主讲老师：Fox

> 有道云笔记链接：https://note.youdao.com/s/Sy9cmgPx

现实中有这样一种场景：对共享资源有读和写的操作，且写操作没有读操作那么频繁（读多写少）。在没有写操作的时候，多个线程同时读一个资源没有任何问题，所以应该允许多个线程同时读取共享资源（读读共享）；但是如果一个线程想去写这些共享资源，就不应该允许其他线程对该资源进行读和写操作了（读写，写写互斥）。
思考：针对这种场景，有没有比ReentrantLock更好的方案?

1. 读写锁介绍

读写锁ReadWriteLock，顾名思义一把锁分为读与写两部分，读锁允许多个线程同时获得，因为读操作本身是线程安全的。而写锁是互斥锁，不允许多个线程同时获得写锁。并且读与写操作也是互斥的。读写锁适合多读少写的业务场景。

2. ReentrantReadWriteLock介绍

针对这种场景，JAVA的并发包提供了读写锁ReentrantReadWriteLock，它内部，维护了一对相关的锁，一个用于只读操作，称为读锁；一个用于写入操作，称为写锁，描述如下：
线程进入读锁的前提条件：

- 没有其他线程的写锁
- 没有写请求或者有写请求，但调用线程和持有锁的线程是同一个。

线程进入写锁的前提条件：

- 没有其他线程的读锁
- 没有其他线程的写锁

而读写锁有以下三个重要的特性：

- 公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。
- 可重入：读锁和写锁都支持线程重入。以读写线程为例：读线程获取读锁后，能够再次获取读锁。写线程在获取

写锁之后能够再次获取写锁，同时也可以获取读锁。

- 锁降级：遵循获取写锁、再获取读锁最后释放写锁的次序，写锁能够降级成为读锁。

2.1 ReentrantReadWriteLock的使用

读写锁接口ReadWriteLock

一对方法，分别获得读锁和写锁 Lock 对象。

ReentrantReadWriteLock类结构

ReentrantReadWriteLock是可重入的读写锁实现类。在它内部，维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 Writer 线程，读锁可以由多个 Reader 线程同时持有。也就是说，写锁是独占的，读锁是共享的。

如何使用读写锁

```
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
private Lock r = readWriteLock.readLock();
private Lock w = readWriteLock.writeLock();

// 读操作上读锁
public Data get(String key) {
  r.lock();
  try {
      // TODO 业务逻辑
  }finally {
       r.unlock();
   }
}

// 写操作上写锁
public Data put(String key, Data value) {
  w.lock();
  try {
      // TODO 业务逻辑
  }finally {
```

```
        w.unlock();
    }
}
```

注意事项

- 读锁不支持条件变量
- 重入时升级不支持：持有读锁的情况下去获取写锁，会导致获取永久等待
- 重入时支持降级： 持有写锁的情况下可以去获取读锁

2.2 应用场景

以下是使用ReentrantReadWriteLock的常见场景：

1. 读多写少：ReentrantReadWriteLock适用于读操作比写操作频繁的场景，因为它允许多个读线程同时访问共享数据，而写操作是独占的。
2. 缓存：ReentrantReadWriteLock可以用于实现缓存，因为它可以有效地处理大量的读操作，同时保护缓存数据的一致性。

读写锁在缓存中的应用

```
public class Cache {
    static Map<String, Object> map = new HashMap<String, Object>();
    static ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    static Lock r = rwl.readLock();
    static Lock w = rwl.writeLock();

    // 获取一个key对应的value
    public static final Object get(String key) {
        r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }

    // 设置key对应的value，并返回旧的value
    public static final Object put(String key, Object value) {
```

```
        w.lock();
        try {
            return map.put(key, value);
        } finally {
            w.unlock();
        }
    }

    // 清空所有的内容
    public static final void clear() {
        w.lock();
        try {
            map.clear();
        } finally {
            w.unlock();
        }
    }
}
```

上述示例中，Cache组合一个非线程安全的HashMap作为缓存的实现，同时使用读写锁的读锁和写锁来保证Cache是线程安全的。在读操作get(String key)方法中，需要获取读锁，这使得并发访问该方法时不会被阻塞。写操作put(String key,Object value)方法和clear()方法，在更新 HashMap时必须提前获取写锁，当获取写锁后，其他线程对于读锁和写锁的获取均被阻塞，而 只有写锁被释放之后，其他读写操作才能继续。Cache使用读写锁提升读操作的并发性，也保证每次写操作对所有的读写操作的可见性，同时简化了编程方式

