- # 17 张图看穿 synchronized 关键字

小林coding *2020-12-16*

引子

小艾和小牛在路上相遇，小艾一脸沮丧。

小牛：小艾小艾，发生甚么事了？

小艾：别提了，昨天有个面试官问了我好几个关于 `synchronized` 关键字的问题，没答上来。

小艾：我后来查了很多资料，有二十多页的概念说明，也有三十来页的源码剖析，看得我头大。

小牛：你那看的是死知识，不好用，你得听我的总结。

小艾：看来是有备而来，那您给讲讲吧。

小牛：那咱们开始！

---

synchronized关键字引入

我们知道，在多线程程序中往往会出现这么一个情况：多个线程同时访问某个线程间的**共享变量**。来举个例子吧：

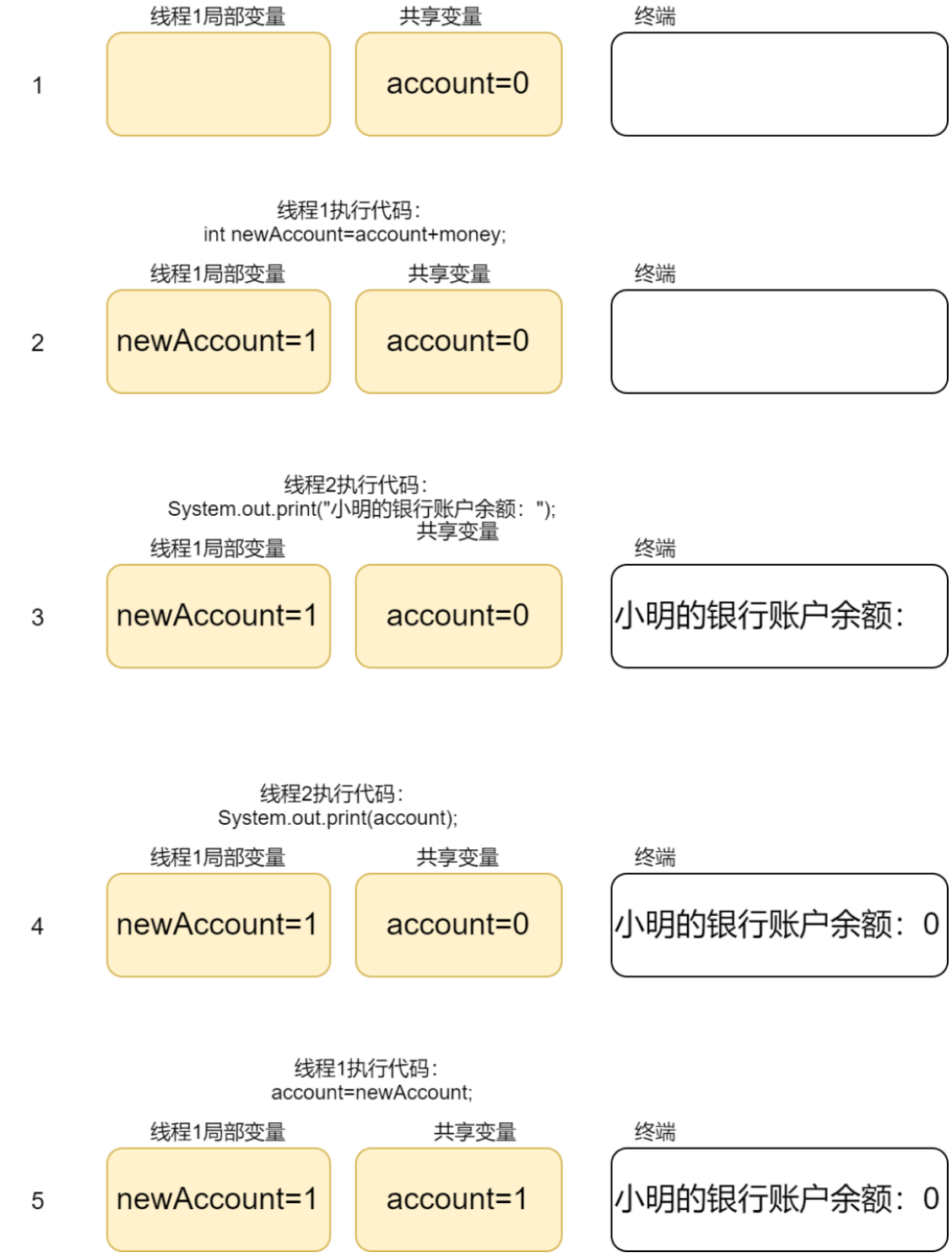
假设银行存款业务写了两个方法，一个是存钱 `store()` 方法，一个是查询余额 `get()` 方法。假设初始客户小明的账户余额为 0 元。（PS：这个例子只是个 `toy demo`，为了方便大家理解写的，真实的业务场景不会这样。）

```
    // account 客户在银行的存款
    public void store(int money){
        int newAccount=account+money;
        account=newAccount;
    }
    public void get(){
        System.out.print("小明的银行账户余额：");
        System.out.print(account);
    }
```

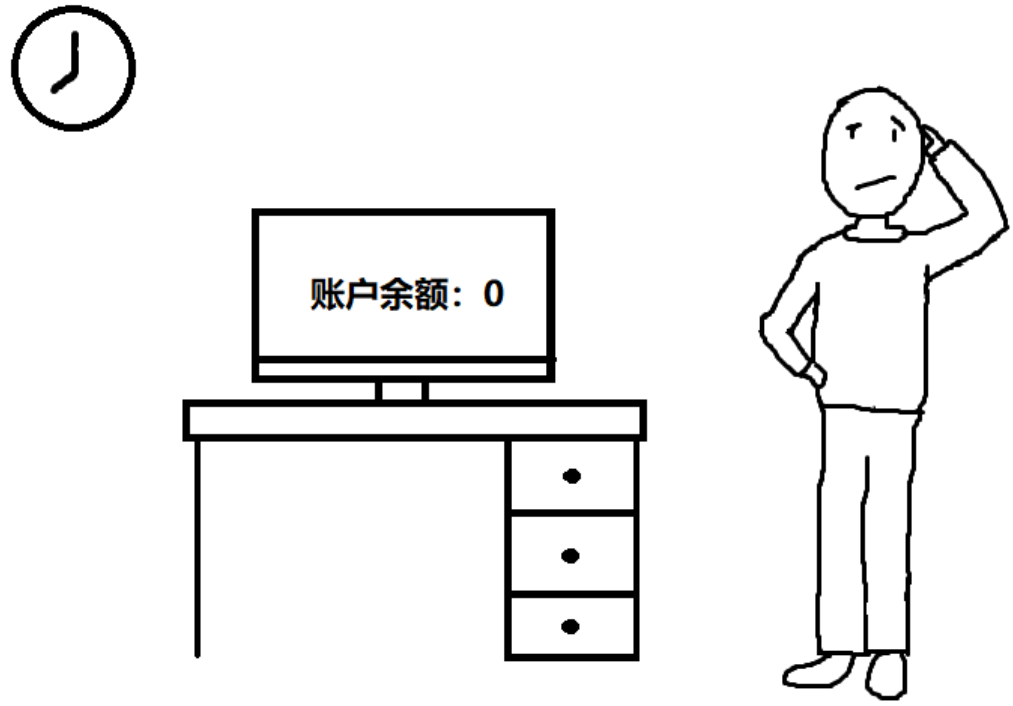
如果小明为自己存款 1 元，我们期望的线程调用情况如下：

> 1. 首先会启动一个线程调用 `store()` 方法，为客户账户余额增加 1；
> 2. 再启动一个线程调用 `get()` 方法，输出客户的新余额为 1。

但实际情况可能由于线程执行的先后顺序，出现如图所示的错误：


线程1局部变量
共享变量
终端
1
account=0
线程1执行代码：
int newAccount=account+money;
线程1局部变量
共享变量
终端
2
newAccount=1
account=0
线程2执行代码：
System.out.print("小明的银行账户余额：");
共享变量
线程1局部变量
终端
3
newAccount=1
account=0
小明的银行账户余额：
线程2执行代码：
System.out.print(account);
线程1局部变量
共享变量
终端
4
newAccount=1
account=0
小明的银行账户余额：0
线程1执行代码：
account=newAccount;
线程1局部变量
共享变量
终端
5
newAccount=1
account=1
小明的银行账户余额：0

小明存钱流程



小明：咱家没钱了

小明会惊奇的以为自己的钱没存上。这就是一个典型的由共享数据引发的**并发数据冲突问题**。

解决方式也很简单，让并发执行会产生问题的代码段不并发行了。

如果 `store()` 方法 执行完，才能执行 `get()` 方法，而不是像上图一样并发执行，自然不会出现这个问题。那如何才能做到呢?

