AQS原理分析
什么是AQS
同步等待队列
条件等待队列
Condition接口详解
等待唤醒机制之await/signal测试
ReentrantLock详解
ReentrantLock的使用
同步执行，类似于synchronized
可重入
可中断
锁超时
公平锁
条件变量
ReentrantLock源码分析

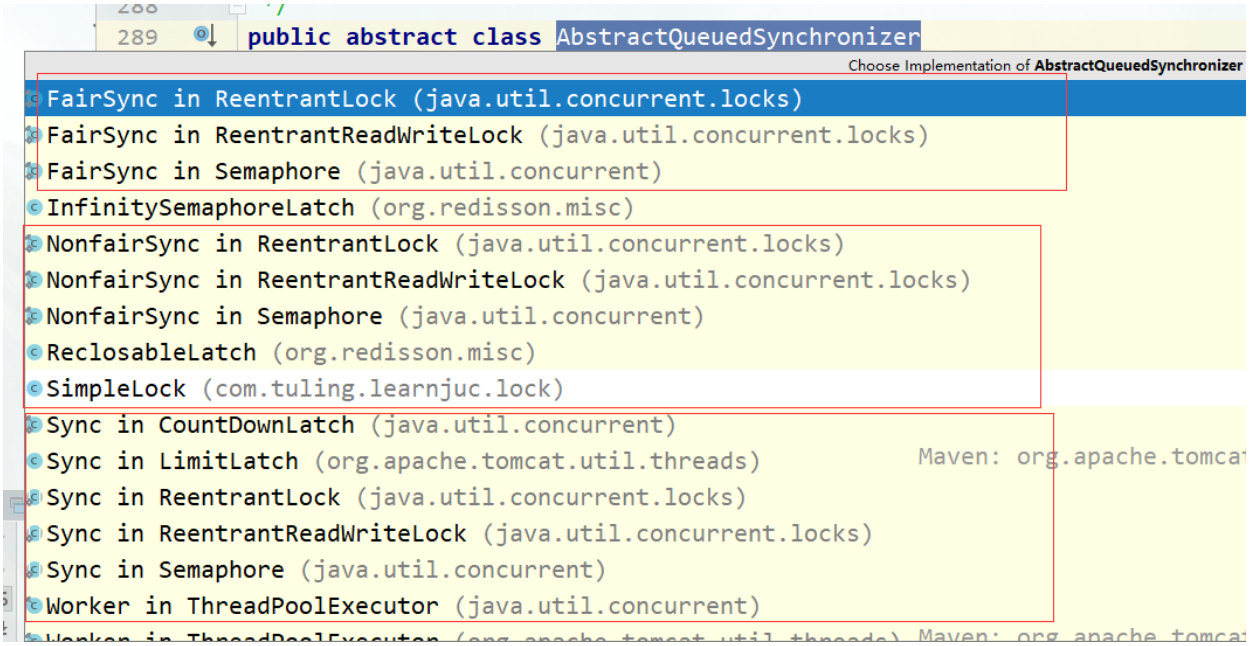
AQS原理分析

什么是AQS

java.util.concurrent包中的大多数同步器实现都是围绕着共同的基础行为，比如等待队列、条件队列、独占获取、共享获取等，而这些行为的抽象就是基于AbstractQueuedSynchronizer（简称AQS）实现的，AQS是一个抽象同步框架，可以用来实现一个依赖状态的同步器。

JDK中提供的大多数的同步器如Lock, Latch, Barrier等，都是基于AQS框架来实现的

- 一般是通过一个内部类Sync继承 AQS
- 将同步器所有调用都映射到Sync对应的方法



AQS具备的特性：

- 阻塞等待队列
- 共享/独占
- 公平/非公平
- 可重入
- 允许中断

AQS内部维护属性volatile int state

- state表示资源的可用状态

State三种访问方式：

- getState()
- setState()
- compareAndSetState()

AQS定义两种资源共享方式

- Exclusive-独占，只有一个线程能执行，如ReentrantLock
- Share-共享，多个线程可以同时执行，如Semaphore/CountDownLatch

AQS定义两种队列

- 同步等待队列： 主要用于维护获取锁失败时入队的线程
- 条件等待队列： 调用await()的时候会释放锁，然后线程会加入到条件队列，调用signal()唤醒的时候会把条件队列中的线程节点移动到同步队列中，等待再次获得锁

