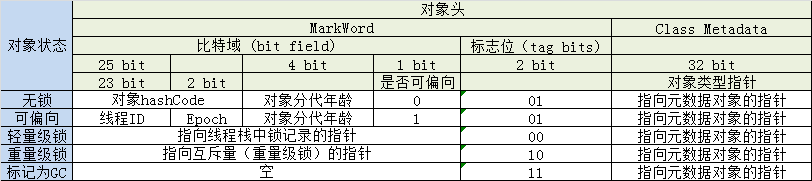
# **一些知识：**

## 1、CAS指令

该指令是是原子性的， 也就是说 CPU 执行该指令时， 是不会被中断执行其他指令的

该指令概念上存在 3 个参数， 第一个参数【目标地址】， 第二个参数【值1】， 第三个参数【值2】， 指令会比较【目标地址存储的内容】和 【值1】 是否一致， 如果一致， 则将【值2】 填写到【目标地址】

## 2、对象头的Mark Word

****

## 3、可重入锁

同一个线程可以多次获取同一把锁。ReentrantLock和synchronized都是可重入锁

## 4、可中断锁

线程尝试获取锁的过程中，是否可以响应中断。synchronized是不可中断锁，而ReentrantLock则提供中断功能

## 5、公平锁与非公平锁

平锁是指多个线程同时尝试获取同一把锁时，获取锁的顺序按照线程达到的顺序，而非公平锁则允许线程“插队”。synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁。

