主讲老师：图灵课堂Fox老师

> 有道笔记地址：https://note.youdao.com/s/8NNDZMst

==========synchronized基础篇==========

1. i++/i--引起的线程安全问题分析

思考： 两个线程对初始值为 0 的静态变量一个做自增，一个做自减，各做 5000 次，结果是 0 吗?

```java
public class SyncDemo {

    private static int counter = 0;

    public static void increment() {
        counter++;
    }

    public static void decrement() {
        counter--;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5000; i++) {
                increment();
            }
        }, "t1");
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5000; i++) {
                decrement();
            }
        }, "t2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        //思考： counter=?
        log.info("{}", counter);
    }
}
```

1.1 原因分析

以上的结果可能是正数、负数、零。为什么呢？因为 Java 中对静态变量的自增，自减并不是原子操作。
我们可以查看 i++和 i--（i 为静态变量）的 JVM 字节码指令 （ 可以在idea中安装一个jclasslib插件）

i++的JVM 字节码指令

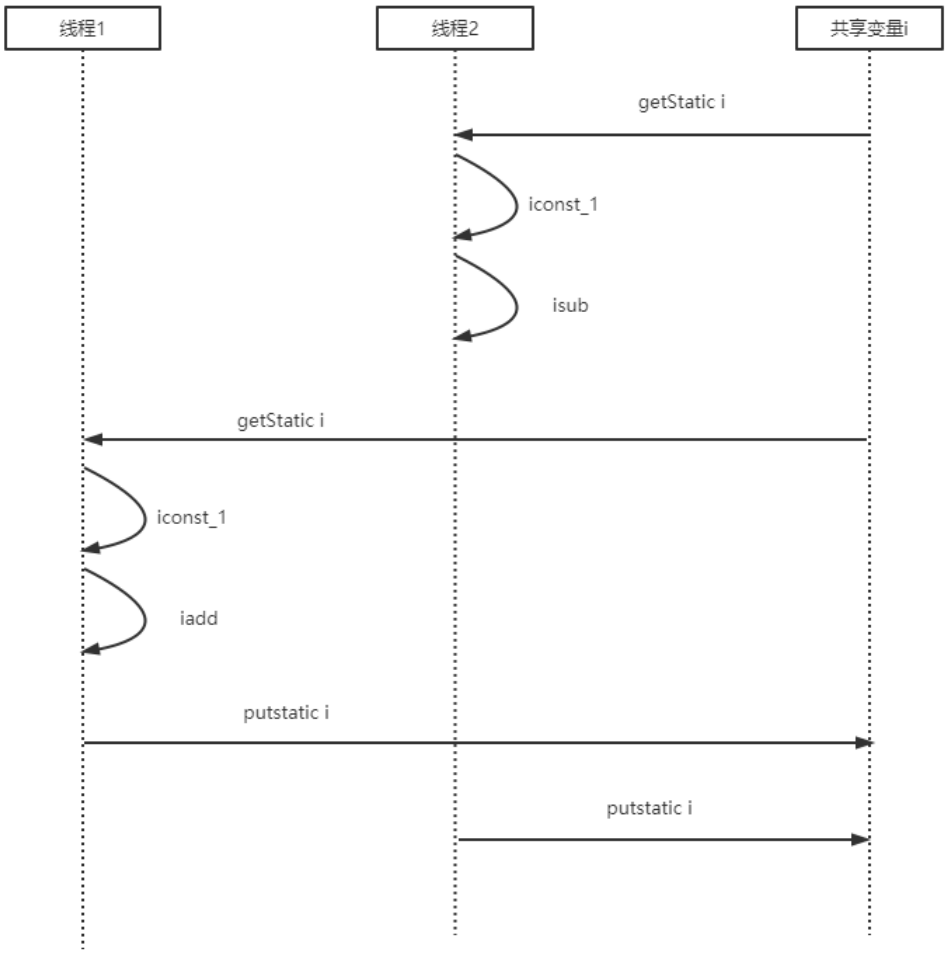
```
getstatic i // 获取静态变量i的值，并将其值压入栈顶
iconst_1 // 将int型常量1压入栈顶
iadd // 将栈顶两int型数值相加并将结果压入栈顶
putstatic i // 将结果赋值给静态变量i
```

> 参考jvm指令集 https://note.youdao.com/s/KQfVzrpY

i--的JVM 字节码指令

```
getstatic i // 获取静态变量i的值，并将其值压入栈顶
iconst_1 // 将int型常量1压入栈顶
isub // 将栈顶两int型数值相减并将结果压入栈顶
putstatic i // 将结果赋值给静态变量i
```

如果是单线程以上 8 行代码是顺序执行（不会交错）没有问题。
但多线程下这 8 行代码可能交错运行：



小结

- 一个程序运行多个线程本身是没有问题的
- 问题出在多个线程访问共享资源
    - 多个线程读共享资源其实也没有问题
    - 在多个线程对共享资源读写操作时发生指令交错，就会出现问题

1.2 解决方案

一段代码块内如果存在对共享资源的多线程读写操作，称这段代码块为临界区，其共享资源为临界资源。

多个线程在临界区内执行，由于代码的执行序列不同而导致结果无法预测，称之为发生了竞态条件。

```java
//临界资源
private static int counter = 0;

public static void increment() { //临界区
    counter++;
}

public static void decrement() {//临界区
    counter--;
}
```

为了避免临界区的竞态条件发生，有多种手段可以达到目的：

- 阻塞式的解决方案：synchronized，Lock
- 非阻塞式的解决方案：原子变量

> **注意：**
> 虽然 java 中互斥和同步都可以采用 synchronized 关键字来完成，但它们还是有区别的：
> 互斥是保证临界区的竞态条件发生，同一时刻只能有一个线程执行临界区代码
> 同步是由于线程执行的先后、顺序不同、需要一个线程等待其它线程运行到某个点

2. synchronized的使用

synchronized 同步块是 Java 提供的一种原子性内置锁，Java 中的每个对象都可以把它当作一个同步锁来使用，这些 Java 内置的使用者看不到的锁被称为内置锁，也叫作监视器锁。

2.1 加锁方式

2.2 使用synchronized解决之前的共享问题

方式一

```
public static synchronized void increment() {
    counter++;
}

public static synchronized void decrement() {
    counter--;
}
```

