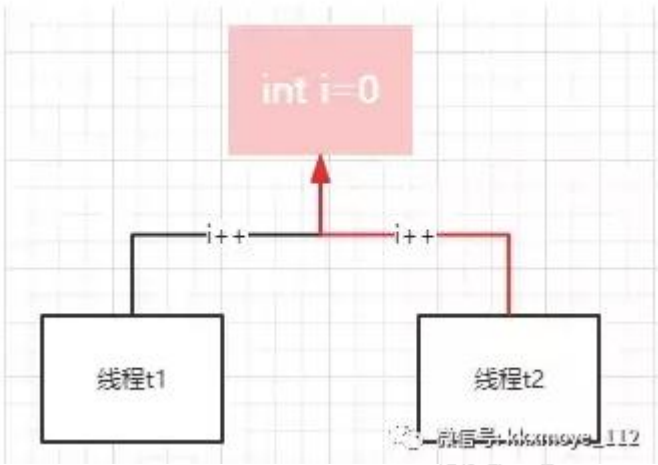
# 由一个问题引发的思考

线程的合理使用能够提升程序的处理性能，主要有两个方面，第一个是能够利用多核 cpu 以及超线程技术来实现线程的并行执行；第二个是线程的异步化执行相比于同步执行来说，异步执行能够很好的优化程序的处理性能提升并发吞吐量

同时，也带来了很多麻烦，举个简单的例子

多线程对于共享变量访问带来的安全性问题

一个变量 i. 假如一个线程去访问这个变量进行修改，这个时候对于数据的修改和访问没有任何问题。但是如果多个线程对于这同一个变量进行修改，就会存在一个数据安全性问题




对于线程安全性，本质上是管理对于数据状态的访问，而

且这个这个状态通常是共享的、可变的。共享，是指这个数据变量可以被多个线程访问；可变，指这个变量的值在它的生命周期内是可以改变的。

一个对象是否是线程安全的，取决于它是否会被多个线程访问，以及程序中是如何去使用这个对象的。所以，如果多个线程访问同一个共享对象，在不需额外的同步以及调用端代码不用做其他协调的情况下，这个共享对象的状态依然是正确的（正确性意味着这个对象的结果与我们预期规定的结果保持一致），那说明这个对象是线程安全的。

```
public class Demo{
    private static int count=0;
    public static void inc(){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count++;
    }
    public  static  void  main(String[]  args)
throws InterruptedException {
```

```
        for(int i=0;i<1000;i++){
            new Thread(()->Demo.inc()).start();
        }
        Thread.sleep(3000);

        System.out.println("运行结果"+count);
    }
}
```

- 在 Java 中如何解决由于线程并行导致的数据安全性问题呢?

思考如何保证线程并行的数据安全性

我们可以思考一下，问题的本质在于共享数据存在并发访问。如果我们能够有一种方法使得线程的并行变成串行，那是不是就不存在这个问题呢?

按照大家已有的知识，最先想到的应该就是锁吧。

毕竟这个场景并不模式，我们在和数据库打交道的时候，就了解过悲观锁、乐观锁的概念。 什么是锁？它是处理并发的一种同步手段，而如果需要达到前面我们说的一个目的，那么这个锁一定需要实现互斥的特性。

Java 提供的加锁方法就是 Synchroinzed 关键字。

# synchronized 的基本认识

在多线程并发编程中 synchronized 一直是元老级角色，很多人都会称呼它为重量级锁。但是，随着 Java SE 1.6 对 synchronized 进行了各种优化之后，有些情况下它就并不那么重，Java SE 1.6 中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁。这块在后续我们会慢慢展开

## synchronized 的基本语法

synchronized 有三种方式来加锁，分别是

1. 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
2. 静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
3. 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

不同的修饰类型，代表锁的控制粒度

## synchronized 的应用

修改前面的案例，使用 synchronized 关键字后，可以达到数据安全的效果

```
public class Demo{
    private static int count=0;
    public static void inc(){
        synchronized (Demo.class) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count++;
        }
    }
    public static void main(String[] args)
throws InterruptedException {
        for(int i=0;i<1000;i++){
            new Thread(()->Demo.inc()).start();
        }
        Thread.sleep(3000);

        System.out.println("运行结果"+count);
    }
}
```

## 思考锁是如何存储的

可以思考一下，要实现多线程的互斥特性，那这把锁需要哪些因素?

