一口气带你弄懂Synchronized锁升级优化的全流程

1.分享概要

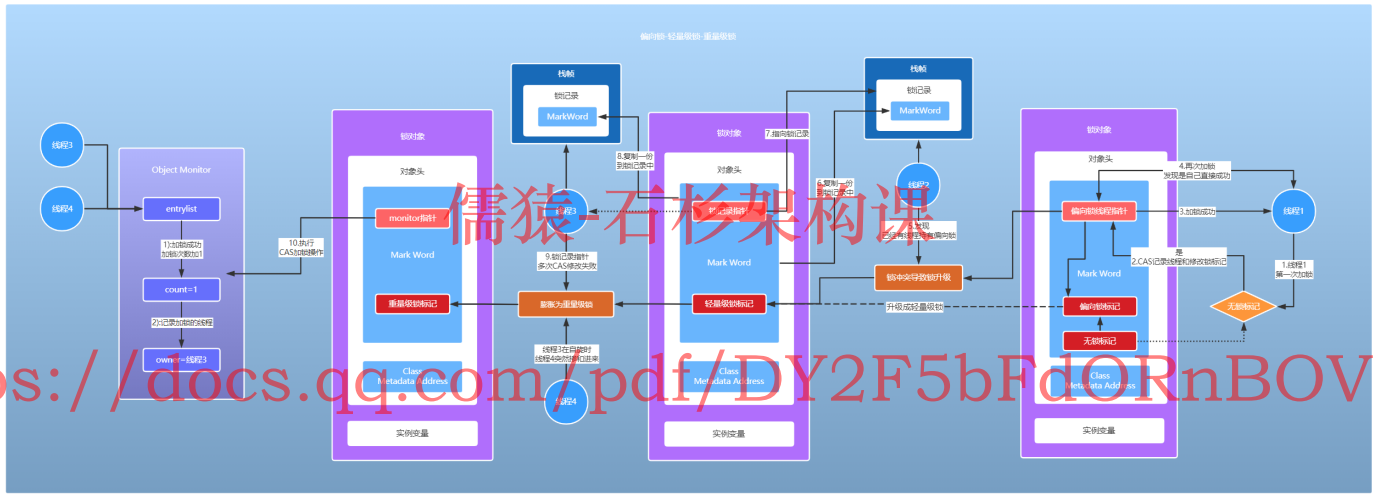
本次分享儒猿–石杉架构课**《精通基础开发技术之JDK并发包源码剖析以及真实生产案例实战》**中，Synchronized锁的各种优化：包括偏向锁、轻量级锁、重量级锁，以及它们之间是如何获取和释放锁、锁之间是如何进行升级和膨胀的。

本次分享会会淡化各种晦涩难懂的概念和参数细节，着重分享各种锁的引入背景、优化思想以及锁升级膨胀的流转过程，并且以流程图的方式还原各个线程加锁和释放锁的场景，在文章开始前，我们先思考下以下几个常见的面试题：

1.偏向锁的加锁流程能简单说下吗？一般适用于什么场景？

2.轻量级锁是如何加锁的？为什么说轻量级锁会比偏向锁更重一点呢？

Synchronized各种锁的升级流转图如下图所示：



2.偏向锁

2.1 偏向锁的引入

从JDK1.6开始，JVM对比较重的Synchronized就引入了各种锁优化的机制，偏向锁算是最轻的一种锁，应用的场景也是最简单的，即偏向锁是专门为了优化线程多次获取同一把锁的场景量身打造的。

比如说方法A是被Synchronized修饰的，线程1可能会多次调用方法A，这样每次都要加锁和释放锁，而获取锁和释放锁是切切实实伴随着性能的损耗的。

因为我们现在已经知道当前场景，即一个线程多次来获取一把锁，能不能说第一次获取锁成功之后，后面要是同一个线程再来获取锁时、就可以快点获取锁，根据这个想法，偏向锁机制被提出来了。

2.2 偏向锁的获取

Synchronized加锁时，肯定是要加锁的，对谁加锁呢：

访问Synchronized修饰的实例方法，锁对象就是方法对应的实例对象；

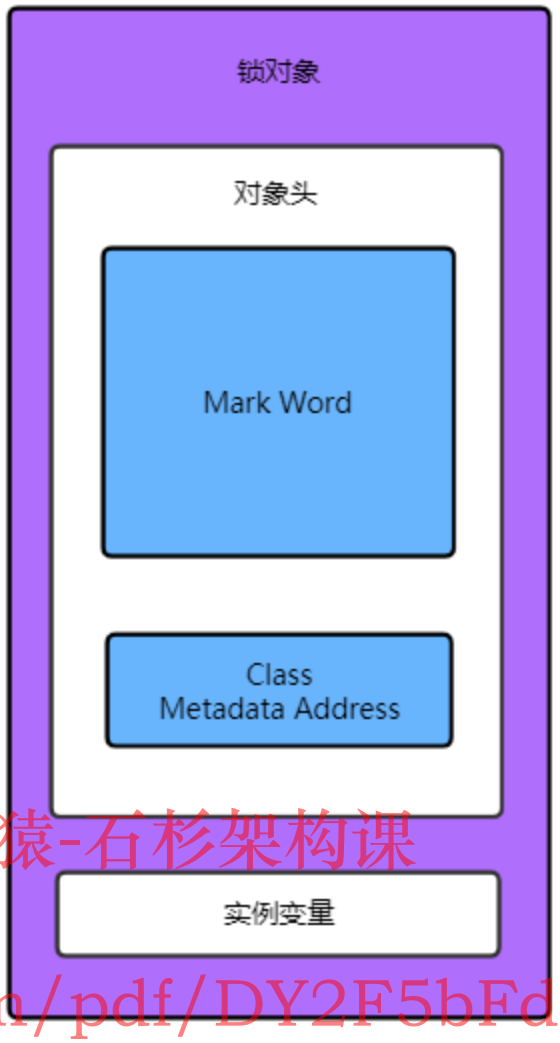
访问Synchronized修饰的静态方法，锁对象就是方法所在的类、即类的字节码对象；