AQS 定义了5个队列中节点状态：

1. 值为0，初始化状态，表示当前节点在sync队列中，等待着获取锁。
2. CANCELLED，值为1，表示当前的线程被取消；
3. SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark；

4. CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中；
5. PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；

不同的自定义同步器竞争共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

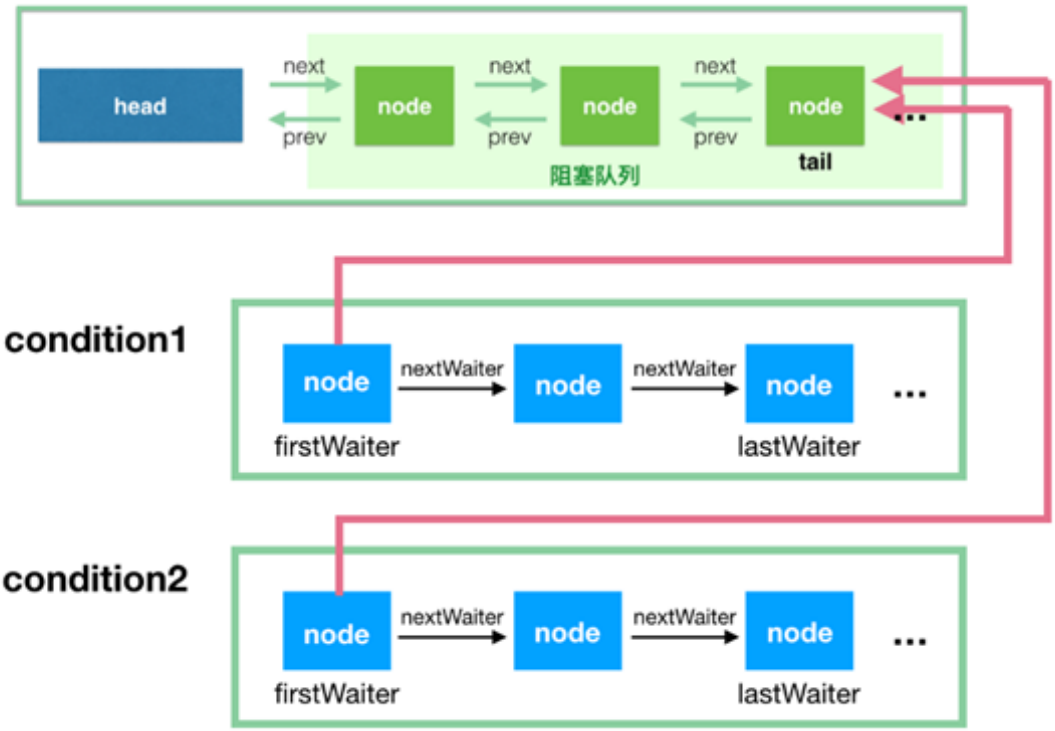
- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

同步等待队列

AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先进先出线程等待队列，Java中的CLH队列是原CLH队列的一个变种,线程由原自旋机制改为阻塞机制。

AQS 依赖CLH同步队列来完成同步状态的管理：

- 当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程
- 当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。
- 通过signal或signalAll将条件队列中的节点转移到同步队列。（由条件队列转化为同步队列）

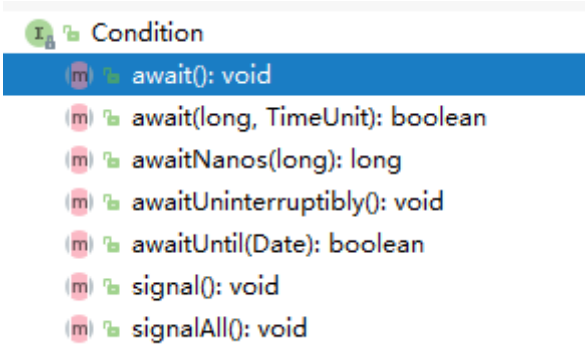


条件等待队列

AQS中条件队列是使用单向列表保存的，用nextWaiter来连接:

- 调用await方法阻塞线程；
- 当前线程存在于同步队列的头结点，调用await方法进行阻塞（从同步队列转化到条件队列）

Condition接口详解



1. 调用Condition#await方法会释放当前持有的锁，然后阻塞当前线程，同时向Condition队列尾部添加一个节点，所以调用Condition#await方法的时候必须持有锁。
2. 调用Condition#signal方法会将Condition队列的首节点移动到阻塞队列尾部，然后唤醒因调用Condition#await方法而阻塞的线程(唤醒之后这个线程就可以去竞争锁了)，所以调用Condition#signal方法的时候必须持有锁，持有锁的线程唤醒被因调用Condition#await方法而阻塞的线程。