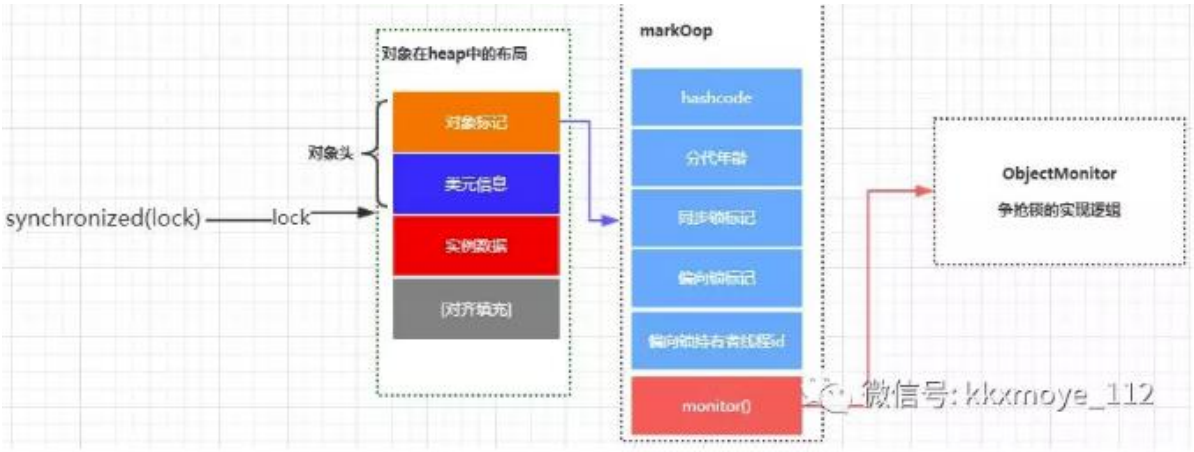
1. 锁需要有一个东西来表示，比如获得锁是什么状态、无锁状态是什么状态
2. 这个状态需要对多个线程共享

那么我们来分析，synchronized 锁是如何存储的呢？观察 synchronized 的整个语法发现，synchronized(lock)是基于 lock 这个对象的生命周期来控制锁粒度的，那是不是锁的存储和这个 lock 对象有关系呢?

于是我们以对象在 jvm 内存中是如何存储作为切入点，去看看对象里面有什么特性能够实现锁

### 对象在内存中的布局

在 Hotspot 虚拟机中，对象在内存中的存储布局，可以分为三个区域:对象头(Header)、实例数据(Instance Data)、对齐填充(Padding)




探究 Jvm 源码实现

当我们在 Java 代码中，使用 new 创建一个对象实例的时候，（hotspot 虚拟机）JVM 层面实际上会创建一个 instanceOopDesc 对象。

*Hotspot 虚拟机采用 OOP-Klass 模型来描述 Java 对象实例，OOP(Ordinary Object Point)指的是普通对象指针，Klass 用来描述对象实例的具体类型。Hotspot 采用 instanceOopDesc 和 arrayOopDesc 来描述对象头，arrayOopDesc 对象用来描述数组类型*

instanceOopDesc 的定义在 Hotspot 源码中的 instanceOop.hpp 文件中，另外，arrayOopDesc 的定义对应 arrayOop.hpp

```
class instanceOopDesc : public oopDesc {
 public:
  // aligned header size.
  static int header_size() { return sizeof(instanceOopDesc)/HeapWordSize; }
  // If compressed, the offset of the fields of the instance may not be aligne
  static int base_offset_in_bytes() {
    // offset computation code breaks if UseCompressedClassPointers
    // only is true
    return (UseCompressedOops && UseCompressedClassPointers) ?
             klass_gap_offset_in_bytes() :
             sizeof(instanceOopDesc);
  }
  static bool contains_field_offset(int offset, int nonstatic_field_size) {
    int base_in_bytes = base_offset_in_bytes();
    return (offset >= base_in_bytes &&
            (offset-base_in_bytes) < nonstatic_field_size * heapOopSize);
  }
};
#endif // SHARE_VM_OOPS_INSTANCEOOP_HPP
```

从 instanceOopDesc 代码中可以看到 instanceOopDesc 继承自 oopDesc，oopDesc 的定义载 Hotspot 源码中的 oop.hpp 文件中

在普通实例对象中，oopDesc 的定义包含两个成员，分别是 _mark 和 _metadata

_mark 表示对象标记、属于 markOop 类型，也就是接下来要讲解的 Mark World，它记录了对象和锁有关的信息

_metadata 表示类元信息，类元信息存储的是对象指向它的类元数据(Klass)的首地址，其中 Klass 表示普通指针、_compressed_klass 表示压缩类指针

