JUC锁: ReentrantLock详解

可重入锁ReentrantLock的底层是通过AbstractQueuedSynchronizer实现

面试问题去理解

- 什么是可重入，什么是可重入锁? 它用来解决什么问题?
- ReentrantLock的核心是AQS，那么它怎么来实现的，继承吗? 说说其类内部结构关系。
- ReentrantLock是如何实现公平锁的?
- ReentrantLock是如何实现非公平锁的?
- ReentrantLock默认实现的是公平还是非公平锁?
- 使用ReentrantLock实现公平和非公平锁的示例?
- ReentrantLock和Synchronized的对比?

ReentrantLock源码分析

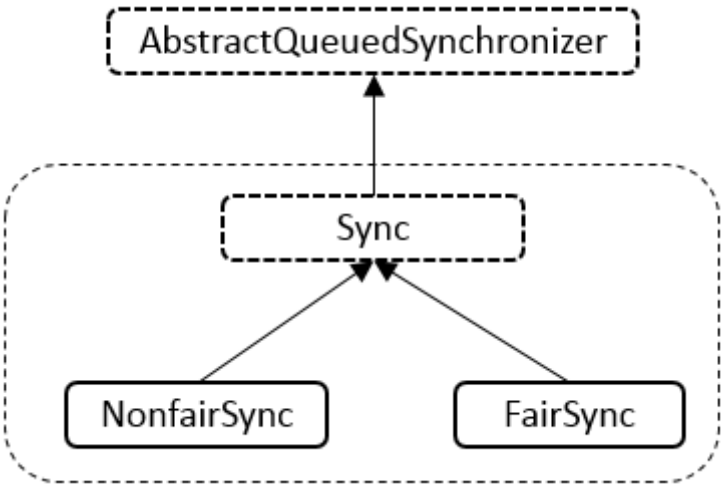
类的继承关系

ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件。

```
public class ReentrantLock implements Lock, java.io.Serializabl
```

类的内部类

ReentrantLock总共有三个内部类，并且三个内部类是紧密相关的，下面先看三个类的关系。



说明: ReentrantLock类内部总共存在Sync、NonfairSync、FairSync三个类，NonfairSync与FairSync类继承自Sync类，Sync类继承自AbstractQueuedSynchronizer抽象类。下面逐个进行分析。

- Sync类

Sync类的源码如下:

```
abstract static class Sync extends AbstractQueuedSynchronizer {
    // 序列号
    private static final long serialVersionUID = -5179523762034025860L;

    // 获取锁
    abstract void lock();

    // 非公平方式获取
    final boolean nonfairTryAcquire(int acquires) {
        // 当前线程
        final Thread current = Thread.currentThread();
        // 获取状态
        int c = getState();
        if (c == 0) { // 表示没有线程正在竞争该锁
            if (compareAndSetState(0, acquires)) { // 比较并设置状态成功，状态0表示锁没有被占用
                // 设置当前线程独占
                setExclusiveOwnerThread(current);
                return true; // 成功
            }
        }
        else if (current == getExclusiveOwnerThread()) { // 当前线程拥有该锁
            int nextc = c + acquires; // 增加重入次数
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            // 设置状态
            setState(nextc);
            // 成功
            return true;
        }
        // 失败
        return false;
    }

    // 试图在共享模式下获取对象状态，此方法应该查询是否允许它在共享模式下获取对象状态，如果允许，则获取它
    protected final boolean tryRelease(int releases) {
        int c = getState() - releases;
        if (Thread.currentThread() != getExclusiveOwnerThread()) // 当前线程不为独占线程
            throw new IllegalMonitorStateException(); // 抛出异常
        // 释放标识
        boolean free = false;
        if (c == 0) {
            free = true;
            // 已经释放，清空独占
            setExclusiveOwnerThread(null);
        }
        // 设置标识
        setState(c);
        return free;
    }

    // 判断资源是否被当前线程占有
    protected final boolean isHeldExclusively() {
        // While we must in general read state before owner,
        // we don't need to do so to check if current thread is owner
```

```
        return getExclusiveOwnerThread() == Thread.currentThread();
    }

    // 新生一个条件
    final ConditionObject newCondition() {
        return new ConditionObject();
    }

    // Methods relayed from outer class
    // 返回资源的占用线程
    final Thread getOwner() {
        return getState() == 0 ? null : getExclusiveOwnerThread();
    }
    // 返回状态
    final int getHoldCount() {
        return isHeldExclusively() ? getState() : 0;
    }

    // 资源是否被占用
    final boolean isLocked() {
        return getState() != 0;
    }

    /**
        * Reconstitutes the instance from a stream (that is, deserializes it).
        */
    // 自定义反序列化逻辑
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();
        setState(0); // reset to unlocked state
    }
}
```

