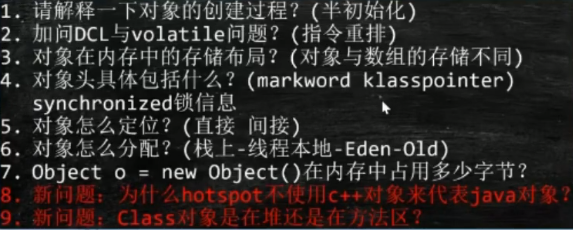
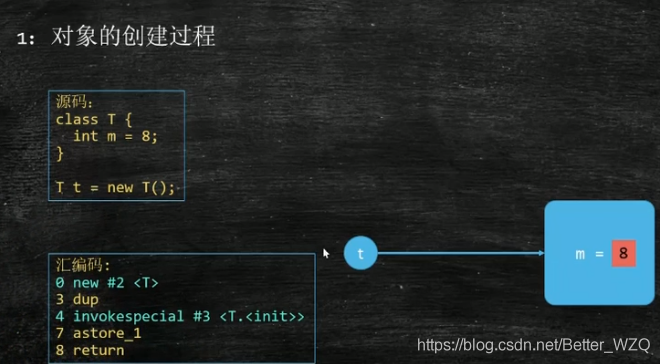
# **美团追魂连环7问**



## 经过前几天美团一面的体验感觉，明显感觉到，美团面试官对于基础的痴迷和追问，不是你看点面经什么的就可以糊弄过去的，与面试管的畅聊也激发了我想要进一步细致学习的兴趣。无意间看到马士兵老师的这个课程，写成博客记录一下 [课程资源](https://www.bilibili.com/video/BV1eE411V7Lm?p=1)关于 Object o = new Object()

## **1:对象的创建过程（半初始化）**





**①类加载检查：** 虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

举个例子，我们的string就算是对象创建，既是 String str= new String("abc");  首先也会在常量池中定位是否存在值。若不存在，则会在常量池也建立一个abc.

**②分配内存：** 在**类加载检查**通过后，接下来虚拟机将为新生对象**分配内存**。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。**分配方式**有 **“指针碰撞”** 和 **“空闲列表”** 两种，**选择那种分配方式由 Java 堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定**。

**③初始化零值：** 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

**④设置对象头：** 初始化零值完成之后，**虚拟机要对对象进行必要的设置**，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。 **这些信息存放在对象头中。** 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

**⑤执行 init 方法：** 在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始， 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行  方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

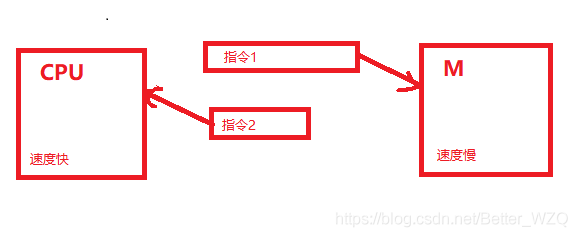
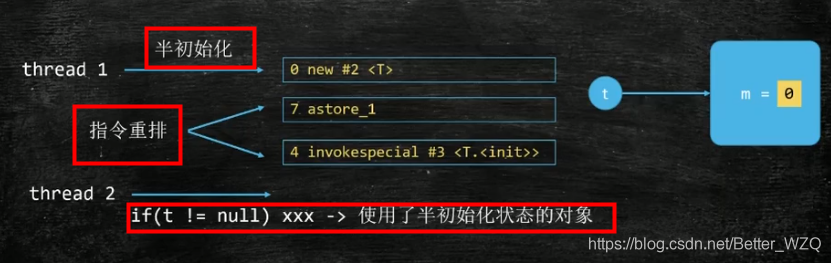
<Java程序初始化顺序>

1、父类的静态变量  
2、父类的静态代码块  
3、子类的静态变量  
4、子类的静态代码块  
5、父类的非静态变量  
6、父类的非静态代码块  
7、父类的构造方法  
8、子类的非静态变量  
9、子类的非静态代码块  
10、子类的构造方法

