

## 13 | 多线程之锁优化（中）：深入了解Lock同步锁的优化方法

2019-06-18 刘超



你好，我是刘超。

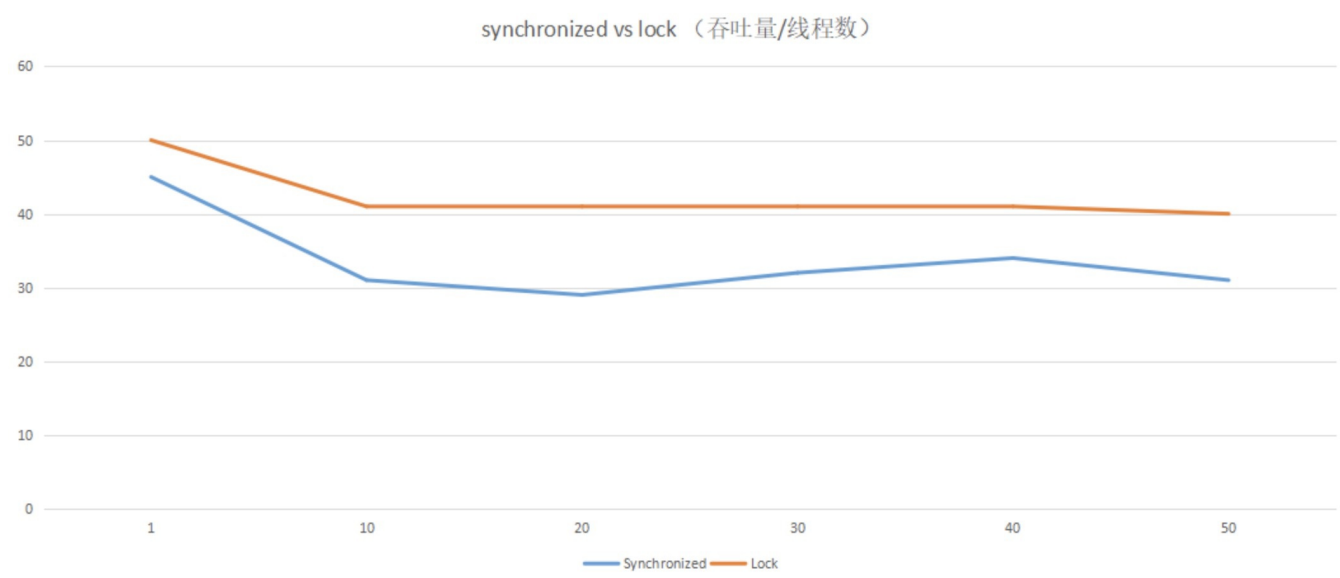
今天这讲我们继续来聊聊锁优化。上一讲我重点介绍了在JVM层实现的Synchronized同步锁的优化方法，除此之外，在JDK1.5之后，Java还提供了Lock同步锁。那么它有什么优势呢？

相对于需要JVM隐式获取和释放锁的Synchronized同步锁，Lock同步锁（以下简称Lock锁）需要的是显示获取和释放锁，这就为获取和释放锁提供了更多的灵活性。Lock锁的基本操作是通过乐观锁来实现的，但由于Lock锁也会在阻塞时被挂起，因此它依然属于悲观锁。我们可以通过一张图来简单对比下两个同步锁，了解下各自的特点：

	Synchronized	Lock
实现方式	JVM层实现	Java底层代码实现
锁的获取	JVM隐式获取	Lock.lock(): 获取锁，如被锁定则等待。 Lock.tryLock(): 如未被锁定才获取锁。 Lock.tryLock(long timeout, TimeUnit unit): 获取锁，如已被锁定，则最多等待timeout时间后返回获取锁状态。 Lock.lockInterruptibly(): 如当前线程未被interrupt才获取锁。
锁的释放	JVM隐式释放	通过Lock.unlock(), 在finally中释放锁
锁的类型	非公平锁、可重入	非公平锁、公平锁、可重入
锁的状态	不可中断	可中断

从性能方面上来说，在并发量不高、竞争不激烈的情况下，**Synchronized**同步锁由于具有分级锁的优势，性能上与**Lock**锁差不多；但在高负载、高并发的情况下，**Synchronized**同步锁由于竞争激烈会升级到重量级锁，性能则没有**Lock**锁稳定。