Sync类存在如下方法和作用如下。

方法	作用
lock	锁定，并为实现，留给具体子类实现
nonfairTryAcquire	非公平方式获取
tryRelease	试图在共享模式下获取对象状态，此方法应该查询是否允许它在共享模式下获取对象状态，如果允许，则获取它
isHeldExclusively	判断资源是否被当前线程占有
newCondition	新生一个条件
getOwner	返回占有资源的线程
getHoldCount	返回状态
isLocked	资源是否被占用
readObject	自定义反序列化逻辑

- NonfairSync类

NonfairSync类继承了Sync类，表示采用非公平策略获取锁，其实现了Sync类中抽象的lock方法，源码如下:

```
// 非公平锁
static final class NonfairSync extends Sync {
    // 版本号
    private static final long serialVersionUID = 7316153563782823691L;
```

```
    // 获得锁
    final void lock() {
        if (compareAndSetState(0, 1)) // 比较并设置状态成功，状态0表示锁没有被占用
            // 把当前线程设置独占了锁
            setExclusiveOwnerThread(Thread.currentThread());
        else // 锁已经被占用，或者set失败
            // 以独占模式获取对象，忽略中断
            acquire(1);
    }

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

说明: 从lock方法的源码可知，每一次都尝试获取锁，而并不会按照公平等待的原则进行等待，让等待时间最久的线程获得锁。

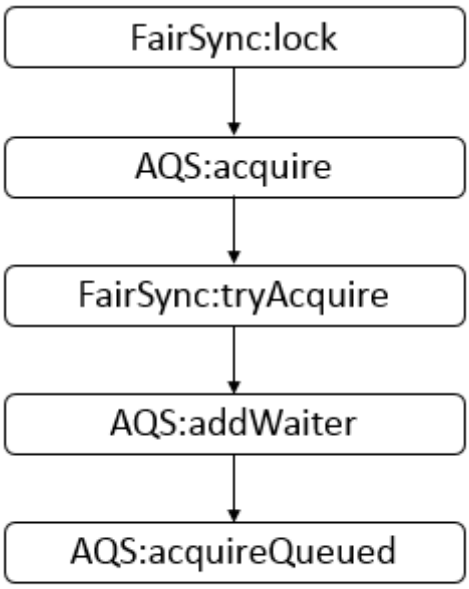
- FairSyn类

FairSync类也继承了Sync类，表示采用公平策略获取锁，其实现了Sync类中的抽象lock方法，源码如下:

```
// 公平锁
static final class FairSync extends Sync {
    // 版本序列化
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() {
        // 以独占模式获取对象，忽略中断
        acquire(1);
    }

    /**
        * Fair version of tryAcquire.  Don't grant access unless
        * recursive call or no waiters or is first.
        */
    // 尝试公平获取锁
    protected final boolean tryAcquire(int acquires) {
        // 获取当前线程
        final Thread current = Thread.currentThread();
        // 获取状态
        int c = getState();
        if (c == 0) { // 状态为0
            if (!hasQueuedPredecessors() &&
                compareAndSetState(0, acquires)) { // 不存在已经等待更久的线程并且比较并且设置状态成
功
                // 设置当前线程独占
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) { // 状态不为0，即资源已经被线程占据
            // 下一个状态
            int nextc = c + acquires;
            if (nextc < 0) // 超过了int的表示范围
                throw new Error("Maximum lock count exceeded");
            // 设置状态
            setState(nextc);
            return true;
```

```
        }
        return false;
    }
}
```

说明: 跟踪lock方法的源码可知，当资源空闲时，它总是会先判断sync队列(AbstractQueuedSynchronizer中的数据结构)是否有等待时间更长的线程，如果存在，则将该线程加入到等待队列的尾部，实现了公平获取原则。其中，FairSync类的lock的方法调用如下，只给出了主要的方法。



当资源已被占用调用情况

说明: 可以看出只要资源被其他线程占用，该线程就会添加到sync queue中的尾部，而不会先尝试获取资源。这也是和Nonfair最大的区别，Nonfair每一次都会尝试去获取资源，如果此时该资源恰好被释放，则会被当前线程获取，这就造成了不公平的现象，当获取不成功，再加入队列尾部。

类的属性

ReentrantLock 类的 sync 非常重要，对 ReentrantLock 类的操作大部分都直接转化为对 Sync 和 AbstractQueuedSynchronizer类的操作。

