总结：

偏向锁--》轻量级锁--》重量锁。升级条件。

偏向锁加锁，解锁。

轻量级锁加锁，解锁。

重量级锁执行流程。

# Synchronized锁

能够解决线程安全问题（原子性、可见性、有序性）。

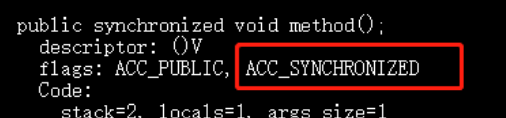
## synchronized关键字最主要的三种使用方式

* **修饰实例方法**: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁。
* **修饰静态方法**: 也就是给当前类加锁。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法（对象加锁），而线程B需要调用这个实例对象所属类的静态 synchronized 方法（类加锁），是允许的，不会发生互斥现象。
* **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。

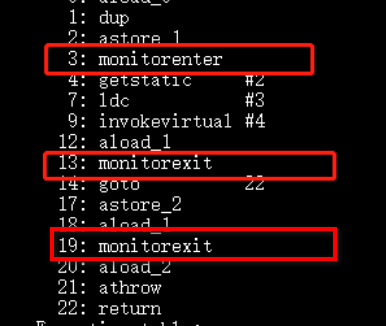
总结： synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。

## 加了synchronized修饰后，它的字节码指令（javap）是

1. 修饰方法，会在flags，出现ACC\_SYNCHRONIZED标识。



1. 修饰方法块，会在代码段中出现：Moniterenter（1个）、Moniterexit（2个）。



Moniterenter表示进入同步代码块，会去获取锁。

Moniterenter表示结束同步代码块，会去释放锁。

两个moniterenter表示：正常返回释放锁，异常返回释放锁。也就是不管如何，一定会释放锁的。

当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。

# 什么是monitor

我们可以把它理解为一个同步工具，也可以描述为一种同步机制。

所有的Java对象是天生的Monitor，每个object的对象里 markOop->monitor() 里可以保存ObjectMonitor的对象。

从源码层面分析一下monitor对象

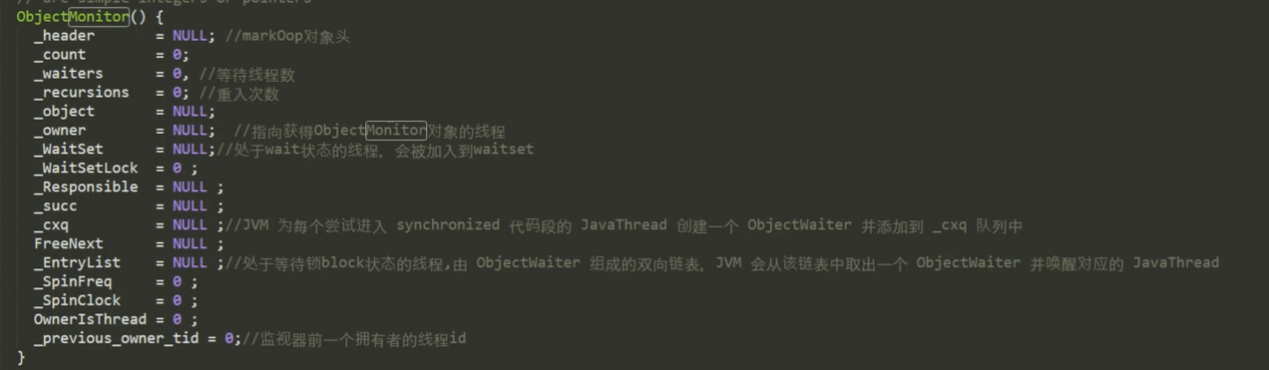
Ø oop.hpp下的oopDesc类是JVM对象的顶级基类，所以每个object对象都包含markOop

Ø markOop.hpp中markOopDesc继承自oopDesc，并扩展了自己的monitor方法，这个方法返回一个ObjectMonitor指针对象

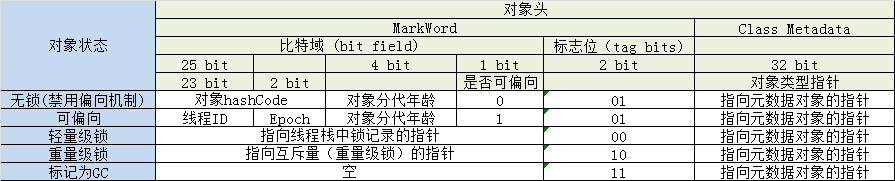
Ø objectMonitor.hpp,在hotspot虚拟机中，采用ObjectMonitor类来实现monitor，

看看markOop.hpp文件有这么一句描述：The markOop describes the header of an object.