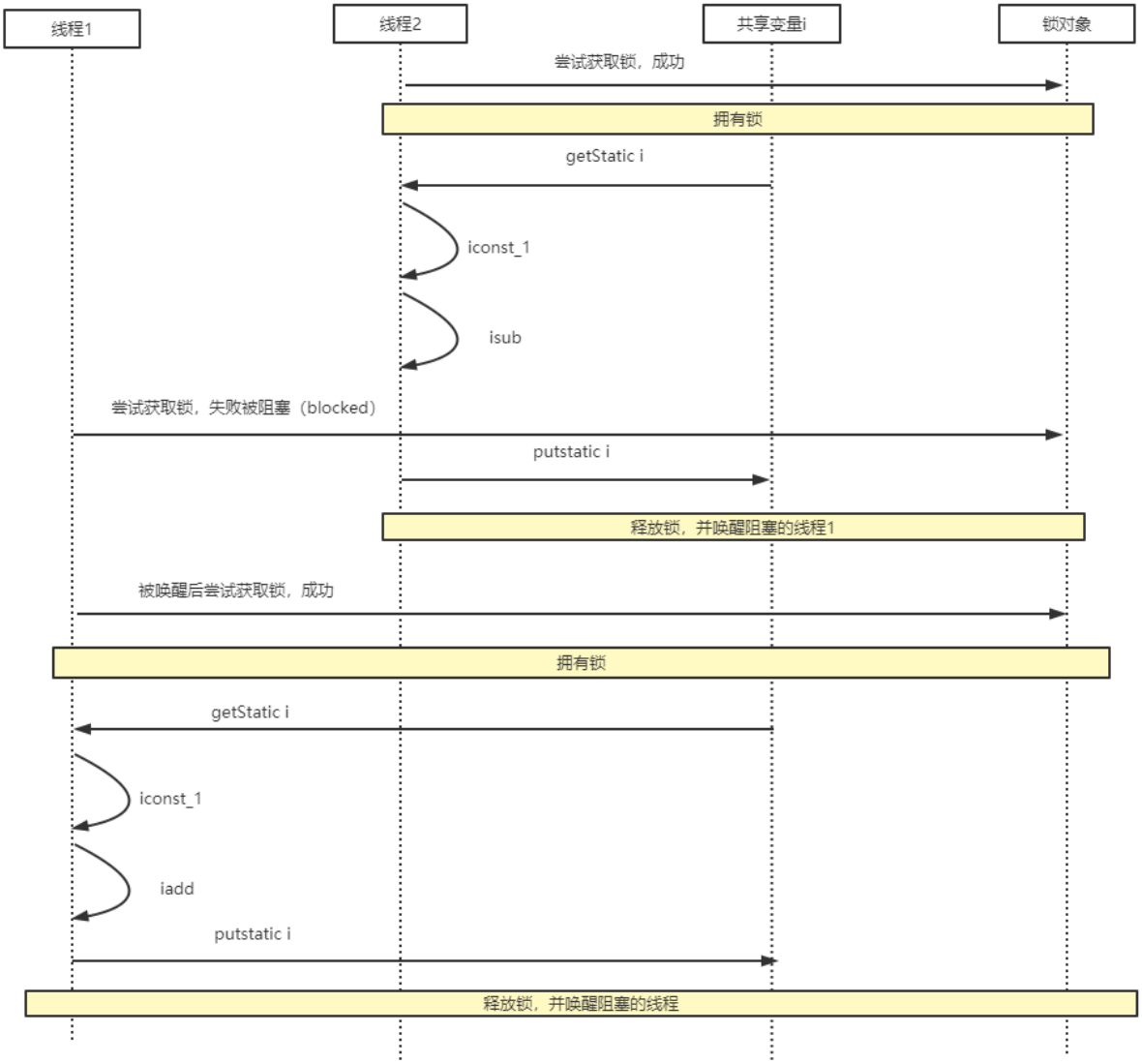
方式二

```
private static String lock = "";

public static void increment() {
    synchronized (lock){
        counter++;
    }
}

public static void decrement() {
    synchronized (lock) {
        counter--;
    }
}
```

synchronized 实际是用对象锁保证了临界区内代码的原子性



==========synchronized高级篇==========

3. synchronized底层实现原理分析

synchronized是JVM内置锁，基于**Monitor**机制实现，依赖底层操作系统的互斥原语Mutex（互斥量），它是一个重量级锁，性能较低。

3.1 查看synchronized的字节码指令序列

Method access and property flags：

同步方法是通过方法中的access_flags中设置ACC_SYNCHRONIZED标志来实现；同步代码块是通过monitorenter和monitorexit来实现。

3.2 重量级锁实现之Monitor（管程/监视器）机制详解

Monitor，直译为“监视器”，而操作系统领域一般翻译为“管程”。管程是指管理共享变量以及对共享变量操作的过程，让它们支持并发。在Java 1.5之前，Java语言提供的唯一并发语言就是管程，Java 1.5之后提供的SDK并发包也是以管程为基础的。除了Java之外，C/C++、C#等高级语言也都是支持管程的。synchronized关键字和wait()、notify()、notifyAll()这三个方法是Java中实现管程技术的组成部分。

> **The Java® Language Specification**
>
> Each object is associated with a monitor (§17.1), which is used by synchronized methods (§8.4.3) and the synchronized statement (§14.19) to provide control over concurrent access to state by multiple threads (§17 (Threads and Locks)).
>
> **The Java® Virtual Machine Specification**
>
> The Java Virtual Machine supports synchronization of both methods and sequences of instructions within a method by a single synchronization construct: the monitor.
>
> Java虚拟机通过一个同步结构支持方法和方法中的指令序列的同步：monitor。

Monitor设计思路

MESA模型分析

在管程的发展史上，先后出现过三种不同的管程模型，分别是Hasen模型、Hoare模型和MESA模型。现在正在广泛使用的是**MESA模型**。下面我们便介绍MESA模型：

管程中引入了条件变量的概念，而且每个条件变量都对应有一个等待队列。条件变量和等待队列的作用是解决线程之间的同步问题。

Java 参考了 MESA 模型，语言内置的管程（synchronized）对 MESA 模型进行了精简。MESA 模型中，条件变量可以有多个，Java 语言内置的管程里只有一个条件变量。模型如下图所示。

