synchronized的特性

原子性：

所谓原子性就是指一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。但是像i++、i+=1等操作字符就不是原子性的，它们是分成读取、计算、赋值几步操作，原值在这些步骤还没完成时就可能已经被赋值了，那么最后赋值写入的数据就是脏数据，无法保证原子性。

被synchronized修饰的类或对象的所有操作都是原子的，因为在执行操作之前必须先获得类或对象的锁，直到执行完才能释放，这中间的过程无法被中断（除了已经废弃的stop()方法），即保证了原子性。

注意！面试时经常会问比较synchronized和volatile，它们俩特性上最大的区别就在于原子性，volatile不具备原子性。

可见性：

可见性是指多个线程访问一个资源时，该资源的状态、值信息等对于其他线程都是可见的。

synchronized和volatile都具有可见性，其中synchronized对一个类或对象加锁时，一个线程如果要访问该类或对象必须先获得它的锁，而这个锁的状态对于其他任何线程都是可见的，并且在释放锁之前会将对变量的修改刷新到主存当中，保证资源变量的可见性，如果某个线程占用了该锁，其他线程就必须在锁池中等待锁的释放。

而volatile的实现类似，被volatile修饰的变量，每当值需要修改时都会立即更新主存，主存是共享的，所有线程可见，所以确保了其他线程读取到的变量永远是最新值，保证可见性。

有序性：

有序性指程序执行的顺序按照代码先后执行。

synchronized和volatile都具有有序性，Java允许编译器和处理器对指令进行重排，但是指令重排并不会影响单线程的顺序，它影响的是多线程并发执行的顺序性。synchronized保证了每个时刻都只有一个线程访问同步代码块，也就确定了线程执行同步代码块是分先后顺序的，保证了有序性。

可重入性：

synchronized和ReentrantLock都是可重入锁。当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁。通俗一点讲就是说一个线程拥有了锁仍然还可以重复申请锁。

同步实现方式：

synchronized可以修饰静态方法、成员函数，同时还可以直接定义代码块，但是归根结底它上锁的资源只有两类：一个是对象，一个是类。

synchronized有两种形式上锁，一个是对方法上锁，一个是构造同步代码块。他们的底层实现其实都一样，在进入同步代码之前先获取锁，获取到锁之后锁的计数器+1，同步代码执行完锁的计数器-1，如果获取失败就阻塞式等待锁的释放。只是他们在同步块识别方式上有所不一样，从class字节码文件可以表现出来：

从反编译的同步方法可以看到是通过方法flags标志该方法是同步方法。

从反编译的同步代码块可以看到同步块是由monitorenter指令进入，然后monitorexit释放锁，在执行monitorenter之前需要尝试获取锁，如果这个对象没有被锁定，或者当前线程已经拥有了这个对象的锁，那么就把锁的计数器加1。当执行monitorexit指令时，锁的计数器也会减1。当获取锁失败时会被阻塞，一直等待锁被释放。

但是为什么会有两个monitorexit呢？其实第二个monitorexit是来处理异常的，仔细看反编译的字节码，正常情况下第一个monitorexit之后会执行goto指令，而该指令转向的就是23行的return，也就是说正常情况下只会执行第一个monitorexit释放锁，然后返回。而如果在执行中发生了异常，第二个monitorexit就起作用了，它是由编译器自动生成的，在发生异常时处理异常然后释放掉锁。

synchronized锁的底层实现：

在JVM中，对象是分成三部分存在的：对象头、实例数据、对其填充。



实例数据存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐；对其填充不是必须部分，由于虚拟机要求对象起始地址必须是8字节的整数倍，对齐填充仅仅是为了使字节对齐。

对象头是我们需要关注的重点，它是synchronized实现锁的基础，因为synchronized申请锁、上锁、释放锁都与对象头有关。对象头主要结构是由Mark Word 和 Class Metadata Address组成，其中Mark Word存储对象的hashCode、锁信息或分代年龄或GC标志等信息，Class Metadata Address是类型指针指向对象的类元数据，JVM通过该指针确定该对象是哪个类的实例。

