ReentrantReadWriteLock的实现原理与锁获取

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：锁、锁降级、Lock接口、读锁与写锁

题目描述

ReentrantReadWriteLock的实现原理与锁获取?

1.面试题分析

在有些业务场景中，我们大多在读取数据，很少写入数据，这种情况下，如果仍使用独占锁，效率将及其低下。

针对这种情况，Java提供了读写锁——ReentrantReadWriteLock

有点类似MySQL数据库为代表的读写分离机制，既然我们知道了读写锁是用于读多写少的场景。那问题来了，ReentrantReadWriteLock是怎样来实现的呢，它与ReentrantLock的实现又有什么的区别呢?

2.ReentrantReadWriteLock简介

很多情况下有这样一种场景：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。

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，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的操作了。

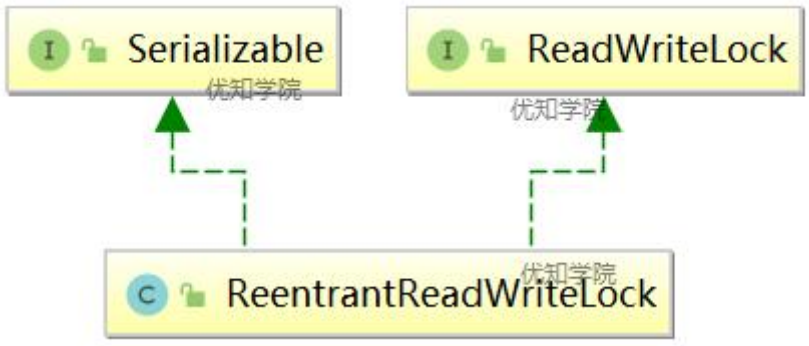
针对这种场景，JAVA的并发包提供了读写锁ReentrantReadWriteLock，它表示两个锁，一个是读操作相关的锁，称为共享锁；一个是写相关的锁，称为排他锁。

3.ReentrantReadWriteLock特性

- 公平性：读写锁支持非公平和公平的锁获取方式，非公平锁的吞吐量优于公平锁的吞吐量，默认构造的是非公平锁
- 可重入：在线程获取读锁之后能够再次获取读锁，但是不能获取写锁，而线程在获取写锁之后能够再次获取写锁，同时也能获取读锁
- 锁降级：线程获取写锁之后获取读锁，再释放写锁，这样实现了写锁变为读锁，也叫锁降级

