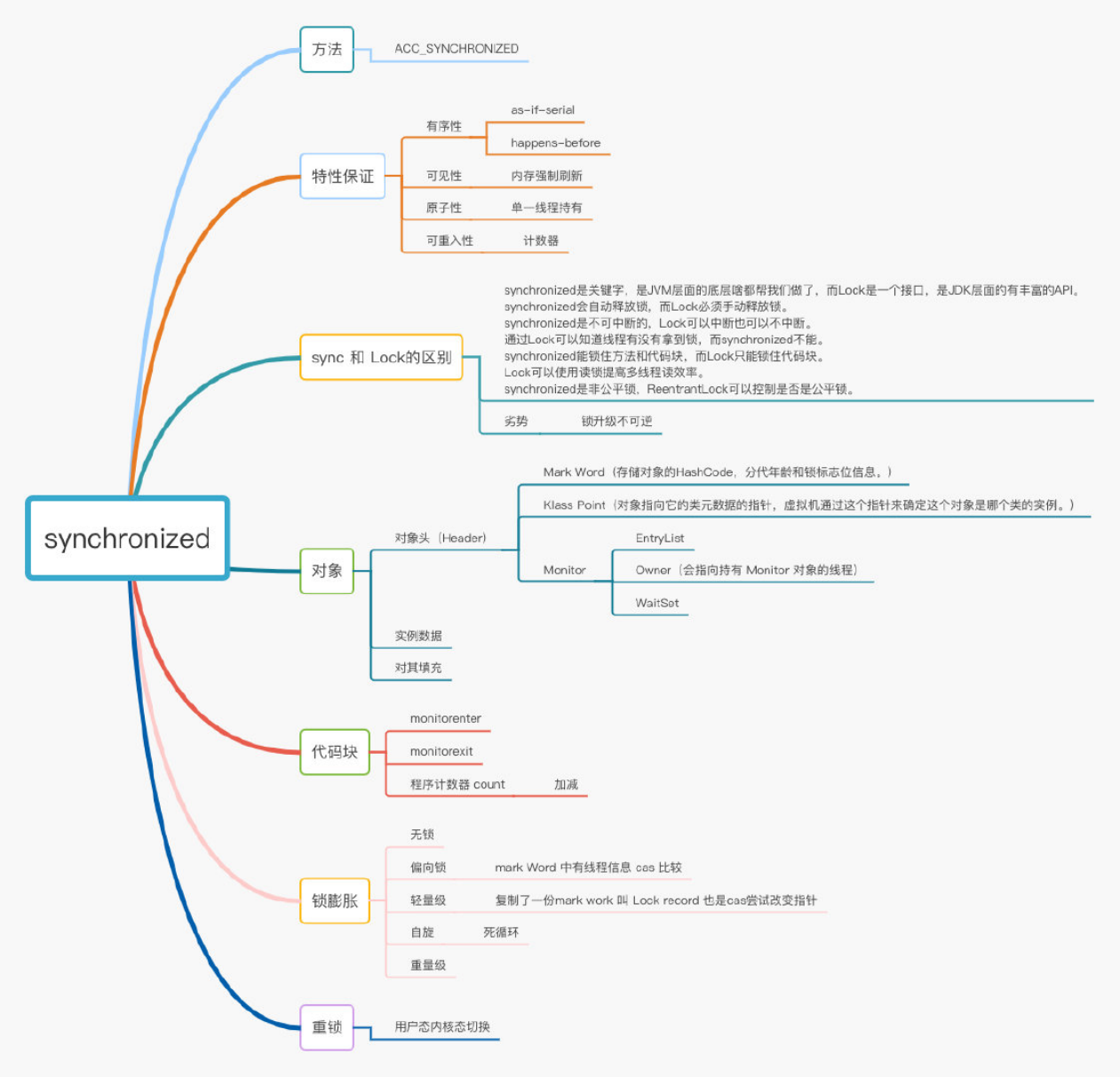
> 点赞再看，养成习惯，微信搜索【**三太子敖丙**】第一时间阅读。
>
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言



多线程的东西很多，也很有意思，所以我最近的重心可能都是多线程的方向去靠了，不知道大家喜欢否？

阅读本文之前阅读以下两篇文章会帮助你更好的理解：

Volatile

乐观锁&悲观锁

正文

场景

我们正常去使用Synchronized一般都是用在下面这几种场景：

- 修饰实例方法，对当前实例对象this加锁

```
public class Synchronized {
    public synchronized void husband(){

    }
}
```

- 修饰静态方法，对当前类的Class对象加锁

```
public class Synchronized {
    public void husband(){
        synchronized(Synchronized.class){

        }
    }
}
```

- 修饰代码块，指定一个加锁的对象，给对象加锁

```
public class Synchronized {
    public void husband(){
        synchronized(new test()){

        }
    }
}
```