在markOop类中有这么一个ObjectMonitor的对象。



# 对象头



第五第六行表示不同的（32、64）操作系统的各列的大小

Synchronized ，是jvm内部实现的，内部根据不同的情况，采取了不同的锁。

# 偏向锁

一个资源，只有一个线程1拥有该资源，没有其他线程来请求获取该资源（否则升级为轻量级锁）。

如果有其他线程2来访问。那么线程1执行到达全局安全点 JVM safepoint ( 此时间点， 没有线程在执行字节码) 之后，就会被挂起，然后偏向锁升级为轻量级锁，升级期间会stop the word，升级完，线程1会继续执行。

## 加锁流程

1. 如果为可偏向状态（无锁可偏向状态，锁标志位为01，是否可偏向为1,但thread id为空）,则尝试用 CAS 操作， 将自己的线程 ID 写入MarkWord
2. 如果 CAS 操作成功， 则认为已经获取到该对象的偏向锁， 执行同步块代码 。
3. 如果失败，则说明，有另外一个线程 Thread B 抢先获取了偏向锁。 这种状态说明该对象的竞争比较激烈， 此时需要撤销 Thread B 获得的偏向锁，将 Thread B 持有的锁升级为轻量级锁。 该操作需要等待全局安全点 JVM safepoint ( 此时间点， 没有线程在执行字节码) 。

4.如果为已偏向状态，（应该是看thread id 是否为空）则检测 MarkWord 中存储的 thread ID 是否等于当前 thread ID 。

如果相等， 则证明本线程已经获取到偏向锁， 可以直接继续执行同步代码块

如果不等， 则证明该对象目前偏向于其他线程， 需要做其他操作，有其他情况，具体继续看。

## 偏向锁释放

一个线程在执行完同步代码块以后， 并不会尝试将 MarkWord 中的 thread ID 赋回原值 。这样做的好处是： 如果该线程需要再次对这个对象加锁，而这个对象之前一直没有被其他线程尝试获取过锁，依旧停留在可偏向的状态下， 即可在不修改对象头的情况下， 直接认为偏向成功。

所以步骤4还得添加一种可能

如果不相等，则判断该对象是否被锁，如果被锁则应该升级为轻量锁。如果没有被锁（因为偏向锁时，同步代码块执行完是不会将thread id清空的，则再偏向如下：）

## 偏向锁的批量再偏向（Bulk Rebias）机制

那么作为开发人员， 很自然会产生的一个问题就是， 如果一个对象先偏向于某个线程， 执行完同步代码后， 另一个线程就不能直接重新获得偏向锁吗？ 答案是可以， JVM 提供了批量再偏向机制（Bulk Rebias）机制

1.引入一个概念 epoch, 其本质是一个时间戳 ， 代表了偏向锁的有效性。从前文描述的对象头结构中可以看到， epoch 存储在可偏向对象的 MarkWord 中。除了对象头中的 epoch, 对象所属的类 class 信息中， 也会保存一个 epoch 值

2.每当遇到一个全局安全点时， 如果要对 class C 进行批量再偏向， 则首先对 class C 中保存的 epoch 进行增加操作， 得到一个新的 epoch\_new。然后扫描所有持有 class C 实例的线程栈， 根据线程栈的信息判断出该线程是否锁定了该对象， 仅将 epoch\_new 的值赋给被锁定的对象中。

3.退出安全点后， 当有线程需要尝试获取偏向锁时， 直接检查 class C 中存储的 epoch 值是否与目标对象中存储的 epoch 值相等， 如果不相等，则说明没有对象被锁 说明该对象的偏向锁已经无效了， 可以尝试对此对象重新进行偏向操作。如果相等，则说明该对象的被锁着，偏向锁有效，则升级为轻量级锁。

升级为轻量级锁会stop the word

# 轻量锁

本来为偏向锁，有其他线程来获取锁，且持有锁的线程如果在升级为轻量级锁时，还未退出同步代码块。就升级为轻量级锁。

## 加锁流程

在当前线程所在栈帧中建立一个名为锁记录（lock record）的空间，然后将对象头上的mark word拷贝到锁记录中。注意：lock record有两部分，一部分存放mark word的拷贝内容，一部分是owner，它是一个指向对象头的mark word的指针。同时，对象头上的mark word的比特域变为指向lock record的指针。同时将锁标志位改为00。

