ReentrantLock支持两种获取锁的方式，一种是公平模型，一种是非公平模型。

公平锁模型：一个线程在获取了锁之后，再次去获取了同一个锁，这时候仅仅是把状态值进行累加。如果线程A释放了一次锁，仅仅是把状态值减了，只有线程A把此锁全部释放了，状态值减到0了，其他线程才有机会获取锁。当A把锁完全释放后，state恢复为0，然后会通知队列唤醒B线程节点，使B可以再次竞争锁。

非公平锁模型：当线程A执行完之后，要唤醒线程B是需要时间的，而且线程B醒来后还要再次竞争锁，

所以如果在切换过程当中，来了一个线程C，那么线程C是有可能获取到锁的，如果C获取到了锁，B就只能继续乖乖休眠了

CAS，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。ReentrantLock是基于AQS实现的，这在下面会讲到，AQS的基础又是CAS。

CAS有三个操作数：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。

1、AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程1和线程2各自持有一份value的副本，值为3

2、线程1运行到第三行获取到当前的value为3，线程切换

3、线程2开始运行，获取到value为3，利用CAS对比内存中的值也为3，比较成功，修改内存，此时内存中的value改变比方说是4，线程切换

4、线程1恢复运行，利用CAS比较发现自己的value为3，内存中的value为4，得到一个重要的结论-->此时value正在被另外一个线程修改，所以我不能去修改它

5、线程1的compareAndSet失败，循环判断，因为value是volatile修饰的，所以它具备可见性的特性，线程2对于value的改变能被线程1看到，只要线程1发现当前获取的value是4，内存中的value也是4，说明线程2对于value的修改已经完毕并且线程1可以尝试去修改它

6、最后说一点，比如说此时线程3也准备修改value了，没关系，因为比较-交换是一个原子操作不可被打断，线程3修改了value，线程1进行compareAndSet的时候必然返回的false，这样线程1会继续循环去获取最新的value并进行compareAndSet，直至获取的value和内存中的value一致为止

整个过程中，利用CAS机制保证了对于value的修改的线程安全性。

CAS的缺点

CAS看起来很美，但这种操作显然无法涵盖并发下的所有场景，并且CAS从语义上来说也不是完美的，存在这样一个逻辑漏洞：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个漏洞称为CAS操作的"ABA"问题。java.util.concurrent包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。不过目前来说这个类比较"鸡肋"，大部分情况下ABA问题并不会影响程序并发的正确性，如果需要解决ABA问题，使用传统的互斥同步可能回避原子类更加高效。

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。

这个类尽管里面的方法都是public的，但是并没有办法使用它们，JDK API文档也没有提供任何关于这个类的方法的解释。总而言之，对于Unsafe类的使用都是受限制的，只有授信的代码才能获得该类的实例，当然JDK库里面的类是可以随意使用的。

ReentrantLock实现的前提就是AbstractQueuedSynchronizer，简称AQS，是java.util.concurrent的核心，CountDownLatch、FutureTask、Semaphore、ReentrantLock等都有一个内部类是这个抽象类的子类。

由于AQS是基于FIFO队列的实现，因此必然存在一个个节点，Node就是一个节点。

synchronized和lock的区别：

|  |  |  |
| --- | --- | --- |
| 类别 | synchronized | lock |
| 存在层次 | java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁  2、线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 假设A线程获得锁，B线程等待，  如果A线程阻塞，B线程会一直等待 | 分情况而定，lock有多个锁获取的方法，可以尝试获得锁，  线程可以不用功一直等待 |
| 锁状态 | 无法判断 | 可以判断 |
| 锁类型 | 可以重入，不可以中断，非公平 | 可重入 可以判断 可公平 |
| 性能 | 少量同步 | 大量同步 |

jdk1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

　　锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。（注意！锁只可以升级不可以降级，这种策略是为了提高获得锁和释放锁的效率）。

所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放。怎么等待呢？执行一段无意义的循环即可（自选）。自旋次数（或自旋等待时间）有限制，一单超过仍未获取到锁，则被挂起。

自旋锁在jdk 1.4.2中引入，默认关闭，但是可以使用-XX:+UseSpinning开启，在jdk1.6中默认开启。同时默认的次数为10次，可以通过参数-XX:PreBlockSpin来调整；如果通过参数-XX:preBlockSpin来调整自旋锁的自旋次数，会带来诸多不便。假如我将参数调整为10，但是系统很多线程都是等你刚刚退出的时候就释放了锁（假如你多自旋一两次就可以获取锁），你是不是很尴尬，于是jdk1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明。

所谓自适应就意味着自旋的次数不再是固定的，他是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定的。它怎么做的呢？线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么这次自旋也很有可能会再次成功，那么他就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋成功的，那么在以后要获取这个锁的时候自旋的次数会减少甚至省略掉自旋的过程，以免浪费处理器资源。

锁消除的依据是逃逸分析的数据支持。

　锁粗化：就是将多个连续的加锁、解锁的操作连接在一起，扩展成一个范围更大的锁。

引入偏向锁主要目的是：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。

引入轻量级锁的主要目的是在多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁

重量级锁通过对象内部的监视器（monitor）实现的，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。

[https://www.cnblogs.com/xrq730/p/4979021.html#top](https://www.cnblogs.com/xrq730/p/4979021.html" \l "top)很多源码，看不太懂。

#### 功能区别

基本意义：Synchronized是Java语言的关键字，因此Synchronized的锁是原生语法层面的互斥，需要JVM来实现。具体是通过对象内部的一个叫做监视器锁（monitor）来实现的。ReentrantLock，字面意思可重入锁，它是JDK1.5之后提供的API层面的互斥锁，锁的功能主要由2个方法完成，即lock()和unlock()。

易用性：Synchronized的使用比较方便简洁，由编译器去保证锁的加锁和释放，而ReentrantLock需要手动写代码来加锁和释放锁。  
注意：为避免忘记手工释放锁而造成了死锁，最好在finally中声明释放锁。

灵活度：ReentrantLock要优于Synchronized，可以灵活控制在哪个位置加锁和解锁。

http://www.sohu.com/a/333014778\_120043609