【第10话：小组Leader级面试题synchronized底层原理】

Hello 小伙伴们，这结课给大家讲解一下：“synchronized底层原理”。

这是之前我们一个学员毕业找工作被问到的一个问题，他面试的小组Leader。

其实这个问题对于一些初学Java的人比较难的，甚至都没有想过synchronized，一个关键字，居然还有底层原理。但实际上，synchronized底层原理还是很重要的，里面涉及到很多底层的概念。

在Java早期，synchronized叫做重量级锁，加锁过程需要操作系统在内核态访问核心资源，因此操作系统会在用户态与内核态之间切换，效率很低下。于是在Java 5版本中推出JUC中的Lock，把这种synchronized这种原拆分成多部分，可以让学员手动控制整个过程，但这是synchronized性能要比Lock低很多。于是JDK1.6之后，JVM为了提高锁的获取与释放效率，对synchronized进行了优化，引入了偏向锁和轻量级锁，根据线程竞争情况对锁进行升级，在线程竞争不激烈的情况避免使用重量级锁。随着不停的优化，在目前Java中synchronized和Lock的性能区别已经不是很大了。

synchronized底层原理这个问题几乎不可能出现在初、中级Java程序员中。出现的场景都是高级程序员面试时或大厂面试时能被问到的。因为想要说明synchronized真正的原理，需要清楚的知道Monitor及Java对象头等概念。

在本节视频中我们只有薪资2万+的版本。

为了能让同学们能够理解synchronized底层原理，在本节视频中我们先来讲解一下相关的概念，而不是直接进行回答示范。这里面的一些概念可能一些5年甚至10年的Java程序员也没有深入研究过。虽然本套视频重在示范面试应该如何回答，但是对于一些特别底层的概念还是有必要去给小伙伴们讲解一下的，待讲解完成后，在视频的最后进行回答示范。

对于Java中synchronized关键字，无论是Java初级、中级、还是高级。都知道他是Java中内置锁，可实现线程同步，保证线程安全。可以使用synchronized关键字来修饰方法或定义代码块。

```
1  public class Demo {
2      // 修饰方法
3      public synchronized void test1(){
4      }
5      // 定义代码块
6      public void test2(){
7          synchronized (this){
8          }
9      }
10 }
```

使用javap可以查看编译后内容。

```
D:\idea\projects\bjsxt\out\production\bjsxt\com\bjsxt>javap -v Demo
警告: 文件 .\Demo.class 不包含类 Demo
Classfile /D:/idea/projects/bjsxt/out/production/bjsxt/com/bjsxt/Demo.class
  Last modified 2022年8月27日; size 493 bytes
  MD5 checksum 610e391958a6020eb03e91bd5973b298
  Compiled from "Demo.java"
public class com.bjsxt.Demo
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #2                          // com/bjsxt/Demo
  super_class: #3                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 3, attributes: 1
Constant pool:
   #1 = Methodref          #3.#17         // java/lang/Object."<init>":()V
   #2 = Class              #18            // com/bjsxt/Demo
   #3 = Class              #19            // java/lang/Object
   #4 = Utf8               <init>
   #5 = Utf8               ()V
   #6 = Utf8               Code
   #7 = Utf8               LineNumberTable
   #8 = Utf8               LocalVariableTable
   #9 = Utf8               this
  #10 = Utf8               Lcom/bjsxt/Demo;
  #11 = Utf8               test1
  #12 = Utf8               test2
  #13 = Utf8               StackMapTable
  #14 = Class              #20            // java/lang/Throwable
  #15 = Utf8               SourceFile
  #16 = Utf8               Demo.java
  #17 = NameAndType        #4:#5          // "<init>":()V
  #18 = Utf8               com/bjsxt/Demo
  #19 = Utf8               java/lang/Object
  #20 = Utf8               java/lang/Throwable
{
  public com.bjsxt.Demo();
    descriptor: ()V
    flags: (0x0001) 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 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/bjsxt/Demo;

  public synchronized void test1();
    descriptor: ()V
    flags: (0x0021) ACC_PUBLIC, ACC_SYNCHRONIZED
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 5: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       1     0  this   Lcom/bjsxt/Demo;

  public void test2();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter
         4: aload_1
         5: monitorexit
         6: goto          14
         9: astore_2
        10: aload_1
        11: monitorexit
        12: aload_2
        13: athrow
```