加锁失败则膨胀为重量级锁。

可能失败的原因：对象的markword的比特域只能存放获取到锁的线程所在栈帧的lockrecord指针。这里应该用到cas来修改markword的比特域，

自旋转的话：cas（内存值： markword空间的地址号 ， 预期值：拷贝的值 ，修改值：打算修改的内存地址（栈的锁记录地址））

## 尝试获取锁线程视角

如果线程2尝试获取锁的时候，轻量锁正被其他线程1占有，自旋一段时间，尝试将对象头的markword指向自己，自旋超过一定时间，修改重量级锁，表示该资源应该用重量锁来锁了。

还有一个注意点：等待轻量锁的线程2不会阻塞，它会一直自旋等待锁，自旋失败后，就会将markword的比特域的指针改为指向重量级锁，修改锁标志位10,并阻塞，等待唤醒。

这就是自旋锁，尝试获取锁的线程，在没有获得锁的时候，不被挂起，而转而去执行一个空循环，即自旋。在若干个自旋后，如果还没有获得锁，则才被挂起，如果获得锁，则执行代码。

## 来自腾讯云：轻量级锁的释放

线程释放锁也是通过CAS操作完成的。当同步代码块中的代码执行完成时，虚拟机就会检查对象的Mark Word是否还在指向当前线程的锁记录。

不是，就直接释放锁的同时，唤醒被挂起的其它线程。

如果是，那么就用CAS操作把对象当前的Mark Word和线程中复制的Mark Word替换。

如果替换成功，那么整个同步代码块执行完了，也就是锁已经释放了。

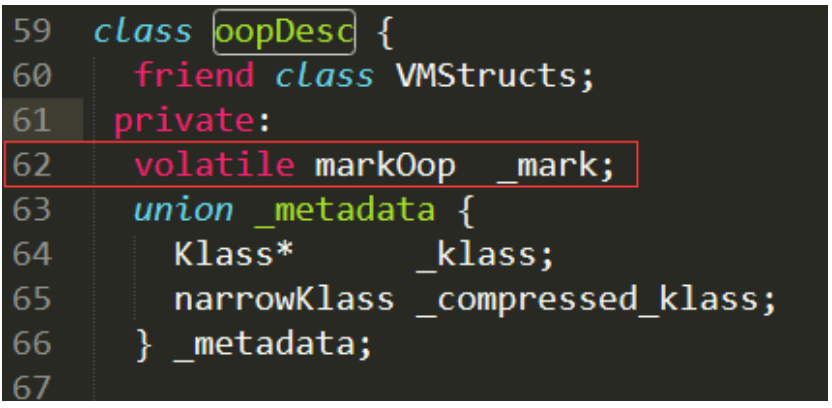
如果替换失败，那么就说明有其它线程尝试获取该锁，那么就要在释放锁的同时，唤醒被挂起的其它线程。