其实就是锁方法、锁代码块和锁对象，那他们是怎么实现加锁的呢？

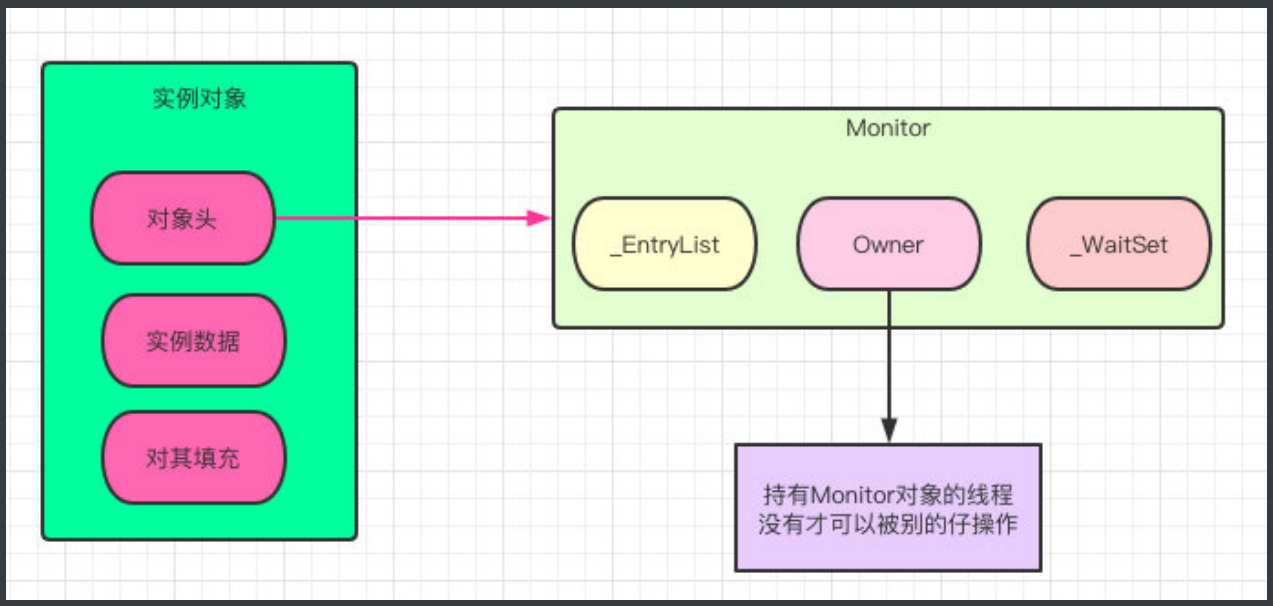
在这之前，我就先跟大家聊一下我们Java对象的构成

在 JVM 中，对象在内存中分为三块区域：

- 对象头
  - **Mark Word（标记字段）**：默认存储对象的HashCode，分代年龄和锁标志位信息。它会根据对象的状态复用自己的存储空间，也就是说在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。

    - **Klass Point（类型指针）**：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。
- 实例数据
    - 这部分主要是存放类的数据信息，父类的信息。
- 对其填充
    - 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。

      Tip：不知道大家有没有被问过一个空对象占多少个字节？就是8个字节，是因为对齐填充的关系哈，不到8个字节对其填充会帮我们自动补齐。



我们经常说到的，有序性、可见性、原子性，synchronized又是怎么做到的呢?

有序性

我在Volatile章节已经说过了CPU会为了优化我们的代码，会对我们程序进行重排序。

as-if-serial

不管编译器和CPU如何重排序，必须保证在单线程情况下程序的结果是正确的，还有就是有数据依赖的也是不能重排序的。

就比如：

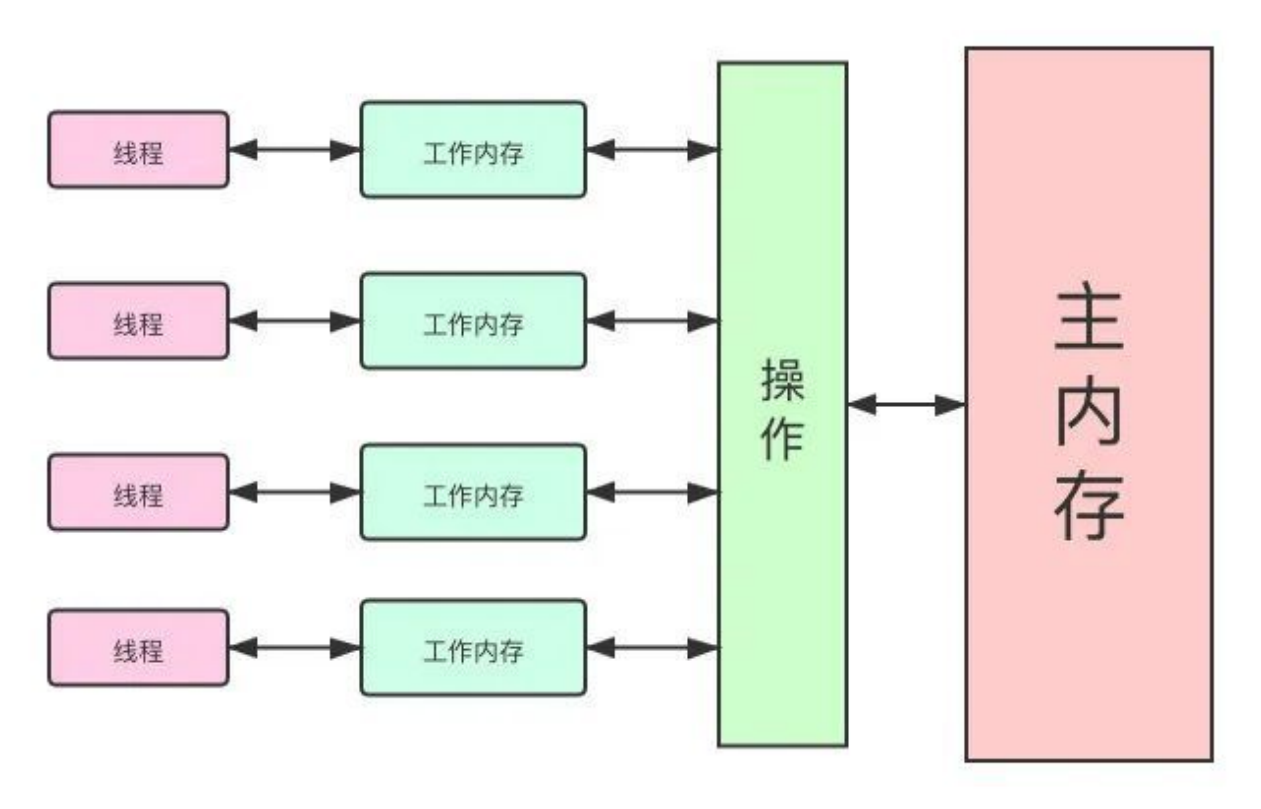
```
int a = 1;
int b = a;
```

这两段是怎么都不能重排序的，b的值依赖a的值，a如果不先赋值，那就为空了。

可见性

同样在Volatile章节我介绍到了现代计算机的内存结构，以及JMM（Java内存模型），这里我需要说明一下就是JMM并不是实际存在的，而是一套规范，这个规范描述了很多java程序中各种变量（线程共享变量）的访问规则，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节，Java内存模型是对共享数据的可见性、有序性、和原子性的规则和保障。

