可重入与不可重入的理解

# 自旋锁、阻塞锁、可重入锁与非可重入锁、偏向锁、轻量级锁、重量级锁、公平锁与非公平锁

## 自旋锁

采用让**当前线程不停的在循环体内执行实现**，当循环的条件被其它线程改变时才能进入临界区。如基于CAS实现的ReentrantLock可重入锁就是一种自旋锁。

优缺点：

**优点：响应速度快**；由于自旋锁只是将当前线程不停地执行循环体，不进行线程状态的改变，所以响应速度更快。

**缺点：吞吐量低，多线程下性能较低。**但当线程数不停增加时，性能下降明显，因为每个线程都需要执行，占用CPU时间。如果线程竞争不激烈，并且保持锁的时间段，适合使用自旋锁。

## 阻塞锁

阻塞锁改变了**线程的运行状态**，让线程进入阻塞状态进行等待，当获得相应的信号（唤醒或者时间）时，才可以进入线程的准备就绪状态，转为就绪状态的所有线程，通过竞争，进入运行状态。

**优缺点分析**：阻塞锁的优势在于，**阻塞的线程不会占用cpu时间，不会导致 CPU占用率过高**，但进入时间以及恢复时间都要比自旋锁略慢。在竞争激烈的情况下，阻塞锁的性能要明显高于自旋锁。

传统的synchronized就是利用阻塞锁，但是自从synchronized优化后，也引入了自旋锁。

## 可重入锁与非可重入锁：见下面。

## 偏向锁、轻量锁、重量锁

## 公平锁与非公平锁

synchronized是非公平锁；

ReentrantLock可以设置公平或非公平，默认为**非公平**。

# 可重入锁与不可重入锁：

锁分为**可重入锁**和**不可重入锁**。

可重入和不可重入的概念是这样的：当一个线程获得了当前实例的锁，并进入方法A，这个线程在没有释放这把锁的时候，能否再次进入方法A呢？

* **可重入锁**：可以再次进入方法A，就是说在释放锁前此线程可以再次进入方法A（方法A递归）。
* **不可重入锁（自旋锁）**：不可以再次进入方法A，也就是说获得锁进入方法A是此线程在释放锁前唯一的一次进入方法A。

狭义的可重入锁指的就是JDK中的**ReentrantLock**锁。

广义上的可重入锁是具有可重入性的锁，在java 中，**synchronized**和**ReentrantLock**都是**可重入锁**。

下面主要是分析广义的可重入锁：

可重入锁，也叫做递归锁，指的是同一线程外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。

可重入锁最大作用(优点)是**避免死锁**。缺点:必须手动开启和释放锁。

可重入的自我理解：可重入的主语是已经获得该锁的线程，可重入指的就是可以再次进入，因此，意思就是**已经获得该锁的线程可以再次进入被该锁锁定的代码块**。

# 举例说明可重入与不可重入

伪代码：

public class UnReentrant{

**Lock lock = new Lock();**

public void outer(){

lock.lock();

inner();

lock.unlock();

}

public void inner(){

lock.lock();

//do something

lock.unlock();

}

}

outer中调用了inner，outer先锁住了lock，这样inner就不能再获取lock。其实调用outer的线程已经获取了lock锁，但是不能在inner中重复利用已经获取的锁资源，这种锁即称之为 **不可重入锁 。**通常也称为 自旋锁spinLock 。相对来说，**可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块**。

# 可重入锁与不可重入锁的基本原理

## 不可重入锁UnReentrantLock的原理：

public class **UnReentrantLock** {

**private boolean isLocked = false;**

**public void lock() {**

**while(isLocked) {**

**//wait();**

**}**

**isLocked = true;**

**}**

**public void unlock() {**

**isLocked = true;**

**//notify();**

**}**

}

测试函数：

public static void main(String[] args) {

UnReentrantLock unReentrantLock = new UnReentrantLock();

unReentrantLock.lock();

System.out.println("第一次进入");

unReentrantLock.lock();//阻塞

System.out.println("第二次进入");

unReentrantLock.unlock();

unReentrantLock.unlock();

System.out.println("结束了");

}

结果：



## 可重入锁的基本原理

**对于同一线程进入同一锁对应的代码块的次数进行统计**。

变量说明：

lockedByThread：保存已经获得锁实例的线程，在lock()判断调用lock的线程是否已经获得当前锁实例，如果已经获得锁，则直接跳过while，无需等待。

count：记录该获取锁的线程重复对该锁对象加锁的次数。否则，一次unlock就会解除所有锁，即使这个锁实例已经加锁多次了。

* lock时：只有锁已经被占据且当前线程不是获取锁的线程的时候，一直等待；其他情况，锁未被占据或当前线程是获取锁的线程，不需要等待。
* unlock：只要当前线程是获取锁的线程，计数值减一，若减为0，则锁修改为未被锁定，获取锁的线程为null。

import java.util.concurrent.atomic.AtomicInteger;

public class MyReentrantLock {

boolean isLocked = false;

AtomicInteger count = new AtomicInteger(0);

