主讲老师：Fox

有道云链接：http://note.youdao.com/noteshare?id=7658b1b38c3b23412a481693f28aa2b9&sub=C1F89A6907AA4A1193D340507D115BD8

==========synchronized基础篇==========

Java共享内存模型带来的线程安全问题

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？

```
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);
  }
}
```

问题分析

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）

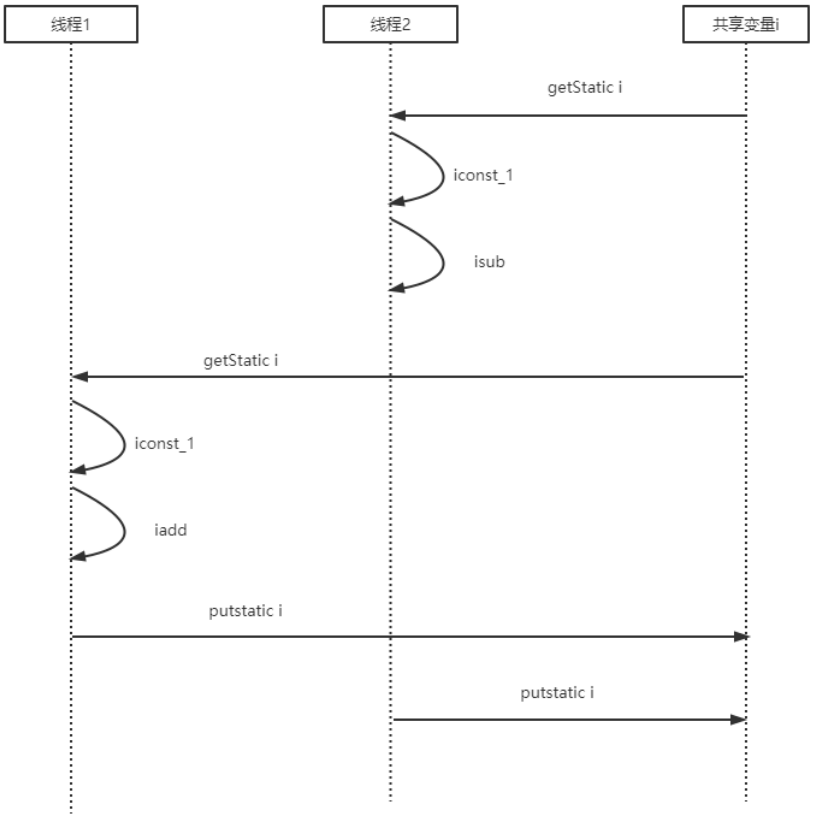
i++的JVM 字节码指令

```
getstatic i // 获取静态变量i的值
iconst_1 // 将int常量1压入操作数栈
iadd // 自增
putstatic i // 将修改后的值存入静态变量i
```

i--的JVM 字节码指令

```
getstatic i // 获取静态变量i的值
iconst_1 // 将int常量1压入操作数栈
isub // 自减
putstatic i // 将修改后的值存入静态变量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：



临界区（Critical Section）

- 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

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

```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--;
}
```

竞态条件（Race Condition）

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

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

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。

加锁方式

分类	具体分类	被锁的对象	伪代码

方法	实例方法	类的实例对象	//实例方法，锁住的是该类的实例对象
			public synchronized void method() {
			……
			}

	静态方法	类对象	//静态方法，锁住的是类对象
			public static synchronized void method1(){
			……
			}

代码块	实例对象	类的实例对象	//同步代码块，锁住的是该类的实例对象
			synchronized (this){
			……
			}

	class对象	类对象	//同步代码块，锁住的是该类的类对象
			synchronized (SynchronizedDemo.class){
			……
			}

	任意实例对象Object	实例对象Object	//同步代码块，锁住的是配置的实例对象
			//String对象作为锁
			String lock = "";
			synchronized (lock){
			……
			}

解决之前的共享问题

方式一

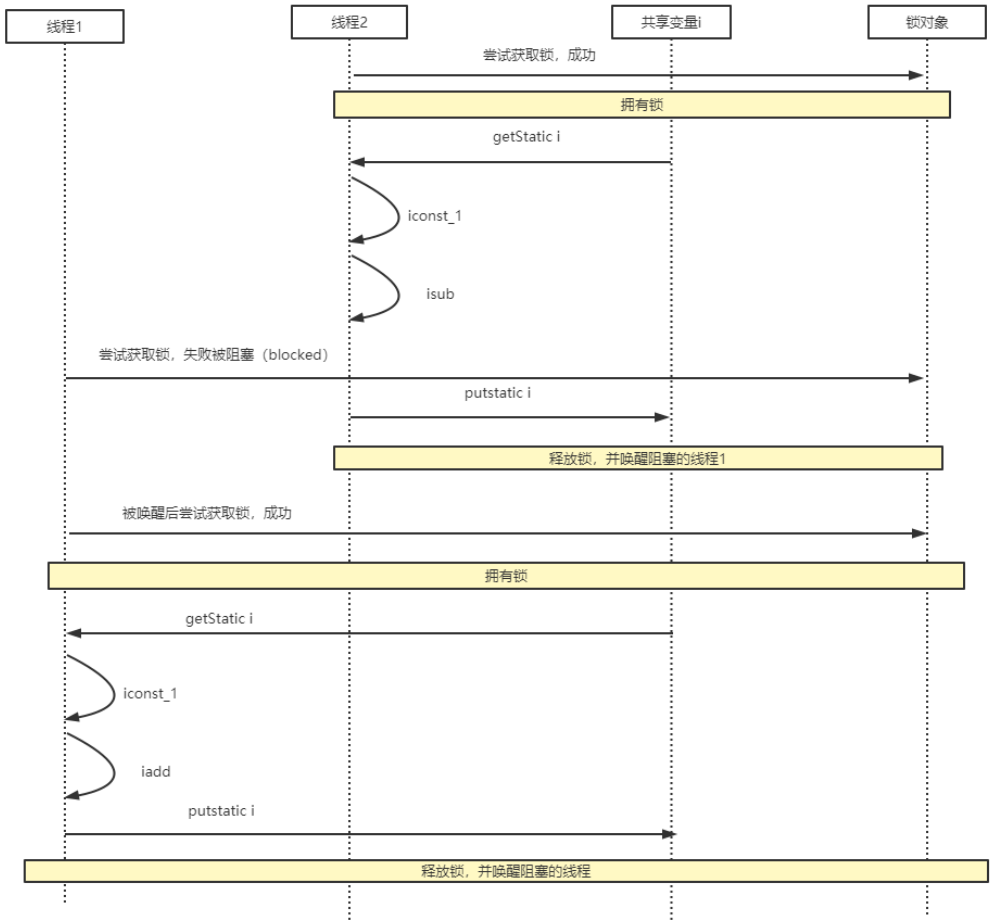
```java
public static synchronized void increment() {
 counter++;
```