大家感兴趣，也记得去了解计算机的组成部分，cpu、内存、多级缓存等，会帮助更好的理解java这么做的原因。



原子性

其实他保证原子性很简单，确保同一时间只有一个线程能拿到锁，能够进入代码块这就够了。

这几个是我们使用锁经常用到的特性，那synchronized他自己本身又具有哪些特性呢?

可重入性

synchronized锁对象的时候有个计数器，他会记录下线程获取锁的次数，在执行完对应的代码块之后，计数器就会-1，直到计数器清零，就释放锁了。

那可重入有什么好处呢?

可以避免一些死锁的情况，也可以让我们更好封装我们的代码。

不可中断性

不可中断就是指，一个线程获取锁之后，另外一个线程处于阻塞或者等待状态，前一个不释放，后一个也一直会阻塞或者等待，不可以被中断。

值得一提的是，Lock的tryLock方法是可以被中断的。

底层实现

这里看实现很简单，我写了一个简单的类，分别有锁方法和锁代码块，我们反编译一下字节码文件，就可以了。

先看看我写的测试类：

```
/**
 *@Description: Synchronize
 *@Author: 敖丙
 *@date: 2020-05-17
 **/
public class Synchronized {
    public synchronized void husband(){
        synchronized(new Volatile()){

        }
    }
}
```

编译完成，我们去对应目录执行 javap -c xxx.class 命令查看反编译的文件：

```
MacBook-Pro-3:juc aobing$ javap -p -v -c Synchronized.class
Classfile
/Users/aobing/IdeaProjects/Thanos/laogong/target/classes/juc/Synchronized.c
lass
  Last modified 2020-5-17; size 375 bytes
  MD5 checksum 4f5451a229e80c0a6045b29987383d1a
  Compiled from "Synchronized.java"
public class juc.Synchronized
  minor version: 0
  major version: 49
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #3.#14         // java/lang/Object."<init>":()V
   #2 = Class              #15            // juc/Synchronized
   #3 = Class              #16            // java/lang/Object
   #4 = Utf8               <init>
   #5 = Utf8               ()V
   #6 = Utf8               Code
   #7 = Utf8               LineNumberTable
```

```
   #8 = Utf8               LocalVariableTable
   #9 = Utf8               this
  #10 = Utf8               Ljuc/Synchronized;
  #11 = Utf8               husband
  #12 = Utf8               SourceFile
  #13 = Utf8               Synchronized.java
  #14 = NameAndType        #4:#5          // "<init>":()V
  #15 = Utf8               juc/Synchronized
  #16 = Utf8               java/lang/Object
{
  public juc.Synchronized();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 8: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Ljuc/Synchronized;

  public synchronized void husband();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED  // 这里
    Code:
      stack=2, locals=3, args_size=1
         0: ldc           #2                  // class juc/Synchronized
         2: dup
         3: astore_1
         4: monitorenter   // 这里
         5: aload_1
         6: monitorexit    // 这里
         7: goto          15
        10: astore_2
        11: aload_1
        12: monitorexit    // 这里
        13: aload_2
        14: athrow
        15: return
      Exception table:
         from    to  target type
```

```
                5     7    10   any
               10    13    10   any
        LineNumberTable:
          line 10: 0
          line 12: 5
          line 13: 15
        LocalVariableTable:
          Start  Length  Slot  Name   Signature
              0      16     0  this   Ljuc/Synchronized;
    }
    SourceFile: "Synchronized.java"
```

同步代码

大家可以看到几处我标记的，我在最开始提到过对象头，他会关联到一个monitor对象。

- 当我们进入一个人方法的时候，执行**monitorenter**，就会获取当前对象的一个所有权，这个时候monitor进入数为1，当前的这个线程就是这个monitor的owner。
- 如果你已经是这个monitor的owner了，你再次进入，就会把进入数+1.
- 同理，当他执行完**monitorexit**，对应的进入数就-1，直到为0，才可以被其他线程持有。

所有的互斥，其实在这里，就是看你能否获得monitor的所有权，一旦你成为owner就是获得者。

同步方法

不知道大家注意到方法那的一个特殊标志位没，**ACC_SYNCHRONIZED**。

同步方法的时候，一旦执行到这个方法，就会先判断是否有标志位，然后，ACC_SYNCHRONIZED会去隐式调用刚才的两个指令：monitorenter和monitorexit。