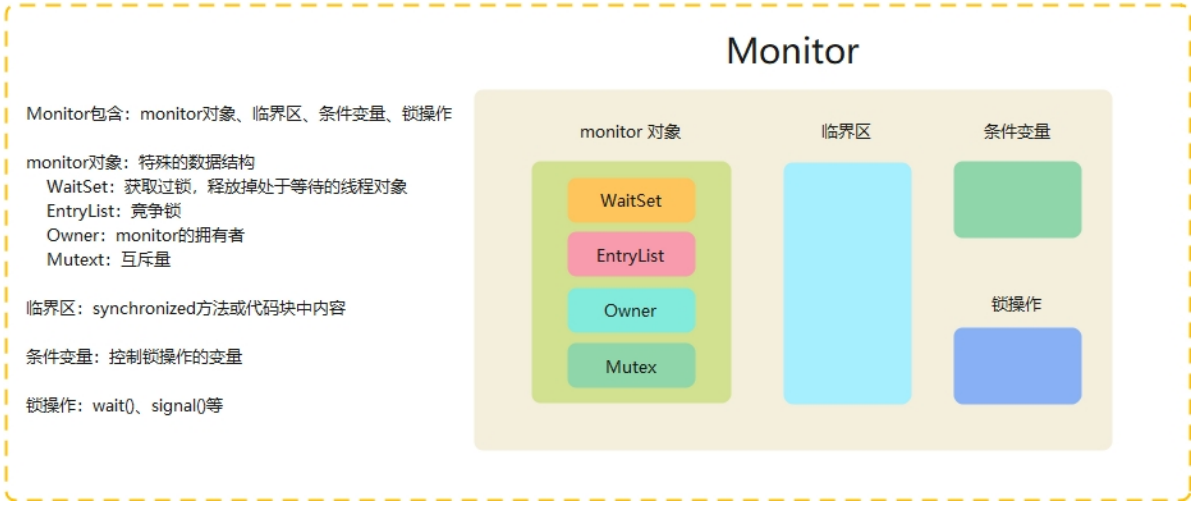
```
        15: athrow
        14: return
      Exception table:
         from    to  target type
             4     6     9   any
             9    12     9   any
      LineNumberTable:
        line 8: 0
        line 9: 4
        line 10: 14
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      15     0  this   Lcom/bjsxt/Demo;
      StackMapTable: number_of_entries = 2
        frame_type = 255 /* full_frame */
          offset_delta = 9
          locals = [ class com/bjsxt/Demo, class java/lang/Object ]
          stack = [ class java/lang/Throwable ]
        frame_type = 250 /* chop */
          offset_delta = 4
}
SourceFile: "Demo.java"
```

标记出来的内容中，可以看出test1()方法使用标记ACC_SYNCHRONIZED，这个好理解，就是表示当前方法使用了synchronized修饰了，而在test2()方法synchronized代码块是使用monitorenter和monitorexit实现。

想要把这两个指令搞清楚，就需要先来了解一下Monitor。

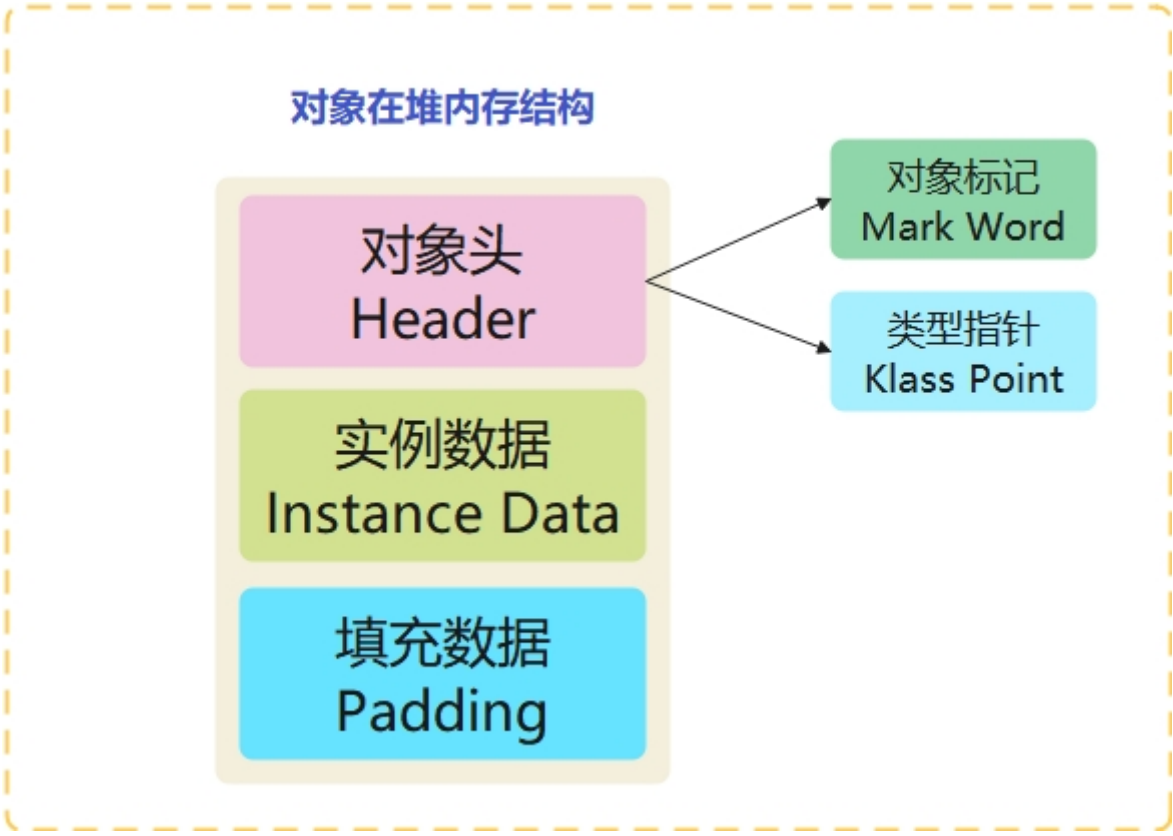
Monitor翻译过来叫做监视器。在操作系统中叫做管程。Monitor的实现是由编程语言完成的。作用是为了简化操作系统访问共享变量，降低线程同步的复杂性，而进行的封装，类似语法糖。synchronized在JDK1.6之前是没有锁膨胀机制的，完全是重量级锁。

