目录

Java6对synchronized关键字进行了大幅度的优化，引入偏向锁和轻量级锁的概念，大幅减少了获得锁和释放锁的性能损耗。从Java6开始，加上原有的重量级锁（又叫监视器锁），Java中的对象锁一共有4种状态，分别是：无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态。对象锁只能按照上述的顺序进行升级（又称锁膨胀）操作，锁只要升级之后，就不能降级。本篇博客基于前面几篇的代码分析总结一下偏向锁、轻量级锁和重量级锁的实现及其锁膨胀的流程。

一、BasicObjectLock / BasicLock

这两个都是加锁时使用的数据结构，其定义如下：

```
class BasicObjectLock VALUE_OBJ_CLASS_SPEC {
  friend class VMStructs;
 private:
  BasicLock _lock;                                    // the lock, must be double word aligned
  oop       _obj;                                     // object holds the lock;

 public:
  // Manipulation
  oop      obj() const                                { return _obj;  }
  void set_obj(oop obj)                               { _obj = obj; }
  BasicLock* lock()                                   { return &_lock; }

  // Note: Use frame::interpreter_frame_monitor_size() for the size of BasicObjectLocks
  //       in interpreter activation frames since it includes machine-specific padding.
  static int size()                                   { return sizeof(BasicObjectLock)/wordSize; }

  // GC support
  void oops_do(OopClosure* f) { f->do_oop(&_obj); }

  static int obj_offset_in_bytes()                    { return offset_of(BasicObjectLock, _obj);  }
  static int lock_offset_in_bytes()                   { return offset_of(BasicObjectLock, _lock); }
};
```

```
class BasicLock VALUE_OBJ_CLASS_SPEC {
  friend class VMStructs;
 private:
  volatile markOop _displaced_header;
 public:
  markOop      displaced_header() const               { return _displaced_header; }
  void         set_displaced_header(markOop header)   { _displaced_header = header; }

  void print_on(outputStream* st) const;

  // move a basic lock (used during deoptimization
  void move_to(oop obj, BasicLock* dest);

  static int displaced_header_offset_in_bytes()       { return offset_of(BasicLock, _displaced_header); }
};
```


其中lock属性的地址用于实现轻量级锁，即所谓的Thread ID；obj属性用于保存关联的锁对象oop；displaced_header属性用于保存锁对象oop的原始对象头，即无锁状态下的对象头，但是在synchronized嵌套的情形下displaced_header为NULL，因为外层synchronized对应的BasicObjectLock已经保存了原始对象头了，此处不需要再保存；另外，如果某个轻量级锁膨胀成重量级锁了，则displaced_header会被置为unused_mark，因为重量级锁本身会保存锁对象oop的原始对象头。synchronized嵌套的示例如下：

```
public static void main(String[] args) {
    Object lock=new Object();
    synchronized (lock){
        System.out.println("start");
        synchronized (lock){
            System.out.println("inner");
        }
        System.out.println("end");
    }
}
```


准备加锁前会从当前方法对应的调用栈帧中查找一个空闲的BasicObjectLock，obj属性为NULL的就表示空闲的，查找时先从低地址即从最近才分配的BasicObjectLock开始遍历，如果找到一个obj属性跟目标对象一致的就终止遍历，BasicObjectLock就是外层synchronized对应的BasicObjectLock，如果没有找到obj属性跟目标对象一致的就会把所有的BasicObjectLock都遍历一遍，取地址最高的一个空闲BasicObjectLock，提高BasicObjectLock的使用率。

准备解锁前会从当前方法对应的调用栈帧中找到一个obj属性与目标对象一致的BasicObjectLock，遍历时从低地址即从最近才分配的BasicObjectLock开始遍历，找到了就立即返回，如果没有找到则抛出异常IllegalMonitorStateException。

注意上述的加锁解锁都是针对于synchronized关键字，如果是通过jni_MonitorEnter/jni_MonitorExit方法，Unsafe类的monitorenter/monitorexit方法加锁解锁则不需要分配BasicObjectLock，而是直接分配并使用重量级锁。