答案就是使用 `synchronized` 关键字。

我们先从直觉上思考一下，如果要实现先执行 `store()` 方法，再执行 `get()` 方法的话该怎么设计。

我们可以设置某个锁，锁会有两种状态，分别是**上锁**和**解锁**。在 `store()` 方法执行之前，先观察这个锁的状态，如果是上锁状态，就进入阻塞，代码不运行；

如果这把锁是解锁状态，那就先将这把锁状态变为上锁，之后接着运行自己的代码。运行完成之后再将锁状态设置为解锁。

对于 `get()` 方法也是如此。

Java 中的 `synchronized` 关键字就是基于这种思想设计的。在 `synchronized` 关键字中，锁就是一个对象。

`synchronized` 一共有三种使用方法：

- **直接修饰某个实例方法**。像上文代码一样，在这种情况下多线程并发访问实例方法时，如果其他线程调用同一个对象的被 `synchronized` 修饰的方法，就会被阻塞。相当于把锁记录在这个方法对应的对象上。

```
    // account 客户在银行的存款
    public synchronized void store(int money){
        int newAccount=account+money;
        account=newAccount;
    }
    public synchronized void get(){
        System.out.print("小明的银行账户余额：");
        System.out.print(account);
    }
```

- **直接修饰某个静态方法。**在这种情况下进行多线程并发访问时，如果其他线程也是调用属于同一类的被 `synchronized` 修饰的静态方法，就会被阻塞。相当于把锁信息记录在这个方法对应的类上。

```
public synchronized static void get(){
    ...
}
```

- **修饰代码块。**如果此时有别的线程也想访问某个被`synchronized(对象0)`修饰的同步代码块时，也会被阻塞。

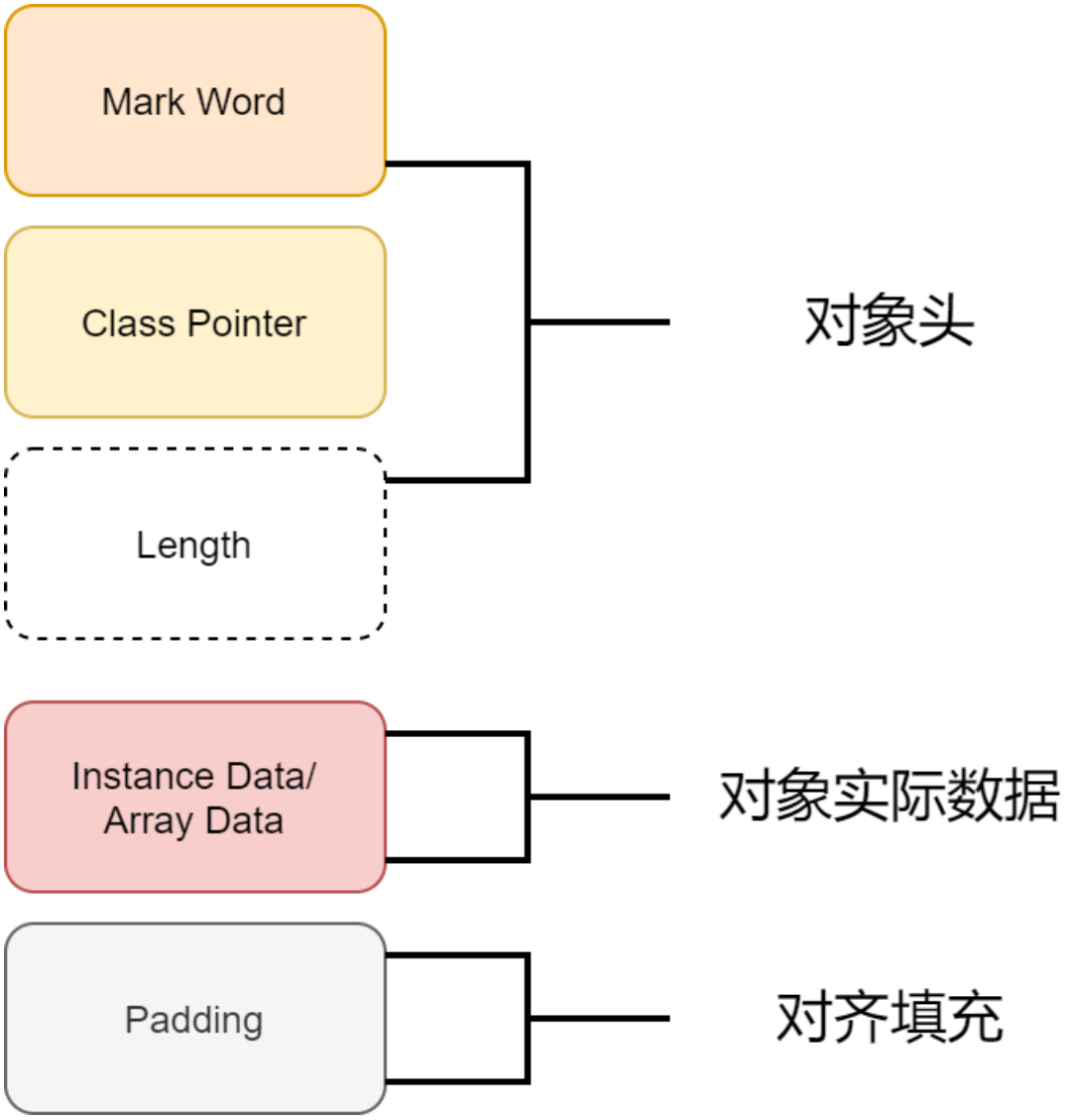
```
public static void get(){
    synchronized(对象0){
        ...
    }
}
```

小艾问：我看了不少参考书还有网上资料，都说 `synchronized` 的锁是锁在对象上的。关于这句话，你能深入讲讲吗？

小牛回答道：别急，我先讲讲 Java 对象在内存中的表示。

Java 对象在内存中的表示

讲清 `synchronized` 关键字的原理前需要理清 Java 对象在内存中的表示方法。



Java 对象在内存中的表示

上图就是一个 Java 对象在内存中的表示。我们可以看到，内存中的对象一般由三部分组成，分别是对象头、对象实际数据和对齐填充。

> 对象头包含 Mark Word、Class Pointer和 Length 三部分。
>
> - Mark Word 记录了对象关于锁的信息，垃圾回收信息等。
> - Class Pointer 用于指向对象对应的 Class 对象（其对应的元数据对象）的内存地址。
> - Length只适用于对象是数组时，它保存了该数组的长度信息。
>
> 对象实际数据包括了对象的所有成员变量，其大小由各个成员变量的大小决定。
>
> 对齐填充表示最后一部分的填充字节位，这部分不包含有用信息。

我们刚才讲的锁 `synchronized` 锁使用的就是对象头的 Mark Word 字段中的一部分。

Mark Word 中的某些字段发生变化，就可以代表锁不同的状态。

由于锁的信息是记录在对象里的，有的开发者也往往会说锁住对象这种表述。

无锁状态的 Mark Word

这里我们以无锁状态的 Mark Word 字段举例：

如果当前对象是无锁状态，对象的 Mark Word 如图所示。

锁状态	25bit	4bit	是否是偏向锁 1bit	锁标志位 2bit
无锁	对象的hashCode	对象分代年龄	0	01

无锁状态的 Mark Word 字段

我们可以看到，该对象头的 Mark Word 字段分为四个部分：

> 1. 对象的 hashCode ；
> 2. 对象的分代年龄，这部分用于对对象的垃圾回收；
> 3. 是否为偏向锁位，1代表是，0代表不是；
> 4. 锁标志位，这里是 01。

---

synchronized关键字的实现原理

讲完了 Java 对象在内存中的表示，我们下一步来讲讲 `synchronized` 关键字的实现原理。

从前文中我们可以看到， `synchronized` 关键字有两种修饰方法

1. **直接作为关键字修饰在方法上**，将整个方法作为同步代码块：

```
public synchronized static void `get()`{
    ...
}
```

1. **修饰在同步代码块上。**

```
public static void `get()`{
    synchronized(对象0){
        ...
    }
}
```

