synchronized的使用很简单，可以用它来修饰实例方法和静态方法，也可以用来修饰代码块。值的注意的是synchronized是一个对象锁，也就是它锁的是一个对象。因此，无论使用哪一种方法，synchronized都需要有一个锁对象

### 1.修饰实例方法

synchronized修饰实例方法只需要在方法上加上synchronized关键字即可。

public synchronized void add(){

i++;

}

此时，synchronized加锁的对象就是这个方法所在实例的本身。

### 2.修饰静态方法

synchronized修饰静态方法的使用与实例方法并无差别，在静态方法上加上synchronized关键字即可

public static synchronized void add(){

i++;

}

此时，synchronized加锁的对象为当前静态方法所在类的Class对象。

### 3.修饰代码块

synchronized修饰代码块需要传入一个对象。

public void add() {

synchronized (this) {

i++;

}

}

很明显，此时synchronized加锁对象即为传入的这个对象实例。

到这里不是道你是否有个疑问，synchronized关键字是如何对一个对象加锁实现代码同步的呢？

在JVM中，对象在内存中存储的布局可以分为三个区域，分别是对象头、实例数据以及填充数据。

实例数据 存放类的属性数据信息，包括父类的属性信息，这部分内存按4字节对齐。

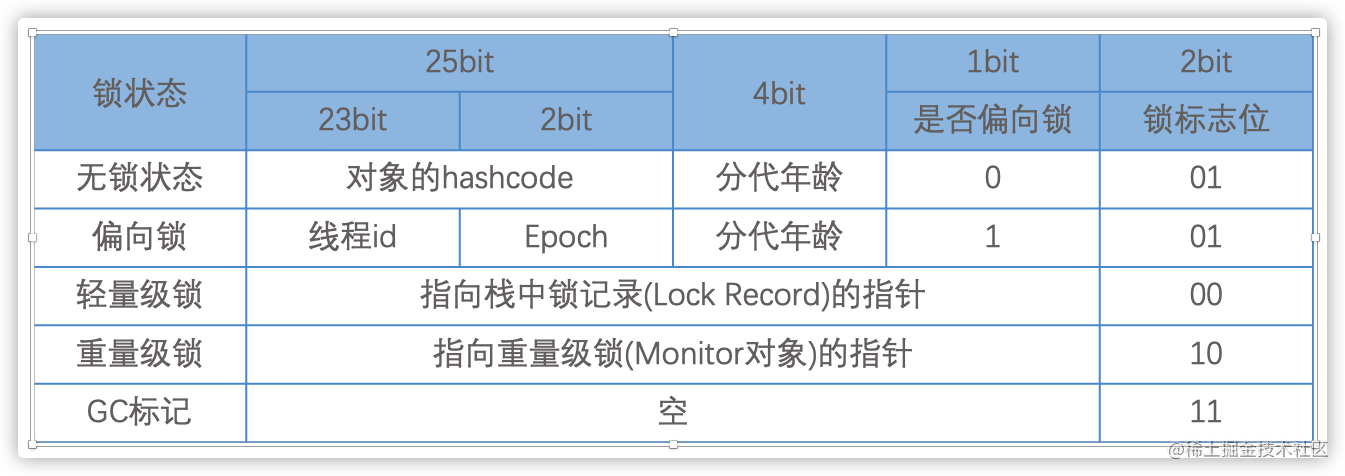
填充数据 由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐。

对象头 在HotSpot虚拟机中，对象头又被分为两部分，分别为：Mark Word(标记字段)、Class Pointer(类型指针)。如果是数组，那么还会有数组长度。对象头是本章内容的重点，下边详细讨论。

### 1.对象头

在对象头的Mark Word中主要存储了对象自身的运行时数据，例如哈希码、GC分代年龄、锁状态、线程持有的锁、偏向线程ID以及偏向时间戳等。同时，Mark Word也记录了对象和锁有关的信息。

当对象被synchronized关键字当成同步锁时，和锁相关的一系列操作都与Mark Word有关。由于在JDK1.6版本中对synchronized进行了锁优化，引入了偏向锁和轻量级锁。Mark Word在不同锁状态下存储的内容有所不同。我们以32位JVM中对象头的存储内容如下图所示。



从图中可以清楚的看到，Mark Word中有2bit的数据用来标记锁的状态。无锁状态和偏向锁标记位为01，轻量级锁的状态为00，重量级锁的状态为10。

当对象为偏向锁时，Mark Word存储了偏向线程的ID；

当状态为轻量级锁时，Mark Word存储了指向线程栈中Lock Record的指针；

当状态为重量级锁时，Mark Word存储了指向堆中的Monitor对象的指针。

当前我们只讨论重量级锁，因为重量级锁相当于对synchronized优化之前的状态。

可以看到，当为重量级锁时，对象头的MarkWord中存储了指向Monitor对象的指针。那么Monitor又是什么呢？

Monitor对象被称为管程或者监视器锁。在Java中，每一个对象实例都会关联一个Monitor对象。这个Monitor对象既可以与对象一起创建销毁，也可以在线程试图获取对象锁时自动生成。当这个Monitor对象被线程持有后，它便处于锁定状态。