看看Java中的监视器Monitor组成部分



http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/095e60e7fc8c/src/share/vm/runtime/objectMonitor.hpp

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0;
    _waiters      = 0,
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL;
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ;
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
}
```



锁状态	56bit		1bit	4bit	1bit 偏向锁位	2bit 锁标志位
	25bit	31bit				
无锁	unused	hashcode(如果有调用)	unused	分代年龄	0	01
偏向锁	线程ID	Epoch (2bit)	unused	分代年龄	1	01
轻量级锁	指向线程栈中Lock Record的指针					00
重量级锁	指向互斥量（重量级锁）的指针					10
GC标志	CMS过程用到的标记信息					11

Monitor包含四个组成部分：monitor对象、临界区、条件变量、锁操作。

其中monitor对象其实就是一种特殊的数据结构。里面包含WaitSet，WaitSet负责存储所有获取过锁，但是目前释放了锁的线程对象。EntryList里面存储所有正在竞争锁的线程对象。Owner中存储了目前持有锁的线程对象。Mutex是互斥量，synchronized之所以是互斥锁就是基于Mutex互斥量实现的。

临界区里面存储了synchronized方法中的代码或synchronized代码块中的代码。所以临界区里面的内容同一时刻只能有一个线程访问。

锁操作：偶作锁时的方法，如wait()、signal()等。

条件变量：用来控制锁操作的变量，主要为了锁操作服务的。

在Java中Monitor是通过C语言实现的 http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/095e60e7fc8c/src/share/vm/runtime/objectMonitor.hpp

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0;
    _waiters      = 0,
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL;
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ;
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
    _previous_owner_tid = 0;
  }
```

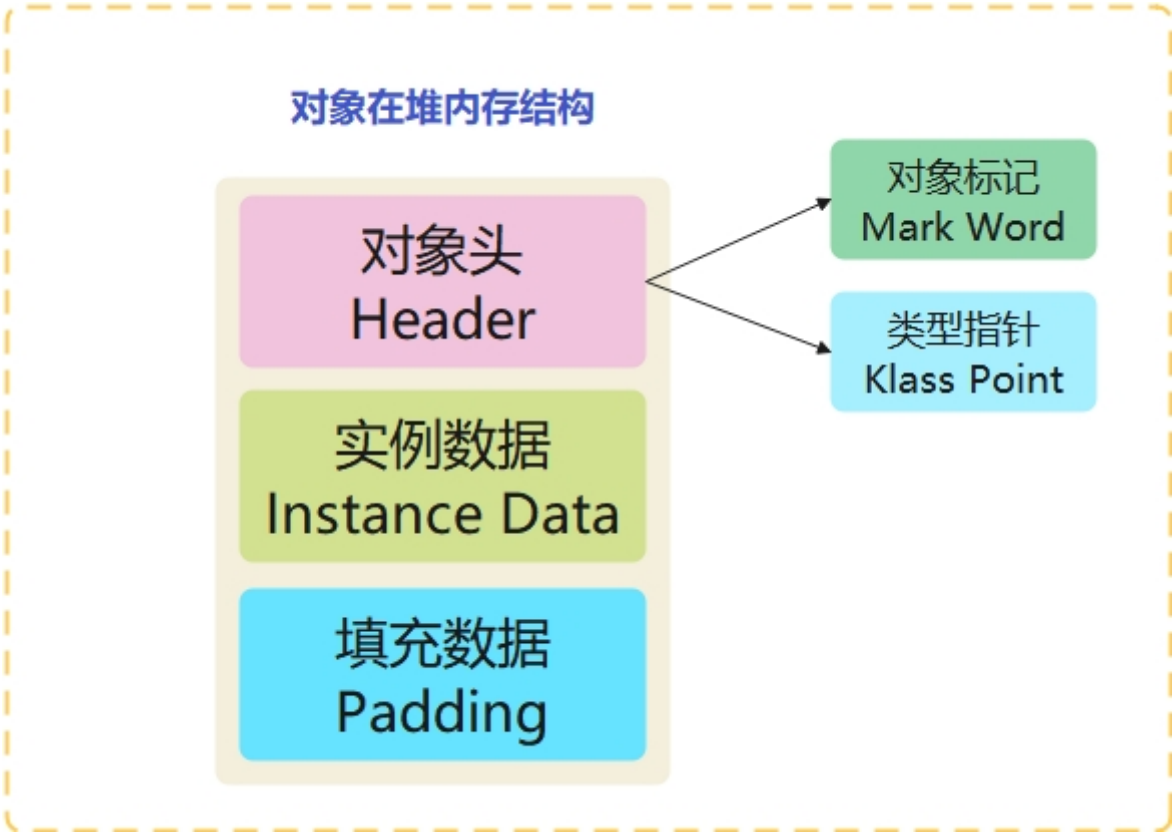
虽然里面有很多属性，但是其实*owner*、WaitSet、_EntrySet就与Monitor对象中三个组成部分相对应。