Thread lockedByThread = null;

public void lock() {

while(isLocked && Thread.currentThread() != lockedByThread) {

//锁已经被占，且占领的线程不是本线程就阻塞；否则不需要阻塞(锁没有被占或占领的线程就是当前线程)

//wait();

}

isLocked = true;

count.incrementAndGet();

lockedByThread = Thread.currentThread();//注意：一个锁只要是被锁定isLocked为，那么只能被一个线程所拥有

}

public void unlock() {

if(Thread.currentThread() == lockedByThread) {

//只要是进入的线程是获取锁的线程，计数减1，只有计数值为0时，释放锁

count.decrementAndGet();

if(count.get() == 0) {//

isLocked = false;

lockedByThread = null;

// notify();

}

}

}

}

测试函数：

public static void main(String[] args) {

MyReentrantLock myReentrantLock = new MyReentrantLock();

myReentrantLock.lock();

System.out.println("第一次进入");

myReentrantLock.lock();

System.out.println("第二次进入");

myReentrantLock.unlock();

myReentrantLock.unlock();

System.out.println("结束了");

}

结果：



# 代码实现可重入锁与不可重入锁

## 不可重入自旋锁示例

import java.util.concurrent.atomic.AtomicReference;

public class **UnReentrantLock** {

//原子引用类

private AtomicReference<Thread> atomicReference = new AtomicReference<Thread>();

public void lock() {

//采用CAS操作完成

while(!atomicReference.compareAndSet(null, Thread.currentThread())) {};

}

public void unlock() {

while(!atomicReference.compareAndSet(Thread.currentThread(),null)) {};

}

//测试

public static void main(String[] args) {

UnReentrantLock unReentrantLock = new UnReentrantLock();

unReentrantLock.lock();

System.out.println("进入第一层");

unReentrantLock.lock();//一直停留在这里，自旋

System.out.println("进入第二层");

unReentrantLock.unlock();

unReentrantLock.unlock();

System.out.println("结束了");

}

}

原理介绍：这里使用原子类AtomicReference的CAS操作的自旋(while循环)。首先第一个CAS操作成功，修改为当前线程对象，然后再进行CAS操作会一直while循环。这就是不可重入了。

对于不可重入自旋锁来说，若有同一线程两次调用lock() ，会导致第二次调用lock位置进行**自旋**，产生了死锁；**说明这个锁并不是可重入的**。（在lock函数内，应验证线程是否为已经获得锁的线程）。

## 改进：可重入自旋锁spinLock；利用计数实现。

import java.util.concurrent.atomic.AtomicReference;

public class **SpinLock** {

// 利用计数器实现的自旋锁是一个可重入锁

private AtomicReference<Thread> owner = new AtomicReference<Thread>();

private int count = 0;

public void lock() {

Thread current = Thread.currentThread();

if (current == owner.get()) {

count++;

return;

}

while (!owner.compareAndSet(null, current)) {}

}

public void unlock() {

Thread current = Thread.currentThread();

if (current == owner.get()) {

if (count != 0) {

count--;

} else {

owner.compareAndSet(current, null);

}

}

}

}

原理：利用计数的方法，可以实现可重入。因为再次调用lock方法的时候，只是count++，避免进入CAS操作。这就是**可重入自旋锁**。

测试：

public static void main(String[] args) {

SpinLock spinLock = new SpinLock();

spinLock.lock();

System.out.println("进入第一层锁");

spinLock.lock();

System.out.println("进入第二层锁");

spinLock.unlock();

spinLock.unlock();

System.out.println("结束");

}

# ReenTrantLock实现的原理

**ReenTrantLock**的实现是一种**自旋锁**，通过**循环调用CAS操作来实现加锁**。它的性能比较好也是因为**避免了使线程进入内核态的阻塞状态**。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

# ReenTrantLock可重入锁与synchronized

## 相同点

### 都是可重入锁

在java 中，synchronized和java.util.concurrent.locks.ReentrantLock都是可重入锁，两者都是同一个线程每进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

### 待定。

## 不同点

### 实现方式不同

**synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的**，有什么区别，说白了就类似于操作系统来控制实现和用户自己敲代码实现的区别。前者的实现是比较难见到的，后者有直接的源码可供阅读。

### 性能区别

在**Synchronized优化**以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，**官方甚至建议使用synchronized**，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。**都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。**

### 功能区别

**便利性**：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而**ReenTrantLock需要手工声明来加锁和释放锁**，为了避免忘记手工释放锁造成死锁，所以**最好在finally中声明释放锁**。

**锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized。**

**ReenTrantLock具有独有的能力，见（3）。**

ReentrantLock锁有时间锁等候（tryLock( )），可中断锁等候（lockInterruptibly( )），锁投票等，适合用于高度竞争锁和多个条件变量的地方如BlockingQueue。

## ReenTrantLock独有的能力：

什么情况下使用ReenTrantLock：

答案，如果你需要实现ReenTrantLock的三个独有功能时就用ReentrantLock。

### ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。

* ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。(**BlockingQueue中源码应用**)
* ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过**lock.lockInterruptibly()**来实现这个机制。