我们可以通过一组简单的性能测试，直观地对比下两种锁的性能，结果见下方，代码可以在[Github](#)上下载查看。



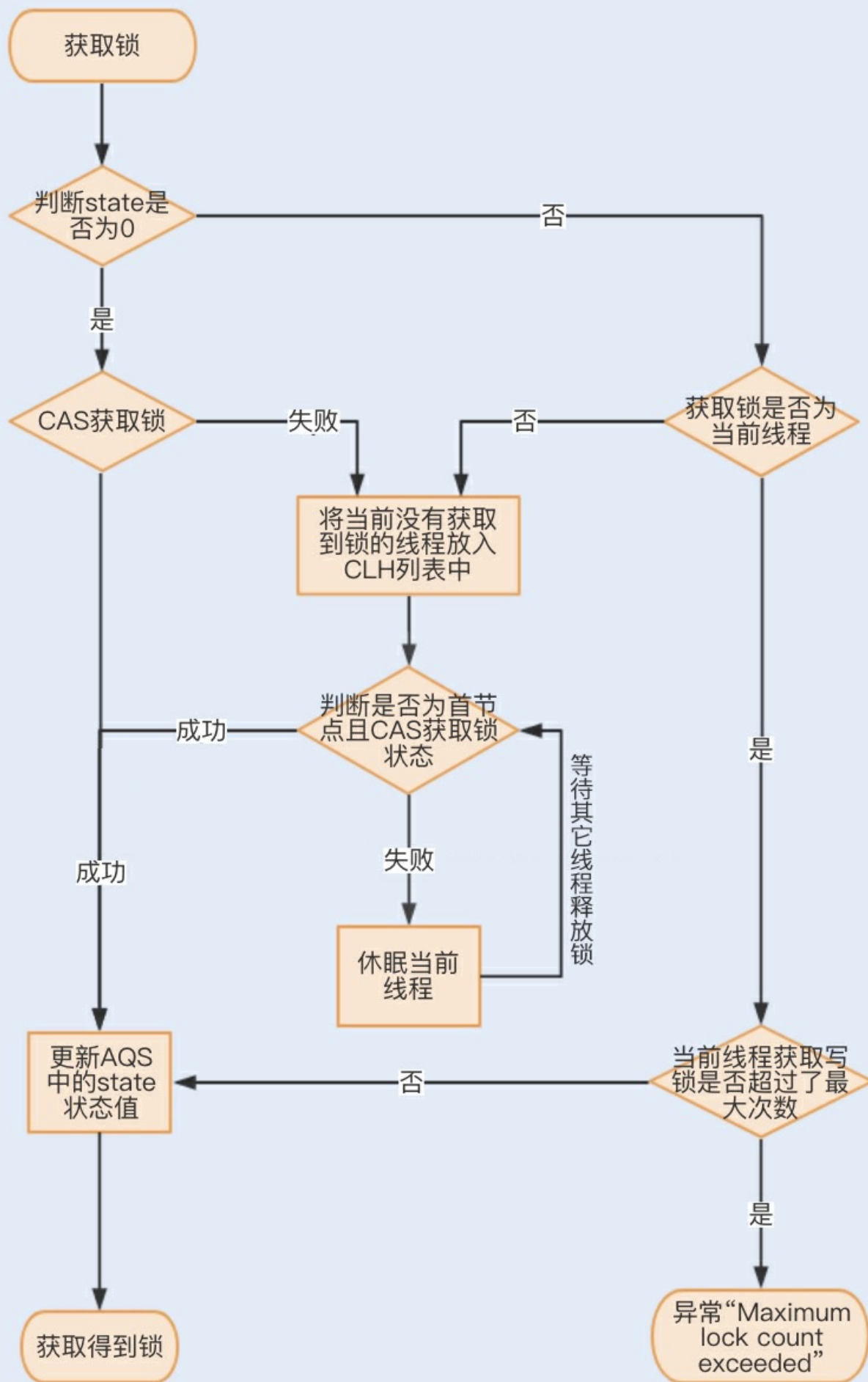
通过以上数据，我们可以发现：**Lock**锁的性能相对来说更加稳定。那它与上一讲的**Synchronized**同步锁相比，实现原理又是怎样的呢？

## Lock锁的实现原理

**Lock**锁是基于Java实现的锁，**Lock**是一个接口类，常用的实现类有**ReentrantLock**、**ReentrantReadWriteLock**（RRW），它们都是依赖**AbstractQueuedSynchronizer**（AQS）类实现的。