访问Synchronized代码块，那锁对象就是Synchronized当前指定的对象；

而偏向锁的具体实现细节，得要从锁对象的结构中进一步寻找答案。

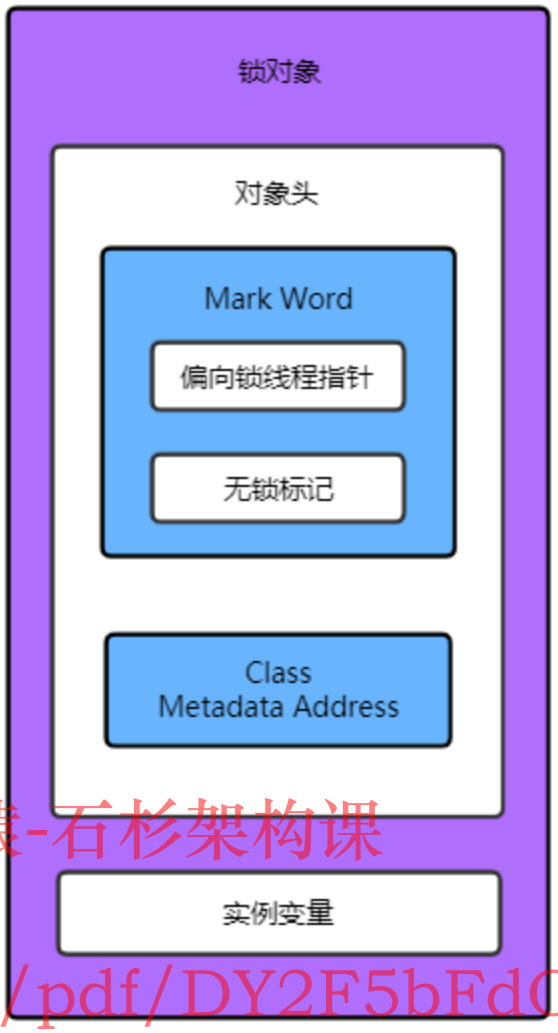
首先锁对象包含对象头以及其他的一些成员变量，包括成员变量、实例方法等，重点我们关注对象头。