```
public class ReentrantLock implements Lock, java.io.Serializable {
    // 序列号
    private static final long serialVersionUID = 7373984872572414699L;
    // 同步队列
    private final Sync sync;
}
```

类的构造函数

- ReentrantLock()型构造函数

默认是采用的非公平策略获取锁

```
public ReentrantLock() {
    // 默认非公平策略
    sync = new NonfairSync();
}
```

- ReentrantLock(boolean)型构造函数

可以传递参数确定采用公平策略或者是非公平策略，参数为true表示公平策略，否则，采用非公平策略:

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

核心函数分析

通过分析ReentrantLock的源码，可知对其操作都转化为对Sync对象的操作，由于Sync继承了AQS，所以基本上都可以转化为对AQS的操作。如将ReentrantLock的lock函数转化为对Sync的lock函数的调用，而具体会根据采用的策略(如公平策略或者非公平策略)的不同而调用到Sync的不同子类。

所以可知，在ReentrantLock的背后，是AQS对其服务提供了支持，由于之前我们分析AQS的核心源码，遂不再累赘。下面还是通过例子来更进一步分析源码。

示例分析

公平锁

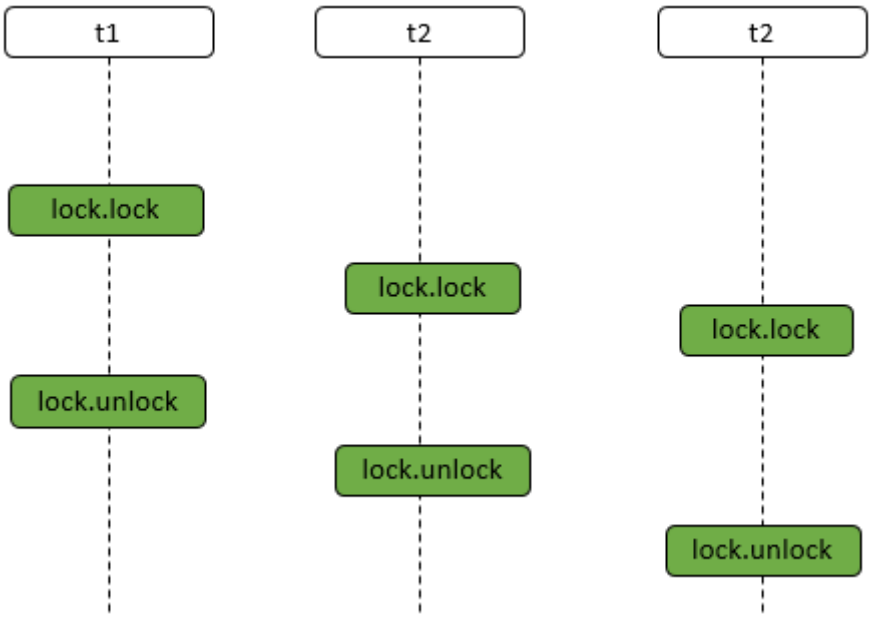
```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class MyThread extends Thread {
    private Lock lock;
    public MyThread(String name, Lock lock) {
        super(name);
        this.lock = lock;
    }

    public void run () {
        lock.lock();
        try {
            System.out.println(Thread.currentThread() + " running");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } finally {
```

```
            lock.unlock();
        }
    }
}

public class AbstractQueuedSynchonizerDemo {
    public static void main(String[] args) throws InterruptedException {
        Lock lock = new ReentrantLock(true);

        MyThread t1 = new MyThread("t1", lock);
        MyThread t2 = new MyThread("t2", lock);
        MyThread t3 = new MyThread("t3", lock);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

运行结果(某一次):

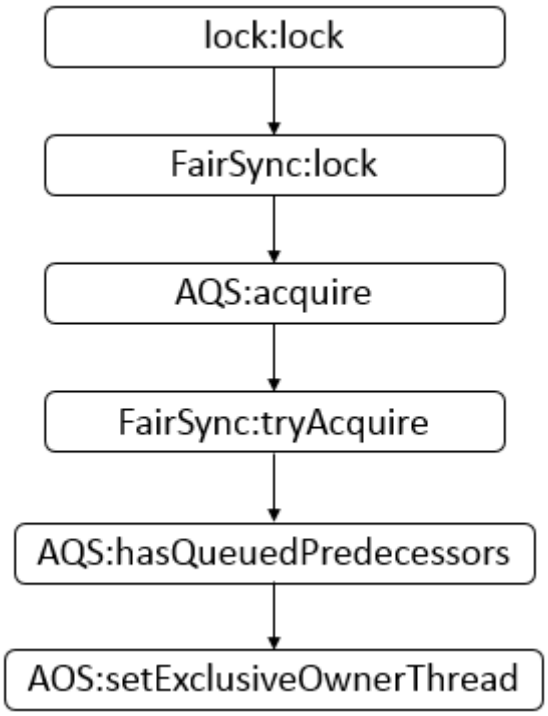
```
Thread[t1,5,main] running
Thread[t2,5,main] running
Thread[t3,5,main] running
```

