1 Lock 接口分析

Method	Descript
void lock()	获取锁(获取不到一直等待)
boolean tryLock()	获取锁(获取不到返回 false)
boolean tryLock(long,unit)	获取锁(指定时间内获取不到返回 false)
void lockInterruptibly()	获取锁(当线程被终止时，退出等待)
void unlock()	释放锁
Condition new Condition()	与 Lock 配合使用，提供多个等待集合，更精确的控制，底层是 park/unpark 机制。同一个 lock 可以有多个 condition

- Condition 实现类由 AQS 提供了，具体源码逻辑留待以后分析。。

2 AQS 抽象类结构分析

- AQS 实现了一个等待队列，用来装没有获取到锁的线程
- AQS 只实现了 **没有获取锁、成功获取锁、没有释放锁、成功释放锁** 后的逻辑(**如何加入等待队列，如何从等待队列唤醒节点**)，具体的怎样获取，怎样释放，由子类实现
- 因为 获取共享锁和获取独占锁 后的逻辑不太一样，AQS 需要分别为它们实现了相应的逻辑
- 因为公平锁、非公平锁只和 **如何获取锁的逻辑** 有关，对 AQS 来说是透明的，所以 AQS 不需要额外区分
- 子类 ReentrantLock：实现了公平独占锁、非公平独占锁
- 子类 ReentrantReadWriteLock：实现了公平(独占锁/共享锁)、非公平(独占锁/共享锁)
- 这里只简单的介绍 AQS 中的方法，具体的源码逻辑留待以后分析

```java
public abstract class AbstractQueuedSynchronizer extends AbstractOwnableSynchronizer {
    //等待队列节点类，通过这个类组成了一个等待队列
    //等待队列是 CLH 锁队列的一种变体，CLH 锁常用于自旋锁。而我们却用来实现阻塞锁，通过使用相同的策略：在节点的前节点的线程中保存一些控制信息
    //每个节点的 status 字段用来跟踪一个线程是否应该被阻塞。但一个节点的前一个节点被释放时，该节点将被通知。队列的每个节点都充当一个特定通知样式的监视器，其中包含一个等待的线程。status 字段并不控制线程是否被授予锁。
    //线程可能会尝试获取它是否在队列中的第一个。但是队列头部的线程不一定能获取锁成功，也可能重新等待
    static final class Node{...}
    //队列的头部
    private transient volatile Node head;
    //队列的尾部
    private transient volatile Node tail;
    //锁的状态，分为高16和低16两把锁
    private volatile int state;
    //自旋 1000 纳秒
    static final long spinForTimeoutThreshold = 1000L;

    //---------------------- 独占模式下，获取锁失败后，如何进入队列(acquireAueued())中的逻辑

    //独占不响应中断模式下，获取锁失败后，如何加入队列
    public final void acquire(int arg){...}
    //独占响应中断模式下，获取锁失败后，如何加入队列
    public final void acquireInterruptibly(int arg){...}
    //独占有时间限制模式，获取锁失败后，如何加入队列
    public final boolean tryAcquireNanos(int arg, long nanosTimeout){...}
    //独占模式释放锁，释放成功后，如何通知等待队列
    public final boolean release(int arg){...}

    //被 acquire 调用，如果被中断返回 true
    final boolean acquireQueued(final Node node, int arg){...}
    //被 acquireInterruptibly 调用
    private void doAcquireInterruptibly(int arg){...}
    //被 tryAcquireNanos 调用，规定时间内获取失败返回 false，成功返回 true
    private boolean doAcquireNanos(int arg, long nanosTimeout){...}


    //---------------------- 共享模式下，获取锁失败后，如何进入队列中的逻辑

    //共享不响应中断模式下，获取锁失败后，如何加入队列
    public final void acquireShared(int arg){...}
    //共享响应中断模式下，获取锁失败后，如何加入队列
    public final void acquireSharedInterruptibly(int arg){...}
    //共享有时间限制模式下，获取锁失败后，如何加入队列
    public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout){}
    //共享模式释下，释放锁成功后，如何通知等待队列
    public final boolean releaseShared(int arg)

    //---------------------- 被上面的代码调用 ---------------
    private void doAcquireShared(int arg){...}
    private void doAcquireSharedInterruptibly(int arg){...}
    private boolean doAcquireSharedNanos(int arg, long nanosTimeout){...}
    private void doReleaseShared(){...}
```

```
    //---------------------- 真正获取锁的逻辑---------由子类自定义
    //尝试获取独占锁，需要具体子类实现
    protected boolean tryAcquire(int arg)；
    //尝试释放独占锁，需要具体子类实现
    protected boolean tryRelease(int arg);
    //尝试获取共享锁，需要具体子类实现
    protected int tryAcquireShared(int arg);
    //尝试释放共享锁，需要具体子类实现
    protected boolean tryReleaseShared(int arg);


    //---------------------------等待队列辅助方法

    //插入节点到队列中
    private Node enq(final Node node){...}
    //通过指定的 node 状态(SHARED/EXCLUSIVE)和当前线程，创建一个 node，并加入队列(里面调用了 enq
方法)
    private Node addWaiter(Node mode){...}
    //取消该节点获取锁
    private void cancelAcquire(Node node){...}
    //唤醒后继节点
    private void unparkSuccessor(Node node){...}
    //检查并且更新没有获取锁成功的节点的 status，返回 true 则说明线程需要 阻塞
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node){...}
    //挂起当前线程，返回线程中断状态，并清除中断状态
    private final boolean parkAndCheckInterrupt(){...}

    //Condition 实现类，基本所有的 condition 都使用的是它！
    public class ConditionObject implements Condition, java.io.Serializable{...}
}
```