示例代码

```java
@Slf4j
public class WaitDemo {
    final static Object obj = new Object();

    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
            log.debug("t1开始执行....");
            synchronized (obj) {
                log.debug("t1获取锁....");
                try {
                    // 让线程在obj上一直等待下去
                    obj.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                log.debug("t1执行完成....");
            }
        },"t1").start();

        new Thread(() -> {
            log.debug("t2开始执行....");
            synchronized (obj) {
                log.debug("t2获取锁....");
                try {
                    // 让线程在obj上一直等待下去
                    obj.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                log.debug("t2执行完成....");
            }
        },"t2").start();

        // 主线程两秒后执行
        Thread.sleep(2000);
        log.debug("准备获取锁，去唤醒 obj 上阻塞的线程");
        synchronized (obj) {
            // 唤醒obj上一个线程
```

```
            //obj.notify();
            // 唤醒obj上所有等待线程
            obj.notifyAll();
            log.debug("唤醒 obj 上阻塞的线程");
        }

    }

}
```

ObjectMonitor数据结构分析

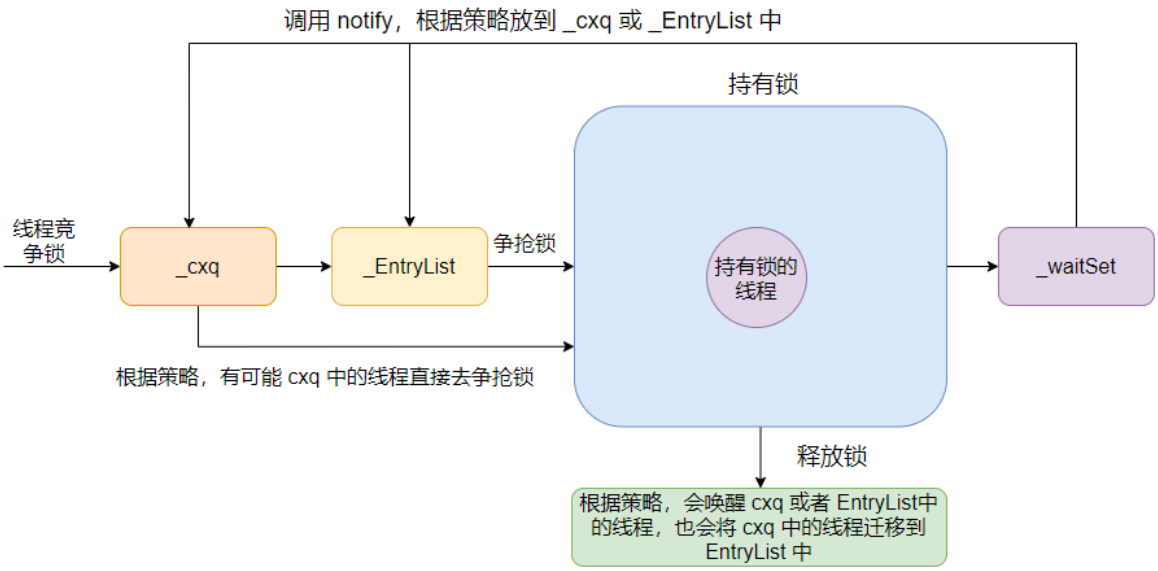
java.lang.Object 类定义了 wait()，notify()，notifyAll() 方法，这些方法的具体实现，依赖于 ObjectMonitor 实现，这是 JVM 内部基于 C++ 实现的一套机制。

ObjectMonitor其主要数据结构如下（hotspot源码ObjectMonitor.hpp）：

```
ObjectMonitor() {
    _header       = NULL; //对象头  markOop
    _count        = 0;
    _waiters      = 0,
    _recursions   = 0;   // 锁的重入次数
    _object       = NULL;  //存储锁对象
    _owner        = NULL;  // 标识拥有该monitor的线程（当前获取锁的线程）
    _WaitSet      = NULL;  // 等待线程（调用wait）组成的双向循环链表，_WaitSet是第一个节点
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ; //多线程竞争锁会先存到这个单向链表中 （FILO栈结构）
    FreeNext      = NULL ;
    _EntryList    = NULL ; //存放在进入或重新进入时被阻塞(blocked)的线程 (也是存竞争锁失败的线程)
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
  }
```

重量级锁实现原理

synchronized 底层是利用 monitor 对象，CAS 和 mutex 互斥锁来实现的，内部会有等待队列(cxq 和 EntryList)和条件等待队列(waitSet)来存放相应阻塞的线程。未竞争到锁的线程存储到等待队列中，获得锁的线程调用 wait 后便存放在条件等待队列中，解锁和 notify 都会唤醒相应队列中的等待线程来争抢锁。然后由于阻塞和唤醒依赖于底层的操作系统实现，系统调用存在用户态与内核态之间的切换，所以有较高的开销，因此称之为重量级锁。



> 在获取锁时，是将当前线程插入到cxq的头部，而释放锁时，默认策略（QMode=0）是：如果EntryList为空，则将cxq中的元素按原有顺序插入到EntryList，并唤醒第一个线程，也就是当EntryList为空时，是后来的线程先获取锁。_EntryList不为空，直接从_EntryList中唤醒线程。
>
> 为什么会有_cxq 和 _EntryList 两个列表来放线程?
>
> 因为会有多个线程会同时竞争锁，所以搞了个 _cxq 这个单向链表基于 CAS 来 hold 住这些并发，然后另外搞一个 _EntryList 这个双向链表，来在每次唤醒的时候搬迁一些线程节点，降低 _cxq 的尾部竞争。

3.3 重量级锁的优化策略

JVM内置锁在1.5之后版本做了重大的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、自适应自旋（Adaptive Spinning）等技术来减少锁操作的开销，内置锁的并发性能已经基本与Lock持平。

锁粗化

锁粗化，简单来说，就是将多个连续的锁扩展为一个更大范围的锁。也就是说，如果 JVM 检测到有连续的对同一对象的加锁、解锁操作，就会把这些加锁、解锁操作合并为对这段区域进行一次连续的加锁和解锁。

```
synchronized (lock) {
    // 代码块 1
}
// 无关代码
synchronized (lock) {
    // 代码块 2
}
#JVM 在运行时可能会选择将上述两个小的同步块合并，形成一个大的同步块：
synchronized (lock) {
    // 代码块 1
    // 无关代码
    // 代码块 2
}
```

为什么锁粗化有效

加锁和解锁操作本身也会带来一定的性能开销，因为每次加锁和解锁都可能会涉及到线程切换、线程调度等开销。如果有大量小的同步块频繁地进行加锁和解锁，那么这部分开销可能会变得很大，从而降低程序的执行效率。

通过锁粗化，可以将多次加锁和解锁操作减少到一次，从而减少这部分开销，提高程序的运行效率。

如何在代码中实现锁粗化

在代码层面上，我们并不能直接控制 JVM 进行锁粗化，因为这是 JVM 在运行时动态进行的优化。不过，我们可以在编写代码时，尽量减少不必要的同步块，避免频繁加锁和解锁。这样，就为 JVM 的锁粗化优化提供了可能。