说明: 该示例使用的是公平策略，由结果可知，可能会存在如下一种时序。



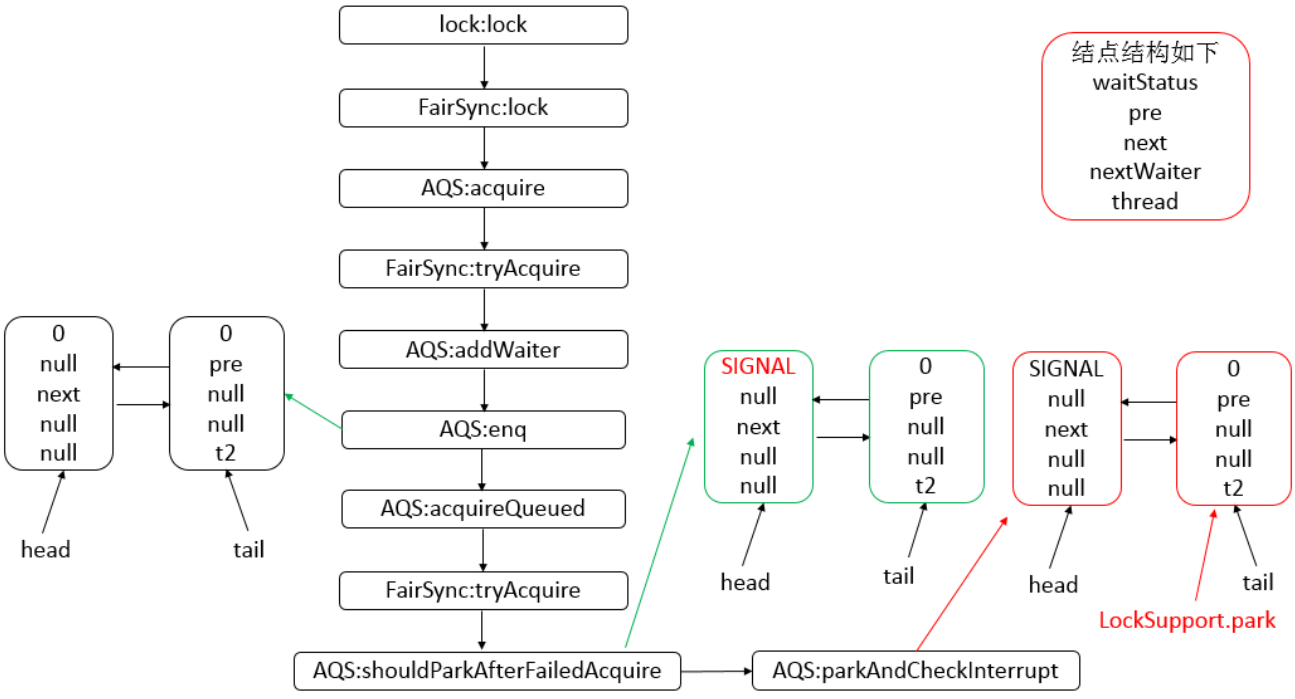
说明: 首先，t1线程的lock操作 -> t2线程的lock操作 -> t3线程的lock操作 -> t1线程的unlock操作 -> t2线程的unlock操作 -> t3线程的unlock操作。根据这个时序图来进一步分析源码的工作流程。