MarkWord

在 Hotspot 中，markOop 的定义在 markOop.hpp 文件

中，代码如下

```
class markOopDesc: public oopDesc {
 private:
  // Conversion
  uintptr_t value() const { return (uintptr_t) this; }
 public:
  // Constants
  enum { age_bits                 = 4,  //分代年龄
         lock_bits                = 2, //锁标识
         biased_lock_bits         = 1, //是否为偏向锁
         max_hash_bits            = BitsPerWord - age_bits - lock_bits - biased_lock_
         hash_bits                = max_hash_bits > 31 ? 31 : max_hash_bits, //对象的
         cms_bits                 = LP64_ONLY(1) NOT_LP64(0),
         epoch_bits               = 2 //偏向锁的时间戳
  };
...
```

Mark word 记录了对象和锁有关的信息，当某个对象被 synchronized 关键字当成同步锁时，那么围绕这个锁的一系列操作都和 Mark word 有关系。Mark Word 在 32 位虚拟机的长度是 32bit、在 64 位虚拟机的长度是 64bit。Mark Word 里面存储的数据会随着锁标志位的变化而变化，Mark Word 可能变化为存储以下 5 中情况

| 锁状态 | 25bit | | 4bit | 1bit | 2bit |
|---|---|---|---|---|---|
| | 23bit | 2bit | | 是否偏向锁 | 锁标志位 |
| 无锁 | 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 | 指向重量级锁的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |



为什么任何对象都可以实现锁

1. 首先，Java 中的每个对象都派生自 Object 类，而每个

Java Object 在 JVM 内部都有一个 native 的 C++对象 oop/oopDesc 进行对应。

2. 线程在获取锁的时候，实际上就是获得一个监视器对象(monitor) ,monitor 可以认为是一个同步对象，所有的 Java 对象是天生携带 monitor。在 hotspot 源码的 markOop.hpp 文件中，可以看到下面这段代码。

```
ObjectMonitor* monitor() const {
    assert(has_monitor(), "check");
    // Use xor instead of &~ to provide one extra tag-bit check.
    return (ObjectMonitor*) (value() ^ monitor_value);
  }
```

多个线程访问同步代码块时，相当于去争抢对象监视器修改对象中的锁标识,上面的代码中 ObjectMonitor 这个对象和线程争抢锁的逻辑有密切的关系

## synchronized 锁的升级

在分析 markword 时，提到了偏向锁、轻量级锁、重量级锁。在分析这几种锁的区别时，我们先来思考一个问题
使用锁能够实现数据的安全性，但是会带来性能的下降。不使用锁能够基于线程并行提升程序性能，但是却不能保证线程安全性。这两者之间似乎是没有办法达到既能满足性能也能满足安全性的要求。

hotspot 虚拟机的作者经过调查发现，大部分情况下，加锁

的代码不仅仅不存在多线程竞争，而且总是由同一个线程多次获得。所以基于这样一个概率，是的 synchronized 在 JDK1.6 之后做了一些优化，为了减少获得锁和释放锁带来的性能开销，引入了偏向锁、轻量级锁的概念。因此大家会发现在 synchronized 中，锁存在四种状态
分别是：无锁、偏向锁、轻量级锁、重量级锁； 锁的状态根据竞争激烈的程度从低到高不断升级。

**偏向锁的基本原理**

前面说过，大部分情况下，锁不仅仅不存在多线程竞争，而是总是由同一个线程多次获得，为了让线程获取锁的代价更低就引入了偏向锁的概念。怎么理解偏向锁呢?
当一个线程访问加了同步锁的代码块时，会在对象头中存储当前线程的 ID，后续这个线程进入和退出这段加了同步锁的代码块时，不需要再次加锁和释放锁。而是直接比较对象头里面是否存储了指向当前线程的偏向锁。如果相等表示偏向锁是偏向于当前线程的，就不需要再尝试获得锁了

**偏向锁的获取和撤销逻辑**

1. 首先获取锁 对象的 Markword，判断是否处于可偏向状态。(biased_lock=1、且 ThreadId 为空)

2. 如果是可偏向状态，则通过 CAS 操作，把当前线程的 ID 写入到 MarkWord
    a) 如果 cas 成功，那么 markword 就会变成这样。表示已经获得了锁对象的偏向锁，接着执行同步代码块
    b) 如果 cas 失败，说明有其他线程已经获得了偏向锁，这种情况说明当前锁存在竞争，需要撤销已获得偏向锁的线程，并且把它持有的锁升级为轻量级锁（这个操作需要等到全局安全点，也就是没有线程在执行字节码）才能执行
3. 如果是已偏向状态，需要检查 markword 中存储的 ThreadID 是否等于当前线程的 ThreadID
    a) 如果相等，不需要再次获得锁，可直接执行同步代码块
    b) 如果不相等，说明当前锁偏向于其他线程，需要撤销偏向锁并升级到轻量级锁