**AQS**类结构中包含一个基于链表实现的等待队列（**CLH**队列），用于存储所有阻塞的线程，**AQS**中还有一个**state**变量，该变量对**ReentrantLock**来说表示加锁状态。

该队列的操作均通过**CAS**操作实现，我们可以通过一张图来看下整个获取锁的流程。



虽然Lock锁的性能稳定，但也并不是所有的场景下都默认使用ReentrantLock独占锁来实现线程同步。

我们知道，对于同一份数据进行读写，如果一个线程在读数据，而另一个线程在写数据，那么读到的数据和最终的数据就会不一致；如果一个线程在写数据，而另一个线程也在写数据，那么线程前后看到的数据也会不一致。这个时候我们可以在读写方法中加入互斥锁，来保证任何时候只能有一个线程进行读或写操作。

在大部分业务场景中，读业务操作要远远大于写业务操作。而在多线程编程中，读操作并不会修改共享资源的数据，如果多个线程仅仅是读取共享资源，那么这种情况下其实没有必要对资源进行加锁。如果使用互斥锁，反倒会影响业务的并发性能，那么在这种场景下，有没有什么办法可以优化下锁的实现方式呢？

## 1.读写锁ReentrantReadWriteLock

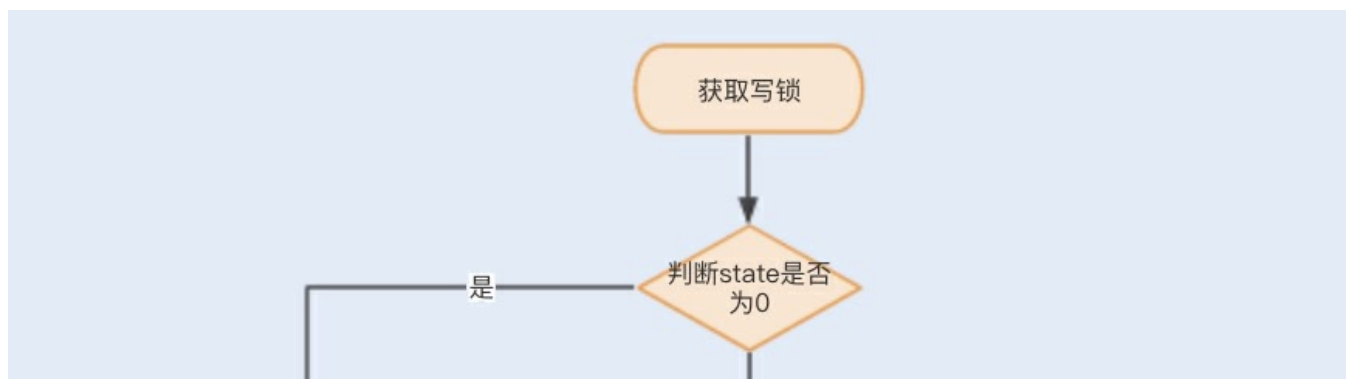
针对这种读多写少的场景，Java提供了另外一个实现Lock接口的读写锁RRW。我们已知ReentrantLock是一个独占锁，同一时间只允许一个线程访问，而RRW允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。读写锁内部维护了两个锁，一个是用于读操作的ReadLock，一个是用于写操作的WriteLock。