2.3 锁降级

锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。锁降级可以帮助我们拿到当前线程修改后的结果而不被其他线程所破坏，防止更新丢失。

锁降级的使用示例

因为数据不常变化，所以多个线程可以并发地进行数据处理，当数据变更后，如果当前线程感知到数据变化，则进行数据的准备工作，同时其他处理线程被阻塞，直到当前线程完成数据的准备工作。

```
private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
private final Lock r = rwl.readLock();
```

```
private final Lock w = rwl.writeLock();
private volatile boolean update = false;

public void processData() {
    readLock.lock();
    if (!update) {
        // 必须先释放读锁
        readLock.unlock();
        // 锁降级从写锁获取到开始
        writeLock.lock();
        try {
            if (!update) {
                // TODO 准备数据的流程（略）
                update = true;
            }
            readLock.lock();
        } finally {
            writeLock.unlock();
        }
        // 锁降级完成，写锁降级为读锁
    }
    try {
        //TODO  使用数据的流程（略）
    } finally {
        readLock.unlock();
    }
}

```

锁降级中读锁的获取是否必要呢？答案是必要的。主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

RentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。目的也是保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

2.4 读写锁设计思路

读写状态的设计

设计的精髓：用一个变量如何维护多种状态

在 ReentrantLock 中，使用 Sync (实际是 AQS)的 int 类型的 state 来表示同步状态，表示锁被一个线程重复获取的次数。但是，读写锁 ReentrantReadWriteLock 内部维护着一对读写锁，如果要用一个变量维护多种状态，需要采用“按位切割使用”的方式来维护这个变量，将其切分为两部分：高16为表示读，低16为表示写。

分割之后，读写锁是如何迅速确定读锁和写锁的状态呢？通过位运算。假如当前同步状态为S，那么：

- 写状态，等于 S & 0x0000FFFF（将高 16 位全部抹去）。 当写状态加1，等于S+1.
- 读状态，等于 S >>> 16 (无符号补 0 右移 16 位)。当读状态加1，等于S+（1<<16）

根据状态的划分能得出一个推论：S不等于0时，当写状态（S&0x0000FFFF）等于0时，则读状态（S>>>16）大于0，即读锁已被获取。

代码实现：java.util.concurrent.locks.ReentrantReadWriteLock.Sync

- exclusiveCount(int c) 静态方法，获得持有写状态的锁的次数。
- sharedCount(int c) 静态方法，获得持有读状态的锁的数量。不同于写锁，读锁可以同时被多个线程持有。而每个线程持有的读锁支持重入的特性，所以需要对每个线程持有的读锁的数量单独计数，这就需要用到 HoldCounter 计数器

HoldCounter 计数器

读锁的内在机制其实就是一个共享锁。一次共享锁的操作就相当于对HoldCounter 计数器的操作。获取共享锁，则该计数器 + 1，释放共享锁，该计数器 - 1。只有当线程获取共享锁后才能对共享锁进行释放、重入操作。

通过 ThreadLocalHoldCounter 类，HoldCounter 与线程进行绑定。HoldCounter 是绑定线程的一个计数器，而 ThreadLocalHoldCounter 则是线程绑定的 ThreadLocal。

- HoldCounter是用来记录读锁重入数的对象
- ThreadLocalHoldCounter是ThreadLocal变量，用来存放不是第一个获取读锁的线程的其他线程的读锁重入数对象

美团面试三连

面试官：了解锁吗?

小明：了解，还经常用过。

面试官：说说synchronized和lock的区别吧

小明：synchronized是可重入锁，由于lock是一个接口，重入性取决于实现，synchronized不支持中断，而lock可以。。。。。。。。。。。。。。。。

面试官：好了，那有没有比这两种锁更快的锁呢？
小明：在读多写少的情况下，读写锁比他们的效率更高。
面试官：那有没有比读写锁更快的锁呢？
小明：。。。。。。。。。。

3. StampedLock介绍

如果我们深入分析ReentrantReadWriteLock，会发现它有个潜在的问题：如果有线程正在读，写线程需要等待读线程释放锁后才能获取写锁，即读的过程中不允许写，这是一种悲观的读锁。