```
}

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==========

synchronized底层原理

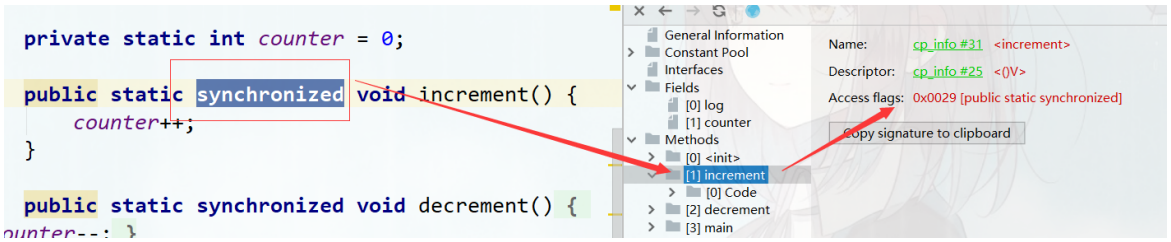
synchronized是JVM内置锁，基于Monitor机制实现，依赖底层操作系统的互斥原语Mutex（互斥量），它是一个重量级锁，性能较低。当然，JVM内置锁在1.5之后版本做了重大的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、自适应自旋（Adaptive Spinning）等技术来减少锁操作的开销，内置锁的并发性能已经基本与Lock持平。

> 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。

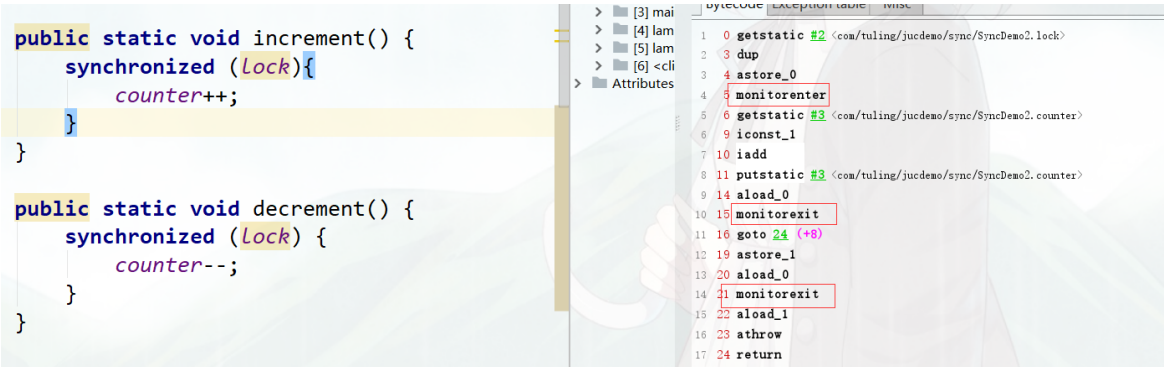
同步方法是通过方法中的access_flags中设置ACC_SYNCHRONIZED标志来实现；同步代码块是通过monitorenter和monitorexit来实现。两个指令的执行是JVM通过调用操作系统的互斥原语mutex来实现，被阻塞的线程会被挂起、等待重新调度，会导致“用户态和内核态”两个态之间来回切换，对性能有较大影响。

查看synchronized的字节码指令序列



Method access and property flags:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; must not be overridden (§5.4.5).
ACC_SYNCHRONIZED	0x0020	Declared synchronized; invocation is wrapped by a monitor use.
ACC_BRIDGE	0x0040	A bridge method, generated by the compiler.
ACC_VARARGS	0x0080	Declared with variable number of arguments.

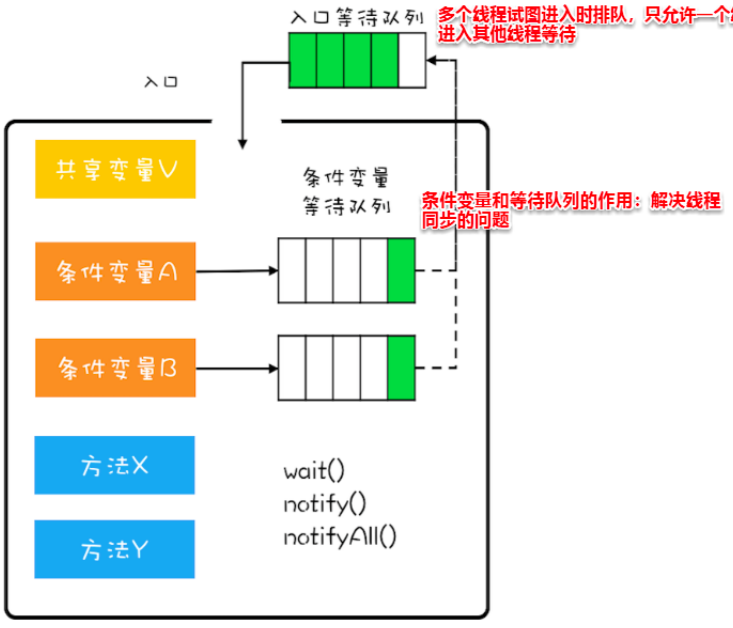


Monitor（管程/监视器）

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。

MESA模型

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：



MESA 管程模型

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。

wait()的正确使用姿势

对于MESA管程来说，有一个编程范式：

```
while(条件不满足) {
  wait();
```

```
}
```

唤醒的时间和获取到锁继续执行的时间是不一致的，被唤醒的线程再次执行时可能条件又不满足了，所以循环检验条件。MESA模型的wait()方法还有一个超时参数，为了避免线程进入等待队列永久阻塞。

