ReentrantLock 源码分析

说明

1. 本文基于 jdk 8 写作。
2. @author JellyfishMIX - github / blog.jellyfishmix.com
3. LICENSE GPL-2.0

锁机制的核心: Sync（锁）

```
    /**
     * Synchronizer providing all implementation mechanics
     * 提供所有实现机制的同步器，这是 ReentrantLock 能实现锁机制的核心
     */
    private final Sync sync;

    /**
     * Base of synchronization control for this lock. Subclassed
     * into fair and nonfair versions below. Uses AQS state to
     * represent the number of holds on the lock.
     */
    abstract static class Sync extends AbstractQueuedSynchronizer
```

ReentrantLock 的 lock 与 unlock 方法，是通过 Sync 实现的，Sync 是一个 ReentrantLock 的内部类，继承了 AbstractQueuedSynchronizer（简称 AQS）。Sync 是 ReentrantLock 能实现锁机制的核心。

AQS 是一个 java 语言层面的同步器开发框架，对于 AbstractQueuedSynchronizer 的源码分析日后完善，这里先把 AbstractQueuedSynchronizer 当作黑盒处理，只使用它的 api。

Sync 的 state 属性（继承自父类）

Sync 的 state 属性继承自父类 AbstractQueuedSynchronizer，AbstractQueuedSynchronizer 中 state 的定义：

```
    /**
     * The synchronization state.
     */
    private volatile int state;
```

对于 Sync，可以理解为这是一个锁的 state 数量。在独占锁中，加锁的时候 state 会 +1（当然可以自己修改这个值），在解锁的时候 -1。同一个锁，在持有锁的线程重入后，state 可能会被叠加为 2, 3, 4... 等等。只有 unlock() 的次数与 lock() 的次数对应，才会将持有锁的线程设置为空，只有这种情况下解锁时的 tryRelease 方法才会返回 true。

公平锁和非公平锁(FairSync, NonfairSync)

公平锁：每个线程抢占锁的顺序为先后调用 lock 方法的顺序依次获取锁，类似于排队吃饭。

非公平锁：每个线程抢占锁的顺序不定，谁运气好，谁就获取到锁，和调用 lock 方法的先后顺序无关。

```
    /**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     * 默认的无参构造方法，实例化一个非公平锁
     */
    public ReentrantLock() {
        sync = new NonfairSync();
    }

    /**
     * Creates an instance of {@code ReentrantLock} with the
     * given fairness policy.
     * 传入 true 为公平锁，传入 false 为非公平锁
     *
     * @param fair {@code true} if this lock should use a fair ordering policy
     */
    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

公平锁与非公平锁是 Sync 的两个实现：NonfairSync 和 FairSync

```
    /**
     * Sync object for non-fair locks
     * 非公平锁：每个线程抢占锁的顺序不定，谁运气好，谁就获取到锁，和调用 lock 方法的先后顺序无关
     */
    static final class NonfairSync extends Sync

    /**
     * Sync object for fair locks
     * 公平锁：每个线程抢占锁的顺序为先后调用 lock 方法的顺序依次获取锁，类似于排队吃饭。
     */
    static final class FairSync extends Sync
```

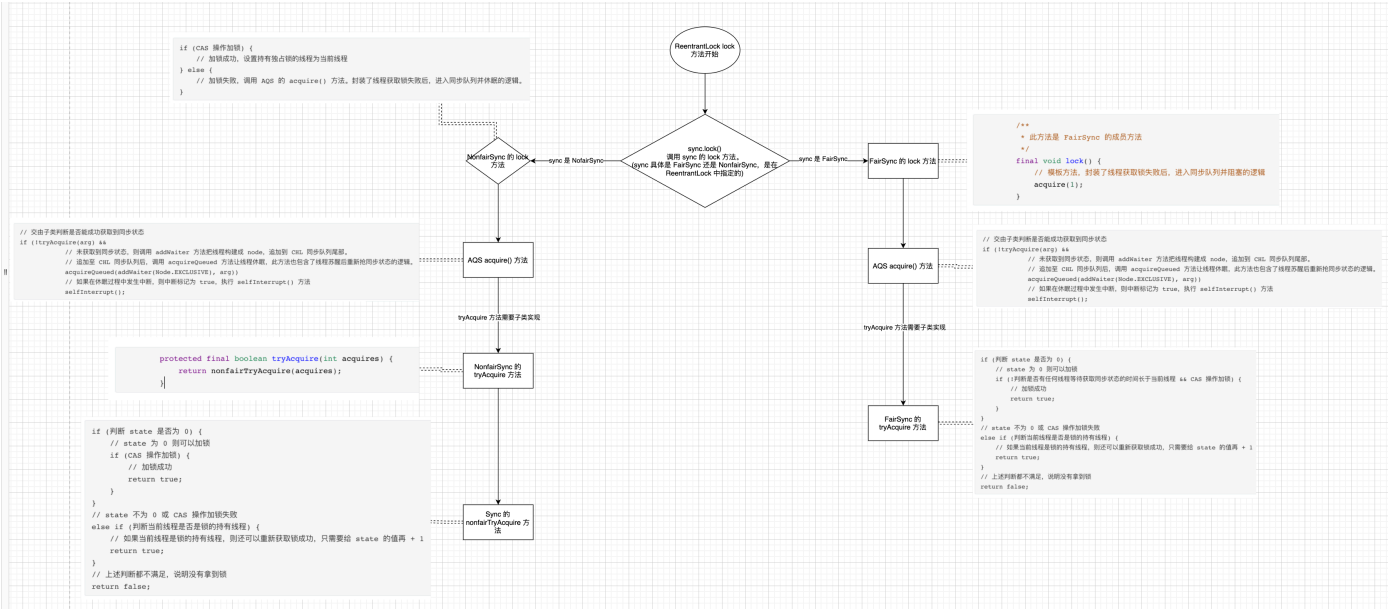
ReentrantLock 的 lock 方法