对象头里面包含的内容比较多，比如MarkWord、ClassMetadataAddress（就是当前锁对象的类在方法区中的地址）等，如下图所示：

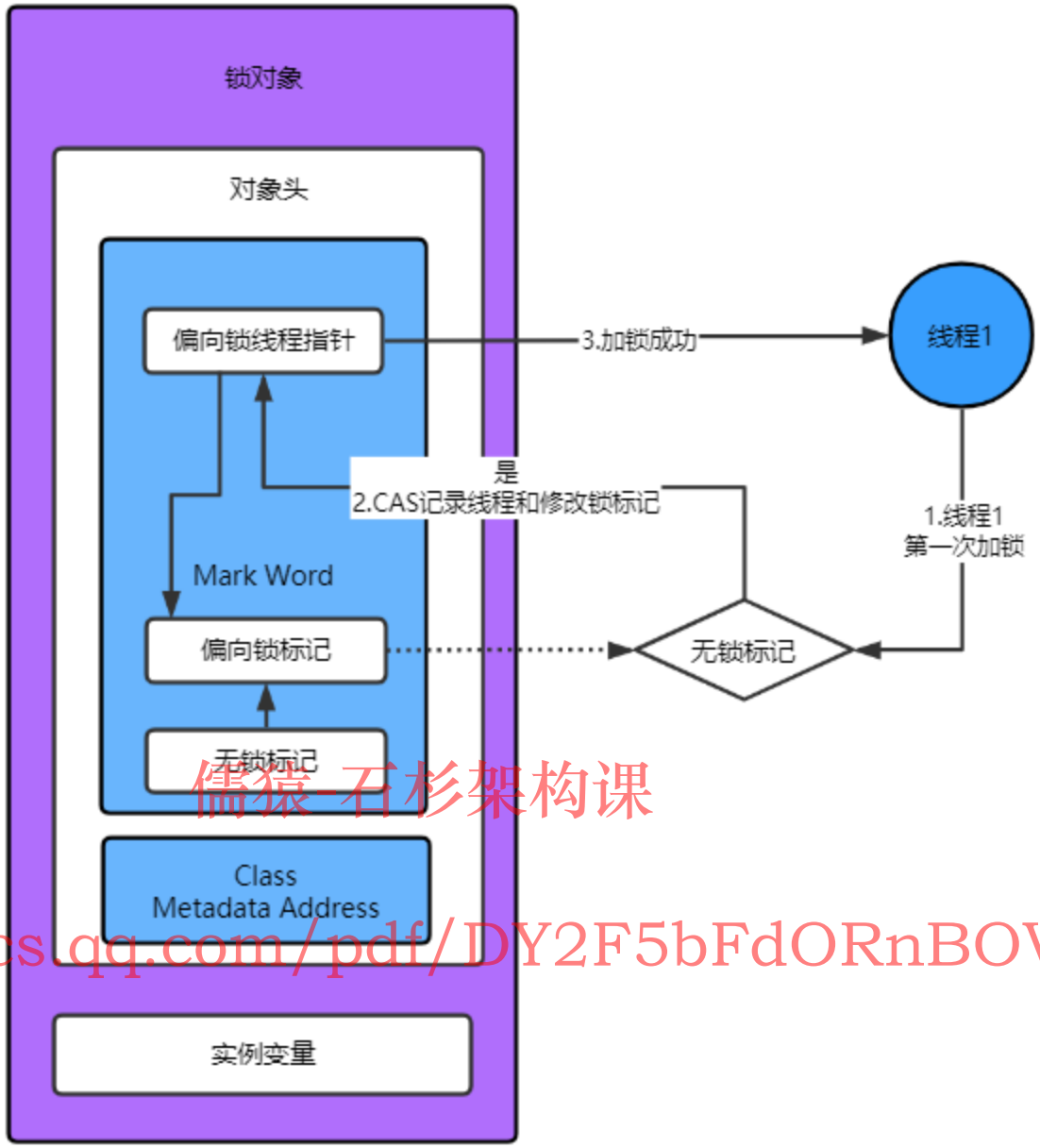


而偏向锁实现的关键主要依附于Mark Word来实现：Mark Word中有一个标志位用来记录当前锁的状态，比如当前是无锁状态还是偏向锁状态呢?

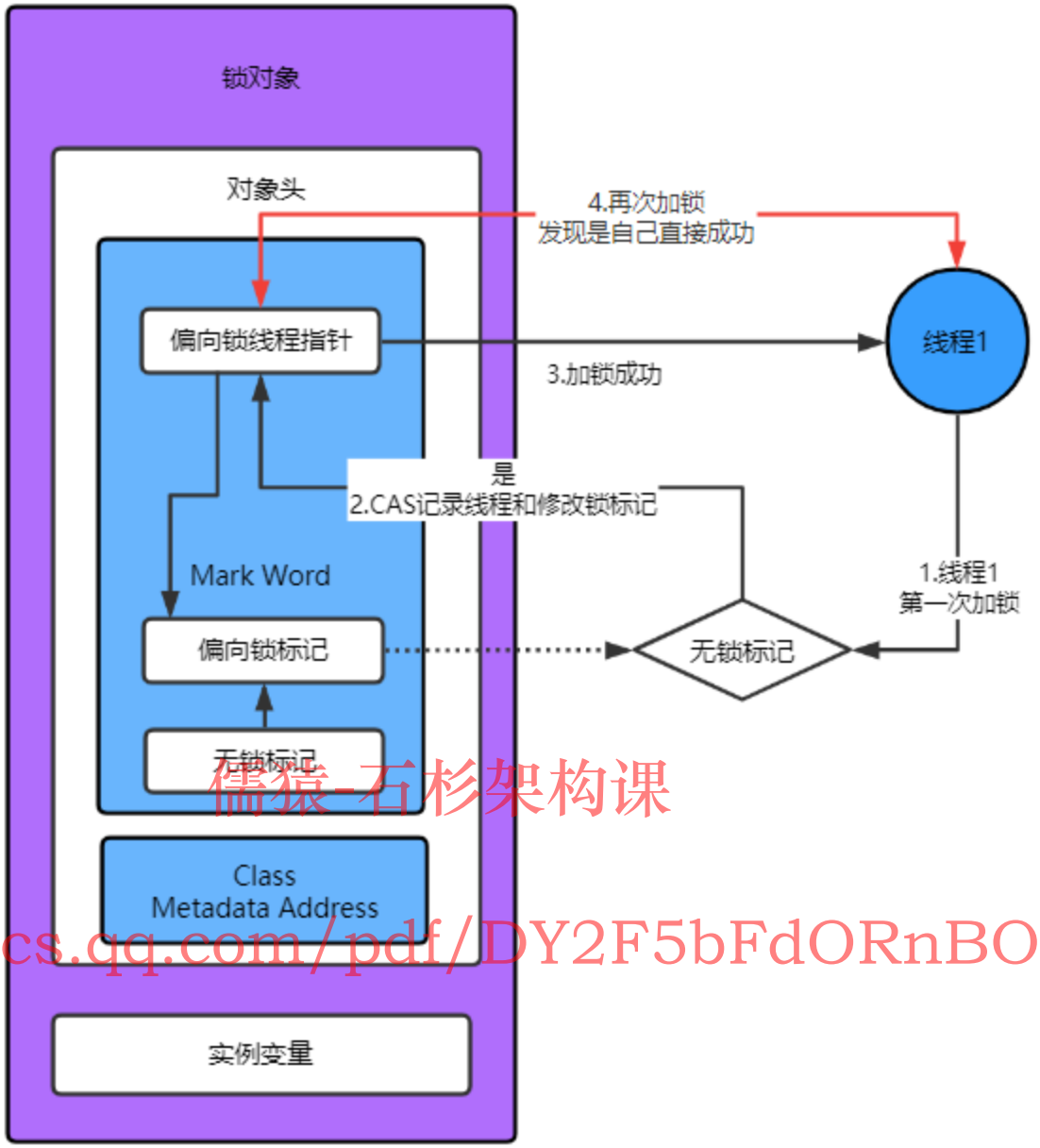
另外当一个线程获取锁成功后，Mark Word中还有一个指向那个获取锁成功的线程指针；一开始获取锁的线程的指针当然为空，并且锁的状态为无锁状态，如下图所示：



