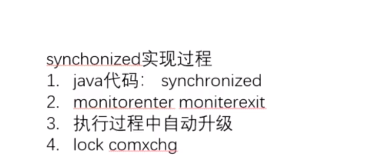
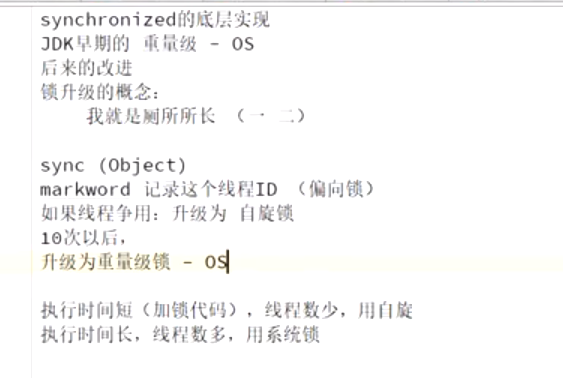
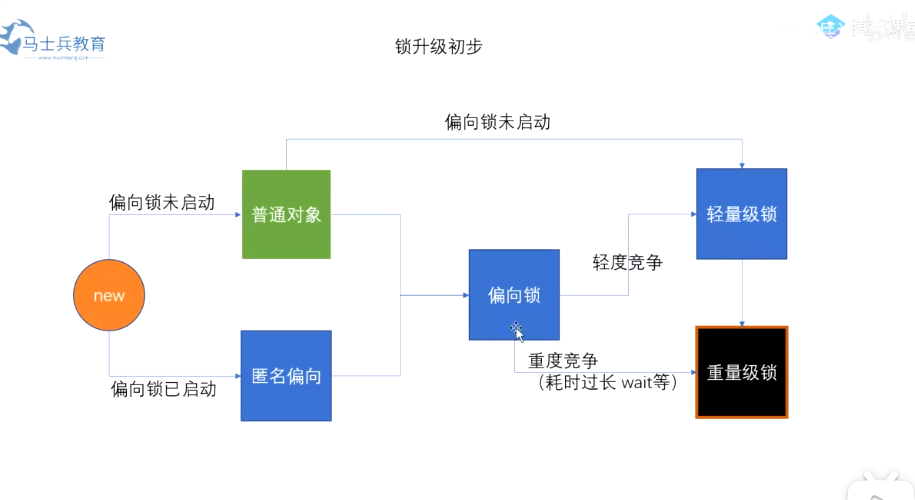


1、synchronize底层原理 --->lock cmpxchg







Markeord：记录锁状态，线程ID

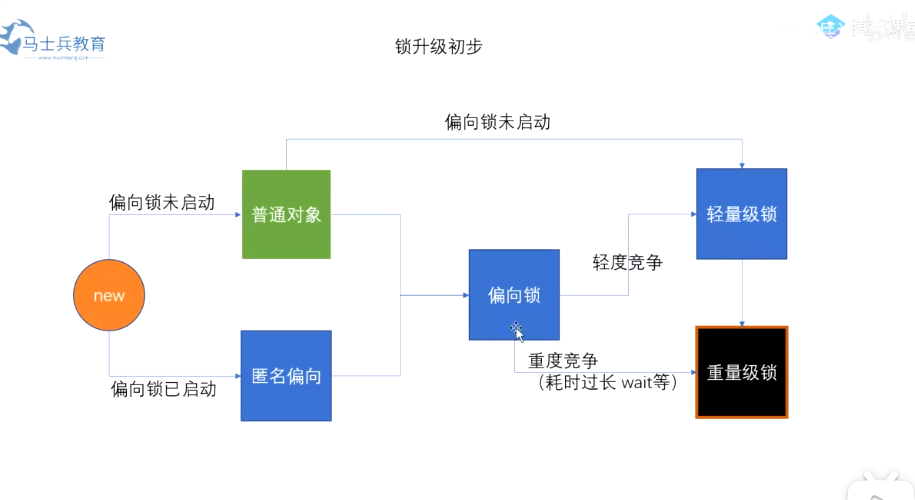
1. 创建对象：MarkWord记录为无锁态
2. 第一个线程访问这个对象：Markword记录为偏向锁，将线程ID记录下来，认为这个对象线程独有，偏向于一个线程使用，下次进来如果还是这个线程就直接使用，效率很高。在用户态，效率高。
3. 出现多个线程竞争资源：新进来线程，发现与偏向锁记录的线程ID不同，则将偏向锁升级为自旋锁。一个线程拿到锁了，其他线程通过while循环判断（CAS）是否可以拿到锁，消耗CPU（得一直CAS）。在用户态，效率高，占用CPU。
4. 若自旋超过10次：若线程while（CAS操作）10次或者等待的线程数大于CPU核数，将自旋锁升级为重量级锁，即向OS申请锁，进入等待队列。不消耗CPU（线程再队列中等待），但是耗费资源（用户态<->内核态）。在内核态，效率低，不占用CPU。



