

14 | 多线程之锁优化（下）：使用乐观锁优化并行操作

2019-06-20 刘超



你好，我是刘超。

前两讲我们讨论了 **Synchronized** 和 **Lock** 实现的同步锁机制，这两种同步锁都属于悲观锁，是保护线程安全最直观的方式。

我们知道悲观锁在高并发的场景下，激烈的锁竞争会造成线程阻塞，大量阻塞线程会导致系统的上下文切换，增加系统的性能开销。那有没有可能实现一种非阻塞型的锁机制来保证线程的安全呢？答案是肯定的。今天我就带你学习下乐观锁的优化方法，看看怎么使用才能发挥它最大的价值。

什么是乐观锁

开始优化前，我们先来简单回顾下乐观锁的定义。

乐观锁，顾名思义，就是说在操作共享资源时，它总是抱着乐观的态度进行，它认为自己可以成功地完成操作。但实际上，当多个线程同时操作一个共享资源时，只有一个线程会成功，那么失败的线程呢？它们不会像悲观锁一样在操作系统中挂起，而仅仅是返回，并且系统允许失败的线程重试，也允许自动放弃退出操作。

所以，乐观锁相比悲观锁来说，不会带来死锁、饥饿等活性故障问题，线程间的相互影响也远远比悲观锁要小。更为重要的是，**乐观锁没有因竞争造成的系统开销，所以在性能上也是更胜一筹。**

乐观锁的实现原理

相信你对上面的内容是有一定的了解的，下面我们来看看乐观锁的实现原理，有助于我们从根本上总结优化方法。

CAS是实现乐观锁的核心算法，它包含了3个参数：**V**（需要更新的变量）、**E**（预期值）和**N**（最新值）。

只有当需要更新的变量等于预期值时，需要更新的变量才会被设置为最新值，如果更新值和预期值不同，则说明已经有其它线程更新了需要更新的变量，此时当前线程不做操作，返回**V**的真实值。

1.CAS如何实现原子操作

在JDK中的concurrent包中，atomic路径下的类都是基于CAS实现的。**AtomicInteger**就是基于CAS实现的一个线程安全的整型类。下面我们通过源码来了解下如何使用CAS实现原子操作。

我们可以看到**AtomicInteger**的自增方法**getAndIncrement**是用了**Unsafe**的**getAndAddInt**方法，显然**AtomicInteger**依赖于本地方法**Unsafe**类，**Unsafe**类中的操作方法会调用CPU底层指令实现原子操作。

```
//基于CAS操作更新值
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}

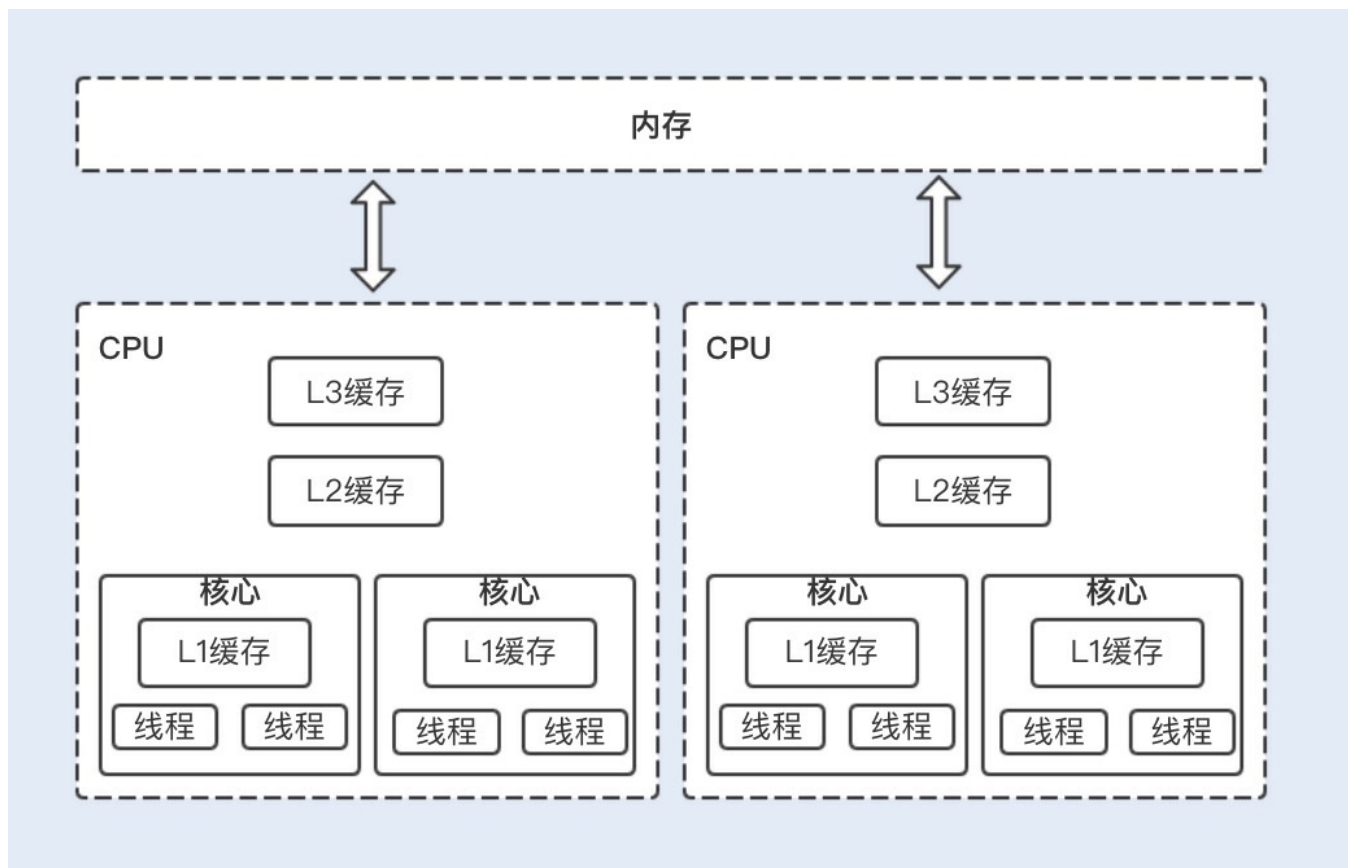
//基于CAS操作增1
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}

//基于CAS操作减1
public final int getAndDecrement() {
    return unsafe.getAndAddInt(this, valueOffset, -1);
}
```