等待唤醒机制之await/signal测试

```
@Slf4j
public class ConditionTest {

    public static void main(String[] args) {

        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();

        new Thread(() -> {
            lock.lock();
            try {
                log.debug(Thread.currentThread().getName() + " 开始处理任务");
                condition.await();
                log.debug(Thread.currentThread().getName() + " 结束处理任务");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }).start();

```

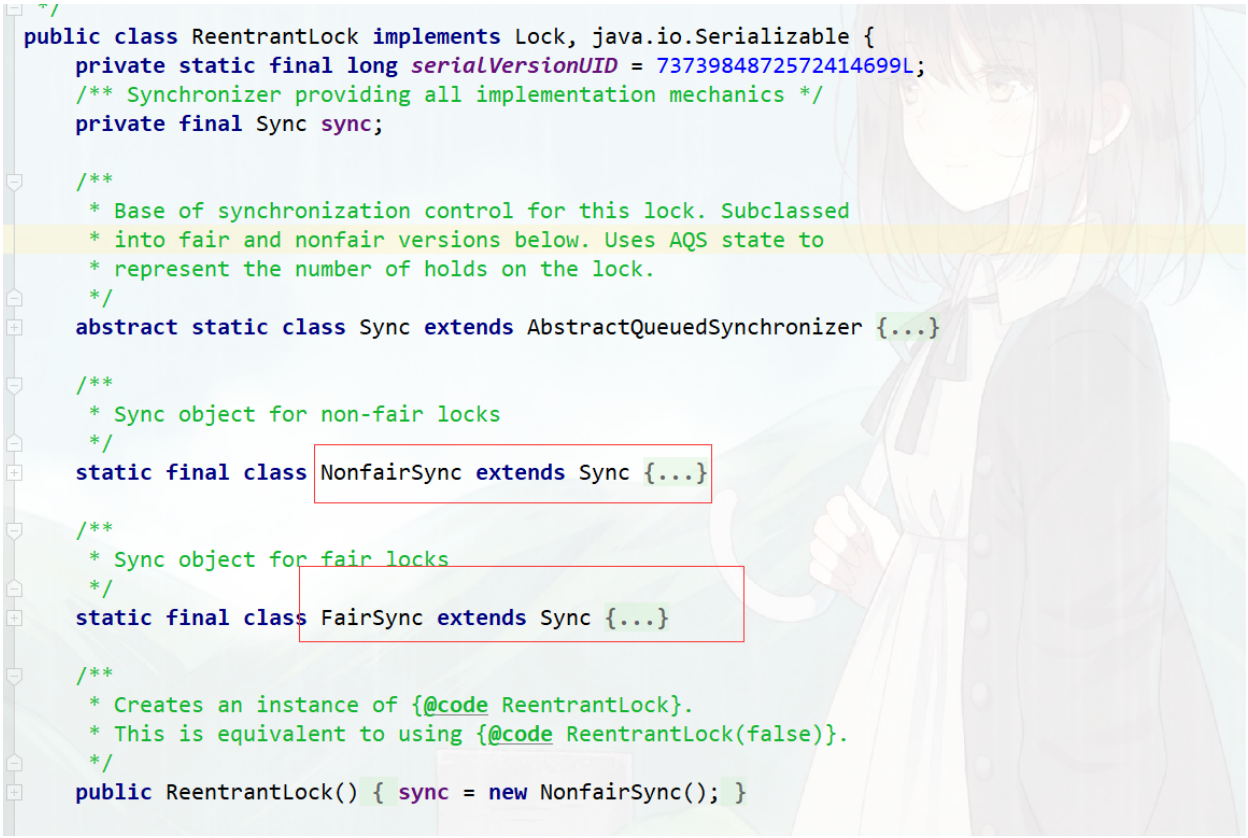
```
22          new Thread(() -> {
23              lock.lock();
24              try {
25                  log.debug(Thread.currentThread().getName() + " 开始处理任务");
26
27                  Thread.sleep(2000);
28                  condition.signal();
29                  log.debug(Thread.currentThread().getName() + " 结束处理任务");
30              } catch (Exception e) {
31                  e.printStackTrace();
32              } finally {
33                  lock.unlock();
34              }
35          }).start();
36      }
```