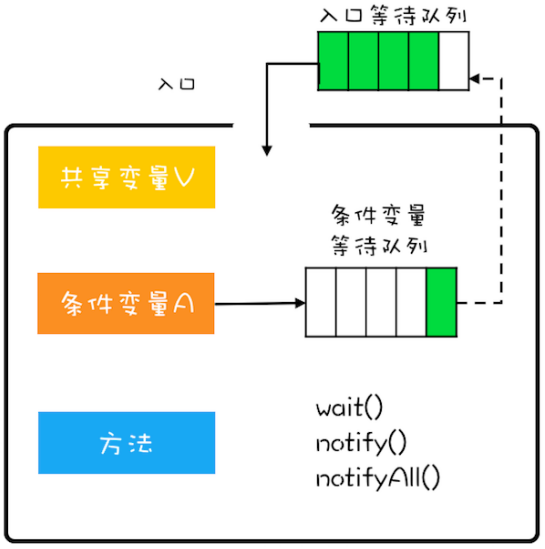
notify()和notifyAll()分别何时使用

满足以下三个条件时，可以使用notify()，其余情况尽量使用notifyAll()：

1. 所有等待线程拥有相同的等待条件；
2. 所有等待线程被唤醒后，执行相同的操作；
3. 只需要唤醒一个线程。

Java语言的内置管程synchronized

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。



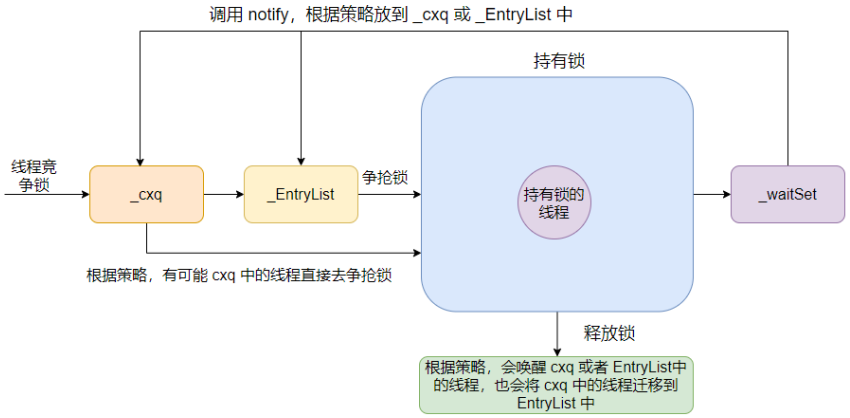
Monitor机制在Java中的实现

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）
```

```
8   _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
9   _WaitSetLock = 0 ;
10   _Responsible = NULL ;
11   _succ = NULL ;
12   _cxq = NULL ; //多线程竞争锁会先存到这个单向链表中 （FILO栈结构）
13   FreeNext = NULL ;
14   _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)
15   _SpinFreq = 0 ;
16   _SpinClock = 0 ;
17   OwnerIsThread = 0 ;
18   _previous_owner_tid = 0;
19   }
```



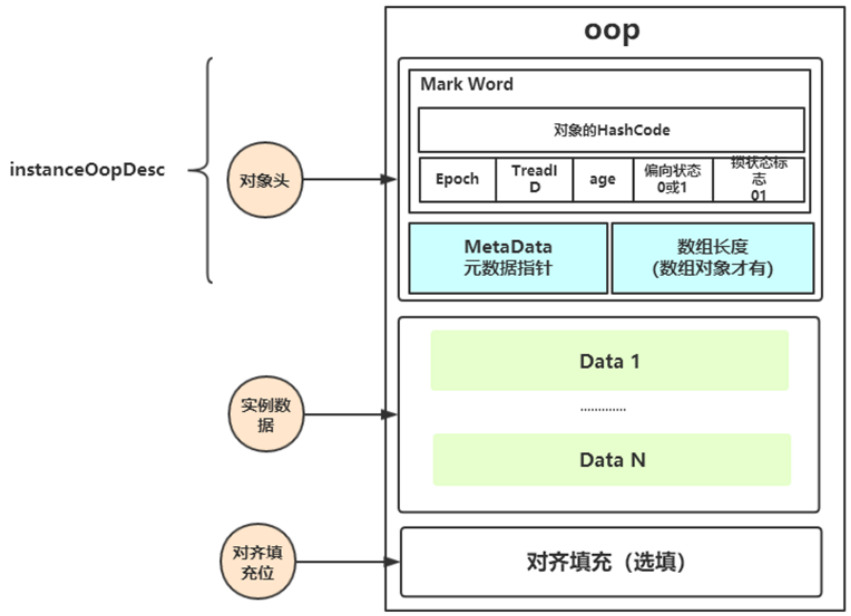
> 在获取锁时，是将当前线程插入到cxq的头部，而释放锁时，默认策略（QMode=0）是：如果EntryList为空，则将cxq中的元素按原有顺序插入到EntryList，并唤醒第一个线程，也就是当EntryList为空时，是后来的线程先获取锁。_EntryList不为空，直接从_EntryList中唤醒线程。

思考：synchronized加锁加在对象上，锁对象是如何记录锁状态的?

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

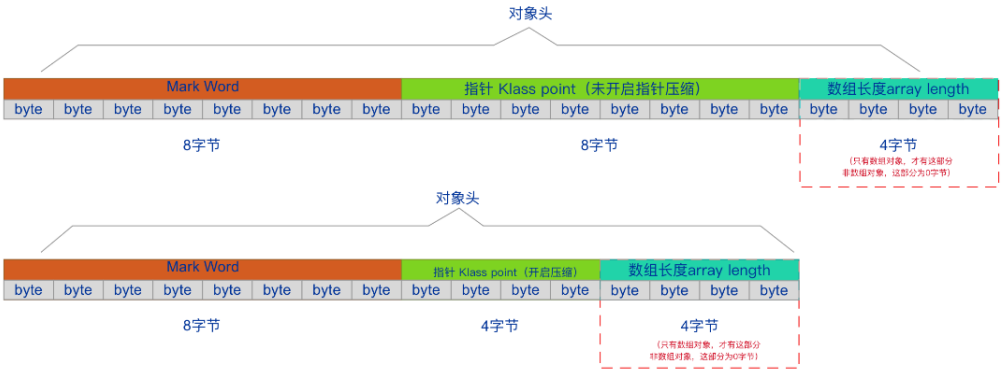
用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机中分别为32bit和64bit，官方称它为“Mark Word”。

- 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

```xml
<!-- 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

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

测试

```java
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

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00 (00
      4     4        (object header)                           00 00 00 00 (00
      8     4        (object header)                           e5 01 00 f8 (11
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

mark word

klass point

padding

- 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
```

- 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： 保存偏向时间戳。偏向锁在CAS锁操作过程中，偏向性标识，表示对象更偏向哪个锁。

32位JVM下的对象结构描述

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥锁（重量级锁）的指针				10
GC标记	空				11

64位JVM下的对象结构描述

锁状态	56bit		1bit	4bit	1bit	2bit
					是否偏向锁	锁标志位
无锁态	unused：25bit	对象的hashCode：31bit	unused	分代年龄	0	01
偏向锁	线程ID：54bit	Epoch：2bit	unused	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针（ptr_to_lock_record）					00
重量级锁	指向互斥锁（重量级锁）的指针（ptr_to_heavyweight_monitor）					10
GC标记	空					11

- 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

Mark Word中锁标记枚举

```
enum { locked_value = 0, //00 轻量级锁
 unlocked_value = 1, //001 无锁
 monitor_value = 2, //10 监视器锁，也叫膨胀锁，也叫重量级锁
 marked_value = 3, //11 GC标记
 biased_lock_pattern = 5 //101 偏向锁
 };
```

