18 | StampedLock: 有没有比读写锁更快的锁?

王宝令 2019-04-09





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,<mark>提供了一种叫 StampedLock 的锁,它的性能就比读写锁还要好</mark>。

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。

```
2
  new StampedLock();
4 // 获取 / 释放悲观读锁示意代码
5 long stamp = sl.readLock();
6 try {
7 // 省略业务相关代码
8 } finally {
9 sl.unlockRead(stamp);
10 }
12 // 获取 / 释放写锁示意代码
13 long stamp = sl.writeLock();
14 try {
15 // 省略业务相关代码
16 } finally {
17 sl.unlockWrite(stamp);
18 }
19
```

StampedLock 的性能之所以比 ReadWriteLock 还要好,<mark>其关键是 StampedLock 支持乐观读的方式。</mark>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;<mark>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。</mark>

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<mark>,乐观读这个操作</mark> **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。

■ 复制代码

```
1 class Point {
private int x, y;
3 final StampedLock sl =
    new StampedLock();
5 // 计算到原点的距离
  int distanceFromOrigin() {
   // 乐观读
    long stamp =
8
9
      sl.tryOptimisticRead();
    // 读入局部变量,
10
    // 读的过程数据可能被修改
11
    int curX = x, curY = y;
13
     // 判断执行读操作期间,
     // 是否存在写操作,如果存在,
14
15
    // 则 sl.validate 返回 false
16
    if (!sl.validate(stamp)){
```

```
// 升级为悲观读锁
      stamp = sl.readLock();
18
19
      try {
       curX = x;
20
       curY = y;
       } finally {
      // 释放悲观读锁
        sl.unlockRead(stamp);
      }
26
     }
     return Math.sqrt(
      curX * curX + curY * curY);
28
29
  }
30 }
```

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:

■ 复制代码

```
1 select id, ..., version
2 from product_doc
3 where id=777
4
```

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。

```
■复制代码
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%。

■ 复制代码

```
1 final StampedLock lock
2 = new StampedLock();
3 Thread T1 = new Thread(()->{
4    // 获取写锁
5    lock.writeLock();
6    // 永远阻塞在此处,不释放写锁
7    LockSupport.park();
8 });
9 T1.start();
10 // 保证 T1 获取写锁
11 Thread.sleep(100);
12 Thread T2 = new Thread(()->
```

所以<mark>,使用 StampedLock 一定不要调用中断操作,</mark>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:

```
自复制代码
```

```
1 final StampedLock sl =
2 new StampedLock();
4 // 乐观读
5 long stamp =
6 sl.tryOptimisticRead();
7 // 读入方法局部变量
8 .....
9 // 校验 stamp
10 if (!sl.validate(stamp)){
11
  // 升级为悲观读锁
stamp = sl.readLock();
13 try {
    // 读入方法局部变量
14
16 } finally {
   // 释放悲观读锁
17
18
    sl.unlockRead(stamp);
19 }
20 }
21 // 使用方法局部变量执行业务操作
22 .....
```

StampedLock 写模板:

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。

```
自复制代码
private double x, y;
2 final StampedLock sl = new StampedLock();
3 // 存在问题的方法
4 void moveIfAtOrigin(double newX, double newY){
5 long stamp = sl.readLock();
6 try {
 7 while(x == 0.0 \&\& y == 0.0){
8 long ws = sl.tryConvertToWriteLock(stamp);
    if (ws != 0L) {
9
13 } else {
     sl.unlockRead(stamp);
      stamp = sl.writeLock();
15
    }
16
17
   }
18 } finally {
19 sl.unlock(stamp);
20 }
21
```

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



(C)

◯ 一手资源 同步更新 加微信 ixuexi66

由作者筛选后的优质留言将会公开显示, 欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(4)



Grubby⊱

老师,调用interrupt引起cpu飙高的原因是什么

2 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



linqw

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

2 2019-04-09

作者回复: 🐿



 $\mathsf{Grubby} \, \mathbb{k}$

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

1 2019-04-09

作者回复: 🐿



扛着锄头闯江湖

先打个卡



2019-04-09