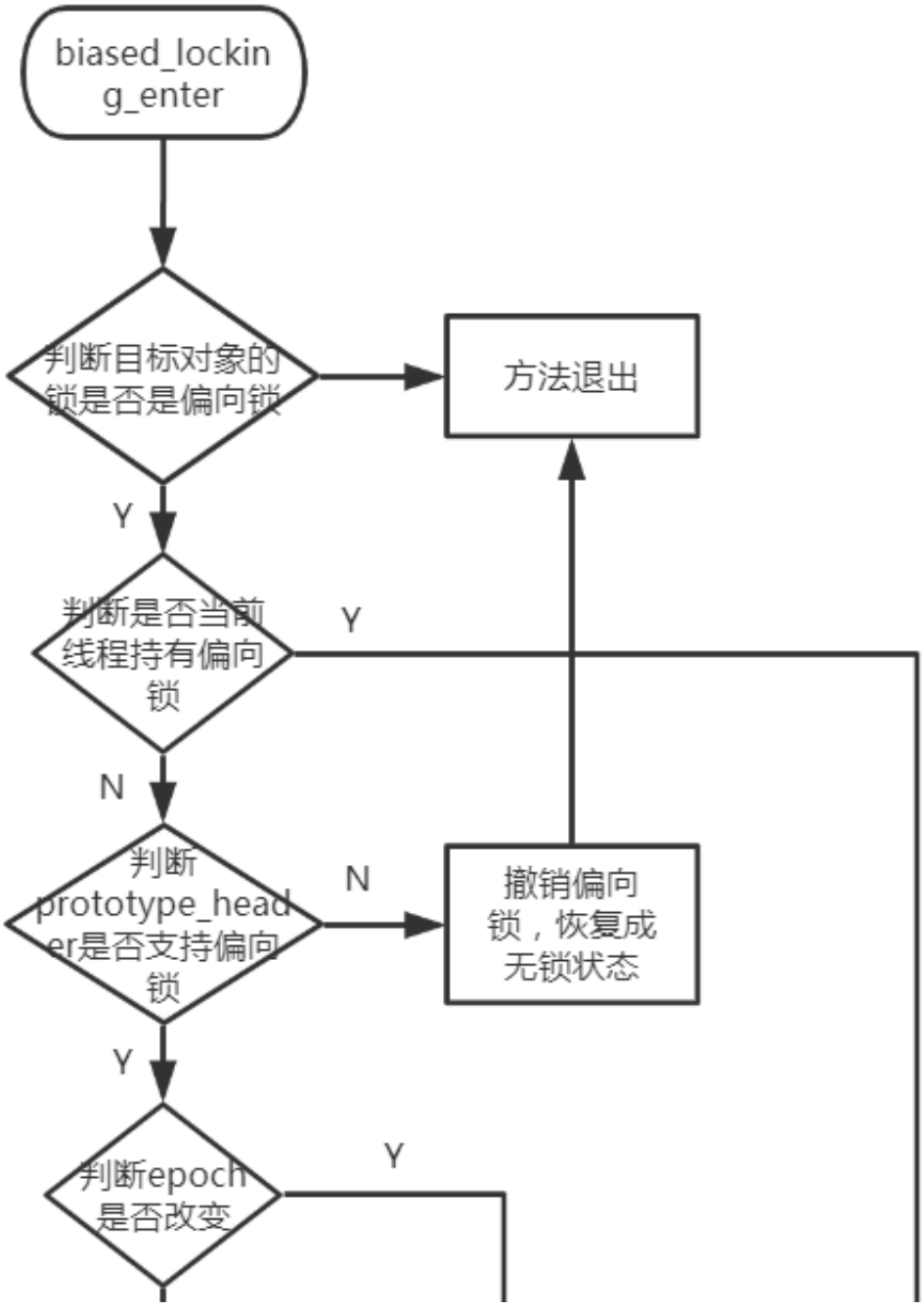
二、偏向锁

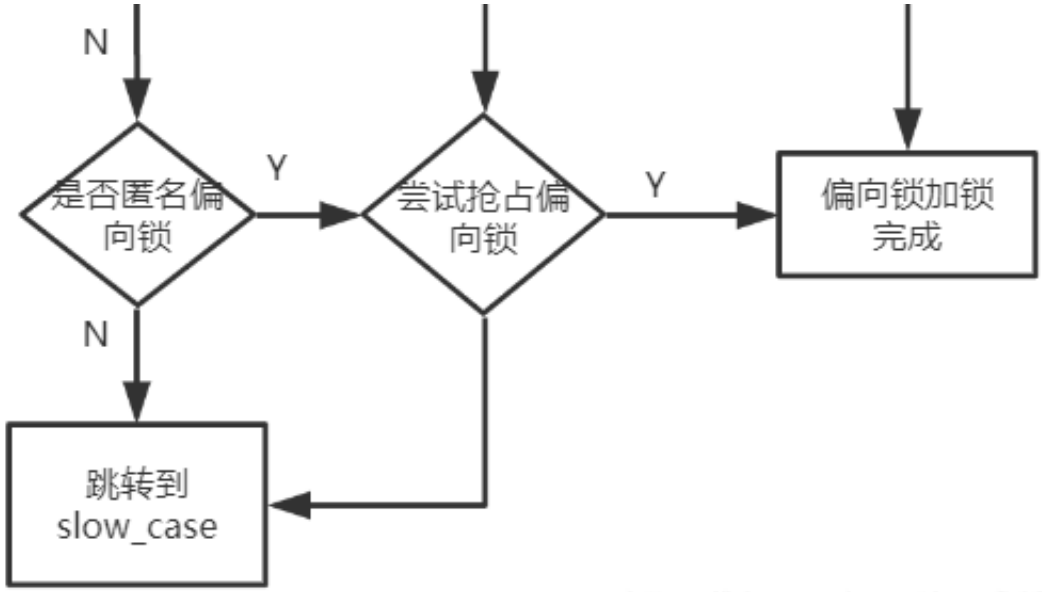
适用场景：只有一个线程进入同步代码块

定义：偏向锁顾名思义会偏向于一个线程，如果有其他线程来抢占这个偏向锁则会导致偏向锁被撤销，恢复成无锁状态或者直接膨胀成轻量级锁。当偏向锁没有被某个线程占用，即对象头中用于保存线程指针的位都是0的时候，这个偏向锁就是匿名偏向锁。偏向锁可以通过-XX:-UseBiasedLocking来关闭，该参数默认为true。

初始化：偏向锁默认是延时初始化的，延迟的时间通过参数BiasedLockingStartupDelay控制，默认是4000ms。初始化是在安全点下通过VMThread完成的，初始化时会把由SystemDictionary维护的所有已加载类的Klass的prototype_header修改成匿名偏向锁对象头，并把_biased_locking_enabled静态属性置为true，后续加载新的Klass时发现该属性为true，会将Klass的prototype_header修改成匿名偏向锁对象头。当创建某个Klass的oop时，会利用Klass的prototype_header来初始化该oop的对象头，即偏向锁初始化完成后，后续所有创建的oop的初始对象头都是匿名偏向锁的，在此之前创建的oop初始对象头都是无锁状态的。

加锁流程：





偏向锁的加锁实现就是MacroAssembler::biased_locking_enter方法，其中slow_case就是调用InterpreterRuntime::monitorenter方法，该方法会判断目标偏向锁是否正在使用，如果是则将其直接膨胀成轻量级锁，如果不是则恢复成无锁状态；其中抢占偏向锁就是通过CAS原子的将当前线程对象的指针通过或运算写入锁对象的对象头中，写入成功视为抢占成功。

解锁：解锁的实现在MacroAssembler::biased_locking_exit方法中，其实现非常简单，只是简单的判断下目标对象的对象头是否还是偏向锁，如果是则解锁完成，注意此时并不会将里面的线程指针去掉，这样做是为了该线程下一次加锁的时候可以很快的完成加锁；如果不是，则退出该方法。

撤销：撤销偏向锁就是将锁对象oop的对象头恢复成无锁状态或者膨胀成轻量级锁状态，执行撤销动作的前提是锁对象oop的对象头处于偏向锁状态。具体而言有以下几种情形：

1. 执行Object类的hashcode方法，会将其恢复成无锁状态。
2. 执行Object类的wait/notify/notifyall方法，会将其恢复成无锁状态，直接膨胀成重量级锁。
3. 执行jni_MonitorEnter或者jni_MonitorExit方法，会将其恢复成无锁状态，直接膨胀成重量级锁。
4. 执行Unsafe类的monitorenter/trymonitorenter/monitorexit方法，会将其恢复成无锁状态，直接膨胀成重量级锁。
5. 尝试获取某个偏向锁，如果该偏向锁被某个线程占用了，但是没有关联的BasicObjectLock，即实际占用该偏向锁的方法已经退出了，则会将其恢复成无锁状态，然后膨胀成轻量级锁，但是在撤销一定次数后触发批量重偏向（rebasic）的情形下也可能重新获取该偏向锁。如果该偏向锁正在被某个方法所使用，即存在对应的BasicObjectLock，则直接将该偏向锁膨胀成轻量级锁。

注意偏向锁的撤销大部分情形下都是需要在安全点下执行，因为需要遍历其他线程的所有调用栈帧，判断是否存在与之关联的BasicObjectLock。在以下情形不需要在安全点下执行：

1. 目标对象的对象头是匿名偏向锁状态
2. 目标对象的Klass的prototype_header变成无锁状态
3. 目标对象的Klass的prototype_header中的epoch值和目标对象对象头中的epoch值不一样
4. 目标对象的偏向锁由当前线程持有

批量重偏向： 因为偏向锁的撤销大部分情形都是在安全点下执行的，安全点同步会导致系统停顿，整体性能损耗较高，所以当个某一类Klass对应的锁对象oop被累计撤销一定次数后就会触发批量重偏向，这个次数通过BiasedLockingBulkRebiasThreshold参数控制，默认是20。批量重偏向会先将Klass的prototype_header中的epoch值加1，然后遍历所有JavaThread的所有栈帧，遍历每个栈帧中包含的BasicObjectLock，如果其关联的锁对象oop是该Klass，则增加该oop的对象头中的epoch值，遍历完成后将触发批量重偏向的这个锁对象oop重新偏向给当前线程。注意此时不在栈帧中的即未被实际占用的锁对象的oop的epoch值就不会改变，重新获取该锁对象oop的偏向锁时因为epoch值不一致就可以重新被其他线程抢占，即提高偏向锁oop的使用率。

批量撤销：触发批量重偏向后如果频繁撤销，在一定的时间段内超过一定次数则触发批量撤销，时间段通过参数BiasedLockingDecayTime控制，默认是25000ms，次数通过BiasedLockingBulkRevokeThreshold参数控制，默认是40，即25s内该Klass的锁对象oop撤销偏向锁累计超过40次后就触发批量撤销。批量撤销会将将Klass的prototype_header恢复成无锁状态，即新创建的oop的对象头都是无锁状态，不支持使用偏向锁了，然后遍历所有JavaThread的所有栈帧，遍历每个栈帧中包含的BasicObjectLock，如果其关联的锁对象oop是该Klass，则将该锁对象oop的对象头膨胀成轻量级锁。