此时线程1过来获取Synchronized锁，线程1先看下锁的状态，如果为无锁状态，就直接将偏向锁的线程指针CAS改为指向线程1，然后将锁标记位CAS改为偏向锁标记，标志着线程1获取偏向锁成功，如下图所示：



如果说线程1以后再要对同一个Synchronized加锁，此时就到锁对象对象头中的Mark Word中看一下，发现当前已经是偏向锁标记了，并且当前偏向锁线程指针也是指向线程自己，此时就可以几乎零性能损耗获取锁成功了，如下图所示：



大家现在应该能够感觉到偏向锁的优化好处了吧，偏向锁、偏的是同一个线程，一个线程只在第一次获取锁时、需要CAS修改下锁的标记以及线程指针、耗费一点性能，而往后的重复加锁都是以极低的代价直接获取锁成功的。

2.3 竞争释放偏向锁

偏向锁的使用场景前面我们提到过一点，它是适合线程多次获取同一把锁而做的优化措施，适用于竞争不是很激烈的场景，一旦多线程竞争激烈了，此时就会导致偏向锁的释放和锁的升级。

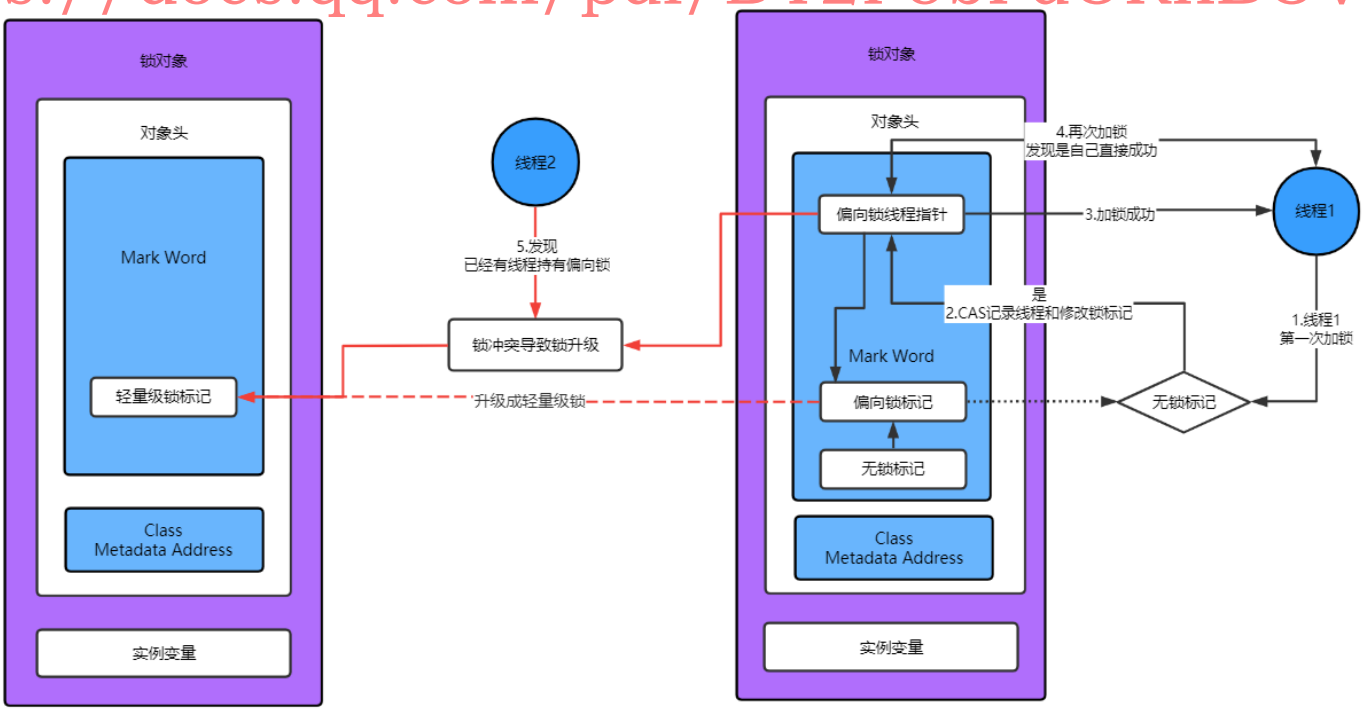
此时，如果线程2也过来也对同一个Synchronized加锁会怎样呢，那就得要看下之前的线程1是否还存活了：