针对这两种情况，Java 编译时的处理方法并不相同。

对于第一种情况，编译器会为其自动生成了一个 `ACC_SYNCHRONIZED` 关键字用来标识。

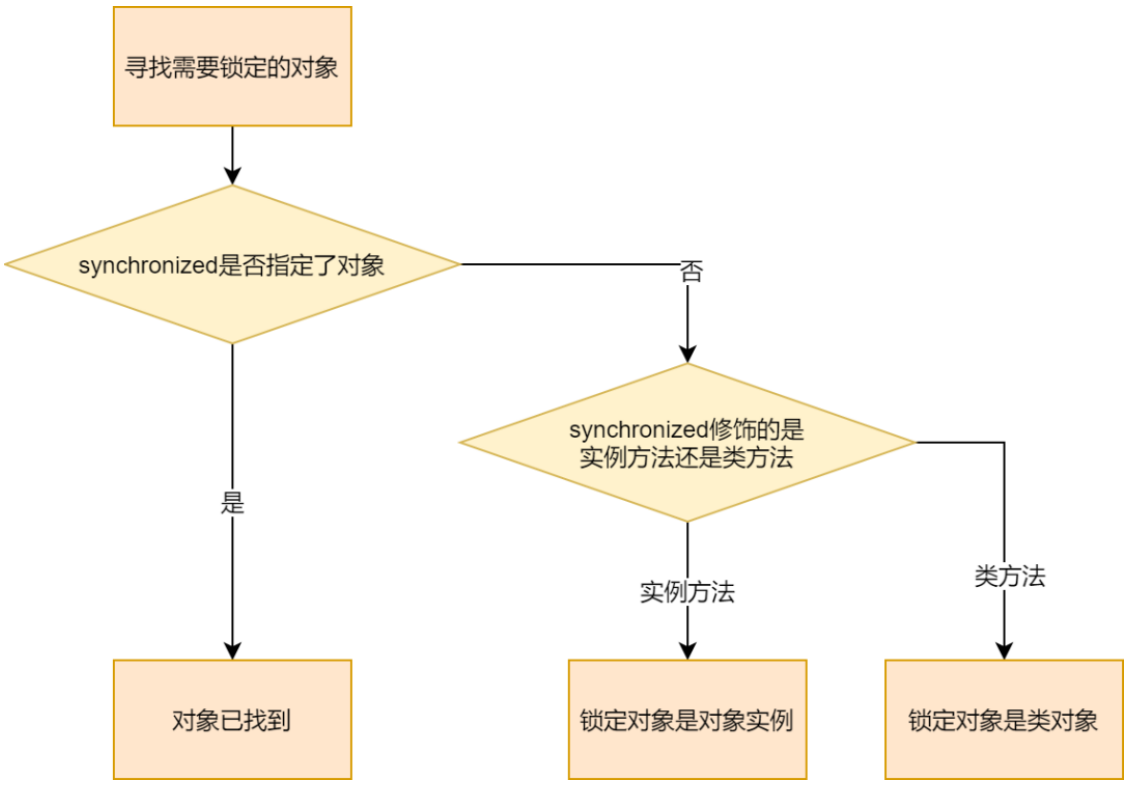
在 JVM 进行方法调用时，当发现调用的方法被 `ACC_SYNCHRONIZED` 修饰，则会先尝试获得锁。

对于第二种情况，编译时在代码块开始前生成对应的1个 `monitorenter` 指令，代表同步块进入。2个 `monitorexit` 指令，代表同步块退出。

这两种方法底层都需要一个 reference 类型的参数，指明要锁定和解锁的对象。

如果 `synchronized` 明确指定了对象参数，那就是该对象。

如果没有明确指定,那就根据修饰的方法是实例方法还是类方法，取对应的对象实例或类对象（Java 中类也是一种特殊的对象）作为锁对象。



确定锁定和解锁的对象

每个对象维护着一个记录着被锁次数的**计数器**。当一个线程执行 `monitorenter`，该计数器自增从 0 变为 1；

当一个线程执行 `monitorexit`，计数器再自减。当计数器为 0 的时候，说明对象的锁已经释放。

小艾问：为什么会有两个 `monitorexit` 指令呢？

小牛答：正常退出，得用一个 `monitorexit` 吧，如果中间出现异常，锁会一直无法释放。所以编译器会为同步代码块添加了一个隐式的 `try-finally` 异常处理，在 `finally` 中会调用 `monitorexit` 命令最终释放锁。

重量级锁

小艾问：那么问题来了，之前你说锁的信息是记录在对象的 Mark Word 中的，那现在冒出来的 `monitor` 又是什么呢？

小牛答：我们先来看一下**重量级锁**对应对象的 Mark Word。

在 Java 的早期版本中，`synchronized` 锁属于重量级锁，此时对象的 Mark Word 如图所示。

锁状态	30bit	锁标志位 2bit
重量级锁	指向重量级锁的指针	00

重量级锁的 Mark Word 字段

我们可以看到，该对象头的 Mark Word 分为两个部分。第一部分是指向重量级锁的指针，第二部分是锁标记位。

而这里所说的指向重量级锁的指针就是 `monitor`。

英文词典翻译 `monitor` 是监视器。Java 中每个对象会对应一个监视器。

这个监视器其实也就是监控锁有没有释放，释放的话会通知下一个等待锁的线程去获取。

`monitor` 的成员变量比较多，我们可以这样理解：

monitor结构

我们可以将 `monitor` 简单理解成两部分，第一部分表示**当前占用锁的线程**，第二部分是**等待这把锁的线程队列。**

如果当前占用锁的线程把锁释放了，那就需要在线程队列中唤醒下一个等待锁的线程。

但是阻塞或唤醒一个线程需要依赖底层的操作系统来实现，Java 的线程是映射到操作系统的原生线程之上的。

而操作系统实现线程之间的切换需要从用户态转换到核心态，这个状态转换需要花费很多的处理器时间，甚至可能比用户代码执行的时间还要长。

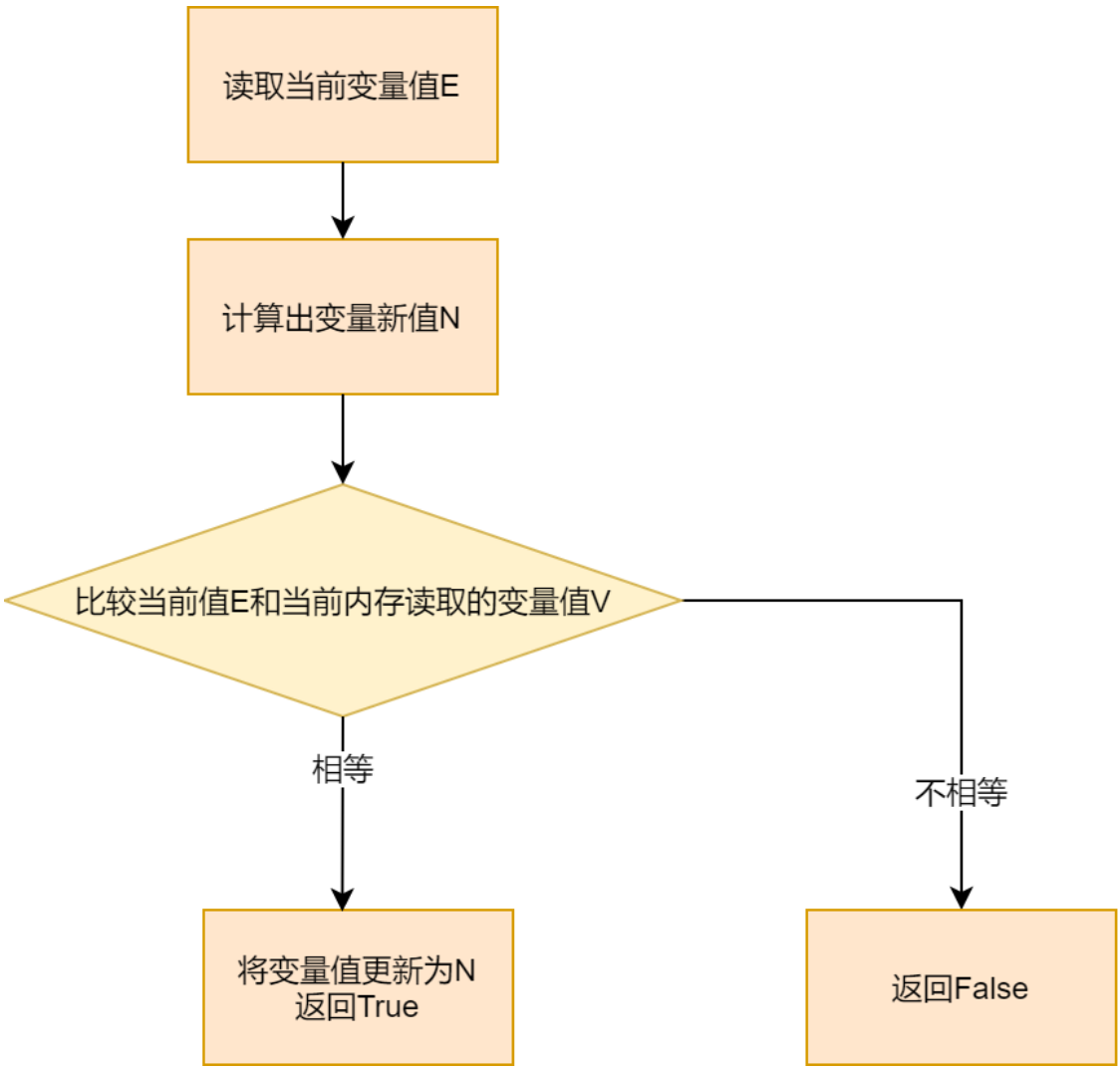
由于这种效率太低，Java 后期做了改进，我再来详细讲一讲。

CAS算法

在讲其他改进之前，我们先来聊聊 CAS 算法。CAS 算法全称为 Compare And Swap。

顾名思义，该算法涉及到了两个操作，**比较**（Compare）和**交换**（Swap）。

怎么理解这个操作呢？我们来看下图：



CAS 算法

我们知道，在对共享变量进行多线程操作的时候，难免会出现线程安全问题。

对该问题的一种解决策略就是对该变量加锁，保证该变量在某个时间段只能被一个线程操作。

但是这种方式的系统开销比较大。因此开发人员提出了一种新的算法，就是大名鼎鼎的 CAS 算法。

CAS 算法的思路如下：

> 1. 该算法认为线程之间对变量的操作进行竞争的情况比较少。
> 2. 算法的核心是对当前读取变量值 `E` 和内存中的变量旧值 `V` 进行比较。
> 3. 如果相等，就代表其他线程没有对该变量进行修改，就将变量值更新为新值 `N`。
> 4. 如果不等，就认为在读取值 `E` 到比较阶段，有其他线程对变量进行过修改，不进行任何操作。

