# 前言

synchronized可以保证方法或代码块在运行时，同一时刻只有一个线程可以进入到临界区（互斥性），同时它还保证了共享变量的内存可见性。

Java中的每个对象都可以作为锁。

1. 普通同步方法，锁是当前实例对象。
2. 静态同步方法，锁是当前类的class对象。
3. 同步代码块，锁是括号中的对象。

# 场景

先看一个场景

**等待 / 通知机制**

直接上代码：

import java.util.concurrent.TimeUnit;

public class WaitNotify {

static boolean flag = true;

static Object lock = new Object();

public static void main(String[] args) throws InterruptedException {

Thread A = new Thread(new Wait(), "wait thread");

A.start();

TimeUnit.SECONDS.sleep(2);

Thread B = new Thread(new Notify(), "notify thread");

B.start();

}

static class Wait implements Runnable {

@Override

public void run() {

synchronized (lock) {

while (flag) {

try {

System.out.println(Thread.currentThread() + " flag is true");

lock.wait();

} catch (InterruptedException e) {

}

}

System.out.println(Thread.currentThread() + " flag is false");

}

}

}

static class Notify implements Runnable {

@Override

public void run() {

synchronized (lock) {

flag = false;

lock.notifyAll();

try {

TimeUnit.SECONDS.sleep(7);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

其相关方法定义在java.lang.Object上，线程A在获取锁后调用了对象lock的wait方法进入了等待状态，线程B调用对象lock的notifyAll()方法，线程A收到通知后从wait方法处返回继续执行，线程B对共享变量flag的修改对线程A来说是可见的。

整个运行过程需要注意以下几点：

1. 使用wait()、notify()和notifyAll()时需要先对调用对象加锁，调用wait()方法后会释放锁。
2. 调用wait()方法之后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列中。
3. notify()或notifyAll()方法调用后，等待线程不会立刻从wait()中返回，需要等该线程释放锁之后，才有机会获取锁之后从wait()返回。
4. notify()方法将等待队列中的一个等待线程从等待队列中移动到同步队列中；notifyAll()方法则是把等待队列中的所有线程都移动到同步队列中；被移动的线程状态从WAITING变为BLOCKED。
5. 从wait()方法返回的前提是，改线程获得了调用对象的锁。

那么，它是如何实现线程之间的互斥性和可见性？

# 互斥性

先看一段代码：

public class SynchronizedTest {

private static Object object = new Object();

public static void main(String[] args) throws Exception{

synchronized(object) {

}

}

public static synchronized void m() {}

}

上述代码中，使用了同步代码块和同步方法，通过使用javap工具查看生成的class文件信息来分析synchronized关键字的实现细节。

public static void main(java.lang.String[]) throws java.lang.Exception;

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=3, args\_size=1

0: getstatic #2 // Field object:Ljava/lang/Object

3: dup

4: astore\_1

5: monitorenter //监视器进入，获取锁

6: aload\_1

7: monitorexit //监视器退出，释放锁

8: goto 16

11: astore\_2

12: aload\_1

13: monitorexit

14: aload\_2

15: athrow

16: return

public static synchronized void m();

descriptor: ()V

flags: ACC\_PUBLIC, ACC\_STATIC, ACC\_SYNCHRONIZED

Code:

stack=0, locals=0, args\_size=0

0: return

LineNumberTable:

line 9: 0

从生成的class信息中，可以清楚的看到

1. 同步代码块使用了 monitorenter 和 monitorexit 指令实现。
2. 同步方法中依靠方法修饰符上的 ACC\_SYNCHRONIZED 实现。

无论哪种实现，本质上都是对指定对象相关联的monitor的获取，这个过程是互斥性的，也就是说同一时刻只有一个线程能够成功，其它失败的线程会被阻塞，并放入到同步队列中，进入BLOCKED状态。

# 锁的内部机制

我们继续深入了解一下锁的内部机制

一般锁有4种状态：无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态。

在进一步深入之前，我们先认识下两个概念：对象头和monitor。

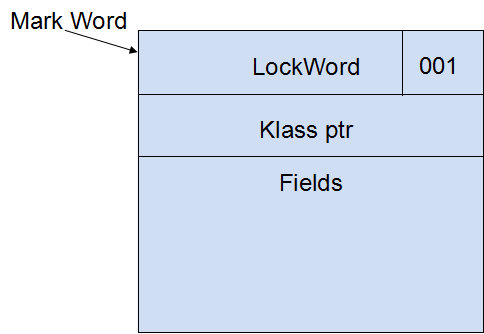
## 什么是对象头？

在hotspot虚拟机中，对象在内存的分布分为3个部分：对象头，实例数据，和对齐填充。

mark word被分成两部分，lock word和标志位。

Klass ptr指向Class字节码在虚拟机内部的对象表示的地址。

Fields表示连续的对象实例字段。



mark word 被设计为非固定的数据结构，以便在极小的空间内存储更多的信息。比如：在32位的hotspot虚拟机中：如果对象处于未被锁定的情况下。mark word 的32bit空间中有25bit存储对象的哈希码、4bit存储对象的分代年龄、2bit存储锁的标记位、1bit固定为0。而在其他的状态下（轻量级锁、重量级锁、GC标记、可偏向）下对象的存储结构为