4.ReentrantReadWriteLock的主要成员和结构图

1. ReentrantReadWriteLock的继承关系



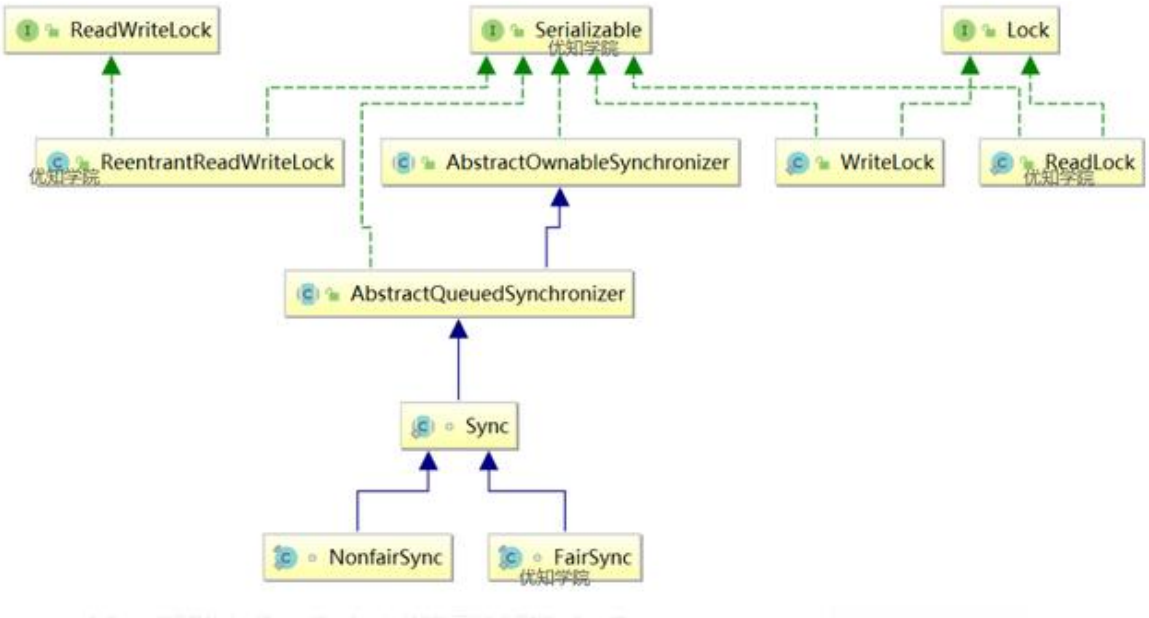

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     *
     * @return the lock used for reading.
     */
    Lock readLock();
    /**
     * Returns the lock used for writing.
     *
     * @return the lock used for writing.
     */
    Lock writeLock();
}
```

读写锁 ReadWriteLock

读写锁维护了一对相关的锁，一个用于只读操作，一个用于写入操作。

只要没有写入，读取锁可以由多个读线程同时保持,写入锁是独占的。

2.ReentrantReadWriteLock的核心变量



ReentrantReadWriteLock类包含三个核心变量：

1. ReaderLock：读锁,实现了Lock接口
2. WriterLock：写锁,也实现了Lock接口
3. Sync：继承自AbstractQueuedSynchronize(AQS),可以为公平锁FairSync 或 非公平锁 NonfairSync

3.ReentrantReadWriteLock的成员变量和构造函数

```
/** 内部提供的读锁 */

    private final ReentrantReadWriteLock.ReadLock readerLock;

    /** 内部提供的写锁 */
    private final ReentrantReadWriteLock.WriteLock writerLock;

    /** AQS来实现的同步器 */
    final Sync sync;

    /**
     * Creates a new {@code ReentrantReadWriteLock} with
     * 默认创建非公平的读写锁
     */
    public ReentrantReadWriteLock() {
        this(false);
    }

    /**
     * Creates a new {@code ReentrantReadWriteLock} with
     * the given fairness policy.
     *
     * @param fair {@code true} if this lock should use a fair ordering policy
     */
    public ReentrantReadWriteLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
        readerLock = new ReadLock(this);
        writerLock = new WriteLock(this);
    }
```

5.ReentrantReadWriteLock的核心实现

ReentrantReadWriteLock实现关键点，主要包括：

- 读写状态的设计
- 写锁的获取与释放
- 读锁的获取与释放
- 锁降级

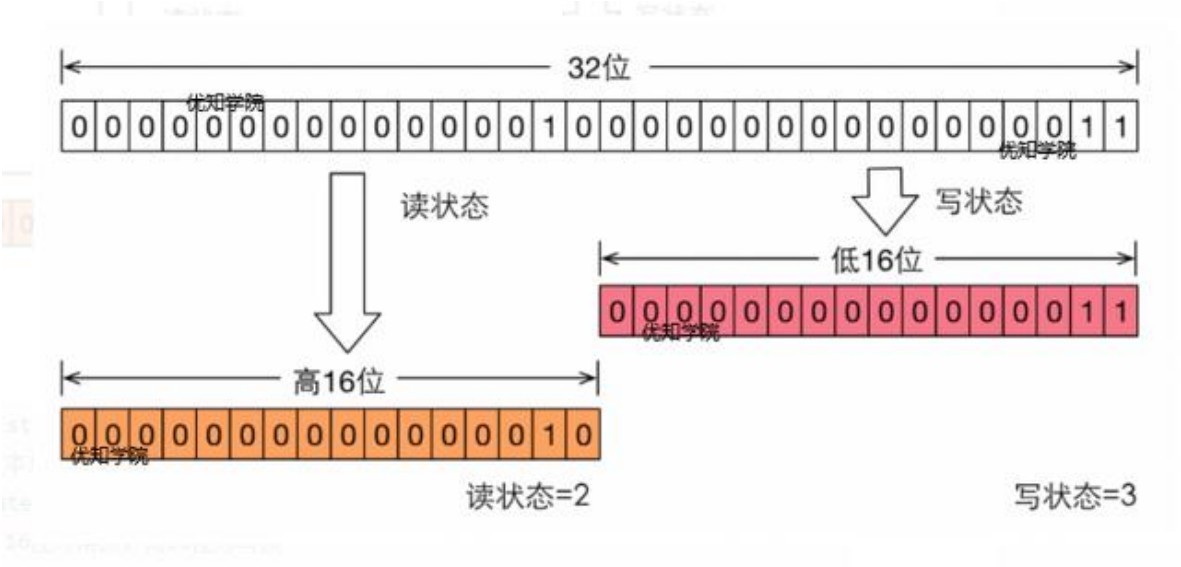
1.读写状态的设计

之前谈ReentrantLock的时候,Sync类是继承于AQS，主要以int state为线程锁状态,0表示没有被线程占用，1表示已经有线程占用。

同样ReentrantReadWriteLock也是继承于AQS来实现同步，那int state怎样同时来区分读锁和写锁的?

如果在一个整型变量上维护多种状态，就一定需要"按位切割使用"这个变量，ReentrantReadWriteLock将int类型的state将变量切割成两部分：

- 高16位记录读锁状态
- 低16位记录写锁状态




