读写锁ReentrantReadWriteLock

什么是读写锁

HashMap线程安全问题体现代码测试

使用读写锁解决HashMap解决线程安全问题测试代码

读写锁的实现原理源码分析

分析前我们先看看读写锁需要保存的状态

源码分析我们从ReentrantReadWriteLock对象的创建开始

```
1 public ReentrantReadWriteLock() {
2     this(false);
3 }
```

```
1 public ReentrantReadWriteLock(boolean fair) {
2     sync = fair ? new FairSync() : new NonfairSync();
3     readerLock = new ReadLock(this);
4     writerLock = new WriteLock(this);
5 }
```

一般我们不传入任何参数默认创建NonfairSync对象
在看获取读锁和写锁的过程

```
1 public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }
2 public ReentrantReadWriteLock.ReadLock  readLock()  { return readerLock; }
```

结合上边创建ReentrantReadWriteLock对象构造器中的代码，我们可以看出创建ReadLock和WriteLock的过程。
在看读锁加锁的过程
读锁为一个可重入的共享锁，它能够被多个线程同时持有，在没有其他写线程访问时，读锁总是获取成功
readLock.lock();
ReadLock类的代码如下所示，删除了全部的注释

```
1 public static class ReadLock implements Lock, java.io.Serializable {
2     private static final long serialVersionUID = -5992448646407690164L;
3     private final Sync sync;
4     protected ReadLock(ReentrantReadWriteLock lock) {
5         sync = lock.sync;
6     }
7     public void lock() {
8         sync.acquireShared(1);
9     }
10    public void lockInterruptibly() throws InterruptedException {
11        sync.acquireSharedInterruptibly(1);
12    }
13    public boolean tryLock() {
14        return sync.tryReadLock();
```

```
    }
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }
    public void unlock() {
        sync.releaseShared(1);
    }
    public Condition newCondition() {
        throw new UnsupportedOperationException();
    }
    public String toString() {
        int r = sync.getReadLockCount();
        return super.toString() +
            "[Read locks = " + r + "]";
    }
}
```

lock方法调用Sync的acquireShared(1)方法在Sync中没有该方法，在Sync的父类AbstractQueuedSynchronizer中找到了该方法

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

看看tryAcquireShared(arg)方法，如果获取成功返回>=0的值，不成功返回小于0的值
java.util.concurrent.locks.AbstractQueuedSynchronizer类中

```
protected int tryAcquireShared(int arg) {
    throw new UnsupportedOperationException();
}
```

只抛出了一个异常，并没有实质性的代码，我们在看AbstractQueuedSynchronizer的子类Sync中重写了该方法代码如下所示

```
 protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    int r = sharedCount(c);
    if (!readerShouldBlock() &&
        r < MAX_COUNT &&
        compareAndSetState(c, c + SHARED_UNIT)) {
        if (r == 0) {
            firstReader = current;
            firstReaderHoldCount = 1;
        } else if (firstReader == current) {
            firstReaderHoldCount++;
        } else {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != getThreadId(current))
                cachedHoldCounter = rh = readHolds.get();
            else if (rh.count == 0)
                readHolds.set(rh);
```

```
            rh.count++;
        }
        return 1;
    }
    return fullTryAcquireShared(current);
}
```

在分析该段代码之前需要先明白
需要关注当前线程写锁重入的次数、当前线程持有读锁的数量、当前线程读锁的重入次数
在ReentrantLock中使用一个int类型的state来表示同步状态，该值表示锁被一个线程重复获取的次数。但是读写锁ReentrantReadWriteLock内部维护着两个一对锁，需要用一个变量维护多种状态。所以读写锁采用"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？通过为运算。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），读状态等于S >>> 16(无符号补0右移16位)。代码如下：

```
static final int SHARED_SHIFT   = 16;
static final int SHARED_UNIT    = (1 << SHARED_SHIFT);
static final int MAX_COUNT      = (1 << SHARED_SHIFT) - 1;
static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

/** Returns the number of shared holds represented in count  */
static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
/** Returns the number of exclusive holds represented in count  */
static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

Thread current = Thread.currentThread();获取当前线程
int c = getState();获取线程的状态
exclusiveCount(c) != 0 判断独占锁也就是写锁的数量就是写锁的重入次数入股不为0表示现在有独占锁，在看第二个判断getExclusiveOwnerThread() != current表示锁的持有者不是当前线程，所以获取锁失败
int r = sharedCount(c);获取当前线程的共享锁也就是读锁的数量
readerShouldBlock()方法在类Sync代码是abstract boolean readerShouldBlock();是一个抽象方法，
在Sync的子类NonfairSync中找到具体的实现如下所示

