Java的内置锁一直都是备受争议的，在JDK 1.6之前，synchronized这个重量级锁其性能一直都是较为低下，虽然在1.6后，进行大量的锁优化策略,但是与Lock相比synchronized还是存在一些缺陷的：虽然synchronized提供了便捷性的隐式获取锁释放锁机制（基于JVM机制），但是它却缺少了获取锁与释放锁的可操作性，可中断、超时获取锁，且它为独占式在高并发场景下性能大打折扣。

**多线程同步内部如何实现的**

**wait/notify, synchronized, ReentrantLock。。。。。**

模拟一些同步的思路

**自旋实现同步**

volatile int status=0;//标识---是否有线程在同步块-----是否有线程上锁成功

void lock(){

while(!compareAndSet(0,1)){

}

*//lock*

//10 t1

}

void unlock(){

status=0;

}

boolean compareAndSet(int except,int newValue){

*//cas操作,修改status成功则返回true*

}

缺点：耗费cpu资源。没有竞争到锁的线程会一直占用cpu资源进行cas操作，假如一个线程获得锁后要花费Ns处理业务逻辑，那另外一个线程就会白白的花费Ns的cpu资源

思路：让得不到锁的线程让出CPU

**yield+自旋**

volatile int status=0;

void lock(){

while(!compareAndSet(0,1)){

yield();//自己实现

}

*//lock* logic

}

void unlock(){

status=0;

}

要解决自旋锁的性能问题必须让竞争锁失败的线程不空转,而是在获取不到锁的时候能把cpu资源给让出来，yield()方法就能让出cpu资源，当线程竞争锁失败时，会调用yield方法让出cpu。自旋+yield的方式并没有完全解决问题，当系统只有两个线程竞争锁时，yield是有效的。需要注意的是该方法只是当前让出cpu，有可能操作系统下次还是选择运行该线程，比如里面有2000个线程，想想会有什么问题？

**sleep+自旋**

volatile int status=0;

void lock(){

while(!compareAndSet(0,1)){

sleep(2);

}

*//lock*---10m

}

void unlock(){

status=0;

}

sleep的时间为什么是10？怎么控制呢？就是你是调用者其实很多时候你也不知道这个时间是多少？

**park+自旋**

volatile int status=0;

Queue parkQueue;//集合 数组 list

void lock(){

while(!compareAndSet(0,1)){

*//*

park();----

}

*//lock* 10分钟

。。。。。。

unlock()

}

void unlock(){

lock\_notify();

}

void park(){

*//将当期线程加入到等待队列*

parkQueue.add(currentThread);

*//将当期线程释放cpu* 阻塞 睡眠

releaseCpu();

}

void lock\_notify(){

//status=0

*//得到要唤醒的线程*头部线程

Thread t=parkQueue.header();

*//唤醒等待线程*

unpark(t);

}

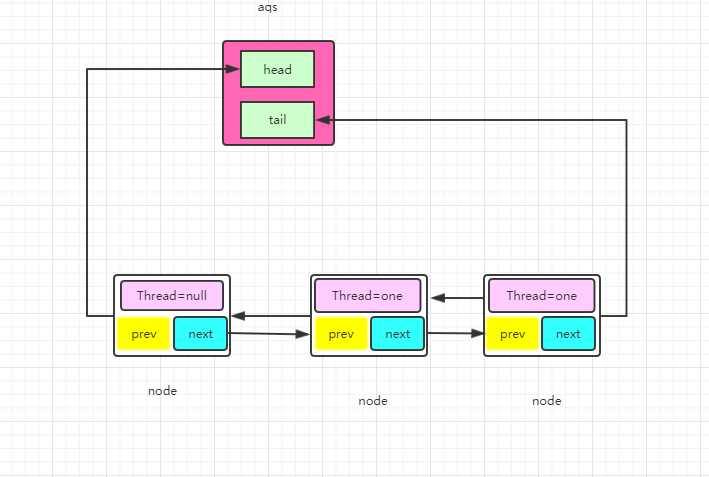
AQS（AbstractQueuedSynchronizer）类的设计主要代码（具体参考源码）

private transient volatile Node head; //队首

private transient volatile Node tail;//尾

private volatile int state;//锁状态，加锁成功则为1，重入+1 解锁则为0

AQS当中的队列示意图



Node类的设计

public class Node{

volatile Node prev;

volatile Node next;

volatile Thread thread;

int ws;

}

**上锁过程重点**

final ReentrantLock lock = new ReentrantLock(true);

Thread t1= new Thread("t1"){

@Override

public void run() {

lock.lock();

*logic*();

lock.unlock();

}

};

t1.start();

*锁对象：其实就是ReentrantLock的实例对象，上述代码第一行中的lock对象就是所谓的锁*

*自由状态：自由状态表示锁对象没有被别的线程持有，计数器为0*

*计数器：再lock对象中有一个字段state用来记录上锁次数，比如lock对象是自由状态则state为0，如果大于零则表示被线程持有了，当然也有重入那么state则>1*

*waitStatus：仅仅是一个状态而已；ws是一个过渡状态，在不同方法里面判断ws的状态做不同的处理，所以ws=0有其存在的必要性*

*tail：队列的队尾*

*head：队列的对首*

*ts：第二个给lock加锁的线程*

*tf：第一个给lock加锁的线程*

*tc：当前给线程加锁的线程*

*tl：最后一个加锁的线程*

*tn：随便某个线程 当然这些线程有可能重复，比如第一次加锁的时候tf==tc==tl==tn*

*节点：就是上面的Node类的对象，里面封装了线程，所以某种意义上node就等于一个线程*

**lock方法的逻辑**

final void lock() {

acquire(1);//1------标识加锁成功之后改变的值

}

**acquire方法的逻辑**

public final void acquire(int arg) {

//tryAcquire(arg)尝试加锁，如果加锁失败则会调用acquireQueued方法加入队列去排队，如果加锁成功则不会调用

//acquireQueued方法下文会有解释

//加入队列之后线程会立马park，等到解锁之后会被unpark，醒来之后判断自己是否被打断了，如果被打断了则执行selfInterrupt方法

//为什么需要执行这个方法？下文解释

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))

*selfInterrupt*();

}

**公平锁首先会调用tryAcquire去尝试加锁，当然这里的尝试加锁并不是直接加锁，事实上tryAcquire当中其实**

**第一步便是判断锁是不是自由状态，如果是则判断直接是否需要排队（**hasQueuedPredecessors方法判断队列是否被初始化（如果没有初始化显然不需要排队），和是否需要排队（队列如果被初始化了，则自己有可能需要排队）**）；如果**hasQueuedPredecessors**返回false，由于取反了故而不需要排队则进行Cas操作去上锁，如果需要排队则不会进入if分支当中，也不会进else if，会直接返回false表示加锁失败（为什么不进else if呢？这是java基础自己想想）**

**第二步如果不是自由状态再判断是不是重入，如果不是重入则直接返回false加锁失败，如果是重入则把计数器+1**

**当然我们这里说的都是公平锁的，那么非公平锁和公平锁的区别再哪里呢？下图说明了公平和非公平的区别**