示例

```
StringBuffer buffer = new StringBuffer();
/**
 * 锁粗化
 */
public void append(){
    buffer.append("aaa").append(" bbb").append(" ccc");
}
```

上述代码每次调用 buffer.append 方法都需要加锁和解锁，如果JVM检测到有一连串的对同一个对象加锁和解锁的操作，就会将其合并成一次范围更大的加锁和解锁操作，即在第一次append方法时进行加锁，最后一次append方法结束后进行解锁。

锁粗化是 JVM 提供的一种优化手段，能够有效地提高并发编程的效率。在我们编写并发代码时，应当注意同步块的使用，尽量减少不必要的加锁和解锁，从而使得锁粗化技术能够发挥作用。

锁消除

锁消除主要应用在没有多线程竞争的情况下。具体来说，当一个数据仅在一个线程中使用，或者说这个数据的作用域仅限于一个线程时，这个线程对该数据的所有操作都不需要加锁。在 Java HotSpot VM 中，这种优化主要是通过逃逸分析（Escape Analysis）来实现的。

为什么锁消除有效

锁消除之所以有效，是因为它消除了不必要的锁竞争，从而减少了线程切换和线程调度带来的性能开销。当数据仅在单个线程中使用时，对此数据的所有操作都不需要同步。在这种情况下，锁操作不仅不会增加安全性，反而会因为增加了额外的执行开销而降低程序的运行效率。

如何在代码中实现锁消除

在代码层面上，我们无法直接控制 JVM 进行锁消除优化，这是由 JVM 的 JIT 编译器在运行时动态完成的。但我们可以通过编写高质量的代码，使 JIT 编译器更容易识别出可以进行锁消除的场景。例如：

```java
public class LockEliminationTest {
    /**
     * 锁消除
     * -XX:+EliminateLocks 开启锁消除(jdk8默认开启)
     * -XX:-EliminateLocks 关闭锁消除
     * @param str1
     * @param str2
     */
    public void append(String str1, String str2) {
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append(str1).append(str2);
    }

    public static void main(String[] args) throws InterruptedException {
        LockEliminationTest demo = new LockEliminationTest();
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            demo.append("aaa", "bbb");
        }
        long end = System.currentTimeMillis();
        System.out.println("执行时间: " + (end - start) + " ms");
    }

}
```

StringBuffer的append是个同步方法，但是append方法中的 StringBuffer 属于一个局部变量，不可能从该方法中逃逸出去，因此其实这过程是线程安全的，可以将锁消除。因此，JIT 编译器会发现这种情况并自动消除 append 操作中的锁竞争。

测试结果：

关闭锁消除执行时间：4688 ms

开启锁消除执行时间：2601 ms

CAS自旋优化

重量级锁竞争的时候，还可以使用自旋来进行优化，如果当前线程自旋成功（即这时候持锁线程已经退出了同步块，释放了锁），这时当前线程就可以避免阻塞。

- 自旋会占用 CPU 时间，单核 CPU 自旋就是浪费，多核 CPU 自旋才能发挥优势。

- 在 Java 6 之后自旋是自适应的，比如对象刚刚的一次自旋操作成功过，那么认为这次自旋成功的可能性会高，就多自旋几次；反之，就少自旋甚至不自旋，比较智能。可以使用 -XX:+UseSpinning 参数来开启自旋锁，使用-XX:PreBlockSpin 参数来设置自旋锁的等待次数。
- Java 7 之后不能控制是否开启自旋功能，自旋锁的参数被取消，自旋锁总是会执行，自旋次数也由虚拟机自行调整。

注意：自旋的目的是为了减少线程挂起的次数，尽量避免直接挂起线程（挂起操作涉及系统调用，存在用户态和内核态切换，这才是重量级锁最大的开销）

轻量级锁

我们再思考一下，是否有这样的场景：多个线程都是在不同的时间段来请求同一把锁，此时根本就用不需要阻塞线程，连 monitor 对象都不需要，所以就引入了轻量级锁这个概念，避免了系统调用，减少了开销。

在锁竞争不激烈的情况下，这种场景还是很常见的，可能是常态，所以轻量级锁的引入很有必要。

轻量级锁是否存在自旋问题分析

错误的理解：轻量级锁加锁失败会自旋，失败一定次数后会膨胀升级为重量级锁

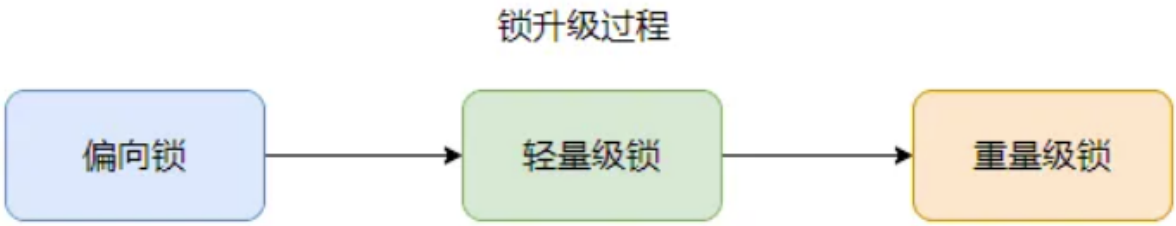
正确理解：轻量级锁不存在自旋，只有重量级锁加锁失败才会自旋。重量级锁加锁失败，会多次尝试cas和自适应自旋，如果一直加锁失败，就会阻塞当前线程等待唤醒

轻量级锁竞争没有自旋的原因其实是其设计并不是用于处理过于激烈的竞争场景，而是为了应对线程之间交替获取锁的场景。

偏向锁

我们再思考一下，是否有这样的场景：一开始一直只有一个线程持有这个锁，也不会有其他线程来竞争，此时频繁的 CAS 是没有必要的，CAS 也是有开销的。所以 synchronized就搞了个偏向锁，就是偏向一个线程，那么这个线程就可以直接获得锁。对于没有锁竞争的场合，偏向锁有很好的优化效果，可以消除锁重入（CAS操作）带来的开销。