# **synchronized关键字**

## 一、作用

1、确保线程互斥的访问同步代码

2、保证共享变量的修改能够及时可见

3、有效解决重排序问题

## 二、用法

1、修饰普通方法：对同一个对象（的方法）进行同步（可与其他对象并行）。但是其他线程还是可以访问该实例对象的其他非synchronized方法

2、修饰静态方法：对同一个类（的静态方法）进行同步。对静态方法的同步本质上是对类的同步，因为静态方法本质上是属于类的方法，而不是对象上的方法。（可与其他类并行）

3、修饰代码块：对同一个对象或类（的代码块）进行同步

## 三、原理

synchronized是一个重量级锁

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.（可重入）

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

注意：

1、代码块（monitorenter和monitorexit）

2、方法级的同步是隐式，即无需通过字节码指令来控制的。ACC\_SYNCHRONIZED：该标识指明了方法是否为一个同步方法

## 四、低效的原因（重量级锁）

监视器锁本质是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间。

## 五、等待唤醒机制

指notify/notifyAll和wait方法，在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

## 六、锁的四种状态（锁的升级是单向的，只能从低到高升级）

### 1、无锁状态

### 2、偏向锁

场景：在只有一个线程执行同步块时进一步提高性能。只需要在置换ThreadID的时候依赖一次CAS原子指令

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。

失败的情况：存在其他线程申请锁（锁竞争）

### 3、轻量级锁00

场景：线程交替执行同步块。目的：减少无实际竞争情况下，使用重量级锁产生的性能消耗。

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

失败的情况：存在锁竞争激烈

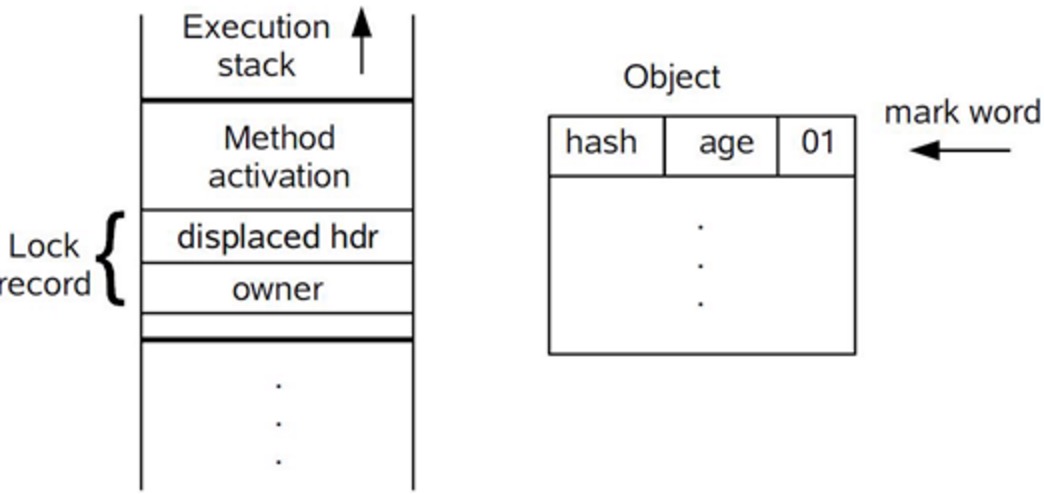
### 4、重量级锁10

当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。

## 七、加锁过程

### 1、轻量级锁的加锁过程

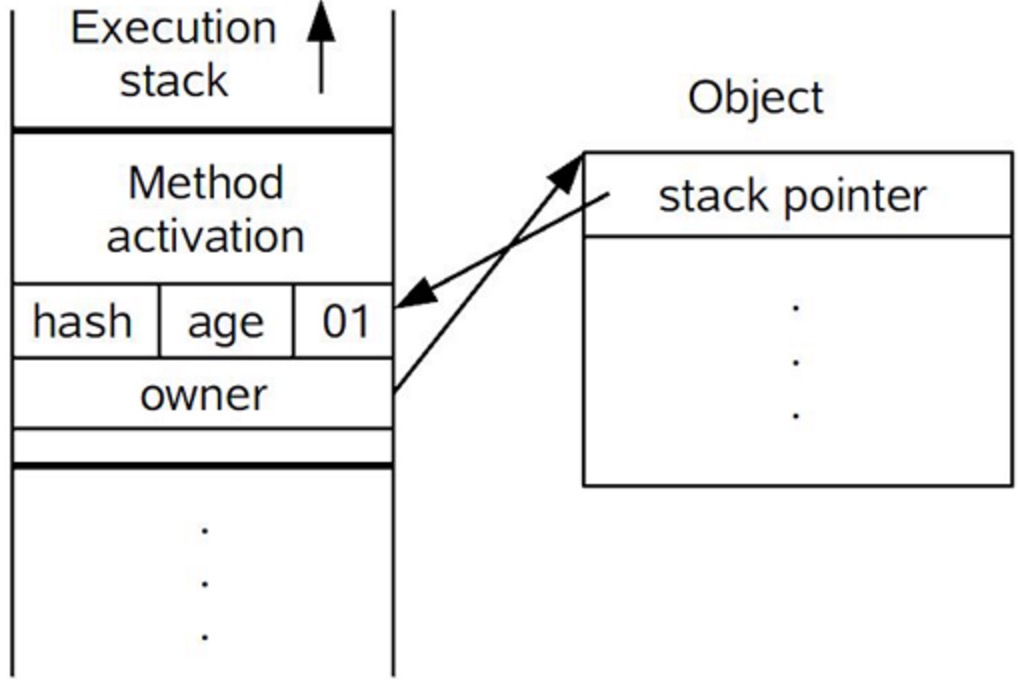
（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位：01，是否为偏向锁：0），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为Displaced Mark Word。这时候线程堆栈与对象头状态如下图：



（2）拷贝对象头中的Mark Word到锁记录中

（3）拷贝成功后，虚拟机将使用CAS操作将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指向object mark word。如果更新成功，则执行步骤4，否则执行步骤5。

（4）如果这个更新操作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为00，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如下图：



(5)如果这个更新操作失败，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志变为10，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞而采取循环获取锁的过程。

### 2、轻量级锁的解锁过程

（1）通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word

（2）如果成功，整个同步过程就完成了

（3）如果失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

### 3、偏向锁的加锁过程

（1）访问Mark Word中偏向锁的标志是否设置成1，锁标志位是否为01（确认可偏向状态）

（2）如果为可偏向状态

* 如果CAS操作成功，则认为已经获取到对象的偏向锁，执行同步代码

补充：一个线程在执行完同步代码块以后，并不会尝试将Mark Word中的Thread ID赎回原值。这样做的好处是：如果该线程需要再次对这个对象加锁，而这个对象之前一直没有被其他线程尝试获取过锁，依旧停留在可偏向的状态下，即可以不修改对象头的情况下，直接认为偏向成功。

* 如果CAS操作失败，则说明有另外一个线程Thread B抢先获取了偏向锁。这种状态说明该对象的竞争比较激烈，此时需要撤销Thread B获得的偏向锁，将Thread B持有的锁升级为轻量级锁。该操作需要等待全局安全点JVM safepoint（此时间点，没有线程在执行字节码）