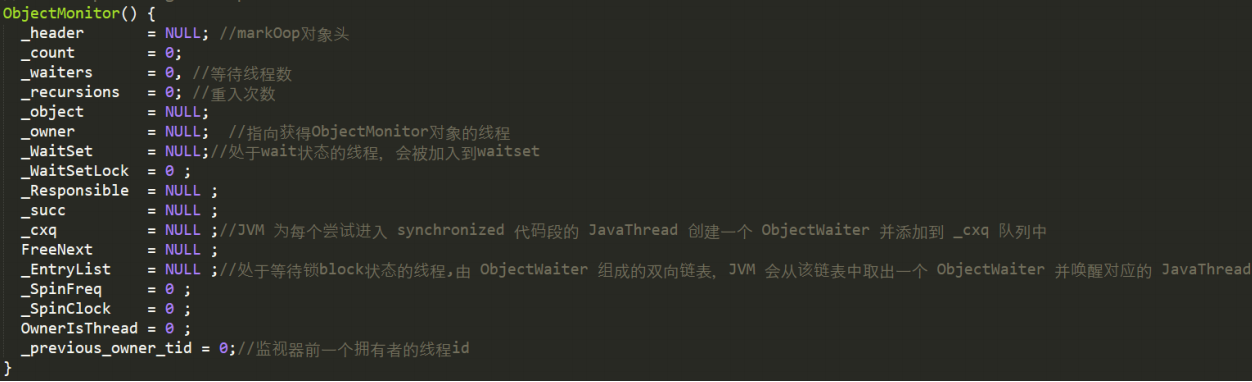
# 重量级锁

本为轻量级锁，有其他线程2来获取锁，陷入自旋，超过一定时间，还无法获取到锁，那就升为重量级锁。将对象指针指向线程2，设置锁标志位为10。

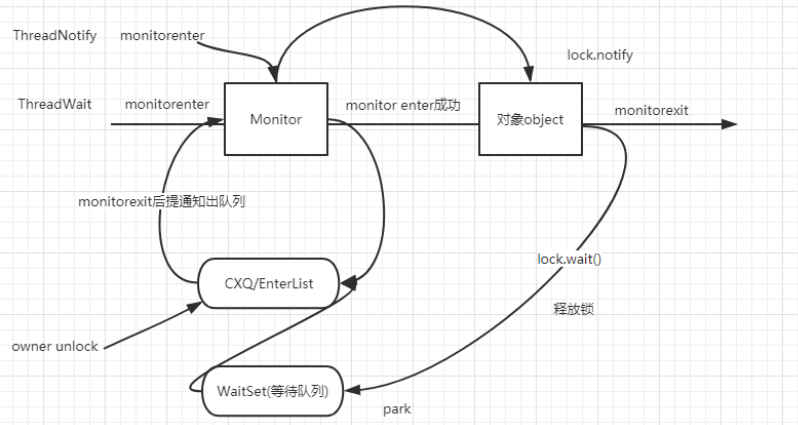
## Moniter对象

每一个对象都存在一个moniter对象，这个对象是jdk层面的，是通过C++实现的，它存在对象头中的markword，\_mark（C++的变量名）有这么一个ObjectMointer（对象）。





而synchronized锁的正是这个moniter对象，这也是为什么所有的对象都能被锁，同时也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因。因为wait和notify也需要依靠锁才能实现，而锁又是依靠这个moniter对象，具体看后面。



<https://blog.csdn.net/iteye_14104/article/details/82653465>

## 执行流程

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

◆ Contention List：所有请求锁的线程将被首先放置到该竞争队列。

◆ Entry List：Contention List中那些有资格成为候选人的线程被移到Entry List。

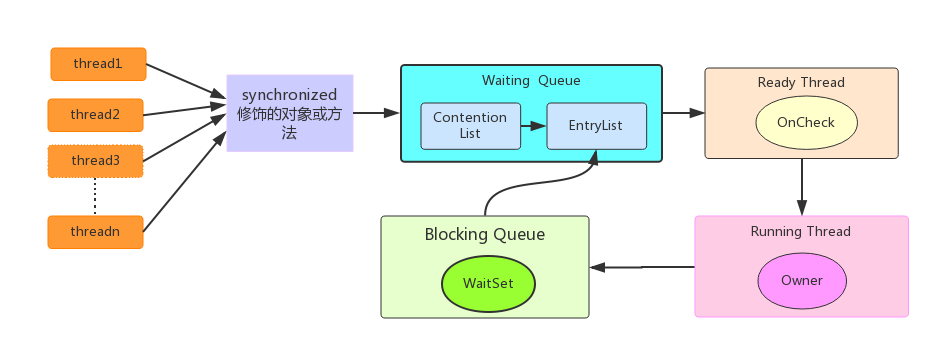
◆ Wait Set：那些调用wait方法被阻塞的线程被放置到Wait Set。

◆ OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck。

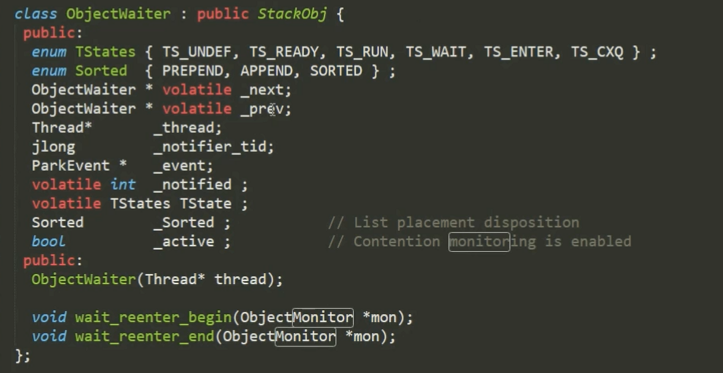
◆ Owner：获得锁的线程称为Owner。

◆ !Owner：释放锁的线程。

下图反映了个状态转换关系：



新请求锁的线程将首先被加入到ConetentionList中，当某个拥有锁的线程（Owner状态）调用unlock之后，如果发现 EntryList为空则从ContentionList中移动线程到EntryList，下面说明下ContentionList和EntryList 的实现方式：