锁也分不同状态，JDK6之前只有两个状态：无锁、有锁（重量级锁），而在JDK6之后对synchronized进行了优化，新增了两种状态，总共就是四个状态：无锁状态、偏向锁、轻量级锁、重量级锁，其中无锁就是一种状态了。锁的类型和状态在对象头Mark Word中都有记录，在申请锁、锁升级等过程中JVM都需要读取对象的Mark Word数据。

每一个锁都对应一个monitor对象，在HotSpot虚拟机中它是由ObjectMonitor实现的（C++实现）。每个对象都存在着一个monitor与之关联，对象与其monitor之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个monitor被某个线程持有后，它便处于锁定状态。

ObjectMonitor() {

\_header = NULL;

\_count = 0; //锁计数器

\_waiters = 0,

\_recursions = 0;

\_object = NULL;

\_owner = NULL;

\_WaitSet = NULL; //处于wait状态的线程，会被加入到\_WaitSet

\_WaitSetLock = 0 ;

\_Responsible = NULL ;

\_succ = NULL ;

\_cxq = NULL ;

FreeNext = NULL ;

\_EntryList = NULL ; //处于等待锁block状态的线程，会被加入到该列表

\_SpinFreq = 0 ;

\_SpinClock = 0 ;

OwnerIsThread = 0 ;

}

ObjectMonitor中有两个队列\_WaitSet和\_EntryList，用来保存ObjectWaiter对象列表(每个等待锁的线程都会被封装ObjectWaiter对象)，\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入\_EntryList 集合，当线程获取到对象的monitor 后进入 \_Owner 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。 monitor对象存在于每个Java对象的对象头中(存储的指针的指向)，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，同时也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因(关于这点稍后还会进行分析)。

JVM对synchronized的优化：

锁膨胀

上面讲到锁有四种状态，并且会因实际情况进行膨胀升级，其膨胀方向是：无锁——>偏向锁——>轻量级锁——>重量级锁，并且膨胀方向不可逆。

偏向锁

一句话总结它的作用：减少统一线程获取锁的代价。在大多数情况下，锁不存在多线程竞争，总是由同一线程多次获得，那么此时就是偏向锁。

核心思想：

如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word的结构也就变为偏向锁结构，当该线程再次请求锁时，无需再做任何同步操作，即获取锁的过程只需要检查Mark Word的锁标记位为偏向锁以及当前线程ID等于Mark Word的ThreadID即可，这样就省去了大量有关锁申请的操作。

轻量级锁

轻量级锁是由偏向锁升级而来，当存在第二个线程申请同一个锁对象时，偏向锁就会立即升级为轻量级锁。注意这里的第二个线程只是申请锁，不存在两个线程同时竞争锁，可以是一前一后地交替执行同步块。

重量级锁

重量级锁是由轻量级锁升级而来，当同一时间有多个线程竞争锁时，锁就会被升级成重量级锁，此时其申请锁带来的开销也就变大。

重量级锁一般使用场景会在追求吞吐量，同步块或者同步方法执行时间较长的场景。

锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，在JIT编译时，对运行上下文进行扫描，去除不可能存在竞争的锁。

锁粗化

锁粗化是虚拟机对另一种极端情况的优化处理，通过扩大锁的范围，避免反复加锁和释放锁。

自旋锁与自适应自旋锁

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：许多情况下，共享数据的锁定状态持续时间较短，切换线程不值得，通过让线程执行循环等待锁的释放，不让出CPU。如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式。但是它也存在缺点：如果锁被其他线程长时间占用，一直不释放CPU，会带来许多的性能开销。

自适应自旋锁：这种相当于是对上面自旋锁优化方式的进一步优化，它的自旋的次数不再固定，其自旋的次数由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定，这就解决了自旋锁带来的缺点。