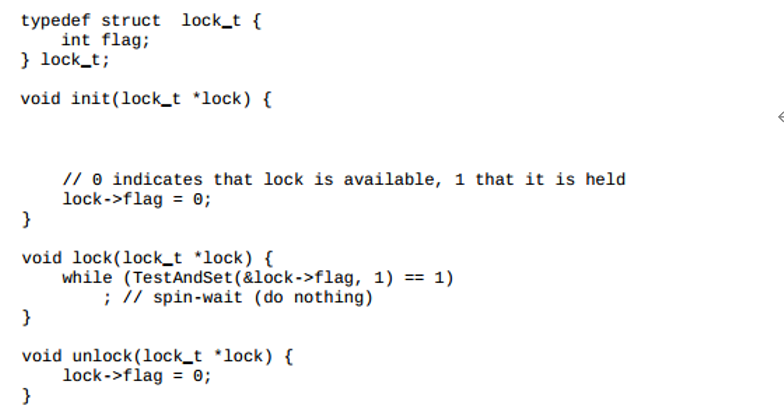
# 操作系统原语

为了能够解决并发问题，操作系统提供了几种原子性操作用于锁的实现。

## TestAndSet

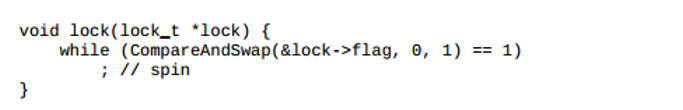
测试并设置指令。这个指令会先将旧值替换为新值，然后返回旧的值，这两步操作是原子性的。有了这个指令，可以实现一个自旋锁。



## CompareAndSwap

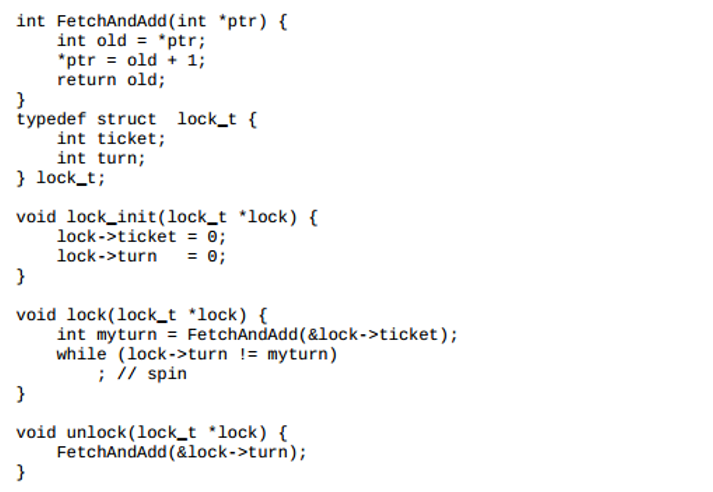
比较并交换指令。它会先检查要替换的旧值是否和预期相符，如果相符会将其替换为新值，否则就什么也不做。

由他来实现锁只需要把TestAndSet的lock方法改为如下：



## FetchAndAdd

获取并增加指令，它可以原子性地返回旧值，并让该值自增1。它可以实现一个公平自旋锁。



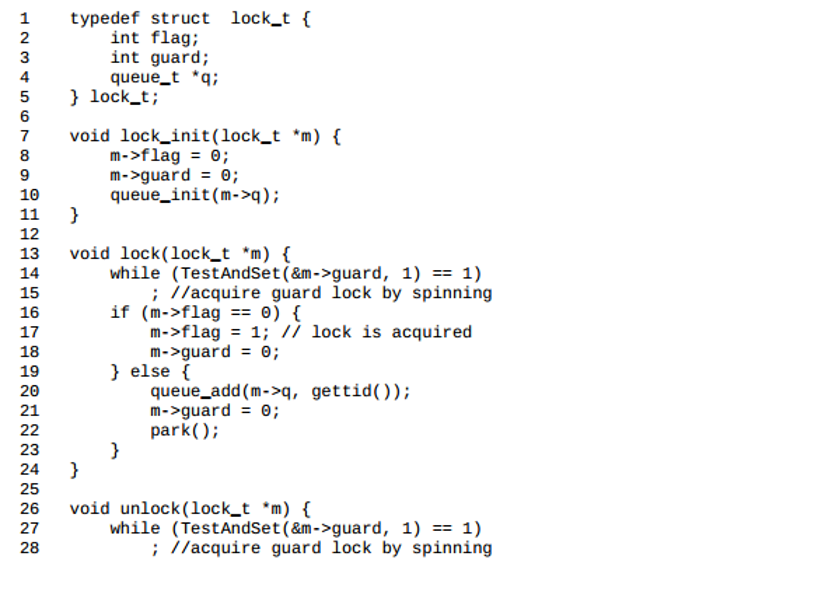
## pack和unpack

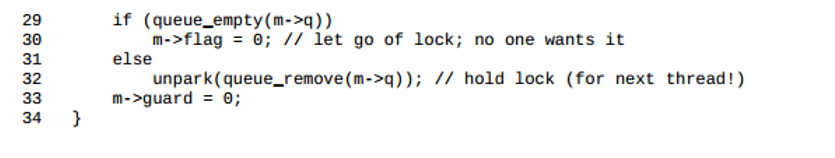
使用之前的三个原子性操作只能实现自旋锁，为了能够更好地利用CPU，操作系统还提供了另外两个操作：

pack: 让调用线程休眠；

unpack(threadID): 唤醒threadID标识的线程。

结合这两个操作原语可以实现非自旋锁。





这里有个问题，如果在即将调用pack()之前，发生了线程切换，然后切换到了刚好持有锁的线程，线程此时调用了unpack()，这时候再切换到之前的线程，那么这时候这个线程调用park()之前就会永久地睡眠。这种问题有时称为唤醒/等待竞争。

