# 概述

同步器AbstractQueuedSynchronizer，简称AQS，主要是为了java的跨平台。C++中在使用信号量时都直接使用操作系统提供的信号量以及信号量自带的同步队列，因此不同操作系统需要使用不同的系统调用。java为了跨平台，则将同步队列抽象出来作为一个抽象类AbstractQueuedSynchronizer。

AQS是一个抽象类，不能直接创建对象，JDK1.8中实现了多个同步器，比如Semaphore、CountDownLatch等都创建了相应的内部类来实现AQS类，AQS的继承关系如下所示。



AQS子类继承关系

为了在不同场景使用同步器，JDK特此提供多种同步器，AQS类是所有这些同步器的核心，各种同步器都只是依赖AQS提供的基本功能做了一层封装。

本文档主要学习JDK1.8中提供的AbstractQueuedSynchronizer类以及其子类的知识点。

# AbstractQueuedSynchronizer

AbstractQueuedSynchronizer同步器，简称AQS。其底层实现依赖于双向链表结构，链表节点来自于其内部类Node。

## 主要成员

|  |  |  |
| --- | --- | --- |
| 名字 | 类型 | 描述 |
| head | Node | 队列头指针 |
| tail | Node | 队列尾指针 |
| state | int | 被同步器同步资源的个数，如果state等于0，获取资源的线程需要排队 |
| headOffset | long | 记录head成员在AQS类中的偏移量 |
| tailOffset | long | 记录tail成员在AQS类中的偏移量 |
| stateOffset | long | 记录state成员在AQS类中的偏移量 |

# 子类

AQS中这几个方法都是直接抛出异常，如果需要基于AQS完成不同的同步器，需要实现这几个中的方法。

* tryAcquire(int)：子类实现各自获取锁的算法（用于排斥性的锁，如ReentrantLock）
* tryRelease(int)：子类实现各自释放锁的算法
* tryAcquireShared(int)：用于共享性的锁（如信号量Semaphore）
* tryReleaseShared(int)
* isHeldExclusively()

## 信号量Semaphore

信号量提供两种同步器，一种是公平同步器FairSync，一种是非公平同步器NonfairSync，这两个同步器具有很多相似的地方，因此对两种同步器抽象出了一个基类Sync。

Sync、FairSync、NonfairSync都是Semaphore的内部类。

### 构造方法

第一种方式只传递允许同步的线程数permits，默认使用非公平同步器NonfairSync。



第二种方式传递允许同步的线程数，同时设置同步器类型，如果fair是true创建公平同步器FairSync，否则为非公平同步器。

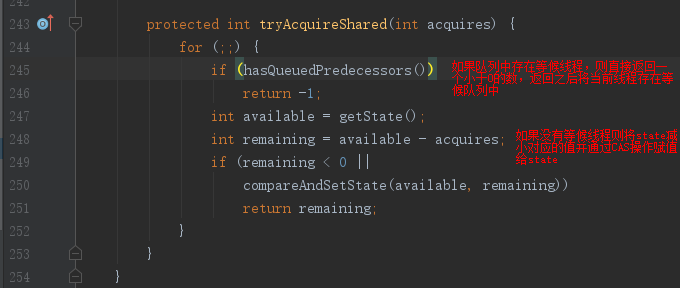


### 重要操作

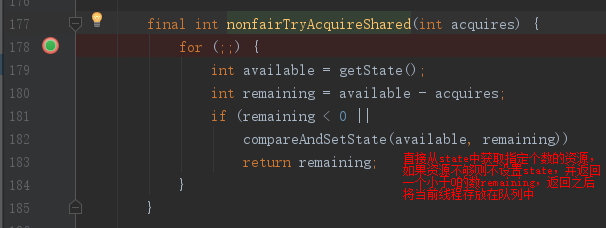
#### acquire

当线程调用acquire方法时，最终都会都会通过多态调用各自的tryAcquireShared方法，因此需要分情况讨论。

公平信号量：



非公平信号量



总结

公平信号量

* 线程调用acquire操作，获取指定个数（permits）的锁资源；
* 判断当前队列是否存在等候线程，如果存在则返回，返回之后将当前线程存在等候队列中
* 如果队列中不存在等候线程，从state 中减去permits得到remaining，如果remaining 小于0则返回，返回后将当前线程存放在队列中；如果remaining大于0，则通过cas操作将state值设置为 remaining并返回，如果cas操作失败则for循环继续执行，执行从第二步重新开始；