锁升级的过程



注意： synchronized事实上是先有的重量级锁的实现，然后根据实际分析优化实现了偏向锁和轻量级锁。

4. synchronized锁升级详解

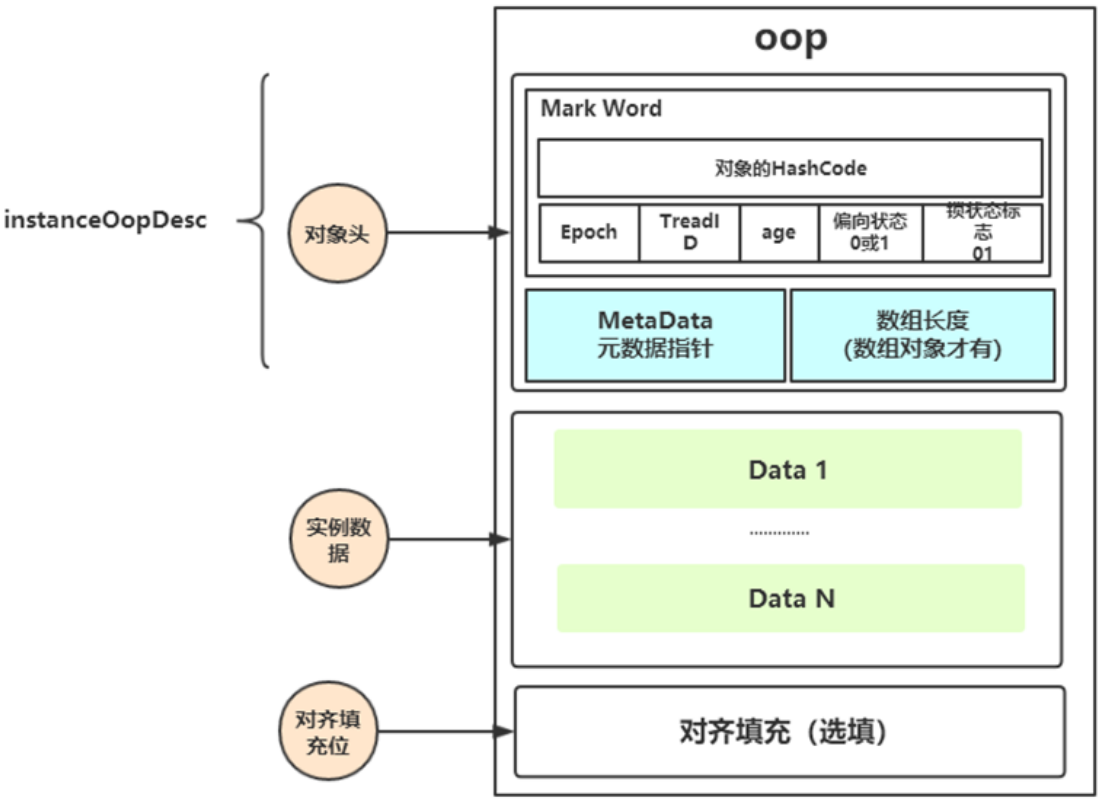
思考：synchronized加锁加在对象上，对象是如何记录锁状态的（如何判断是否加锁成功，锁状态记录在哪儿）？

4.1 sychronized多种锁状态设计详解

对象的内存布局

Hotspot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

- 对象头：比如 hash码，对象所属的年代，对象锁，锁状态标志，偏向锁（线程）ID，偏向时间，数组长度（数组对象才有）等。
- 实例数据：存放类的属性数据信息，包括父类的属性信息；
- 对齐填充：由于虚拟机要求 **对象起始地址必须是8字节的整数倍**。填充数据不是必须存在的，仅仅是为了字节对齐。



对象头详解

HotSpot虚拟机的对象头包括：

- Mark Word

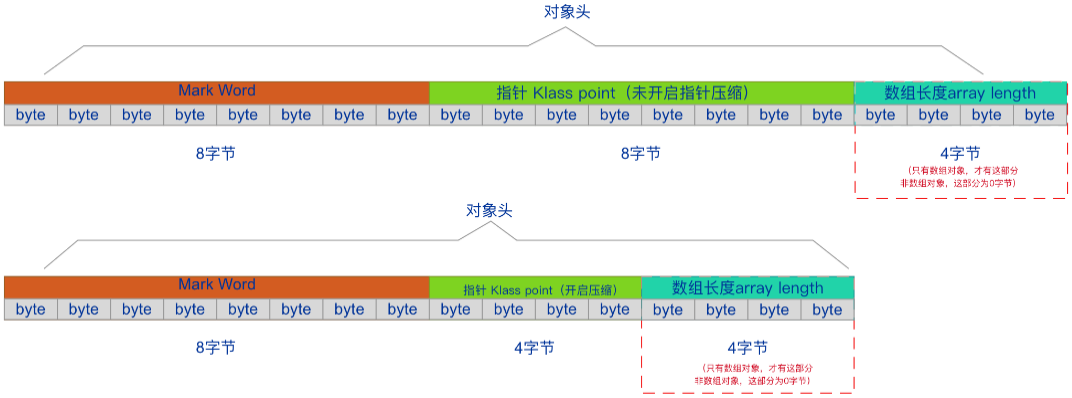
用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，官方称它为“Mark Word”。

- Klass Pointer

对象头的另外一部分是klass类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。 32位4字节，64位开启指针压缩或最大堆内存<32g时4字节，否则8字节。jdk1.8默认开启指针压缩后为4字节，当在JVM参数中关闭指针压缩（-XX:-UseCompressedOops）后，长度为8字节。

- 数组长度（只有数组对象有）

如果对象是一个数组, 那在对象头中还必须有一块数据用于记录数组长度。 4字节



使用JOL工具查看内存布局

给大家推荐一个可以查看普通java对象的内部布局工具JOL(JAVA OBJECT LAYOUT)，使用此工具可以查看new出来的一个java对象的内部布局,以及一个普通的java对象占用多少字节。

引入maven依赖

```
<!-- 查看Java 对象布局、大小工具 -->
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.10</version>
</dependency>
```

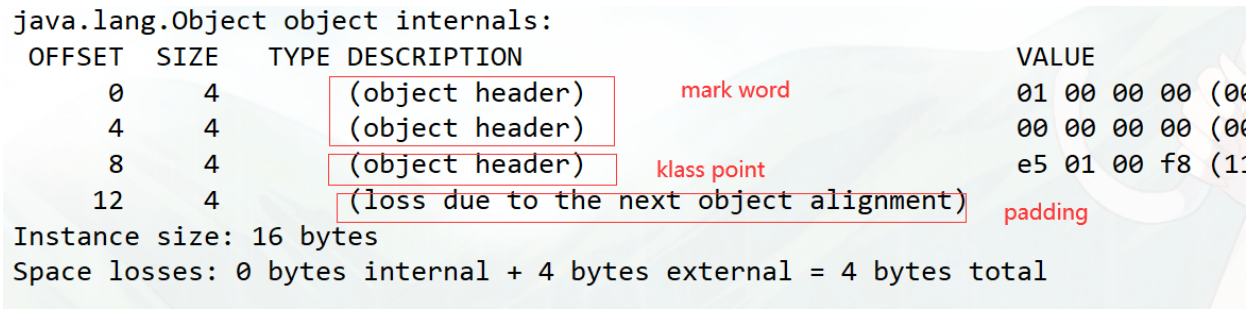
使用方法

```
//查看对象内部信息
System.out.println(ClassLayout.parseInstance(obj).toPrintable());
```