## 什么是monitor？

monitor是线程私有的数据结构，每一个线程都有一个可用monitor列表，同时还有一个全局的可用列表，先来看monitor的内部



* Owner：初始时为NULL表示当前没有任何线程拥有该monitor，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；
* EntryQ：关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor失败的线程。
* RcThis：表示blocked或waiting在该monitor上的所有线程的个数。
* Nest：用来实现重入锁的计数。
* HashCode：保存从对象头拷贝过来的HashCode值（可能还包含GC age）。
* Candidate：用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值：0表示没有需要唤醒的线程，1表示要唤醒一个继任线程来竞争锁。

那么monitor的作用是什么呢？在 java 虚拟机中，线程一旦进入到被synchronized修饰的方法或代码块时，指定的锁对象通过某些操作将对象头中的LockWord指向monitor 的起始地址与之关联，同时monitor 中的Owner存放拥有该锁的线程的唯一标识，确保一次只能有一个线程执行该部分的代码，线程在获取锁之前不允许执行该部分的代码。

接下去，我们可以深入了解下在锁各个状态下，底层是如何处理多线程之间对锁的竞争。

## 偏向锁

下述代码中，当线程访问同步方法method1时，会在对象头（SynchronizedTest.class对象的对象头）和栈帧的锁记录中存储锁偏向的线程ID，下次该线程在进入method2，只需要判断对象头存储的线程ID是否为当前线程，而不需要进行CAS操作进行加锁和解锁（因为CAS原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本地延迟）。

public class SynchronizedTest {

private static Object lock = new Object();

public static void main(String[] args) {

method1();

method2();

}

synchronized static void method1() {}

synchronized static void method2() {}

}

## 轻量级锁

利用了CPU原语Compare-And-Swap(CAS，汇编指令CMPXCHG)。

线程可以通过两种方式锁住一个对象：

1. 通过膨胀一个处于无锁状态（状态位001）的对象获得该对象的锁；
2. 对象处于膨胀状态（状态位00），但LockWord指向的monitor的Owner字段为NULL，则可以直接通过CAS原子指令尝试将Owner设置为自己的标识来获得锁。

获取锁（monitorenter）的大概过程：

1. 对象处于无锁状态时（LockWord的值为hashCode等，状态位为001），线程首先从monitor列表中取得一个空闲的monitor，初始化Nest和Owner值为1和线程标识，一旦monitor准备好，通过CAS替换monitor起始地址到LockWord进行膨胀。如果存在其它线程竞争锁的情况而导致CAS失败，则回到monitorenter重新开始获取锁的过程即可。
2. 对象已经膨胀，monitor中的Owner指向当前线程，这是重入锁的情况（reentrant），将Nest加1，不需要CAS操作，效率高。

* 对象已经膨胀，monitor中的Owner为NULL，此时多个线程通过CAS指令试图将Owner设置为自己的标识获得锁，竞争失败的线程则进入第4种情况。
* 对象已经膨胀，同时Owner指向别的线程，在调用操作系统的重量级的互斥锁之前自旋一定的次数，当达到一定的次数如果仍然没有获得锁，则开始准备进入阻塞状态，将rfThis值原子加1，由于在加1的过程中可能被其它线程破坏对象和monitor之间的联系，所以在加1后需要再进行一次比较确保lock word的值没有被改变，当发现被改变后则要重新进行monitorenter过程。同时再一次观察Owner是否为NULL，如果是则调用CAS参与竞争锁，锁竞争失败则进入到阻塞状态。

释放锁（monitorexit）的大概过程：

1. 检查该对象是否处于膨胀状态并且该线程是这个锁的拥有者，如果发现不对则抛出异常。

* 检查Nest字段是否大于1，如果大于1则简单的将Nest减1并继续拥有锁，如果等于1，则进入到步骤3。
* 检查rfThis是否大于0，设置Owner为NULL然后唤醒一个正在阻塞或等待的线程再一次试图获取锁，如果等于0则进入到步骤4。
* 缩小（deflate）一个对象，通过将对象的LockWord置换回原来的HashCode等值来解除和monitor之间的关联来释放锁，同时将monitor放回到线程私有的可用monitor列表。

public class SynchronizedTest implements Runnable {

private static Object lock = new Object();

public static void main(String[] args) {

Thread A = new Thread(new SynchronizedTest(), "A");

A.start();

Thread B = new Thread(new SynchronizedTest(), "B");

B.start();

}

@Override

public void run() {

method1();

method2();

}

synchronized static void method1() {}

synchronized static void method2() {}

}

## 重量级锁

当锁处于这个状态下，其他线程试图获取锁都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程。

# 内存可见性

对java内存模型不熟悉的同学，可以参考这边文章[java内存模型](https://www.jianshu.com/p/d3fda02d4cae)。

线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

线程获取锁时，JMM会把该线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须从主内存中读取共享变量。

本部分内容高级进阶：[JVM源码分析之synchronized实现](https://www.jianshu.com/p/c5058b6fe8e5)