目录

[一、 synchronized 1](#_Toc19453082)

[1.1 synchronized的应用 1](#_Toc19453083)

[1.2对象的组成，与对象头详解 1](#_Toc19453084)

[1.3synchronized的实现 2](#_Toc19453085)

[1.4 各个锁概念及运行流程 3](#_Toc19453086)

[1.5 synchronized原理 5](#_Toc19453087)

[二、读写锁 ReadWriteLock 6](#_Toc19453088)

[1.2公平和非公平锁 12](#_Toc19453089)

# 一、 synchronized

Jdk1.6之后进行了优化，无锁——》偏向锁——》轻量级锁——》重量级锁

## 1.1 synchronized的应用

1、方法

静态方法（相当于加锁加在类对象上）、非静态方法（加锁加在当前this类对象上），此时如果类对象由spring容器管理则必须设置为单例

1. 代码块

对象锁、类锁

3、当作用的对象为静态时，则作用于整个类

4、跨方法加对象锁可以使用unsafe的moniterEnter与monitorExit来实现

5、synchronized从八大内存操作的Lock开始，volatile则从read开始

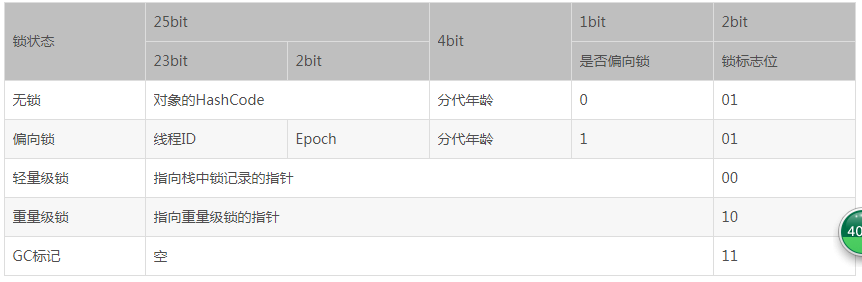
## 1.2对象的组成，与对象头详解

**1、对象组成**

Java对象由三部分组成，对象头（Header）、实例数据（Instance Data）、对齐填充字节（Padding）

**A、对象头**

对象头由Mark Word（32位系统32位，64位系统64位）、指向类的指针、数组长度（只有数组对象有）



**垃圾回收age最大为15，因为存储分代年龄的空间4位，最大值为15**

#### 2、指向类的的指针（class pointer）

这一部分用于存储对象的类型指针，该指针指向它的类元数据，JVM通过这个指针确定对象是哪个类的实例。该指针的位长度为JVM的一个字大小，即32位的JVM为32位，64位的JVM为64位。  
如果应用的对象过多，使用64位的指针将浪费大量内存，统计而言，64位的JVM将会比32位的JVM多耗费50%的内存。为了节约内存可以使用选项+UseCompressedOops开启指针压缩，其中，oop即ordinary object pointer普通对象指针。开启该选项后，下列指针将压缩至32位：

1. 每个Class的属性指针（即静态变量）
2. 每个对象的属性指针（即对象变量）
3. 普通对象数组的每个元素指针

当然，也不是所有的指针都会压缩，一些特殊类型的指针JVM不会优化，比如指向PermGen的Class对象指针(JDK8中指向元空间的Class对象指针)、本地变量、堆栈元素、入参、返回值和NULL指针等。

1. 数组长度

如果对象是一个数组，那么对象头还需要有额外的空间用于存储数组的长度，这部分数据的长度也随着JVM架构的不同而不同：32位的JVM上，长度为32位；64位JVM则为64位。64位JVM如果开启+UseCompressedOops选项，该区域长度也将由64位压缩至32位。

1. **实例数据**

真正存储有效信息，即程序代码中所定义的各种类型的字段内容，包括父类继承下来的和子类定义的。这部分存储顺序受虚拟机分配策略参数（FiledsAllocationStyle)和字段在Java源码中定义的顺序的影响。HotSpot虚拟机默认分配策略为longs/double、ints、shorts/chars、bytes/booleans、oops(Ordinary Object Pointers)，从分配策略可以看出，相同宽度的字段总是被分配到一起。在此前提条件下，父类中定义的变量会出在子类之前，若CompactFileds的参数值为true，那么子类中较**窄**的变量也可能会插到父类变量的空隙中。