（3）如果是已偏向状态，则检测Mark Word中存储的Thread ID是否等于当前Thread ID

* 如果相等，则证明本线程已经获取到偏向锁，可以直接继续执行同步代码块
* 如果不等，则证明该对象目前偏向于其他线程，需要撤销偏向锁

### 4、偏向锁的撤销

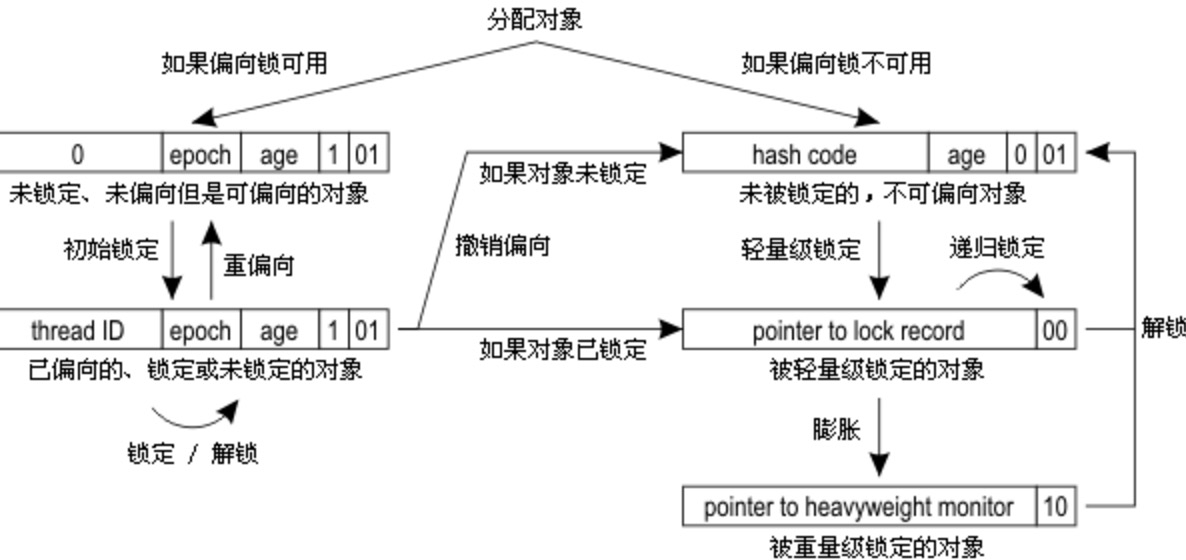
偏向锁的撤销并不是将对象恢复到无锁可偏向的状态，而是在偏向锁的获取过程中，发现竞争时，直接将一个被偏向的对象升级到被加了轻量级锁的状态

* 在偏向锁CAS更新操作失败以后，等待到达全局安全点

● 通过Mark Word中已经存在的Thread ID找到成功获取了偏向锁的那个进程，然后在该线程的栈帧中补充上轻量级加锁时，会保存的锁记录，然后将获取了偏向锁的对象的Mark Word更新为指向这条锁记录的指针。

● 至此，锁撤销操作完成，阻塞在安全点的线程可以继续执行。

### 5、锁的相互转换



## 八、其他优化

### 1、自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

缺点：自旋需要消耗CPU（单处理器情况下，完全是浪费），最好指定自旋次数或使用jdk的适应性自旋：线程如果成功了，就增加下次的自旋次数，失败则减少自旋次数。

### 2、锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

### 3、锁粗化

将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。

## 九、局限性

1、当线程尝试获取锁的时候，如果获取不到锁会一直阻塞。

2、如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待。

## 十、总结

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗。 | 适用于只有一个线程访问同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU。 | 追求响应时间。  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | 线程阻塞，响应时间缓慢。 | 追求吞吐量。  同步块执行速度较长。 |

# UnSafe类

## 描述

Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的原子操作。Unsafe类使Java拥有了像C语言的指针一样操作内存空间的能力，同时也带来了指针的问题。过度的使用Unsafe类会使得出错的几率变大，因此Java官方并不建议使用的，官方文档也几乎没有。绕过JVM调用JNI，需要手动释放内存。

## 功能

### 一、内存管理

### 二、非常规的对象实例化

### 三、操作类、对象、变量

### 四、数组操作

### 五、多线程同步。包括锁机制、CAS操作等

### 六、挂起与恢复。包括了park、unpark等方法

### 七、内存屏障

# CAS

## 描述

Compare and Swap即比较并交换，设计并发算法时常用到的一种技术，java.util.concurrent包全完建立在CAS之上，没有CAS也就没有此包，可见CAS的重要性。