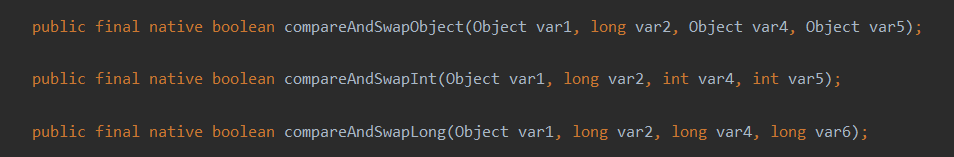
为了解决这个问题，操作系统提供了另外一个原语,separk()，通过separk()，一个线程表明自己马上要park，如果另一个线程刚好被调度，并且调用了unpark,那么后续的park就会直接返回。所以lock方法可以做一点修改：

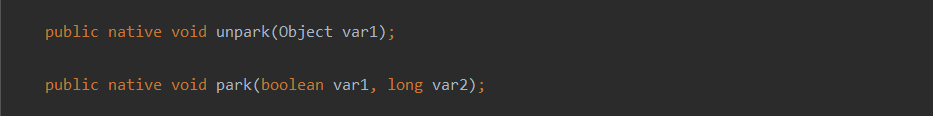


# Java的锁

## Java实现锁的基础

Java的锁也是基于上面提供的几个操作来实现的。在Java中要利用这些操作需要使用到Unsafe类，这些操作都集成在Unsafe类中。





Java提供的park和unpark是不会出现操作系统的park和unpark的唤醒/等待竞争问题的，因为如果在调用park之前调用了unpark，那么park会立刻返回(这应该也是使用了操作系统提供的其他原语，例如之前提到的separk操作)。

## 实现一个简单的锁

有了这几个操作，我们就可以实现锁了，下面是利用这些操作实现的一个简单的互斥锁。这个锁实现只是一个示例，还有很多异常情况未考虑进去。





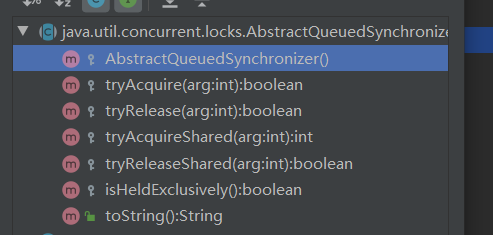
# Java的AQS框架

## 概述

自己实现锁时需要考虑何时入队，何时出队，还有锁状态的管理，线程中断等等情况，为了能够简化实现锁的难度，Java提供了AQS框架用于实现锁。

AQS是指的AbstractQueuedSynchronizer，它提供了对队列的管理，对锁状态的管理，另外实现上屏蔽了很多并发细节问题，使用者只需要根据锁状态告知何时是锁住的，何时可以释放锁即可。

使用这个框架只需要实现以下几个方法即可：



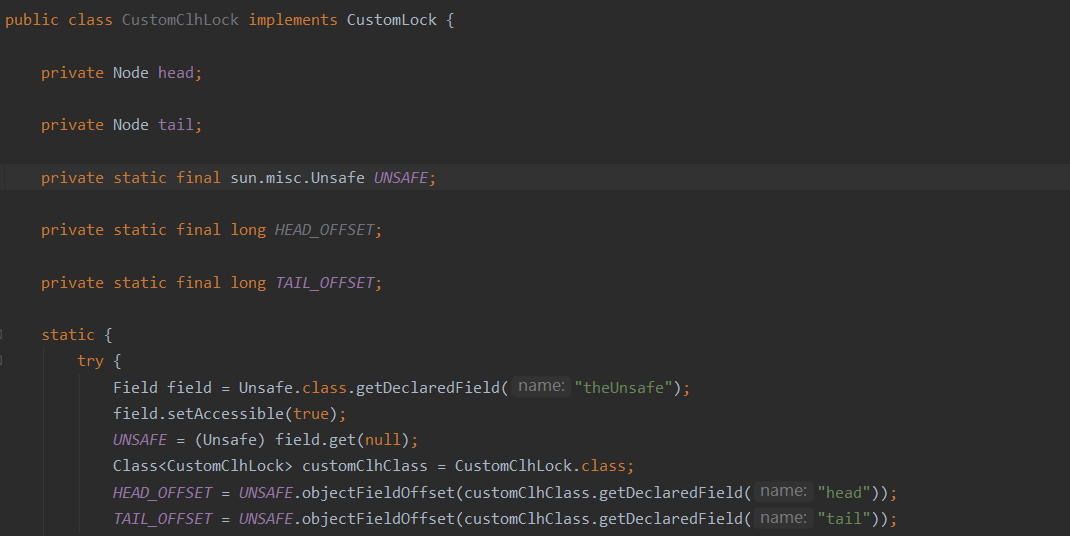
如果不需要用到共享锁，那么只需要实现tryAcquire、tryRelease、isHeldExclusively三个方法即可。