所以归根究底，还是monitor对象的争夺。

monitor

我说了这么多次这个对象，大家是不是以为就是个虚无的东西，其实不是，monitor监视器源码是C++写的，在虚拟机的ObjectMonitor.hpp文件中。

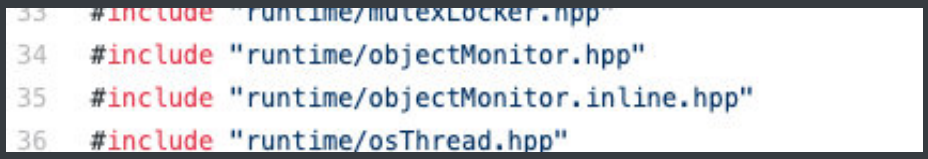
我看了下源码，他的数据结构长这样：

```
    ObjectMonitor() {
      _header       = NULL;
      _count        = 0;
      _waiters      = 0,
      _recursions   = 0;  // 线程重入次数
      _object       = NULL;  // 存储Monitor对象
```

```
    _owner          = NULL;  // 持有当前线程的owner
    _WaitSet        = NULL;  // wait状态的线程列表
    _WaitSetLock    = 0 ;
    _Responsible    = NULL ;
    _succ           = NULL ;
    _cxq            = NULL ;  // 单向列表
    FreeNext        = NULL ;
    _EntryList      = NULL ;  // 处于等待锁状态block状态的线程列表
    _SpinFreq       = 0 ;
    _SpinClock      = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
  }
```

这块c++代码，我也放到了我的开源项目了，大家自行查看。

synchronized底层的源码就是引入了ObjectMonitor，这一块大家有兴趣可以看看，反正我上面说的，还有大家经常听到的概念，在这里都能找到源码。



大家说熟悉的锁升级过程，其实就是在源码里面，调用了不同的实现去获取获取锁，失败就调用更高级的实现，最后升级完成。

1.5 重量级锁

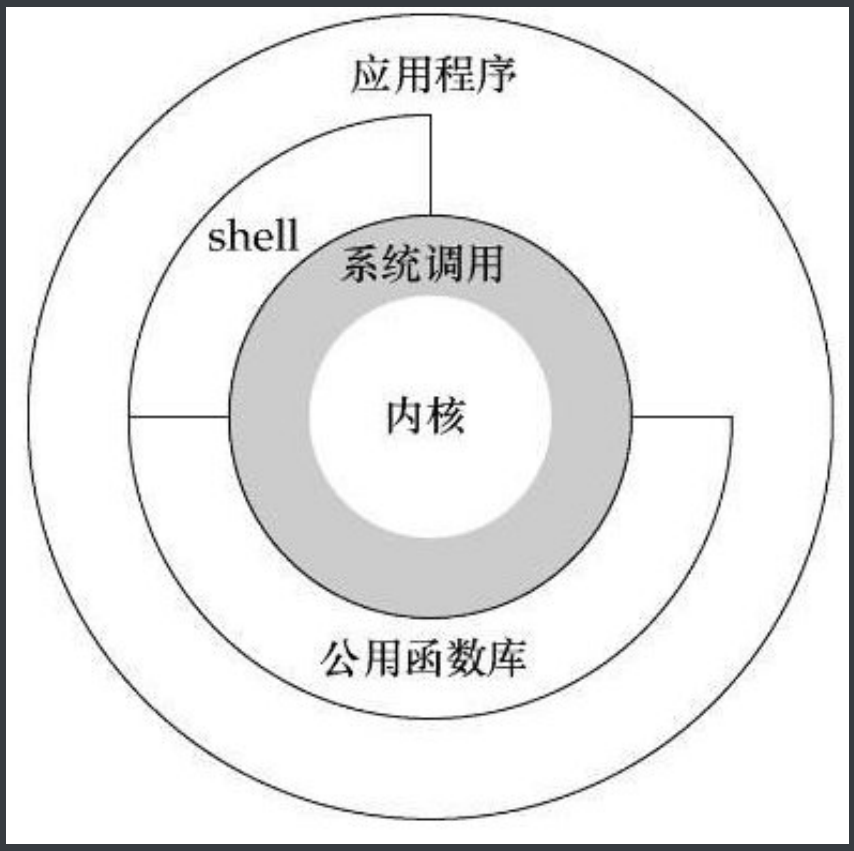
大家在看ObjectMonitor源码的时候，会发现Atomic::cmpxchg_ptr，Atomic::inc_ptr等内核函数，对应的线程就是park()和upark()。

这个操作涉及用户态和内核态的转换了，这种切换是很耗资源的，所以知道为啥有自旋锁这样的操作了吧，按道理类似死循环的操作更费资源才是对吧？其实不是，大家了解一下就知道了。

那用户态和内核态又是啥呢?

Linux系统的体系结构大家大学应该都接触过了，分为用户空间（应用程序的活动空间）和内核。

我们所有的程序都在用户空间运行，进入用户运行状态也就是（用户态），但是很多操作可能涉及内核运行，比我I/O，我们就会进入内核运行状态（内核态）。



这个过程是很复杂的，也涉及很多值的传递，我简单概括下流程：

1. 用户态把一些数据放到寄存器，或者创建对应的堆栈，表明需要操作系统提供的服务。
2. 用户态执行系统调用（系统调用是操作系统的最小功能单位）。
3. CPU切换到内核态，跳到对应的内存指定的位置执行指令。
4. 系统调用处理器去读取我们先前放到内存的数据参数，执行程序的请求。
5. 调用完成，操作系统重置CPU为用户态返回结果，并执行下个指令。