- t1线程执行lock.lock，下图给出了方法调用中的主要方法。



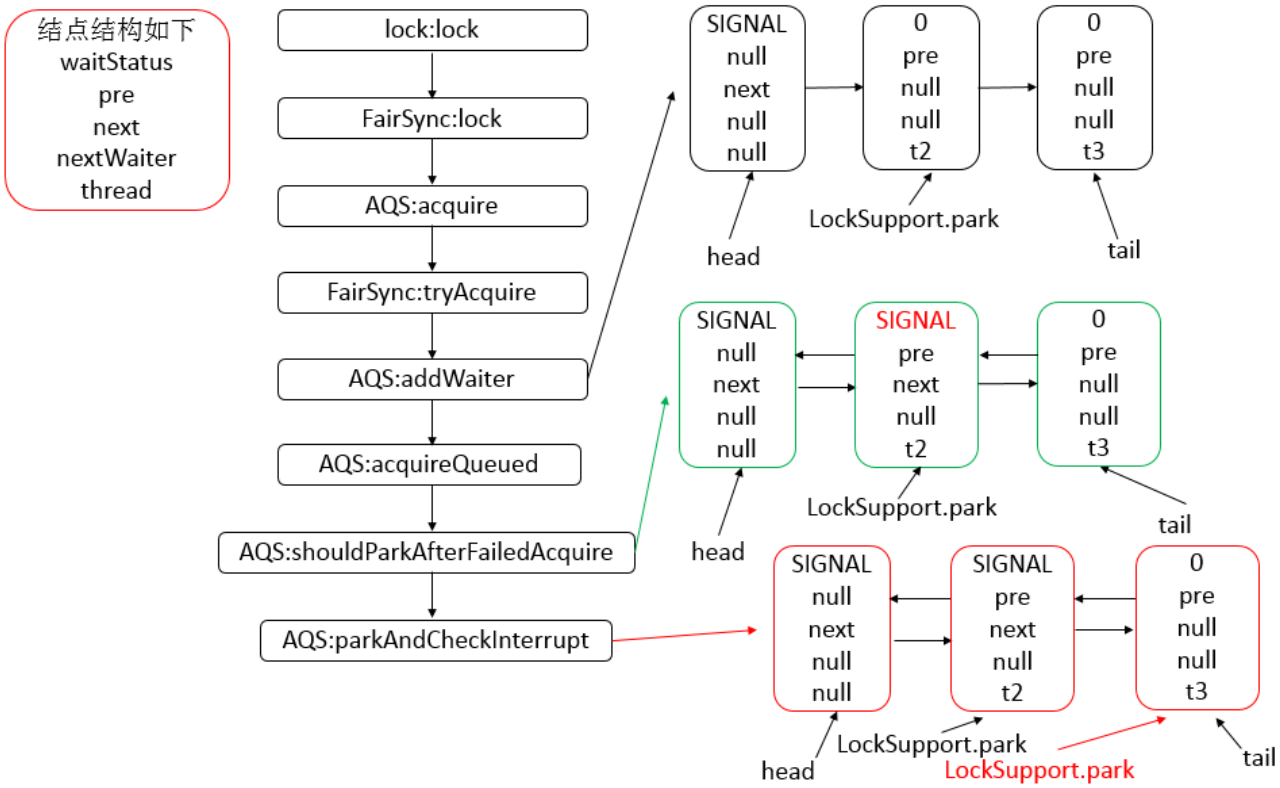
说明: 由调用流程可知，t1线程成功获取了资源，可以继续执行。

- t2线程执行lock.lock，下图给出了方法调用中的主要方法。



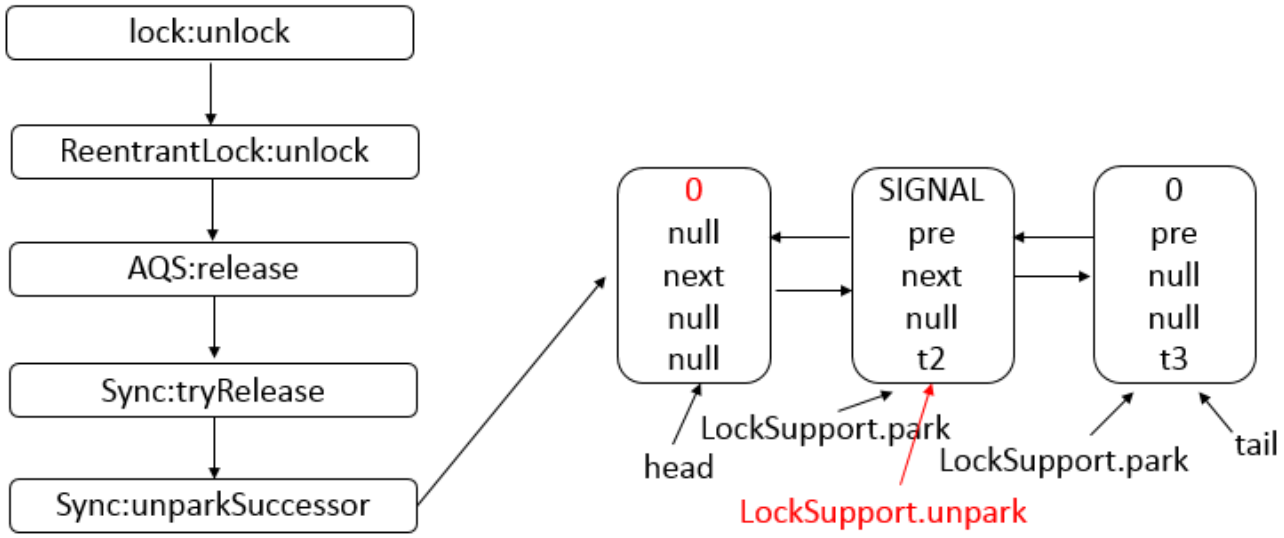
说明: 由上图可知，最后的结果是t2线程会被禁止，因为调用了LockSupport.park。