1. **填充字节**

不是必然存在的，仅起着占位符作用。HotSpot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，即对象大小必须是8字节的整数倍，而对象头正好是8字节的整数倍，而对象实例数据部分没有对齐，需要通过对齐填充来补全。

## 1.3synchronized的实现

在Hot Spot中，monitor是由ObjectMonitor类来实现的

其中有两个队列 \_EntryList和 \_WaitSet，它们是用来保存ObjectMonitor对象列表， \_owner指向持有ObjectMonitor对象的线程。 当多个线程访问同步代码时，线程会进入\_EntryList区，当线程获取对象的monitor后(对于线程获得锁的优先级，还有待考究)进入 \_Owner区并且将 \_owner指向获得锁的线程(monitor对象被线程持有)， \_count++，其他线程则继续在 \_EntryList区等待。若线程调用wait方法，则该线程进入 \_WaitSet区等待被唤醒。线程执行完后释放monitor锁并且对ObjectMonitor中的值进行复位。 上面说到synchronized使用的锁都放在对象头里，大概指的就是Mark Word中指向互斥量的指针指向的monitor对象内存地址了。

1.4 各个锁概念及运行流程  
**JVM一般是这样使用锁和Mark Word的：**

1，当没有被当成锁时，这就是一个普通的对象，Mark Word记录对象的HashCode，锁标志位是01，是否偏向锁那一位是0。

2，当对象被当做同步锁并有一个线程A抢到了锁时，锁标志位还是01，但是否偏向锁那一位改成1，前23bit记录抢到锁的线程id，表示进入偏向锁状态。

3，当线程A再次试图来获得锁时，JVM发现同步锁对象的标志位是01，是否偏向锁是1，也就是偏向状态，Mark Word中记录的线程id就是线程A自己的id，表示线程A已经获得了这个偏向锁，可以执行同步锁的代码。

4，当线程B试图获得这个锁时，JVM发现同步锁处于偏向状态，但是Mark Word中的线程id记录的不是B，那么线程B会先用CAS操作试图获得锁，这里的获得锁操作是有可能成功的，因为线程A一般不会自动释放偏向锁。如果抢锁成功，就把Mark Word里的线程id改为线程B的id，代表线程B获得了这个偏向锁，可以执行同步锁代码。如果抢锁失败，则继续执行步骤5。

5，偏向锁状态抢锁失败，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。JVM会在当前线程的线程栈中开辟一块单独的空间，里面保存指向对象锁Mark Word的指针，同时在对象锁Mark Word中保存指向这片空间的指针。上述两个保存操作都是CAS操作，如果保存成功，代表线程抢到了同步锁，就把Mark Word中的锁标志位改成00，可以执行同步锁代码。如果保存失败，表示抢锁失败，竞争太激烈，继续执行步骤6。

6，轻量级锁抢锁失败，JVM会使用自旋锁，自旋锁不是一个锁状态，只是代表不断的重试，尝试抢锁。从JDK1.7开始，自旋锁默认启用，自旋次数由JVM决定。如果抢锁成功则执行同步锁代码，如果失败则继续执行步骤7。

7，自旋锁重试之后如果抢锁依然失败，同步锁会升级至重量级锁，锁标志位改为10。在这个状态下，未抢到锁的线程都会被阻塞。

**偏向锁的获取与释放**

1、访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认为可偏向状态。

2、如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。

3、如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。

4、如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint，无字节码被执行）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。（撤销偏向锁的时候会导致stop the word）

5、执行同步代码。

**1、轻量级锁的加锁过程**

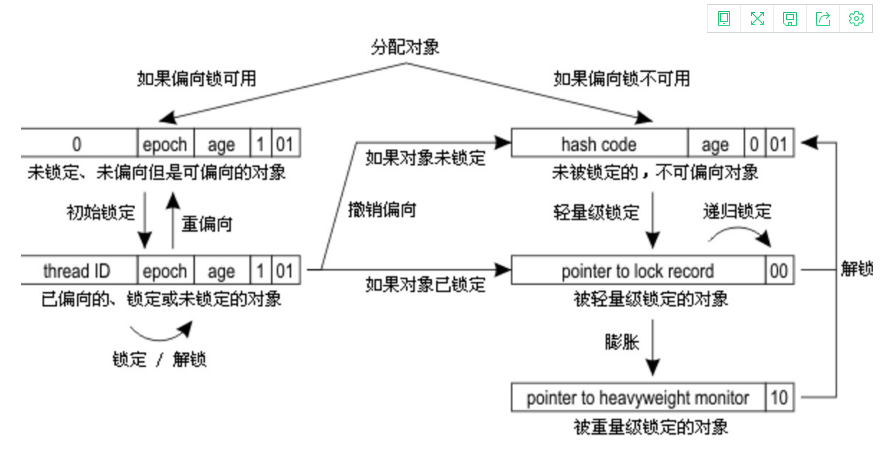
　　（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示。