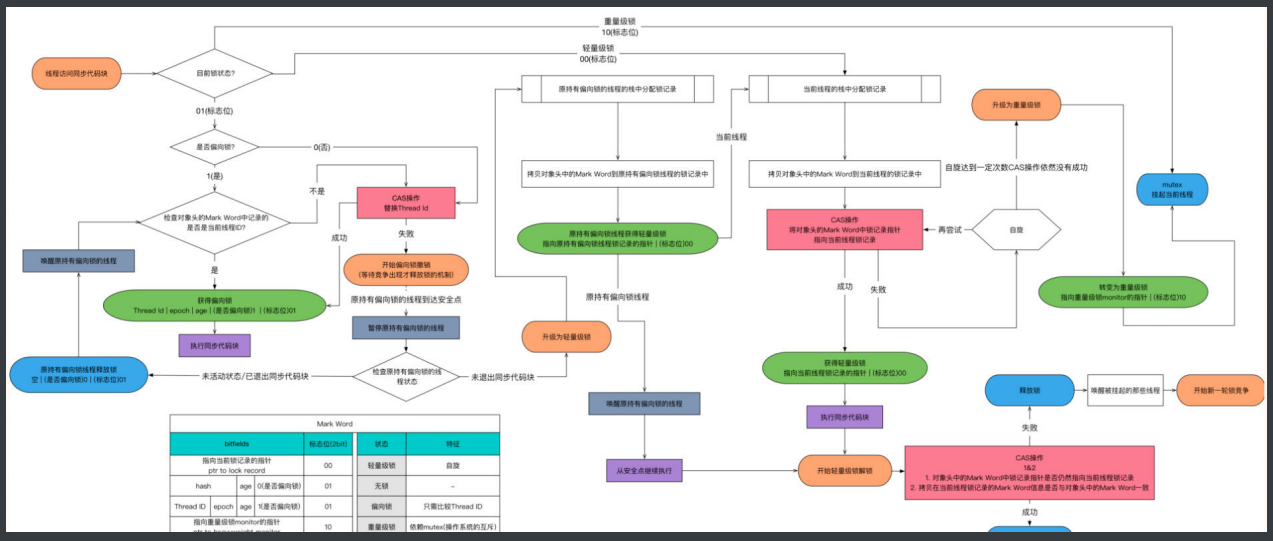
所以大家一直说，1.6之前是重量级锁，没错，但是他重量的本质，是ObjectMonitor调用的过程，以及Linux内核的复杂运行机制决定的，大量的系统资源消耗，所以效率才低。

还有两种情况也会发生内核态和用户态的切换：异常事件和外围设备的中断 大家也可以了解下。

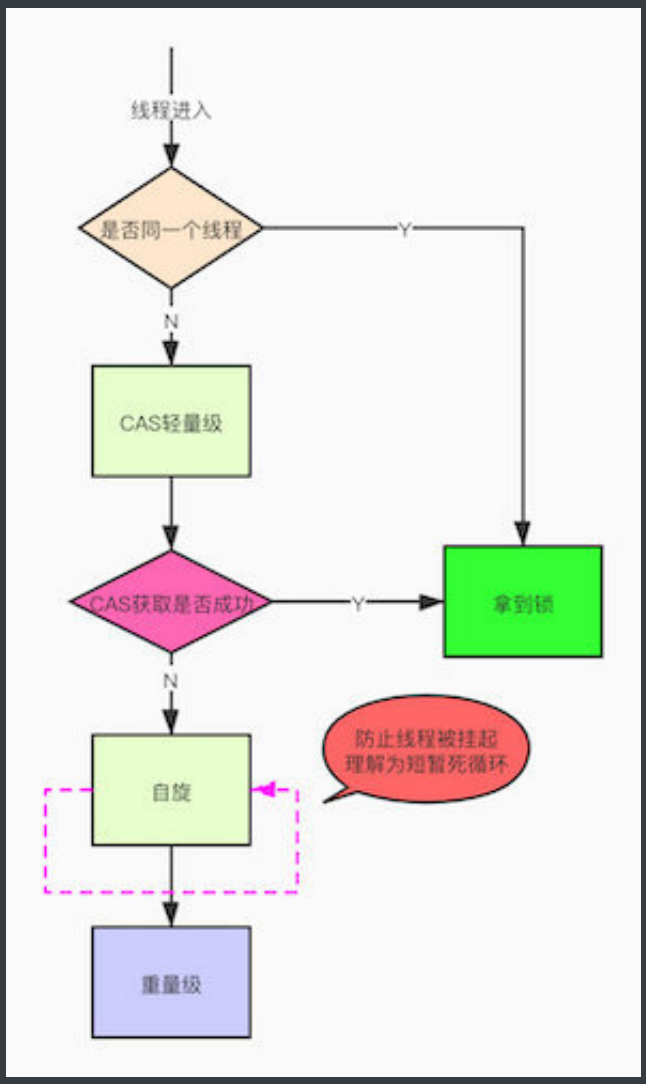
1.6 优化锁升级

那都说过了效率低，官方也是知道的，所以他们做了升级，大家如果看了我刚才提到的那些源码，就知道他们的升级其实也做得很简单，只是多了几个函数调用，不过不得不设计还是很巧妙的。