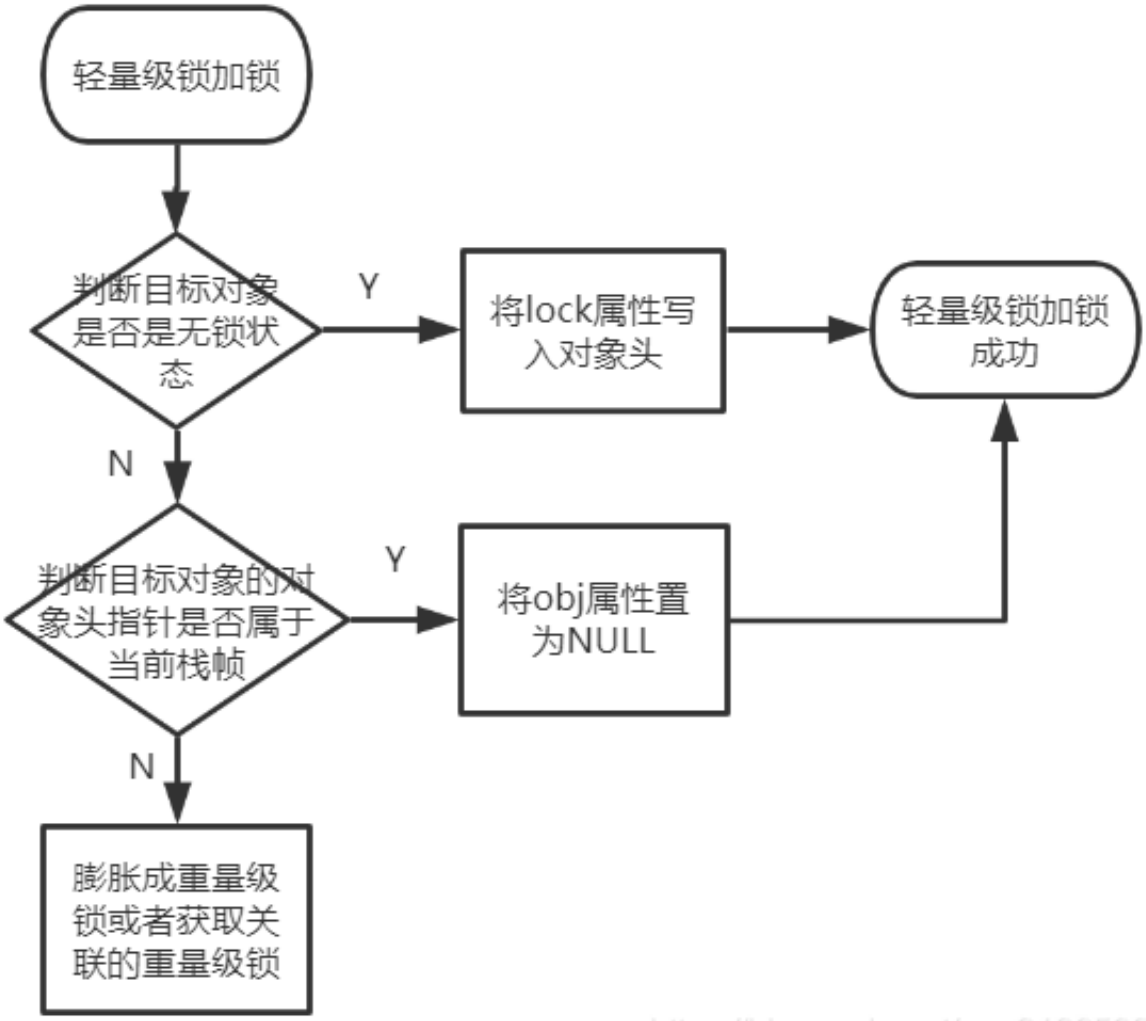
上述批量动作的触发逻辑都在BiasedLocking::update_heuristics方法中，具体的执行逻辑在BiasedLocking::bulk_revoke_or_rebias_at_safepoint方法中，跟单个撤销不同，批量动作必须在安全点下执行，因为需要遍历所有Java线程的所有栈帧。

三、轻量级锁

适用场景：多个线程交替进入同步代码块

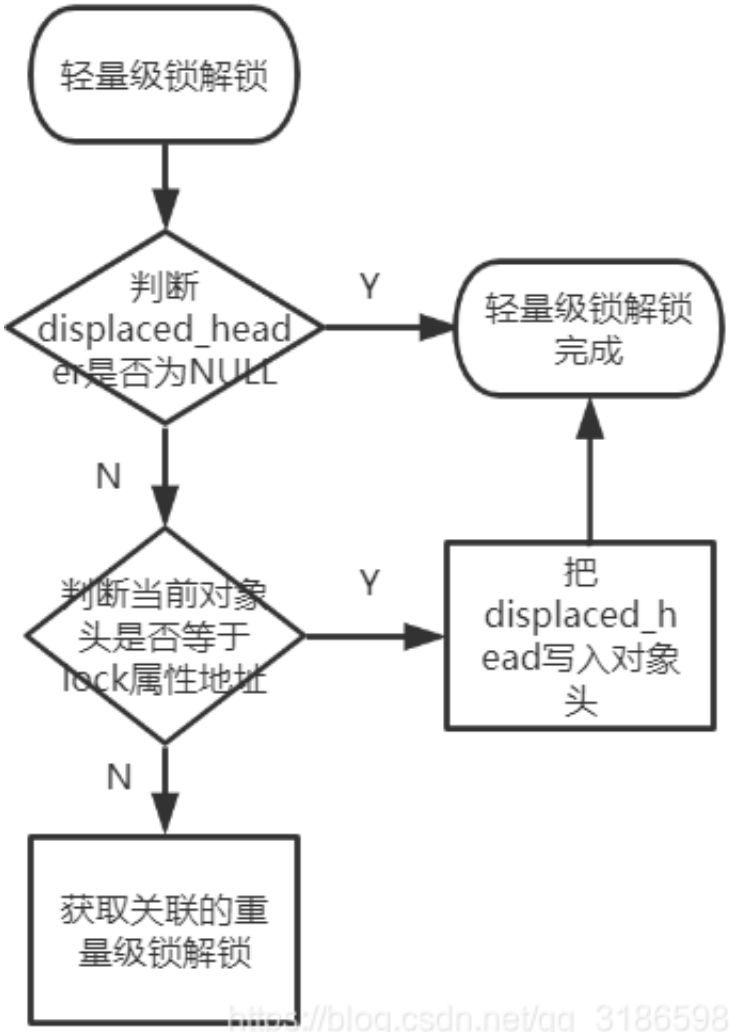
定义：轻量级锁是相对于重量级锁的，即加锁的成本更低，性能更好，只需要将lock属性地址通过CAS写入对象头即视为加锁成功，因为BasicLock只有一个8字节属性，所以lock属性的地址是8字节对齐的，其最后3位刚好是000。轻量级锁是必须开启的，没有参数将其关闭。当存在多个线程抢占轻量级锁的时候，只有一个能够抢占成功，获取轻量级锁恢复正常执行，其他线程都会尝试将该轻量级锁膨胀成重量级锁，也只有一个线程完成锁膨胀。