更直观的理解方式：

锁状态	存储内容	偏向锁标识位	锁标识位
无锁	哈希码、分代年龄	0	01
偏向锁	线程ID、时间戳、分代年龄	1	01
轻量级锁	指向栈中锁记录的指针ptr	无	00
重量级锁	指向monitor的指针ptr	无	10
GC标记	无	无	11

测试：利用JOL工具跟踪锁标记变化

偏向锁

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();
 }
 }
```

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

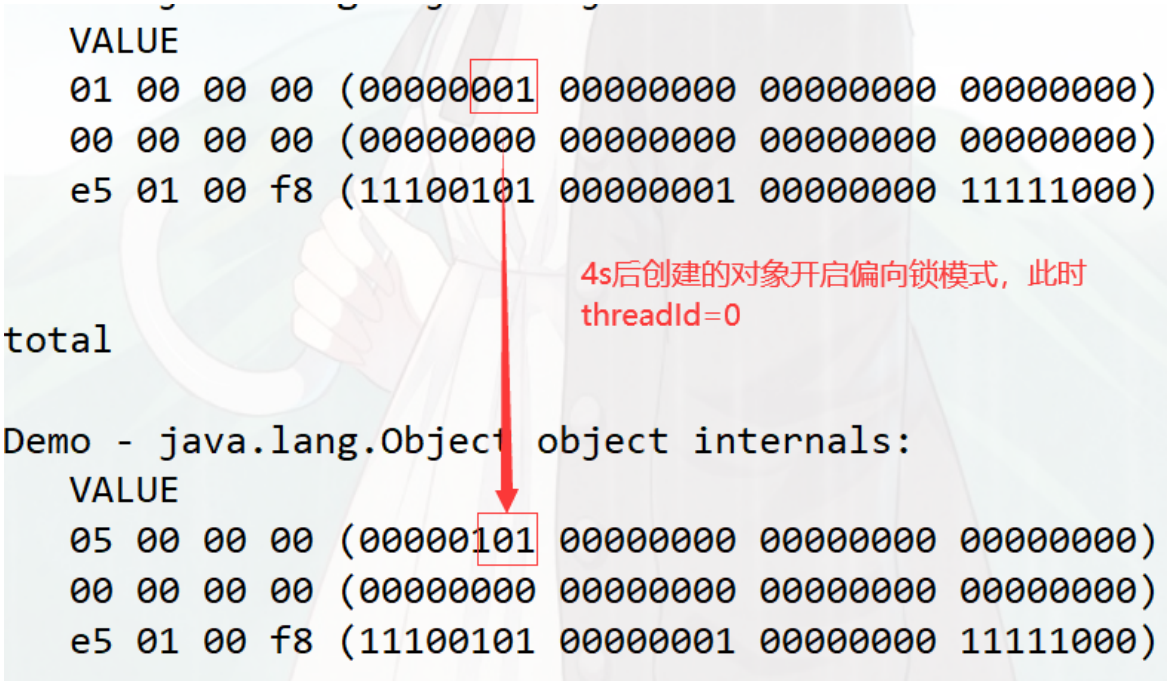
偏向锁模式存在偏向锁延迟机制：HotSpot 虚拟机在启动后有个 4s 的延迟才会对每个新建的对象开启偏向锁模式。JVM启动时会进行一系列的复杂活动，比如装载配置，系统类初始化等等。在这个过程中会使用大量synchronized关键字对对象加锁，且这些锁大多数都不是偏向锁。为了减少初始化时间，JVM默认延时加载偏向锁。

```
//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

```
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());
 }
}
```

```
.LockEscalationDemo - thread1开始执行。。。

           VALUE
           05 00 00 00 (00000101 00000000 00000000 00000000) (5)
           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
ignment)

4 bytes total

.LockEscalationDemo - thread1获取锁执行中。。。

           VALUE
           05 68 f9 1f (00000101 01101000 11111001 00011111) (536438789)
           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
ignment)

4 bytes total

.LockEscalationDemo - thread1释放锁。。。

           VALUE                                          偏向锁
           05 68 f9 1f (00000101 01101000 11111001 00011111) (536438789)
           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
ignment)
```

思考：如果对象调用了hashCode,还会开启偏向锁模式吗?

```
        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( millis: 4000);
        Object obj = new Object();
        // 思考： 如果对象调用了hashCode,还会开启偏向锁模式吗
        obj.hashCode();
        log.debug(ClassLayout.parseInstance(obj).toPrintable());
    }
edLockDemo › main()

.8.0_181\bin\java.exe" ...
om.tuling.jucdemo.sync.BiasedLockDemo - java.lang.Object object internals:
PTION                                VALUE
t header)                            01 00 00 00 (00000001 00000000 00000000 00000000) (
t header)                            00 00 00 00 (00000000 00000000 00000000 00000000) (
t header)                            e5 01 00 f8 (11100101 00000001 00000000 11111000) (
due to the next object alignment)

nal + 4 bytes external = 4 bytes total

om.tuling.jucdemo.sync.BiasedLockDemo - java.lang.Object object internals:
PTION                                VALUE
t header)                            01 79 f7 f5 (00000001 01111001 11110111 11110101) (
t header)                            46 00 00 00 (01000110 00000000 00000000 00000000) (
t header)                            e5 01 00 f8 (11100101 00000001 00000000 11111000) (
```