当线程运行 CAS 算法时，该运行过程是**原子操作**，原子操作的含义就是线程开始跑这个函数后，运行过程中不会被别的程序打断。

我们来看看实际上 Java 语言中如何使用这个 CAS 算法，这里我们以 `AtomicInteger` 类中的 `compareAndSwapInt()` 方法举例：

```
public final native boolean compareAndSwapInt
(Object var1, long var2, int var3, int var4)
```

可以看到，该函数原型接受四个参数:

> 1. 第一个参数是一个 `AtomicInteger` 对象。
> 2. 第二个参数是该 `AtomicInteger` 对象对应的成员变量在内存中的地址。

3. 第三个参数是上图中说的线程之前读取的值 `P`。
4. 第四个参数是上图中说的线程计算的新值 `V`。

偏向锁

JDK 1.6 中提出了**偏向锁**的概念。该锁提出的原因是，开发者发现多数情况下锁并不存在竞争，一把锁往往是由同一个线程获得的。

如果是这种情况，不断的加锁解锁是没有必要的。

那么能不能让 JVM 直接负责在这种情况下加解锁的事情，不让操作系统插手呢?

因此开发者设计了偏向锁。偏向锁在获取资源的时候，会在资源对象上记录该对象是否偏向该线程。

偏向锁并不会主动释放，这样每次偏向锁进入的时候都会判断该资源是否是偏向自己的，如果是偏向自己的则不需要进行额外的操作，直接可以进入同步操作。

下图表示偏向锁的 Mark Word结构:

锁状态	23bit	2bit	4bit	是否是偏向锁 1bit	锁标志位 2bit
偏向锁	线程ID	EPOCH	对象分代年龄	1	01

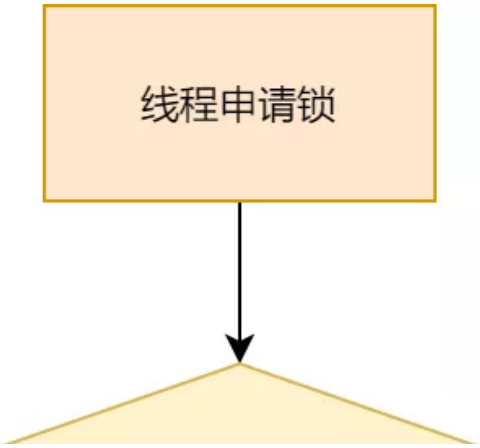
偏向锁的 Mark Word 字段

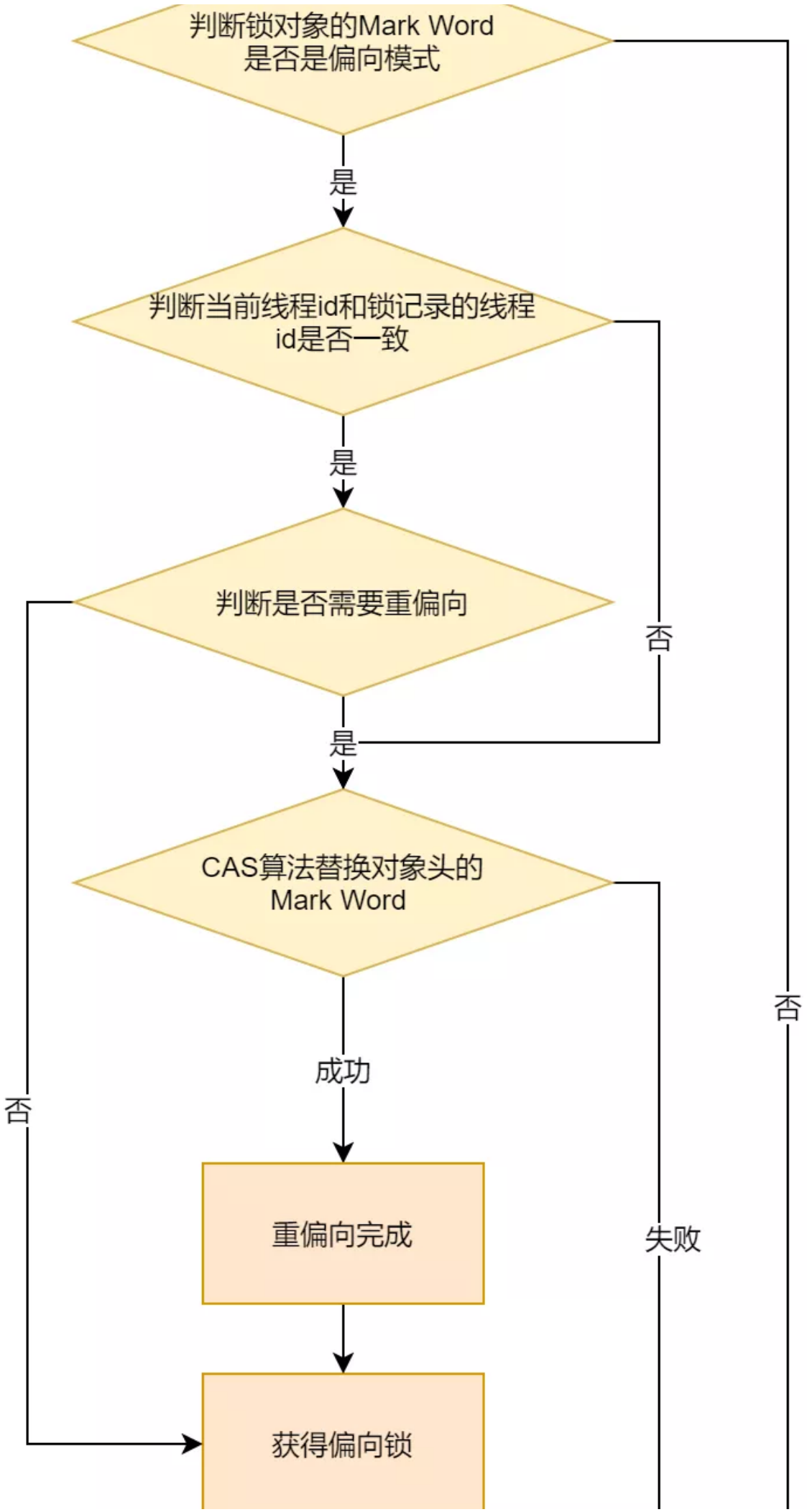
可以看到，偏向锁对应的 Mark Word 包含该偏向锁对应的线程 ID、偏向锁的时间戳和对象分代年龄。

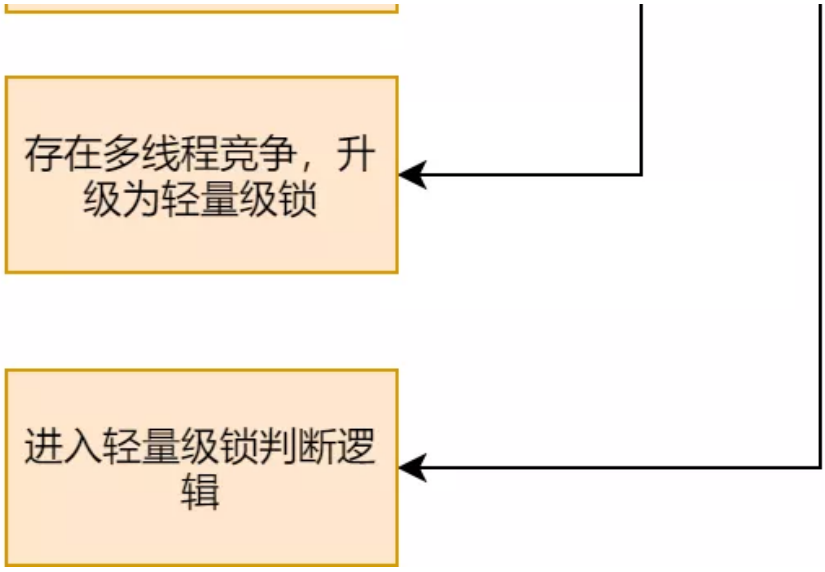
偏向锁的申请流程

我们再来看一下偏向锁的申请流程:

1. 首先需要判断对象的 Mark Word 是否属于偏向模式，如果不属于，那就进入轻量级锁判断逻辑。否则继续下一步判断;
2. 判断目前请求锁的线程 ID 是否和偏向锁本身记录的线程 ID 一致。如果一致，继续下一步的判断，如果不一致，跳转到步骤4;
3. 判断是否需要重偏向，重偏向逻辑在后面一节批量重偏向和批量撤销会说明。如果不用的话，直接获得偏向锁;
4. 利用 CAS 算法将对象的 Mark Word 进行更改，使线程 ID 部分换成本线程 ID。如果更换成功，则重偏向完成，获得偏向锁。如果失败，则说明有多线程竞争，升级为轻量级锁。




判断锁对象的Mark Word
是否是偏向模式
是
判断当前线程id和锁记录的线程
id是否一致
是
判断是否需要重偏向
否
是
CAS算法替换对象头的
Mark Word
否
成功
重偏向完成
失败
否
获得偏向锁



偏向锁的申请流程

值得注意的是，在执行完同步代码后，线程不会主动去修改对象的 Mark Word，让它重回无锁状态。

所以一般执行完 `synchronized` 语句后，如果是偏向锁的状态的话，线程对锁的释放操作可能是什么都不做。

匿名偏向锁

在 JVM 开启偏向锁模式下，如果一个对象被新建，在四秒后，该对象的对象头就会被置为偏向锁。

一般来说，当一个线程获取了一把偏向锁时，会在对象头和栈帧中的锁记录里不仅说明目前是偏向锁状态，也会存储锁偏向的线程 ID。

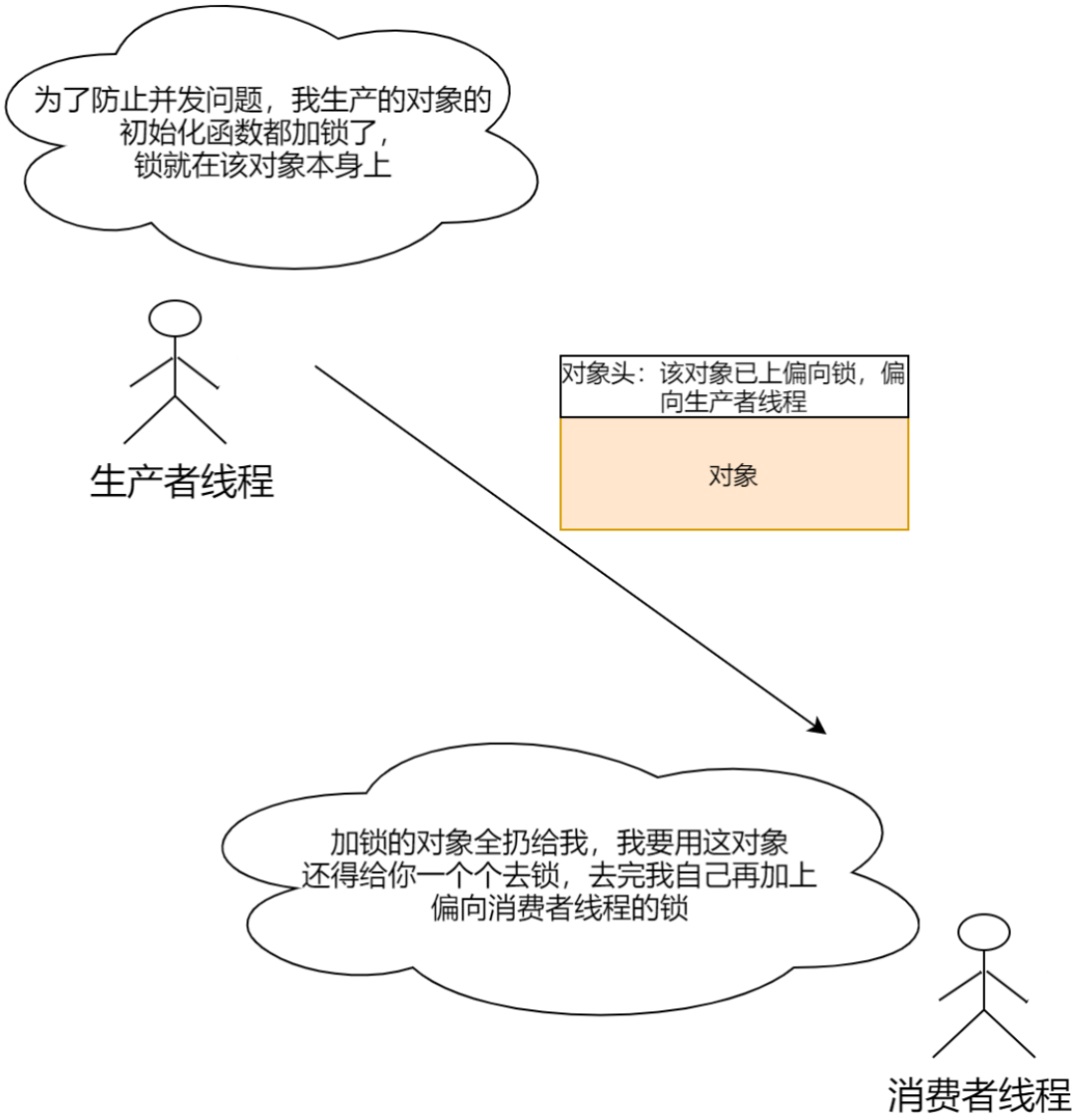
在 JVM 四秒自动创建偏向锁的情况下，线程 ID 为0。

由于这种情况下的偏向锁不是由某个线程求得生成的，这种情况下的偏向锁也称为匿名偏向锁。

批量重偏向和批量撤销

在**生产者消费者模式**下，生产者线程负责对象的创建，消费者线程负责对生产出来的对象进行使用。

当生产者线程创建了大量对象并执行加偏向锁的同步操作，消费者对对象使用之后，会产生大量偏向锁执行和偏向锁撤销的问题。



大量偏向锁执行和偏向锁撤销的问题

Russell K和 Detlefs D在他们的文章提出了批量重偏向和批量撤销的过程。

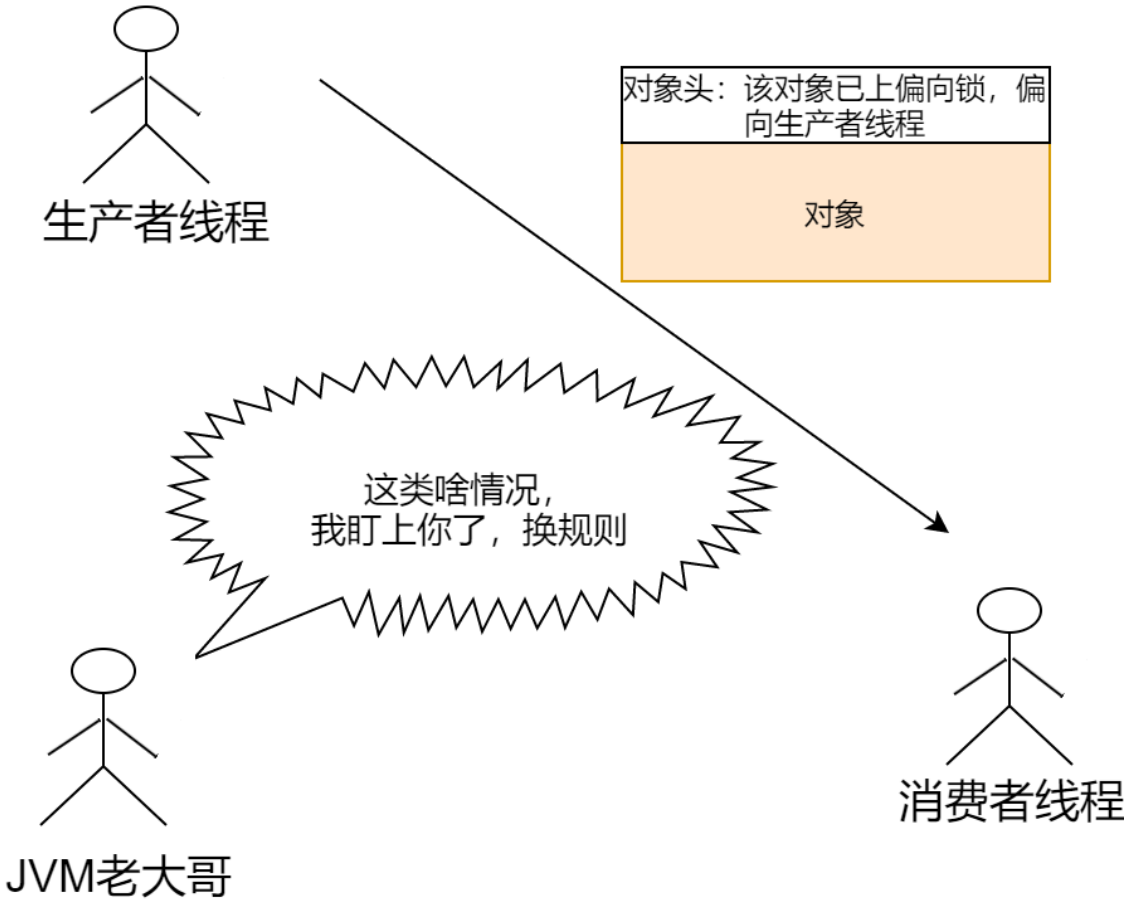
在上图情景下，他们探讨了能不能直接将偏向的线程换成消费者的线程。

替换不是一件容易事，需要在 JVM 的众多线程中找到类似上文情景的线程。

他们最后提出的解决方法是：

> 以类为单位，为每个类维护一个偏向锁撤销计数器，每一次该类的对象发生偏向撤销操作时，该计数器计数 +1，当这个计数值达到重偏向阈值时，JVM 就认为该类可能不适合正常逻辑，适合批量重偏向逻辑。这就是对应上图流程图里的是否需要重偏向过程。