自旋锁：执行时间短（自旋快），线程数少（CPU资源受限）

OS锁：执行时间长，线程数多

1. 偏向锁





1. 为什么会有偏向锁启动延迟时间？

若明知道资源会有很多个资源竞争，加个偏向锁就没有意义，把偏向锁关了，会提高效率。

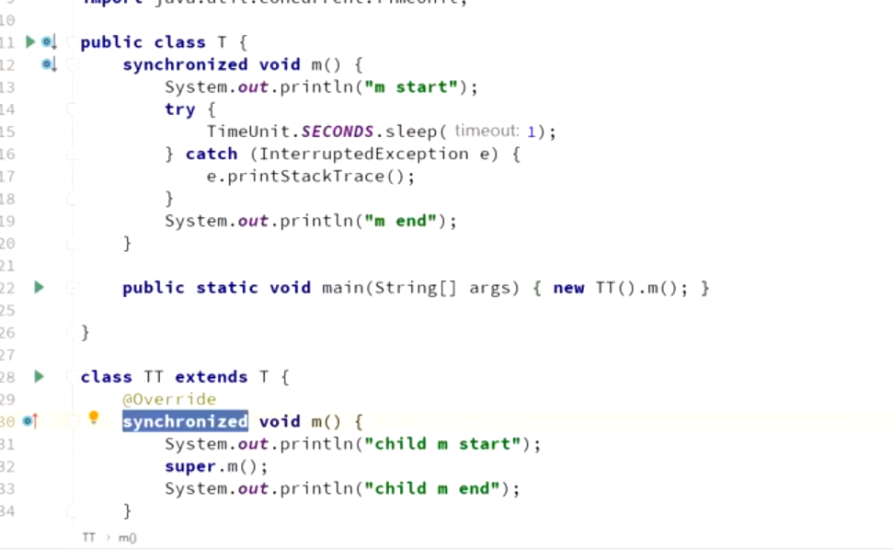
1. 匿名偏向

启动偏向锁，但是还不属于任何一个线程

1. 轻量级锁

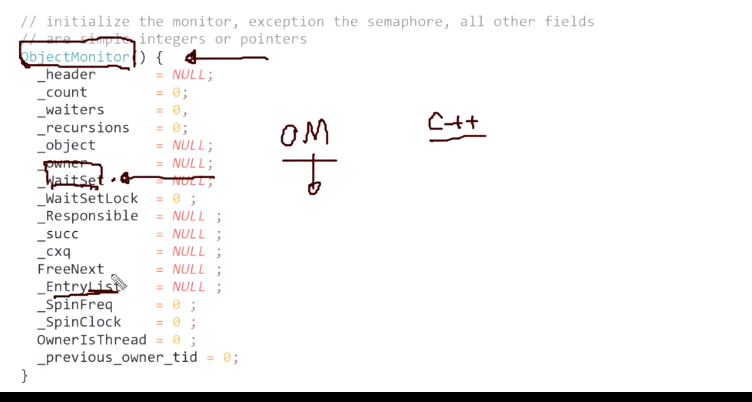
支持synchrnoize可重入性：同一个线程，允许多次申请同一个锁；每个线程会生成有个LockRecord，只要把它替换为轻量级锁的62位即可。

子父类的synchronize，锁的都是this对象，是同一把锁



1. 重量级锁

底层：objectMonitor

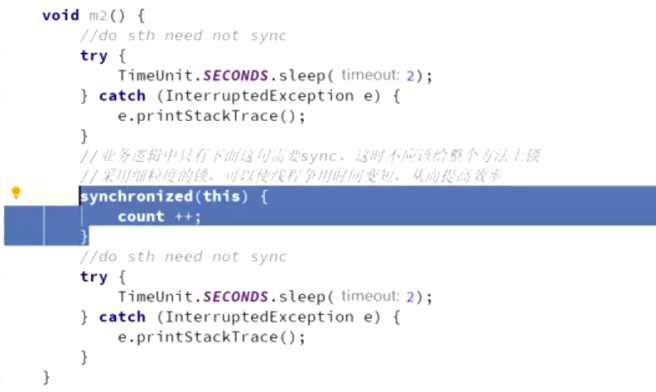


1. 异常会释放锁



1. Synchronize不能锁String、Integer、Long
2. Synchronize优化
3. 锁细化

业务逻辑应该只在需要加锁的地方加锁，不应该整个方法加锁



1. 锁粗化

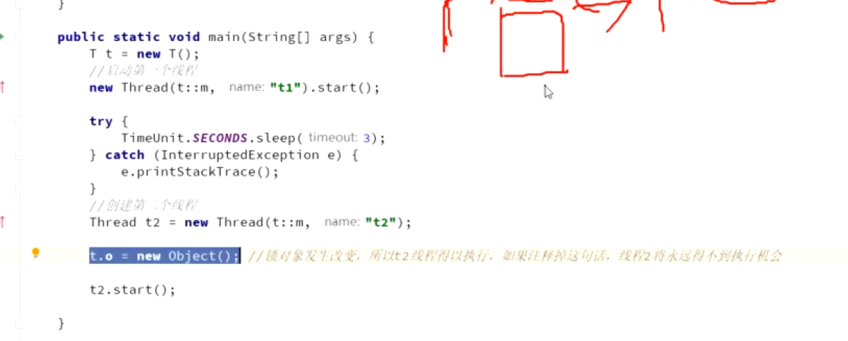
多次执行相同加锁代码，直接加个大锁



1. Synchronize锁对象问题

当对象发生变化时，会引发问题





解决：将o声明为final