非常好的文章：http://blog.csdn.net/zjkc050818/article/details/78493427

https://www.cnblogs.com/dsj2016/p/5535729.html

1. [java多线程synchronized底层实现](http://www.cnblogs.com/dsj2016/p/5535729.html)

一直想把这个特别重要的关键词的底层实现搞明白。（当然现在也没有完全明白，如果有错误以后修改这篇文章）

首先，这个关键词synchronize可以说是个语法糖，它的具体用法网上很多博客都讲的比较明了了。

简而言之就是对一个对象“加锁”。首先，找个地方的对象不一定是堆里面的类的实例对象，也有可能是方法区的类对象。其次，这个关键词修饰的代码块的加锁过程有两个，进入的时候尝试获得锁（java字节码 monitorenter），退出时释放锁（java字节码monitorexit）。这两个操作的再下一层是基于mutex lock的lock()和unluck()。

这两个函数的具体实现由操作系统提供。lock()操作是“获得锁”“上锁”“进入临界区”，等等，不同的地方描述不一致。它的具体过程是：查看一个信号量（由这个锁持有），看当前能否获得锁，如果能直接获得，并且修改这个信号量的值（比如把1改成0）。如果不能，就把这个索取锁的线程自己加入一个队列，这队列专门放“困”在这个对象（这个锁）上的线程，接着阻塞这个线程自己。unlock()操作是“释放锁”“解锁”“离开临界区”。他可以直接修改信号量的值。同时他看是否有进程“困”在这个对象（锁）上。如果有，唤醒它放入就绪队列。（信号量的具体实现各不相同，记录型信号量可以更方便理解这个过程）。当然这些基础操作也是原子性的。

这个地方还有一个很重要的点，锁，线程，对象这3个东西到底怎么联系到一起的。而要讲明白这个，又不得不讲一下锁不一定是重量级的由操作系统提供的“互斥锁”。还有一种锁：轻量级锁。这种锁是一种运行时优化，如果用synchronize修饰的代码块没有发生并发行为就可以直接用这种“锁”。

一开始，要明白Mark Word。这是每一个对象的对象头中有的一个32bit或者64bit（由JVM确定）的一个区域（叫Mark Word，对象头还有一个和它一样大的区域保存了一个指针指向方法区的类对象）。保存了对象在运行过程中的一些数据，比如哈希码，GC分代年龄，上锁标志位等等。它的存在是必要的，因为确实有些运行时信息要通过这种形式保留。一个线程根据java字节码找到上锁的对象，查看上锁标志位，看看是否已经被别的线程获得锁，如果没有，就在这个线程的栈帧中建立一个锁记录（Lock Record，有的地方叫Monitor Record），保存这个Mark Word的一份拷贝，接着用一个Owner指针指向这个对象。对象则直接把Mark Word改成一个指针指向这个锁记录。（记得前面提到Mark Word的大小是刚好和当前操作系统指针的大小一样，所以可以直接改而不需要补位等操作）

但是这个地方一开始我很不理解，为什么要这么绕呢？既然锁在对象上面，为什么不直接在对象头保留一个空间，记录或者这个“锁”的线程呢，比如用线程的ID或者内部标识符。每次线程进入临界区，访问这个对象的时候直接去对象头看这个值是不是自己这个线程，如果不是就阻塞自己。

原因是：对象头是很珍贵的，因为每一个对象都有，它虽然有必要但是它的内容又确实不是对象本身真正的内容。也就是说要想尽一切办法缩小它的大小否则效率很低（试想你开辟的空间有一大半都储存了一些杂七杂八的信息）。相比这个对象头，运行时的程序栈可谓是非常广阔的空间。一个珍惜这点空间一个无所谓这点空间。这样就刚好通过“复用”来实现储存空间的优化。把本来需要额外增加的空间直接用Mark Word储存而它本来的值丢给线程的栈。这样就一举两得，首先储存了持有这个对象的锁的线程，同时也没有弄丢Mark Word（反正我用了指针指向这个线程也不怕找不到）。而线程则再用一个指针Owner指向这个对象。很完美。

这个是轻量级锁的做法，如果不是轻量级呢？其实JVM的优化策略保证了一开始都把他当做轻量级来处理（JVM的优化策略有自旋锁，锁消除，锁粗化，轻量级锁，偏向锁等等）。这个地方也要解释2点，第一，好处都有啥，第二，为什么能这么做。第三，为什么要这么做。

首先，如果直接用操作系统提供的Mutex Lock互斥锁的话，会使用操作系统调用，从用户态转为核心态，开销很大。用这种方式（轻量级锁）则只是一个CAS操作（要保证其原子性）的花费。第二，马上下面讲到如果轻量级锁没用了（也就是发生了竞争别的线程试图拿这个锁），它可以直接“膨胀”成一个重量级锁（Mutex Lock）。第三，现实情况是很多加了synchronize修饰的代码其实在实际运行过程中并没有发生竞争的情况，这么做在运行时直接减少了很多开销。

然后，什么情况下会从一个轻量级膨胀成一个重量级的Mutex Lock呢？其实jvm这部分的优化是这样的，一开始先“认为”这次加锁和大多数情况一样并没有发生竞争，于是先“机智”地用轻量级，这个时候如果发生竞争，也就是有别的线程尝试获得锁，就“膨胀”为一个重量级。再具体一点就是别的线程调用Lock()，发现当前这个对象的对象头的标志位是“加了轻量级锁的”。它再去看Owner，如果是自己就是一个“重入”。如果不是就说明发生了竞争，接着进行“膨胀”。这是第一种可能，也就是一个线程先在跑，后一个加入发生膨胀。其实还有第二种可能，发生在前一个线程刚放锁的时候，这个时候所有线程都认为没锁，同时通过CAS竞争，有一个会成功，其他的会失败，于是也进行膨胀。

接着来讲膨胀的过程。第一，改变对象锁标志的状态值，把Mark Word中保存的指针指向Mutex Lock（当然Mark Word的内容还是不能丢）我看的是深入理解Java虚拟机这本书，再加上一些网络上的博客。都没把这个部分讲明白，这个地方我“猜”一下：调用操作系统互斥锁，生成一个互斥锁并且把Mark Word的值指向它。同时让这个互斥锁或者别的什么数据结构和方式保存Mark Word原本的运行时信息。这个地方的Mutex Lock可能就是真真正正的“重量级”锁了，它的具体实现我估计和记录型信号量的PV原语操作差不多，同时还要保留一些标志位储存Owner，重入个数，等待队列的元素个数，等待队列指针，如果当前释放锁是否有线程需要唤醒等信息。

以上内容能够保证正确的是Mark Word指向一个Mutex Lock。这之后的过程就是再有线程调用Lock()尝试访问临界区，发先对象头指向一个锁，再进入锁发现已经上了锁并且自己不是Owner，于是就阻塞自己。出临界区的时候去唤醒别的阻塞线程。这都没啥说的了。