在HotSpot虚拟机中，Monitor是由[ObjectMonitor](https://link.juejin.cn?target=https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.hpp" \o "https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.hpp" \t "/Users/zpw/Documents\\x/_blank)实现的,它是一个使用C++实现的类，主要数据结构如下：

ObjectMonitor() {

\_header = NULL;

\_count = 0; //记录个数

\_waiters = 0,

\_recursions = 0; // 线程重入次数

\_object = NULL;

\_owner = NULL;

\_WaitSet = NULL; // 调用wait方法后的线程会被加入到\_WaitSet

\_WaitSetLock = 0 ;

\_Responsible = NULL ;

\_succ = NULL ;

\_cxq = NULL ; // 阻塞队列，线程被唤醒后根据决策判读是放入cxq还是EntryList

FreeNext = NULL ;

\_EntryList = NULL ; // 没有抢到锁的线程会被放到这个队列

\_SpinFreq = 0 ;

\_SpinClock = 0 ;

OwnerIsThread = 0 ;

}

ObjectMonitor中有五个重要部分，分别为\_ower, \_WaitSet, \_cxq, \_EntryList和count。

\_ower 用来指向持有monitor的线程，它的初始值为NULL,表示当前没有任何线程持有monitor。当一个线程成功持有该锁之后会保存线程的ID标识，等到线程释放锁后\_ower又会被重置为NULL;

\_WaitSet 调用了锁对象的wait方法后的线程会被加入到这个队列中；

\_cxq 是一个阻塞队列，线程被唤醒后根据决策判读是放入cxq还是EntryList;

\_EntryList 没有抢到锁的线程会被放到这个队列；

count 用于记录线程获取锁的次数，成功获取到锁后count会加1，释放锁时count减1。

如果线程获取到对象的monitor后，就会将monitor中的ower设置为该线程的ID，同时monitor中的count进行加1. 如果调用锁对象的wait()方法，线程会释放当前持有的monitor，并将owner变量重置为NULL，且count减1,同时该线程会进入到\_WaitSet集合中等待被唤醒。

另外\_WaitSet，\_cxq与\_EntryList都是链表结构的队列，存放的是封装了线程的ObjectWaiter对象。如果不深入虚拟机查看相关源码很难理解这几个队列的作用，关于源码会在后边系列文章中分析。这里我简单说下它们之间的关系，如下：

在多条线程竞争monitor锁的时候，所有没有竞争到锁的线程会被封装成ObjectWaiter并加入\_EntryList队列。 当一个已经获取到锁的线程，调用锁对象的wait方法后，线程也会被封装成一个ObjectWaiter并加入到\_WaitSet队列中。 当调用锁对象的notify方法后，会根据不同的情况来决定是将\_WaitSet集合中的元素转移到\_cxq队列还是\_EntryList队列。 等到获得锁的线程释放锁后，又会根据条件来执行\_EntryList中的线程或者将\_cxq转移到\_EntryList中再执行\_EntryList中的线程。

所以，可以看得出来，\_WaitSet存放的是处于WAITING状态等待被唤醒的线程。而\_EntryList队列中存放的是等待锁的BLOCKED状态。\_cxq队列仅仅是临时存放，最终还是会被转移到\_EntryList中等待获取锁。

了解了对象头和Monitor，那么synchronized关键字到底是如何做到与monitor关联的呢？

### 1.同步代码块

通过javap -v来反汇编下面的一段代码。

public void add() {

synchronized (this) {

i++;

}

}

可以得到如下的字节码指令：

public class com.zhangpan.text.TestSync {

public com.zhangpan.text.TestSync();

Code:

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

public void add();

Code:

0: aload\_0

1: dup

2: astore\_1

3: monitorenter // synchronized关键字的入口

4: getstatic #2 // Field i:I

7: iconst\_1

8: iadd

9: putstatic #2 // Field i:I

12: aload\_1

13: monitorexit // synchronized关键字的出口

14: goto 22

17: astore\_2

18: aload\_1

19: monitorexit // synchronized关键字的出口

20: aload\_2

21: athrow

22: return

Exception table:

from to target type

4 14 17 any

17 20 17 any

}

从字节码指令中可以看到add方法的第3条指令处和第13、19条指令处分别有monitorenter和moniterexit两条指令。另外第4、7、8、9、13这几条指令其实就是i++的指令。由此可以得出在字节码中会在同步代码块的入口和出口加上monitorenter和moniterexit指令。当执行到monitorenter指令时，线程就会去尝试获取该对象对应的Monitor的所有权，即尝试获得该对象的锁。