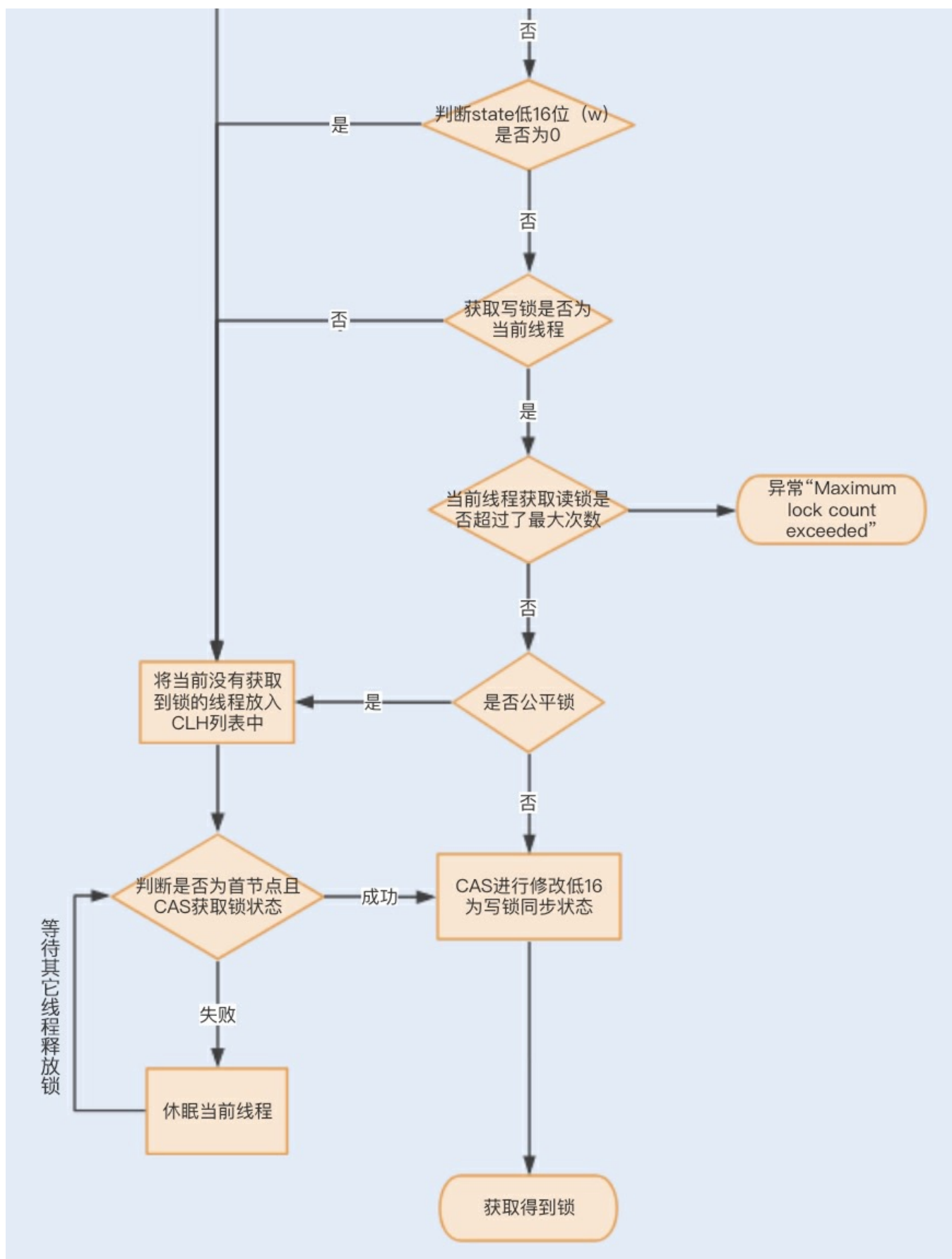
那读写锁又是如何实现锁分离来保证共享资源的原子性呢？

RRW也是基于AQS实现的，它的自定义同步器（继承AQS）需要在同步状态state上维护多个读线程和一个写线程的状态，该状态的设计成为实现读写锁的关键。RRW很好地使用了高低位，来实现一个整型控制两种状态的功能，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写。

一个线程尝试获取写锁时，会先判断同步状态state是否为0。如果state等于0，说明暂时没有其它线程获取锁；如果state不等于0，则说明有其它线程获取了锁。

此时再判断同步状态state的低16位（w）是否为0，如果w为0，则说明其它线程获取了读锁，此时进入CLH队列进行阻塞等待；如果w不为0，则说明其它线程获取了写锁，此时要判断获取了写锁的是不是当前线程，若不是就进入CLH队列进行阻塞等待；若是，就应该判断当前线程获取写锁是否超过了最大次数，若超过，抛异常，反之更新同步状态。