CAS也是通过Unsafe实现的，由于CAS都是硬件级别的操作，因此效率会高一些。

## 操作

内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。

# ReentrantLock

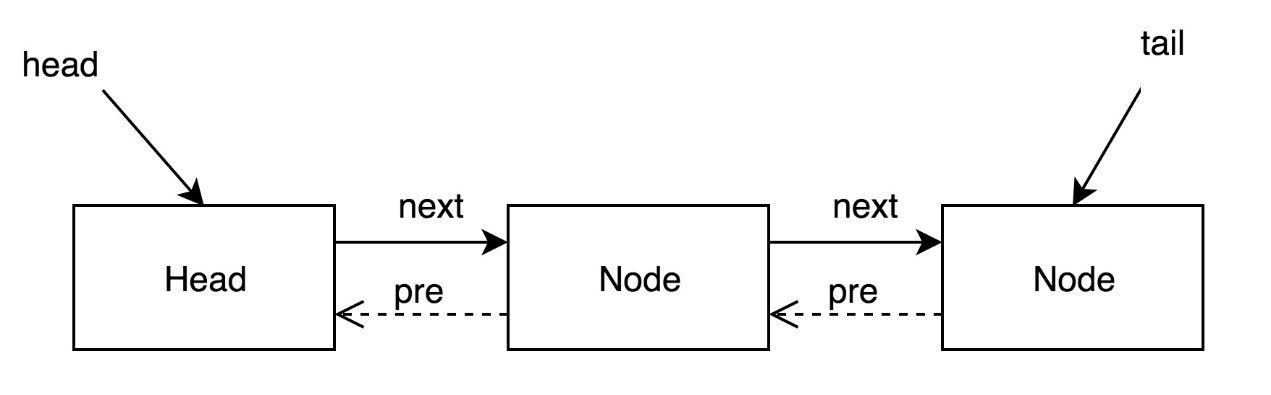
## 一、基于AQS实现

### 描述

AbstractQueuedSynchronizer简称AQS，是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。

### 原理

AQS使用一个FIFO的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态waitStatus。如图：



## 二、内部结构

默认构造器初始化为NonfairSync对象，即非公平锁，而带参数的构造器可以指定使用公平锁和非公平锁。由lock()和unlock的源码可以看到，它们只是分别调用了sync对象的lock()和release(1)方法。

可以看到Sync扩展了AbstractQueuedSynchronizer。



### 1、非公平锁（默认）

#### （1）lock（）

首先用一个CAS操作，判断state是否为0（表示当前锁未被占用），如果是0则把它置为1，并且设置当前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS操作只能保证一个线程的成功，剩下的只能排队。

“非公平”即体现在这里，如果占用锁的线程刚释放了锁，state置为0，而排队等待锁的线程还未唤醒时，新来的线程就直接抢占了该锁，那么就“插队”了。

##### 获取成功

1、设置AbstractQueuedSynchronizer的state为1

2、设置AbstractOwnableSynchronizer的thread为当前线程

###### lock()：将当前线程设置为独占线程，state修改为1

final void lock**()** **{**

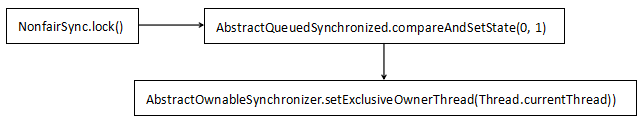
**if** **(**compareAndSetState**(**0**,** 1**))**// 修改state为1