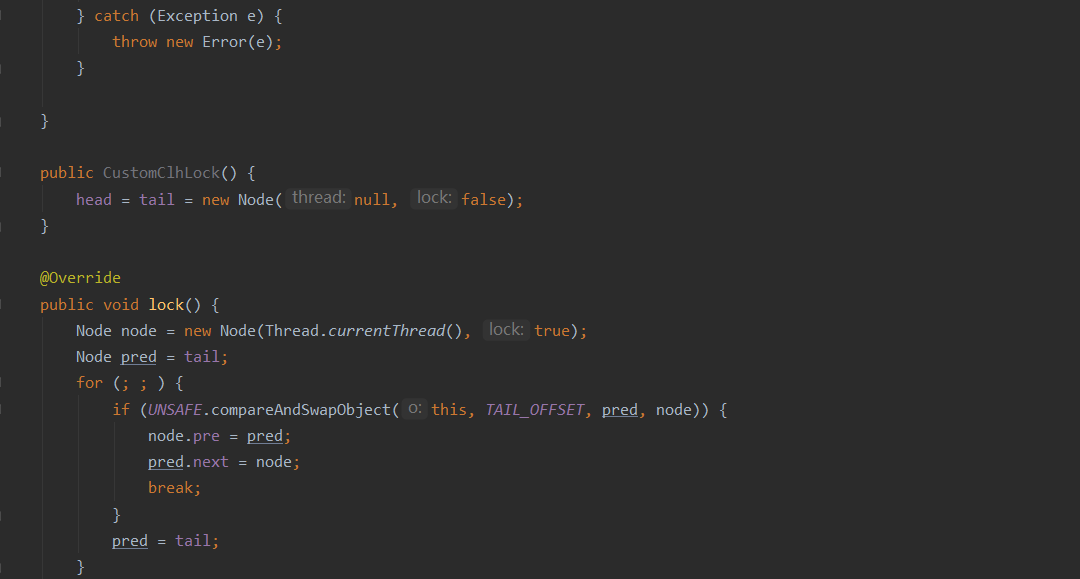
## 原理

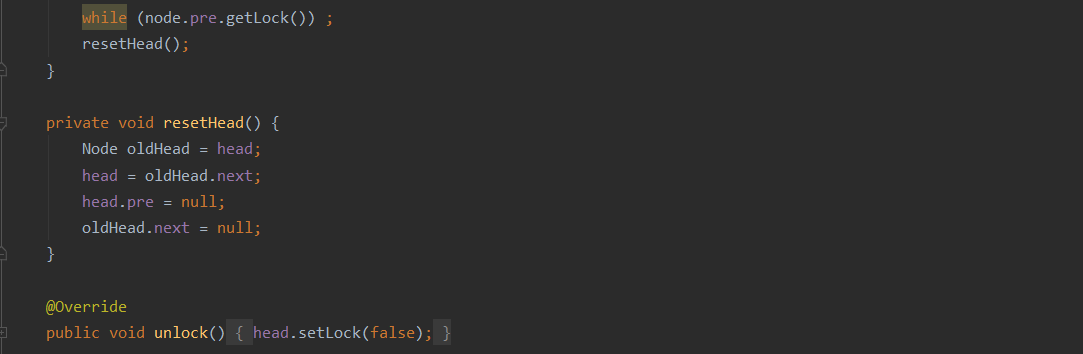
### CLH锁

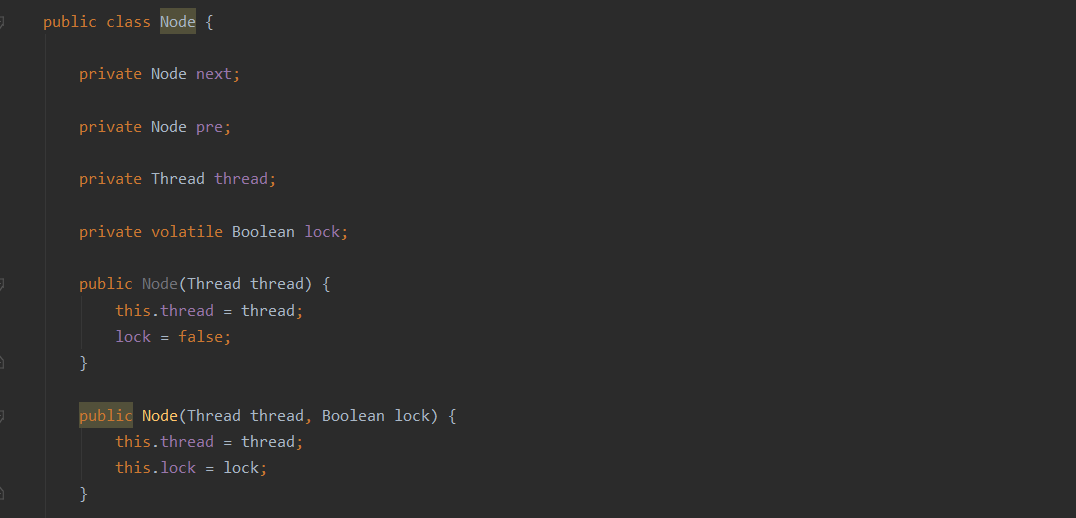
AQS中使用的队列借鉴了CLH队列的方式，CLH队列是用于自旋锁的实现的，CLH锁的数据结构简单，是一个链表结构，每个链表结点会有两个属性，线程和状态，所有请求获取锁的线程会排列在链表队列中，然后自旋访问队列中前一个节点的状态。