加锁流程：



轻量级锁的加锁流程在TemplateTable::monitorenter 和InterpreterRuntime::monitorenter方法中都有，其逻辑是一样的，前者是monitorenter字节码指令的实现，后者是获取偏向锁或者轻量级锁失败后执行的逻辑。注意将lock属性写入对象头是CAS原子动作，只有一个线程能够写入成功，写入成功视为抢占成功。

解锁流程：



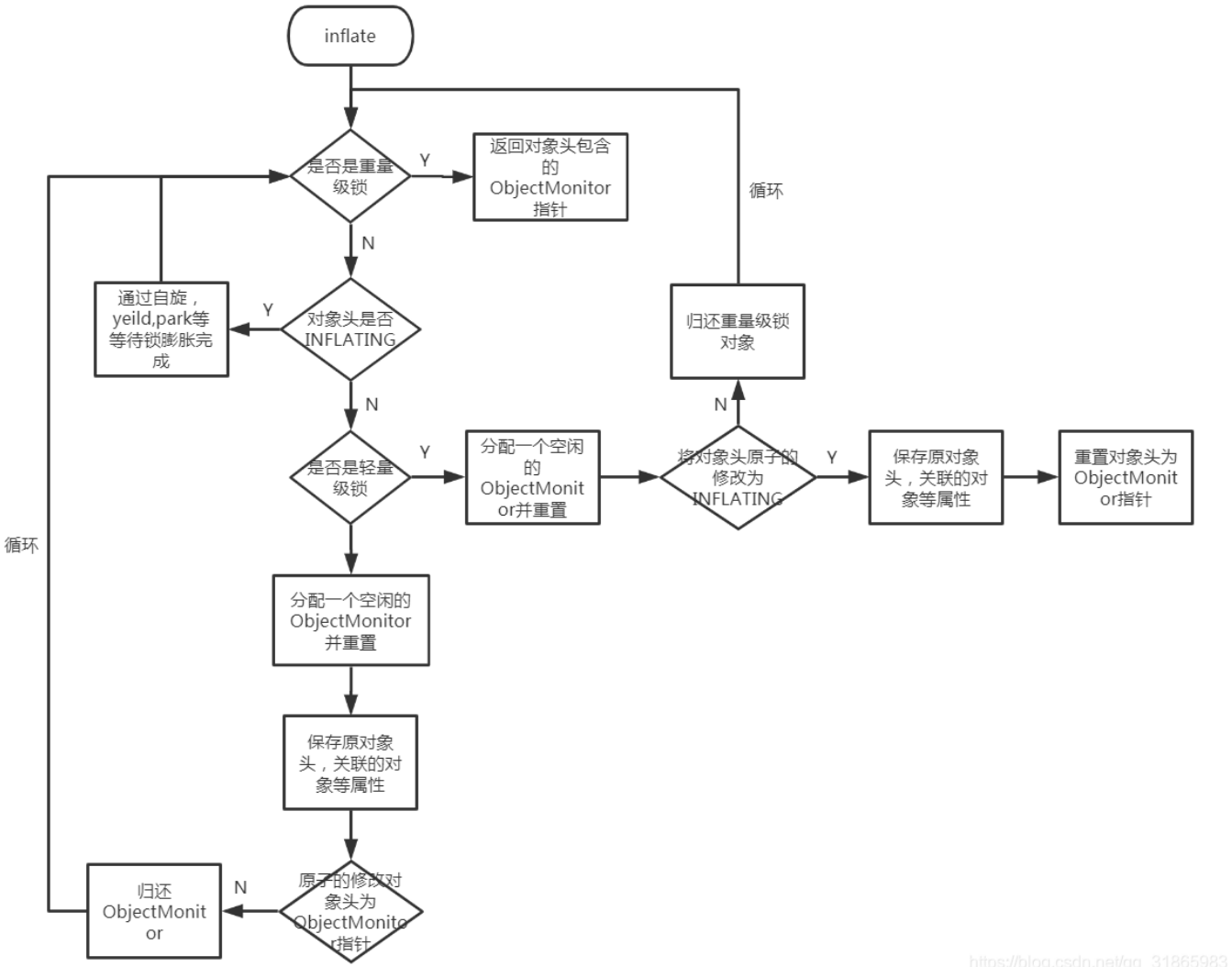
displaced_header为NULL时就是synchronized嵌套的情形，此时解锁不需要恢复锁对象的对象头，最外层synchronized解锁时会恢复对象头。将displaced_header中的对象头写入锁对象动作是CAS的，如果成功，解锁成功，如果失败，说明某个线程已经将该轻量级锁膨胀成重量级锁了，需要获取对应的重量级锁，完成解锁动作。对比偏向锁的解锁实现可知，轻量级锁支持多个线程占有，但是必须是交替的，不能是同时的。另外很多博客说轻量级锁膨胀成重量级锁前有一个自旋等待的动作，这其实是错误的，轻量级锁的实现只使用了BasicObjectLock一个数据结构，无法支持自旋等待，因为没有地方记录自旋等待的次数，倒是重量级在采用互斥量阻塞当前线程前会先尝试自旋等待一段时间，重量级锁有单独的数据结构可以支持复杂的自旋逻辑。

四、重量级锁

使用场景：多个线程同时进入同步代码块

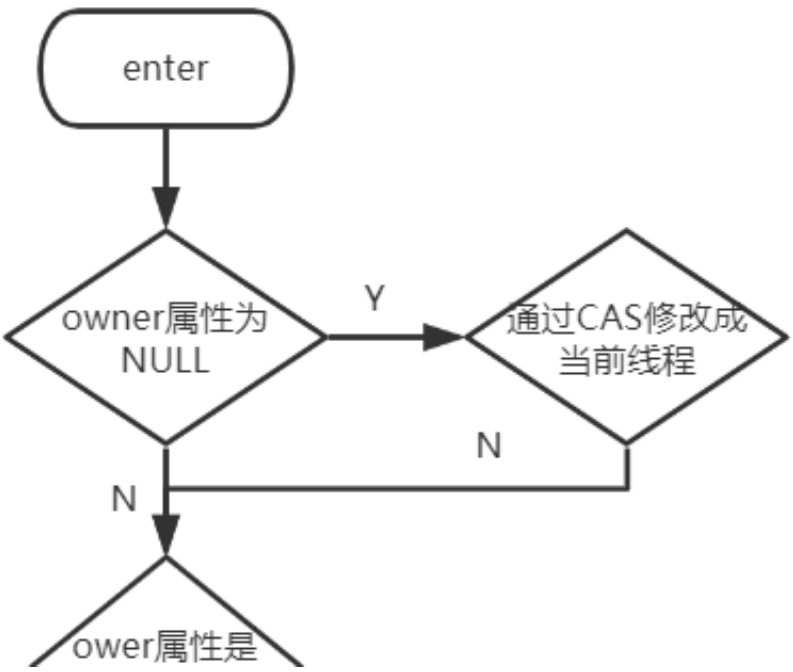
定义：重量级锁就是利用底层操作系统mutex相关API和线程自旋实现的锁，因为mutex API会让线程休眠，导致线程的上下文切换，性能损耗较大，所以称之为重量级锁，也叫监视器锁。重量级锁有专门的数据结构ObjectMonitor，不依赖于BasicObjectLock。