ReentrantLock详解

ReentrantLock是一种基于AQS框架的应用实现，是JDK中的一种线程并发访问的同步手段，它的功能类似于synchronized是一种互斥锁，可以保证线程安全。

相对于 synchronized， ReentrantLock具备如下特点：

- 可中断
- 可以设置超时时间
- 可以设置为公平锁
- 支持多个条件变量
- 与 synchronized 一样，都支持可重入



```
*/
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    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 {...}

    /**
     * Sync object for non-fair locks
     */
    static final class NonfairSync extends Sync {...}

    /**
     * Sync object for fair locks
     */
    static final class FairSync extends Sync {...}

    /**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     */
    public ReentrantLock() { sync = new NonfairSync(); }
```

顺便总结了几点synchronized和ReentrantLock的区别：

- synchronized是JVM层次的锁实现，ReentrantLock是JDK层次的锁实现；
- synchronized的锁状态是无法在代码中直接判断的，但是ReentrantLock可以通过ReentrantLock#isLocked判断；
- synchronized是非公平锁，ReentrantLock是可以是公平也可以是非公平的；
- synchronized是不可以被中断的，而ReentrantLock#lockInterruptibly方法是可以被中断的；
- 在发生异常时synchronized会自动释放锁，而ReentrantLock需要开发者在finally块中显示释放锁；
- ReentrantLock获取锁的形式有多种：如立即返回是否成功的tryLock(),以及等待指定时长的获取，更加灵活；
- synchronized在特定的情况下对于已经在等待的线程是后来的线程先获得锁（回顾一下sychronized的唤醒策略），而ReentrantLock对于已经在等待的线程是先来的线程先获得锁；

ReentrantLock的使用

同步执行，类似于synchronized

```
1  ReentrantLock lock = new ReentrantLock(); //参数默认false，不公平锁
2  ReentrantLock lock = new ReentrantLock(true); //公平锁
3
4  //加锁
5  lock.lock();
6  try {
7      //临界区
8  } finally {
9      // 解锁
10     lock.unlock();
```

测试

```
public class ReentrantLockDemo {

    private static  int sum = 0;
    private static Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {

        for (int i = 0; i < 3; i++) {
            Thread thread = new Thread(()->{
                //加锁
                lock.lock();
                try {
                    for (int j = 0; j < 10000; j++) {
                        sum++;
                    }
                } finally {
                    // 解锁
                    lock.unlock();
                }
            });
            thread.start();
        }
        Thread.sleep(2000);
        System.out.println(sum);
    }
```

可重入

```
@Slf4j
public class ReentrantLockDemo2 {

    public static ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        method1();
    }


```

```
    public static void method1() {
        lock.lock();
        try {
            log.debug("execute method1");
            method2();
        } finally {
            lock.unlock();
        }
    }
    public static void method2() {
        lock.lock();
        try {
            log.debug("execute method2");
            method3();
        } finally {
            lock.unlock();
        }
    }
    public static void method3() {
        lock.lock();
        try {
            log.debug("execute method3");
        } finally {
            lock.unlock();
        }
    }
```

可中断

```
@Slf4j
public class ReentrantLockDemo3 {

    public static void main(String[] args) {
        ReentrantLock lock = new ReentrantLock();

        Thread t1 = new Thread(() -> {

            log.debug("t1启动...");

```

```
            try {
                lock.lockInterruptibly();
                try {
                    log.debug("t1获得了锁");
                } finally {
                    lock.unlock();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
                log.debug("t1等锁的过程中被中断");
            }

        }, "t1");

        lock.lock();
        try {
            log.debug("main线程获得了锁");
            t1.start();
            //先让线程t1执行
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            t1.interrupt();
            log.debug("线程t1执行中断");
        } finally {
            lock.unlock();
        }
    }
```

锁超时

立即失败

```
@Slf4j
public class ReentrantLockDemo4 {

    public static void main(String[] args) {
        ReentrantLock lock = new ReentrantLock();
```

```
6
7           Thread t1 = new Thread(() -> {
8
9               log.debug("t1启动...");
10              // 注意： 即使是设置的公平锁，此方法也会立即返回获取锁成功或失败，公平策略不生效
11              if (!lock.tryLock()) {
12                  log.debug("t1获取锁失败，立即返回false");
13                  return;
14              }
15              try {
16                  log.debug("t1获得了锁");
17              } finally {
18                  lock.unlock();
19              }
20
21          }, "t1");
22
23
24          lock.lock();
25          try {
26              log.debug("main线程获得了锁");
27              t1.start();
28              //先让线程t1执行
29              try {
30                  Thread.sleep(1000);
31              } catch (InterruptedException e) {
32                  e.printStackTrace();
33              }
34          } finally {
35              lock.unlock();
36          }
37
38      }
39
```