　　（2）拷贝对象头中的Mark Word复制到锁记录中。

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

　　（4）如果这个更新动作成功了，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图2.2所示。

（5）如果这个更新操作失败了，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，锁标志的状态值变为“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，而采用循环去获取锁的过程。



## 1.5 synchronized原理

Synchronized底层依赖8大原子操作的lock与unlock指令实现

Synchronized是可重入锁，owern置空后其它线程即可进入

Synchronized升级为重量级锁时需要申请互斥量，需要调用内核，未获得锁的所有线程由用户态转为内核态，被挂起

**锁的粗化**：当有多个连续的加锁操作时，jvm将多个加锁操作合并为一个，增大锁的粒度

**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。

**逃逸分析：**

一、 同步省略。 如果一个对象被发现只能从一个线程被访问到， 那么对于这个对象的操作可以不考虑同步。  
二、 将堆分配转化为栈分配。 如果一个对象在子程序中被分配， 要使指向该对象的指针永远不会逃逸， 对象可能是栈分配的候选， 而不是堆分配。  
三、 分离对象或标量替换。 有的对象可能不需要作为一个连续的内存结构存在也可以被访问到， 那么对象的部分（或全部） 可以不存储在内存， 而是存储在CPU寄存器中。

**Java实例对象存储在哪？**

1、如果实例对象存储在堆区时：实例对象内存存在堆区，实例的引用存在栈上，实例的元数据class存在方法区或者元空间；

2、如果无逃逸行为则对象可能存放于栈空间；

3、对象部分或全部存放于寄存器中。

**标量替换**：

标量替换，scalar replacement。Java中的原始类型无法再分解，可以看作标量（scalar）；指向对象的引用也是标量；而对象本身则是聚合量（aggregate），可以包含任意个数的标量。如果把一个Java对象拆散，将其成员变量恢复为分散的变量，这就叫做标量替换。拆散后的变量便可以被单独分析与优化，可以各自分别在活动记录（栈帧或寄存器）上分配空间；原本的对象就无需整体分配空间了

标量即不可被进一步分解的量，而JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量

通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间。

我们通过JVM内存分配可以知道JAVA中的对象都是在堆上进行分配，当对象没有被引用的时候，需要依靠GC进行回收内存，如果对象数量较多的时候，会给GC带来较大压力，也间接影响了应用的性能。为了减少临时对象在堆内分配的数量，JVM通过逃逸分析确定该对象不会被外部访问。那就通过标量替换将该对象分解在栈上分配内存，这样该对象所占用的内存空间就可以随栈帧出栈而销毁，就减轻了垃圾回收的压力

# 二、读写锁 ReadWriteLock

读写锁实现读读不影响，读写、写写互斥，ReadWriteLock管理一组锁，一个只是读锁，一个是写锁，java并发库中ReetrantReadWriteLock实现ReadWriteLock并加入可重入特性，

**ReetrantReadWriteLock的特性：**

1. **获取锁的顺序**
2. 非公平模式（默认）

当以非公平初始化时，读锁和写锁的获取的顺序是不确定的。非公平锁主张竞争获取，可能会延缓一个或多个读或写线程，但是会比公平锁有更高的吞吐量。

（原理，公平模式会维护一个队列，所有线程会按照先后顺序依次执行，而非公平模式则没有执行顺序，后面的线程可能先获取锁，从而省去线程解除休眠状态的时间）

**B、公平模式**

当以公平模式初始化时，线程将会以队列的顺序获取锁。当当前线程释放锁后，等待时间最长的写锁线程就会被分配写锁；或者有一组读线程组等待时间比写线程长，那么这组读线程组将会被分配读锁。

1. **可重入**

可重入锁指一个线程获取某个锁后，还可以继续获取该锁，即允许线程多次获取同一个锁，synchronized内置锁就是可以重入的，如果A类有两个synchronized方法，1、2，那么1调用2是允许的

package test;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Test1 {

public static void main(String[] args) throws InterruptedException {

final ReentrantReadWriteLock lock = new ReentrantReadWriteLock ();