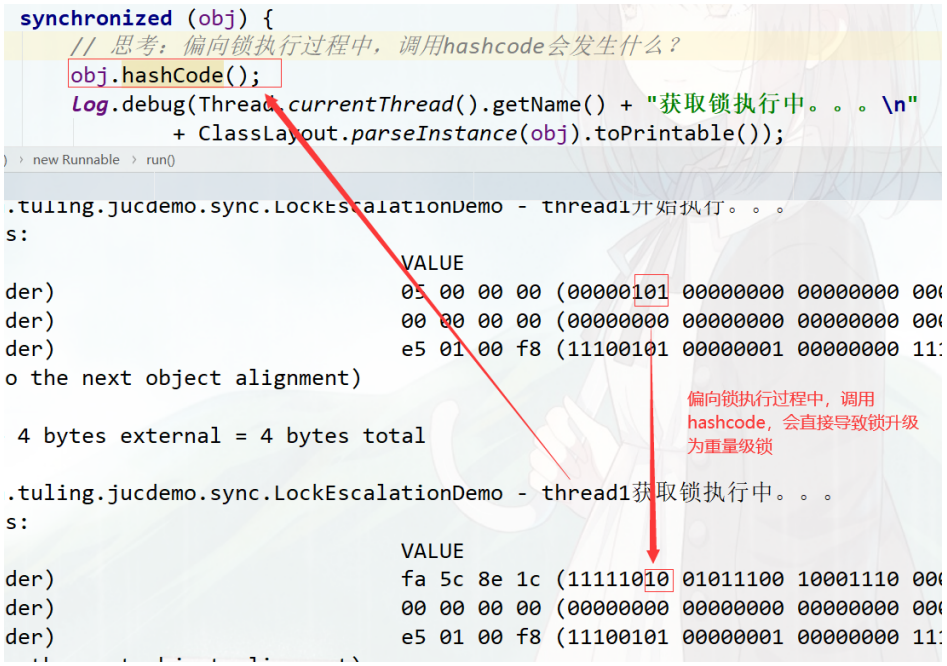
偏向锁撤销之调用对象HashCode

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：

```
..
:.LockEscalationDemo - thread1开始执行。。。

            VALUE
            05 00 00 00 (00000101 00000000 00000000 00000000)
            00 00 00 00 (00000000 00000000 00000000 00000000)
            e5 01 00 f8 (11100101 00000001 00000000 11111000)
lignment)

 4 bytes total

:.LockEscalationDemo - thread1获取锁执行中。。。

            VALUE
            08 f0 c3 20 (00001000 11110000 11000011 00100000)
            00 00 00 00 (00000000 00000000 00000000 00000000)
            e5 01 00 f8 (11100101 00000001 00000000 11111000)
```

调用notify偏向锁撤销，升级为轻量级锁

```
" ...
sync.LockEscalationDemo - thread1开始执行。。。

              VALUE
              05 00 00 00 (00000101 00000000 00000000 00000000) (5)
              00 00 00 00 (00000000 00000000 00000000 00000000) (0)
              e5 01 00 f8 (11100101 00000001 00000000 11111000) (-1
t alignment)

l = 4 bytes total

sync.LockEscalationDemo - thread1获取锁执行中。。。

              VALUE
              0a 31 48 1d (00001010 00110001 01001000 00011101) (49
              00 00 00 00 (00000000 00000000 00000000 00000000) (0)
              e5 01 00 f8 (11100101 00000001 00000000 11111000) (-1
t alignment)

l = 4 bytes total
```

调用wait,偏向锁撤销升级为重量级锁

轻量级锁

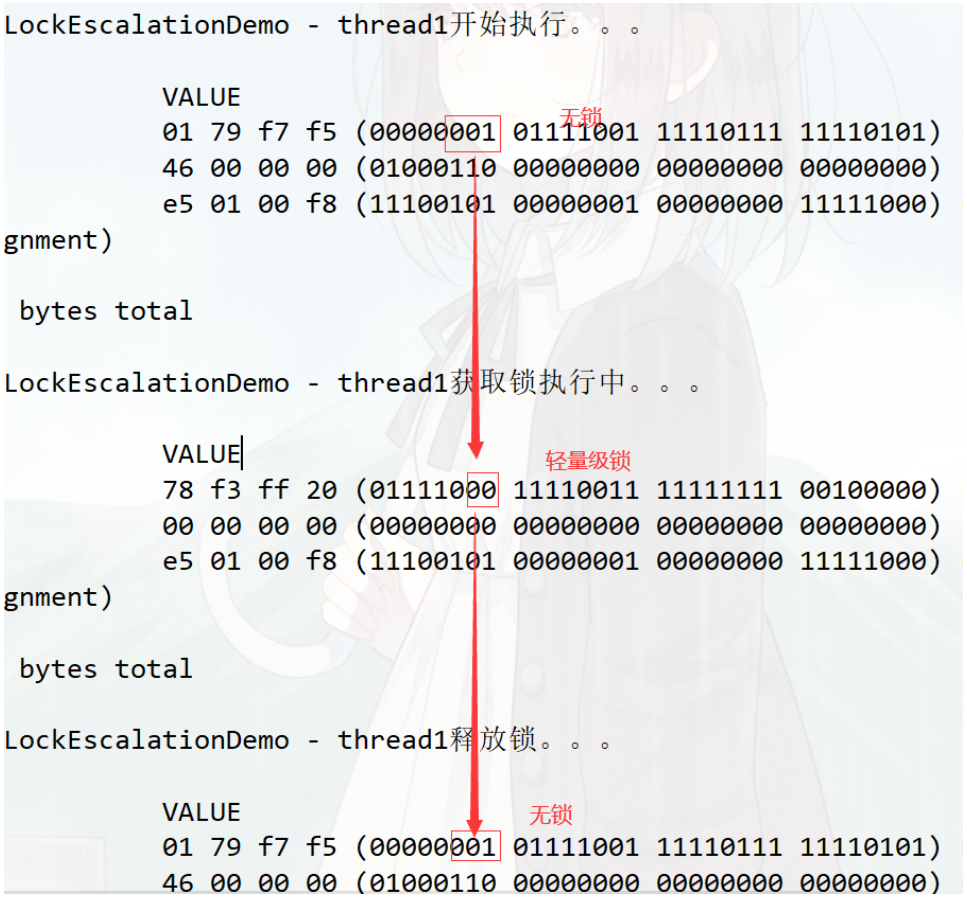
倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间多个线程访问同一把锁的场合，就会导致轻量级锁膨胀为重量级锁。

轻量级锁跟踪

```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();
 // 思考： 如果对象调用了hashCode,还会开启偏向锁模式吗
```

```
9   obj.hashCode();
10   //log.debug(ClassLayout.parseInstance(obj).toPrintable());
11
12   new Thread(new Runnable() {
13   @Override
14   public void run() {
15   log.debug(Thread.currentThread().getName()+"开始执行。。。\n"
16   +ClassLayout.parseInstance(obj).toPrintable());
17   synchronized (obj){
18   log.debug(Thread.currentThread().getName()+"获取锁执行中。。。\n"
19   +ClassLayout.parseInstance(obj).toPrintable());
20   }
21   log.debug(Thread.currentThread().getName()+"释放锁。。。\n"
22   +ClassLayout.parseInstance(obj).toPrintable());
23   }
24   },"thread1").start();
25
26   Thread.sleep(5000);
27   log.debug(ClassLayout.parseInstance(obj).toPrintable());
28   }
29 }
```

思考： 轻量级锁是否可以降级为偏向锁?