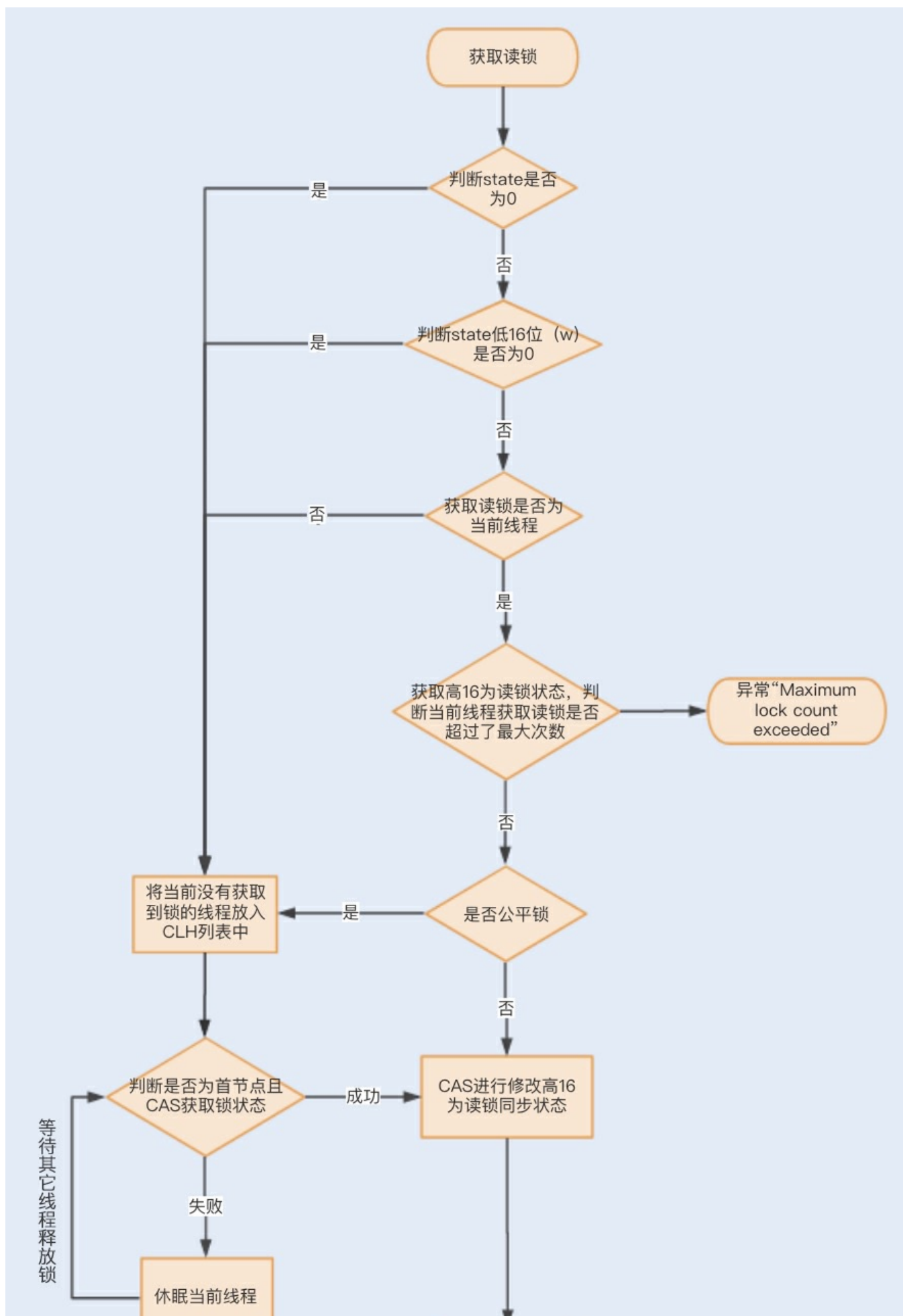


一个线程尝试获取读锁时，同样会先判断同步状态`state`是否为0。如果`state`等于0，说明暂时没有其它线程获取锁，此时判断是否需要阻塞，如果需要阻塞，则进入CLH队列进行阻塞等待；如果不需要阻塞，则CAS更新同步状态为读状态。

如果`state`不等于0，会判断同步状态低16位，如果存在写锁，则获取读锁失败，进入CLH阻塞队



列；反之，判断当前线程是否应该被阻塞，如果不应该阻塞则尝试CAS同步状态，获取成功更新同步锁为读状态。



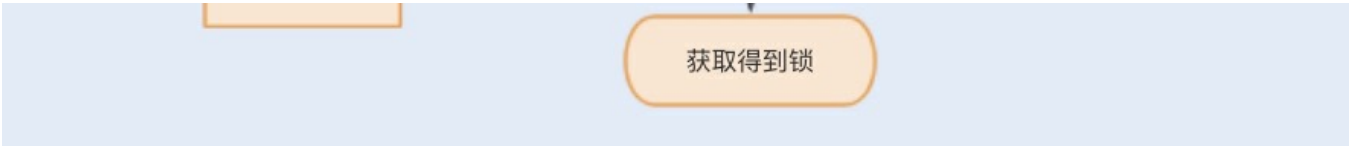


Diagram illustrating a process flow. A light blue rectangular area contains two orange rounded rectangular boxes. The first box is on the left, and the second box is on the right, connected by a horizontal line. The second box contains the text '获取得到锁' (Acquire the lock).