Thread t = new Thread(new Runnable() {

@Override

public void run() {

lock.writeLock().lock();

System.out.println("Thread real execute");

lock.writeLock().unlock();

}

});

lock.writeLock().lock();

lock.writeLock().lock();

t.start();

Thread.sleep(200);

System.out.println("realse one once");

lock.writeLock().unlock();

}

}

运行结果只有，realse one once，上面的代码会出现死锁，因为主线程2次获取了锁，但却只释放一次锁，导致线程永远不能获取锁，一个线程获取几次锁就必须释放几次锁，对于内置锁也适用，每一次进入和离开synchronization方法，就是一次完整的锁获取与释放

**3锁降级**

锁降级：从写锁变成读锁，锁升级：从读锁变为写锁

**ReetrantReadWriteLock**支持锁降级（由写锁直接转换为读锁），不支持升级（需要释放锁之后才能转换）

读写关系

package test;

import java.util.concurrent.ExecutorService;import java.util.concurrent.Executors;import java.util.concurrent.locks.ReentrantReadWriteLock;

/\*\*

\*

\* ReetrantReadWriteLock实现

\* @author itbird

\*

\*/public class ReadAndWriteLockTest {

public static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

public static void main(String[] args) {

//同时写

ExecutorService service = Executors.newCachedThreadPool();

service.execute(new Runnable() {

@Override

public void run() {

writeFile(Thread.currentThread());

}

});

service.execute(new Runnable() {

@Override

public void run() {

writeFile(Thread.currentThread());

}

});

}

// 读操作

public static void readFile(Thread thread) {

lock.readLock().lock();

boolean readLock = lock.isWriteLocked();

if (!readLock) {

System.out.println("当前为读锁！");

}

try {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(thread.getName() + ":正在进行读操作……");

}

System.out.println(thread.getName() + ":读操作完毕！");

} finally {

System.out.println("释放读锁！");

lock.readLock().unlock();

}

}

// 写操作

public static void writeFile(Thread thread) {

lock.writeLock().lock();

boolean writeLock = lock.isWriteLocked();

if (writeLock) {

System.out.println("当前为写锁！");

}

try {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(thread.getName() + ":正在进行写操作……");

}

System.out.println(thread.getName() + ":写操作完毕！");

} finally {

System.out.println("释放写锁！");

lock.writeLock().unlock();

}

}

}

\*\*\*\*\*\*\*boolean readLock = lock.isWriteLocked();判断当前锁类型

### 1.2公平和非公平锁

公平和非公平锁的队列都基于锁内部维护的一个双向链表，表结点Node的值就是每一个请求当前锁的线程。公平锁则在于每次都是依次从队首取值。

锁的实现方式是基于如下几点：

表结点Node和状态state的volatile关键字。

sum.misc.Unsafe.compareAndSet的原子操作(见附录)。

**非公平锁和公平锁的两处不同：**

非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

公平锁的实现

static final class FairSync extends Sync {

final void lock() {

acquire(1);

}

// AbstractQueuedSynchronizer.acquire(int arg)

public final void acquire(int arg) {

//tryAcquire是一个抽象方法，是公平与非公平的实现所在

if (!tryAcquire(arg) &&

//acquireQueued 在多次循环在尝试获取锁或者将线程阻塞，addWaiter将当前线程加入到等待队列中

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

selfInterrupt();//如果线程阻塞，调用Thread.CurrentThread.interrupt中断线程

}

protected final boolean tryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

// 1. 和非公平锁相比，这里多了一个判断：是否有线程在等待

if (!hasQueuedPredecessors() &&

compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0)

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

setState(nextc);

return true;

}

return false;

}

}

在注释1的位置，有个!hasQueuedPredecessors()条件，意思是说当前同步队列没有前驱节点（也就是没有线程在等待）时才会去compareAndSetState(0, acquires)使用CAS修改同步状态变量。所以就实现了公平锁，根据线程发出请求的顺序获取锁。