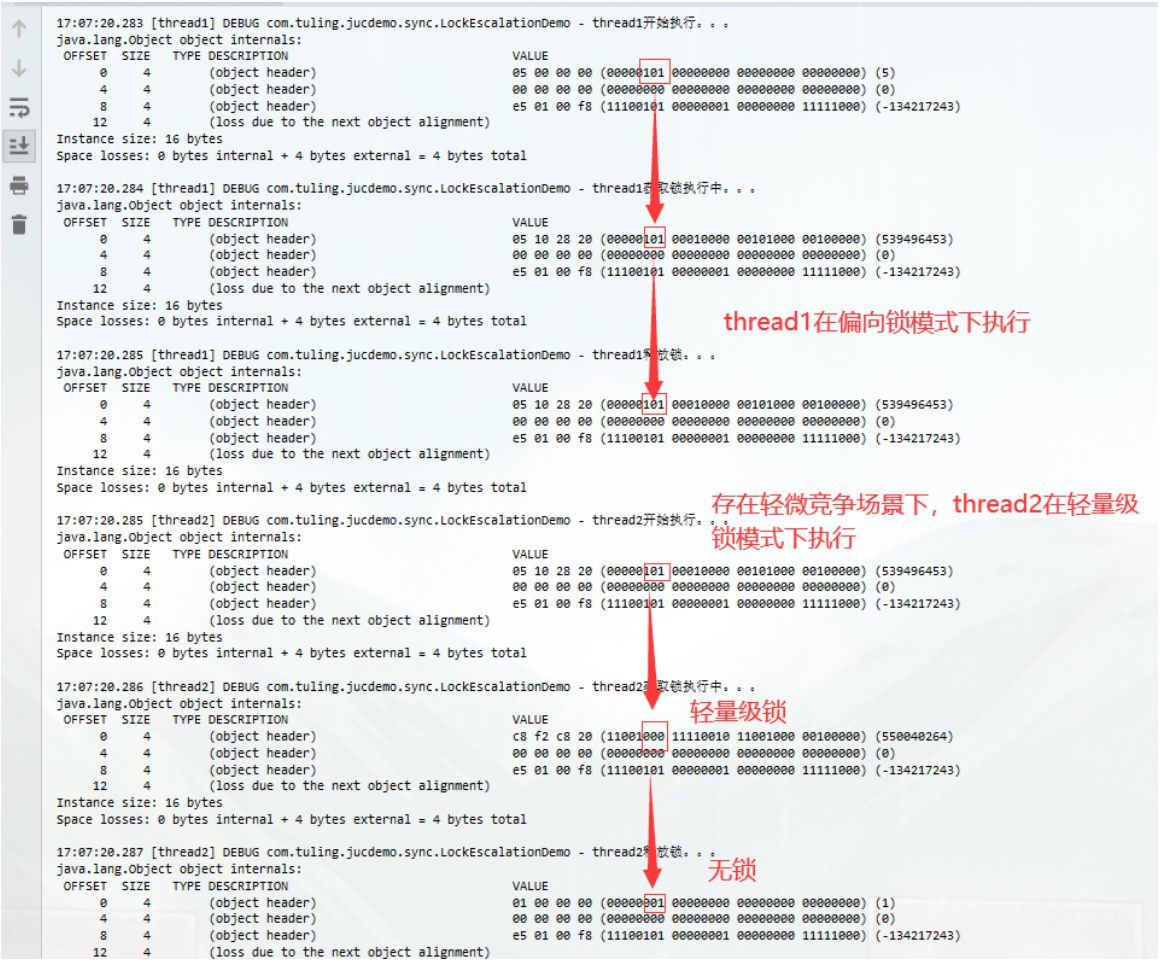
测试：锁升级场景

偏向锁升级轻量级锁

模拟两个线程轻微竞争场景

```
@Slf4j
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();
 // 思考： 如果对象调用了hashCode,还会开启偏向锁模式吗
  //obj.hashCode();
  //log.debug(ClassLayout.parseInstance(obj).toPrintable());

  Thread thread1 = new Thread(new Runnable() {
  @Override
  public void run() {
  log.debug(Thread.currentThread().getName() + "开始执行。。。\n"
  + ClassLayout.parseInstance(obj).toPrintable());
  synchronized (obj) {
  // 思考：偏向锁执行过程中，调用hashcode会发生什么？
  //obj.hashCode();
  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");
  thread1.start();

  //控制线程竞争时机
  Thread.sleep(1);

  Thread thread2 = 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"
```

```
41    +ClassLayout.parseInstance(obj).toPrintable());
42    }
43    log.debug(Thread.currentThread().getName()+"释放锁。。。\n"
44    +ClassLayout.parseInstance(obj).toPrintable());
45    }
46    },"thread2");
47    thread2.start();
48
49    Thread.sleep(5000);
50    log.debug(ClassLayout.parseInstance(obj).toPrintable());
51
52    }
53  }
```



轻量级锁膨胀为重量级锁

```
1  @Slf4j
2  public class LockEscalationDemo {
3
4   public static void main(String[] args) throws InterruptedException {
5
6   log.debug(ClassLayout.parseInstance(new Object()).toPrintable());
7   //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
```

```
8   Thread.sleep(4000);
9   Object obj = new Object();
10
11   new Thread(new Runnable() {
12   @Override
13   public void run() {
14   log.debug(Thread.currentThread().getName()+"开始执行。。。\n"
15   +ClassLayout.parseInstance(obj).toPrintable());
16   synchronized (obj){
17   log.debug(Thread.currentThread().getName()+"获取锁执行中。。。\n"
18   +ClassLayout.parseInstance(obj).toPrintable());
19   }
20   log.debug(Thread.currentThread().getName()+"释放锁。。。\n"
21   +ClassLayout.parseInstance(obj).toPrintable());
22   }
23   },"thread1").start();
24
25   new Thread(new Runnable() {
26   @Override
27   public void run() {
28   log.debug(Thread.currentThread().getName()+"开始执行。。。\n"
29   +ClassLayout.parseInstance(obj).toPrintable());
30   synchronized (obj){
31   log.debug(Thread.currentThread().getName()+"获取锁执行中。。。\n"
32   +ClassLayout.parseInstance(obj).toPrintable());
33   }
34   log.debug(Thread.currentThread().getName()+"释放锁。。。\n"
35   +ClassLayout.parseInstance(obj).toPrintable());
36   }
37   },"thread2").start();
38
39   Thread.sleep(5000);
40   log.debug(ClassLayout.parseInstance(obj).toPrintable());
41   }
42  }
```