2.处理器如何实现原子操作

CAS是调用处理器底层指令来实现原子操作，那么处理器底层又是如何实现原子操作的呢？

处理器和物理内存之间的通信速度要远慢于处理器间的处理速度，所以处理器有自己的内部缓存。如下图所示，在执行操作时，频繁使用的内存数据会缓存在处理器的**L1**、**L2**和**L3**高速缓存中，以加快频繁读取的速度。



一般情况下，一个单核处理器能自我保证基本的内存操作是原子性的，当一个线程读取一个字节时，所有进程和线程看到的字节都是同一个缓存里的字节，其它线程不能访问这个字节的内存地址。

但现在的服务器通常是多处理器，并且每个处理器都是多核的。每个处理器维护了一块字节的内存，每个内核维护了一块字节的缓存，这时候多线程并发就会存在缓存不一致的问题，从而导致数据不一致。

这个时候，处理器提供了**总线锁定**和**缓存锁定**两个机制来保证复杂内存操作的原子性。

当处理器要操作一个共享变量的时候，其在总线上会发出一个**Lock**信号，这时其它处理器就不能操作共享变量了，该处理器会独享此共享内存中的变量。但总线锁定在阻塞其它处理器获取该共享变量的操作请求时，也可能会导致大量阻塞，从而增加系统的性能开销。

于是，后来的处理器都提供了缓存锁定机制，也就是说当某个处理器对缓存中的共享变量进行了操作，就会通知其它处理器放弃存储该共享资源或者重新读取该共享资源。**目前最新的处理器都支持缓存锁定机制。**

优化CAS乐观锁

虽然乐观锁在并发性能上要比悲观锁优越，但是在写大于读的操作场景下，**CAS**失败的可能性会增大，如果不放弃此次**CAS**操作，就需要循环做**CAS**重试，这无疑会长时间地占用**CPU**。

在**Java7**中，通过以下代码我们可以看到：**AtomicInteger**的**getAndSet**方法中使用了**for**循环不断

重试CAS操作，如果长时间不成功，就会给CPU带来非常大的执行开销。到了Java8，for循环虽然被去掉了，但我们反编译Unsafe类时就可以发现该循环其实是被封装在了Unsafe类中，CPU的执行开销依然存在。

```
public final int getAndSet(int newValue) {  
    for (;;) {  
        int current = get();  
        if (compareAndSet(current, newValue))  
            return current;  
    }  
}
```