非公平锁

static final class NonfairSync extends Sync {

final void lock() {

// 2. 和公平锁相比，这里会直接先进行一次CAS，成功就返回了

1. compareAndSetState的实现主要是通过Unsafe类实现的。 \* 2.之所以命名为Unsafe，是因为这个类对于JVM来说是不安全的，我们平时也是使用不了这个类的。 \* 3.Unsafe类内封装了一些可以直接操作指定内存位置的接口，是不是感觉和C有点像了？ \* 4.Unsafe类封装了CAS操作，来达到乐观的锁的争抢的效果(protected final boolean compareAndSetState(int expect, int update) { // See below for intrinsics setup to support this return unsafe.compareAndSwapInt(this, stateOffset, expect, update); }compareAndSwapInt是一个native方法，看一下stateOffset
2. static { try { //这个方法很有意思，主要的意思是获取AbstractQueuedSynchronizer的state成员的偏移量 //通过这个偏移量来更新state成员，另外state是volatile的来保证可见性。 stateOffset = unsafe.objectFieldOffset (AbstractQueuedSynchronizer.class.getDeclaredField("state")); headOffset = unsafe.objectFieldOffset (AbstractQueuedSynchronizer.class.getDeclaredField("head")); tailOffset = unsafe.objectFieldOffset (AbstractQueuedSynchronizer.class.getDeclaredField("tail")); waitStatusOffset = unsafe.objectFieldOffset (Node.class.getDeclaredField("waitStatus")); nextOffset = unsafe.objectFieldOffset (Node.class.getDeclaredField("next")); } catch (Exception ex) { throw new Error(ex); } }

stateOffset 是AbstractQueuedSynchronizer内部定义的一个状态量，AbstractQueuedSynchronizer是线程的竞态条件，所以只要某一个线程CAS改变状态成功，同时在没有释放的情况下，其他线程必然失败（对于Unsafe类还不是很熟悉，后面还需要系统的学习）。  
对于竞争成功的线程会调用 setExclusiveOwnerThread方法：protected final void setExclusiveOwnerThread(Thread t) { exclusiveOwnerThread = t; }这个实现是比较简单的，只是获取当前线程的引用，令AbstractOwnableSynchronizer中的exclusiveOwnerThread引用到当前线程。竞争失败的线程，会调用acquire方法，这个方法也是ReentrantLock设计的精华之处：此处主要是处理没有获取到锁的线程 \* tryAcquire：重新进行一次锁获取和进行锁重入的处理。 \* addWaiter：将线程添加到等待队列中。 \* acquireQueued：自旋获取锁。 \* selfInterrupt：中断线程。 \* 三个条件的关系为and,如果 acquireQueued返回true，那么线程被中断selfInterrupt会中断线程

public final void acquire(int arg) { if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt(); }

final boolean nonfairTryAcquire(int acquires) { final Thread current = Thread.currentThread(); int c = getState(); if (c == 0) { if (compareAndSetState(0, acquires)) { setExclusiveOwnerThread(current); return true; } } else if (current == getExclusiveOwnerThread()) { int nextc = c + acquires; if (nextc < 0) // overflow throw new Error("Maximum lock count exceeded"); setState(nextc); return true; } return false; }

nonfairTryAcquire方法主要是做重入锁的实现，synchronized本身支持锁的重入，而ReentrantLock则是通过此处实现。在锁状态为0时，重新尝试获取锁。如果已经被占用，那么做一次是否当前线程为占用锁的线程的判断，如果是一样的那么进行计数，当然在锁的relase过程中会进行递减，保证锁的正常释放。  
如果没有重新获取到锁或者锁的占用线程和当前线程是一个线程，方法返回false。那么把线程添加到等待队列中，调用addWaiter：

private Node addWaiter(Node mode) { Node node = new Node(Thread.currentThread(), mode); // Try the fast path of enq; backup to full enq on failure Node pred = tail; if (pred != null) { node.prev = pred; if (compareAndSetTail(pred, node)) { pred.next = node; return node; } } enq(node); return node; }

private Node addWaiter(Node mode) { Node node = new Node(Thread.currentThread(), mode); // Try the fast path of enq; backup to full enq on failure Node pred = tail; if (pred != null) { node.prev = pred; if (compareAndSetTail(pred, node)) { pred.next = node; return node; } } enq(node); return node; }

private Node enq(final Node node) { for (;;) { Node t = tail; if (t == null) { // Must initialize if (compareAndSetHead(new Node())) tail = head; } else { node.prev = t; if (compareAndSetTail(t, node)) { t.next = node; return t; } } } }