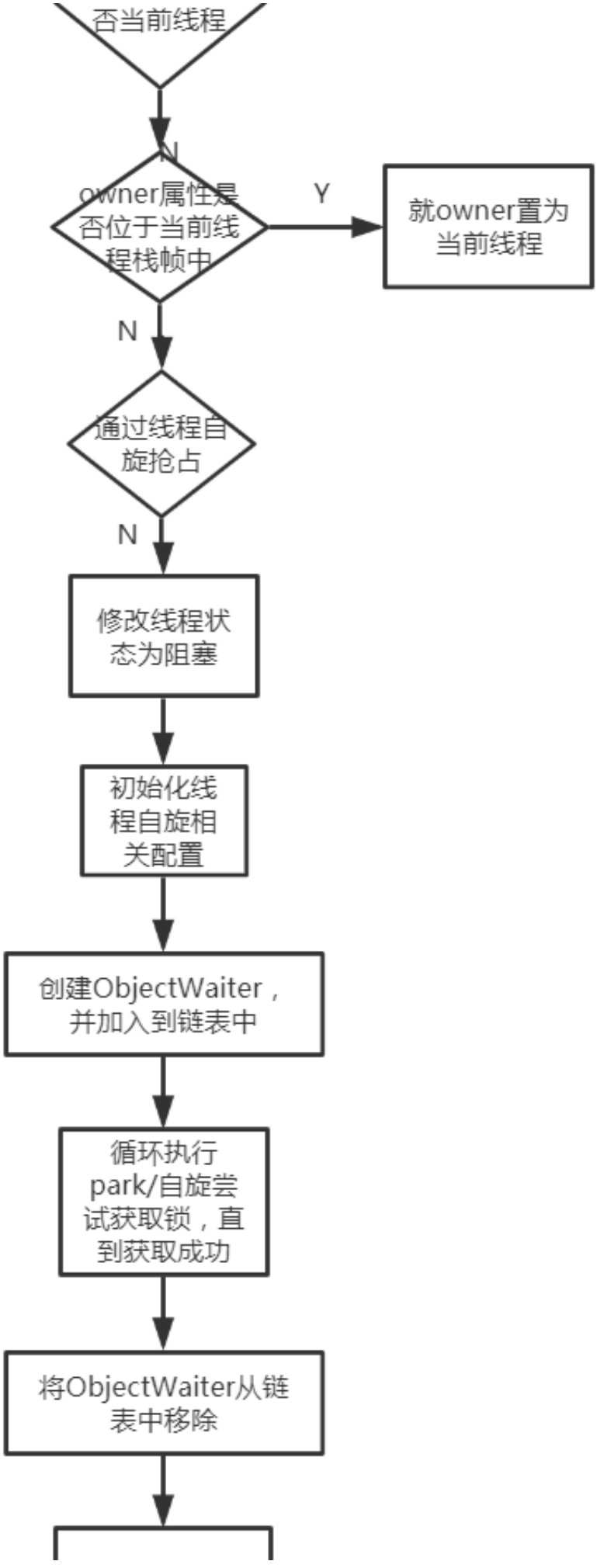
锁膨胀流程：

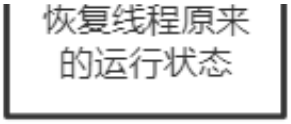


INFLATING是一个非常短暂的中间状态，只存在于轻量级锁膨胀成重量级锁的过程中，主要为了防止对象的hashCode出现闪烁。上述流程中有两处CAS原子修改动作，修改失败就表示有其他线程成功的修改了该对象头，需要重新开始下一次循环，重新获取新的对象头状态。

加锁流程：




否当前线程
N
owner属性是否位于当前线程栈帧中
Y
就owner置为当前线程
N
通过线程自旋抢占
N
修改线程状态为阻塞
初始化线程自旋相关配置
创建ObjectWaiter，并加入到链表中
循环执行park/自旋尝试获取锁，直到获取成功
将ObjectWaiter从链表中移除

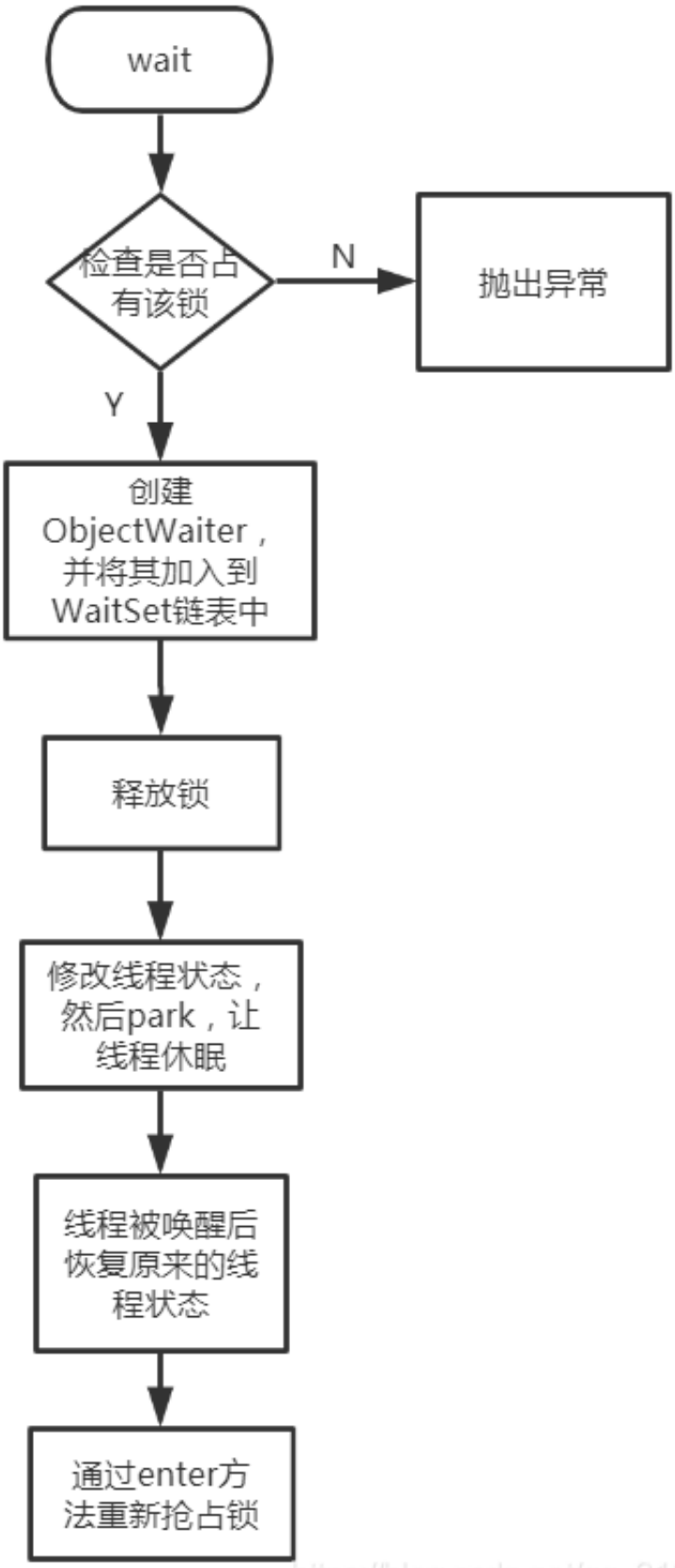


owner属性表示占用当前重量级锁的线程，如果owner属性就是当前线程说明是嵌套加锁，需要将记录嵌套加锁的属性加1，如果owner属性位于当前线程的调用栈帧中，说明该重量级锁是由轻量级锁膨胀而来，原来占用轻量级锁的线程依然占用该锁，此处再调用enter则说明是第一次嵌套加锁，将记录嵌套加锁的属性置为1，同时将owner属性置为当前线程。