```
    /**
     * Acquires the lock.
     * 此方法是 ReentrantLock 的成员方法
     *
```

```
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately, setting the lock hold count to one.
     *
     * <p>If the current thread already holds the lock then the hold
     * count is incremented by one and the method returns immediately.
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until the lock has been acquired,
     * at which time the lock hold count is set to one.
     */
    public void lock() {
        // 调用 sync 的 lock 方法加锁
        sync.lock();
    }
```

ReentrantLock 中的 lock 方法直接调用了 sync 的 lock 方法加锁。

lock 方法流程图：



Sync 中的 lock 方法

```
    /**
     * Performs {@link Lock#lock}. The main reason for subclassing
     * is to allow fast path for nonfair version.
     *
     * Sync 的成员方法。由子类实现 lock 方法。
     */
    abstract void lock();
```

Sync 的 lock 方法由子类实现，子类有两个：NonfairSync 非公平锁，和 FairSync 公平锁。公平与非公平的体现，就体现在对 lock 方法的实现上。也就是说，这里 sync 是公平锁还是非公平锁，是在构造 ReentrantLock 的时候，通过构造方法指定的。

NonfairSync 的 lock 方法（非公平锁）

```
        /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         *
         * 加锁。此方法是 NonfairSync 的成员方法。
         */
        final void lock() {
            // 先尝试进行一次 CAS 原子操作加锁
            if (compareAndSetState(0, 1))
                // 加锁成功，设置持有独占锁的线程为当前线程
                setExclusiveOwnerThread(Thread.currentThread());
            else
                // 模板方法，封装了线程获取锁失败后，进入同步队列并阻塞的逻辑
                acquire(1);
        }
```

lock 方法先调用了一次 CAS 原子操作加锁。CAS（Compare-And-Swap），它是一条 cpu 原语，用于判断内存中某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的。由于是 cpu 原语，所以天然不会有并发竟态风险。在 lock 方法中，它尝试从 0 设置为 1，表示尝试加锁。能设置成功，则说明拿到了锁。因为如果多个线程同时进行 CAS 操作，只有一个线程能成功从 0 设置成 1，这也是从 cpu 层面保证的。

如果 CAS 加锁失败，则会调用 acquire 方法，acquire 方法继承自 AbstractQueuedSynchronizer。

总结一下逻辑：

```
if (CAS 操作加锁) {
    // 加锁成功，设置持有独占锁的线程为当前线程
} else {
    // 加锁失败，调用 AQS 的 acquire() 方法。封装了线程获取锁失败后，进入同步队列并休眠的逻辑。
}
```

AbstractQueuedSynchronizer 的 acquire 方法（模板方法）

```
    /**
     * Acquires in exclusive mode, ignoring interrupts.  Implemented
     * by invoking at least once {@link #tryAcquire},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquire} until success.  This method can be used
     * to implement method {@link Lock#lock}.
     *
     * 模板方法，封装了线程获取锁失败后，进入同步队列并阻塞的逻辑
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
```

```
     */
    public final void acquire(int arg) {
        // tryAcquire 方法由子类实现
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