获取得到锁

下面我们通过一个求平方的例子，来感受下RRW的实现，代码如下：



```
public class TestRTTLock {

    private double x, y;

    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    // 读锁
    private Lock readLock = lock.readLock();
    // 写锁
    private Lock writeLock = lock.writeLock();

    public double read() {
        //获取读锁
        readLock.lock();
        try {
            return Math.sqrt(x * x + y * y);
        } finally {
            //释放读锁
            readLock.unlock();
        }
    }

    public void move(double deltaX, double deltaY) {
        //获取写锁
        writeLock.lock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            //释放写锁
            writeLock.unlock();
        }
    }

}
```

## 2.读写锁再优化之StampedLock

RRW被很好地应用在了读大于写的并发场景中，然而RRW在性能上还有可提升的空间。在读取很多、写入很少的情况下，RRW会使写入线程遭遇饥饿（Starvation）问题，也就是说写入线程会因迟迟无法竞争到锁而一直处于等待状态。

在JDK1.8中，Java提供了StampedLock类解决了这个问题。StampedLock不是基于AQS实现的，但实现的原理和AQS是一样的，都是基于队列和锁状态实现的。与RRW不一样的是，StampedLock控制锁有三种模式：写、悲观读以及乐观读，并且StampedLock在获取锁时会返回一个票据stamp，获取的stamp除了在释放锁时需要校验，在乐观读模式下，stamp还会作为读取共享资源后的二次校验，后面我会讲解stamp的工作原理。

我们先通过一个官方的例子来了解下StampedLock是如何使用的，代码如下：

```
public class Point {  
    private double x, y;  
    private final StampedLock s1 = new StampedLock();  
  
    void move(double deltaX, double deltaY) {  
        //获取写锁  
        long stamp = s1.writeLock();  
        try {  
            x += deltaX;  
            y += deltaY;  
        } finally {  
            //释放写锁  
            s1.unlockWrite(stamp);  
        }  
    }  
  
    double distanceFormOrigin() {  
        //乐观读操作  
        long stamp = s1.tryOptimisticRead();  
        //拷贝变量  
        double currentX = x, currentY = y;  
        //判断读期间是否有写操作  
        if (!s1.validate(stamp)) {  
            //升级为悲观读  
            stamp = s1.readLock();  
            try {  
                currentX = x;  
                currentY = y;  
            } finally {  
                s1.unlockRead(stamp);  
            }  
        }  
        return Math.sqrt(currentX * currentX + currentY * currentY);  
    }  
}
```

我们可以发现：一个写线程获取写锁的过程中，首先是通过**WriteLock**获取一个票据**stamp**，**WriteLock**是一个独占锁，同时只有一个线程可以获取该锁，当一个线程获取该锁后，其它请求的线程必须等待，当没有线程持有读锁或者写锁的时候才可以获取到该锁。请求该锁成功后会返回一个**stamp**票据变量，用来表示该锁的版本，当释放该锁的时候，需要**unlockWrite**并传递参数**stamp**。

接下来就是一个读线程获取锁的过程。首先线程会通过乐观锁**tryOptimisticRead**操作获取票据**stamp**，如果当前没有线程持有写锁，则返回一个非0的**stamp**版本信息。线程获取该**stamp**后，将会拷贝一份共享资源到方法栈，在这之前具体的操作都是基于方法栈的拷贝数据。

之后方法还需要调用**validate**，验证之前调用**tryOptimisticRead**返回的**stamp**在当前是否有其它线程持有了写锁，如果是，那么**validate**会返回0，升级为悲观锁；否则就可以使用该**stamp**版本的锁对数据进行操作。

相比于RRW，**StampedLock**获取读锁只是使用与或操作进行检验，不涉及CAS操作，即使第一次乐观锁获取失败，也会马上升级至悲观锁，这样就可以避免一直进行CAS操作带来的CPU占用性能的问题，因此**StampedLock**的效率更高。

## 总结

不管使用**Synchronized**同步锁还是**Lock**同步锁，只要存在锁竞争就会产生线程阻塞，从而导致线程之间的频繁切换，最终增加性能消耗。因此，**如何降低锁竞争，就成为了优化锁的关键。**

在**Synchronized**同步锁中，我们了解了可以通过减小锁粒度、减少锁占用时间来降低锁的竞争。在这一讲中，我们知道可以利用**Lock**锁的灵活性，通过锁分离的方式来降低锁竞争。

