**18、StampedLock:有没有比读写锁更快的锁？**[](#18stampedlock)

* StampedLock支持的三种锁模式
* 进一步的理解乐观锁
* StampedLock使用注意事项
* 总结

我们知道读写锁” **允许多个线程同时读共享变量，适用于读多写少的场景**“。那么在读多写少的场景中，还有没有更快的技术方案吗？

还真有，Java在1.8这个版本里，提供一种叫StampedLock的锁，它的性能就比读写锁还要好。

**StampedLock支持的三种锁模式**[](#stampedlock)

StampedLock和ReadWriteLock有哪些区别？

ReadWriteLock支持两种模式：一种是读锁，一种是写锁。而StampedLock支持三种模式，分别是： **写锁、悲观读锁和乐观读。** 其中写锁和悲观锁的语义和ReadWriteLock的写锁、读锁的语义非常类似，允许多个线程同时获取悲观读锁，但是只允许一个线程获取写锁，写锁和悲观读锁是互斥的。不同的是：StampedLock里的写锁和悲观读锁加锁成功之后，都会返回一个stamp；然后解锁的时候，需要传入这个stamp。相关示例代码如下：

**final** StampedLock sl **=** **new** StampedLock**();**

*//获取/释放悲观读硕示意代码*

**long** stamp **=** sl**.**readLock**();**

**try{**

*//业务代码*

**}finally{**

sl**.**unlockRead**(**stamp**);**

**}**

*//获取释放写锁示意代码*

**long** stamp **=** sl**.**writeLock**();**

**try{**

*//业务代码*

**}finally{**

sl**.**unlockWrite**(**stamp**)**

**}**

StampedLock的性能比ReadWriteLock还要好，其关键是StampedLock支持乐观读的方式。ReadWriteLock支持多个线程同时读，但是当多个线程同时读的时候，所有的写操作都会被阻塞；而StatmpedLock提供乐观读，是允许一个线程获取写锁的，也就是说不是所有的写操作都是被阻塞。

注意这里是乐观读而不是乐观锁。**乐观读这个操作是无锁的**，所以相比较ReadWriteLock的读锁，乐观读的性能更好一些。

JavaSDK官方示例

**class** **Point{**

**private** **int** x**,** y**;**

**final** StampedLock sl **=** **new** StampedLock**();**

**int** **distanceFromOrigin(){**

*//无锁乐观读*

**long** stamp **=** sl**.**tryOptimisticRead**();**

*// 读入局部变量，*

*// 读的过程数据可能被修改*

**int** curX **=**x**;**

**int** curY **=** y**;**

*// 判断执行读操作期间，*

*// 是否存在写操作，如果存在，*

*// 则 sl.validate 返回 false*

**if(!**sl**.**validate**(**stamp**)){**

*//升级为悲观读锁*

stamp **=** sl**.**readLock**();**

**try{**

curX **=** x**;**

curY **=** y**;**

**}finally{**

*//释放悲观读锁*

sl**.**unlockRead**(**stamp**);**

**}**

**}**

**return** Math**.**sqrt**(**curX**\***curX**+** curY**\***curY**);**

**}**

**}**

首先通过调用 tryOptimisticRead() 获取了一个 stamp，这里的 tryOptimisticRead() 就是我们前面提到的乐观读。之后将共享变量 x 和 y 读入方法的局部变量中，不过需要注意的是，由于 tryOptimisticRead() 是无锁的，所以共享变量 x 和 y 读入方法局部变量时，x 和 y 有可能被其他线程修改了。因此最后读完之后，还需要再次验证一下是否存在写操作，这个验证操作是通过调用 validate(stamp) 来实现的。

在上述的代码示例中，如果执行乐观读操作期间，存在写操作，会把乐观读升级为悲观读锁，这个做法挺合理的，否则你就需要在一个循环里反复执行乐观读，知道执行乐观读操作期间没有写操作（只有这样才能保证x和y的正确性和一致性），而循环读会浪费大量的CPU。升级为悲观读，代码简练且不易出错，建议你在具体实践时也采用这样的方法。

**进一步的理解乐观读**[](#_1)

你会发现数据库里的乐观锁，查询的时候需要把 version 字段查出来，更新的时候要利用 version 字段做验证。这个 version 字段就类似于 StampedLock 里面的 stamp。这样对比着看，相信你会更容易理解 StampedLock 里乐观读的用法。

**StampedLock使用注意事项**[](#stampedlock_1)

对于读多写少的场景 StampedLock 性能很好，简单的应用场景基本上可以替代 ReadWriteLock，但是 **StampedLock 的功能仅仅是 ReadWriteLock 的子集**，在使用的时候，还是有几个地方需要注意一下。

StampedLock 在命名上并没有增加Reentrant, 想必StampedLock应该是不可重入的。

**事实上 StampedLock不支持重入。** 这个是在使用中必须要特别注意的。

**StampedLock的悲观读锁、写锁都不支持Condition条件变量，这个也需要你注意。**

还有一点需要特别注意，那就是：如果线程阻塞在 StampedLock 的 readLock() 或者 writeLock() 上时，此时调用该阻塞线程的 interrupt() 方法，会导致 CPU 飙升。例如下面的代码中，线程 T1 获取写锁之后将自己阻塞，线程 T2 尝试获取悲观读锁，也会阻塞；如果此时调用线程 T2 的 interrupt() 方法来中断线程 T2 的话，你会发现线程 T2 所在 CPU 会飙升到 100%。

**final** StampedLock lock

**=** **new** StampedLock**();**

Thread T1 **=** **new** Thread**(()->{**

*// 获取写锁*

lock**.**writeLock**();**

*// 永远阻塞在此处，不释放写锁*

LockSupport**.**park**();**

**});**

T1**.**start**();**

*// 保证 T1 获取写锁*

Thread**.**sleep**(**100**);**

Thread T2 **=** **new** Thread**(()->**

*// 阻塞在悲观读锁*

lock**.**readLock**()**

**);**

T2**.**start**();**

*// 保证 T2 阻塞在读锁*

Thread**.**sleep**(**100**);**

*// 中断线程 T2*

*// 会导致线程 T2 所在 CPU 飙升*

T2**.**interrupt**();**

T2**.**join**();**

所以，使用StampedLock一定不要调用中断操作，如果需要支持中断操作，可以使用悲观读锁 readLockInterruptibly()和写锁writeLockInterruptibly()。

原因：内部实现里While循环里面面对中断的处理有点问题

**总结**[](#_2)

StampedLock 读模板：

**final** StampedLock sl **=**

**new** StampedLock**();**

*// 乐观读*

**long** stamp **=**

sl**.**tryOptimisticRead**();**

*// 读入方法局部变量*

**......**

*// 校验 stamp*

**if** **(!**sl**.**validate**(**stamp**)){**

*// 升级为悲观读锁*

stamp **=** sl**.**readLock**();**

**try** **{**

*// 读入方法局部变量*

**.....**

**}** **finally** **{**

*// 释放悲观读锁*

sl**.**unlockRead**(**stamp**);**

**}**

**}**

*// 使用方法局部变量执行业务操作*

**......**

StampedLock 写模板：

**long** stamp **=** sl**.**writeLock**();**

**try** **{**

*// 写共享变量*

**......**

**}** **finally** **{**

sl**.**unlockWrite**(**stamp**);**

**}**