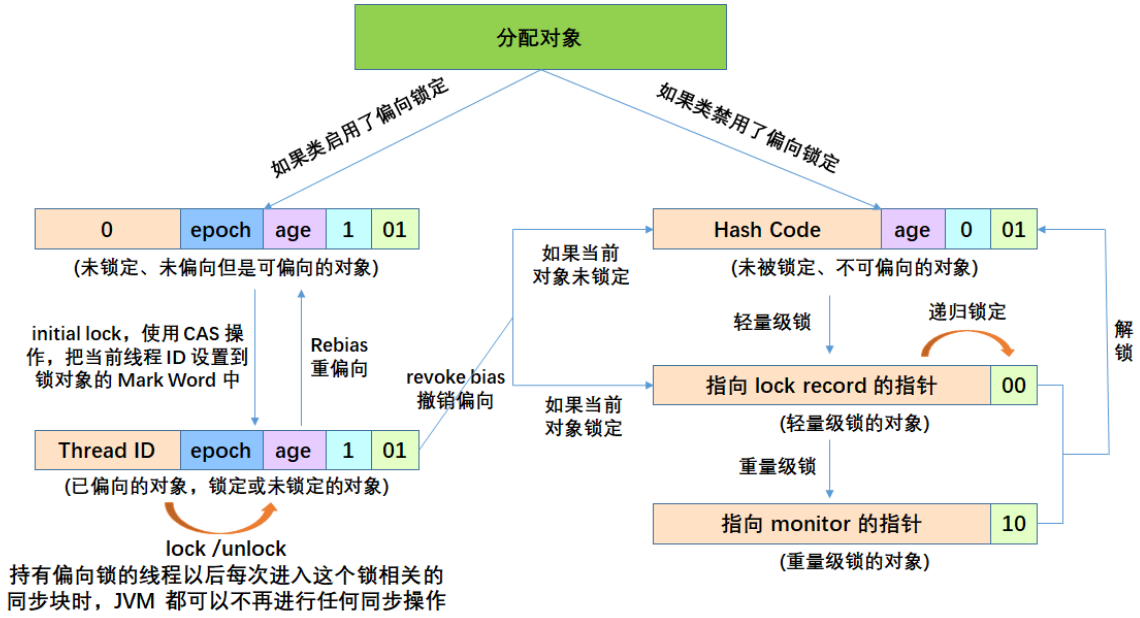
```
ing.jucdemo.sync.LockEscalationDemo - thread1开始执行...

                           VALUE
                           05 00 00 00 (00000101 00000000 00000000 00000000) (5)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total

ing.jucdemo.sync.LockEscalationDemo - thread2开始执行...

                           VALUE
                           05 00 00 00 (00000101 00000000 00000000 00000000) (5)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total

ing.jucdemo.sync.LockEscalationDemo - thread1获取锁执行中...

                           VALUE
                           05 d0 9b 20 (00000101 11010000 10011011 00100000) (547082245)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total

ing.jucdemo.sync.LockEscalationDemo - thread2获取锁执行中...

                           VALUE
                           3a 32 32 1d (00111010 00110010 00110010 00011101) (489828922)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total

ing.jucdemo.sync.LockEscalationDemo - thread1释放锁...

                           VALUE
                           3a 32 32 1d (00111010 00110010 00110010 00011101) (489828922)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total

ing.jucdemo.sync.LockEscalationDemo - thread2释放锁...

                           VALUE
                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
e next object alignment)

rtes external = 4 bytes total
```

线程2升级为重量级锁

无锁

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？

总结：锁对象状态转换



锁升级的原理分析

会结合Hotspot源码重点分析

进阶：synchronized锁优化

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。

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。

每个class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class中的epoch的值。每次发生批量重偏向时，就将该值+1，同时遍历JVM中所有线程的栈，找到该class所有正处于加锁状态的偏向锁，将其epoch字段改为新值。下次获得锁时，发现当前对象的epoch值和class的epoch不相等，那就算当前已经偏向了其他线程，也不会执行撤销操作，而是直接通过CAS操作将其Mark Word的Thread Id 改成当前线程Id。

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。

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。

JVM的默认参数值

设置JVM参数-XX:+PrintFlagsFinal，在项目启动时即可输出JVM的默认参数值

```
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

测试：批量重偏向

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

```
@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 Object();
 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);
```

```
47    } catch (InterruptedException e) {
48    e.printStackTrace();
49    }
50    }, "thead2").start();
51
52    LockSupport.park();
53    }
54  }
```

测试结果：

thread1：创建50个偏向线程thread1的偏向锁　1-50 偏向锁

```
iles\Java\jdk1.8.0_181\bin\java.exe" ...
[main] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - 打印thead1，list中第20个对象的对象头：
[main] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - java.lang.Object object internals:
   TYPE DESCRIPTION                               VALUE
        (object header)                           05 a8 0d 20 (00000101 10101000 00001101 00100000) (537765893)
        (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
        (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
        (loss due to the next object alignment)
: 16 bytes
 0 bytes internal + 4 bytes external = 4 bytes total
```

偏向锁　偏向的线程ID信息

thread2：

1-18 偏向锁撤销，升级为轻量级锁　（thread1释放锁之后为偏向锁状态）

19-40 偏向锁撤销达到阈值（20），执行了批量重偏向　（测试结果在第19就开始批量重偏向了）

```
16:19:32.590 [thead2] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread2-第18次加锁执行中java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           20 f4 d5 21 (00100000 11110100 11010101 00100001) (567669792)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

16:19:32.592 [thead2] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread2-第19次加锁执行中java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 a9 54 20 (00000101 10101001 01010100 00100000) (542419205)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

16:19:32.592 [thead2] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread2-第20次加锁执行中java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 a9 54 20 (00000101 10101001 01010100 00100000) (542419205)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
```

前1-18次偏向锁撤销，升级为轻量级锁

偏向锁

从第19次开始，偏向锁撤销，进行了重偏向

测试：批量撤销

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

```
1  @Slf4j
2  public class BiasedLockingTest {
3   public static void main(String[] args) throws InterruptedException {
4   //延时产生可偏向对象
5   Thread.sleep(5000);
6   // 创建一个list，来存放锁对象
7   List<Object> list = new ArrayList<>();
8
```

```
// 线程1
 new Thread(() -> {
 for (int i = 0; i < 50; i++) {
 // 新建锁对象
 Object lock = new Object();
 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();
```

```
52
53
54      Thread.sleep(3000);
55
56      new Thread(() -> {
57      for (int i = 0; i < 50; i++) {
58      Object lock =list.get(i);
59      if(i>=17&&i<=21||i>=35&&i<=41){
60      log.debug("thread3-第" + (i + 1) + "次准备加锁\t"+
61      ClassLayout.parseInstance(lock).toPrintable());
62      }
63      synchronized (lock){
64      if(i>=17&&i<=21||i>=35&&i<=41){
65      log.debug("thread3-第" + (i + 1) + "次加锁执行中\t"+
66      ClassLayout.parseInstance(lock).toPrintable());
67      }
68      }
69      }
70      },"thread3").start();
71
72      Thread.sleep(3000);
73      log.debug("查看新创建的对象");
74      log.debug((ClassLayout.parseInstance(new Object()).toPrintable()));
75
76      LockSupport.park();
77      }
78  }
```

测试结果：

thread3：

1-18 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）

```
ad3] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第18次准备加锁  java.lang.Object object internals:
PE DESCRIPTION                               VALUE
   (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
   (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
   (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
   (loss due to the next object alignment)
bytes
tes internal + 4 bytes external = 4 bytes total

ad3] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第18次加锁执行中    java.lang.Object object internals:
PE DESCRIPTION                               VALUE
   (object header)                           70 f6 32 22 (01110000 11110110 00110010 00100010) (573765232)
   (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
   (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
   (loss due to the next object alignment)
bytes
```


19-40 偏向锁撤销，升级为轻量级锁 （thread2释放锁之后为偏向锁状态）