- t3线程执行lock.lock，下图给出了方法调用中的主要方法。



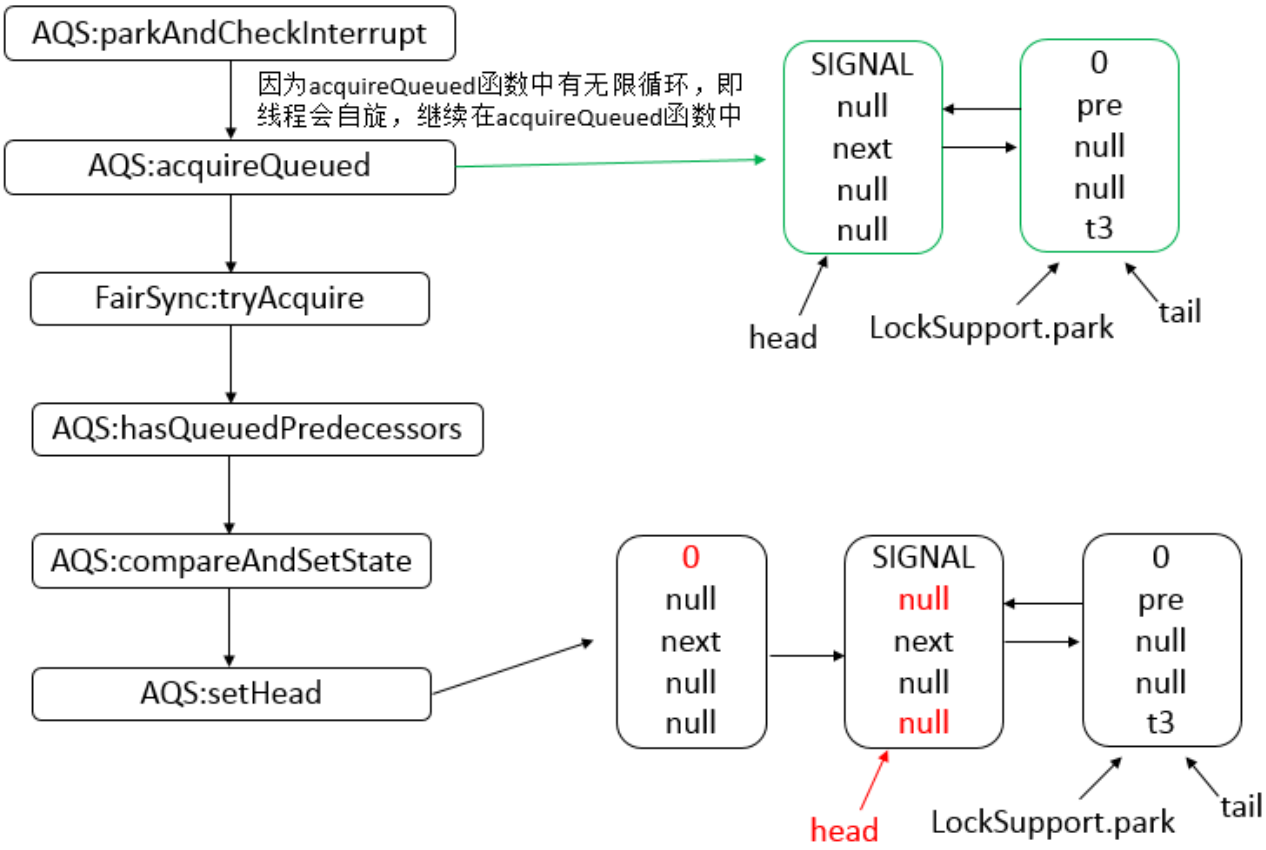
说明: 由上图可知，最后的结果是t3线程会被禁止，因为调用了LockSupport.park。