在JDK1.8中，Java提供了一个新的原子类LongAdder。LongAdder在高并发场景下会比AtomicInteger和AtomicLong的性能更好，代价就是会消耗更多的内存空间。

LongAdder的原理就是降低操作共享变量的并发数，也就是将对单一共享变量的操作压力分散到多个变量值上，将竞争的每个写线程的value值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的value值进行CAS操作，最后在读取值的时候会将原子操作的共享变量与各个分散在数组的value值相加，返回一个近似准确的数值。

LongAdder内部由一个base变量和一个cell[]数组组成。当只有一个写线程，没有竞争的情况下，LongAdder会直接使用base变量作为原子操作变量，通过CAS操作修改变量；当有多个写线程竞争的情况下，除了占用base变量的一个写线程之外，其它各个线程会将修改的变量写入到自己的槽cell[]数组中，最终结果可通过以下公式计算得出：

$$value = base + \sum_{i=0}^n Cell[i]$$

我们可以发现，LongAdder在操作后的返回值只是一个近似准确的数值，但是LongAdder最终返回的是一个准确的数值，所以在一些对实时性要求比较高的场景下，LongAdder并不能取代AtomicInteger或AtomicLong。

总结

在日常开发中，使用乐观锁最常见的场景就是数据库的更新操作了。为了保证操作数据库的原子

性，我们常常会为每一条数据定义一个版本号，并在更新前获取到它，到了更新数据库的时候，还要判断下已经获取的版本号是否被更新过，如果没有，则执行该操作。

CAS乐观锁在平常使用时比较受限，它只能保证单个变量操作的原子性，当涉及到多个变量时，CAS就无能为力了，但前两讲讲到的悲观锁可以通过对整个代码块加锁来做到这点。

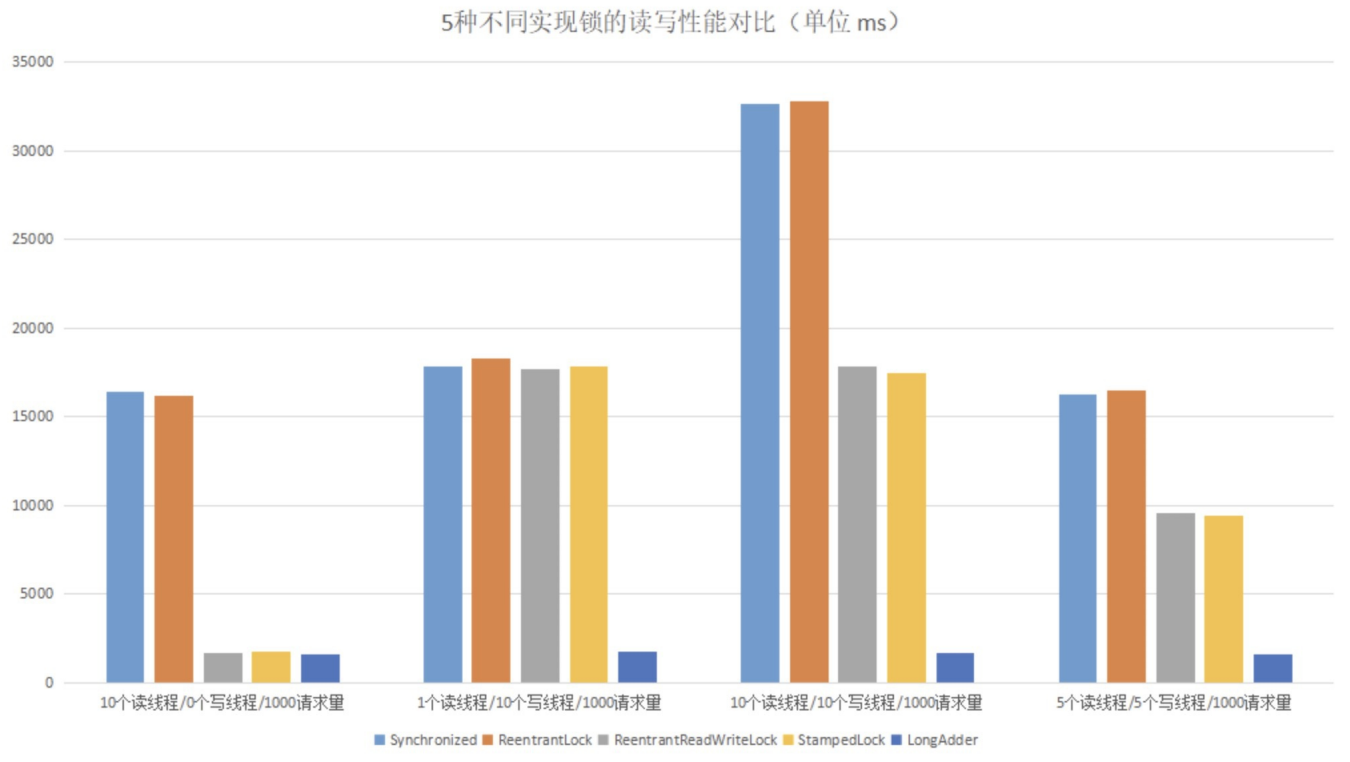
CAS乐观锁在高并发写大于读的场景下，大部分线程的原子操作会失败，失败后的线程将会不断重试CAS原子操作，这样就会导致大量线程长时间地占用CPU资源，给系统带来很大的性能开销。在JDK1.8中，Java新增了一个原子类LongAdder，它使用了空间换时间的方法，解决了上述问题。