```
abstract static class Sync extends AbstractQueuedSynchronizer {
    // 版本序列号
    private static final long serialVersionUID = 6317671515068378041L;
    // 高16位为读锁，低16位为写锁
    static final int SHARED_SHIFT   = 16;
    // 读锁单位
    static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
    // 读锁最大数量
    static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
    // 写锁最大数量
    static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;
    // 本地线程计数器
    private transient ThreadLocalHoldCounter readHolds;
    // 缓存的计数器
    private transient HoldCounter cachedHoldCounter;
```

```
    // 第一个读线程
    private transient Thread firstReader = null;
    // 第一个读线程的计数
    private transient int firstReaderHoldCount;
}
```

2.写锁的获取与释放

```
protected final boolean tryAcquire(int acquires) {
            /*
             * Walkthrough:
             * 1. If read count nonzero or write count nonzero
             *    and owner is a different thread, fail.
             * 2. If count would saturate, fail. (This can only
             *    happen if count is already nonzero.)
             * 3. Otherwise, this thread is eligible for lock if
             *    it is either a reentrant acquire or
             *    queue policy allows it. If so, update state
             *    and set owner.
             */
            Thread current = Thread.currentThread();
            int c = getState();
            //获取独占锁(写锁)的被获取的数量
            int w = exclusiveCount(c);
            if (c != 0) {
                // (Note: if c != 0 and w == 0 then shared count != 0)
                //1.如果同步状态不为0，且写状态为0,则表示当前同步状态被读锁获取
                //2.或者当前拥有写锁的线程不是当前线程
                if (w == 0 || current != getExclusiveOwnerThread())
                    return false;
                if (w + exclusiveCount(acquires) > MAX_COUNT)
                    throw new Error("Maximum lock count exceeded");
                // Reentrant acquire
                setState(c + acquires);
                return true;
            }
            if (writerShouldBlock() ||
                !compareAndSetState(c, c + acquires))
                return false;
            setExclusiveOwnerThread(current);
            return true;
        }
```

1)c是获取当前锁状态,w是获取写锁的状态。

2)如果锁状态不为零，而写锁的状态为0，则表示读锁状态不为0，所以当前线程不能获取写锁。或者锁状态不为零，而写锁的状态也不为0，但是获取写锁的线程不是当前线程，则当前线程不能获取写锁。

3)写锁是一个可重入的排它锁，在获取同步状态时，增加了一个读锁是否存在的判断。

写锁的释放与ReentrantLock的释放过程类似，每次释放将写状态减1，直到写状态为0时，才表示该写锁被释放了。

3.读锁的获取与释放