非公平信号量

* 线程调用acquire操作，获取指定个数（permits）的锁资源
* 无需判断队列中是否存在等候线程，直接从state 中减去permits得到remaining，如果remaining 小于0则返回，返回后将当前线程存放在队列中；如果remaining大于0，则通过cas操作将state值设置为 remaining并返回，如果cas操作失败则for循环继续执行，执行从第二步重新开始

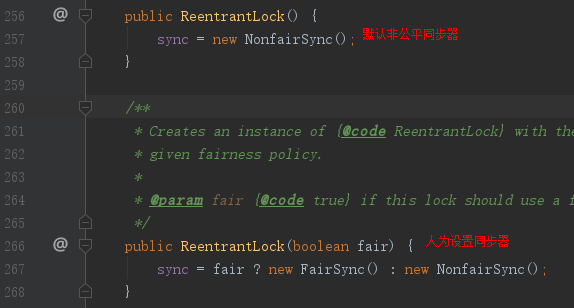
#### release

release操作不区分公平性，也就是无论是公平信号量还是非公平信号量，release操作都执行一样的算法；最终执行子类实现的tryReleaseShared方法。

## 可重入锁ReentrantLock

可重入锁和Semaphore类似，都包含公平同步器和非公平同步器，只不过可重入锁的state变量的值如果为0表示没有线程占有锁，如果为1表示有一个线程占有锁，如果大于1表示该锁被同一个线程多次获取并占有（可以通过多次调用lock操作，调试查看锁中state变量值的变化，每调用一次lock，state自增1，调用一次unlock，state自减）。

### 构造方法



### 重要操作

#### lock

调用lock方法时，如果是非公平的可重入锁，首先判断state变量是否为0，为0则将state通过CAS设置为1，同时将当前线程作为占有锁的线程；如果不为0，则调用对应类实现的tryAcquire方法（多态）；如果是公平的可重入锁，直接调用对应类的tryAcquire方法。

公平可重入锁的tryAcquire方法



非公平可重入锁的tryAcquire方法



总结

公平可重入锁lock操作

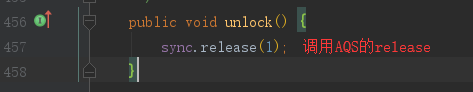
* 判断state是否为0
* 如果state=0，并且无其他线程排队，则通过CAS将state设置为1，同时将占有锁的线程设置为当前线程；
* 如果state>0，判断占有锁的线程和当前线程是否相同，相同则将state+1，否则返回false；
* 上述操作都不成功时，说明获取锁失败，则将当前线程存放在AQS队列，并将当前线程中断。

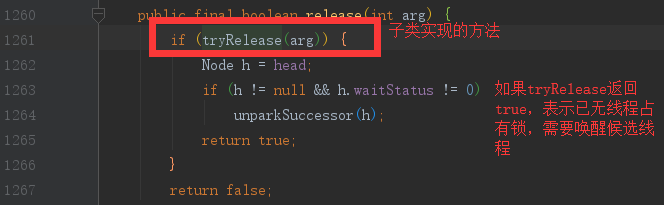
非公平可重入锁lock操作

* 判断state是否为0
* 如果state=0，则通过CAS将state设置为1，同时将占有锁的线程设置为当前线程；
* 如果state>0，判断占有锁的线程和当前线程是否相同，相同则将state+1，否则返回false；
* 上述操作都不成功时，说明获取锁失败，则将当前线程存放在AQS队列，并将当前线程中断。

#### unlock

无论是公平锁还是非公平说，unlock操作都采用相同的方式，代码如下：







总结

* 公平和非公平可重入锁的unlock操作算法是一样的；
* state减1
* 判断调用线程和锁占有线程是否相同，如果不同抛出异常
* 如果state为0，把占有锁的线程置为null，设置state为0，同时返回true，之后唤醒候选线程；
* 如果state不为0，设置state的新值，返回false

## 可重入读写锁ReentrantWriteReadLock