11~13讲的内容，我详细地讲解了基于JVM实现的同步锁Synchronized，AQS实现的同步锁Lock以及CAS实现的乐观锁。相信你也很好奇，这三种锁，到底哪一种的性能最好，现在我们来对比下三种不同实现方式下的锁的性能。

鉴于脱离实际业务场景的性能对比测试没有意义，我们可以分别在“读多写少”“读少写多”“读写差不多”这三种场景下进行测试。又因为锁的性能还与竞争的激烈程度有关，所以除此之外，我们还将做三种锁在不同竞争级别下的性能测试。

综合上述条件，我将对四种模式下的五个锁Synchronized、ReentrantLock、ReentrantReadWriteLock、StampedLock以及乐观锁LongAdder进行压测。

这里简要说明一下：我是在不同竞争级别的情况下，用不同的读写线程数组合出了四组测试，测试代码使用了计算并发计数器，读线程会去读取计数器的值，而写线程会操作变更计数器值，运行环境是4核的i7处理器。结果已给出，具体的测试代码可以点击[Github](#)查看下载。



通过以上结果，我们可以发现：在读大于写的场景下，读写锁`ReentrantReadWriteLock`、`StampedLock`以及乐观锁的读写性能是最好的；在写大于读的场景下，乐观锁的性能是最好的，其它4种锁的性能则相差不多；在读和写差不多的场景下，两种读写锁以及乐观锁的性能要优于`Synchronized`和`ReentrantLock`。

思考题

我们在使用CAS操作的时候要注意的ABA问题指的是什么呢？

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。



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。

精选留言



crazypokerk

4

`LongAdder`原理如下：将热点数据value被分离成多个单元的cell，每个cell独自维护内部的值，当前对象的实际值由cell[]数组中所有的cell累计合成。这样，热点就进行了有效的分离，提高了并行度，所以`LongAdder`虽然降低了并发竞争，但是却对实时更新的数据不友好。

2019-06-20

作者回复

是的

2019-06-20



张学磊

4



2019-06-20

变量的原值为A，当线程T读取后到更新前这段时间，可能被其他线程更新为B值后又更新回A值，到当线程T进行CAS操作时感知不到这个变化，依然可以更新成功；StampdLock通过过去锁时返回一个时间戳可以解决该问题。

2019-06-20

作者回复

不仅回答了问题，还给出了解决方案，赞一个

2019-06-20



colin

3

老师您好，cell数组里存数得是+1 -1这种操作值么？

还有，“LongAdder 在操作后的返回值只是一个近似准确的数值，但是 LongAdder 最终返回的是一个准确的数值”这句话中“操作后返回值”和“最终返回值”怎么理解？

2019-06-20

作者回复

假设操作后立即要获取到值，这个值可能是一个不准确的值。如果我们等待所有线程执行完成之后去获取，这个值肯定是准确的值。一般在做统计时，会经常用到这种操作，实时展现的只要求一个近似值，但最终的统计要求是准确的。