```
final boolean readerShouldBlock() {
    /* As a heuristic to avoid indefinite writer starvation,
     * block if the thread that momentarily appears to be head
     * of queue, if one exists, is a waiting writer.  This is
     * only a probabilistic effect since a new reader will not
     * block if there is a waiting writer behind other enabled
     * readers that have not yet drained from the queue.
     */
    return apparentlyFirstQueuedIsExclusive();
}
```

主要是判断读锁是否需要等待，公平锁和非公平锁的判断
r < MAX_COUNT 读锁的重入次数不能大于65535
compareAndSetState(c, c + SHARED_UNIT)设置锁的状态
满足以上3个田间，读锁不等待，重入次数小于65535，设置锁的状态也成功之后，说明获取锁成功
接着看获取到锁之后的一系列判断，先看几个变量
firstReader、firstReaderHoldCount、cachedHoldCounter这三个变量分别表示第一个获取读锁的线程、firstReaderHoldCount为第一个获取读锁的重入数、cachedHoldCounter为HoldCounter的缓存
这里出现了HoldCounter类我们需要看一下

```
static final class HoldCounter {
    int count = 0;
```

```
    // Use id, not reference, to avoid garbage retention
    final long tid = getThreadId(Thread.currentThread());
}
```

HoldCounter的作用就是当前线程持有共享锁的数量，这个数量必须要与线程绑定在一起，否则操作其他线程锁就会抛出异常，有了该类任然不能和线程绑定，我们看到了

```
static final class ThreadLocalHoldCounter
    extends ThreadLocal<HoldCounter> {
    public HoldCounter initialValue() {
        return new HoldCounter();
    }
}
```

重新定义了ThreadLocalHoldCounter，返回了HoldCounter，这样HoldCounter就和每一个线程绑定在一起了，同时也保证了线程的安全性

```
if (r == 0) {
    firstReader = current;
    firstReaderHoldCount = 1;
}
```

获取读锁的线程为第一个线程，并且把线程锁的重入数设置为1

```
 else if (firstReader == current) {
    firstReaderHoldCount++;
}
```

获取读锁的线程重入了，把重入数加1

```
else {
    HoldCounter rh = cachedHoldCounter;
    if (rh == null || rh.tid != getThreadId(current))
        cachedHoldCounter = rh = readHolds.get();
    else if (rh.count == 0)
        readHolds.set(rh);
    rh.count++;
}
```

不是第一个获取读锁的线程所走的逻辑
先从缓存中获取到当前线程所持有锁的计数器HoldCounter，如果为null，需要创建一个HoldCounter，并且于当前的线程绑定，因此调用了readHolds.get()方法，并且放在了缓存中
如果获取锁失败走下边的逻辑

```
return fullTryAcquireShared(current);
```

我们看看fullTryAcquireShared的代码

```
final int fullTryAcquireShared(Thread current) {
    /*
     * This code is in part redundant with that in
     * tryAcquireShared but is simpler overall by not
     * complicating tryAcquireShared with interactions between
     * retries and lazily reading hold counts.
     */
    HoldCounter rh = null;
    for (;;) {
        int c = getState();
```

```
        if (exclusiveCount(c) != 0) {
            if (getExclusiveOwnerThread() != current)
                return -1;
            // else we hold the exclusive lock; blocking here
            // would cause deadlock.
        } else if (readerShouldBlock()) {
            // Make sure we're not acquiring read lock reentrantly
            if (firstReader == current) {
                // assert firstReaderHoldCount > 0;
            } else {
                if (rh == null) {
                    rh = cachedHoldCounter;
                    if (rh == null || rh.tid != getThreadId(current)) {
                        rh = readHolds.get();
                        if (rh.count == 0)
                            readHolds.remove();
                    }
                }
                if (rh.count == 0)
                    return -1;
            }
        }
        if (sharedCount(c) == MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        if (compareAndSetState(c, c + SHARED_UNIT)) {
            if (sharedCount(c) == 0) {
                firstReader = current;
                firstReaderHoldCount = 1;
            } else if (firstReader == current) {
                firstReaderHoldCount++;
            } else {
                if (rh == null)
                    rh = cachedHoldCounter;
                if (rh == null || rh.tid != getThreadId(current))
                    rh = readHolds.get();
                else if (rh.count == 0)
                    readHolds.set(rh);
                rh.count++;
                cachedHoldCounter = rh; // cache for release
            }
            return 1;
        }
    }
}
```

一个循环不断的获取锁，直到获取失败或者获取成功为止
接着来分析读锁的释放