在Java官方明确说明，每个对象对应一个Monitor监视器。在JVM堆中存储对象分为三个部分：对象头、实例数据、填充数据。

其中实例数据就是对象中各个属性所占的空间。

填充数据是因为HotSpot虚拟机要求对象的字节必须是8的整数倍。所以不够的内容使用Padding补全。

最重要的对象头。对象头分为对象标记Mark Word 和类型指针。在64位虚拟机中对象标记和类型指针各占8位。



而对象头在64位虚拟机结构如下图，其中重量级锁就是指向的Monitor，而偏向锁和轻量级锁是JDK1.6之后对synchronized优化后添加的两种状态。所以synchronized底层原理其实就是在回答对象头中Mark Word在不同状态下的变化。

锁状态	56bit		1bit	4bit	1bit 偏向锁位	2bit 锁标志位
	25bit	31bit				
无锁	unused	hashcode(如果有调用)	unused	分代年龄	0	01
偏向锁	线程ID	Epoch (2bit)	unused	分代年龄	1	01
轻量级锁	指向线程栈中Lock Record的指针					00
重量级锁	指向互斥量（重量级锁）的指针					10
GC标志	CMS过程用到的标记信息					11

无锁：前25位没有被使用，接下来31bit的空间来存储对象的hashcode，在1位没有被使用。4bit用于存放对象分代年龄，1bit来表示是否是偏向锁，2bit存放锁标志位，偏向锁位与锁标志位合起来“001”就代表无锁。无锁就是没有对任何资源进行锁定，所有线程都能访问并修改资源。

偏向锁：对象头中记录了获得偏向锁的线程ID，偏向锁与锁标志位合起来“101”就代表偏向锁。在大多数情况下，锁很少被多个线程同时竞争，而且总是由同一个线程多次获得，因此只需要将获得锁的线程ID写入到锁对象Mark Word中，相当于告诉其他线程，这块资源已经被我占了。当线程访问资源结束后，不会主动释放偏向锁，当线程再次需要访问资源时，JVM就会通过Mark Word中记录的线程ID判断是否是当前线程，如果是，则继续访问资源。所以，在没有其他线程参与竞争时，锁就一直偏向被当前线程持有，当前线程就可以一直占用资源或者执行代码。

自旋锁（轻量级锁）：一旦有另外一个线程参与锁竞争，偏向锁就会升级为自旋锁，此时撤销偏向锁，锁标志位变为“00”。竞争的两个线程都在各自的线程栈帧中生成一个Lock Record空间，用于存储锁对象目前Mark Word的拷贝，用CAS操作将Mark Word设置为指向自己这个线程的LR（Lock Record）指针，设置成功者获得锁，其他参与竞争的线程如果未获取到锁，则会一直处于自旋等待的状态，直到竞

争到锁。

重量级锁：长时间的自旋操作是很消耗CPU资源的，为了避免这种盲目的消耗，JVM会在有线程超过10次自旋，或者自旋次数超过CPU核数的一半（JDK1.6以后加入了自适应自旋-Adaptive Self Spinning，由JVM自己控制自旋次数）时，会升级到重量级锁。重量级锁底层是依赖操作系统的mutex互斥锁，本质就是指向Monitor，也就是有操作系统来负责线程间的调度。重量级锁减少了自旋锁带来的CPU消耗，但是由于操作系统调度线程带来的线程阻塞会使程序响应速度变慢。此时锁标志位变为10.