

18 | StampedLock: 有没有比读写锁更快的锁？

2019-04-09 王宝令



在[上一篇文章](#)中，我们介绍了读写锁，学习完之后你应该已经知道“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”。那在读多写少的场景中，还有没有更快的技术方案呢？还真有，**Java**在**1.8**这个版本里，提供了一种叫**StampedLock**的锁，它的性能就比读写锁还要好。

下面我们就来介绍一下**StampedLock**的使用方法、内部工作原理以及在使用过程中需要注意的事项。

StampedLock支持的三种锁模式

我们先来看看在使用上**StampedLock**和上一篇文章讲的**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、悲观读锁的语义和**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**。相关的示例代码如下。

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

// 获取/释放悲观读锁示意代码
long stamp = sl.readLock();
try {
    //省略业务相关代码
} finally {
    sl.unlockRead(stamp);
}

// 获取/释放写锁示意代码
long stamp = sl.writeLock();
try {
    //省略业务相关代码
} finally {
    sl.unlockWrite(stamp);
}
```

StampedLock的性能之所以比**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，所有的写操作会被阻塞；而**StampedLock**提供的乐观读，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被阻塞。

注意这里，我们用的是“乐观读”这个词，而不是“乐观读锁”，是要提醒你，**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。

文中下面这段代码是出自**Java SDK**官方示例，并略做了修改。在**distanceFromOrigin()**这个方法中，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。

```

class Point {
    private int x, y;
    final StampedLock sl =
        new StampedLock();
    //计算到原点的距离
    int distanceFromOrigin() {
        // 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, 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);
    }
}

```

在上面这个代码示例中，如果执行乐观读操作的期间，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，直到执行乐观读操作的期间没有写操作（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。升级为悲观读锁，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。

进一步理解乐观读

如果你曾经用过数据库的乐观锁，可能会发现**StampedLock**的乐观读和数据库的乐观锁有异曲同工之妙。的确是这样的，就拿我个人来说，我是先接触的数据库里的乐观锁，然后才接触的**StampedLock**，我就觉得我前期数据库里乐观锁的学习对于后面理解**StampedLock**的乐观读有很大帮助，所以这里有必要再介绍一下数据库里的乐观锁。

还记得我第一次使用数据库乐观锁的场景是这样的：在**ERP**的生产模块里，会有多个人通过**ERP**系统提供的**UI**同时修改同一条生产订单，那如何保证生产订单数据是并发安全的呢？我采用的方案就是乐观锁。

乐观锁的实现很简单，在生产订单的表 `product_doc` 里增加了一个数值型版本号字段 `version`，每次更新`product_doc`这个表的时候，都将 `version` 字段加1。生产订单的**UI**在展示的时候，需要查询数据库，此时将这个 `version` 字段和其他业务字段一起返回给生产订单**UI**。假设用户查询的生产订单的`id=777`，那么**SQL**语句类似下面这样：

```
select id, ... , version
from product_doc
where id=777
```

用户在生产订单**UI**执行保存操作的时候，后台利用下面的**SQL**语句更新生产订单，此处我们假设该条生产订单的 `version=9`。

```
update product_doc
set version=version+1, ...
where id=777 and version=9
```

如果这条**SQL**语句执行成功并且返回的条数等于1，那么说明从生产订单**UI**执行查询操作到执行保存操作期间，没有其他人修改过这条数据。因为如果这期间其他人修改过这条数据，那么版本号字段一定会大于9。

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

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**，想必你已经猜测到**StampedLock**应该是不可重入的。事实上，的确是这样的，**StampedLock**不支持重入。这个是在使用中必须要特别注意的。

另外，**StampedLock**的悲观读锁、写锁都不支持条件变量，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，如果需要使用中断功能，一定使用可中断的悲观读锁**readLockInterruptibly()**和写锁**writeLockInterruptibly()**。这个规则一定要记清楚。

总结

StampedLock的使用看上去有点复杂，但是如果你能理解乐观锁背后的原理，使用起来还是比较流畅的。建议你认真揣摩**Java**的官方示例，这个示例基本上就是一个最佳实践。我们把**Java**官方示例精简后，形成下面的代码模板，建议你在实际工作中尽量按照这个模板来使用**StampedLock**。

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);
}
```

