JUC

进程：一个程序就是一个进程

线程：一个程序内开启同时进行的程序。

并发：多个线程操作同一份资源

并行：每个线程操作各自独立分开的资源，互不干涉

juc指的是三个包：

java.util.concurrent

java.util.concurrent.atomic

java.util.concurrent.locks

Interface Lock：

ReentrantLock可重入锁

JDK1.6之前syn是一个重量级锁，他需要调用操作系统的函数。

操作系统将线程停止，这个过程系统需要从{**用户态}**转换为{**内核态}**。所以执行效率低。

JDK1.7之后，线程同步概念尽量在JVM级别解决了，减少了操作系统的操作。

自旋锁：

缺点，自旋锁while循环耗费cpu，为了降低cpu耗费，需要在while循环中添加降低cpu占用的方法：

在while循环中可以添加Thread.yield方法来降低cpu调用率。但是yield的是由cpu进行调度的，不能保证代码的可控性。

在while循环中使用Thread.sleep方法，但是sleep的睡眠时间无法保证。睡眠时间过短不起作用，睡眠时间长阻塞浪费。

在while循环中使用LockSupport.park方法

park方法：LockSupport.park();这里的LockSupport是工具包，对park方法的封装。它实际调用的是Unsafe类里面的park方法。

Unsale实际上是java高级里面的方法。它甚至可以申请堆外内存。

唤醒方法LockSupport.unpark(【唤醒的线程Thread对象】);

wait主要目地是为了线程通讯，阻塞只是其中一个小功能，wait必须要在syn代码块中一起使用，否则会报错。

不推荐用wait来实现阻塞。

wait的唤醒方法notify()、notifyAll();唤醒一个和唤醒全部。但是被唤醒的是哪个无法保证。

实现代码🡪

普通自旋锁

volatile int lock = 0;

void lock(){

while (compareAndSet(0,1)){//compareAndSet是一个CAS，比较赋值，它具有原子性。

Thread.yield();//保证线程让出资源，降低cpu占用率。

或者用

Thread.sleep(10)；

};

}

void ulock(){

lock =1;

}

自旋+park+队列 来实现锁：

volatile int lock = 0;

Queue parkQueue;//队列

void lock(){

while (compareAndSet(0,1)){ //compareAndSet是一个CAS，比较赋值，它具有原子性。

//将当前线程加入到队列。

parkQueue.add(currentThread);

//让线程阻塞的方法实现。

LockSupport.park();

};

}

void ulock(){

//得到要唤醒的线程

Thread t = parkQueue.header();

LockSupport.unpark(t);

lock=1;

}

ReentrantLock：

AQS：abstractQueuedSynchronizer

ReentrantLock：

私有成员变量：（锁抽象类继承自AQS）

private final Sync sync;🡪

抽象锁的实现类：

1.FairSync公平锁

线程执行前先CAS判断是否能拿到锁。

拿不到锁就进入队列等待

为了线程在释放锁时，恰巧又有线程进入，导致新进入的线程抢到了本应给队列中等待线程的锁，不能直接CAS，如果直接CAS就是非公平锁

所以公平锁在取锁前还会判断队列中是否有等待的线程。只有当队列中没有节点，且锁也处在空闲状态下才会拿到锁。否则进入队列等待。

AQS：

成员变量：head（头节点），tail（尾节点），state（锁状态），exclusiveOwnerThread（当前持有锁的线程）

重入锁实现：拿到当前线程和当前持有锁的线程，如果当前线程等于持有锁线程相同则可以重入。

state+1；表示重入次数。

2.NofairSync非公平锁

并发代码有两种情况，第一种情况是竞争执行，另一种是交替执行。

竞争执行是指代码在