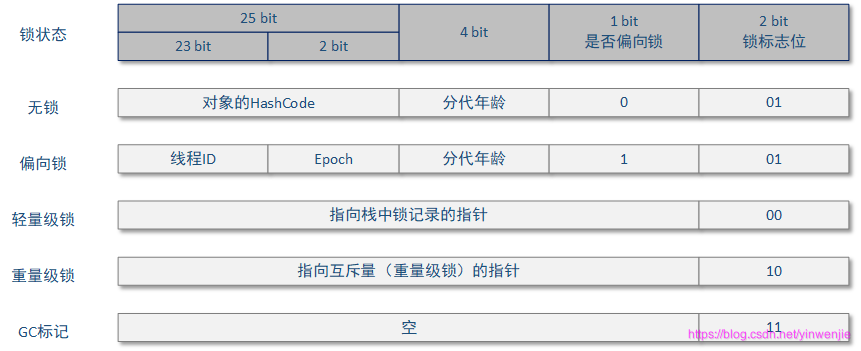
1. 什么是线程安全？
2. Synchronized的实现原理？
3. ReentrantLock的实现原理？
4. AQS的实现。 state: volatile int, CAS， CLH
5. Synchronized竞争线程在等待的时候是用户态还是核心态？(锁升级机制， 自旋锁、轻量锁、重量锁) ， CLH等待线程是用户态还是核心态？
6. ConcurrentHashMap为什么是线程安全的？
7. 线程池有哪些参数？ Coresize, maxsize, blokcingqueue, keepalivetime, rejectpolicy
8. 如果让你来设计一个线程池你怎么设计？ （其实就是讲一下线程池的设计和核心流程， 源码类ThreadPoolExecutor， 推荐文章：<https://tech.meituan.com/2020/04/02/java-pooling-pratice-in-meituan.html>）
9. ReentrantLock的公平模式和非公平模式是如何实现的？
10. 锁优化
    1. 自旋锁：当物理机上有一个以上的处理器时，让后请求锁的线程“**稍等一下**”， 但**不放弃处理器的时间**， 看看持有锁的线程是否很快就会释放锁。| 原因： 1. 互斥同步对**性能最大的影响**是**阻塞的实现**， 挂起线程和恢复线程的操作都需要转入内核态中完成 2. 许多应用上，共享数据的**锁定状态只会持续很短**的一段时间，为了这段时间区挂起和恢复线程并不值得。
    2. 自适应自旋：**自旋的时间不再固定**， 更加“聪明”。 基于同一个锁上的自旋时间。 如果之前很快获得过锁， 认为这次也很有可能成功， 进而它将允许更长的自旋等待时间；如果对于某个锁， 自旋很少成功获得过， 那么将可能省略掉自旋过程，以避免浪费处理器资源。 （基于统计预测）
    3. 锁消除：JVM编译优化， 对于不可能出现竞争的加锁情况予以消除。
    4. 锁粗化：JVM编译优化，将加锁的范围自动扩大。
    5. 轻量级锁：使用CAS尝试将monitor中的锁记录对象替换为自己。
    6. 偏向锁：有一个线程获取到锁且还没有其他线程来进行竞争。

**请参考MarkWord在不同锁状态下的结构**

**可参考**[**https://blog.csdn.net/yinwenjie/article/details/84922958**](https://blog.csdn.net/yinwenjie/article/details/84922958)**，如下图**



**锁的升级机制**

Thread1 获取到锁， ->偏向锁

Thread2 尝试获取锁， 自旋 ->自旋失败->检测到已经monitor已经有锁记录 ->升级为轻量锁->thread 2 使用CAS多次尝试将monitor中的锁记录对象替换为自己->多次失败后认为锁的竞争激励->升级为重量锁。

重量锁->**object monitor**的**entry set** ->被调度->object owner设为该线程并开始执行同步代码块->执行完毕，退出同步代码块，释放锁 or-> 调用monitor的wait()或其重载方法使该线程进入**wait set. 该线程必须拥有操作权即在获取到锁之后才能执行锁的wait()方法** -> 另外获取到锁对象的线程调用notify()或notifyAll()可以将wait set中的一个或所有对象重新放入entry set重新进行竞争。

Object monitor的结构定义请参考：<https://zhuanlan.zhihu.com/p/75880892>， ctrl+f + ObjectMonitor()

1. Wait(),sleep(), notify(),notifyAll():