这里主要是用当前线程构建一个Node的等待队列双向链表，这里addWaiter中和enq中的部分逻辑是重复的，个人感觉可能是如果能一次成功就避免了enq中的死循环。因为tail节点是volatile的同时node也是不会发生竞争的所以node.prev = pred;是安全的。但是tail的next是不断竞争的，所以利用compareAndSetTail保证操作的串行化。接下来调用acquireQueued方法：

final boolean acquireQueued(final Node node, int arg) { boolean failed = true; try { boolean interrupted = false; for (;;) { final Node p = node.predecessor(); if (p == head && tryAcquire(arg)) { setHead(node); p.next = null; // help GC failed = false; return interrupted; } if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt()) interrupted = true; } } finally { if (failed) cancelAcquire(node); } }

此处是做Node节点线程的自旋过程，自旋过程主要检查当前节点是不是head节点的next节点，如果是，则尝试获取锁，如果获取成功，那么释放当前节点，同时返回。至此一个非公平锁的锁获取过程结束。  
如果这里一直不断的循环检查，其实是很耗费性能的，JDK的实现肯定不会这么“弱智”，所以有了shouldParkAfterFailedAcquire和parkAndCheckInterrupt，这两个方法就实现了线程的等待从而避免无限的轮询：

首先，检查一下当前Node的前置节点pred是否是SIGNAL，如果是SIGNAL，那么证明前置Node的线程已经Park了，如果waitStatus>0,那么当前节点已经Concel或者中断。那么不断调整当前节点的前置节点，将已经Concel的和已经中断的线程移除队列。如果waitStatus<0,那么设置waitStatus为SIGNAL，因为调用shouldParkAfterFailedAcquire的方法为死循环调用，所以终将返回true。接下来看parkAndCheckInterrupt方法，当shouldParkAfterFailedAcquire返回True的时候执行parkAndCheckInterrupt方法：

private final boolean parkAndCheckInterrupt() { LockSupport.park(this); return Thread.interrupted(); }

此方法比较简单，其实就是使当前的线程park，即暂停了线程的轮询。当Unlock时会做后续节点的Unpark唤醒线程继续争抢锁。  
接下来看一下锁的释放过程，锁释放主要是通过unlock方法实现：

public void unlock() { sync.release(1); }

public final boolean release(int arg) { if (tryRelease(arg)) { Node h = head; if (h != null && h.waitStatus != 0) unparkSuccessor(h); return true; } return false; }

protected final boolean tryRelease(int releases) { int c = getState() - releases; if (Thread.currentThread() != getExclusiveOwnerThread()) throw new IllegalMonitorStateException(); boolean free = false; if (c == 0) { free = true; setExclusiveOwnerThread(null); } setState(c); return free; }

tryRelease方法主要是做了一个释放锁的过程，将同步状态state -1，直到减到0为止，这主要是兼容重入锁设计的，同时setExclusiveOwnerThread(null)清除当前占用的线程。这些head节点后的线程和新进的线程就可以开始争抢。这里需要注意的是对于同步队列中的线程来说在setState(c)，且c为0的时候，同步队列中的线程是没有竞争锁的，因为线程被park了还没有唤醒。但是此时对于新进入的线程是有机会获取到锁的。  
下面代码是进行线程的唤醒：

Node h = head; if (h != null && h.waitStatus != 0) unparkSuccessor(h); return true;

因为在setState(c)释放了锁之后，是没有线程竞争的，所以head是当前的head节点，先检查当前的Node是否合法，如果合法则unpark it。开始锁的获取。就回到了上面的for循环执行获取锁逻辑：)

if (compareAndSetState(0, 1))

setExclusiveOwnerThread(Thread.currentThread());

else

acquire(1);

}

// AbstractQueuedSynchronizer.acquire(int arg)

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

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

selfInterrupt();

}

protected final boolean tryAcquire(int acquires) {

return nonfairTryAcquire(acquires);

}

}/\*\*

\* Performs non-fair tryLock. tryAcquire is implemented in

\* subclasses, but both need nonfair try for trylock method.

\*/final boolean nonfairTryAcquire(int acquires) {

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

//3.这里也是直接CAS，没有判断前面是否还有节点。

if (compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

else if (current == getExclusiveOwnerThread()) {

int nextc = c + acquires;

if (nextc < 0) // overflow

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

setState(nextc);

return true;

}

return false;

}

非公平锁的实现在刚进入lock方法时会直接使用一次CAS去尝试获取锁，不成功才会到acquire方法中，如注释2。而在nonfairTryAcquire方法中并没有判断是否有前驱节点在等待，直接CAS尝试获取锁，如注释3。由此实现了非公平锁。