ContentionList并不是一个真正的Queue，而只是一个虚拟队列，原因在于ContentionList是由Node（ObjectWaiter对象）及其指针逻辑构成，并不存在一个Queue的数据结构。ContentionList是一个后进先出（LIFO）的队列，每次新加入Node时都会在一端进行， 通过CAS改变第一个节点的的指针为新增节点，同时设置新增节点的next指向后续节点，而poll操作也是在同一端取出并放到EntryList中。

EntryList与ContentionList逻辑上同属等待队列，ContentionList会被线程并发访问，为了降低对 ContentionList一端的争用，而建立EntryList。Owner线程在unlock时会从ContentionList中迁移线程到 EntryList，并会指定EntryList中的某个线程（一般为Head）为Ready（OnDeck）线程。Owner线程并不是把锁传递给 OnDeck线程，只是把竞争锁的权利交给OnDeck，OnDeck线程需要重新竞争锁。

OnDeck线程并不一定能获取到锁，因为Synchronized是非公平锁。在线程进入ContentionList时，会先尝试自旋获取锁，自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。如果获取不到，才会挂起（park），这明显对于已经进入队列的线程是不公平的。

这样做虽然牺牲了一定的公平性，但极大的提高了整体吞吐量，在 Hotspot中把OnDeck的选择行为称之为“竞争切换”。

OnDeck线程获得锁后即变为owner线程，无法获得锁则会依然留在EntryList中，考虑到公平性，在EntryList中的位置不发生变化（依然在队头）。如果Owner线程被wait方法阻塞，则转移到WaitSet队列；如果在某个时刻被notify/notifyAll唤醒， 则再次转移到EntryList。

重量级锁依赖于操作系统的互斥量（mutex） 实现，此处暂时只需要了解该操作会导致进程从用户态与内核态之间的切换（1.6之前synchronized都是这样）， 是一个开销较大的操作。

# 自旋锁

因为锁的状态是变化很频繁的，有可能当前线程没获取到锁，但是如果在接下来的很短时间内锁会被释放，而如果该线程采用：在没有获取到锁的时候便进行挂起，等待唤醒。这是很耗费性能的。

所以引入自旋锁：在没有获取到锁时，不断地尝试获取锁，达到一定次数，挂起。不过自旋次数过多，也会导致cpu占用，所以应该在两者之间权衡，也就是自旋一定次数，还没获取到锁，就挂起，这个次数的值jvm内部自己设置的。