课后思考

StampedLock支持锁的降级（通过tryConvertToReadLock()方法实现）和升级（通过tryConvertToWriteLock()方法实现），但是建议你要慎重使用。下面的代码也源自Java的官方示例，我仅仅做了一点修改，隐藏了一个Bug，你来看看Bug出在哪里吧。

```
private double x, y;
final StampedLock sl = new StampedLock();
// 存在问题的方法
void moveIfAtOrigin(double newX, double newY){
    long stamp = sl.readLock();
    try {
        while(x == 0.0 && y == 0.0){
            long ws = sl.tryConvertToWriteLock(stamp);
            if (ws != 0L) {
                x = newX;
                y = newY;
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



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



linqw

14

课后思考题：在锁升级成功的时候，最后没有释放最新的写锁，可以在if块的break上加个stamp=ws进行释放

2019-04-09

作者回复

1

2019-04-09



Presley

6

老师，StampedLock 读模板，先通过乐观读或者悲观读锁获取变量，然后利用这些变量处理业务逻辑，会不会存在线程安全的情况呢？比如，读出来的变量没问题，但是进行业务逻辑处理的时候，这时，读出的变量有可能发生变化了吧(比如被写锁改写了)？所以，当使用乐观读锁时，是不是等业务都处理完了（比如先利用变量把距离计算完），再判断变量是否被改写，如果没改写，直接return;如果已经改写，则使用悲观读锁做同样的事情。不过如果业务比较耗时，可能持有悲观锁的时间会比较长，不知道理解对不对

2019-04-09

作者回复

两种场景，如果处理业务需要保持互斥，那么就用互斥锁，如果不需要保持互斥才可以用读写锁。一般来讲缓存是不需要保持互斥性的，能接受瞬间的不一致

2019-04-11



GrubbyJ

👍 6

bug是tryConvertToWriteLock返回的write stamp没有重新赋值给stamp

2019-04-09

作者回复

👍

2019-04-09



胡桥

👍 4

乐观锁的想法是“没事，肯定没被改过”，于是就开心地获取到数据，不放心吗？那就再验证一下，看看真的没被改过吧？这下可以放心使用数据了。

我的问题是，验证完之后、使用数据之前，数据被其他线程改了怎么办？我看不出validate的意义。这个和数据库更新好像还不一样，数据库是在写的时候发现已经被其他人写了。这里validate之后也难免数据在进行业务计算之前已经被改掉了啊？

2019-04-09

作者回复

改了就改了，读的数据是正确的一致就可以了。如果这个规则不满足业务需求，可以总互斥锁。不同的锁用不同地方。

2019-04-09



GrubbyJ

👍 3

老师，调用interrupt引起cpu飙高的原因是什么

2019-04-09

作者回复

内部实现里while循环里面对中断的处理有点问题

2019-04-09



冯传博

👍 2

解释一下 cpu 飙升的原因呗

2019-04-09



echo__陈

👍 2

以前看过java并发编程实战，讲jdk并发类库.....不过那个书籍是jdk1.7版本.....所以是头一次接触StampedLock.....涨知识了

2019-04-09



密码123456

👍 2

悲观锁和乐观锁。悲观锁，就是普通的锁。乐观锁，就是无锁，仅增加一个版本号，在取完数据验证一下版本号。如果不一致那么就进行悲观锁获取锁。能够这么理解吗？

2019-04-09



linqw

👍 1

从头重新看一篇，也自己大致写了下对StampedLock的源码分析<https://juejin.im/editor/posts/5d00a6c8e51d45105d63a4ed>，老师有空帮忙看下哦

2019-06-15

作者回复

👋, 有空我也学习一下👋

2019-06-15



ban

👍 1

老师, 你好,

如果我在前面`long stamp = sl.readLock();`升级锁后`long ws = sl.tryConvertToWriteLock(stamp);`这个 `stamp`和`ws`是什么关系来的, 是`sl.unlockRead()`(是关`stamp`还是`ws`)。两者有什么区别呢

2019-04-13

作者回复

`stamp`和`ws`没关系, `tryConvertToWriteLock(stamp)`这个方法内部会释放悲观读锁`stamp` (条件是能够升级成功)。所以我们需要释放的是`ws`

2019-04-14



ttang

👍 1

老师, `ReadWriteLock`锁和`StampedLock`锁都是可以同时读的, 区别是`StampedLock`乐观读不加锁。那`StampedLock`比`ReadWriteLock`性能高的原因就是节省了加读锁的性能损耗吗? 另外`StampedLock`用乐观读的时候是允许一个线程获取写锁的, 是不是可以理解为`StampedLock`对写的性能更高, 会不会因为写锁获取概率增大大, 导致不能获取读锁。导致`StampedLock`读性能反而没有`ReadWriteLock`高?

2019-04-10

作者回复

乐观读升级到悲观读, 就和`ReadWriteLock`一样了。

2019-04-10



楊_宵夜

👍 0

王老师, 您好, 文章中的`StampedLock`模板, 只适用于单机应用吧? 如果是集群部署, 那还是得用数据库乐观锁, 是吗??

2019-06-14

作者回复

是的

2019-06-14



渡码

👍 0

请教老师一下, 乐观读升级悲观读业务上有些不理解, 其实升级完乐观读读到数据后调用`math.sqrt`这个时候共享数据仍然被改。既然数据任何时候都可能被改何必多读一次?

2019-06-04



孟桐说的对啊

👍 0

解锁对象不一致, 一开始读锁是 `stamp` 后来锁升级后是 `ws` 。 `stamp`解锁了, `ws` 并没有

2019-05-28



小辉辉

👍 0



最近的项目里面就用到过乐观锁，用来防止并发提交更新数据。

2019-05-20



南北少卿

👍 0

jdk源码StampedLock中的示例，if (ws != 0L) 时使用了stamp=ws

2019-05-12

作者回复

👍

2019-05-13



狂风骤雨

👍 0

老师，你上章讲的ReadWriteLock，说的是当有一个线程在执行写操作时所有的读线程都被阻塞，本章你又提了一下ReadWriteLock，说的是当有多个线程进行读操作时，所有的写操作都被阻塞，这样是

2019-05-06

作者回复

读写是互斥的

2019-05-20



WuV1Up

👍 0

StampedLock 使用注意事项 这个小节的代码，我在本地机器测试了下，T2.interrupt没有某个CPU到100%的现象... 只是CPU略微升高了点。

2019-04-23



xiyuesmiling

👍 0

老师，当两个线程T1和T2如果都获取读锁，T1先升级失败而释放读锁，并阻塞在写锁；T2在1释放读锁时升级锁成功并且新值是0.0和0.0即与原来一样，最后释放写锁，T1在T2释放写锁之后获取写锁，下一个循环T2自己又尝试将写锁升级为写锁，这导致死锁？还是直接异常了走finally呢？是不是我哪里理解错了？👍

2019-04-20



发条橙子。

👍 0

老师，我看事例里面成员变量都给了一个final关键字。请问这里给变量加final的用意是什么，仅仅是为了防止下面方法中代码给他赋新的对象么。我在平常写代码中很少有给变量加final的习惯，希望老师能指点一下👍

2019-04-15

作者回复

使用final是个好习惯

2019-04-15