以生产者消费者为例，生产者生产同一类型的对象给消费者，然后消费者对这些对象都需要执行偏向锁撤销，当撤销过程过多时就会触发上文规则，JVM 就注意到这个类了。



批量重偏向和批量撤销

具体规则是：

1. 每个类对象会有一个对应的 `epoch` 字段，每个处于偏向锁状态对象的 Mark Word 中也有该字段，其初始值为创建该对象时，类对象中的 `epoch` 的值。
2. 每次发生批量重偏向时，就将类对象的 `epoch` 字段 +1，得到新的值 `epoch_new`。
3. 遍历 JVM 中所有线程的栈，找到该类对象，将其 `epoch` 字段改为新值。根据线程栈的信息判断出该线程是否锁定了该对象，将现在偏向锁还在被使用的对象赋新值 `epoch_new`。
4. 下次有线程想获得锁时，如果发现当前对象的 `epoch` 值和类的 `epoch` 不相等，不会执行撤销操作，而是直接通过 CAS 操作将其 Mark Word 的 Thread ID 改成当前线程 ID。

批量撤销相对于批量重偏向好理解得多，JVM 也会统计重偏向的次数。

假设该类计数器计数继续增加，当其达到批量撤销的阈值后（默认40），JVM 就认为该类的使用场景存在多线程竞争，会标记该类为不可偏向，之后对于该类的锁升级为轻量级锁。

轻量级锁

轻量级锁的设计初衷在于并发程序开发者的经验“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”。

所以它的设计出发点也在线程竞争情况较少的情况下。我们先来看一下轻量级锁的 Mark Word 布局。

如果当前对象是轻量级锁状态，对象的 Mark Word 如下图所示。

锁状态	30bit	锁标志位 2bit
轻量级锁	指向栈中锁记录的指针	00

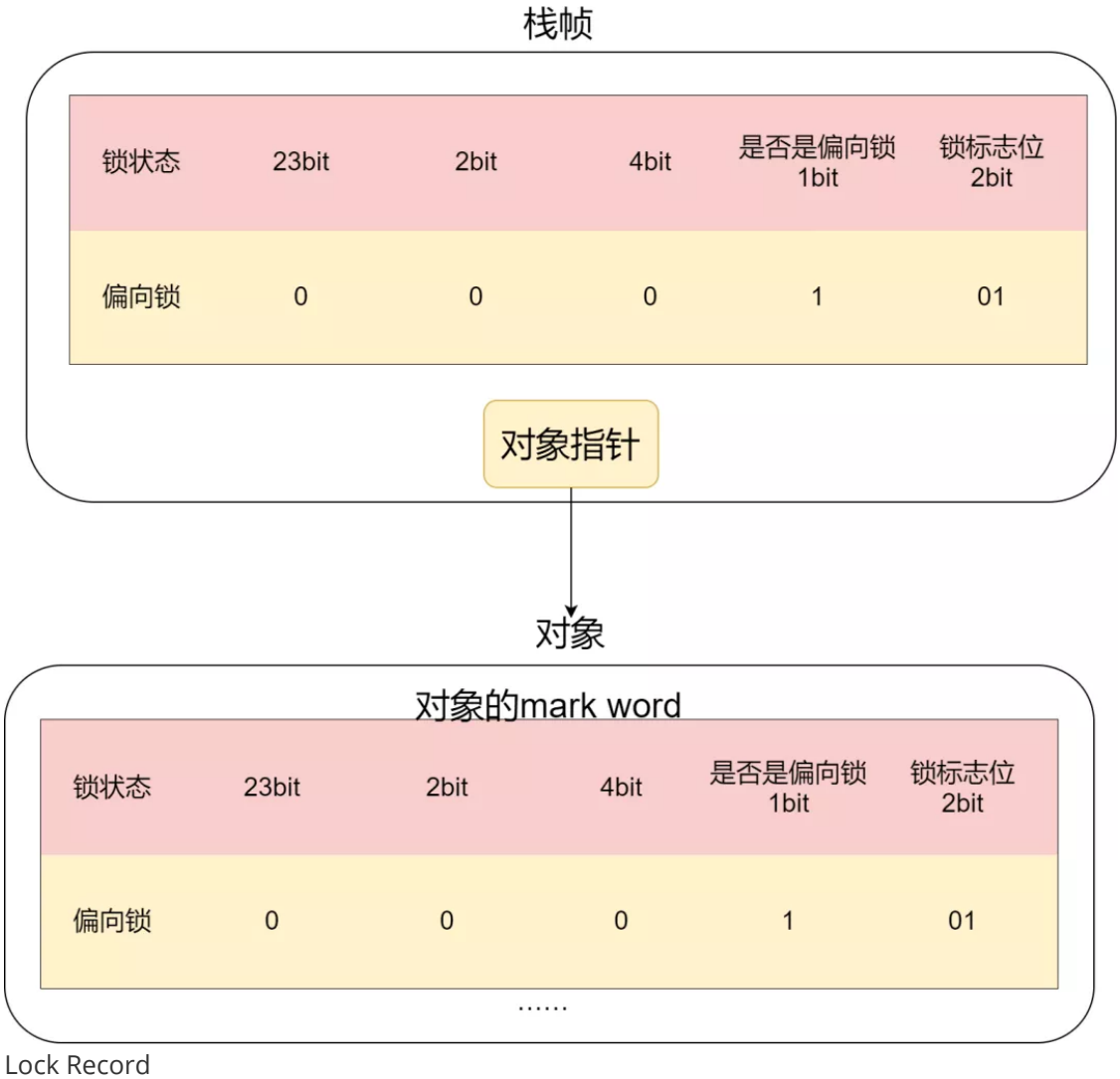
轻量级锁 Mark Word 字段

我们可以看到，该对象头Mark Word分为两个部分。第一部分是指向栈中的锁记录的指针，第二部分是锁标记位，针对轻量级锁该标记位为 00。

小艾问：那这指向栈中的锁记录的指针是什么意思呢?