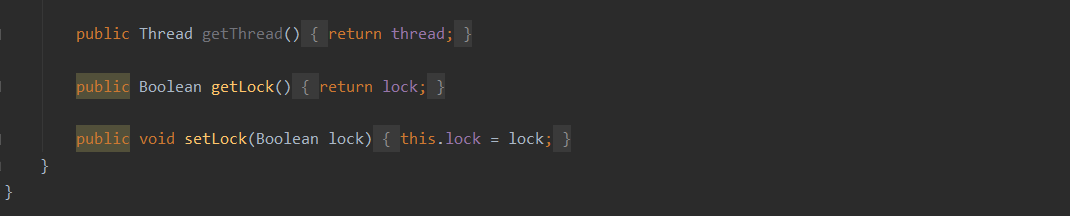
实现一个简单的CLH锁。











### 队列的管理

AQS的队列和CLH锁中的队列类似，也是一个链表结构，每个结点存储了线程和状态，AQS中有两个FIFO队列，一个是等待锁的队列，一个是条件变量的队列。

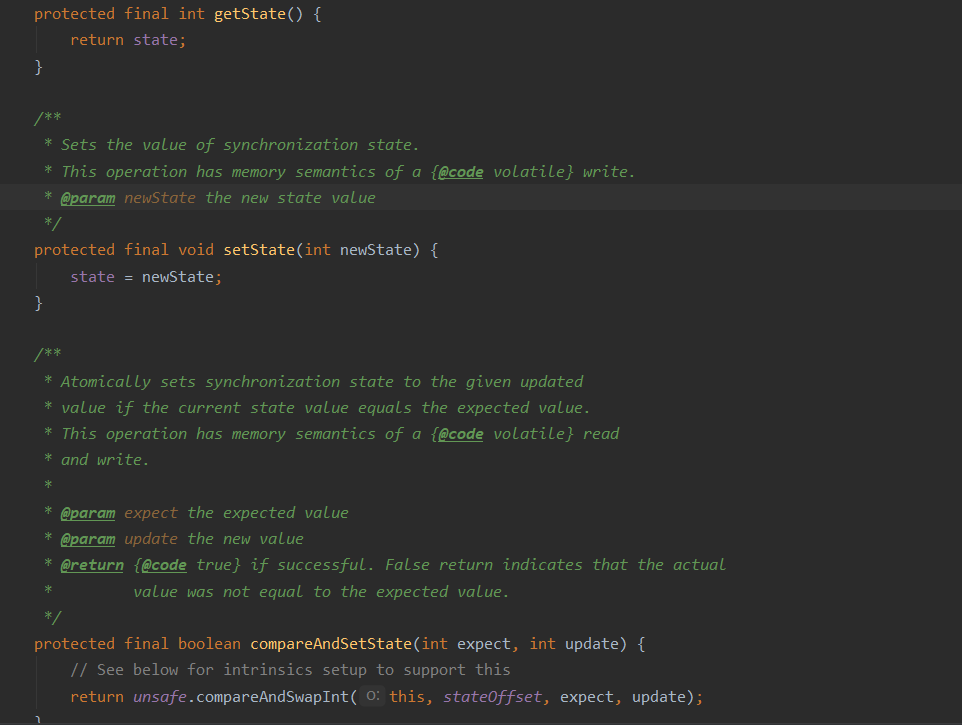
对于锁的队列，调用lock方法会将没有持有锁的线程加入到队列的尾部中，调用unlock释放锁时会释放队列的头部结点。

对于条件变量的队列，调用wait方法会将当前线程加入到队列的尾部，并唤醒锁的队列的头部结点，调用signal方法会将队列的头部结点加入到锁的队列的尾部结点(并不会唤醒其他线程，这个操作由后续的unlock完成)。

### 状态管理

AQS中维护了一个state字段用于管理锁的状态，当它是0时，表明没有线程占有锁，当它不是0时，表明有线程占有了锁。如果state只有0/1，那么可以用于实现互斥锁(非重入)，如果state不止0/1，可以用于实现共享锁，也可以用于实现互斥重入锁。