- t1线程调用了lock.unlock，下图给出了方法调用中的主要方法。



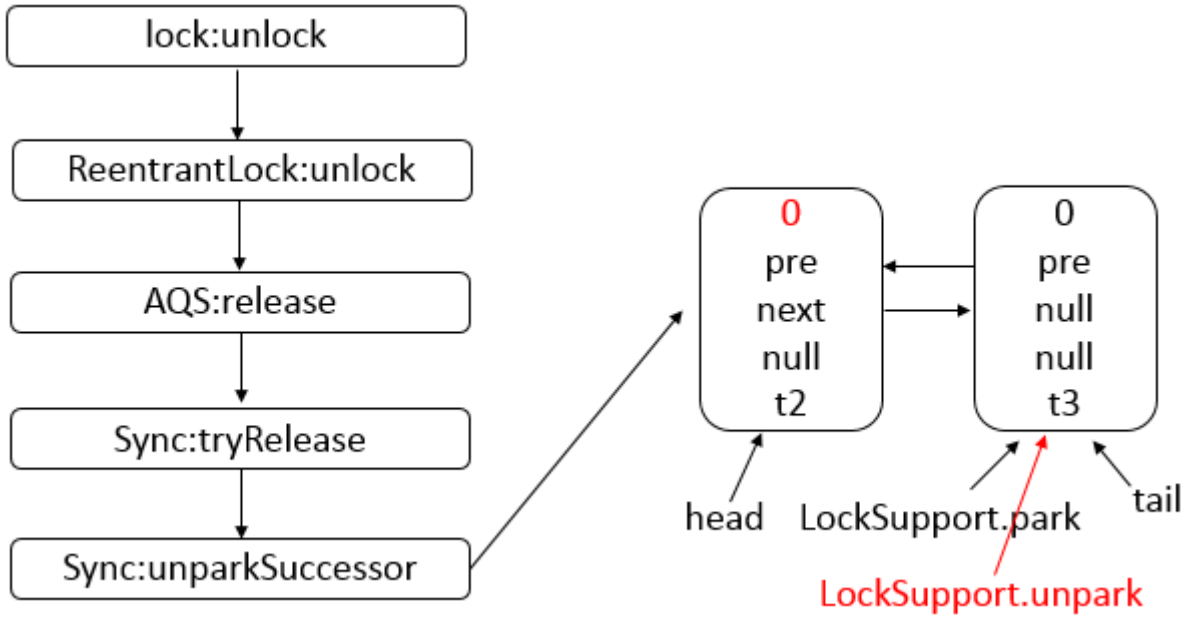
说明: 如上图所示，最后，head的状态会变为0，t2线程会被unpark，即t2线程可以继续运行。此时t3线程还是被禁止。