**Lock**锁实现了读写锁分离来优化读大于写的场景，从普通的RRW实现到读锁和写锁，到**StampedLock**实现了乐观读锁、悲观读锁和写锁，都是为了降低锁的竞争，促使系统的并发性能达到最佳。

## 思考题

**StampedLock**同RRW一样，都适用于读大于写操作的场景，**StampedLock**青出于蓝结果却不好说，毕竟RRW还在被广泛应用，就说明它还有**StampedLock**无法替代的优势。**你知道**StampedLock**没有被广泛应用的原因吗？或者说它还存在哪些缺陷导致没有被广泛应用。**

期待在留言区看到你的见解。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



-W.LI-

5

老师好!读写锁那个流程图看不太明白，没有写线程的时候，判断不是当前线程在读就会进入CLF阻塞等待。

问题1:不是可以并发读的嘛?按这图读线程也要阻塞等待的意思么?

问题二:CLF阻塞队列里是读写线程公用的么?队列里，读写交替出现。那不就没法并发读了么?

2019-06-18

作者回复

第一个问题，这里有一个公平锁和非公平锁的情况，如果是公平锁，即使无锁竞争的情况下，也会进入阻塞队列中排队获取锁；否则，会立即CAS获取到读锁。

第二个问题，是公用的，这里同样涉及到了公平锁和非公平锁，读写线程对于程序来说都是一样的。如果是非公平锁，如果没有锁竞争的情况下CAS获取锁成功，是无需进入阻塞队列。如果是公平锁，都会进入阻塞队列。

2019-06-18



Liam

4

StampLock不支持重入，不支持条件变量，线程被中断时可能导致CPU暴涨

2019-06-18

作者回复

回答很全面

2019-06-19



趙衍

👍 2

谢谢老师的回复！关于**StampedLock**，我的理解是乐观读的时候，线程把stamp的值读出来，通过与运算来判断当前是否存在写操作。这个过程是不涉及**CAS**操作的。可是如果有线程需要修改当前的资源，要加写锁，那么就需要使用**CAS**操作修改stamp的值。不知道这样理解是否准确。

此外，前排@-W.LI同学提出的那个问题，并发读的时候也需要按照是否是公平锁进入**CLH**队列进行阻塞我还不是很明白，既然大家都是读操作，互相之间没有冲突，我每个线程都直接用**CAS**操作获取锁不就行了吗，为什么还要进队列阻塞等待呢？

2019-06-19



我知道了呢

👍 1

可重入锁是什么？另外什么场景下会使用到？

2019-06-20



QQ怪

👍 1

老师这篇干货很多，看了2~3遍，大体理解了底层**AQS**锁原理，期待老师多多分享更多相关的文章

2019-06-18



趙衍

👍 1

老师我有几个问题：

1.在**ReentrantLock**中，**state**这个变量，为0的时候表示当前的锁是没有被占用的。这个时候线程应该用**CAS**尝试修改**state**变量的值对锁进行抢占才对呀，为什么在您的图里当**state=0**的时候还需要判断是否为当前线程呢？

2.老师提到读写锁在读多写少的情况下会使得写线程遭遇饥饿问题，那我是不是只需要将锁设置为公平锁，这样先申请写锁的线程就可以先获得锁，从而避免饥饿问题呢？

3.**StampedLock**中引入了一个**stamp**版本对版本进行控制，那么对这个**stamp**变量进行写入的时候是否需要使用**CAS**操作？如果不是，那如何保证对**stamp**变量的读写是线程安全的呢？

谢谢老师！

2019-06-18

作者回复

第一个问题，是老师笔误，搞错方向了，现在已更正。

第二个问题，如果读多写少的情况下，即使是公平锁，也是需要长时间等待，不是想获取时就能立即获取到锁。**StampedLock**如果是处于乐观读时，写锁是可以随时获取到锁。

第三个问题，**StampedLock**源码中存在大量**compareAndSwapObject**操作来保证原子性。

2019-06-19



QQ怪

👍 1

刚想反馈图片一个字母写反了，刷新一下，立马被修复了，厉害厉害，佩服老师的效率

2019-06-18



-W.LI-

1

StampedLock在写多读少的时候性能会很差吧

2019-06-18

作者回复

是的，写多读少的性能没有优势。

2019-06-19



密码123456

1

为什么？因为锁不可重入？

2019-06-18

作者回复

是的，StampedLock不支持可重入。如果在一些需要重入的代码中使用StampedLock，会导致死锁、饿死等情况出现。

2019-06-18



张学磊

1

老师，tryOptimisticRead操作获取的不应该叫乐观读锁，应该是乐观读，是无锁的；StampedLock名字中没有Reentrant，所以不支持重入；StampedLock也不支持条件变量。