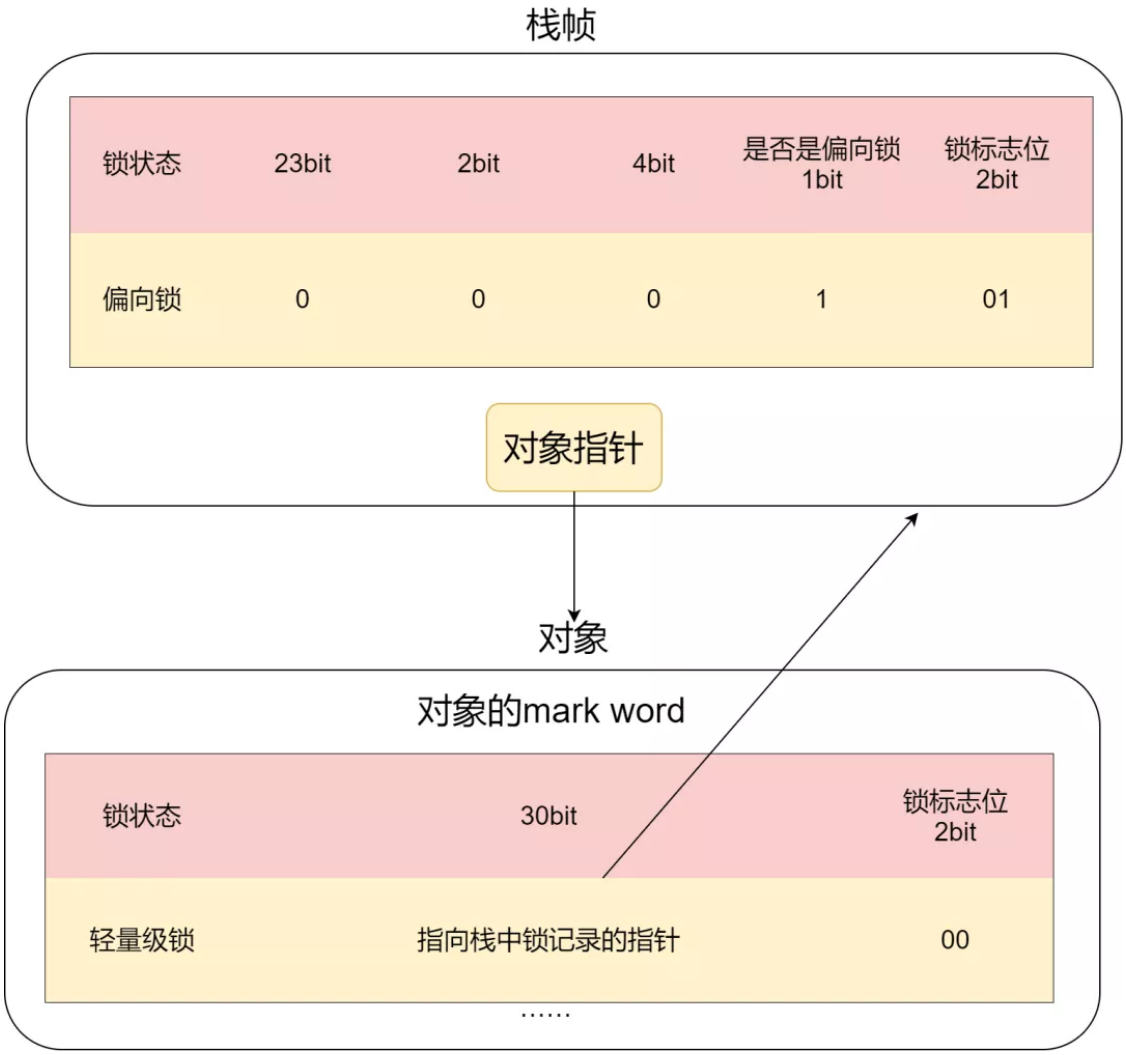
小牛答：这得结合轻量级锁的上锁步骤来慢慢讲。

如果当前这个对象的锁标志位为 01（即无锁状态或者轻量级锁状态），线程在执行同步块之前，JVM 会先在当前的线程的栈帧中创建一个 Lock Record，包括一个用于存储对象头中的 Mark Word 以及一个指向对象的指针。



Lock Record

然后 JVM 会利用 CAS 算法对这个对象的 Mark Word 进行修改。如果修改成功，那该线程就拥有了这个对象的锁。我们来看一下如果上图的线程执行 CAS 算法成功的结果。



执行 CAS 算法

当然 CAS 也会有失败的情况。如果 CAS 失败，那就说明同时执行 CAS 操作的线程可不止一个了，Mark Word 也做了更改。

首先虚拟机会检查对象的 Mark Word 字段指向栈中的锁记录的指针是否指向当前线程的栈帧。如果是，那就说明可能出现了类似 `synchronized` 中套 `synchronized` 情况：

```
synchronized (对象0) {
    synchronized (对象0) {
        ...
    }
}
```

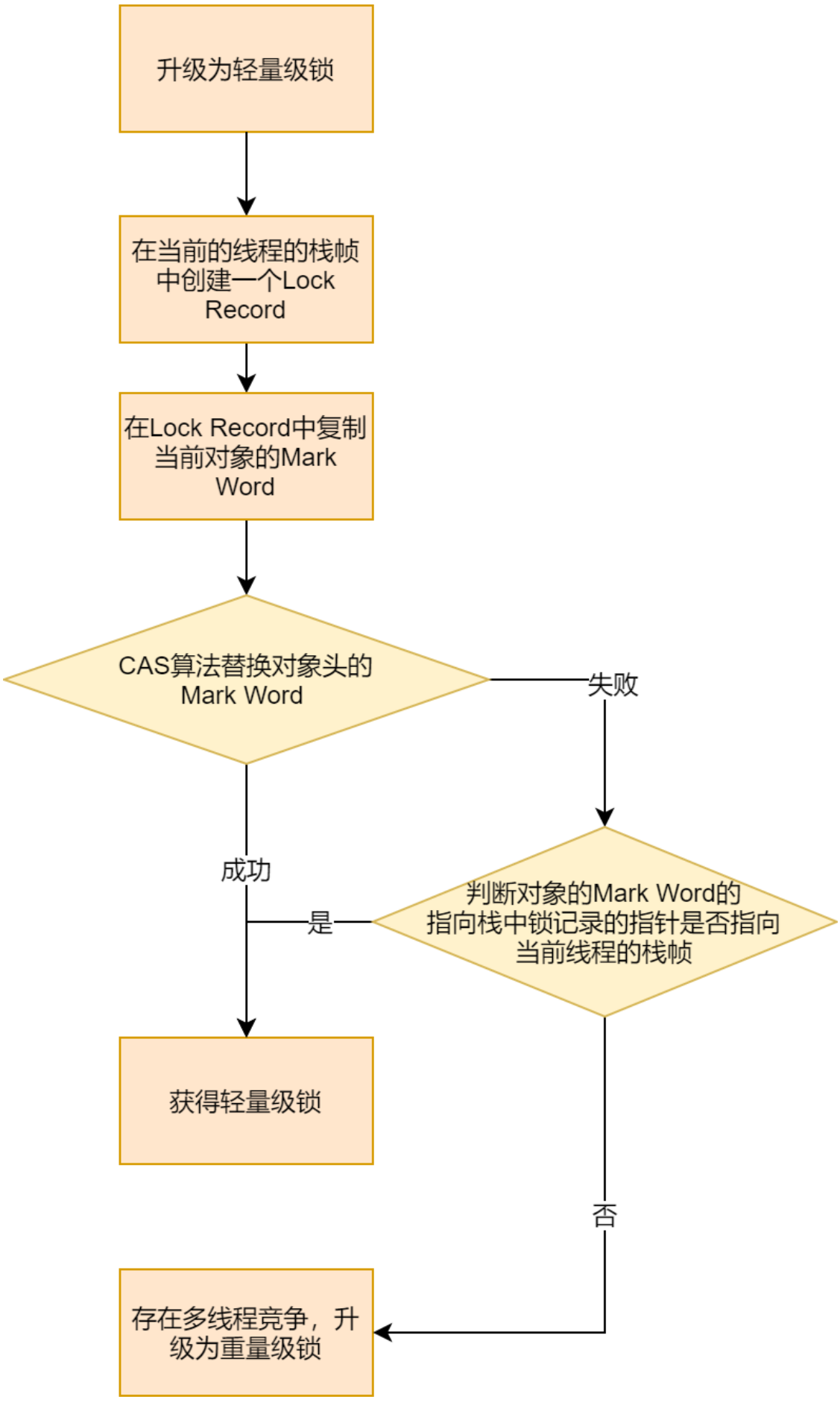
当然这种情况下当前线程已经拥有这个对象的锁，可以直接进入同步代码块执行。

否则说明锁被其他线程抢占了，该锁还需要升级为重量级锁。

和偏向锁不同的是，执行完同步代码块后，需要执行轻量级锁的**解锁**过程。解锁过程如下：

1. 通过 CAS 操作尝试把线程栈帧中复制的 Mark Word 对象替换当前对象的 Mark Word。
2. 如果 CAS 算法成功，整个同步过程就完成了。
3. 如果 CAS 算法失败，则说明存在竞争，锁升级为重量级锁。