如果线程1已经执行结束了，那么此时线程2过来一看锁标记位依然为是偏向锁，但是线程指针指向的线程1已经没了，此时就将偏向锁线程的指针指向线程2，线程2获取偏向锁成功，偏向锁后续就偏向线程2了。

但是如果当前锁标记为偏向锁，并且当前线程1依然还是存活的，此时线程2过来也要对同一个Synchronized获取锁，此时就构成竞争、也就不符合偏向锁适用的场景了，此时就会释放线程1获取的偏向锁，并且将锁标记CAS改为轻量级锁，轻量级锁我们后面继续看。

从这里我们可以感受到，偏向锁的使用场景并不是很狭义的只有一个线程工作的场景，而是说竞争非常小，小到不足以发生竞争，这样对单个线程的多次获取同一个锁优化效率就可以充分发挥出来了。

此时释放偏向锁的场景如下图所示：



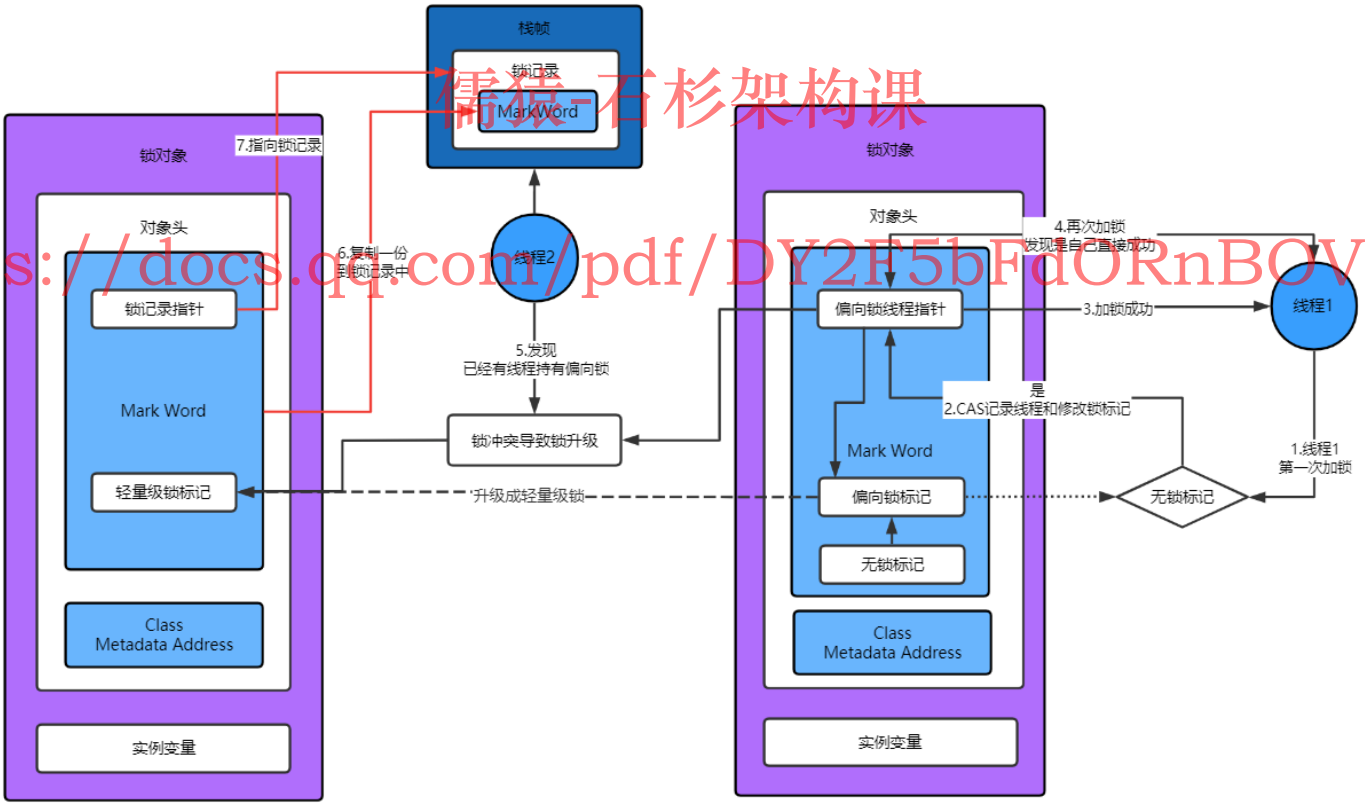
3.轻量级锁

3.1 轻量级锁的获取

前面我们分析到线程2过来加锁时、和线程1产生冲突，线程1的偏向锁释放并被迫升级为了轻量级锁，那此时线程1和线程2都没有获取到锁，线程1和线程2都会同时CAS去获取轻量级锁。