AbstractQueuedSynchronizer 的 acquire 方法是一个模板方法。模板方法的定义：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

我们先不需要关心 acquire 模板方法的其他逻辑，站在本文主角 ReentrantLock 的角度，只需要关注自己要实现的 tryAcquire 方法即可。AbstractQueuedSynchronizer 中对 tryAcquire 方法的声明如下：

```
    /**
     * Attempts to acquire in exclusive mode. This method should query
     * if the state of the object permits it to be acquired in the
     * exclusive mode, and if so to acquire it.
     *
     * 尝试在独占模式下获取锁时，是否允许获取锁，由具体子类实现
     *
     * <p>This method is always invoked by the thread performing
     * acquire.  If this method reports failure, the acquire method
     * may queue the thread, if it is not already queued, until it is
     * signalled by a release from some other thread. This can be used
     * to implement method {@link Lock#tryLock()}.
     *
     * <p>The default
     * implementation throws {@link UnsupportedOperationException}.
     *
     * @param arg the acquire argument. This value is always the one
     *        passed to an acquire method, or is the value saved on entry
     *        to a condition wait.  The value is otherwise uninterpreted
     *        and can represent anything you like.
     * @return {@code true} if successful. Upon success, this object has
     *         been acquired.
     * @throws IllegalMonitorStateException if acquiring would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if exclusive mode is not supported
     */
    protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }
```

在 ReentrantLock 中，内部类 NonfairSync 实现了 tryAcquire 方法。

NonfairSync 的 tryAcquire 方法

```
        protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);
        }
```

NonfairSync 调用了父类 Sync 的 nonfairTryAcquire 方法。

Sync 的 nonfairTryAcquire 方法

```
        /**
         * Performs non-fair tryLock.  tryAcquire is implemented in
         * subclasses, but both need nonfair try for trylock method.
         *
         * 非公平获取锁
         */
        final boolean nonfairTryAcquire(int acquires) {
            // 拿到当前线程
            final Thread current = Thread.currentThread();
            // 当前锁的 state
            int c = getState();
            // 如果当前锁的 state 为 0，即锁空闲
            if (c == 0) {
                // CAS 原子操作加锁，尝试把 0 设置成指定值
                if (compareAndSetState(0, acquires)) {
                    // 加锁成功，设置持有锁的线程为当前线程
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            // 如果当前锁的 state 不为 0，但是当前线程是持有锁的线程，则允许重入锁，这里也体现了
ReentrantLock 是可重入锁
            else if (current == getExclusiveOwnerThread()) {
                // 重入次数
                int nextc = c + acquires;
                // 如果重入次数 < 0（重入次数溢出数据类型范围）
                if (nextc < 0) // overflow
                    // 抛出 Error，提示超出了锁的最大计数
                    throw new Error("Maximum lock count exceeded");
                // 设置锁的 state
                setState(nextc);
                // 成功拿到锁
                return true;
            }
            // 如果如上逻辑都不满足，则没有拿到锁
            return false;
        }
```

简单来讲，nonfairTryAcquire 方法通过判断 state 是否为 0，来判断是否可以获取同步状态。如果 state 不为 0 则不可以获取同步状态。当然有一个例外，如果尝试获取同步状态的线程，就是已经持有锁的线程，则可以成功获取同步状态，同时 state + acquires。这里也体现了 ReentrantLock 是可重入锁。

总结一下逻辑：

```
if (判断 state 是否为 0) {
    // state 为 0 则可以加锁
    if (CAS 操作加锁) {
        // 加锁成功
        return true;
    }
}
// state 不为 0 或 CAS 操作加锁失败
else if (判断当前线程是否是锁的持有线程) {
    // 如果当前线程是锁的持有线程，则还可以重新获取锁成功，只需要给 state 的值再 + 1
    return true;
}
// 上述判断都不满足，说明没有拿到锁
return false;
```

FairSync 的 lock 方法（公平锁）

```
        /**
         * 此方法是 FairSync 的成员方法
         */
        final void lock() {
            // 模板方法，封装了线程获取锁失败后，进入同步队列并阻塞的逻辑
            acquire(1);
        }
```

公平锁的 lock 逻辑很简单，acquire(1); 在非公平锁中已经解释过了。继承自 AbstractQueuedSynchronizer 的 acquire 方法，前面已经分析过。