为了进一步提升并发执行效率，Java 8引入了新的读写锁：StampedLock。

StampedLock和ReentrantReadWriteLock相比，改进之处在于：读的过程中也允许获取写锁后写入！在原先读写锁的基础上新增了一种叫乐观读（Optimistic Reading）的模式。该模式并不会加锁，所以不会阻塞线程，会有更高的吞吐量和更高的性能。

它的设计初衷是作为一个内部工具类，用于开发其他线程安全的组件，提升系统性能，并且编程模型也比ReentrantReadWriteLock 复杂，所以用不好就很容易出现死锁或者线程安全等莫名其妙的问题。

3.1 StampedLock的使用

StampLock三种访问模式

- **Writing（独占写锁）**：writeLock 方法会使线程阻塞等待独占访问，可类比ReentrantReadWriteLock 的写锁模式，同一时刻有且只有一个写线程获取锁资源；
- **Reading（悲观读锁）**：readLock方法，允许多个线程同时获取悲观读锁，悲观读锁与独占写锁互斥，与乐观读共享。
- **Optimistic Reading（乐观读）**：这里需要注意了，乐观读并没有加锁，也就是不会有 CAS 机制并且没有阻塞线程。仅当当前未处于 Writing 模式 tryOptimisticRead 才会返回非 0 的邮戳（Stamp），如果在获取乐观读之后没有出现写模式线程获取锁，则在方法validate返回 true ，允许多个线程获取乐观读以及读锁，同时允许一个写线程获取写锁。

在使用乐观读的时候一定要按照固定模板编写，否则很容易出 bug，我们总结下乐观读编程模型的模板：

```
1 public void optimisticRead() {
2     // 1. 非阻塞乐观读模式获取版本信息
3     long stamp = lock.tryOptimisticRead();
```

```
    // 2. 拷贝共享数据到线程本地栈中
    copyVaraibale2ThreadMemory();
    // 3. 校验乐观读模式读取的数据是否被修改过
    if (!lock.validate(stamp)) {
        // 3.1 校验未通过，上读锁
        stamp = lock.readLock();
        try {
            // 3.2 拷贝共享变量数据到局部变量
            copyVaraibale2ThreadMemory();
        } finally {
            // 释放读锁
            lock.unlockRead(stamp);
        }
    }
    // 3.3 校验通过，使用线程本地栈的数据进行逻辑操作
    useThreadMemoryVarables();
}
```

思考：为何 StampedLock 性比 ReentrantReadWriteLock 好?

关键在于StampedLock 提供的乐观读。ReentrantReadWriteLock 支持多个线程同时获取读锁，但是当多个线程同时读的时候，所有的写线程都是阻塞的。StampedLock 的乐观读允许一个写线程获取写锁，所以不会导致所有写线程阻塞，也就是当读多写少的时候，写线程有机会获取写锁，减少了线程饥饿的问题，吞吐量大大提高。

思考：允许多个乐观读和一个写线程同时进入临界资源操作，那读取的数据可能是错的怎么办?

乐观读不能保证读取到的数据是最新的，所以将数据读取到局部变量的时候需要通过lock.validate(stamp)校验是否被写线程修改过，若是修改过则需要上悲观读锁，再重新读取数据到局部变量。

演示乐观读

```
public class StampedLockTest{

    public static void main(String[] args) throws InterruptedException {
        Point point = new Point();

        //第一次移动x,y
        new Thread(()-> point.move(100,200)).start();
```

```
        Thread.sleep(100);
        new Thread(()-> point.distanceFromOrigin()).start();
        Thread.sleep(500);
        //第二次移动x,y
        new Thread(()-> point.move(300,400)).start();

    }
}

@Slf4j
class Point {
    private final StampedLock stampedLock = new StampedLock();

    private double x;
    private double y;

    public void move(double deltaX, double deltaY) {
        // 获取写锁
        long stamp = stampedLock.writeLock();
        log.debug("获取到writeLock");
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            // 释放写锁
            stampedLock.unlockWrite(stamp);
            log.debug("释放writeLock");
        }
    }

    public double distanceFromOrigin() {
        // 获得一个乐观读锁
        long stamp = stampedLock.tryOptimisticRead();
        // 注意下面两行代码不是原子操作
        // 假设x,y = (100,200)
        // 此处已读取到x=100，但x,y可能被写线程修改为(300,400)
        double currentX = x;
        log.debug("第1次读，x:{},y:{},currentX:{}",
                x,y,currentX);
        try {
```

```
48              Thread.sleep(2000);
49          } catch (InterruptedException e) {
50              e.printStackTrace();
51          }
52
53          // 此处已读取到y，如果没有写入，读取是正确的(100,200)
54          // 如果有写入，读取是错误的(100,400)
55          double currentY = y;
56          log.debug("第2次读，x:{},y:{},currentX:{},currentY:{}",
57                  x,y,currentX,currentY);
58
59          // 检查乐观读锁后是否有其他写锁发生
60          if (!stampedLock.validate(stamp)) {
61              // 获取一个悲观读锁
62              stamp = stampedLock.readLock();
63              try {
64                  currentX = x;
65                  currentY = y;
66
67                  log.debug("最终结果，x:{},y:{},currentX:{},currentY:{}",
68                          x,y,currentX,currentY);
69              } finally {
70                  // 释放悲观读锁
71                  stampedLock.unlockRead(stamp);
72              }
73          }
74          return Math.sqrt(currentX * currentX + currentY * currentY);
75      }
76
77  }
```