自旋抢占时自旋的次数可以固定，也可以自适应的动态调整，默认是后者；自旋的过程中会检查是否进入了安全点同步，如果是则退出自旋。所谓的park是利用底层操作系统的mutex相关API，让当前线程休眠，一旦占有锁的线程释放了锁就会唤醒休眠的线程，然后尝试占用该锁，占有失败则尝试自旋抢占，还是失败则继续park。

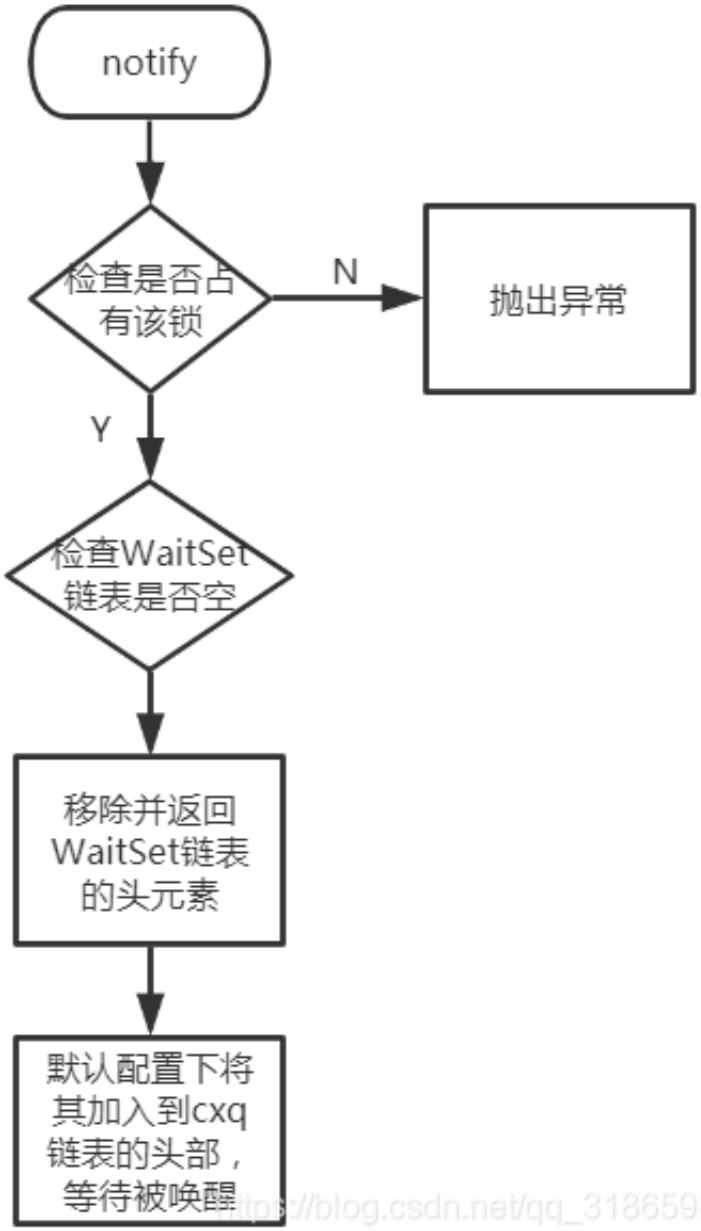
在修改线程状态为阻塞和恢复线程原来的运行状态时都要检查是否进入安全点同步，如果是则会阻塞当前线程完成状态切换，即强制当前线程进入安全点等待。

wait流程：



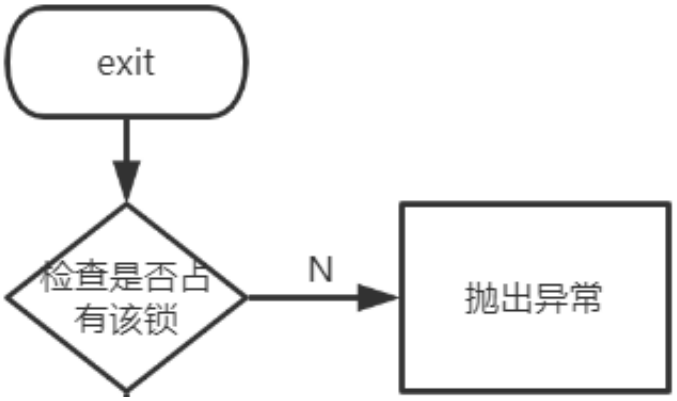
注意线程被唤醒有几种原因，wait方法设置的等待时间超时了，当前线程通过Thread.interrupt()方法被中断了，持有锁的线程释放锁时通过unpark唤醒当前线程。线程只是被唤醒了，不再处于休眠状态，然后一样调用enter方法抢占锁。

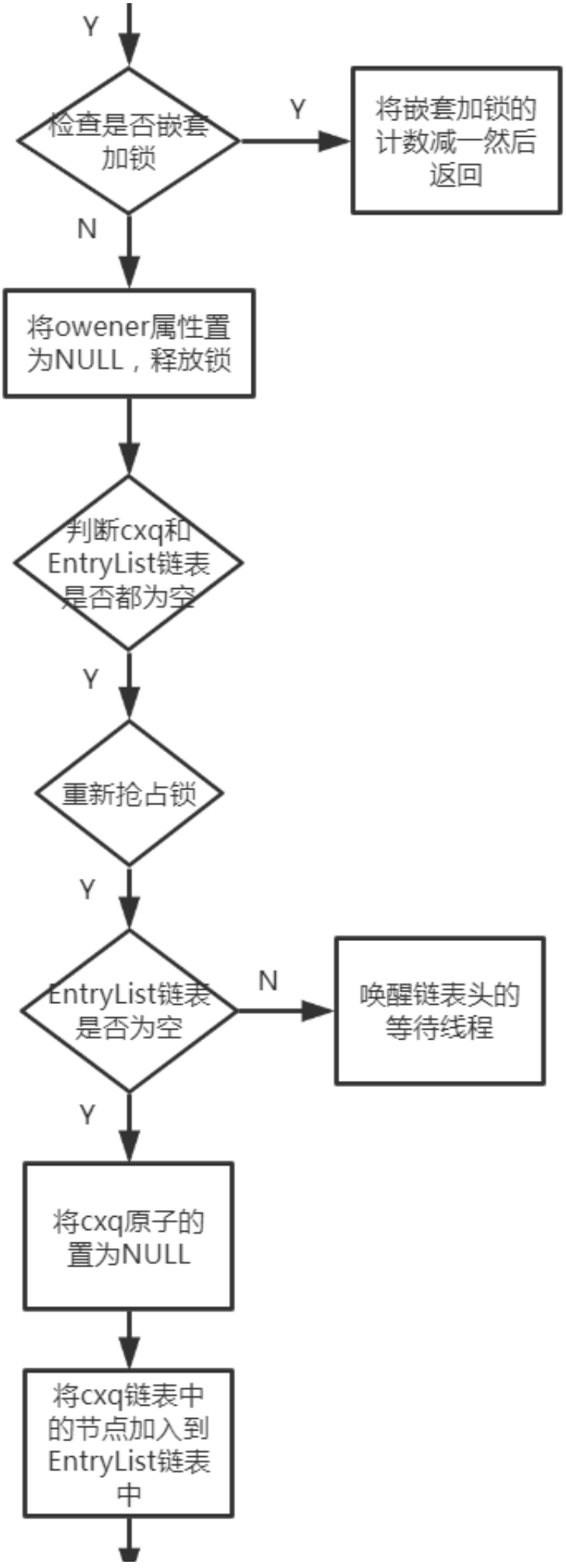
notify流程：



调用enter方法获取锁时，如果自旋抢占锁失败，同样会创建一个ObjectWaiter并不将其添加到cxq链表中然后park，等待锁释放时将其唤醒，因此上述加入cxq链表的动作相当于调用了一次enter方法，比起直接unpark可以提高抢占锁的效率。另外WaitSet链表的头元素是最早加入到链表中的线程，因此notify优先“唤醒”的是最早调用wait方法的线程。notifyAll和notify的核心流程是一样的，就多了一个for循环会以同样的方式处理所有等待的线程。

