18-StampedLock：有没有比读写锁更快的锁？

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，如果需要支持中断功能，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。

猜你喜欢

精选留言：

- linqw 2019-04-09 00:29:22
  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 [11赞]

  作者回复2019-04-09 08:50:48
  👍

- Presley 2019-04-09 13:14:44
  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 [5赞]

  作者回复2019-04-11 09:11:42
  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

- Grubby 2019-04-09 02:35:16
  bug是tryConvertToWriteLock返回的write stamp没有重新赋值给stamp [4赞]

  作者回复2019-04-09 08:50:58
  👍

- 胡桥 2019-04-09 16:17:46
  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？ [3赞]

  作者回复2019-04-09 22:33:10
  改了就改了，读的数据是正确的一致的就可以了。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。

- Grubby 2019-04-09 02:31:59
  老师，调用interrupt引起cpu飙高的原因是什么 [3赞]

  作者回复2019-04-09 08:50:33
  内部实现里while循环里面对中断的处理有点问题

- echo__陈 2019-04-09 08:38:51
  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StempLock……涨知识了 [2赞]

- 密码123456 2019-04-09 08:38:27
  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？ [2赞]

- ttang 2019-04-10 20:23:22
  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？ [1赞]

  作者回复2019-04-10 21:46:38

乐观读升级到悲观读，就和ReadWriteLock一样了。

- 冯传博 2019-04-09 08:58:21
  解释一下 cpu 飙升的原因呗 [1赞]

- 发条橙子 。 2019-04-15 09:10:39
  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 😄

  作者回复2019-04-15 11:51:13
  使用final是个好习惯

- Geek_zy 2019-04-14 09:15:35
  王老师还有一个问题，最近做一些关于秒杀的业务，是不是可以用到乐观读的性质。
  将库存量放在redis里边，然后所有的节点操作的时候通过缓存读出来，在代码逻辑里边对库存加一个乐观读的操作。然后库存量等于0 的时候再去和数据库进行交互。 这样做会存在并发安全问题吗。

  作者回复2019-04-14 10:34:40
  如果用redis，就完全依赖redis，本地不能有缓存，有缓存就可能数据不一致。不清楚你有没有用本地缓存。redis做秒杀有很多成熟的方案，好像都没法用乐观读。

- Geek_zy 2019-04-14 09:09:32
  王老师，秒杀的场景下 对订单的数量加乐观读。会不会出现数据安全的问题呢

- ban 2019-04-13 02:01:23
  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

  作者回复2019-04-14 19:28:24
  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

- 冰激凌的眼泪 2019-04-12 18:16:04
  读多写少的场景，减少了加锁操作，大大提高了效率

- on the way 2019-04-11 08:18:59
  有点没看明白示例interrupt那个代码里的 Thread.sleep（100）…

  作者回复2019-04-13 09:25:01
  就是等一会儿，保证前面的子线程已经启动包

- 包子 2019-04-10 14:21:39
  老师，一直有个问题想不明白，就是对一个变量的读和写是否会存在线程安全问题。
  文章中举例是同时对x和y进行读写操作，那xy的读写不能保证原子性，所以需要用到锁。
  如果是对一个变量x的读和写，我们对x加volatile，保证其多线程的可见性是不是就可以了？

  作者回复2019-04-10 20:16:49
  如果只是写，只是读，没有任何其他逻辑，是可以的

- 胡桥 2019-04-10 10:30:39
  validate也无法保证一致性是吗？如果是那么应该怎么用validate？

  作者回复2019-04-10 12:12:28
  没法保证两个变量的一致性

- 刘章周 2019-04-09 17:52:30
  老师：如果线程走了else获取到读锁后又进去循环里，然后执行锁升级，会不会报错？

- 张天屹 2019-04-09 16:34:21
  说到数据库乐观锁，有个问题想请教老师，比如spring使用事务的时候，底层就已经使用了读写相关的锁来保证并发了，在我们的程序中还需要显式的使用乐观锁机制来保证并发安全吗

  作者回复2019-04-09 22:29:13
  看有没有多线程共享的变量，没有就不需要。文中数据库乐观锁的例子，有数据库事务也需要乐观锁。具体问题具体分析

- ZOU志伟 2019-04-09 16:12:25
  老师，上一篇的缓存获取的用StampedLock的读代码这样写可以吗？
  public V get(K key) {
  long stamp = sl.tryOptimisticRead();
  V value = map.get(key);
  if (!sl.validate(stamp)) {
  stamp = sl.readLock();
  try {
  value = map.get(key);
  } finally {
  sl.unlock(stamp);
  }
  }
  return value;
  }