3 AQS 子类 —— ReentrantLock 结构分析

- ReentrantLock 实现了**获取/释放独占锁**的逻辑，和 AQS 结合就形成了完整的锁

```
//实现了 Lock 接口，客户只能调用 Lock 接口中的方法
public class ReentrantLock implements Lock, java.io.Serializable {
    //内部实现了公平锁和非公平锁
    private final Sync sync;
    abstract static class Sync extends AbstractQueuedSynchronizer{...}
    static final class NonfairSync extends Sync{...}
    static final class FairSync extends Sync{...}

    //--------------------- Lock 接口中的方法，供客户调用-----------
    //--------------------- 内部统一调用 sync 类中的方法，内有公平锁、非公平锁供选择-----------
    //获取锁,不响应中断
    public void lock(){...}
    //获取锁，线程被中断，就会释放锁(如果持有锁)，并抛异常
    public void lockInterruptibly(){...}
    //尝试获取锁，失败返回 false，不响应中断
    public boolean tryLock(){...}
```

```
    //规定时间内尝试获取锁，失败返回 false，响应中断
    public boolean tryLock(long timeout, TimeUnit unit){...}
    //释放锁
    public void unlock(){...}
}
```

4 AQS 子类 —— ReentrantReadWriteLock 结构分析

```
public class ReentrantReadWriteLock implements ReadWriteLock, java.io.Serializable {
    //----------------- 实现了共享锁、独占锁，这两把锁共存
    private final ReentrantReadWriteLock.ReadLock readerLock;
    private final ReentrantReadWriteLock.WriteLock writerLock;
    public static class ReadLock implements Lock, java.io.Serializable{...}
    public static class WriteLock implements Lock, java.io.Serializable{...}


    //----------------- 实现了公平锁、非公平锁，只能指定一个
    //-----------------
    final Sync sync;
    abstract static class Sync extends AbstractQueuedSynchronizer {}
    //很有意思的是，ReentrantReadWriteLock 将公平锁、非公平锁的区别仅仅归纳到了两个方法！
    //即，公平独占锁和非公平独占锁、公平共享锁和非公平共享锁，之间的区别，都仅仅是两个方法的区别！
    static final class NonfairSync extends Sync{...}
    static final class FairSync extends Sync{...}
}
```

5 ReentrantLock 源码分析

- `Lock lock0 = new ReentrantLock()` -- 非公平锁
- `Lock lock1 = new ReentrantLock(true)` -- 公平锁