- t2获得cpu资源，继续运行，由于t2之前被park了，现在需要恢复之前的状态，下图给出了方法调用中的主要方法。



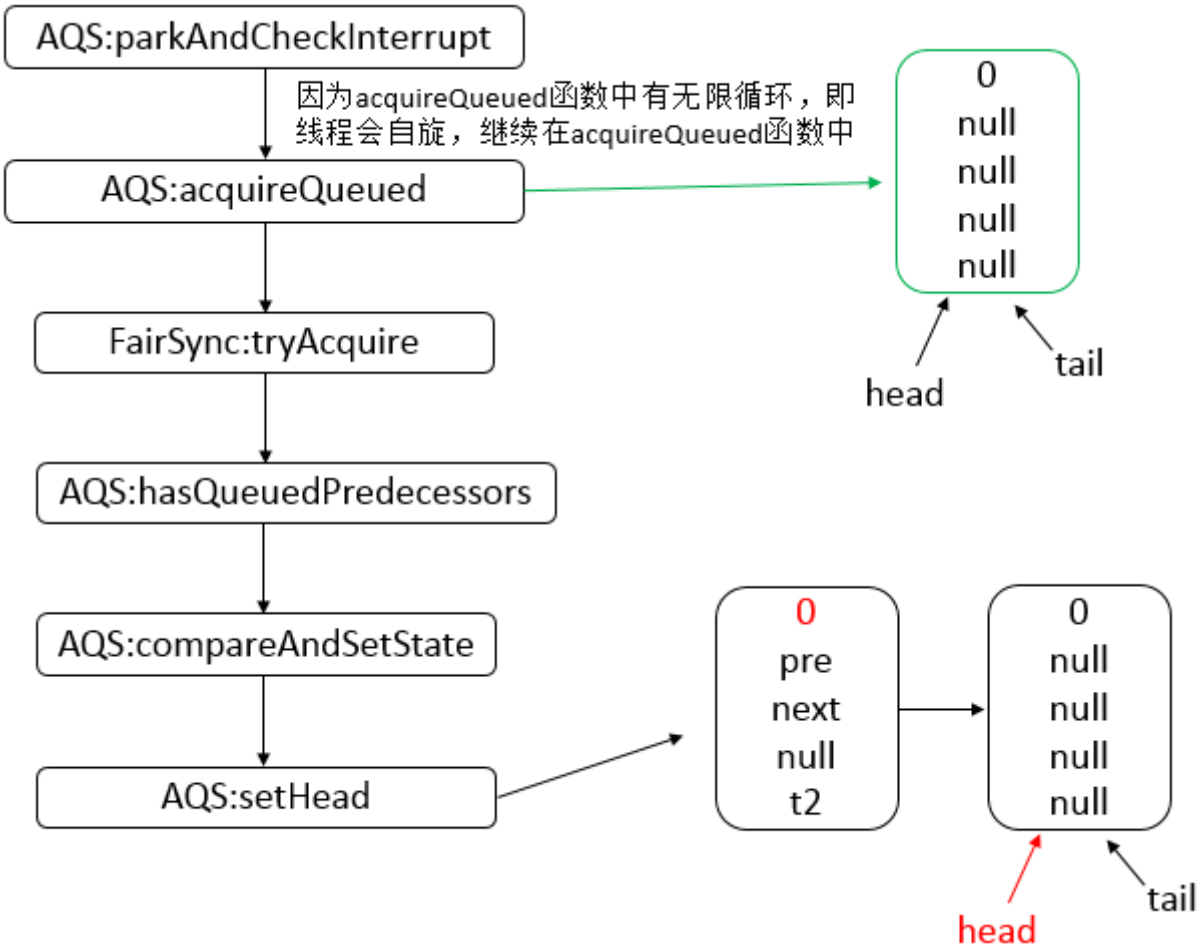
说明：在setHead函数中会将head设置为之前head的下一个结点，并且将pre域与thread域都设置为null，在acquireQueued返回之前，sync queue就只有两个结点了。

- t2执行lock.unlock，下图给出了方法调用中的主要方法。



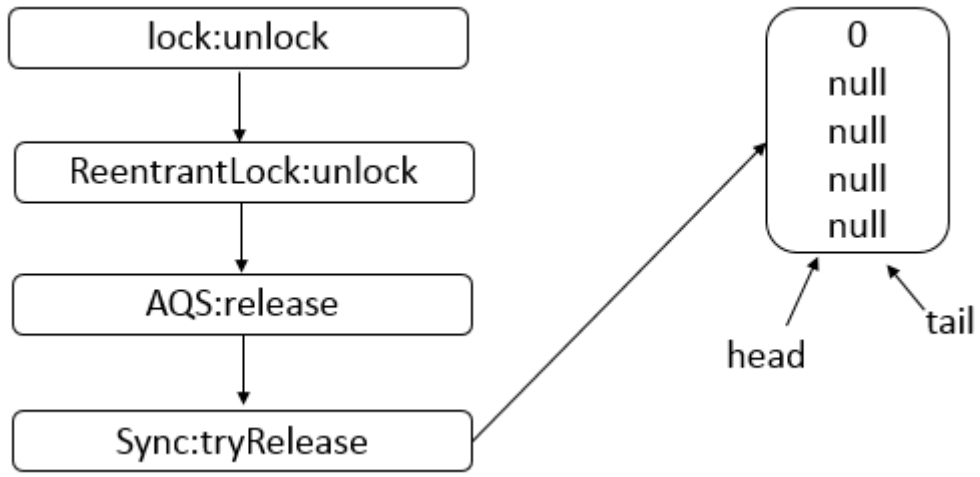
说明: 由上图可知，最终unpark t3线程，让t3线程可以继续运行。

- t3线程获取cpu资源，恢复之前的状态，继续运行。



说明: 最终达到的状态是sync queue中只剩下了一个结点，并且该节点除了状态为0外，其余均为null。

- t3执行lock.unlock，下图给出了方法调用中的主要方法。



说明: 最后的状态和之前的状态是一样的，队列中有一个空节点，头结点为尾节点均指向它。

使用公平策略和Condition的情况可以参考上一篇关于AQS的源码示例分析部分，不再累赘。