测试

```
public static void main(String[] args) throws InterruptedException {
    Object obj = new Object();
    //查看对象内部信息
    System.out.println(ClassLayout.parseInstance(obj).toPrintable());
}
```

1. 利用jol查看64位系统java对象（空对象），默认开启指针压缩，总大小显示16字节，前12字节为对象头



- OFFSET：偏移地址，单位字节；

- SIZE：占用的内存大小，单位为字节；
- TYPE DESCRIPTION：类型描述，其中object header为对象头；
- VALUE：对应内存中当前存储的值，二进制32位；

2. 关闭指针压缩后，对象头为16字节：-XX:-UseCompressedOops

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00 (00000001 00000000 0
      4     4        (object header)                           00 00 00 00 (00000000 00000000 0
      8     4        (object header)                           00 1c 47 1c (00000000 00011100 0
     12     4        (object header)                           00 00 00 00 (00000000 00000000 0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

mark word

klass point

思考： 下面例子中obj对象占多少个字节?

```java
public class ObjectTest {
    public static void main(String[] args) throws InterruptedException {
        Object obj = new Test();
        //查看对象内部信息
        System.out.println(ClassLayout.parseInstance(obj).toPrintable());
    }
}
class Test{
    private long p;
}
```

回到之前的问题： synchronized加锁加在对象上，对象是如何记录锁状态的?
锁状态被记录在每个对象的对象头的Mark Word中

Mark Word是如何记录锁状态的

Hotspot通过markOop类型实现Mark Word，具体实现位于markOop.hpp文件中。由于对象需要存储的运行时数据很多，考虑到虚拟机的内存使用，markOop被设计成一个非固定的数据结构，以便在极小的空间存储尽量多的数据，根据对象的状态复用自己的存储空间。
简单点理解就是：MarkWord 结构搞得这么复杂，是因为需要节省内存，让同一个内存区域在不同阶段有不同的用处。

Mark Word的锁标记结构

```
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//             size:32 ------------------------------------------>| (CMS free block)
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)

。。。。。。
//    [JavaThread* | epoch | age | 1 | 01]       lock is biased toward given thread
//    [0           | epoch | age | 1 | 01]       lock is anonymously biased
//
//  - the two lock bits are used to describe three states: locked/unlocked and monitor.
//
//    [ptr             | 00]  locked             ptr points to real header on stack
//    [header      | 0 | 01]  unlocked           regular object header
//    [ptr             | 10]  monitor            inflated lock (header is wapped out)
//    [ptr             | 11]  marked             used by markSweep to mark an object
//                                               not valid at any other time
```

32位JVM下的对象结构描述

64位JVM下的对象结构描述

- hash： 保存对象的哈希码。运行期间调用System.identityHashCode()来计算，延迟计算，并把结果赋值到这里。
- age： 保存对象的分代年龄。表示对象被GC的次数，当该次数到达阈值的时候，对象就会转移到老年代。
- biased_lock： 偏向锁标识位。由于无锁和偏向锁的锁标识都是 01，没办法区分，这里引入一位的偏向锁标识位。
- lock： 锁状态标识位。区分锁状态，比如11时表示对象待GC回收状态, 只有最后2位锁标识(11)有效。
- JavaThread*： 保存持有偏向锁的线程ID。偏向模式的时候，当某个线程持有对象的时候，对象这里就会被置为该线程的ID。 在后面的操作中，就无需再进行尝试获取锁的动作。这个线程ID并不是JVM分配的线程ID号，和Java Thread中的ID是两个概念。
- epoch： 偏向锁撤销的计数器，可用于偏向锁批量重偏向和批量撤销的判断依据。
- ptr_to_lock_record：轻量级锁状态下，指向栈中锁记录的指针。当锁获取是无竞争时，JVM使用原子操作而不是OS互斥，这种技术称为轻量级锁定。在轻量级锁定的情况下，JVM通过CAS操作在对象的Mark Word中设置指向锁记录的指针。
- ptr_to_heavyweight_monitor：重量级锁状态下，指向对象监视器Monitor的指针。如果两个不同的线程同时在同一个对象上竞争，则必须将轻量级锁定升级到Monitor以管理等待的线程。在重量级锁定的情况下，JVM在对象的ptr_to_heavyweight_monitor设置指向Monitor的指针

4.2 锁升级场景实验

示例代码： 演示锁升级的过程

```
public class LockUpgrade {

    public static void main(String[] args) throws InterruptedException {
        User userTemp = new User();
        System.out.println("无锁状态（001）: "+ ClassLayout.parseInstance(userTemp).toPrintable());

        /* jvm默认延时4s自动开启偏向锁，可通过-XX:BiasedLockingStartupDelay=0取消延时;
        如果不要偏向锁，可通过-XX:-UseBiasedLocking=false来设置*/
        Thread.sleep(5000);

        User  user = new User();
        System.out.println("启用偏向锁(101):"+ ClassLayout.parseInstance(user).toPrintable());

        for(int i=0;i<2;i++){
            synchronized (user){
                System.out.println("偏向锁(101)(带线程id):"+ ClassLayout.parseInstance(user).toPrintable());
            }
            System.out.println("偏向锁释放(101)(带线程id): "+ ClassLayout.parseInstance(user).toPrintable());
        }

        new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (user){
                    System.out.println("轻量级锁(00):"+ ClassLayout.parseInstance(user).toPrintable());
                    try {
                        System.out.println("睡眠3秒=======================");
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    System.out.println("轻量级锁--->重量级锁(10):"+ ClassLayout.parseInstance(user).toPrintable());
                }
            }
```

```
35          }).start();
36
37          Thread.sleep(1000);
38          System.out.println("重量级锁(10):" +
    ClassLayout.parseInstance(user).toPrintable());
39          new Thread(new Runnable() {
40              @Override
41              public void run() {
42                  synchronized (user) {
43                      System.out.println("重量级锁(10):" +
    ClassLayout.parseInstance(user).toPrintable());
44                  }
45              }
46          }).start();
47
48          Thread.sleep(5000);
49          System.out.println("无锁状态(001):" +
    ClassLayout.parseInstance(user).toPrintable());
50
51      }
52
53  }
```

思考：重量级锁释放之后变为无锁，此时有新的线程来调用同步块，会获取什么锁?