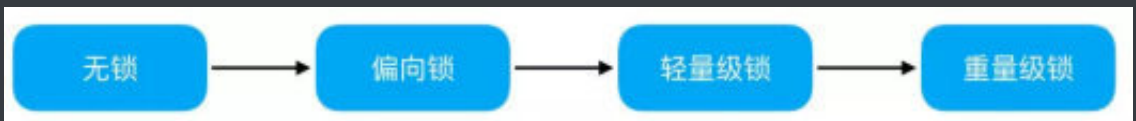
我们就来看一下升级后的锁升级过程：


重量级锁
10(标志位)
轻量级锁
00(标志位)
线程访问同步代码块
目前锁状态?
01(标志位)
是否偏向锁?
0(否)
1(是)
不是
是
检查对象头的Mark Word中记录的是否是当前线程ID?
CAS操作
替换Thread Id
成功
失败
唤醒原持有偏向锁的线程
开始偏向锁撤销
(等待竞争出现才释放锁的机制)
获得偏向锁
Thread Id | epoch | age | (是否偏向锁)1 | (标志位)01
原持有偏向锁的线程到达安全点
执行同步代码块
暂停原持有偏向锁的线程
原持有偏向锁线程释放锁
空 | (是否偏向锁)0 | (标志位)01
未活动状态/已退出同步代码块
检查原持有偏向锁的线程状态
未退出同步代码块
升级为轻量级锁
原持有偏向锁线程
唤醒原持有偏向锁的线程
从安全点继续执行
原持有偏向锁的线程的栈中分配锁记录
拷贝对象头中的Mark Word到原持有偏向锁线程的锁记录中
原持有偏向锁线程获得轻量级锁
指向原持有偏向锁线程锁记录的指针 | (标志位)00
当前线程的栈中分配锁记录
当前线程
拷贝对象头中的Mark Word到当前线程的锁记录中
CAS操作
将对象头的Mark Word中锁记录指针
指向当前线程锁记录
再尝试
成功
失败
获得轻量级锁
指向当前线程锁记录的指针 | (标志位)00
执行同步代码块
开始轻量级锁解锁
CAS操作
1&2
1. 对象头中的Mark Word中锁记录指针是否仍然指向当前线程锁记录
2. 拷贝在当前线程锁记录的Mark Word信息是否与对象头中的Mark Word一致
成功
自旋达到一定次数CAS操作依然没有成功
自旋
升级为重量级锁
mutex
挂起当前线程
转变为重量级锁
指向重量级锁monitor的指针 | (标志位)10
释放锁
唤醒被挂起的那些线程
开始新一轮锁竞争
失败
Mark Word
bitfields
标志位(2bit)
状态
特征
指向当前锁记录的指针
ptr_to_lock_record
00
轻量级锁
自旋
hash
age
0(是否偏向锁)
01
无锁
-
Thread ID
epoch
age
1(是否偏向锁)
01
偏向锁
只需比较Thread ID
指向重量级锁monitor的指针
10
重量级锁
依赖mutex(操作系统的互斥)

简单版本：


线程进入
是否同一个线程
Y
N
CAS轻量级
CAS获取是否成功
Y
拿到锁
N
自旋
防止线程被挂起
理解为短暂死循环
重量级

升级方向：


无锁
偏向锁
轻量级锁
重量级锁

Tip：切记这个升级过程是不可逆的，最后我会说明他的影响，涉及使用场景。

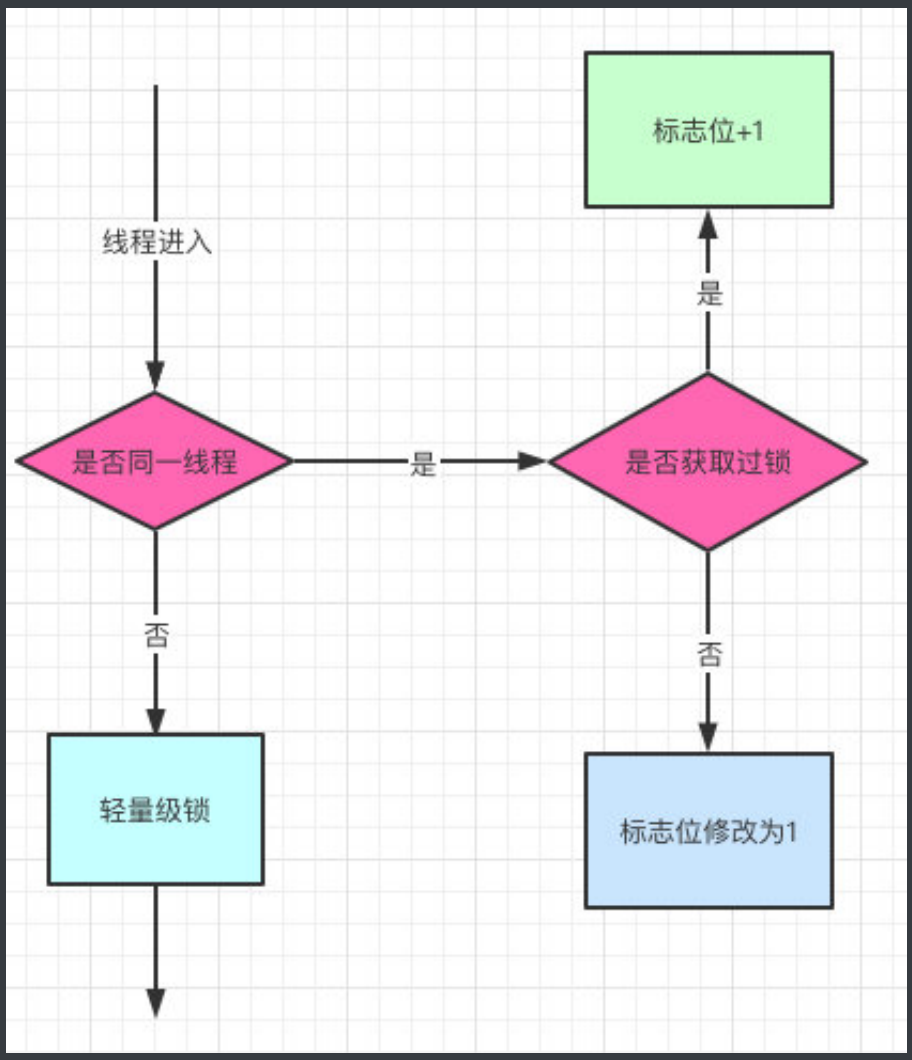
看完他的升级，我们就来好好聊聊每一步怎么做的吧。

偏向锁

之前我提到过了，对象头是由Mark Word和Klass pointer 组成，锁争夺也就是对象头指向的Monitor对象的争夺，一旦有线程持有了这个对象，标志位修改为1，就进入偏向模式，同时会把这个线程的ID记录在对象的Mark Word中。