```
public void unlock() {
    sync.releaseShared(1);
}
```

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
```

```
    return false;
}
```

```
protected boolean tryReleaseShared(int arg) {
    throw new UnsupportedOperationException();
}
```

```
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    if (firstReader == current) {
        // assert firstReaderHoldCount > 0;
        if (firstReaderHoldCount == 1)
            firstReader = null;
        else
            firstReaderHoldCount--;
    } else {
        HoldCounter rh = cachedHoldCounter;
        if (rh == null || rh.tid != getThreadId(current))
            rh = readHolds.get();
        int count = rh.count;
        if (count <= 1) {
            readHolds.remove();
            if (count <= 0)
                throw unmatchedUnlockException();
        }
        --rh.count;
    }
    for (;;) {
        int c = getState();
        int nextc = c - SHARED_UNIT;
        if (compareAndSetState(c, nextc))
            // Releasing the read lock has no effect on readers,
            // but it may allow waiting writers to proceed if
            // both read and write locks are now free.
            return nextc == 0;
    }
}
```

该段代码不做详细的解释了
我们继续分析写锁的加锁过程

```
public void lock() {
    sync.acquire(1);
}
```

调用sync的acquire方法

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

java.util.concurrent.locks.ReentrantReadWriteLock.Sync类中实现了tryAcquire(arg)方法

```
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();
    int w = exclusiveCount(c);
    if (c != 0) {
        // (Note: if c != 0 and w == 0 then shared count != 0)
        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;
}
```

我们逐行来分析分析
Thread current = Thread.currentThread(); 获取当前线程
int c = getState();获取当前线程的状态
int w = exclusiveCount(c);获取写锁的重入次数
if (c != 0)如果该条件成立，说明当前线程的读锁不是第一次获取，也有可能是读线程进来获取锁了
c != 0 && w == 0 表示存在读锁
current != getExclusiveOwnerThread()当前线程不是已经获取写锁的线程
直接返回false
if (w + exclusiveCount(acquires) > MAX_COUNT)锁的重入次数不能大于65535
setState(c + acquires);设置读锁的状态
获取锁成功返回true

if (writerShouldBlock() || !compareAndSetState(c, c + acquires))return false;
第一个判断writerShouldBlock()我们看它的代码在NonfairSync类中

```
final boolean writerShouldBlock() {
    return false; // writers can always barge
}
```

我们看到永远返回的是false，这是一个非公平的锁，这个判断是是否需要等到
!compareAndSetState(c, c + acquires)
如果第一次进来需要通过CAS设置锁的状态，状态设置成功返回true在取反为false，所以不执行return false,继续往下执行
setExclusiveOwnerThread(current); 把当前线程存储起来，返回true,获取锁成功

看写锁的释放

```
public void unlock() {
    sync.release(1);
}
```

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

写锁的释放最终还是会调用AQS的模板方法release(int arg)方法，该方法首先调用tryRelease(int arg)方法尝试释放锁，tryRelease(int arg)方法为读写锁内部类Sync中定义了，如下：

```
 protected final boolean tryRelease(int releases) {
    //释放的线程不为锁的持有者
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    int nextc = getState() - releases;
    //若写锁的新线程数为0，则将锁的持有者设置为null
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
    setState(nextc);
    return free;
}
```

写锁释放锁的整个过程和独占锁ReentrantLock相似，每次释放均是减少写状态，当写状态为0时表示 写锁已经完全释放了，从而等待的其他线程可以继续访问读写锁，获取同步状态，同时此次写线程的修改对后续的线程可见

锁降级

1. 锁降级是指将写锁降级为读锁
2. 在写锁没有释放的时候，获取到读锁，在释放写锁
3. 看一段代码

```
public class LockLower {
    private Map<String ,Object> map ;
    private CountDownLatch countDownLatch;
    private ReadWriteLock readWriteLock ;
    private Lock readLock ;
    private Lock writeLock ;
    private volatile boolean isUpdate = false;
    public void wirteLockToReadLock(){
        readLock.lock();
        if (isUpdate){
            readLock.unlock();
            writeLock.lock();
            map.put("","");
            //锁降级
            readLock.lock();
            writeLock.unlock();
        }
        Object obj =  map.get("");
```

```
19          System.out.println(obj);
20      }
21      public LockLower(Map<String ,Object> map, CountDownLatch countDownLatch, ReadWriteLock
    readWriteLock) {
22          this.map = map;
23          this.countDownLatch = countDownLatch;
24          this.readWriteLock = readWriteLock;
25          readLock = readWriteLock.readLock();
26          writeLock = readWriteLock.writeLock();
27      }
28  }
```

锁升级

1. 把读锁升级为写锁
2. 在读锁没有释放的时候，获取到写锁，在释放读锁

在ReentrantReadWriteLock中不支持锁升级，因为，读锁和写锁互斥的，在读锁没有释放的时候不能获取到写锁