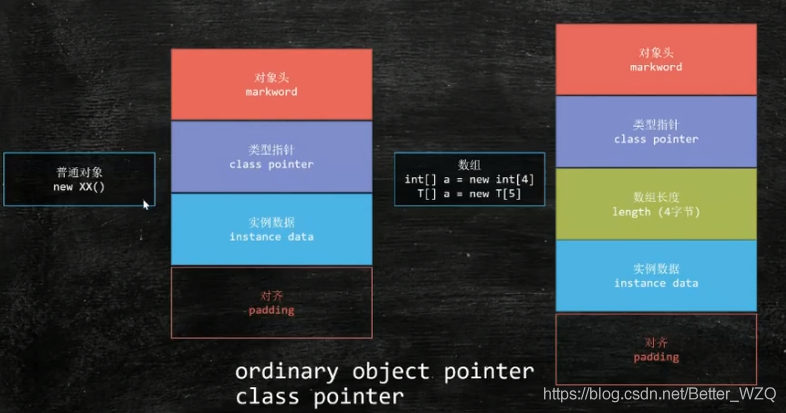
## **2：DCL与volatile问题（指令重排）**

****DCL：Double Check Lock****  
是在单例模式中，我们在getInstance()方法中判断，如果instance为null，则创建单例对象。此时，如果两个对象同时调用该方法，则instance==null判断时，都为null，则会创建出两个示例对象，就不符合单例模式了，即出现了线程不安全。为了保证线程安全，我们需要使用synchronized,添加锁机制。若直接修饰getInstance()方法，如果方法里的一些操作不需要加锁，则会降低代码的并发性，即锁的粒度太大，我们这里只需要在创建实例对象的时候，添加锁就可保证线程安全。  
第一次判断instance为null，如果为空，就申请锁，创建对象。  
（如果此时，线程A获得锁，进入创建实例对象，线程B等待，当A释放锁，B获得锁，此时instance已经不为null，但是如果我们没有进行二次判断，这时，就会再次创建新的实例对象，就不符合单例的场景了）  
所以我们需要在申请到锁后，再次判断instance是否为null。  
代码如下：

|  |
| --- |
| package singleton;  /\*  \* (1)构造器私有化  \* (2)不自行创建，只定义一个静态变量  \* (3)提供一个静态方法，获取实例对象  \* Double Check Lock  \*/  public class Singleton5 {  //要不要加volatile  //volatile:作用1 线程可见性 作用2 禁止指令重排序  private static volatile Singleton5 instance;  private Singleton5() {}  //为了线程安全，可以对getInstanc（）上锁，但是锁的范围（粒度）太大了，如果方法里的一些操作不需要加锁。  public static Singleton5 getInstance(){  //添加一个同步锁，保证线程安全  //为了保证线程安全得同时，提供效率，把对instance==null得判断放在锁得外面  if(instance==null) {  synchronized (Singleton5.class) {  //双重检查  if(instance==null) {  //测试当两个线程，先后进入到这里，休眠，那么就会创建两个实例，所以这个例子存在线程安全问题  try {  Thread.sleep(100);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  instance = new Singleton5();  }  }  }  return instance;  }  } |