超时失败

```
1   @Slf4j
2   public class ReentrantLockDemo4 {
3
```

```
4       public static void main(String[] args) {
5           ReentrantLock lock = new ReentrantLock();
6           Thread t1 = new Thread(() -> {
7               log.debug("t1启动...");
8               //超时
9               try {
10                  if (!lock.tryLock(1, TimeUnit.SECONDS)) {
11                      log.debug("等待 1s 后获取锁失败，返回");
12                      return;
13                  }
14              } catch (InterruptedException e) {
15                  e.printStackTrace();
16                  return;
17              }
18              try {
19                  log.debug("t1获得了锁");
20              } finally {
21                  lock.unlock();
22              }
23
24          }, "t1");
25
26
27          lock.lock();
28          try {
29              log.debug("main线程获得了锁");
30              t1.start();
31              //先让线程t1执行
32              try {
33                  Thread.sleep(2000);
34              } catch (InterruptedException e) {
35                  e.printStackTrace();
36              }
37          } finally {
38              lock.unlock();
39          }
40
41      }
42
```

公平锁

ReentrantLock 默认是不公平的

```
@Slf4j
public class ReentrantLockDemo5 {

    public static void main(String[] args) throws InterruptedException {
        ReentrantLock lock = new ReentrantLock(true); //公平锁

        for (int i = 0; i < 500; i++) {
            new Thread(() -> {
                lock.lock();
                try {
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    log.debug(Thread.currentThread().getName() + " running...");
                } finally {
                    lock.unlock();
                }
            }, "t" + i).start();
        }
        // 1s 之后去争抢锁
        Thread.sleep(1000);

        for (int i = 0; i < 500; i++) {
            new Thread(() -> {
                lock.lock();
                try {
                    log.debug(Thread.currentThread().getName() + " running...");
                } finally {
                    lock.unlock();
                }
            }, "强行插入" + i).start();
        }
    }
```

思考：ReentrantLock公平锁和非公平锁的性能谁更高?

条件变量

java.util.concurrent类库中提供Condition类来实现线程之间的协调。调用Condition.await() 方法使线程等待，其他线程调用Condition.signal() 或 Condition.signalAll() 方法唤醒等待的线程。

注意：调用Condition的await()和signal()方法，都必须在lock保护之内。

```
@Slf4j
public class ReentrantLockDemo6 {
    private static ReentrantLock lock = new ReentrantLock();
    private static Condition cigCon = lock.newCondition();
    private static Condition takeCon = lock.newCondition();

    private static boolean hashcig = false;
    private static boolean hastakeout = false;

    //送烟
    public void cigratee(){
        lock.lock();
        try {
            while(!hashcig){
                try {
                    log.debug("没有烟，歇一会");
                    cigCon.await();

                }catch (Exception e){
                    e.printStackTrace();
                }
            }
            log.debug("有烟了，干活");
        }finally {
            lock.unlock();
        }
    }

    //送外卖
    public void takeout(){
        lock.lock();
        try {
```

```
            while(!hastakeout){
                try {
                    log.debug("没有饭，歇一会");
                    takeCon.await();

                }catch (Exception e){
                    e.printStackTrace();
                }
            }
            log.debug("有饭了，干活");
        }finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ReentrantLockDemo6 test = new ReentrantLockDemo6();
        new Thread(() ->{
            test.cigratee();
        }).start();

        new Thread(() -> {
            test.takeout();
        }).start();

        new Thread(() ->{
            lock.lock();
            try {
                hashcig = true;
                //唤醒送烟的等待线程
                cigCon.signal();
            }finally {
                lock.unlock();
            }


        },"t1").start();

        new Thread(() ->{
```