AQS提供了几个方法操作这个state字段，我们自己使用AQS框架时主要也是利用这几个操作管理状态从而让AQS知道什么时候加锁，什么时候释放锁。



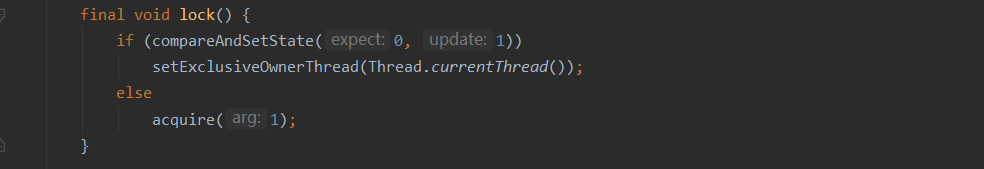
## 锁的源码解析

Java的很多并发工具都是使用的AQS实现的，比如ReentrantLock,CountDownLatch。

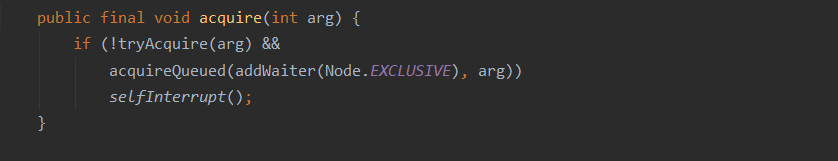
下面来看下ReentrantLock中非公平互斥锁的操作逻辑：

### lock

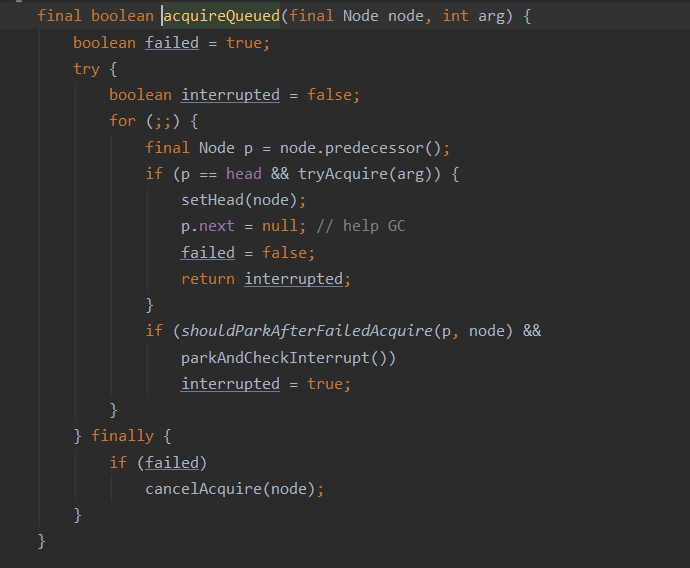
调用lock时，代码如下：



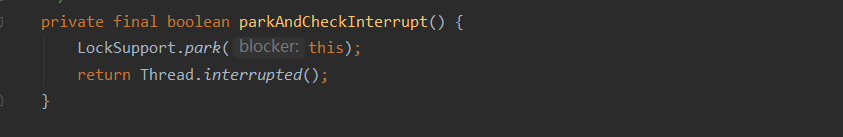
会先使用原子性操作尝试获取锁，如果能够获取锁，那么设置当前持有锁的线程，如果不能获取锁，那么调用acquire将线程加入到队列中。



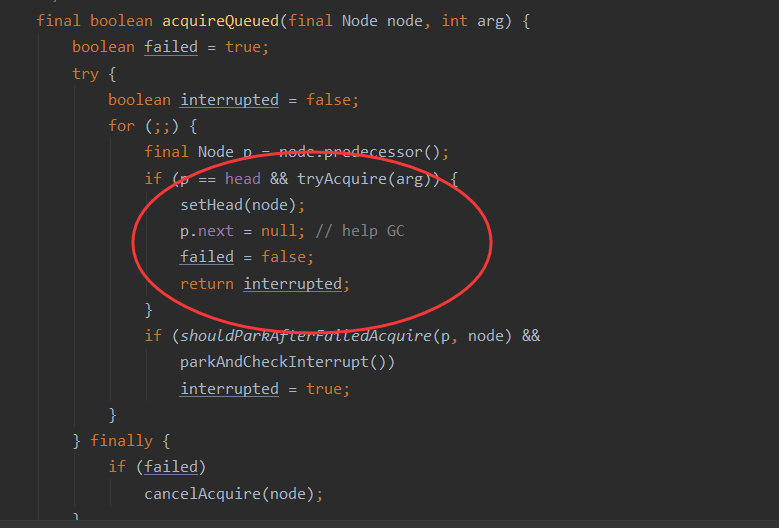
调用acquire会再次尝试获取锁，如果不能获取锁，那么调用addWaiter初始化一个结点并加入到队列中，然后调用acquireQueued排队等待获取锁。