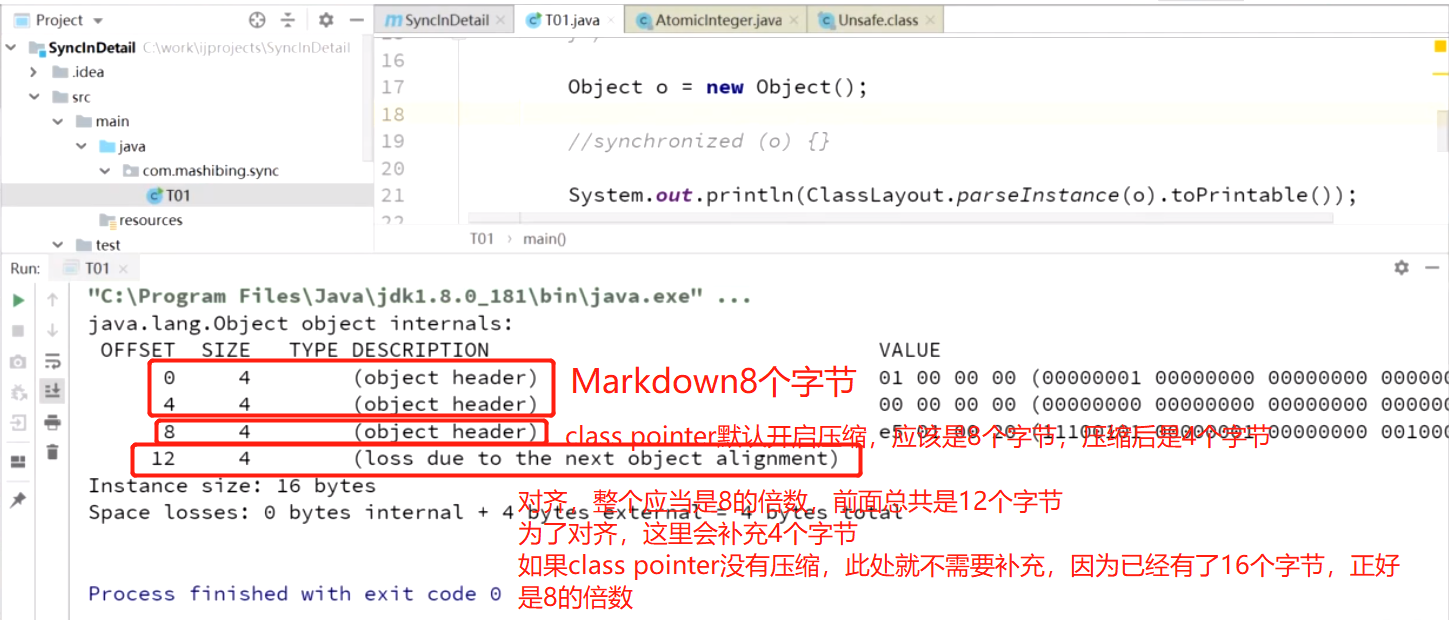
再思考一个问题：  
DCL单例到底需不需要volatile-----需要  
volatile  
作用1 ：线程可见性  
作用2 ： 禁止指令重排序  
指令重排序的意思就说，若指令1读内存很慢，cpu不会一直等待，而是先去执行与指令1无相关性的指令2，这样按理指令的执行顺序是1->2，而实际是2->1.这就是指令重排序。  
  
如果不用volatile 禁止指令重排序（底层实现是会对这块区域上下都加内存屏障），那么当线程1半初始化后，发生指令重排，先执行astore使得t去指向半初始化的对象，此时，线程2在进行排队判断的时候，就会去使用这个半初始化的对象，例如，成员变量，本应该是8，这时候是0.，从而出现错误。

## **3：对象在内存中的存储布局（对象与数组的存储不同）**

  
mark word+classpointer是对象头  
mark word（8字节）:存放有锁状态，分代年龄，hashcode等  
classpointer（4字节）：本质是指针，例如T t = new T();那么classpointer会指向T.class  
instance data（?字节）:放的是该类型实例的成员变量，若成员变量是一个int 一个string ,则为（4+4字节），此时padding 补4  
padding（？字节）:若前3项占用的空间不够8整除，会用来补齐（cpu总线在读取数据内容的时候，是希望读取的内容和它对应的位数是一一对应的，才是最好。对于64位的计算机，它的寄存器也是64位，就希望我们读到数据的大小能刚好被8整除，可以提高效率）