需要注意的是，acquire 作为模版方法调用的 tryAcquire 方法，对于 FairSync，这个 tryAcquire 方法与 NonfairSync 是不同的。主要是在 if 调用 CAS 原子操作加锁时，FairSync 多了一步 hasQueuedPredecessors，判断是否有任何线程等待获取的时间长于当前线程，如果没有比当前线程等待时间更长的，再去 CAS 加锁，这也是公平锁的体现。

FairSync 的 tryAcquire 方法

```
        /**
         * Fair version of tryAcquire.  Don't grant access unless
         * recursive call or no waiters or is first.
         *
         * 此方法是 FairSync 的成员方法
         */
        protected final boolean tryAcquire(int acquires) {
```

```
            // 获取当前线程
            final Thread current = Thread.currentThread();
            // 当前锁的 state
            int c = getState();
            // 如果当前锁的 state 为 0，即锁空闲
            if (c == 0) {
                // 判断是否有任何线程等待获取的时间长于当前线程
                if (!hasQueuedPredecessors() &&
                        // CAS 原子操作加锁，尝试把 0 设置成指定值
                    compareAndSetState(0, acquires)) {
                    // 加锁成功，设置持有锁的线程为当前线程
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            // 如果当前锁的 state 不为 0，但是当前线程是持有锁的线程，则允许重入锁，这里也体现了
ReentrantLock 是可重入锁
            else if (current == getExclusiveOwnerThread()) {
                // 重入次数
                int nextc = c + acquires;
                // 如果重入次数 < 0（重入次数溢出数据类型范围）
                if (nextc < 0)
                    // 抛出 Error，提示超出了锁的最大计数
                    throw new Error("Maximum lock count exceeded");
                // 设置锁的 state
                setState(nextc);
                // 成功拿到锁
                return true;
            }
            // 如果如上逻辑都不满足，则没有拿到锁
            return false;
        }
```

简单来讲，FairSync 的 tryAcquire 方法通过判断 state 是否为 0，来判断是否可以获取同步状态。如果 state 不为 0 则不可以获取同步状态。当然有一个例外，如果尝试获取同步状态的线程，就是已经持有锁的线程，则可以成功获取同步状态，同时 state + acquires。这里也体现了 ReentrantLock 是可重入锁。

总结一下逻辑：

```
if (判断 state 是否为 0) {
    // state 为 0 则可以加锁
    if (!判断是否有任何线程等待获取同步状态的时间长于当前线程 && CAS 操作加锁) {
        // 加锁成功
        return true;
    }
}
// state 不为 0 或 CAS 操作加锁失败
else if (判断当前线程是否是锁的持有线程) {
    // 如果当前线程是锁的持有线程，则还可以重新获取锁成功，只需要给 state 的值再 + 1
```

```
    return true;
}
// 上述判断都不满足，说明没有拿到锁
return false;
```

FairSync 公平锁与 NonfairSync 非公平锁 lock 方法的区别

下面对比一下非公平锁与公平锁 lock 方法实现的区别，lock 方法也是公平锁与非公平锁的全部区别。

```
        /**
         * NonfairSync 非公平锁的 lock 方法
         */
        final void lock() {
            // 先尝试进行一次 CAS 原子操作加锁
            if (compareAndSetState(0, 1))
                // 加锁成功，设置持有独占锁的线程为当前线程
                setExclusiveOwnerThread(Thread.currentThread());
            else
                // 模板方法，封装了线程获取锁失败后，进入同步队列并阻塞的逻辑
                acquire(1);
        }

        /**
         * FairSync 公平锁的 lock 方法
         */
        final void lock() {
            // 模板方法，封装了线程获取锁失败后，进入同步队列并阻塞的逻辑
            acquire(1);
        }
```

可以看到，NonfairSync 在 lock 时，比 FairSync 多了一步，先尝试进行一次 CAS 原子操作加锁，这就体现出了不公平。当前线程直接就参加了抢锁，而不是按照线程间调用 lock 的顺序，先来后到进行抢锁。

lock 方法调用了父类 AbstractQueuedSynchronizer 的模版方法 acquire，acquire 作为模版方法调用的 tryAcquire 方法。对于 FairSync，这个 tryAcquire 方法与 NonfairSync 是不同的。

```
                // NonfairSync
                // CAS 原子操作加锁，尝试把 0 设置成指定值
                if (compareAndSetState(0, acquires)) {
                    // 加锁成功，设置持有锁的线程为当前线程
                    setExclusiveOwnerThread(current);
                    return true;
                }

                // FairSync
                // 判断是否有任何线程排队等待获取早于当前线程
                if (!hasQueuedPredecessors() &&
                        // CAS 原子操作加锁，尝试把 0 设置成指定值
```

```
                    compareAndSetState(0, acquires)) {
                    // 加锁成功，设置持有锁的线程为当前线程
                    setExclusiveOwnerThread(current);
                    return true;
                }
```