```
d3] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第19次准备加锁  java.lang.Object object internals:
E DESCRIPTION                               VALUE
  (object header)                           05 81 5e 20 (00000101 10000001 01011110 00100000) (543064325)
  (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
  (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
  (loss due to the next object alignment)
ytes
es internal + 4 bytes external = 4 bytes total

d3] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第19次加锁执行中    java.lang.Object object internals
E DESCRIPTION                               VALUE
  (object header)                           70 f6 32 22 (01110000 11110110 00110010 00100010) (573765232)
  (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
  (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
  (loss due to the next object alignment)
ytes
es internal + 4 bytes external = 4 bytes total
```

偏向锁撤销，升级为轻量级锁

41-50 达到偏向锁撤销的阈值40，批量撤销偏向锁，升级为轻量级锁 （thread1释放锁之后为偏向锁状态）

```
 DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第41次准备加锁  java.lang.Object object internals:
ESCRIPTION                               VALUE
bject header)                            05 10 5a 20 (00000101 00010000 01011010 00100000) (542773253)
bject header)                            00 00 00 00 (00000000 00000000 00000000 00000000) (0)
bject header)                            e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
loss due to the next object alignment)
s
internal + 4 bytes external = 4 bytes total

 DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - thread3-第41次加锁执行中    java.lang.Object object internals:
ESCRIPTION                               VALUE
bject header)                            70 f6 32 22 (01110000 11110110 00110010 00100010) (573765232)
bject header)                            00 00 00 00 (00000000 00000000 00000000 00000000) (0)
bject header)                            e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
loss due to the next object alignment)
```

达到偏向锁撤销阈值之后，批量撤销偏向锁，升级为轻量级锁

新创建的对象： 无锁状态

```
1] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - 查看新创建的对象
1] DEBUG com.tuling.jucdemo.sync.BiasedLockingTest - java.lang.Object object internals:
YPE DESCRIPTION                               VALUE
    (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
    (object header)                           00 00 00 00 (00000000 00000000 00000000 00000000) (0)
    (object header)                           e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
    (loss due to the next object alignment)
```

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

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。
- Java 7 之后不能控制是否开启自旋功能

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）

锁粗化

假设一系列的连续操作都会对同一个对象反复加锁及解锁，甚至加锁操作是出现在循环体中的，即使没有出现线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。如果JVM检测到有一连串零碎的操作都是对同一对象的加锁，将会扩大加锁同步的范围（即锁粗化）到整个操作序列的外部。

```
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。

锁消除

锁消除即删除不必要的加锁操作。锁消除是Java虚拟机在JIT编译期间，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过锁消除，可以节省毫无意义的请求锁时间。

```
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");
```

```
19    }
20    long end = System.currentTimeMillis();
21    System.out.println("执行时间: " + (end - start) + " ms");
22    }
23
24 }
```

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。

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

逃逸分析(Escape Analysis)

逃逸分析，是一种可以有效减少Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。通过逃逸分析，Java Hotspot编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。逃逸分析的基本行为就是分析对象动态作用域。

方法逃逸(对象逃出当前方法)

当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中。

线程逃逸((对象逃出当前线程)

这个对象甚至可能被其它线程访问到，例如赋值给类变量或可以在其它线程中访问的实例变量。

使用逃逸分析，编译器可以对代码做如下优化：

1.同步省略或锁消除(Synchronization Elimination)。如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。

2.将堆分配转化为栈分配(Stack Allocation)。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配。

3.分离对象或标量替换(Scalar Replacement)。有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

jdk6才开始引入该技术，jdk7开始默认开启逃逸分析。在Java代码运行时，可以通过JVM参数指定是否开启逃逸分析：

```
1 -XX:+DoEscapeAnalysis  //表示开启逃逸分析 (jdk1.8默认开启)
2 -XX:-DoEscapeAnalysis //表示关闭逃逸分析。
3 -XX:+EliminateAllocations //开启标量替换(默认打开)
4 -XX:+EliminateLocks //开启锁消除(jdk1.8默认开启)
```

测试

```
1 /**
2  * @author Fox
3  *
```

```
 * 进行两种测试
 * 关闭逃逸分析，同时调大堆空间，避免堆内GC的发生，如果有GC信息将会被打印出来
 * VM运行参数：-Xmx4G -Xms4G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError
 *
 * 开启逃逸分析 jdk8默认开启
 * VM运行参数：-Xmx4G -Xms4G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError
 *
 * 执行main方法后
 * jps 查看进程
 * jmap -histo 进程ID
 *
 */
@Slf4j
public class EscapeTest {

 public static void main(String[] args) {
 long start = System.currentTimeMillis();
 for (int i = 0; i < 500000; i++) {
 alloc();
 }

 long end = System.currentTimeMillis();

 log.info("执行时间: " + (end - start) + " ms");
 try {
 Thread.sleep(Integer.MAX_VALUE);
 } catch (InterruptedException e1) {
 e1.printStackTrace();
 }
 }

 /**
 * JIT编译时会对代码进行逃逸分析
 * 并不是所有对象存放在堆区，有的一部分存在线程栈空间
 * Ponit没有逃逸
 */
 private static String alloc() {
 Point point = new Point();
 return point.toString();
 }

 /**
```

```
46   *同步省略（锁消除） JIT编译阶段优化，JIT经过逃逸分析之后发现无线程安全问题，就会做锁消除
47   */
48   public void append(String str1, String str2) {
49   StringBuffer stringBuffer = new StringBuffer();
50   stringBuffer.append(str1).append(str2);
51   }
52
53   /**
54   * 标量替换
55   *
56   */
57   private static void test2() {
58   Point point = new Point(1,2);
59   System.out.println("point.x="+point.getX()+"; point.y="+point.getY());
60
61  // int x=1;
62  // int y=2;
63  // System.out.println("point.x="+x+"; point.y="+y);
64   }
65  }
66
67  @Data
68  @AllArgsConstructor
69  @NoArgsConstructor
70  class Point{
71   private int x;
72   private int y;
73
74  }
```

测试结果：开启逃逸分析，部分对象会在栈上分配

```
F:\Resource\learn-concurrent>jmap -histo 13472

 num     #instances         #bytes  class name
----------------------------------------------
   1:        556308       27904536  [C
   2:        178443        4282632  java.lang.String
   3:          8183        3893224  [B
   4:        134917        3238008  com.tuling.jucdemo.sync.Point
   5:          1054        3018192  [I
```

```
F:\Resource\learn-concurrent>jmap -histo 63320

 num     #instances         #bytes  class name
----------------------------------------------
   1:          1026       49468448  [I
   2:        559894       28076664  [C
   3:        509844       12236256  java.lang.String
   4:        500000       12000000  com.tuling.jucdemo.sync.Point
```

关闭逃逸分析