很多场景会用到。

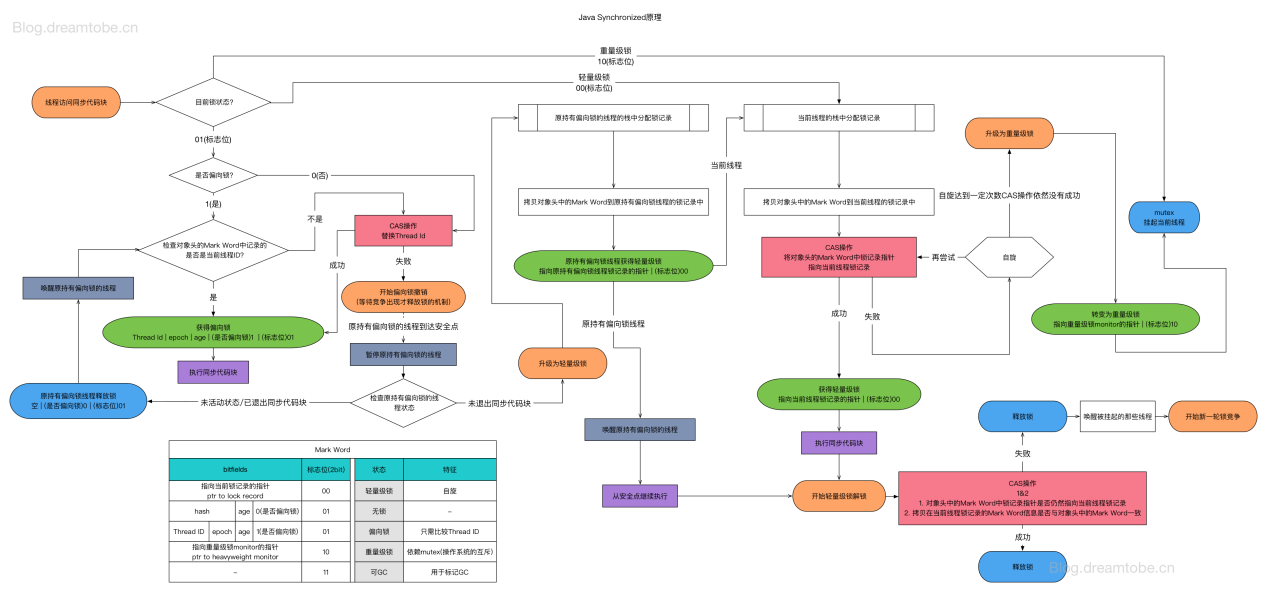
# 总结

偏向锁：只有一个线程。

轻量锁：可以有多个线程，不过没有获取到锁的线程在自旋期间内获取到锁。

重量锁：持有锁的线程执行时间长，其他线程自旋到期，还无法获取到锁。

# synchronized的执行过程



上面几种锁都是JVM自己内部实现，当我们执行synchronized同步块的时候jvm会根据启用的锁和当前线程的争用情况，决定如何执行同步操作；

在所有的锁都启用的情况下线程进入临界区时会先去获取偏向锁，如果已经存在偏向锁了，则会尝试获取轻量级锁，启用自旋锁，如果自旋也没有获取到锁，则使用重量级锁，没有获取到锁的线程阻塞挂起，直到持有锁的线程执行完同步块唤醒他们；

偏向锁是在无锁争用的情况下使用的，也就是同步开在当前线程没有执行完之前，没有其它线程会执行该同步块，一旦有了第二个线程的争用，偏向锁就会升级为轻量级锁，如果轻量级锁自旋到达阈值后，没有获取到锁，就会升级为重量级锁；

如果线程争用激烈，那么应该禁用偏向锁。

# 锁消除

public static void test(){

//局部变量不可能出现线程安全问题，所以将锁消除了。

StringBuffer stringBuffer = new StringBuffer();

//append被synchronized修饰

stringBuffer.append("a");

stringBuffer.append("b");

stringBuffer.append("c");

}

# 锁粗化

public class Test {

private static StringBuffer stringBuffer = new StringBuffer();

public static void main(String[] args) {

//append被synchronized修饰

//在第一次append方法进行加锁，最后一个append结束后进行解锁

stringBuffer.append("a");

stringBuffer.append("b");

stringBuffer.append("c");