如果此时其他线程还没有释放锁，那么进入到parkAndCheckInterrupt逻辑中，它会使当前线程睡眠；



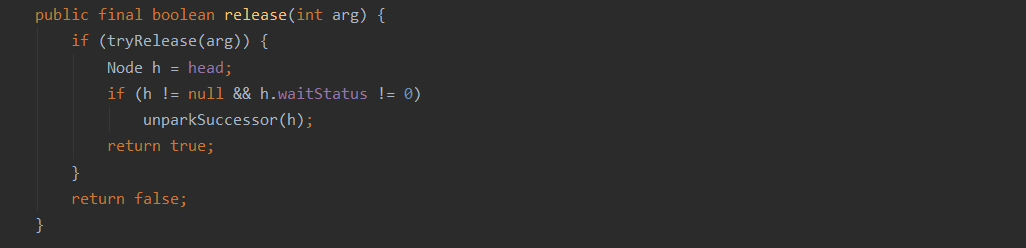
如果之后线程被唤醒，由于是循环，所以当线程苏醒后，那么执行下面的逻辑，它会使当前线程称为头部结点，并修改结点状态：



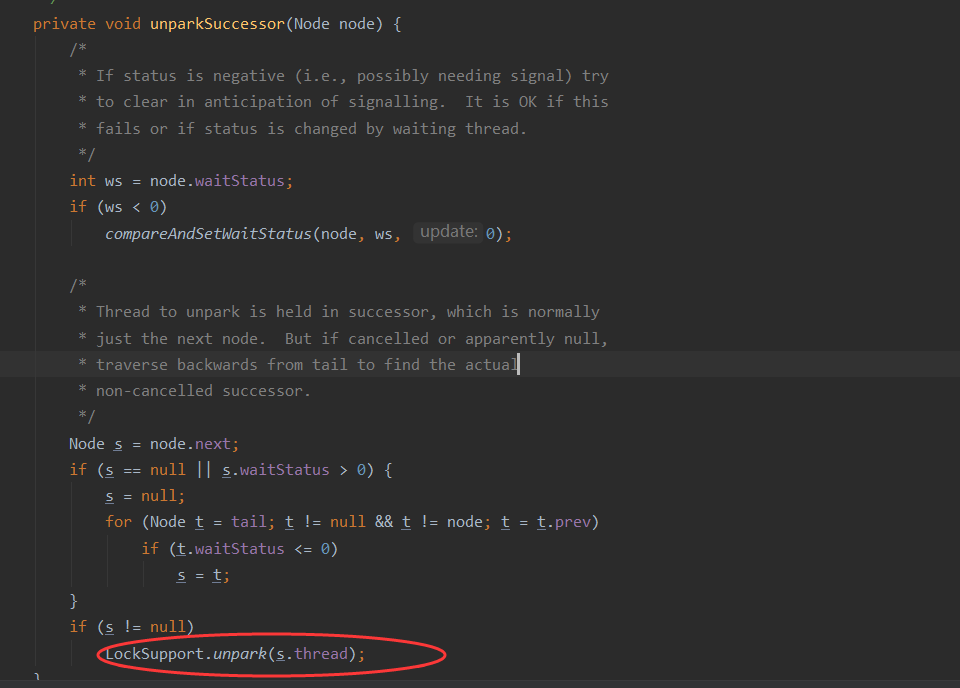
### unlock

调用unlock后，会调用release释放节点操作





释放节点时会调用unparkSuccessor唤醒头部结点中的线程：



## 条件变量的源码解析

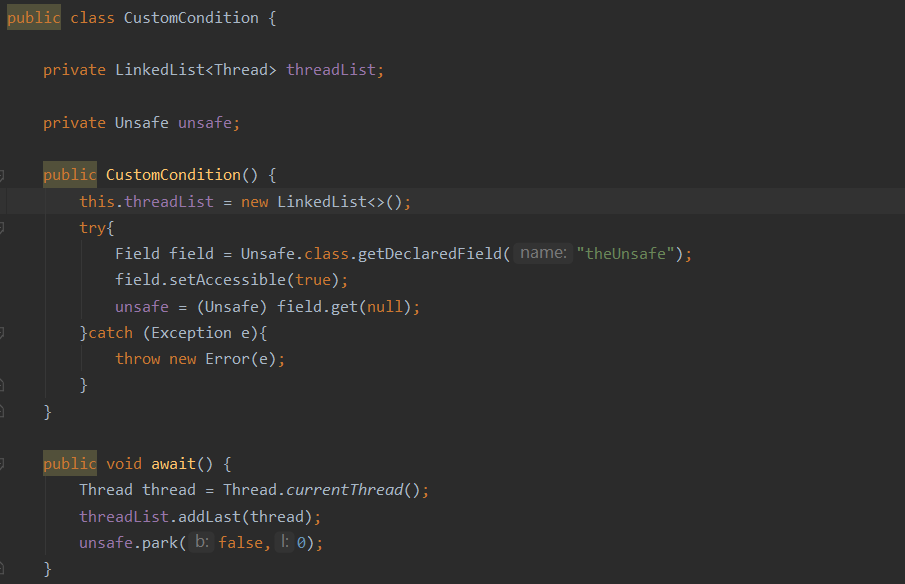
### 条件变量概述

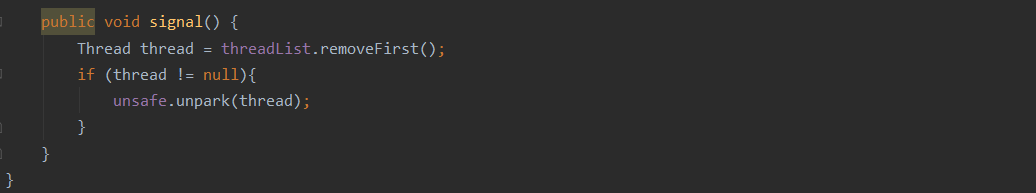
在并发场景中，锁并不是唯一解决问题的方式，还有其他问题是锁不能解决的，例如总会出现一个线程需要等待另一个线程执行某种操作后，或者等待某种条件成熟后才能继续执行的情况。所以需要使用条件变量。