偏向锁的加锁机制为修改Mark Word中锁标记位为偏向锁、并且线程指针指向当前线程，那么轻量级锁的加锁机制是怎样的呢，我们继续看下。

比如现在是线程2过来加锁，此时就会在线程2对应的栈帧中开辟一块锁记录的内存空间，然后将锁对象中的Mark Word复制一份到栈帧中的锁记录中，并且将锁对象中的线程指针指向栈帧中的锁记录空间，这样轻量级锁算是加成功了，如下图所示：



3.2 轻量级锁释放锁

当线程2的轻量级锁释放时，此时就会逆向操作，将线程2栈帧锁记录中的Mark Word CAS替换回锁对象中的Mark Word，如果替换成功，就表示释放轻量级锁成功。

看到这里我们发现，之所以轻量级锁比偏向锁稍微重那么一点，就是相比于偏向锁而言，轻量级锁每次都要先加锁，然后还需要再释放锁，不像偏向锁、只要不产生竞争，那么第一次加锁一次即可，后续再次获取锁的性能耗费极低，并且也不需要释放锁，自然也就不用承担释放锁性能损耗了。

4.重量级锁

4.1 什么时候膨胀为重量级锁

（1）情况1–其他线程CAS自旋失败导致膨胀为重量级锁

假设线程2还没有释放轻量级锁，此时线程3过来加锁，线程3和线程2一样，先把Mark Word复制一份到自己线程栈帧中的锁记录空间，然后在修改锁记录的指针时、因为锁记录的指针已经指向了线程1，所以此时线程2的CAS修改锁记录指针就会失败。

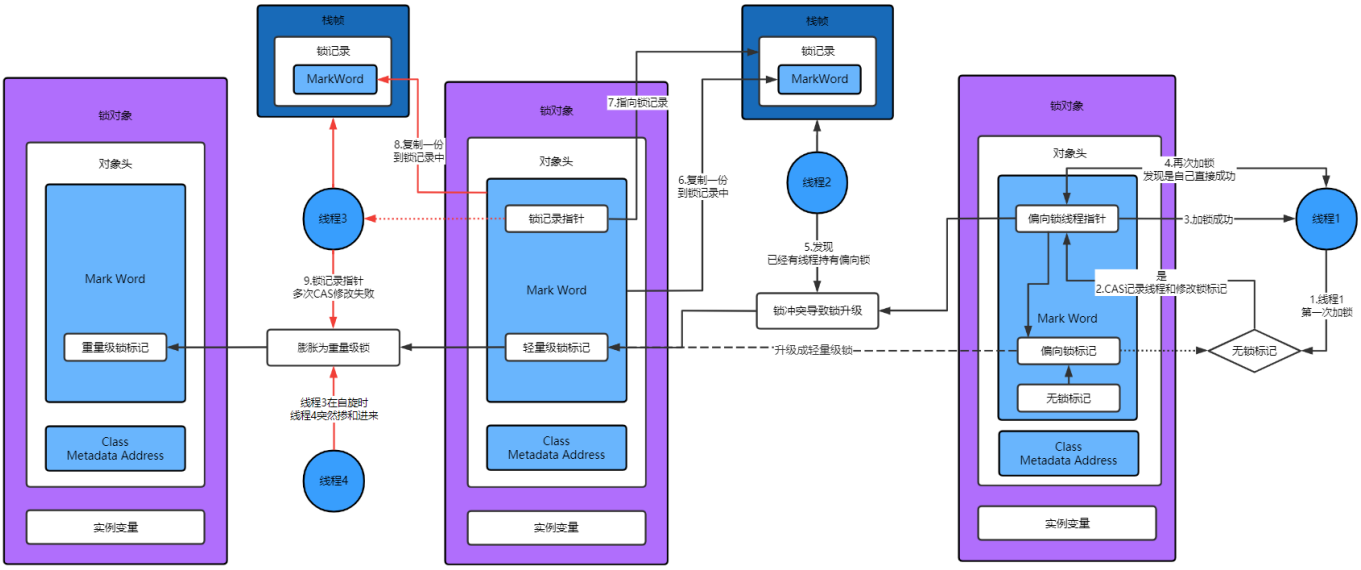
此时线程2的CAS操作失败并不会像偏向锁一样直接导致锁升级，而是先会CAS自旋一会，如果自旋获取锁失败的次数过多，此时才会升级为重量级锁。

（2）情况2–三个线程竞争导致直接膨胀为重量级锁

假设此时也是线程2获取了轻量级锁，线程3过来CAS加锁时、修改锁记录指针时失败了，然后线程3正在自旋获取锁，这时倒好线程4也来获取锁了，此时JVM就会毫不犹豫地将锁膨胀为重量级锁。