2019-06-18

作者回复

这就是一种按版本号实现的读乐观锁，我们经常会在数据库更新操作时用到这种基于版本号实现的写乐观锁。

对的，StampedLock不支持重入。

2019-06-18



Geek\_ebda96

0

老师，RRW锁的读锁在获取锁的时候如果没有写锁，直接就可以获取到锁，只不过获取锁的过程中要用到CAS操作，相比于stampedlock，这个的乐观锁操作其实没用到任何锁操作，try的过程只是判断有没有写锁，没有则把共享变量的值拷贝到栈里面，后面的validate操作，也是再判断有没有写锁，没有则继续操作，这样理解对吗？那思考题里的问题，是因为乐观锁的过程除了try和validate操作判断有没有写锁，实际更新共享变量的值过程中没有cas和锁的操作，乐观锁的过程中其他线程还是可以获取到写锁，没法操作结果一定正确

最后还有一个问题这两种锁的读锁的cas操作只是在保证获取锁的过程和更新锁状态的过程吧，加锁的过程本身是要把内存中共享变量的值更新到栈中，共享变量本身不用volatile修饰？

2019-06-28

作者回复

RRW的读也是有锁的，所以不需要volatile修饰。

2019-06-28



Geek\_ebda96

0





Geek\_Educa3U

0

老师好，请问一下为什么`rrw`获取锁的时候，`state`状态为0还是需要把当前线程先加入`clh`等待队列，不直接去`cas`更新状态获取锁，为了公平性吗，是根据参数设置的值判断的吗

2019-06-27

作者回复

这里纠正一下，应该要先去判断是否是公平锁，如果是，则进入到`CLH`队列中，否则直接`CAS`获取锁。

2019-06-28



余冲

0

老师，你那个`rrw`获取写锁时的图，第一个判断及其后面的判断，直线的逻辑，应该是:是。否才加入`clh`队列。

2019-06-25

作者回复

没有错，如果`state`为0，则表示没有被其他线程占用锁资源，进入`CAS`获取锁；否则，则需要继续判断高低位状态。

2019-06-26



小布丁

0

老师我有一个问题，`RRW`在没有写锁的情况下，可以并发读，既然可以并发读为什么还要获取锁呢？是不是意味着读锁是可以被很多线程同时拥有的？而写锁就是独占的？

2019-06-24

作者回复

是的，读锁是一个共享锁，而写锁是一个独占锁。

2019-06-26



小橙橙

0

老师好！为什么读写锁判断`state`的地方要使用高低位这种设计呢，直接使用0、1、2这样的枚举判断理解上不是要更简单吗？

2019-06-22

作者回复

这里用到`state`高低位设计，可以优化锁类型的判断，例如只要被获取锁，`state`的变量就不为0，之后再去通过`state`的高低位判断是读锁还是写锁。

当然用枚举判断也行，如果用一个`int`类型能完成的事情，那就不要去用枚举，这样不仅逻辑清晰，也节约了内存空间。

2019-06-23



周星星

0

`sync`使用的是操作系统的`Mutex Lock`来实现的锁，`Lock`是使用线程等待来实现锁的，线程也会存在用户态内核态的切换，这样理解对吗？

2019-06-21

作者回复

对的。进程上下文切换，是指用户态和内核态的来回切换。我们知道，如果一旦Synchronized锁资源竞争激烈，线程将会被阻塞，阻塞的线程将会从用户态调用内核态，尝试获取mutex，这个过程就是进程上下文切换。

2019-06-23



shoo

👍 0

有个问题，最近在优化程序，刚好看到这一节

```
private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

//写锁

```
private Lock writeLock = lock.writeLock();
```

```
writeLock.lock();//这里为什么回报空指针异常
```

2019-06-20



英长

👍 0

希望老师能多结合实践讲讲应用场景

2019-06-19



G

👍 0

不可重入

2019-06-18



趙衍

👍 0

既然Lock的性能又比synchronized好，又提供了专门用于读多写少场景下的读写锁和StampedLock，那我们什么时候应该用synchronized而不用Lock呢？

2019-06-18

作者回复

在锁资源竞争不是很激烈的情况下，偶尔需要同步时，使用synchronized既简单又方便，而且JVM的编译器会尽可能的优化锁。

2019-06-19