**偏向锁的撤销**

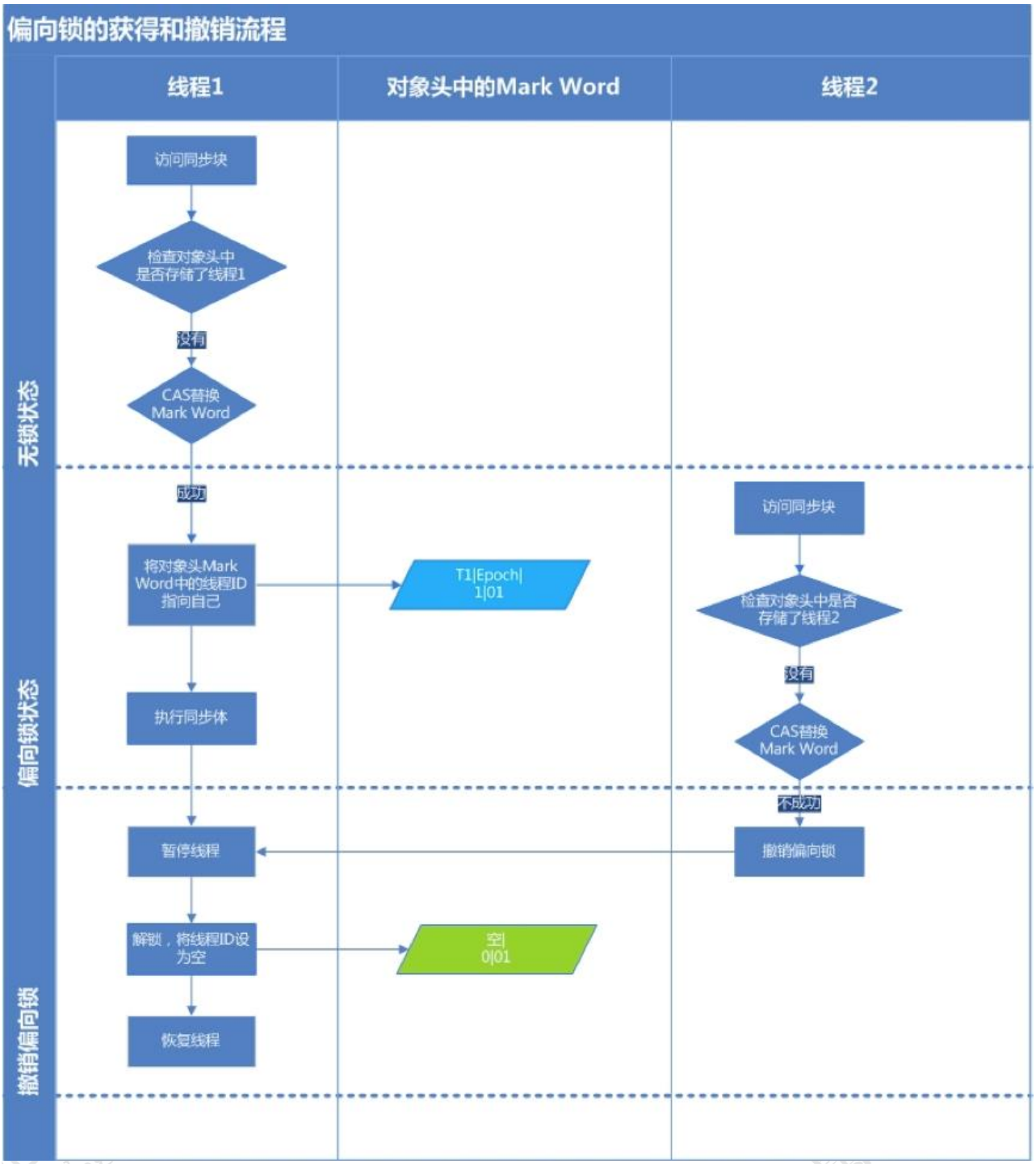
偏向锁的撤销并不是把对象恢复到无锁可偏向状态（因为偏向锁并不存在锁释放的概念），而是在获取偏向锁的过程中，发现 cas 失败也就是存在线程竞争时，直接把被偏向的锁对象升级到被加了轻量级锁的状态。