这个过程是采用了CAS乐观锁操作的，每次同一线程进入，虚拟机就不进行任何同步的操作了，对标志位+1就好了，不同线程过来，CAS会失败，也就意味着获取锁失败。

偏向锁在1.6之后是默认开启的，1.5中是关闭的，需要手动开启参数是xx:-UseBiasedLocking=false。



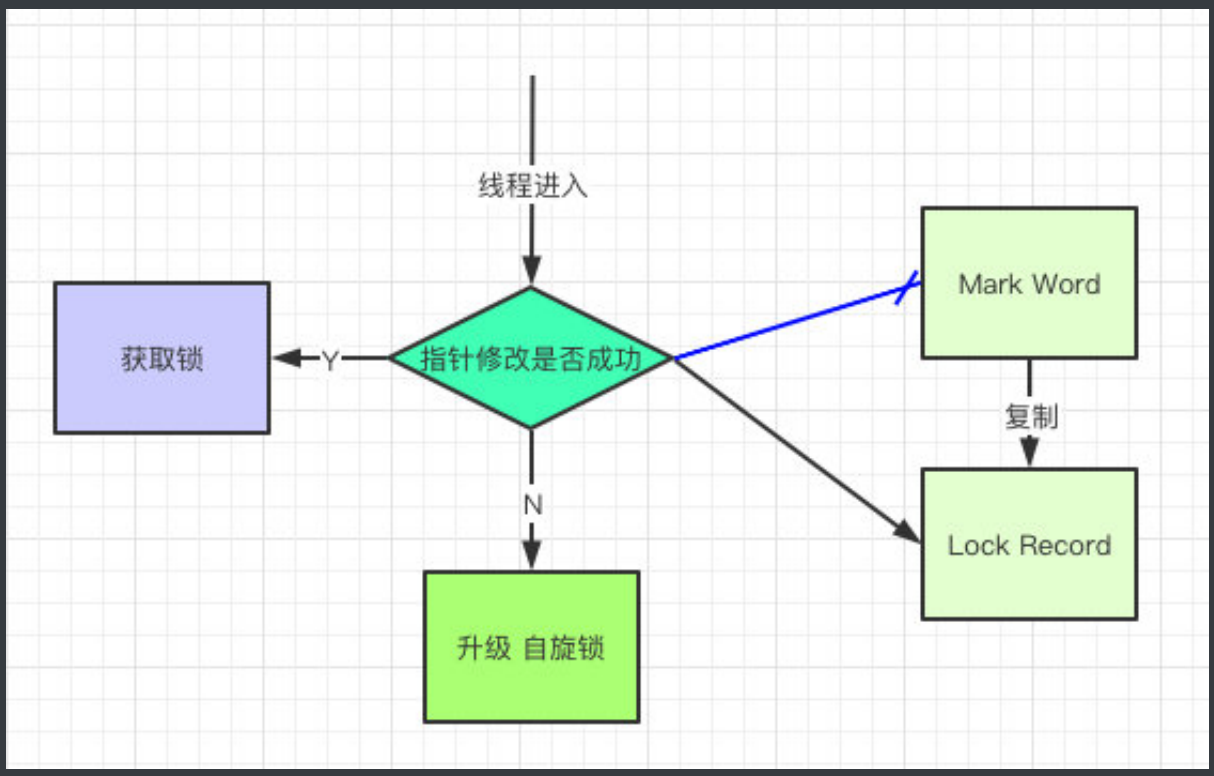
偏向锁关闭，或者多个线程竞争偏向锁怎么办呢?

轻量级锁

还是跟Mark Work 相关，如果这个对象是无锁的，jvm就会在当前线程的栈帧中建立一个叫锁记录（Lock Record）的空间，用来存储锁对象的Mark Word 拷贝，然后把Lock Record中的owner指向当前对象。

JVM接下来会利用CAS尝试把对象原本的Mark Word 更新会Lock Record的指针，成功就说明加锁成功，改变锁标志位，执行相关同步操作。

如果失败了，就会判断当前对象的Mark Word是否指向了当前线程的栈帧，是则表示当前的线程已经持有了这个对象的锁，否则说明被其他线程持有了，继续锁升级，修改锁的状态，之后等待的线程也阻塞。

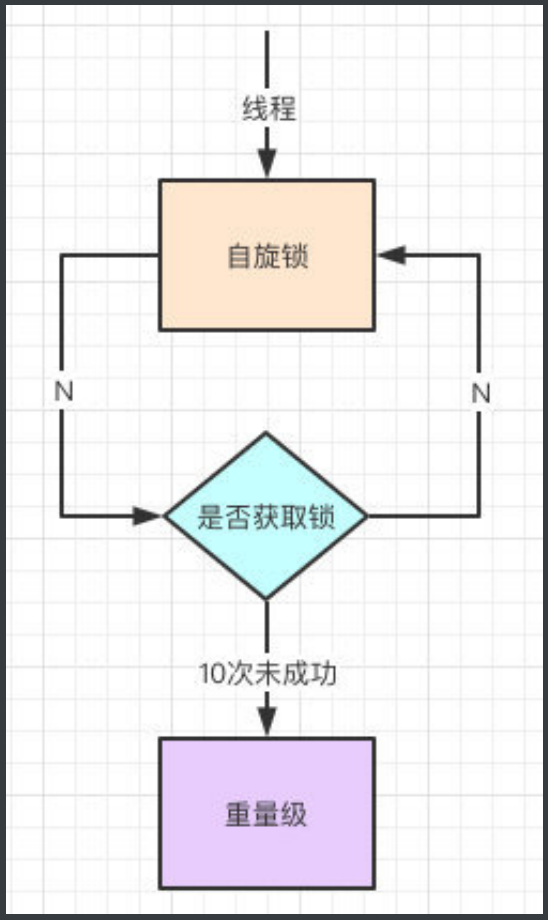


自旋锁

我不是在上面提到了Linux系统的用户态和内核态的切换很耗资源，其实就是线程的等待唤起过程，那怎么才能减少这种消耗呢?