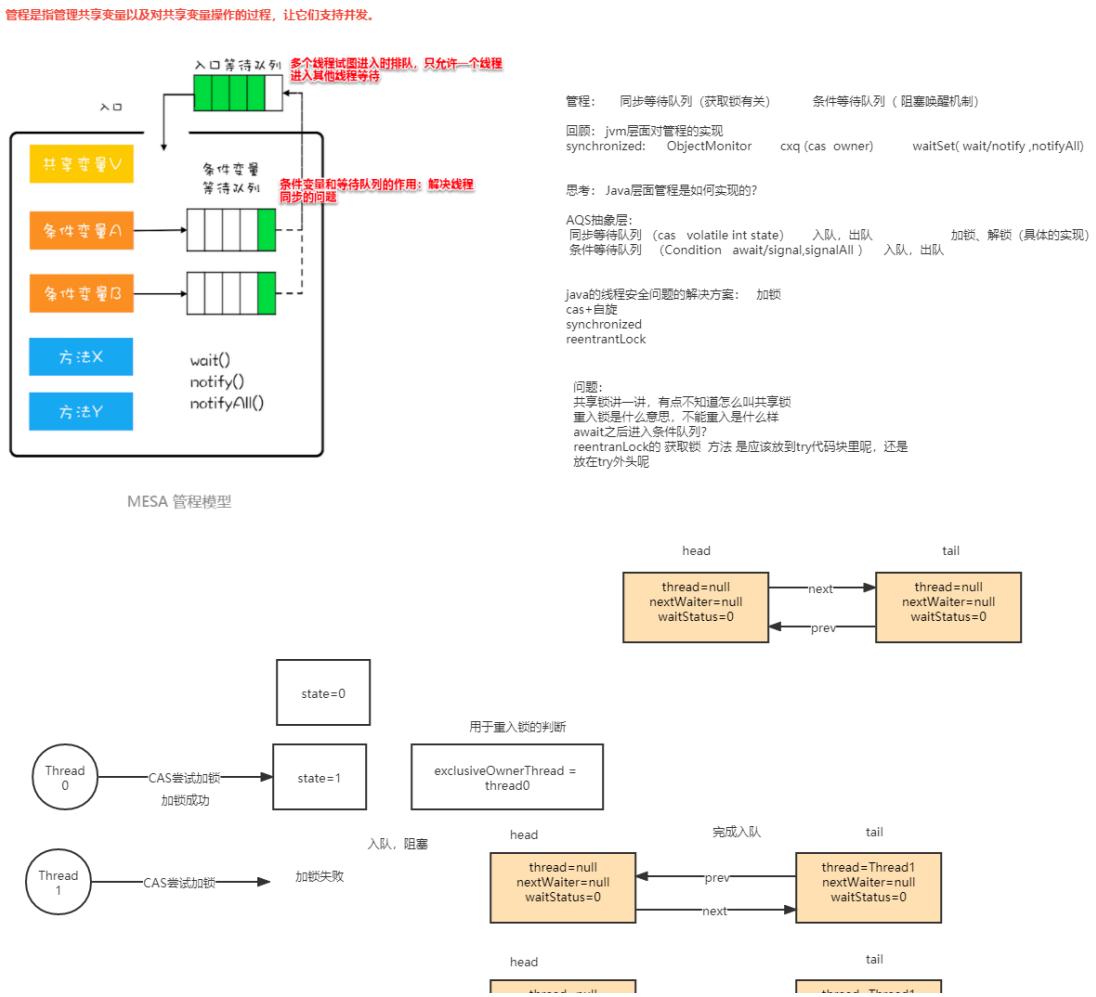
```
72              lock.lock();
73              try {
74                  hastakeout = true;
75                  //唤醒送饭的等待线程
76                  takeCon.signal();
77              }finally {
78                  lock.unlock();
79              }
80          },"t2").start();
81      }
82
```