对原持有偏向锁的线程进行撤销时，原获得偏向锁的线程有两种情况：

1. 原获得偏向锁的线程如果已经退出了临界区，也就是同步代码块执行完了，那么这个时候会把对象头设置成无锁状态并且争抢锁的线程可以基于 CAS 重新偏向但前线程
2. 如果原获得偏向锁的线程的同步代码块还没执行完，处于临界区之内，这个时候会把原获得偏向锁的线程升级为轻量级锁后继续执行同步代码块

在我们的应用开发中，绝大部分情况下一定会存在 2 个以上的线程竞争，那么如果开启偏向锁，反而会提升获取锁的资源消耗。所以可以通过 jvm 参数 UseBiasedLocking 来设置开启或关闭偏向锁

## 流程图分析

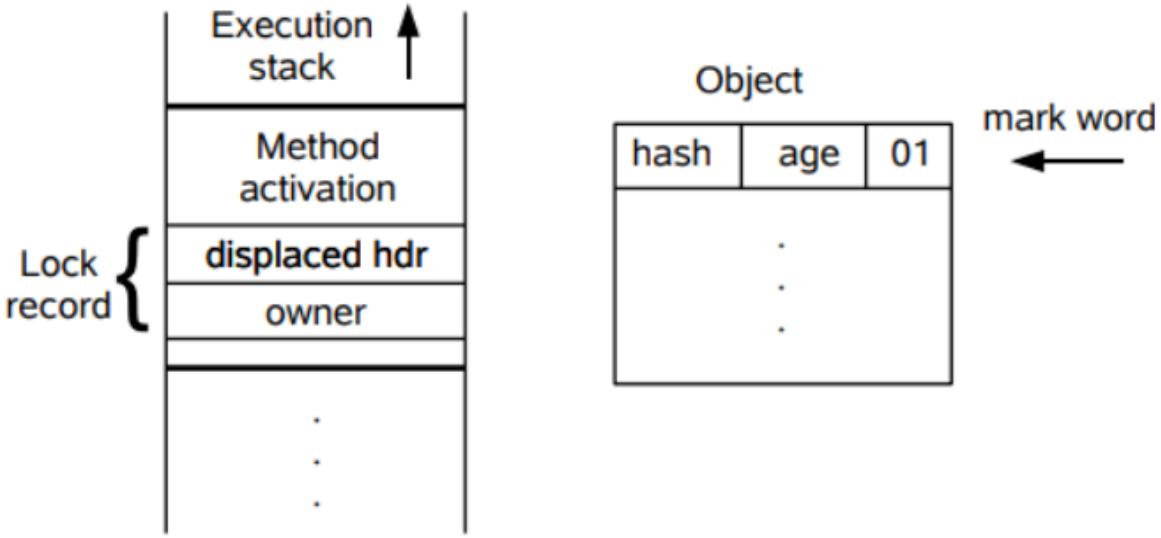

偏向锁的获得和撤销流程
线程1
对象头中的Mark Word
线程2
无锁状态
偏向锁状态
撤销偏向锁
访问同步块
检查对象头中是否存储了线程1
没有
CAS替换Mark Word
成功
将对象头Mark Word中的线程ID指向自己
T1|Epoch|1|01
执行同步体
访问同步块
检查对象头中是否存储了线程2
没有
CAS替换Mark Word
不成功
暂停线程
撤销偏向锁
解锁，将线程ID设为空
空|0|01
恢复线程