不同点上文也介绍了，再重复一下，主要是在 if 调用 CAS 原子操作加锁时，FairSync 多了一步 hasQueuedPredecessors，判断是否有任何线程排队等待早于当前线程，如果没有比当前线程排队等待获取更早的，再去 CAS 加锁，这也是公平锁的体现。

ReentrantLock 的 unlock 方法

解锁方法，如下：

```
    /**
     * Attempts to release this lock.
     * 解锁。此方法是 ReentrantLock 的成员方法
     *
     * <p>If the current thread is the holder of this lock then the hold
     * count is decremented.  If the hold count is now zero then the lock
     * is released.  If the current thread is not the holder of this
     * lock then {@link IllegalMonitorStateException} is thrown.
     *
     * @throws IllegalMonitorStateException if the current thread does not
     *         hold this lock
     */
    public void unlock() {
        // 调用 sync 的 release 方法
        sync.release(1);
    }
```

sync 的 release 方法继承自 AbstractQueuedSynchronizer。

AbstractQueuedSynchronizer 的 release 方法（模板方法）

```
    /**
     * Releases in exclusive mode.  Implemented by unblocking one or
     * more threads if {@link #tryRelease} returns true.
     * This method can be used to implement method {@link Lock#unlock}.
     *
     * 模版方法，其中 tryRelease 方法由子类实现。当前锁释放指定的 state 数量，并返回锁是否释放成功。
     *
     * @param arg the release argument.  This value is conveyed to
     *        {@link #tryRelease} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @return the value returned from {@link #tryRelease}
     */
    public final boolean release(int arg) {
```

```
        // tryRelease 方法由子类实现
        if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0)
                unparkSuccessor(h);
            return true;
        }
        return false;
    }
```

AbstractQueuedSynchronizer 的 release方法是一个模板方法。模板方法的定义：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

我们先不需要关心 release 模板方法的其他逻辑，站在本文主角 ReentrantLock 的角度，只需要关注自己要实现的 tryRelease 方法即可。AbstractQueuedSynchronizer 中对 tryRelease 方法的声明如下：

```
    /**
     * Attempts to set the state to reflect a release in exclusive
     * mode.
     *
     * 尝试设置同步器的 state 值，来反映一次在独占模式中的 release(释放) 操作。是否允许释放锁，由具体子类实现。
     *
     * <p>This method is always invoked by the thread performing release.
     *
     * <p>The default implementation throws
     * {@link UnsupportedOperationException}.
     *
     * @param arg the release argument. This value is always the one
     *        passed to a release method, or the current state value upon
     *        entry to a condition wait.  The value is otherwise
     *        uninterpreted and can represent anything you like.
     * @return {@code true} if this object is now in a fully released
     *         state, so that any waiting threads may attempt to acquire;
     *         and {@code false} otherwise.
     * @throws IllegalMonitorStateException if releasing would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if exclusive mode is not supported
     */
    protected boolean tryRelease(int arg) {
        throw new UnsupportedOperationException();
    }
```

在 ReentrantLock 中，内部类 Sync 实现了 tryRelease 方法。

Sync 的 tryRelease 方法

```
        /**
         * 可以认为是一个设置锁状态的操作，通过将锁当前的 state 数量，减掉传入的要释放的 state 数量(比如参数可以是 1)
         * 如果结果 state 为 0，就将持有独占锁的线程设置为 null，即释放锁，以使其它的线程有机会拿锁。
         * 此方法是 Sync 的成员方法
         *
         * @param releases 要释放的 state 数量
         * @return
         */
        protected final boolean tryRelease(int releases) {
            // 结果 state = (锁当前的 state 数量) - (要释放的 state 数量)
            int c = getState() - releases;
            // 如果当前线程不是持有锁的线程，抛出异常
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            // 表示锁是否被释放
            boolean free = false;
            // 如果结果 state 为 0，说明持有锁的线程已经完全释放
            if (c == 0) {
                free = true;
                // 设置持有独占锁的线程为 null，即释放锁
                setExclusiveOwnerThread(null);
            }
            // 设置锁的 state 为计算后结果
            setState(c);
            return free;
        }
```

假设方法入参 int releases 为 1。可以简单理解为，给同步状态 state - 1。-1 计算完成后，如果 state 为 0，就判断为成功释放同步状态。