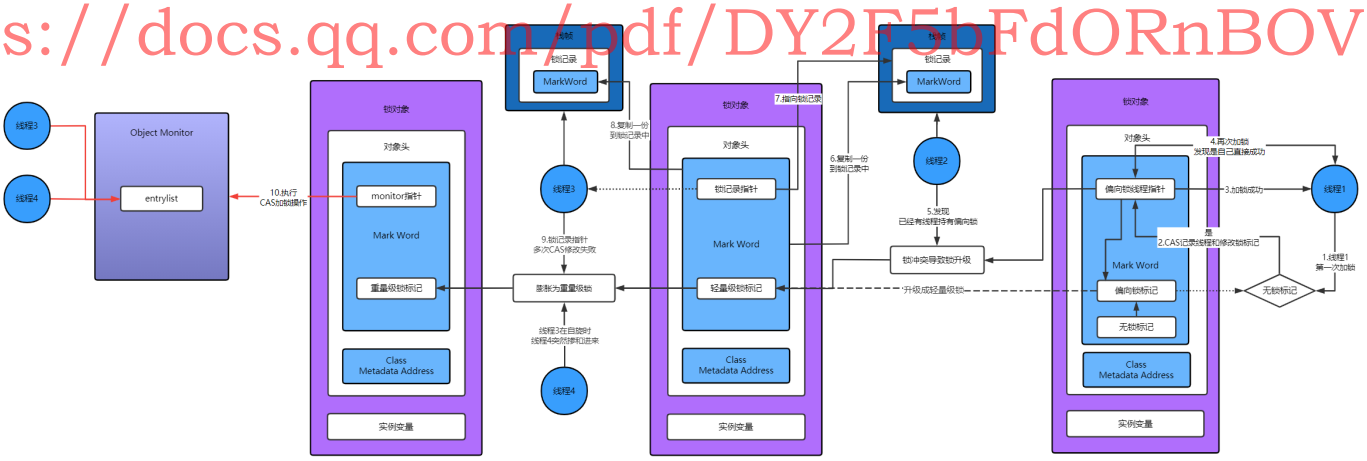
毕竟线程3在那自旋时，要是线程2能快点释放锁，那么线程3还是有机会在最大允许自旋次数内获取到轻量级锁的，这下倒好，线程4直接掺和进来，JVM会认为现在线程3获取锁的希望不大、竞争太激烈了，所以就直接膨胀为重量级锁了。

以上两种膨胀情况如下图所示：

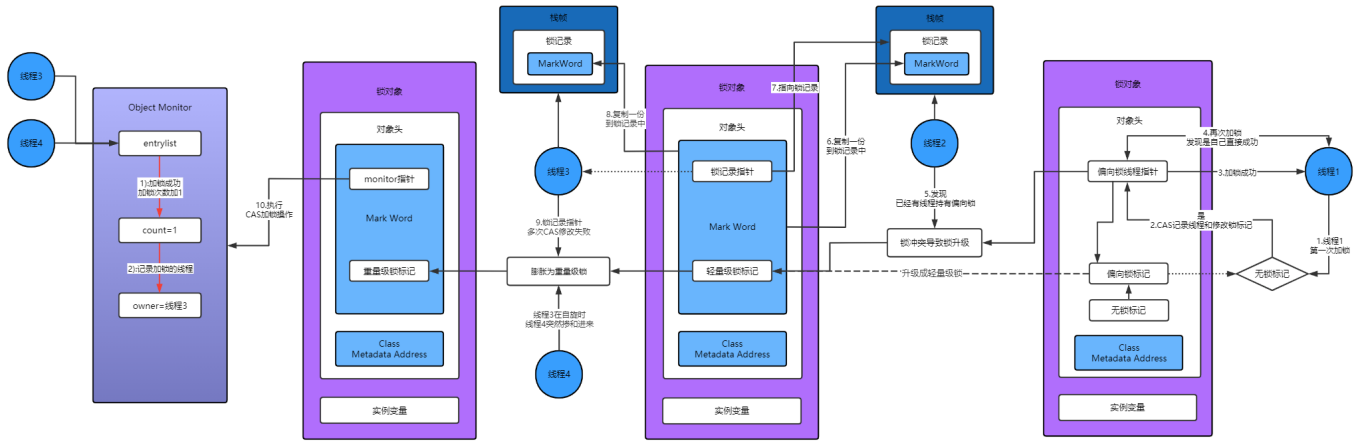


4.2 重量级锁的获取

现在由于锁膨胀为重量级锁，原本的锁记录的指针将会变为指向重量级锁Monitor对象的指针了，并且当前所有没获取到锁的线程优先进入到Monitor对象中entrylist中等待获取锁，如下图所示：



然后多线程依次去获取锁，比如现在线程3开始获取锁，此时底层会执行monitorenter指令，然后在Object Monitor中锁的计数器加1，并且记录当前持有锁的线程为线程3，后面如果线程3再次过来加锁时发现当前持有锁的线程正是线程3，直接将计数器加1，表示又重入了一次锁，如下图所示：



4.3 重量级锁的释放

重量级锁的释放比较简单，直接递减加锁次数count：

递减后如果count为0，表示当前线程该释放的锁都释放完了，此时直接将线程指针owner置空，释放锁完毕；

如果递减后count还不为0，那就表示线程之前对这个锁重入过好几次，那还得线程再次执行释放锁操作，直到count为0才算释放彻底了、才能将线程指针置空。

5.JIT编译器的优化

java代码编译生成class字节码文件是静态编译过程，那么class字节码文件编译成计算机能执行的机器指令，这个阶段就包含了JIT编译。

Synchronized加锁时，除了对不同竞争程度做的各种锁优化之外，在最开始的JIT编译阶段也对Synchronized也做了一定程度的优化。

5.1 锁消除

优化点之一就是锁消除，锁消除顾名思义就是不要你加的锁，把你加的Synchronized锁删除掉，免得加锁还要浪费性能，那什么场景下要锁消除呢?