5.1 非公平锁的获取与释放

- 调用 `lock0.lock()` 获取非公平锁
- 首先 CAS 将独占锁设置为当前线程，不成功再走 AQS 进入阻塞队列流程
- AQS 流程：首先调用 子类的 tryAcquire() 操作，尝试获取锁，不成功则进入等待队列
- 在释放锁之前可以先判断下获取锁的线程是否是当前线程，避免走异常流程

```
public void lock() {sync.lock();}
final void lock() {
    //直接 CAS，成功则将独占锁标志设为当前线程
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        //失败，走 AQS 加入等待队列逻辑，AQS 中也进行了几次 tryAcquire() 尝试获取锁，仍不成功，才
进入等待队列的
        acquire(1);
}
// AQS 中又先尝试了 tryAcquire() 操作获取锁，不成功则加入等待队列
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
```

```
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    //如果 state 为 0，说明没有线程获取锁，再一次 CAS 设值
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //如果已有线程持有锁，判断是否是当前线程，进行重入锁操作
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //没有获取锁，则返回 false 尝试加入等待队列
    return false;
}
```

- 调用 `lock0.unlock()` 释放锁

```
//直接走 AQS 中的 release() 流程
public void unlock() {sync.release(1);}
// AQS
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            //释放锁成功，且头结点满足一点条件，则唤醒头结点的下一个节点
            unparkSuccessor(h);
        return true;
    }
    //释放不成功返回 false
    return false;
}

protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    //如果当前线程不是独占锁线程，抛锁状态异常，所以在释放锁之前可以先判断下，避免走异常流程
    //什么情况下会出现这种状况呢？响应中断的锁被中断了，如果在 finally 块中执行了 release 方法，就会触发这个异常！！可以对中断锁单独的 try/catch/finally 处理，这样就不用在 unlock() 方法中，每次都判断是否是当前线程了
    //不响应中断的锁，走到这步不可能触发这个异常，除非乱写，走来就 unlock()。。
```

```
    //所以，所有的 unlock() 方法都先判断是否是获取锁的线程，是最稳妥的！！等需要优化的时候在说嘛，不要过分设计
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

5.2 公平锁的获取与释放

- 调用 `lock1.lock()` 获取公平锁
- 和公平锁的区别在于
  - 非公平锁当前线程先直接尝试 CAS 获取独占锁，对于早已进入等待队列的线程来说，就不公平了
  - 并且非公平锁在 tryAquire() 时，也是先直接 CAS 的
  - 公平锁则直接走 AQS 流程，而且 tryAquire() 的时候还要判断等待队列中有没有线程在它前面，所以所有的线程是按顺序进，按顺序出的，很公平
  - 综上：区别在于 lock() 和 tryAcquire() 内部代码的不同
- 公平锁和非公平锁的**释放都是一样的流程**，并没有区别

```
public void lock() {sync.lock();}
//公平锁的 lock() 操作，直接走 AQS 流程！
final void lock() {
    acquire(1);
}
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //先判断等待队列中有没有线程在它前面！
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