线程可以使用条件变量，来等待一个条件变成真。条件变量是一个显式队列，当某些执行状态不满足时，线程可以把自己加入到队列，等待该条件。另外某个线程，当它改变了上述状态时，就可以唤醒一个或者多个等待的线程，让它们继续执行。

条件变量可以使用Unsafe提供的park和unpark实现。

**条件变量简单例子(只是示例，还有很多细节未考虑进去)：**





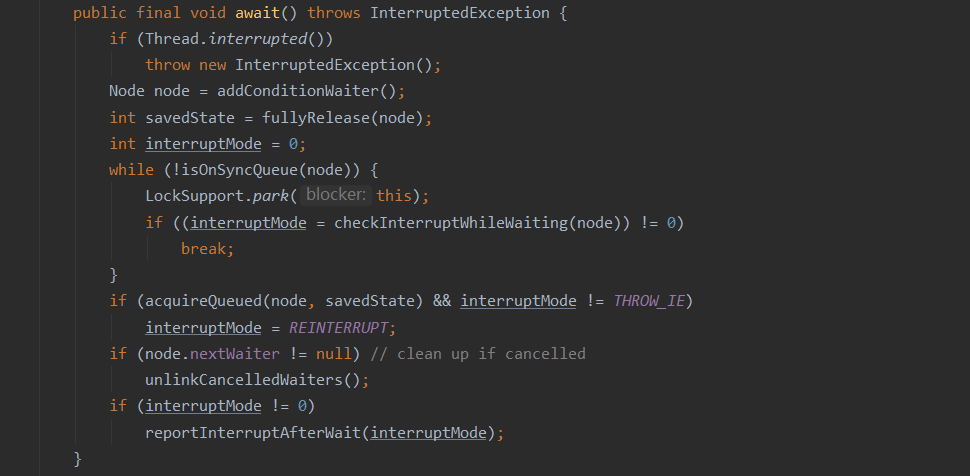
### AQS中的条件变量

AQS中实现了一个条件变量—ConditionObject。

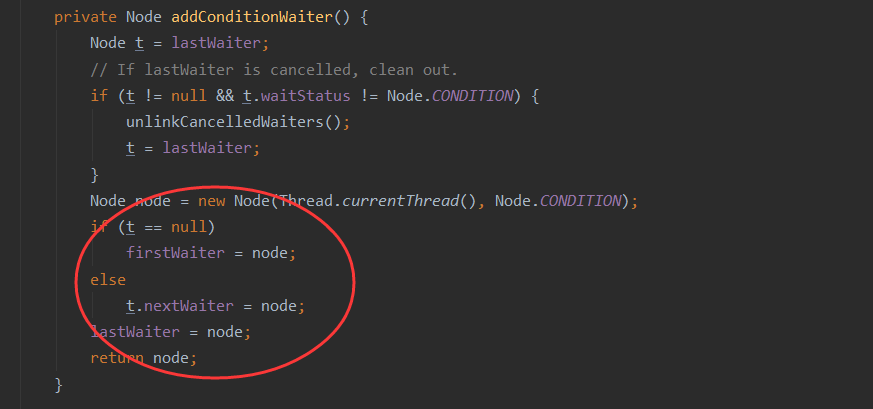


它内部存在一个FIFO队列，用于存储因调用await操作而陷入等待的线程。

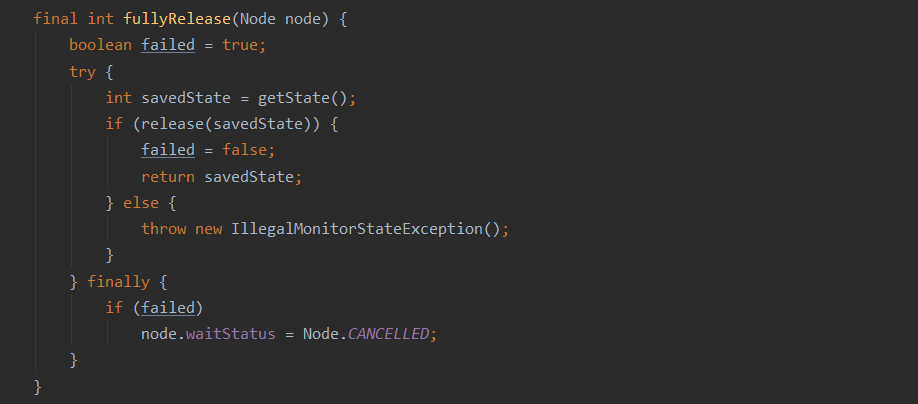
#### await

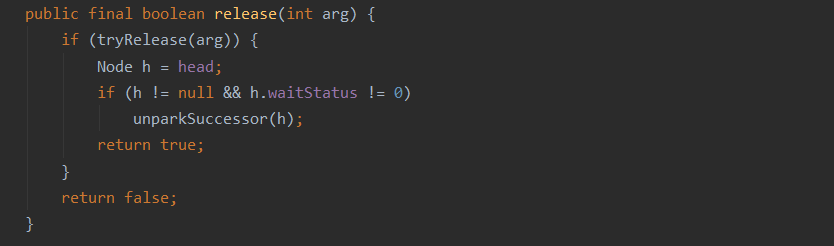