自旋，过来的现在就不断自旋，防止线程被挂起，一旦可以获取资源，就直接尝试成功，直到超出阈值，自旋锁的默认大小是10次，-XX：PreBlockSpin可以修改。

自旋都失败了，那就升级为重量级的锁，像1.5的一样，等待唤起咯。



至此我基本上吧synchronized的前后概念都讲到了，大家好好消化。

资料参考:《高并发编程》《黑马程序员讲义》《深入理解JVM虚拟机》

用synchronized还是Lock呢?

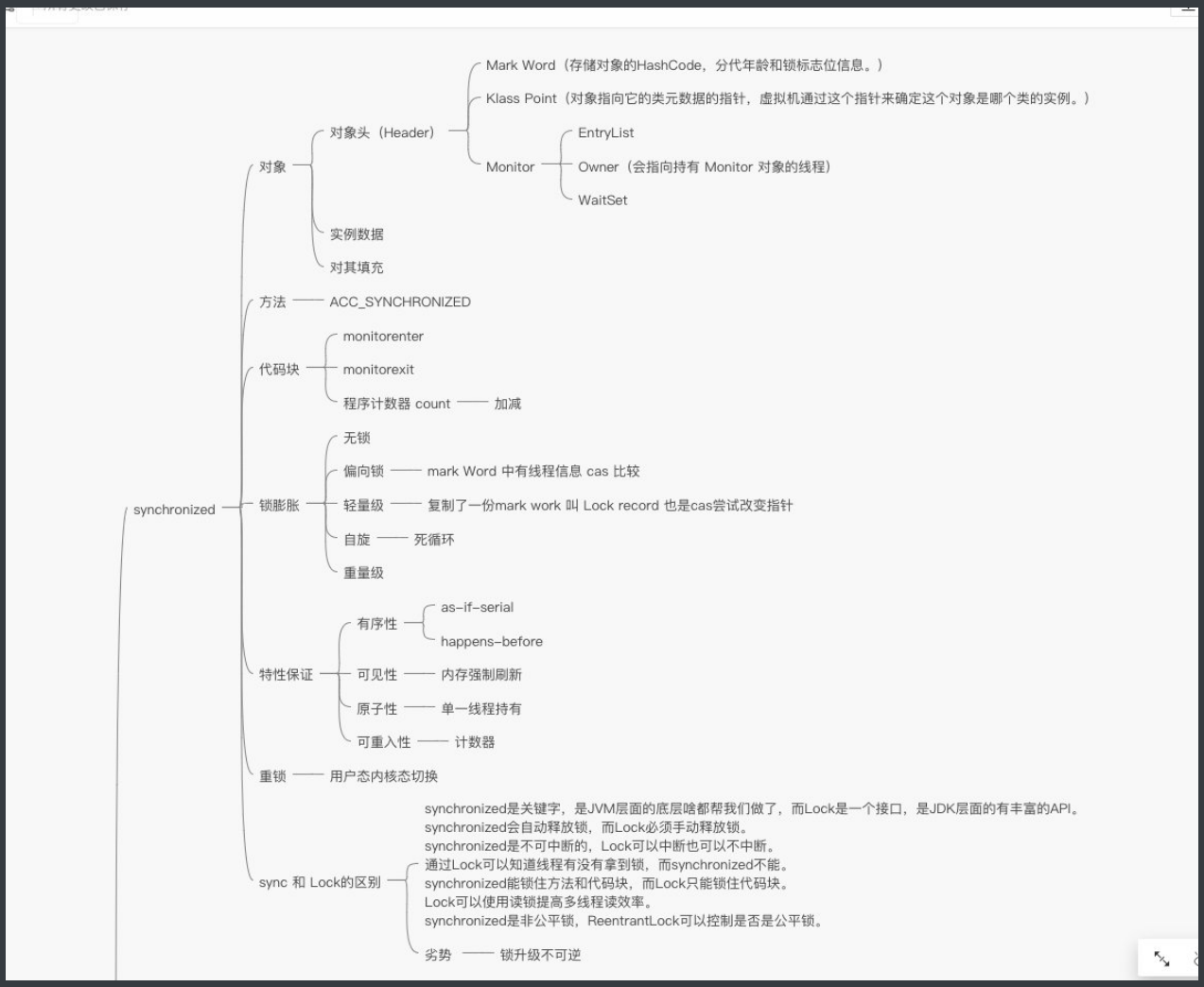
我们先看看他们的区别：

- synchronized是关键字，是JVM层面的底层啥都帮我们做了，而Lock是一个接口，是JDK层面的有丰富的API。
- synchronized会自动释放锁，而Lock必须手动释放锁。
- synchronized是不可中断的，Lock可以中断也可以不中断。
- 通过Lock可以知道线程有没有拿到锁，而synchronized不能。
- synchronized能锁住方法和代码块，而Lock只能锁住代码块。
- Lock可以使用读锁提高多线程读效率。
- synchronized是非公平锁，ReentrantLock可以控制是否是公平锁。

两者一个是JDK层面的一个是JVM层面的，我觉得最大的区别其实在，我们是否需要丰富的api，还有一个我们的场景。

比如我现在是滴滴，我早上有打车高峰，我代码使用了大量的synchronized，有什么问题？锁升级过程是不可逆的，过了高峰我们还是重量级的锁，那效率是不是大打折扣了？这个时候你用Lock是不是很好?

场景是一定要考虑的，我现在告诉你哪个好都是扯淡，因为脱离了业务，一切技术讨论都没有了价值。



我是敖丙，一个在互联网苟且偷生的工具人。

你知道的越多，你不知道的越多，人才们的 **【三连】** 就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎人才们留言！

> 文章持续更新，可以微信搜索「 **三太子敖丙** 」第一时间阅读，回复【**资料**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。


三太子教育
微信搜索