6 ReentrantReadWriteLock 源码分析

- `ReadWriteLock lock = new ReadWriteLock()` -- 获取非公平读写锁

6.1 非公平共享锁的获取与释放

- 调用 `lock.readLock().lock()`，获取读锁
  - 如果独占锁被别的线程持有，则获取失败
  - 如果等待队列第一个不是写线程，则尝试 CAS，成功就获取成功
  - 否则走 fullTryAcquireShared() 流程，直到成功或失败为止
- 这里有两个关键点：
  - 独占锁被别的线程获取，则走 AQS 流程
  - 等待队列第一个是写线程的话，如果当前线程**不是重入锁**，就乖乖走 AQS 流程
  - 关键：**这样才能保证写线程不会一直被读线程阻塞，可以想象写线程很容易就成为第一个等待队列节点的**
- 如果读线程没有获取到锁，且等待队列第一个不是写线程，则读线程是不会加入等待队列的哦

```
public void lock() {
    sync.acquireShared(1);
}
//走 AQS 流程
public final void acquireShared(int arg) {
    //走子类 tryAcquireShared() 操作尝试获取锁，获取失败，加入等待队列
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    //如果独占锁已被别的线程持有，则获取锁失败
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;
    int r = sharedCount(c);// 状态变量 state 的高16位表示读线程的数量！
    //判断等待队列的头一个节点是不是写线程，如果不是且读线程数小于最大(因为16位的限制)的，并且 CAS 成功，获取锁成功
    if (!readerShouldBlock() &&
        r < MAX_COUNT &&
        compareAndSetState(c, c + SHARED_UNIT)) {//cas 将读锁的数量加1
        //关键点1：因为 上面的 cas 保证了只有一个线程走 r==0 的流程
        //关键点2：仔细观察，会发现 firstReader/firstReaderHoldCount 这两个变量只可能被第一个获取读锁的人访问到！！所以根本不需要同步的！！
        if (r == 0) {
            firstReader = current;
            firstReaderHoldCount = 1;
        } else if (firstReader == current) {
            firstReaderHoldCount++;
        } else {
            //这里使用了 ThreadLocal 思想，避免了同步，，
            HoldCounter rh = cachedHoldCounter;
            //rh.tid != getThreadId(current)：这里保证了每个线程使用的 cacheHoldCounter 都是自己独有的，有意思，用这个方法 + threadLocal 保证了 全局变量的线程独有特性！！！
            if (rh == null || rh.tid != getThreadId(current))
```

```
                //走到这里说明读取到了别的线程的值，所以要获取当前线程的值
                cachedHoldCounter = rh = readHolds.get();
            else if (rh.count == 0)
                readHolds.set(rh);//确保设置了 threadId
            rh.count++;//每个线程的重入次数！这里没有在 else 语句中哦
            //这里我有个疑问：如果当前线程的 rh 值在工作内存中丢失了，如果这时去主内存取的话，那不就
要重置次数了吗？
        }
        return 1;
    }
    //如果以上捷径没成功的话，走这里
    return fullTryAcquireShared(current);
}
//一直循环到，写锁被别的线程持有或等待队列第一个是写锁，first reader 不是自己且不是重入。返回 -1
//CAS 成功返回 1
//即如果是重入获取读锁，则要一直循环到写锁被别的线程持有或自己 CAS 成功才会返回
//不是重入锁，直接返回
final int fullTryAcquireShared(Thread current) {
    HoldCounter rh = null;
    for (;;) {//这个 for 循环一直到底
        int c = getState();
        if (exclusiveCount(c) != 0) {
            //如果写锁已被别的线程持有，还是直接失败
            if (getExclusiveOwnerThread() != current)
                return -1;
            // else we hold the exclusive lock; blocking here
            // would cause deadlock.
        } else if (readerShouldBlock()) {
            //等待队列的第一个是写线程
            // Make sure we're not acquiring read lock reentrantly
            if (firstReader == current) {
                // assert firstReaderHoldCount > 0;
            } else {
                //且写线程不是自己
                if (rh == null) {
                    rh = cachedHoldCounter;
                    if (rh == null || rh.tid != getThreadId(current)) {
                        //获取自己的 cacheHoldCounter
                        rh = readHolds.get();
                        //这是干啥
                        if (rh.count == 0)
                            readHolds.remove();
                    }
                }
                //不是重入锁直接返回
                if (rh.count == 0)
                    return -1;
            }
        }
        if (sharedCount(c) == MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        //再一次尝试 CAS，这个 if 到底
        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;
        }
    }
}
```

- 调用 `lock.readLock().unlock()`，释放读锁
- 走重入流程，且只有所有的读线程都释放了，才算读锁释放了！

```
public void unlock() {sync.releaseShared(1);}
//走 AQS 流程
public final boolean releaseShared(int arg) {
    //走子类 tryReleaseShared() 操作
    if (tryReleaseShared(arg)) {
        //成功后再走 AQS 流程
        doReleaseShared();
        return true;
    }
    return false;
}
protected final boolean tryReleaseShared(int unused) {
    Thread current = Thread.currentThread();
    //重入锁逻辑
    if (firstReader == current) {
        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;
        }
        //只有所有的读线程都释放了，才算读锁释放了！
        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;
        }
    }
```