当该对象的 monitor 的计数器count为0时，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功。如果当前线程已经拥有该对象monitor的持有权，那它可以重入这个 monitor ，计数器的值也会加 1。而当执行monitorexit指令时，锁的计数器会减1。

倘若其他线程已经拥有monitor 的所有权，那么当前线程获取锁失败将被阻塞并进入到\_EntryList中，直到等待的锁被释放为止。也就是说，当所有相应的monitorexit指令都被执行，计数器的值减为0，执行线程将释放 monitor(锁)，其他线程才有机会持有 monitor 。

2.同步方法的实现

同步方法的字节码指令与同步代码块的字节指令有所差异。我们先来通过javap -v查看下面代码的字节码指令。

public synchronized void add(){

i++;

}

反汇编后可得到如下的字节指令

public synchronized void add();

descriptor: ()V

flags: (0x0021) ACC\_PUBLIC, ACC\_SYNCHRONIZED

Code:

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

0: aload\_0

1: dup

2: getfield #2 // Field i:I

5: iconst\_1

6: iadd

7: putfield #2 // Field i:I

10: return

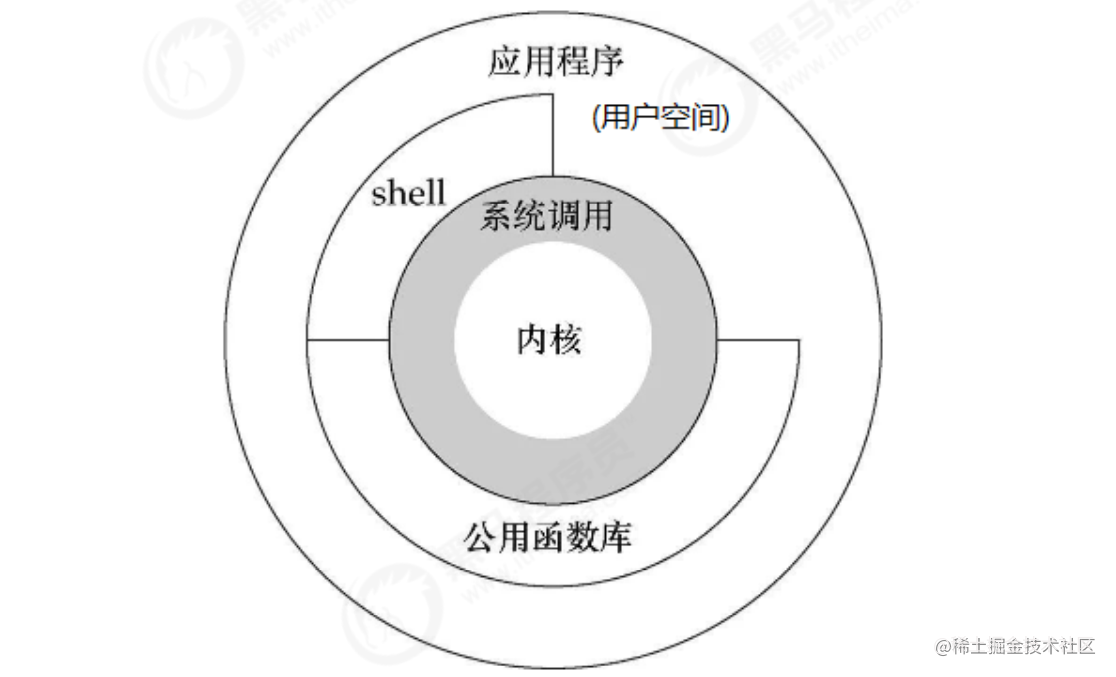
LineNumberTable:

line 5: 0

line 6: 10

可以看到这里并没有monitorenter和moniterexit两条指令，而是在方法的flag上加入了ACC\_SYNCHRONIZED的标记位。这其实也容易理解，因为整个方法都是同步代码，因此就不需要标记同步代码的入口和出口了。当线程线程执行到这个方法时会判断是否有这个ACC\_SYNCHRONIZED标志，如果有的话则会尝试获取monitor对象锁。

在Linux系统架构中可以分为用户空间和内核，我们的程序都运行在用户空间，进入用户运行状态就是所谓的用户态。在用户态可能会涉及到某些操作如I/O调用，就会进入内核中运行，此时进程就被称为内核运行态，简称内核态。