轻量锁

3.4 轻量级锁详解

轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间多个线程访问同一把锁的场合，就会导致轻量级锁膨胀为重量级锁。

思考： 轻量级锁是否可以降级为偏向锁?

不能

轻量级锁实现原理

在介绍轻量级锁的原理之前，再看看之前 MarkWord 图。

轻量级锁操作的就是对象头的 MarkWord 。

如果判断当前处于无锁状态，会在当前线程栈的当前栈帧中划出一块叫 LockRecord 的区域，然后把锁对象的 MarkWord 拷贝一份到 LockRecord 中称之为 dhw(就是那个set_displaced_header 方法执行的)里。
然后通过 CAS 把锁对象头指向这个 LockRecord 。
轻量级锁的加锁过程：

如果当前是有锁状态，并且是当前线程持有的，则将 null 放到 dhw 中，这是重入锁的逻辑。

我们再看下轻量级锁解锁的逻辑：

逻辑还是很简单的，就是要把当前栈帧中 LockRecord 存储的 markword （dhw）通过 CAS 换回到对象头中。如果获取到的 dhw 是 null 说明此时是重入的，所以直接返回即可，否则就是利用 CAS 换，如果 CAS 失败说明此时有竞争，那么就膨胀！

3.5 偏向锁详解

偏向锁是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了消除数据在无竞争情况下锁重入（CAS操作）的开销而引入偏向锁。对于没有锁竞争的场合，偏向锁有很好的优化效果。

```
/***StringBuffer内部同步***/
public synchronized int length() {
    return count;
}
//System.out.println 无意识的使用锁
public void println(String x) {
  synchronized (this) {
     print(x); newLine();
  }
 }
```

偏向锁匿名偏向状态

当JVM启用了偏向锁模式（jdk6默认开启），新创建对象的Mark Word中的Thread Id为0，说明此时处于可偏向但未偏向任何线程，也叫做匿名偏向状态(anonymously biased)。

偏向锁延迟偏向

偏向锁模式存在偏向锁延迟机制：HotSpot 虚拟机在启动后有个 4s 的延迟才会对每个新建的对象开启偏向锁模式。JVM启动时会进行一系列的复杂活动，比如装载配置，系统类初始化等等。在这个过程中会使用大量synchronized关键字对对象加锁，且这些锁大多数都不是偏向锁。为了减少初始化时间，JVM默认延时加载偏向锁。

```
//关闭延迟开启偏向锁
-XX:BiasedLockingStartupDelay=0
//禁止偏向锁
-XX:-UseBiasedLocking
//启用偏向锁
-XX:+UseBiasedLocking
```

验证

```
@Slf4j
public class LockEscalationDemo{

    public static void main(String[] args) throws InterruptedException {
        log.debug(ClassLayout.parseInstance(new Object()).toPrintable());
        Thread.sleep(4000);
        log.debug(ClassLayout.parseInstance(new Object()).toPrintable());
    }
}
```

4s后偏向锁为可偏向或者匿名偏向状态：

思考：如果锁LockEscalationDemo.class会是什么状态?

偏向锁状态跟踪实验

```java
public class LockEscalationDemo {
    public static void main(String[] args) throws InterruptedException {
        log.debug(ClassLayout.parseInstance(new Object()).toPrintable());
        //HotSpot 虚拟机在启动后有个 4s 的延迟才会对每个新建的对象开启偏向锁模式
        Thread.sleep(4000);
        Object obj = new Object();

        new Thread(new Runnable() {
            @Override
            public void run() {
                log.debug(Thread.currentThread().getName()+"开始执行。。。\n"
                        +ClassLayout.parseInstance(obj).toPrintable());
                synchronized (obj){
                    log.debug(Thread.currentThread().getName()+"获取锁执行中。。。\n"
                            +ClassLayout.parseInstance(obj).toPrintable());
                }
                log.debug(Thread.currentThread().getName()+"释放锁。。。\n"
                        +ClassLayout.parseInstance(obj).toPrintable());
            }
        },"thread1").start();

        Thread.sleep(5000);
        log.debug(ClassLayout.parseInstance(obj).toPrintable());
    }
}
```

思考：如果对象调用了hashCode,还会开启偏向锁模式吗?

偏向锁撤销场景

偏向锁撤销之调用对象HashCode

调用锁对象的obj.hashCode()或System.identityHashCode(obj)方法会导致该对象的偏向锁被撤销。因为对于一个对象，其HashCode只会生成一次并保存，偏向锁是没有地方保存hashcode的。

- 轻量级锁会在锁记录中记录 hashCode
- 重量级锁会在 Monitor 中记录 hashCode

当对象处于可偏向（也就是线程ID为0）和已偏向的状态下，调用HashCode计算将会使对象再也无法偏向：

- 当对象可偏向时，MarkWord将变成未锁定状态，并只能升级成轻量锁；
- 当对象正处于偏向锁时，调用HashCode将使偏向锁强制升级成重量锁。

偏向锁撤销之调用wait/notify

偏向锁状态执行obj.notify() 会升级为轻量级锁，调用obj.wait(timeout) 会升级为重量级锁

```
synchronized (obj) {
    // 思考：偏向锁执行过程中，调用hashcode会发生什么？
    //obj.hashCode();
    //obj.notify();
    try {
        obj.wait(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    log.debug(Thread.currentThread().getName() + "获取锁执行中。。。\n"
            + ClassLayout.parseInstance(obj).toPrintable());
}
```

测试结果：

偏向锁批量重偏向&批量撤销

从偏向锁的加锁解锁过程中可看出，当只有一个线程反复进入同步块时，偏向锁带来的性能开销基本可以忽略，但是当有其他线程尝试获得锁时，就需要等到safe point时，再将偏向锁撤销为无锁状态或升级为轻量级，会消耗一定的性能，所以在多线程竞争频繁的情况下，偏向锁不仅不能提高性能，还会导致性能下降。于是，就有了批量重偏向与批量撤销的机制。

实现原理

以class为单位，为每个class维护一个偏向锁撤销计数器，每一次该class的对象发生偏向撤销操作时，该计数器+1，当这个值达到重偏向阈值（默认20）时，JVM就认为该class的偏向锁有问题，因此会进行批量重偏向。