一般是在一些不需要加Synchronized锁的地方加锁了，比如一个方法内，涉及到的变量都是局部变量，并且要是根据逃逸分析技术分析，发现完全没有将当前线程资源暴露给其他线程看见的可能，既然其他线程没有任何机会看见当前线程的资源，也就不会出现并发的问题，既然没有并发的问题，那你还莫名其妙加个锁岂不是浪费性能了。

所以JIT底层默默为该现象把了道关，即锁消除的优化，要是在没有多线程并发冲突的地方还加Synchronized锁，底层就不会执行monitorenter和monitorexit相关的指令了，性能能优化一点是一点。

5.2 锁粗化

我们都知道Synchronized锁比较重、也是比较耗费性能的。比如有个场景，你在一个方法内、对同一个对象加了多次Synchronized锁，此时可能就对应有多个Synchronized代码块，面对这种场景，作为coder的我们要是看到这个场景，都忍不住想要优化、即将多个Synchronized代码块合并在一个Synchronized代码块中，否则你就要先加锁再释放锁，重复多次这种耗费性能的操作。

确实JIT编译时也是这样设计的，底层将不会多次执行monitorenter和monitorexit指令，如果发现居然是对同一个对象多次加Synchronized锁，底层将会把他们合并成一个Synchronized代码块，然后只会执行一次monitorenter和monitorexit，达到了优化的目的，底层这里又对代码的性能把了道关。

现在我们可以知道，锁粗化，顾名思义就是多个Synchronized代码块在底层JIT编译时合并成一个，好似一个锁变粗了一样，尽可能的避免了不必要的、多次加锁和释放锁带来的性能损耗。

5.3 自适应加锁

我们前面都看到了，在前面多个线程获取锁的时候，如果获取锁失败，都会自旋一会儿，然后再去尝试获取锁，也就是自旋加锁。

其实这个做法已经比较先进了，相比于一旦获取锁失败就阻塞、然后线程上下文切换执行其他线程而言，确实已经很好了，毕竟线程上下文切换是很耗费性能的，如果说你能够自旋一会儿，等到其他线程

释放锁后就能获取到锁了，虽然说自旋一会儿耗费了点CPU，但是相比于线程上线文切换的代价，如果自旋一小会儿就可以获取锁那还是更好的。

但是问题来了，要是当前的环境竞争非常激烈、我一直自旋都获取锁失败，可能都超过了线程上下文切换带来的性能损耗，那自旋获取锁岂不是性能更差。面对这个问题，提出了自适应加锁的概念。

即根据最近几次线程自旋获取锁成功的概率，来决定当前线程是自旋一会儿来等待获取锁、还是直接阻塞切换线程上下文让其他线程执行；

如果最近一段时间线程竞争程度比较激烈，那就不适合一直自旋等待；如果最近几次线程竞争很小，自旋获取锁的成功率比较高，那可以自旋等待一会来获取锁，通过这样比较智能的自适应过程，来权衡自旋加锁和线程上线文切换带来的性能损耗。

6.面试题剖析

1.偏向锁的加锁流程能简单说下吗？一般适用于什么场景？

偏向锁加锁时，首先会在Synchronized对应锁对象对象头中的Mark Word中，先看下锁的标记：

如果是无锁状态，CAS修改为偏向锁标记，然后将Mark Word中的线程指针指向当前线程，表示当前线程已经获取到了一个偏向锁；那么后续该线程再来加锁时，发现自己已经获取偏向锁了，此时直接几乎零成本获取偏向锁成功。

如果已经是偏向锁状态，那就看下当前Mark Word中、线程指针指向的线程是不是自己，如果是自己当然直接获取偏向锁成功；如果指针指向的线程不是当前线程、但恰好该指向的线程已经结束了，那么直接将Mark Word中、线程指针指向当前线程自己，表示现在改为偏向当前新来的线程了。

从偏向锁的加锁流程我们可以看到，偏向锁只适合于竞争不激烈的场景，主要是优化同一个线程多次获取同一个锁的性能，如果说线程1已经获取偏向锁成功，并且在线程1还没有结束时，紧接着线程2又过来获取锁，这下一看锁标记为偏向锁、还被线程1持有，那就直接就发生竞争了。

2.轻量级锁是如何加锁的？为什么说轻量级锁会比偏向锁更重一点呢？

首先线程会在自己线程的栈帧中创建一个锁记录空间，然后将Mark Word复制一份到这个锁记录空间中，然后将锁对象中的Mark Word指向这个锁记录空间，轻量级锁就算获取成功了。

因为轻量级锁相比于偏向锁而言，每次加锁使用完毕后都需要释放锁，也就是说每次都需要老老实实的执行加锁和释放锁，而偏向锁只有在线程第一次过来加锁时，才需要CAS修改下锁标记和线程指针，后续重复过来加锁直接零成本获取，同时在没有竞争的场景下一般不会产生释放锁带来的性能损耗。