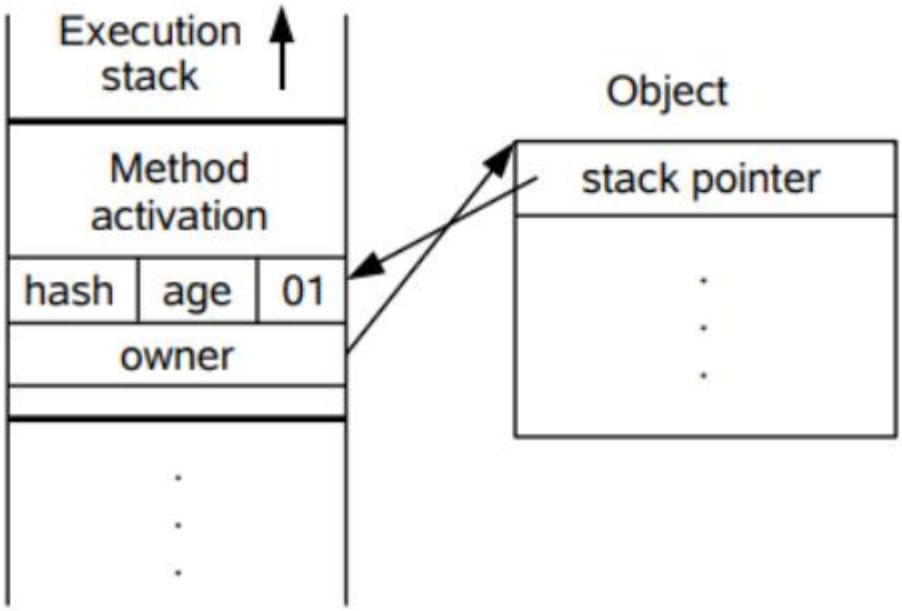
轻量级锁的基本原理

**轻量级锁的加锁和解锁逻辑**

锁升级为轻量级锁之后，对象的 Markword 也会进行相应的的变化。升级为轻量级锁的过程：

1. 线程在自己的栈桢中创建锁记录 LockRecord。
2. 将锁对象的对象头中的 MarkWord 复制到线程的刚刚创建的锁记录中。
3. 将锁记录中的 Owner 指针指向锁对象。
4. 将锁对象的对象头的 MarkWord 替换为指向锁记录的指针。






**自旋锁**

轻量级锁在加锁过程中，用到了自旋锁

所谓自旋，就是指当有另外一个线程来竞争锁时，这个线程会在原地循环等待，而不是把该线程给阻塞，直到那个获得锁的线程释放锁之后，这个线程就可以马上获得锁的。

注意，锁在原地循环的时候，是会消耗 cpu 的，就相当于在执行一个啥也没有的 for 循环。

所以，轻量级锁适用于那些同步代码块执行的很快的场景，这样，线程原地等待很短的时间就能够获得锁了。

自旋锁的使用，其实也是有一定的概率背景，在大部分同步代码块执行的时间都是很短的。所以通过看似无异议的循环反而能提升锁的性能。

但是自旋必须要有一定的条件控制，否则如果一个线程执

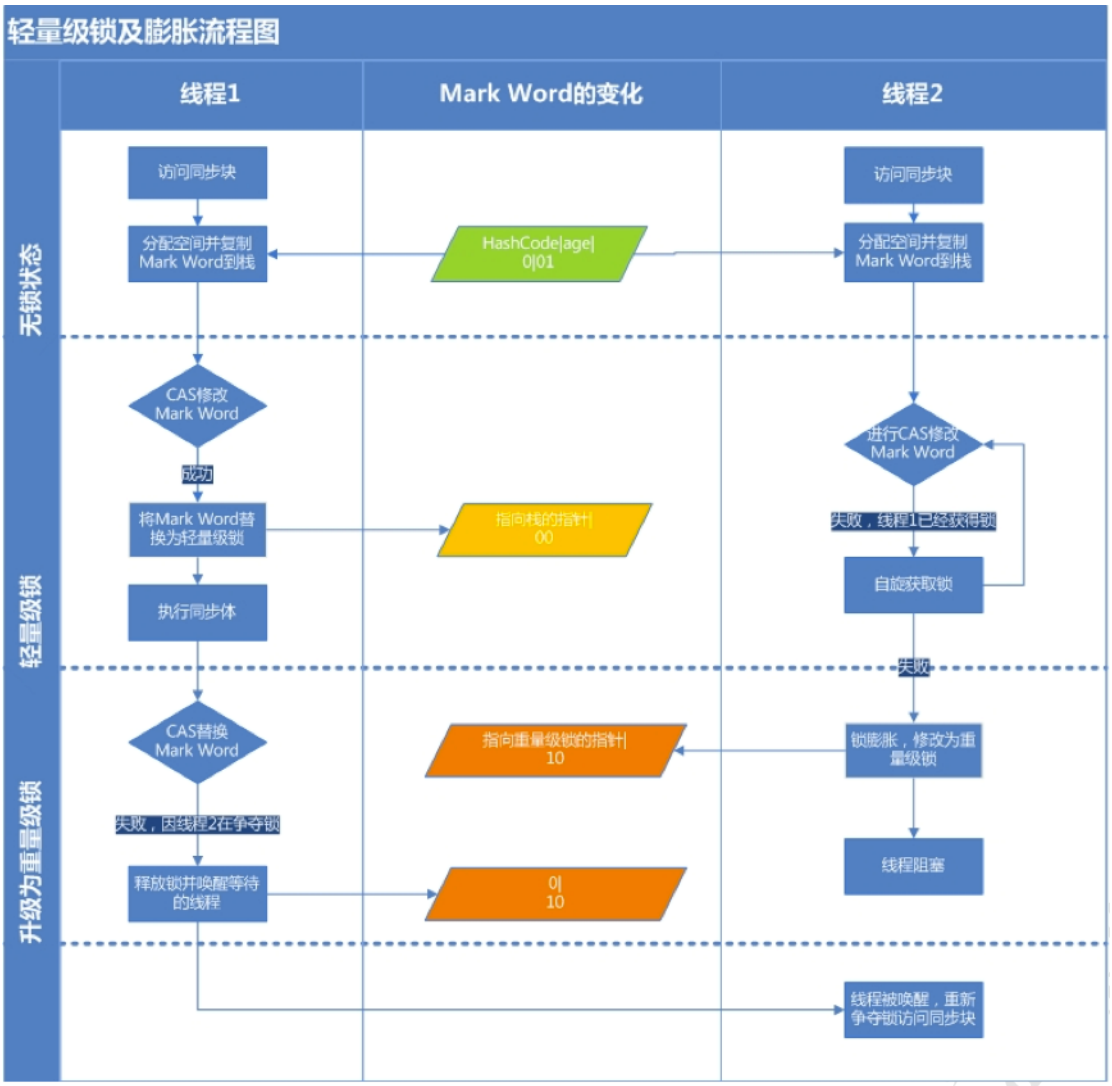
行同步代码块的时间很长，那么这个线程不断的循环反而会消耗 CPU 资源。默认情况下自旋的次数是 10 次，
可以通过 preBlockSpin 来修改