}

}

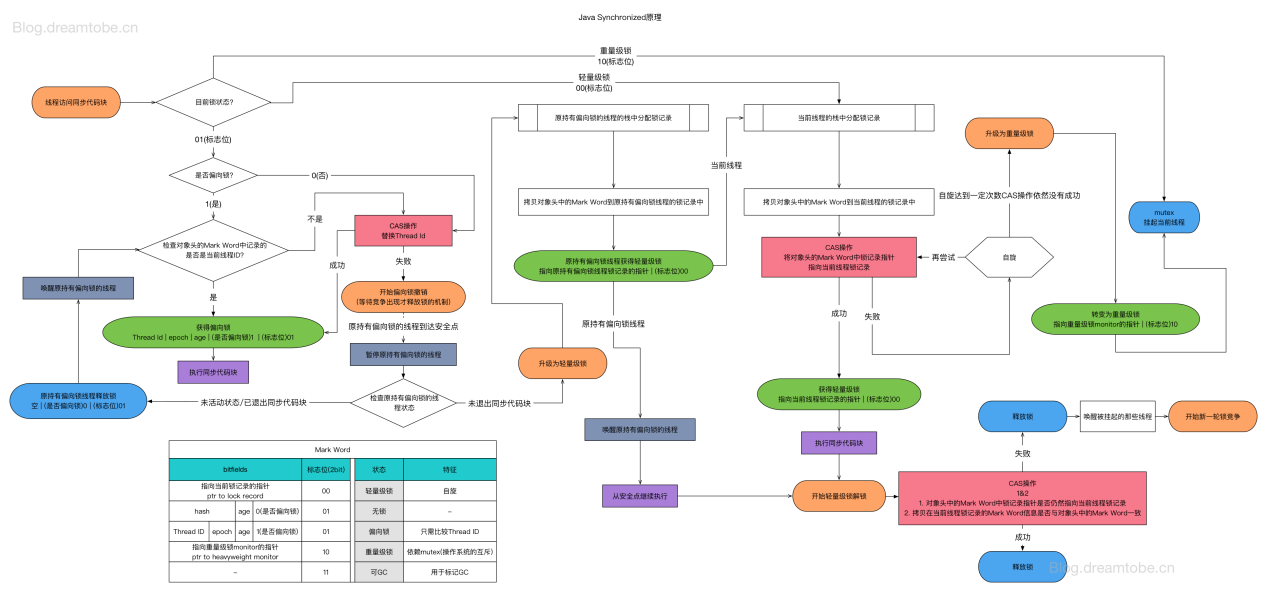
参考：

<https://blog.csdn.net/zqz_zqz/article/details/70233767> 7W

<https://blog.csdn.net/lengxiao1993/article/details/81568130> 1w

这两篇一起看

# Synchronized锁的流程图



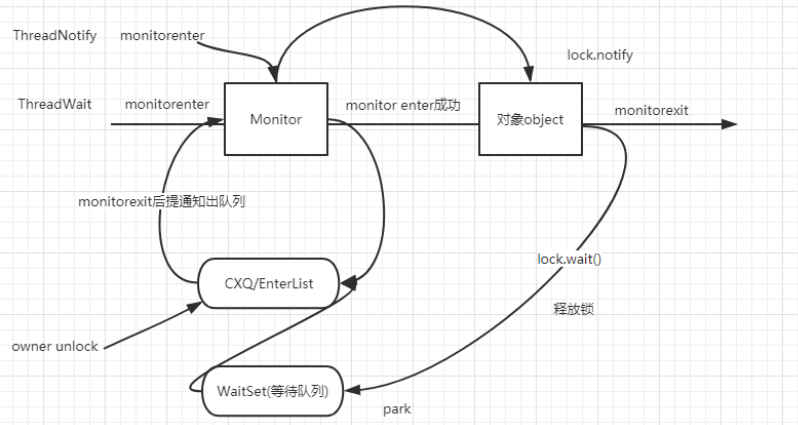
# wait和notify

|  |
| --- |
| public class ThreadWait extends Thread{  private Object lock;  public ThreadWait(Object lock) {  this.lock = lock;  }  @Override  public void run() {  synchronized (lock){  System.out.println("开始执行 thread wait");  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("执行结束 thread wait");  }  }  } |

|  |
| --- |
| public class ThreadNotify extends Thread{  private Object lock;  public ThreadNotify(Object lock) {  this.lock = lock;  }  @Override  public void run() {  synchronized (lock){  System.out.println("开始执行 thread notify");  lock.notify();  System.out.println("执行结束 thread notify");  }  }  } |

# wait和notify的原理

调用wait方法前，首先会获取监视器锁，获得成功以后，会让当前线程进入等待状态进入等待队列并且释放锁；然后 当其他线程调用notify或者notifyall以后，会选择从等待队列中唤醒任意一个线程，而执行完notify方法以后，并不会立马唤醒线程。原因是当前的线程仍然持有这把锁，处于等待状态的线程无法获得锁。必须要等到当前的线程执行完按monitorexit指令以后，也就是锁被释放以后，处于等待队列中的线程就可以开始竞争锁了。



# wait和notify为什么需要在synchronized里面

因为wait和notify之间需要建立关系，不然，如何实现你想要的被wait线程，能够准确地被notify，而不会notify其他线程。所以需要通过锁来实现。

所以wait和notify都必须基于同一个锁来实现。

所以wait必须要获得一个监视器锁。而且wait了之后会被放到锁对应的等待队列中。 而且只有wait释放了锁，notify的线程才能获取这个锁，通过这个锁找到所有在这个锁睡眠的所有线程，才能去notify。这也是为什么wait会释放锁，而sleep不用。

而对于notify来说也是一样，它是唤醒一个线程，既然要去唤醒，首先得知道这个线程在哪里？所以就必须要找到这个对象获取到这个对象的锁，然后到这个对象的等待队列中去唤醒一个线程。