setExclusiveOwnerThread**(**Thread**.**currentThread**());**// 把当前线程设置为独占线程

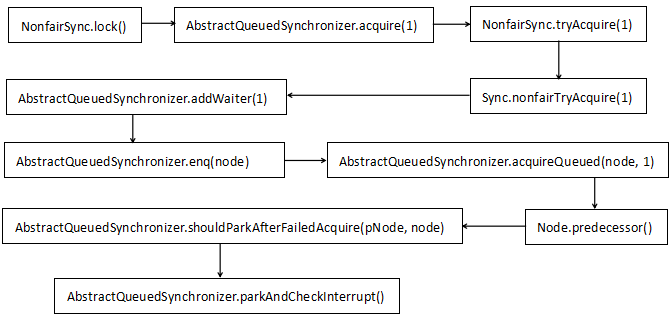
**else**

acquire**(**1**);**// 修改state失败

**}**



##### 获取失败



###### acquire()：1、再次尝试获取锁（因为可能是被本线程占有，也可能已经锁被释放）2、反之，添加到FIFO等待队列

public final void acquire**(**int arg**)** **{**

**if** **(!**tryAcquire**(**arg**)** **&&**// 尝试获取一次锁

acquireQueued**(**addWaiter**(**Node**.**EXCLUSIVE**),** arg**))**// 添加到FIFO等待队列

selfInterrupt**();**

**}**

###### nonfairTryAcquire()：判断是否为可重入锁，或锁被释放

此处公平锁和非公平锁代码大致相同，公平锁多了hasQueuedPredecessors()：关于真正头节点的判断

final boolean nonfairTryAcquire**(**int acquires**)** **{**

final Thread current **=** Thread**.**currentThread**();**

int c **=** getState**();**

**if** **(**c **==** 0**)** **{**// 再次判断，因为线程1可能已经释放了锁

**if** **(**compareAndSetState**(**0**,** acquires**))** **{**// 设置state

setExclusiveOwnerThread**(**current**);**// 设置当前线程为独占线程

**return** **true;**

**}**

**}**

**else** **if** **(**current **==** getExclusiveOwnerThread**())** **{**// 可重入，相当于偏向锁，不需要CAS

int nextc **=** c **+** acquires**;**

**if** **(**nextc **<** 0**)** // 错误，最多可以冲入int的最大值，然后变成负数

**throw** **new** Error**(**"Maximum lock count exceeded"**);**

setState**(**nextc**);**

**return** **true;**

**}**

**return** **false;**

**}**

###### addWaiter()：添加节点到队列中

private Node addWaiter**(**Node mode**)** **{**

Node node **=** **new** Node**(**Thread**.**currentThread**(),** mode**);**// 独占模式