内核： 本质上可以理解为一种软件，控制计算机的硬件资源，并提供上层应用程序运行的环境。

用户空间： 上层应用程序活动的空间。应用程序的执行必须依托于内核提供的资源，包括CPU资源、存储资源、I/O资源等。

系统调用： 为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。

上边我们已经提到了使用monitor是重量级锁的加锁方式。在[objectMonitor.cpp](https://link.juejin.cn?target=https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \o "https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \t "/Users/zpw/Documents\\x/_blank)中会涉及到Atomic::cmpxchg\_ptr，Atomic::inc\_ptr等内核函数， 执行同步代码块，没有竞争到锁的对象会park()被挂起，竞争到锁的线程会unpark()唤醒。这个时候就会存在操作系统用户态和内核态的转换，这种切换会消耗大量的系统资源。试想，如果程序中存在大量的锁竞争，那么会引起程序频繁的在用户态和内核态进行切换，严重影响到程序的性能。这也是为什么说synchronized效率低的原因

为了解决这一问题，在JDK1.6中引入了偏向锁和轻量级锁来优化synchronized。

JDK1.6中引入偏向锁和轻量级锁对synchronized进行优化。此时的synchronized一共存在四个状态：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态。锁着锁竞争激烈程度，锁的状态会出现一个升级的过程。即可以从偏向锁升级到轻量级锁，再升级到重量级锁。锁升级的过程是单向不可逆的，即一旦升级为重量级锁就不会再出现降级的情况。

1.几种锁状态

接下来我们来详细的认识一下这几种锁状态。

1).偏向锁

经研究发现，在大多数情况下锁不仅不存在多线程竞争关系，而且大多数情况都是被同一线程多次获得。因此，为了减少同一线程获取锁的代价而引入了偏向锁的概念。

偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word的结构也变为偏向锁结构，即将对象头中Mark Word的第30bit的值改为1，并且在Mark Word中记录该线程的ID。当这个线程再次请求锁时，无需再做任何同步操作，即可获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提升了程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。

但是，对于锁竞争比较激烈的情况，偏向锁就有问题了。因为每次申请锁的都可能是不同线程。这种情况使用偏向锁就会得不偿失，此时就会升级为轻量级锁。

2).轻量级锁

轻量级锁优化性能的依据是对于大部分的锁，在整个同步生命周期内都不存在竞争。 当升级为轻量级锁之后，MarkWord的结构也会随之变为轻量级锁结构。JVM会利用CAS尝试把对象原本的Mark Word 更新为Lock Record的指针，成功就说明加锁成功，改变锁标志位为00，然后执行相关同步操作。

轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁就会失效，进而膨胀为重量级锁。

3).自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

自旋锁是基于在大多数情况下，线程持有锁的时间都不会太长。如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，不断的尝试获取锁。空循环一般不会执行太多次，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入同步代码。如果还不能获得锁，那就会将线程在操作系统层面挂起，即进入到重量级锁。

这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。

2.synchronized锁升级过程

在了解了jdk1.6引入的这几种锁之后，我们来详细的看一下synchronized是怎么一步步进行锁升级的。

(1）当没有被当成锁时，这就是一个普通的对象，Mark Word记录对象的HashCode，锁标志位是01，是否偏向锁那一位是0;

(2）当对象被当做同步锁并有一个线程A抢到了锁时，锁标志位还是01，但是否偏向锁那一位改成1，前23bit记录抢到锁的线程id，表示进入偏向锁状态;

(3) 当线程A再次试图来获得锁时，JVM发现同步锁对象的标志位是01，是否偏向锁是1，也就是偏向状态，Mark Word中记录的线程id就是线程A自己的id，表示线程A已经获得了这个偏向锁，可以执行同步中的代码;

(4) 当线程B试图获得这个锁时，JVM发现同步锁处于偏向状态，但是Mark Word中的线程id记录的不是B，那么线程B会先用CAS操作试图获得锁，这里的获得锁操作是有可能成功的，因为线程A一般不会自动释放偏向锁。如果抢锁成功，就把Mark Word里的线程id改为线程B的id，代表线程B获得了这个偏向锁，可以执行同步代码。如果抢锁失败，则继续执行步骤5;

(5) 偏向锁状态抢锁失败，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。JVM会在当前线程的线程栈中开辟一块单独的空间，里面保存指向对象锁Mark Word的指针，同时在对象锁Mark Word中保存指向这片空间的指针。上述两个保存操作都是CAS操作，如果保存成功，代表线程抢到了同步锁，就把Mark Word中的锁标志位改成00，可以执行同步代码。如果保存失败，表示抢锁失败，竞争太激烈，继续执行步骤6;

(6) 轻量级锁抢锁失败，JVM会使用自旋锁，自旋锁不是一个锁状态，只是代表不断的重试，尝试抢锁。从JDK1.7开始，自旋锁默认启用，自旋次数由JVM决定。如果抢锁成功则执行同步代码，如果失败则继续执行步骤7;

(7) 自旋锁重试之后如果抢锁依然失败，同步锁会升级至重量级锁，锁标志位改为10。在这个状态下，未抢到锁的线程都会被阻塞。