6.1 非公平独占锁的获取与释放

- 调用 `lock.writeLock().lock()`，获取写锁
  - 如果既没有读线程也没有写线程获取到锁，则直接 CAS，成功则获取，不成功则走 AQS 流程
  - **如果读锁不为0，直接失败**，走 AQS 流程
  - 如果读锁为0，写线程不是当前线程，则失败。如果是，则重入

```
public void lock() {sync.acquire(1);}
//走 AQS 流程
public final void acquire(int arg) {
    //走子类 tryAcquire() 流程
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
protected final boolean tryAcquire(int acquires) {
    Thread current = Thread.currentThread();
    int c = getState();
    int w = exclusiveCount(c);
    //c 不为0，说明有读线程或写线程成功获取锁了
    if (c != 0) {
        // (Note: if c != 0 and w == 0 then shared count != 0)
        //写锁等于0，说明读锁不为0，直接失败
        //如果写锁不等于0，说明读锁为0。如果其它线程获取到了写锁，则直接失败
        if (w == 0 || current != getExclusiveOwnerThread())
            return false;
        //当前线程获取了读锁，则重入，返回成功
        if (w + exclusiveCount(acquires) > MAX_COUNT)
            throw new Error("Maximum lock count exceeded");
        // Reentrant acquire
        setState(c + acquires);
        return true;
    }
    //非公平锁，第一个方法始终返回false
    //如果 c==0，则直接 CAS，成功则获取成功，失败走 AQS 流程
    if (writerShouldBlock() ||
        !compareAndSetState(c, c + acquires))
```

```
            return false;
        setExclusiveOwnerThread(current);
        return true;
    }
```

- 调用 `lock.writeLock().unlock()` ，获取写锁

```
public void unlock() {sync.release(1);}
//走 AQS 流程，和重入锁一样
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
protected final boolean tryRelease(int releases) {
    //不是当前线程持有写锁，抛异常
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    //重入为 0，才释放写锁
    int nextc = getState() - releases;
    boolean free = exclusiveCount(nextc) == 0;
    if (free)
        setExclusiveOwnerThread(null);
    setState(nextc);
    return free;
}
```

7 lock、tryLock()、lockInterruptibly() 的区别

- lock()：线程不获取到锁会一直阻塞，并且线程不响应中断
- tryLock()：线程尝试获取锁，获取不成功返回false，线程不会阻塞，并且不会响应中断
- lockInterruptibly()：线程不获取到锁会被阻塞，但是会响应中断，以下源码分析如何响应中断的
- 调用 `lock.lockInterruptibly()` 获取可响应中断的锁
- 可以看出，响应中断与否，是在 AQS 中实现的，子类并不需要实现

```
Lock lock = new ReentrantLock();
lock.lockInterruptibly();
public void lockInterruptibly() throws InterruptedException {
    sync.acquireInterruptibly(1);
}
public final void acquireInterruptibly(int arg)
    throws InterruptedException {
    //如果线程处于中断状态，清除中断状态并抛异常
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        //这里是关键的代码
```

```
        doAcquireInterruptibly(arg);
}
private void doAcquireInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                //线程走到这里，说明被中断了（parkAndCheckInterrupt 返回 true 了）
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
//关键：当线程被中断时，线程会从 park() 中醒来！！
//这时因为线程处于中断状态， Thread.interrupted() == true !!
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

8 Lock 与 Synchronized 的区别

8.1 Synchronized 的优缺点

优点：

- 使用简单：语义清晰，且由虚拟机来释放锁，不需要人为操作
- 由 JVM 提供，会持续不断地进行优化，目前已经提供了多种优化方案(锁粗化、锁消除、轻量级锁、偏向锁)

缺点：

- 功能单一(使用简单带来的副作用)，无法实现一些锁的高级特性：公平锁、中断锁、超时锁、读写锁

8.2 Lock 的优缺点

优点：

- 由 JDK 提供，可以实现很多高级特性（见 Synchronized 的缺点）
- 可以实现自定义锁(通过继承 AQS)

缺点：

- 使用复杂(功能高级导致)，需要手动释放锁，操作不当容易产生死锁

- 因为是 JDK 提供，持续的优化力度可能没有 Synchronized 大