```
protected final int tryAcquireShared(int unused) {
    for(;;) {
        int c = getState();
        int nextc = c + (1<<16);
        if(nextc < c) {
           throw new Error("Maxumum lock count exceeded");
        }
        if(exclusiveCount(c)!=0 && owner != Thread.currentThread())
           return -1;
        if(compareAndSetState(c,nextc))
           return 1;
    }
}
```

1)读锁是一个支持重进入的共享锁，可以被多个线程同时获取。

2)在没有写状态为0时，读锁总会被成功获取，而所做的也只是增加读状态（线程安全）

3)读状态是所有线程获取读锁次数的总和，而每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，由线程自身维护。

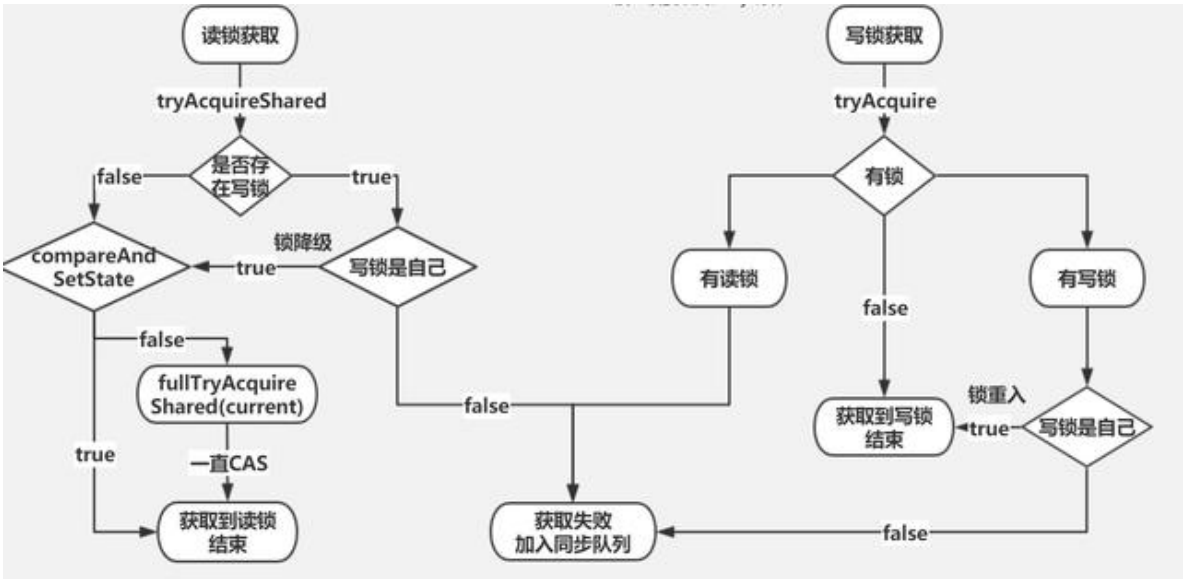
读锁的每次释放均减小状态（线程安全的，可能有多个读线程同时释放锁），减小的值是1<<16。

4.锁降级

降级是指当前把持住写锁，再获取到读锁，随后释放(先前拥有的)写锁的过程。

锁降级过程中的读锁的获取是否有必要，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直接释放写锁，假设此刻另一个线程获取的写锁，并修改了数据，那么当前线程就步伐感知到线程T的数据更新，如果当前线程遵循锁降级的步骤，那么线程T将会被阻塞，直到当前线程使数据并释放读锁之后，线程T才能获取写锁进行数据更新。

5.读锁与写锁的整体流程



6.ReentrantReadWriteLock总结

本篇详细介绍了ReentrantReadWriteLock的特征、实现、锁的获取过程，通过4个关键点的核心设计：

- 读写状态的设计
- 写锁的获取与释放
- 读锁的获取与释放
- 锁降级

从而才能实现：共享资源有读和写的操作，且写操作没有读操作那么频繁的应用场景。