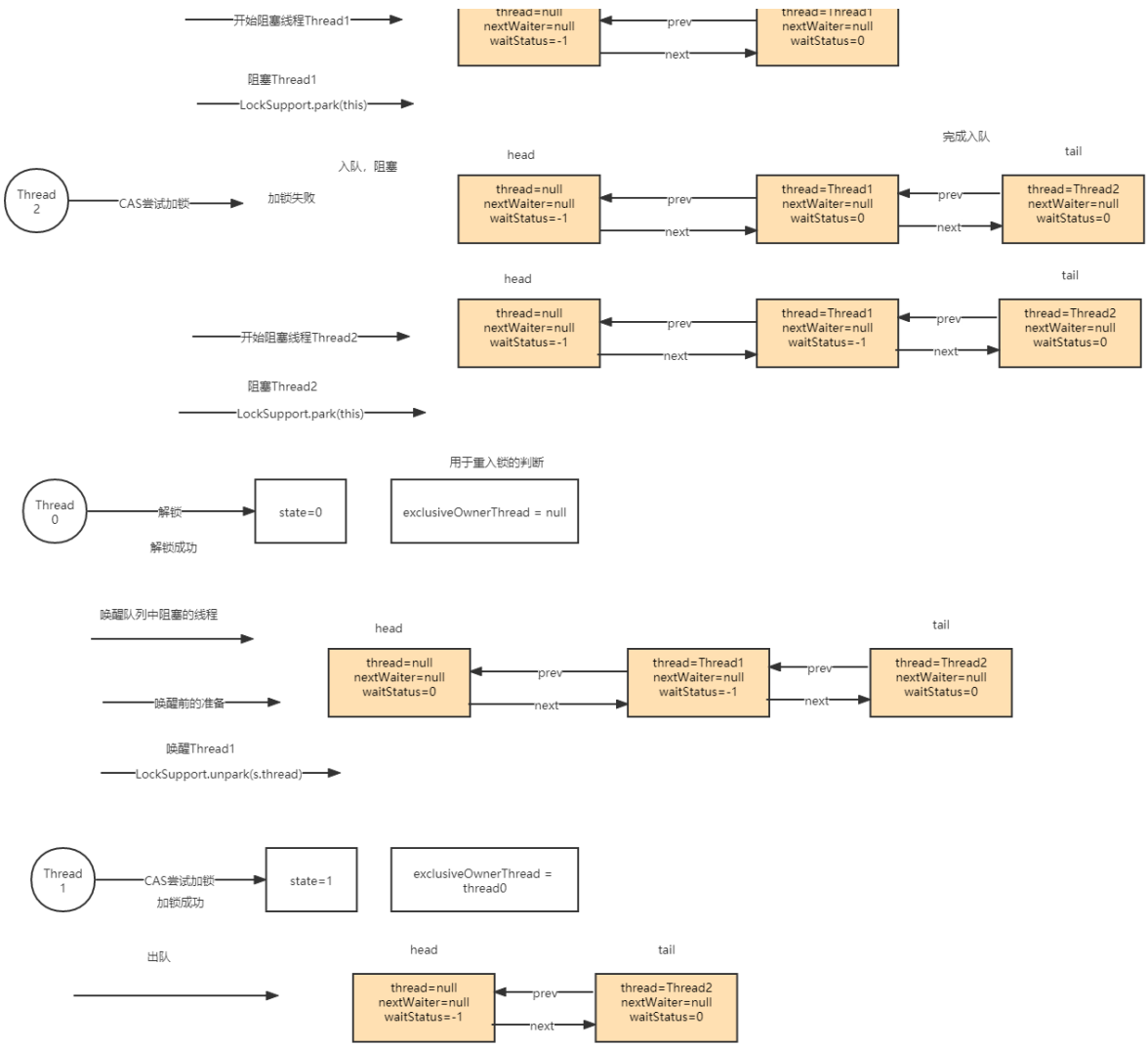
ReentrantLock源码分析

关注点：

1. ReentrantLock加锁解锁的逻辑
2. 公平和非公平，可重入锁的实现
3. 线程竞争锁失败入队阻塞逻辑和获取锁的线程释放锁唤醒阻塞线程竞争锁的逻辑实现 （设计的精髓：并发场景下入队和出队操作）

https://www.processon.com/view/link/6191f070079129330ada1209




开始阻塞线程Thread1
thread=null
nextWaiter=null
waitStatus=-1
prev
next
thread=Thread1
nextWaiter=null
waitStatus=0
阻塞Thread1
LockSupport.park(this)
完成入队
head
tail
入队，阻塞
Thread 2
CAS尝试加锁
加锁失败
thread=null
nextWaiter=null
waitStatus=-1
thread=Thread1
nextWaiter=null
waitStatus=0
thread=Thread2
nextWaiter=null
waitStatus=0
开始阻塞线程Thread2
阻塞Thread2
LockSupport.park(this)
用于重入锁的判断
Thread 0
解锁
解锁成功
state=0
exclusiveOwnerThread = null
唤醒队列中阻塞的线程
唤醒前的准备
唤醒Thread1
LockSupport.unpark(s.thread)
Thread 1
CAS尝试加锁
加锁成功
state=1
exclusiveOwnerThread = thread0
出队