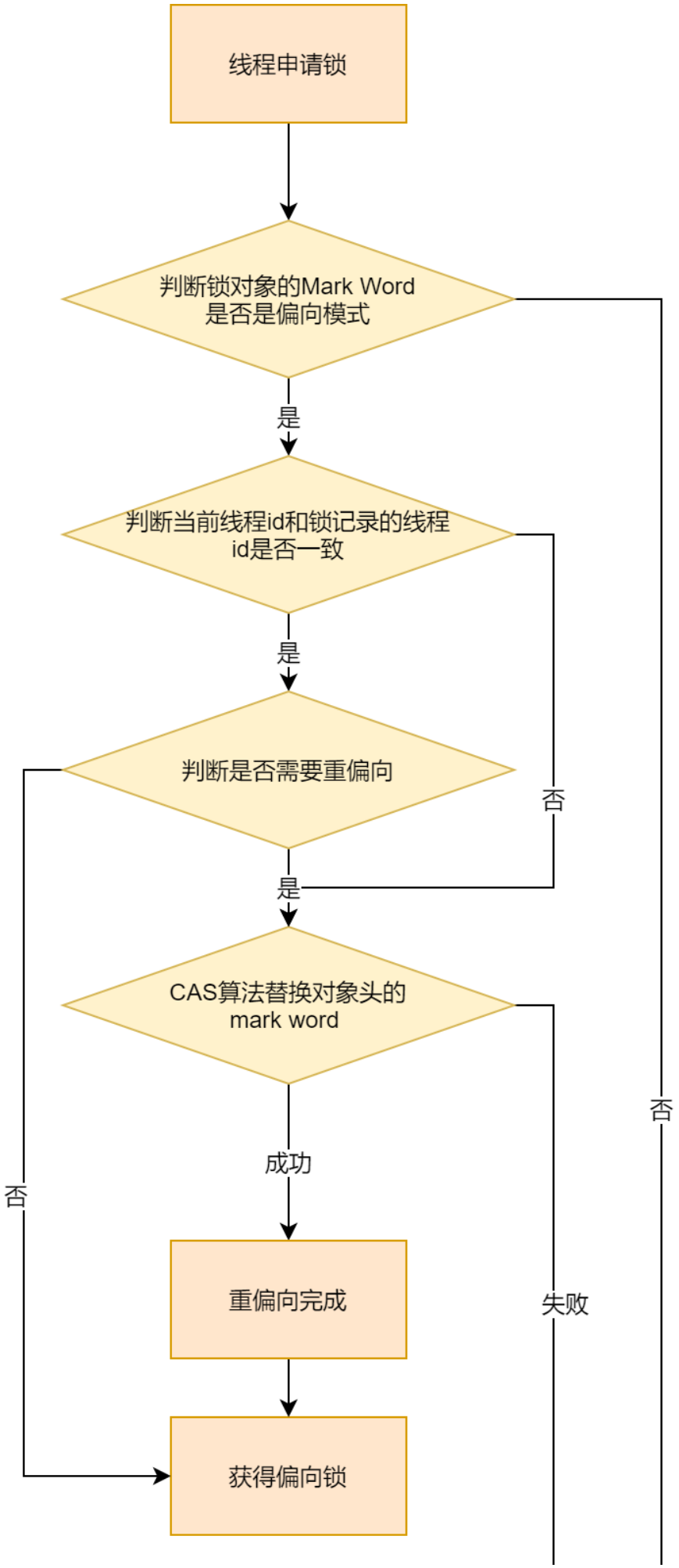
我们来总结一下轻量级锁升级过程吧：

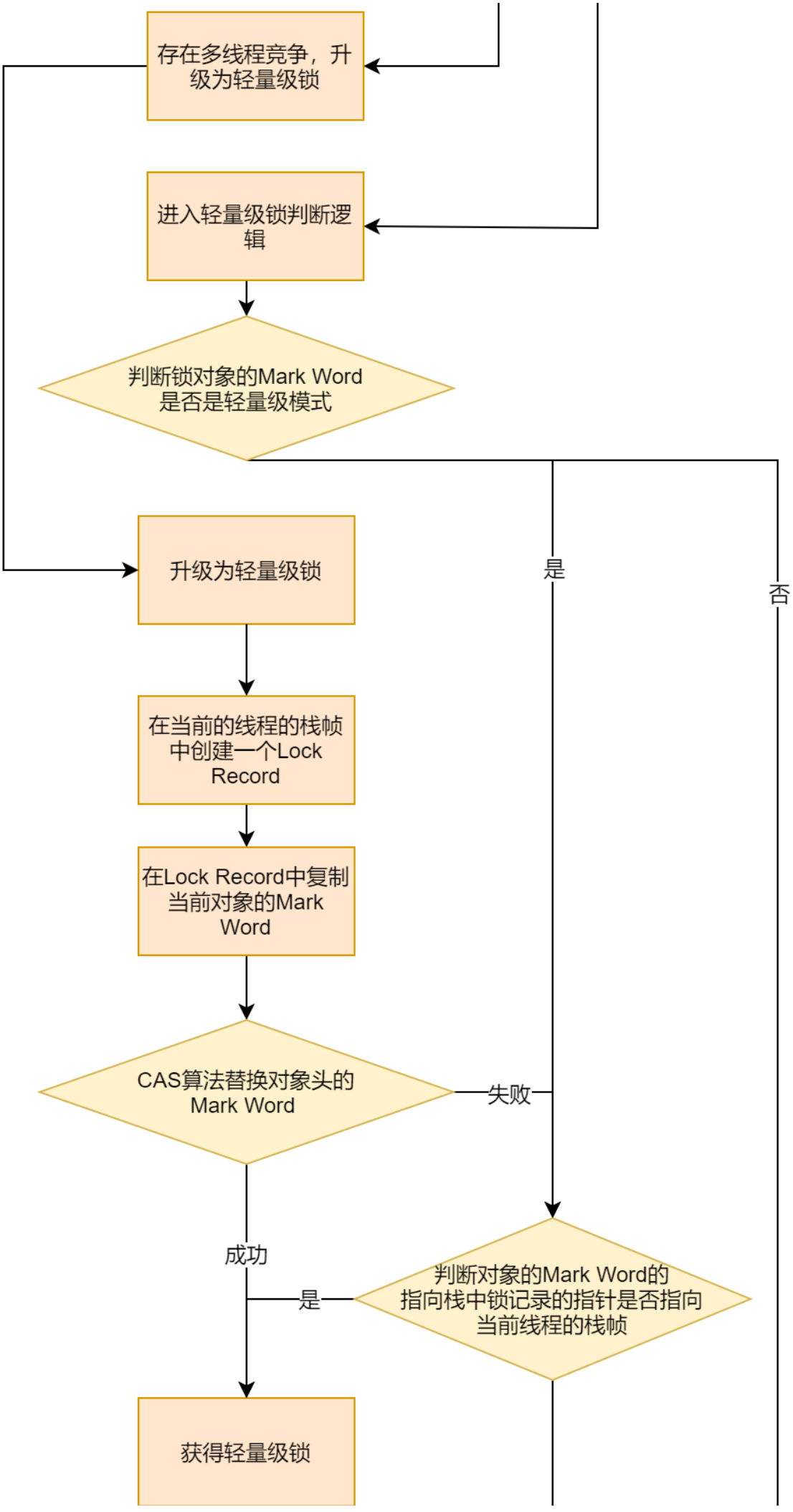


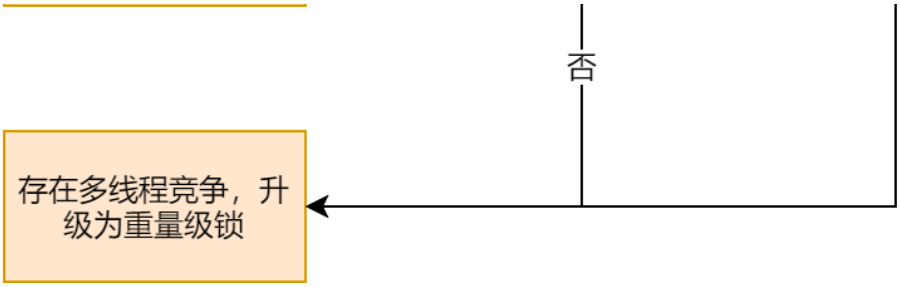
轻量级锁的升级过程

总结

这次我们了解了 `synchronized` 底层实现原理和对应的锁升级过程。最后我们再通过这张流程图来回顾一下 `synchronized` 锁升级过程吧。




存在多线程竞争，升级为轻量级锁
进入轻量级锁判断逻辑
判断锁对象的Mark Word是否是轻量级模式
是
否
升级为轻量级锁
在当前的线程的栈帧中创建一个Lock Record
在Lock Record中复制当前对象的Mark Word
CAS算法替换对象头的Mark Word
失败
成功
是
判断对象的Mark Word的指向栈中锁记录的指针是否指向当前线程的栈帧
获得轻量级锁



锁申请完整流程

巨人肩膀

1. 实现Java虚拟机：JVM故障诊断与性能优化
2. 深入理解java虚拟机 JVM高级特性与最佳实践
3. Russell K , Detlefs D . Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing[C]// Acm Sigplan Conference on Object-oriented Programming Systems. ACM, 2006.
4. Dice D , Moir M S , Scherer Iii W N . Quickly reacquirable locks: US 2010.
5. https://github.com/farmerjohngit/myblog/issues/12
6. https://www.itqiankun.com/article/bias-lightweight-synchronized-lock
7. https://www.itqiankun.com/article/bias-lock-epoch-effect
8. https://www.hollischuang.com/archives/1883
9. http://www.ideabuffer.cn/2017/05/06/Java%E5%AF%B9%E8%B1%A1%E5%86%85%E5%AD%98%E5%B8%83%E5%B1%80/
10. http://www.ideabuffer.cn/2017/04/21/java-%E4%B8%AD%E7%9A%84%E9%94%81-%E5%81%8F%E5%90%91%E9%94%81%E3%80%81%E8%BD%BB%E9%87%8F%E7%BA%A7%E9%94%81%E3%80%81%E8%87%AA%E6%97%8B%E9%94%81%E3%80%81%E9%87%8D%E9%87%8F%E7%BA%A7%E9%94%81/
11. https://blog.csdn.net/zhao_miao/article/details/84500771