Lock的原理

依赖于AQS

Reentrantlock把所有lock接口都委派到一个sync类上，该类继承了AQS

Sync有两个子类 分别支持了公平锁与非公平锁

简单的说 AQS把所有的请求线程构成一个CLH队列 当一个线程执行完毕时 激活自己的后继节点 正在执行的行程不在队列中 等待执行的线程全部处于阻塞状态

与synchronized相同的是，这也是一个虚拟队列，不存在队列实例，仅存在节点之间的前后关系

原生的CLH队列是用于自旋锁，但Doug Lea把其改造为阻塞锁。

当有线程竞争锁时，该线程会首先尝试获得锁，这对于那些已经在队列中排队的线程来说显得不公平，这也是非公平锁的由来，与synchronized实现类似，这样会极大提高吞吐量。

如果已经存在Running线程，则新的竞争线程会被追加到队尾，具体是采用基于CAS的Lock-Free算法，因为线程并发对Tail调用CAS可能会导致其他线程CAS失败，解决办法是循环CAS直至成功

Synchronized锁的3种使用形式（使用场景）：

Synchronized修饰普通同步方法：锁对象当前实例对象；

Synchronized修饰静态同步方法：锁对象是当前的类Class对象；

Synchronized修饰同步代码块：锁对象是Synchronized后面括号里配置的对象，这个对象可以是某个对象（xlock），也可以是某个类（Xlock.class）；

锁升级：

Markworld



重量级锁对应的锁标志位是10，存储了指向重量级监视器锁的指针，在Hotspot中，对象的监视器（monitor）锁对象由ObjectMonitor对象实现

1）偏向锁：

为什么要引入偏向锁？

大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

偏向锁的升级

当线程1访问代码块并获取锁对象时，会在java对象头和栈帧中记录偏向的锁的threadID，因为偏向锁不会主动释放锁，因此以后线程1再次获取锁的时候，需要比较当前线程的threadID和Java对象头中的threadID是否一致，如果一致（还是线程1获取锁对象），则无需使用CAS来加锁、解锁；如果不一致（其他线程，如线程2要竞争锁对象，而偏向锁不会主动释放因此还是存储的线程1的threadID），那么需要查看Java对象头中记录的线程1是否存活，如果没有存活，那么锁对象被重置为无锁状态，其它线程（线程2）可以竞争将其设置为偏向锁；如果存活，那么立刻查找该线程（线程1）的栈帧信息，如果还是需要继续持有这个锁对象，那么暂停当前线程1，撤销偏向锁，升级为轻量级锁，如果线程1 不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

偏向锁的取消：

偏向锁是默认开启的，而且开始时间一般是比应用程序启动慢几秒，如果不想有这个延迟，那么可以使用-XX:BiasedLockingStartUpDelay=0；

如果不想要偏向锁，那么可以通过-XX:-UseBiasedLocking = false来设置；

（2）轻量级锁

为什么要引入轻量级锁？

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。因为阻塞线程需要CPU从用户态转到内核态，代价较大，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

轻量级锁什么时候升级为重量级锁？

线程1获取轻量级锁时会先把锁对象的对象头MarkWord复制一份到线程1的栈帧中创建的用于存储锁记录的空间（称为DisplacedMarkWord），然后使用CAS把对象头中的内容替换为线程1存储的锁记录（DisplacedMarkWord）的地址；

如果在线程1复制对象头的同时（在线程1CAS之前），线程2也准备获取锁，复制了对象头到线程2的锁记录空间中，但是在线程2CAS的时候，发现线程1已经把对象头换了，线程2的CAS失败，那么线程2就尝试使用自旋锁来等待线程1释放锁。

但是如果自旋的时间太长也不行，因为自旋是要消耗CPU的，因此自旋的次数是有限制的，比如10次或者100次，如果自旋次数到了线程1还没有释放锁，或者线程1还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。

\*注意：为了避免无用的自旋，轻量级锁一旦膨胀为重量级锁就不会再降级为轻量级锁了；偏向锁升级为轻量级锁也不能再降级为偏向锁。一句话就是锁可以升级不可以降级，但是偏向锁状态可以被重置为无锁状态。