁的资源消耗。但是当有其他线程在竞争锁的时候，那么就必须释放锁资源，才能获得这把锁，所以需要撤销偏向锁。

支持(0) 反对(0)

#4楼 2017-09-25 17:12 就这个名

在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

针对这句话有个疑问，如果我能保证对于一个同步块不存在同一时间访问同一个锁的情况，那就是没有竞争，那我就没必要做同步了嘛，也就不存在锁的问题，可能我还没理解到位，还希望能给予解答，谢谢！！

支持(0) 反对(0)

#5楼 2018-04-13 14:39 还好可以改名字

@ 就这个名

引用

在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

针对这句话有个疑问，如果我能保证对于一个同步块不存在同一时间访问同一个锁的情况，那就是没有竞争，那我就没必要做同步了嘛，也就不存在锁的问题，可能我还没理解到位，还希望能给予解答，谢谢！！

轻量级锁膨胀为重量级锁是JVM的特性，对于程序员来说都是锁，程序员就是不能保证代码运行时同步代码严格交替才要使用锁，把严格交替的效果交个JVM保证。JVM就看实际运行的情况，如果实际发生时同步代码交替近似于严格的交替，也就是一段同步代码等待另一个线程的同步代码的时间很短就能等到，那么这个时候JVM就让这个线程连续竞争几次锁，而不是挂起等待所释放时OS唤醒。如果JVM让竞争者竞争了足够多的次数，还是没有获得锁，那么就挂起线程等待唤醒，也就是进入重量级锁运行机制

支持(0) 反对(0)

#6楼 2018-04-13 14:41 还好可以改名字

博主写得很棒

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【活动】2050 大会 - 年青人因科技而团聚（ 5.26-27杭州·云栖小镇 ）

【活动】华为云全新一代云服务器·限时特惠5.6折

【推荐】腾讯云多款高规格服务器，免费申请试用6个月