在 JDK1.6 之后，引入了自适应自旋锁，自适应意味着自旋的次数不是固定不变的，而是根据前一次在同一个锁上自旋的时间以及锁的拥有者的状态来决定。
如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源

**轻量级锁的解锁**

轻量级锁的锁释放逻辑其实就是获得锁的逆向逻辑，通过 CAS 操作把线程栈帧中的 LockRecord 替换回到锁对象的 MarkWord 中，如果成功表示没有竞争。如果失败，表示当前锁存在竞争，那么轻量级锁就会膨胀成为重量级锁

## 流程图分析




## 重量级锁的基本原理

当轻量级锁膨胀到重量级锁之后，意味着线程只能被挂起阻塞来等待被唤醒了。

## 重量级锁的 monitor

创建一个类如下

```
public class App {

    public static vo
        synchronized

        }
        test();
    }
    public static sy
}
```

运行以后通过 javap 工具查看生成的 class 文件信息分析 synchronized 关键字的实现细节

```
public static void main(java.la
  descriptor: ([Ljava/lang/Stri
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size
       0: ldc           #2
       2: dup
       3: astore_1
       4: monitorenter
       5: aload_1
       6: monitorexit
       7: goto          15
      10: astore_2
      11: aload_1
      12: monitorexit
      13: aload_2
      14: athrow
      15: invokestatic  #3
      18: return
```

| | | |
|---|---|---|
| | p -v app.class | |

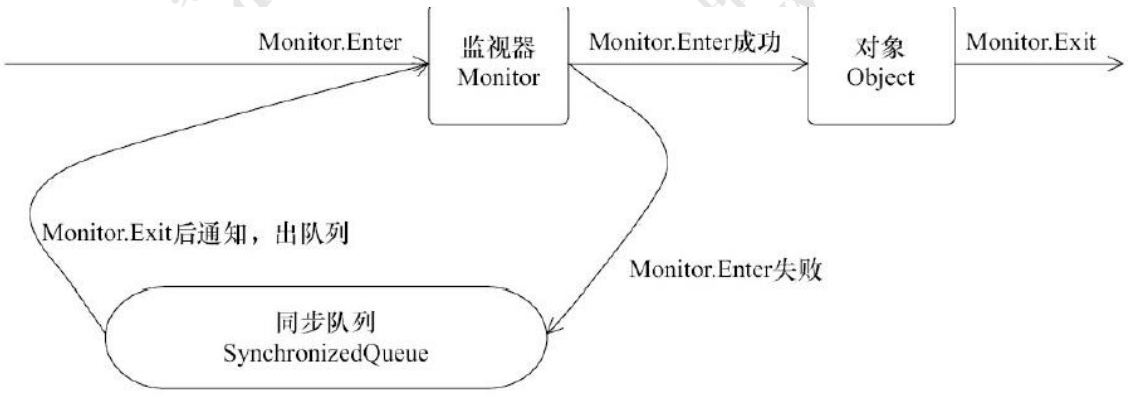
加了同步代码块以后，在字节码中会看到一个 monitorenter 和 monitorexit。