2019-06-20



QQ怪

2

Longaddr还是不能理解,能否在举个简单点的例子理解吗？

2019-06-20



Loubobooo

1

一个变量V，如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。

2019-06-20



寻

0

很有帮助，系统性的重新审视学习各个锁，顺带将老师的测试代码用JMH测试框架、面向对象化重构了下。

<https://github.com/seasonsolt/lockTest>，有助于自己进一步深度学习研究。

2019-06-27

作者回复

赞

2019-06-28



左瞳

0

说乐观锁会占用大量的cpu导致性能下降，如果不是线程数影响，那哪些场景下乐观锁效率会低于悲观锁？

2019-06-26



左瞳

0

根据你的测试结果，都是乐观锁最优，是不是线程变为100个或者以上，其他测试结果才会优于乐观锁？

2019-06-26

作者回复

通常情况下，乐观锁的性能是要优于悲观锁，跟线程数量没有太大关系

2019-06-26



z.l

0

cas方法的三个参数是如何和cpu的缓存锁定机制联系到一起的呢？感觉没有理解，还请老师解答。

2019-06-23

作者回复

原理就是，当某个处理器对缓存中的共享变量进行了操作，就会通知其它处理器放弃对存储该共享资源或者重新读取该共享资源。

2019-06-24



陆离

0

解决ABA可以利用一个版本号进行验证，每次更新，版本号都+1，同时满足版本号与旧值相同才更新

2019-06-21



晓杰

0

ABA问题指的是假设现在有一个变量count的值为A，线程T1在未读取count值之前，线程T2把count值改为B，线程T3又把count值改为A，当线程T1读取count发现值为A，虽然值还是A，但是已经被其他线程改变过。

数值的原子递增可以不关注ABA问题，但是如果是原子化的更新对象，就需要关注ABA问题，因为有可能属性值发生了改变

2019-06-21



slowChef

0

如果从这个图看，LongAdder在几乎所有场景都远优于其他锁呀，是不是有问题呢？

2019-06-21

作者回复

乐观锁的性能要优于悲观锁，这个没问题。但乐观锁并不是适合所有场景，所以很多时候还是需要使用到悲观锁。

2019-06-23



明翼

0

关于最后的问题老师是不是可以通过版本号之类来控制，版本号只增加不减少是不是可以解决这个问题那

2019-06-21

作者回复

是的

2019-06-23



明翼

0

老师总线锁和缓存锁，这个缓存是L1还是L2还是L3那，总线锁可以理解成锁内存吗

2019-06-21

作者回复

这三个等级的缓存都会涉及到，理解没问题。

2019-06-23



WL

0

请教老师两个问题：

1. 文章中的这句话我不太理解，"我们可以发现，LongAdder 在操作后的返回值只是一个近似准确的数值，但是 LongAdder 最终返回的是一个准确的数值"。这么判断的依据是value的计算公式吗，为什么value的计算公式可以保证最终返回的准确性，公式中base和数组中各个槽的权重不一样，为什么base有这么大的权重呢？

2. 单核CPU是靠什么机制保证其他线程和进程都看到的缓存中的内容而不是内存中的内容呢？

2019-06-20



趙衍

0

老师好，我有两个问题想请教老师：

1.CAS实现原子性的方式我有点不太明白，缓存锁定解决的不应该是可见性的问题吗，保证当前线程对内存中数值的修改一定对其他线程可见，它是如何保证当前线程在获取值-->修改值-->写入值这个过程中不被其他的线程打断的呢？

2.通过性能对比其实可以发现，读写锁和StampedLock的性能相较于synchronized和可重入锁，要么差不多，要么更好，甚至在读少写多的情况下也体现出了一点点的优势，那么在使用锁的时候，我们为什么还要用后两种锁呢？他们的优点在哪里？

谢谢老师！

2019-06-20



Jxin

0

参考参数从a到b再到a，当前更新看不到变更过程，会误以为未变，从而更新成功。解决办法是引入累增版本，时间戳也是累增版本。最后是一个性能对比图，和我以前看的问章性能对比差距有点大，隐式锁性能不该这么差的。

2019-06-20



-W.LI-

0

老师那个测试的图是不是有问题啊，怎么一直是sysn和relock。两个悲观锁性能最好啊。

2019-06-20

编辑回复


同学你好～测试图单位是时间，时间花费越多，性能越差。

2019-06-20



Liam

0



CAS compare的依据是变量的值，ABA是指该变量从A到B再到A的变化过程，虽然变量已经被修改，从结果来看，CAS还是会认为变量没有被修改

2019-06-20

 作者回复

对的，Liam基础功扎实

2019-06-20