解锁流程：

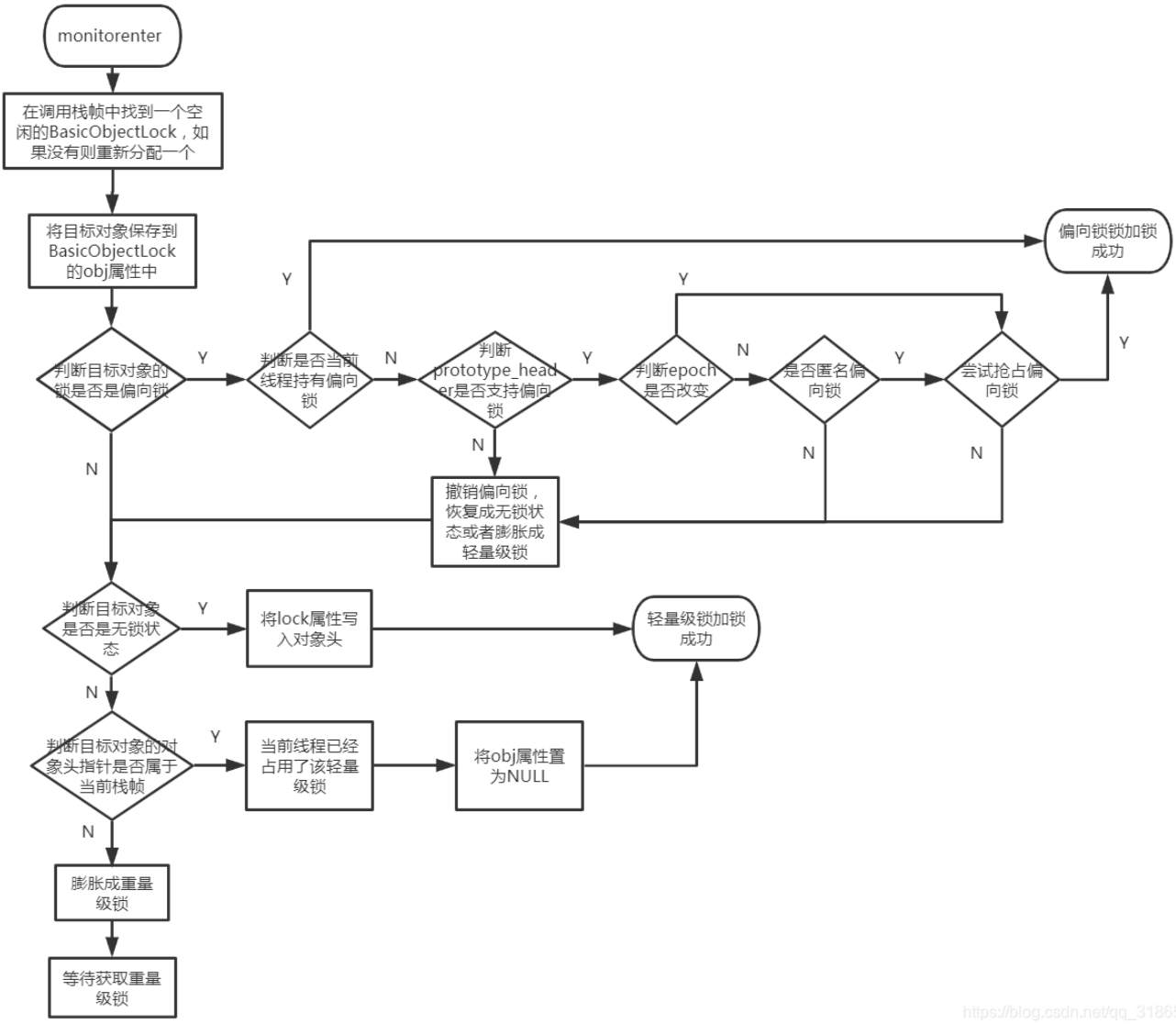



Y
检查是否嵌套加锁
Y
将嵌套加锁的计数减一然后返回
N
将owener属性置为NULL，释放锁
判断cxq和EntryList链表是否都为空
Y
重新抢占锁
Y
EntryList链表是否为空
N
唤醒链表头的等待线程
Y
将cxq原子的置为NULL
将cxq链表中的节点加入到EntryList链表中

先释放锁再判断链表是否为空可保证正在自旋抢占锁的线程相对于处于休眠状态的等待线程优先获取锁，EntryList初始化时就是空的，因此对于从cxq链表中转移到EntryList链表中的元素，优先处理cxq链表头，即最近才等待的线程优先获取锁。