每一个 JAVA 对象都会与一个监视器 monitor 关联，我们可以把它理解成为一把锁，当一个线程想要执行一段被 synchronized 修饰的同步方法或者代码块时，该线程得先获取到 synchronized 修饰的对象对应的 monitor。

monitorenter 表示去获得一个对象监视器。monitorexit 表示释放 monitor 监视器的所有权，使得其他被阻塞的线程可以尝试去获得这个监视器

monitor 依赖操作系统的 MutexLock(互斥锁)来实现的，线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能

**重量级锁的加锁的基本流程**



任意线程对 Object（Object 由 synchronized 保护）的访问，首先要获得 Object 的监视器。如果获取失败，线程进入同步队列，线程状态变为 BLOCKED。当访问 Object 的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

回顾线程的竞争机制

再来回顾一下线程的竞争机制对于锁升级这块的一些基本流程。方便大家更好的理解
加入有这样一个同步代码块，存在 Thread#1、Thread#2 等多个线程

```
synchronized (lock) {
    // do something
}
```

情况一：只有 Thread#1 会进入临界区；
情况二：Thread#1 和 Thread#2 交替进入临界区,竞争不激烈；
情况三：Thread#1/Thread#2/Thread3... 同时进入临界区，竞争激烈

**偏向锁**

此时当 Thread#1 进入临界区时，JVM 会将 lockObject 的对象头 Mark Word 的锁标志位设为“01”，同时会用 CAS 操

作把 Thread#1 的线程 ID 记录到 Mark Word 中，此时进入偏向模式。所谓“偏向”，指的是这个锁会偏向于 Thread#1，若接下来没有其他线程进入临界区，则 Thread#1 再出入临界区无需再执行任何同步操作。也就是说，若只有 Thread#1 会进入临界区，实际上只有 Thread#1 初次进入临界区时需要执行 CAS 操作，以后再出入临界区都不会有同步操作带来的开销。

**轻量级锁**

偏向锁的场景太过于理想化，更多的时候是 Thread#2 也会尝试进入临界区， 如果 Thread#2 也进入临界区但是 Thread#1 还没有执行完同步代码块时，会暂停 Thread#1 并且升级到轻量级锁。Thread#2 通过自旋再次尝试以轻量级锁的方式来获取锁

**重量级锁**

如果 Thread#1 和 Thread#2 正常交替执行，那么轻量级锁基本能够满足锁的需求。但是如果 Thread#1 和 Thread#2 同时进入临界区，那么轻量级锁就会膨胀为重量级锁，意味着 Thread#1 线程获得了重量级锁的情况下，Thread#2 就会被阻塞

# Synchronized 结合 Java Object 对象中的 wait,notify,notifyAll

前面我们在讲 synchronized 的时候，发现被阻塞的线程什么时候被唤醒，取决于获得锁的线程什么时候执行完同步代码块并且释放锁。那怎么做到显示控制呢？我们就需要借助一个信号机制： 在 Object 对象中，提供了 wait/notify/notifyall，可以用于控制线程的状态

wait/notify/notifyall 基本概念

wait：表示持有对象锁的线程 A 准备释放对象锁权限，释放 cpu 资源并进入等待状态。

notify：表示持有对象锁的线程 A 准备释放对象锁权限，通知 jvm 唤醒某个竞争该对象锁的线程 X。线程 A synchronized 代码执行结束并且释放了锁之后，线程 X 直接获得对象锁权限，其他竞争线程继续等待(即使线程 X 同步完毕，释放对象锁，其他竞争线程仍然等待，直至有新的 notify ,notifyAll 被调用)。

notifyAll：notifyall 和 notify 的区别在于，notifyAll 会唤醒所有竞争同一个对象锁的所有线程，当已经获得锁的线程 A 释放锁之后，所有被唤醒的线程都有可能获得对象锁权限

需要注意的是:三个方法都必须在 synchronized 同步关键字所限定的作用域中调用，否则会报错 java.lang.IllegalMonitorStateException，意思是因为没有同步，所以线程对对象锁的状态是不确定的，不能调用这些方法。

另外，通过同步机制来确保线程从 wait 方法返回时能够感知到感知到 notify 线程对变量做出的修改

wait/notify 的基本使用

参见源码

wait/notify 的基本原理