在缓存中的应用

将用户id与用户名数据保存在 共享变量 idMap 中，并且提供 put 方法添加数据、get 方法获取数据、以及 getIfNotExist 先从 map 中获取数据，若没有则模拟从数据库查询数据并放到 map 中。

```java
public class CacheStampedLock {
    /**
     * 共享变量数据
     */
    private final Map<Integer, String> idMap = new HashMap<>();
    private final StampedLock lock = new StampedLock();


    /**
     * 添加数据，独占模式
     */
    public void put(Integer key, String value) {
        long stamp = lock.writeLock();
        try {
            idMap.put(key, value);
        } finally {
            lock.unlockWrite(stamp);
        }
    }

    /**
     * 读取数据，只读方法
     */
    public String get(Integer key) {
        // 1. 尝试通过乐观读模式读取数据，非阻塞
        long stamp = lock.tryOptimisticRead();
        // 2. 读取数据到当前线程栈
        String currentValue = idMap.get(key);
        // 3. 校验是否被其他线程修改过,true 表示未修改，否则需要加悲观读锁
        if (!lock.validate(stamp)) {
            // 4. 上悲观读锁，并重新读取数据到当前线程局部变量
            stamp = lock.readLock();
            try {
                currentValue = idMap.get(key);
            } finally {
                lock.unlockRead(stamp);
            }
        }
```

```
        // 5. 若校验通过，则直接返回数据
        return currentValue;
    }

    /**
     * 如果数据不存在则从数据库读取添加到 map 中,锁升级运用
     * @param key
     * @return
     */
    public String getIfNotExist(Integer key) {
        // 获取读锁，也可以直接调用 get 方法使用乐观读
        long stamp = lock.readLock();
        String currentValue = idMap.get(key);
        // 缓存为空则尝试上写锁从数据库读取数据并写入缓存
        try {
            while (Objects.isNull(currentValue)) {
                // 尝试升级写锁
                long wl = lock.tryConvertToWriteLock(stamp);
                // 不为 0 升级写锁成功
                if (wl != 0L) {
                    stamp = wl;
                    // 模拟从数据库读取数据，写入缓存中
                    currentValue = "query db";
                    idMap.put(key, currentValue);
                    break;
                } else {
                    // 升级失败，释放之前加的读锁并上写锁，通过循环再试
                    lock.unlockRead(stamp);
                    stamp = lock.writeLock();
                }
            }
        } finally {
            // 释放最后加的锁
            lock.unlock(stamp);
        }
        return currentValue;
    }

}
```

上面的使用例子中，需要引起注意的是 get()和 getIfNotExist() 方法，第一个使用了乐观读，使得读写可以并发执行，第二个则是使用了读锁转换成写锁的编程模型，先查询缓存，当不存在的时候从数据库读取数据并添加到缓存中。

3.2 使用场景和注意事项

对于读多写少的高并发场景 StampedLock的性能很好，通过乐观读模式很好的解决了写线程“饥饿”的问题，我们可以使用StampedLock 来代替ReentrantReadWriteLock ，但是需要注意的是 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 写锁是不可重入的，如果当前线程已经获取了写锁，再次重复获取的话就会死锁，使用过程中一定要注意；
- 悲观读、写锁都不支持条件变量 Conditon ，当需要这个特性的时候需要注意；
- 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。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()。