六、整体加锁流程

加锁流程如下：



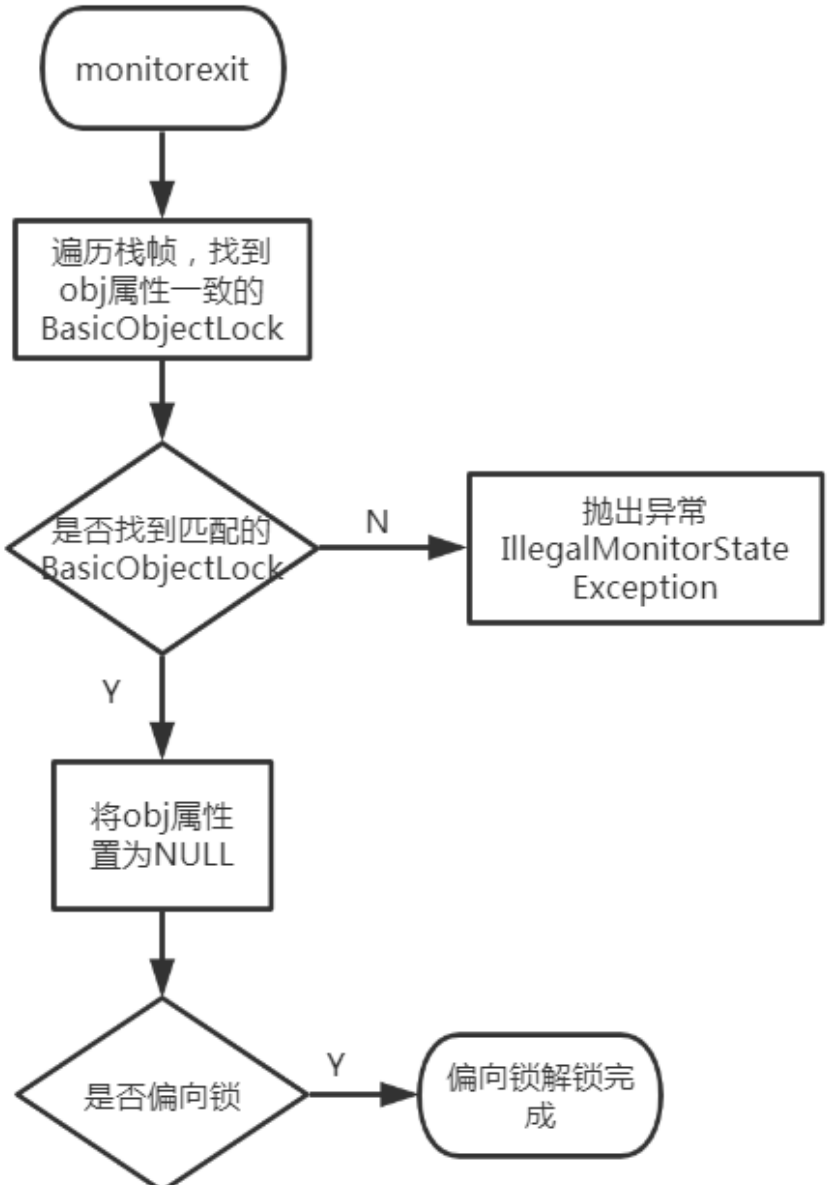
补充说明如下：

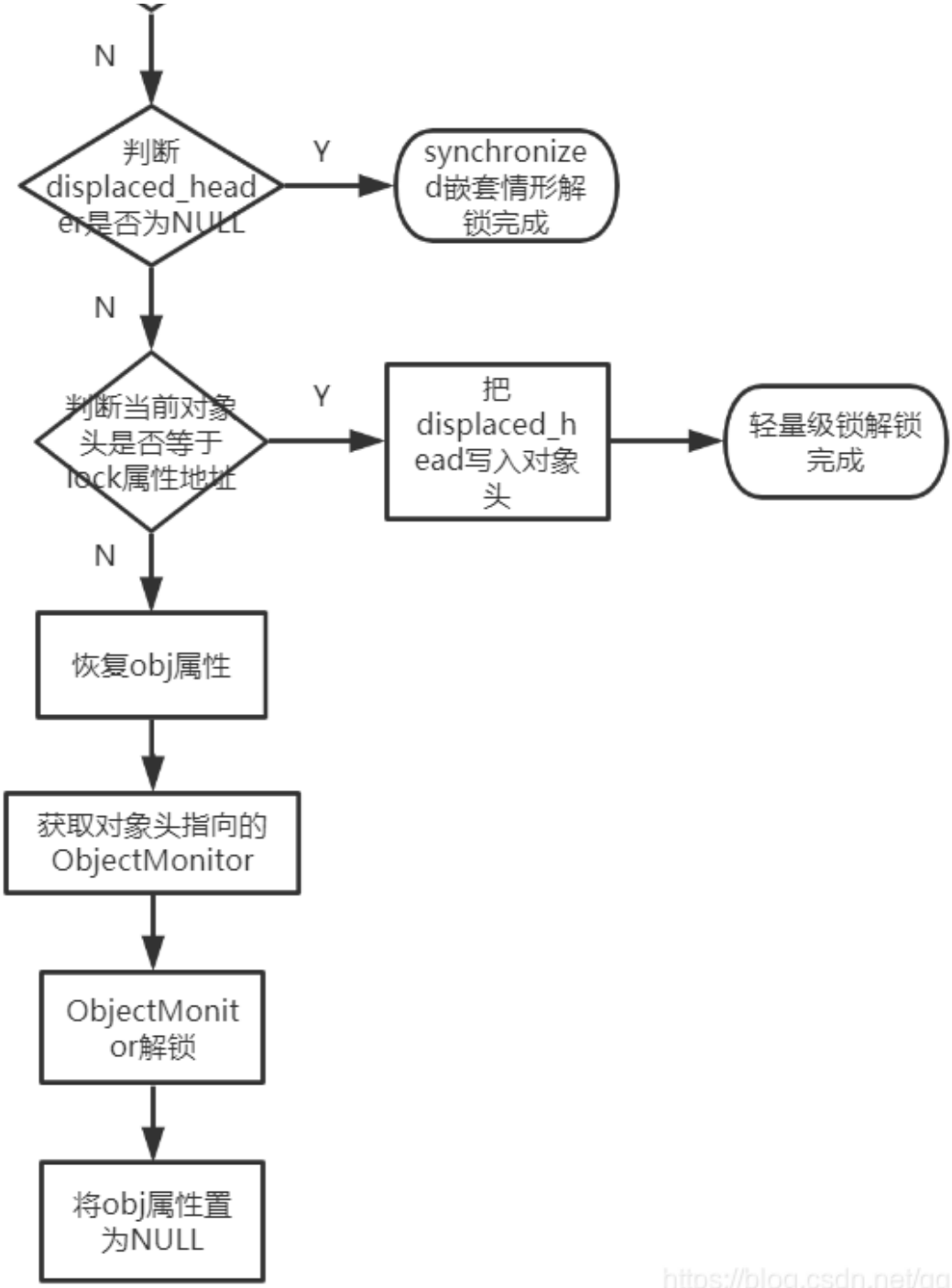
1. 注意如果当前调用栈帧中存在多个空闲的BasicObjectLock，则取地址最高的一个；如果当前调用栈帧中已经存在obj属性跟目标对象一样的BasicObjectLock，则取该BasicObjectLock后面的空闲BasicObjectLock，如果没有则重新分配一个，这样做是为了在synchronized嵌套使用的情形下解锁时能够很快的找到对应的BasicObjectLock。

2. prototype_header只在安全点下通过bulk_revoke_or_rebias_at_safepoint方法更改，发生更改时会将所有线程的所有栈帧中包含属于同一Klass的对象锁oop的对象头做相同的变更，如果某个对象锁是偏向锁但是与prototype_header不一致，则说明当时不在栈帧中，即没有实际被占用，因此可以直接抢占该偏向锁。
3. 一个支持偏向锁的Klass对应的实例在创建时是匿名偏向锁状态，即对象头中的线程指针为NULL，一旦该偏向锁被某个线程获取成功后就会将线程指针写入对象头，且在解锁的时候也不会清除里面的线程指针。再次获取该偏向锁时需要判断对象头中包含的线程指针对应的线程是否存在，如果存在且依然存在关联的BasicObjectLock，说明该线程依然占用该偏向锁，需要将该偏向锁膨胀成轻量级锁，否则需要将偏向锁撤销恢复成无锁状态，然后获取轻量级锁。所有抢占偏向锁失败的线程都会通过该逻辑膨胀成轻量级锁。
4. 将lock属性写入对象头是一个原子动作，通过lock指令前缀实现，即如果存在多个线程同时抢占轻量级锁，只能有一个线程抢占成功，其他线程执行时发现对象头已经不是无锁状态了，只能将轻量级锁膨胀成重量级锁，等待获取重量级锁。
5. 如果不是当前线程持有的偏向锁，且prototype_header未发生改变，则可能是其他线程占用了偏向锁或者该对象的偏向锁未被占用（对象头中包含的线程指针为NULL，又称匿名偏向锁），如果是后者则尝试抢占该偏向锁，如果抢占失败则撤销偏向锁触发锁膨胀

七、整体解锁流程

主要流程如下：





补充说明如下：

1. 偏向锁解锁不需要恢复对象头，也不需要去掉对象头中包含的线程指针，但是轻量级锁和重量级锁解锁时需要将目标对象的对象头恢复成无锁状态
2. 嵌套情形加锁时displaced_header就是NULL，因为外层的synchronized对应的BasicObjectLock已经保存了对象的原始对象头了，内层synchronized对应的BasicObjectLock不需要再保存了，此时解锁不需要恢复对象的原始对象头，外层synchronized解锁时会恢复原始对象头