当达到重偏向阈值（默认20）后，假设该class计数器继续增长，当其达到批量撤销的阈值后（默认40），JVM就认为该class的使用场景存在多线程竞争，会标记该class为不可偏向，之后，对于该class的锁，直接走轻量级锁的逻辑。

```
intx BiasedLockingBulkRebiasThreshold   = 20   //默认偏向锁批量重偏向阈值
intx BiasedLockingBulkRevokeThreshold  = 40   //默认偏向锁批量撤销阈值
```

我们可以通过-XX:BiasedLockingBulkRebiasThreshold 和 -XX:BiasedLockingBulkRevokeThreshold 来手动设置阈值

应用场景

批量重偏向（bulk rebias）机制是为了解决：一个线程创建了大量对象并执行了初始的同步操作，后来另一个线程也来将这些对象作为锁对象进行操作，这样会导致大量的偏向锁撤销操作。
批量撤销（bulk revoke）机制是为了解决：在明显多线程竞争剧烈的场景下使用偏向锁是不合适的。

批量重偏向实验

当撤销偏向锁阈值超过 20 次后，jvm 会这样觉得，我是不是偏向错了，于是会在给这些对象加锁时重新偏向至加锁线程，重偏向会重置对象 的 Thread ID

```java
@Slf4j
public class BiasedLockingTest {
    //延时产生可偏向对象
    Thread.sleep(5000);
    // 创建一个list，来存放锁对象
    List<Object> list = new ArrayList<>();

    // 线程1
    new Thread(() -> {
        for (int i = 0; i < 50; i++) {
            // 新建锁对象
            Object lock = new BiasedLockingTest();
            synchronized (lock) {
                list.add(lock);
            }
        }
        try {
            //为了防止JVM线程复用，在创建完对象后，保持线程thead1状态为存活
            Thread.sleep(100000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "thead1").start();

    //睡眠3s钟保证线程thead1创建对象完成
    Thread.sleep(3000);
    log.debug("打印thead1，list中第20个对象的对象头：");
    log.debug((ClassLayout.parseInstance(list.get(19)).toPrintable()));

    // 线程2
    new Thread(() -> {
        for (int i = 0; i < 40; i++) {
            Object obj = list.get(i);
            synchronized (obj) {
                if(i>=15&&i<=21||i>=38){
                    log.debug("thread2-第" + (i + 1) + "次加锁执行中\t"+
                            ClassLayout.parseInstance(obj).toPrintable());
                }
```

```
            }
            //if(i==17||i==19){
            //    log.debug("thread2-第" + (i + 1) + "次释放锁\t"+
            //            ClassLayout.parseInstance(obj).toPrintable());
            //}
        }
        try {
            Thread.sleep(100000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }, "thead2").start();

    LockSupport.park();
    }
}
```

测试结果：
thread1: 创建50个偏向线程thread1的偏向锁 1-50 偏向锁

thread2：
1-19 偏向锁撤销，升级为轻量级锁 （thread1释放锁之后为偏向锁状态）
20-40 偏向锁撤销达到阈值（20），执行了批量重偏向

批量撤销实验

当撤销偏向锁阈值超过 40 次后，jvm 会认为不该偏向，于是整个类的所有对象都会变为不可偏向的，新建的对象也是不可偏向的。

注意：时间-XX:BiasedLockingDecayTime=25000ms范围内没有达到40次，撤销次数清为0，重新计时

```java
@Slf4j
public class BiasedLockingTest {
    public static void main(String[] args) throws  InterruptedException {
        //延时产生可偏向对象
        Thread.sleep(5000);
        // 创建一个list，来存放锁对象
        List<Object> list = new ArrayList<>();

        // 线程1
        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                // 新建锁对象
                Object lock = new BiasedLockingTest();
                synchronized (lock) {
                    list.add(lock);
                }
            }
            try {
                //为了防止JVM线程复用，在创建完对象后，保持线程thead1状态为存活
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "thead1").start();

        //睡眠3s钟保证线程thead1创建对象完成
        Thread.sleep(3000);
        log.debug("打印thead1，list中第20个对象的对象头：");
        log.debug((ClassLayout.parseInstance(list.get(19)).toPrintable()));

        // 线程2
        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                Object obj = list.get(i);
                synchronized (obj) {
                    if(i>=15&&i<=21||i>=38){
                        log.debug("thread2-第" + (i + 1) + "次加锁执行中\t"+
                                ClassLayout.parseInstance(obj).toPrintable());
```

```
                    }
                }
                //if(i==17||i==19){
                 //   log.debug("thread2-第" + (i + 1) + "次释放锁\t"+
                //            ClassLayout.parseInstance(obj).toPrintable());
                //}
            }
            try {
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "thead2").start();


        Thread.sleep(3000);

        new Thread(() -> {
            for (int i = 0; i < 40; i++) {
                Object lock =list.get(i);
                if(i>=17&&i<=21){
                    log.debug("thread3-第" + (i + 1) + "次准备加锁\t"+
                            ClassLayout.parseInstance(lock).toPrintable());
                }
                synchronized (lock){
                    if(i>=17&&i<=21){
                        log.debug("thread3-第" + (i + 1) + "次加锁执行中\t"+
                                ClassLayout.parseInstance(lock).toPrintable());
                    }
                }
            }
        },"thread3").start();

        Thread.sleep(3000);
        log.debug("查看新创建的对象");
        log.debug((ClassLayout.parseInstance(new BiasedLockingTest()).toPrintable()));

        LockSupport.park();
    }
```

```
78 }
```

测试结果：

thread3：

1-19 从无锁状态直接获取轻量级锁 （thread2释放锁之后变为无锁状态）

20-40 偏向锁撤销，升级为轻量级锁 （thread2释放锁之后为偏向锁状态）

达到偏向锁撤销的阈值40，BiasedLockingTest会设置为不可偏向，所以新创建的对象是无锁状态

总结

1. 批量重偏向和批量撤销是针对类的优化，和对象无关。
2. 偏向锁重偏向一次之后不可再次重偏向。
3. 当某个类已经触发批量撤销机制后，JVM会默认当前类产生了严重的问题，剥夺了该类的新实例对象使用偏向锁的权利

4.3 锁升级的流程分析

synchronized源码分析

偏向锁源码分析

https://www.processon.com/view/link/6188c6c95653bb1471107283

轻量级锁源码分析

https://www.processon.com/view/link/618e7dd61e0853689b0c0df0

重量级锁源码分析

https://www.processon.com/view/link/618e7e1b7d9c08562aec6638