// Try the fast path of enq; backup to full enq on failure

Node pred **=** tail**;**

**if** **(**pred **!=** **null)** **{**// 非空队列，添加至尾节点

node**.**prev **=** pred**;**

**if** **(**compareAndSetTail**(**pred**,** node**))** **{**

pred**.**next **=** node**;**

**return** node**;**

**}**

**}**

enq**(**node**);**// 空队列，则创建一个队列

**return** node**;**

**}**

###### enq()：创建新的队列。因为多线程下头节点可能已经被更新

private Node enq**(**final Node node**)** **{**// 创建新队列

**for** **(;;)** **{**// 多线程下，头节点可能已经更新。所以要循环

Node t **=** tail**;**

**if** **(**t **==** **null)** **{**

Node h **=** **new** Node**();** // 头节点什么都没有

h**.**next **=** node**;**

node**.**prev **=** h**;**

**if** **(**compareAndSetHead**(**h**))** **{**

tail **=** node**;**

**return** h**;**

**}**

**}**

**else** **{**// 尾节点不为空，说明其他线程已经创建了队列

node**.**prev **=** t**;**

**if** **(**compareAndSetTail**(**t**,** node**))** **{**

t**.**next **=** node**;**

**return** t**;**

**}**

**}**

**}**

**}**

###### acquireQueued()：判断真正头节点是否可以执行

final boolean acquireQueued**(**final Node node**,** int arg**)** **{**

**try** **{**

boolean interrupted **=** **false;**

**for** **(;;)** **{**

final Node p **=** node**.**predecessor**();**// 判断是否为队列中第一个真正的节点

**if** **(**p **==** head **&&** tryAcquire**(**arg**))** **{**// 再次判断锁是否被释放

setHead**(**node**);**// 把当前节点设置为头node

p**.**next **=** **null;** // help GC

**return** interrupted**;**

**}**

**if** **(**shouldParkAfterFailedAcquire**(**p**,** node**)** **&&**

parkAndCheckInterrupt**())**

interrupted **=** **true;**

**}**

**}** **catch** **(**RuntimeException ex**)** **{**

cancelAcquire**(**node**);**

**throw** ex**;**

**}**

**}**

###### parkAndCheckInterrupt()：阻塞当前的线程

private final boolean parkAndCheckInterrupt**()** **{**

LockSupport**.**park**(this);**

**return** Thread**.**interrupted**();**

**}**

public static void park**(**Object blocker**)** **{**

Thread t **=** Thread**.**currentThread**();**

setBlocker**(**t**,** blocker**);**

UNSAFE**.**park**(false,** 0L**);**

setBlocker**(**t**,** **null);**

**}**

#### （2）unlock()

public void unlock**()** **{**

sync**.**release**(**1**);**

**}**

##### release()

public final boolean release**(**int arg**)** **{**

**if** **(**tryRelease**(**arg**))** **{**

Node h **=** head**;**

**if** **(**h **!=** **null** **&&** h**.**waitStatus **!=** 0**)**

unparkSuccessor**(**h**);**// 恢复被阻塞的线程

**return** **true;**

**}**

**return** **false;**

**}**

##### tryRelease()：尝试释放锁

protected final boolean tryRelease**(**int releases**)** **{**

int c **=** getState**()** **-** releases**;**

**if** **(**Thread**.**currentThread**()** **!=** getExclusiveOwnerThread**())**

**throw** **new** IllegalMonitorStateException**();**

boolean free **=** **false;**

**if** **(**c **==** 0**)** **{**// (可重入)锁全部解开

free **=** **true;**

setExclusiveOwnerThread**(null);**// 独占线程设置为null

**}**

setState**(**c**);**// 反之可重入锁-1

**return** free**;**

**}**

##### unparkSuccessor()：释放第一个被阻塞的线程

private void unparkSuccessor**(**Node node**)** **{**

/\*

\* Try to clear status in anticipation of signalling. It is

\* OK if this fails or if status is changed by waiting thread.

\*/

compareAndSetWaitStatus**(**node**,** Node**.**SIGNAL**,** 0**);**

/\*

\* Thread to unpark is held in successor, which is normally

\* just the next node. But if cancelled or apparently null,

\* traverse backwards from tail to find the actual

\* non-cancelled successor.

\*/

Node s **=** node**.**next**;**// 真正第一个节点是头节点的下一个

**if** **(**s **==** **null** **||** s**.**waitStatus **>** 0**)** **{**// 线程可能因中断而取消，变为null

s **=** **null;**

**for** **(**Node t **=** tail**;** t **!=** **null** **&&** t **!=** node**;** t **=** t**.**prev**)**

**if** **(**t**.**waitStatus **<=** 0**)**// 逆序找到第一个被阻塞的线程

s **=** t**;**

**}**

**if** **(**s **!=** **null)**// 恢复被阻塞的线程

LockSupport**.**unpark**(**s**.**thread**);**

**}**

## 三、注意

### 1、头节点

为什么不用一个在等待的线程作为HeadNode？

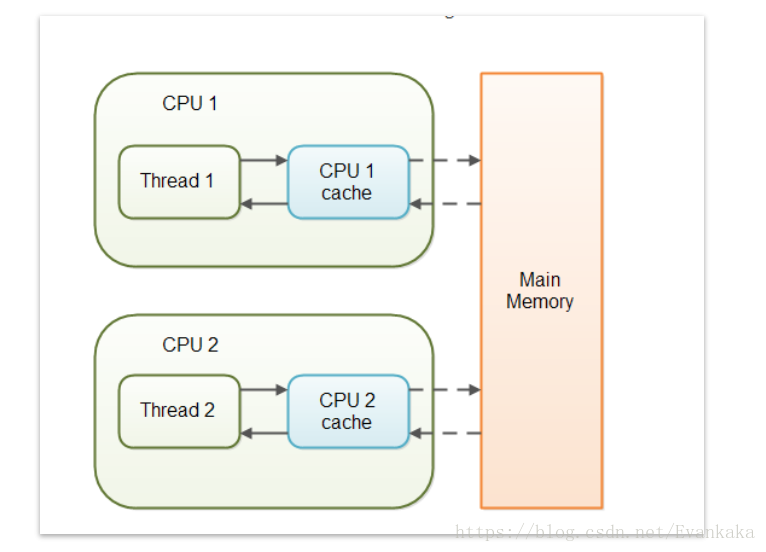
因为一个线程随时有可能因中断而取消，以至于被GC变为null。

### 2、如何区分公平和非公平

非公平在lock时候，会多进行一次CAS操作进行“插队”

# volatile关键字

访问volatile变量时无加锁操作，这个变量是共享的。是比synchronized关键字更轻量级的同步机制。可用于线程通信。



## 非volatile变量

当对非volatile变量进行读写操作的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这也就意味着每个线程可以拷贝到不同的CPU cache中。

## volatile变量

1、JVM保证了每次读取变量都是从内存中读

2、保证此变量对所有内存的可见性

3、禁止指令重排序优化

## 性能

读性能消耗与普通变量几乎相同，写操作稍慢：内存屏障

## 注意

1、volatile修饰后依然需要将变量拷贝到工作内存中（栈内存）。但是写操作会被立即刷新到主内存中，并且强制让缓存了该变量的线程中的数据清空，必须从堆内存中重新读取最新数据

2、不能保证线程安全

## 使用条件

1、对变量的写入操作不依赖当前值

2、该变量没有包含在具有其他变量的不变式中