await操作中会调用addConditionWaiter操作生成一个结点，放入队列尾部中。



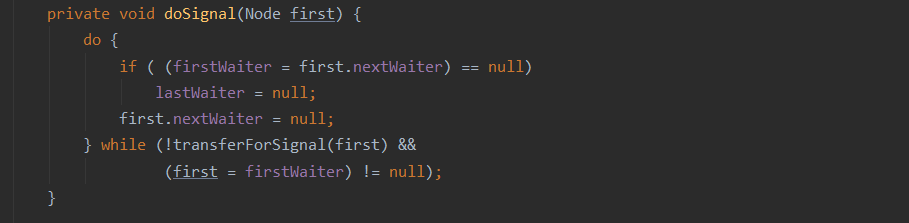
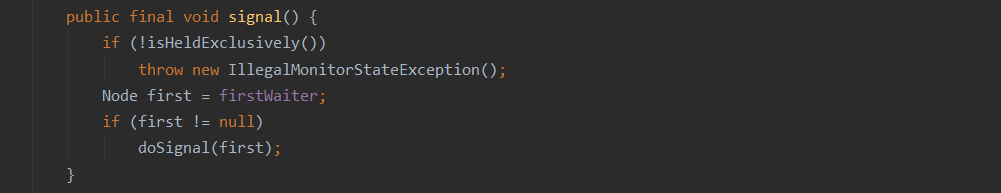
添加队列结点后，会唤醒锁的队列(注意非条件变量的队列)的头部结点。

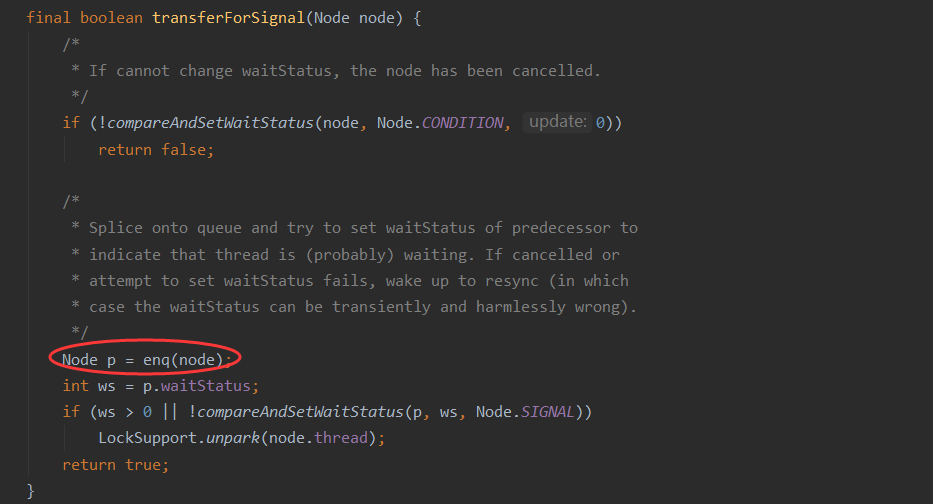


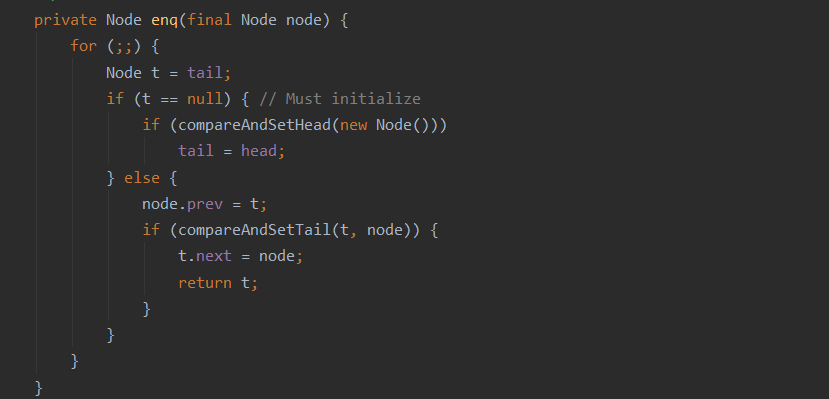


#### signal

调用signal操作，会通过transferForSignal操作会将条件变量的头部结点放入到锁的队列尾部中。这里正常情况是不会唤醒线程的，唤醒线程的操作由unlock来执行。







## AQS使用

下面简单使用AQS实现一个互斥重入锁。