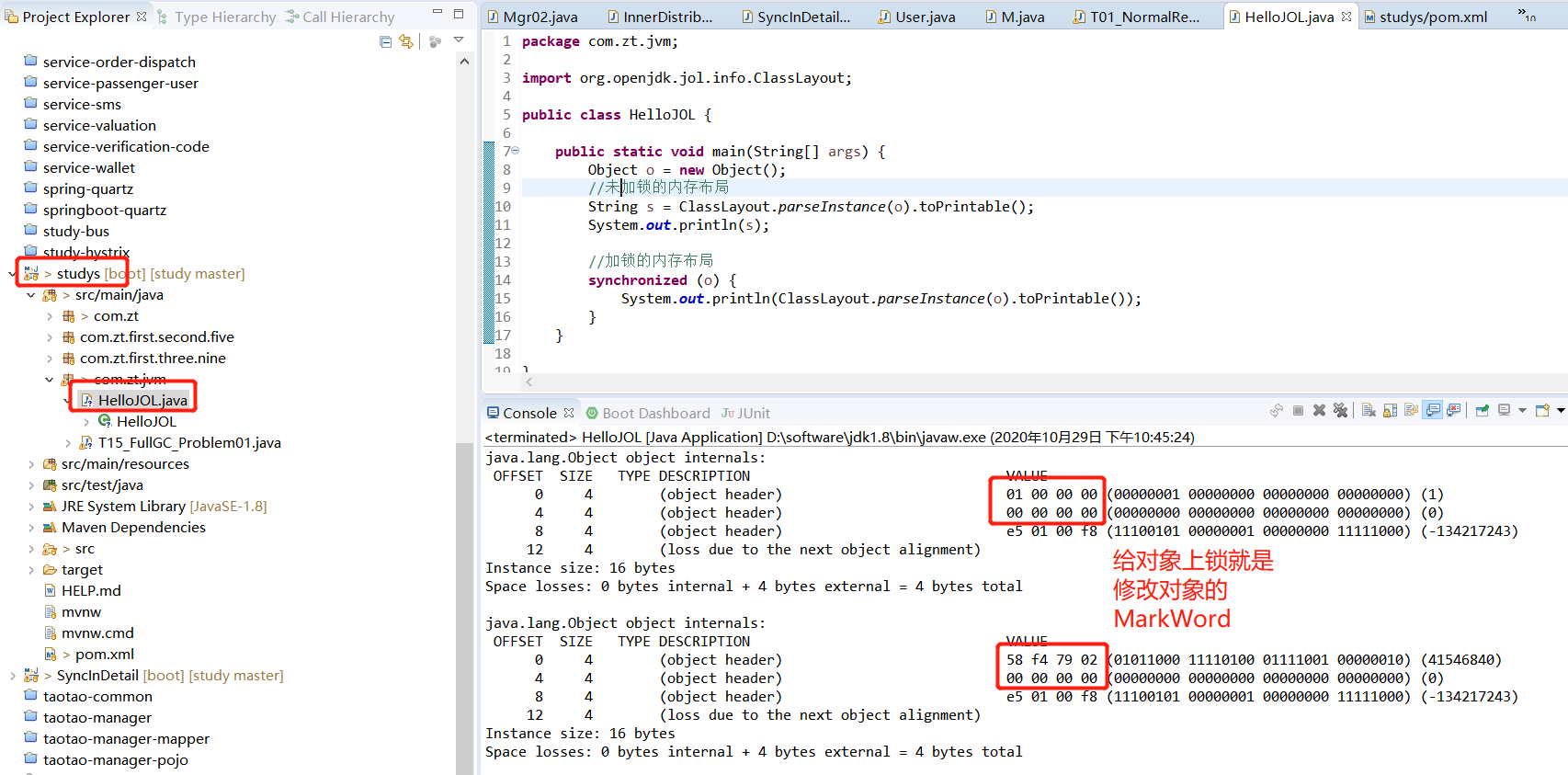
## **4：对象头具体包括什么（markword classpointer）synchronized锁信息**

包括：锁信息，GC信息，hashcode(原始的hashcode，不是重写的那种)

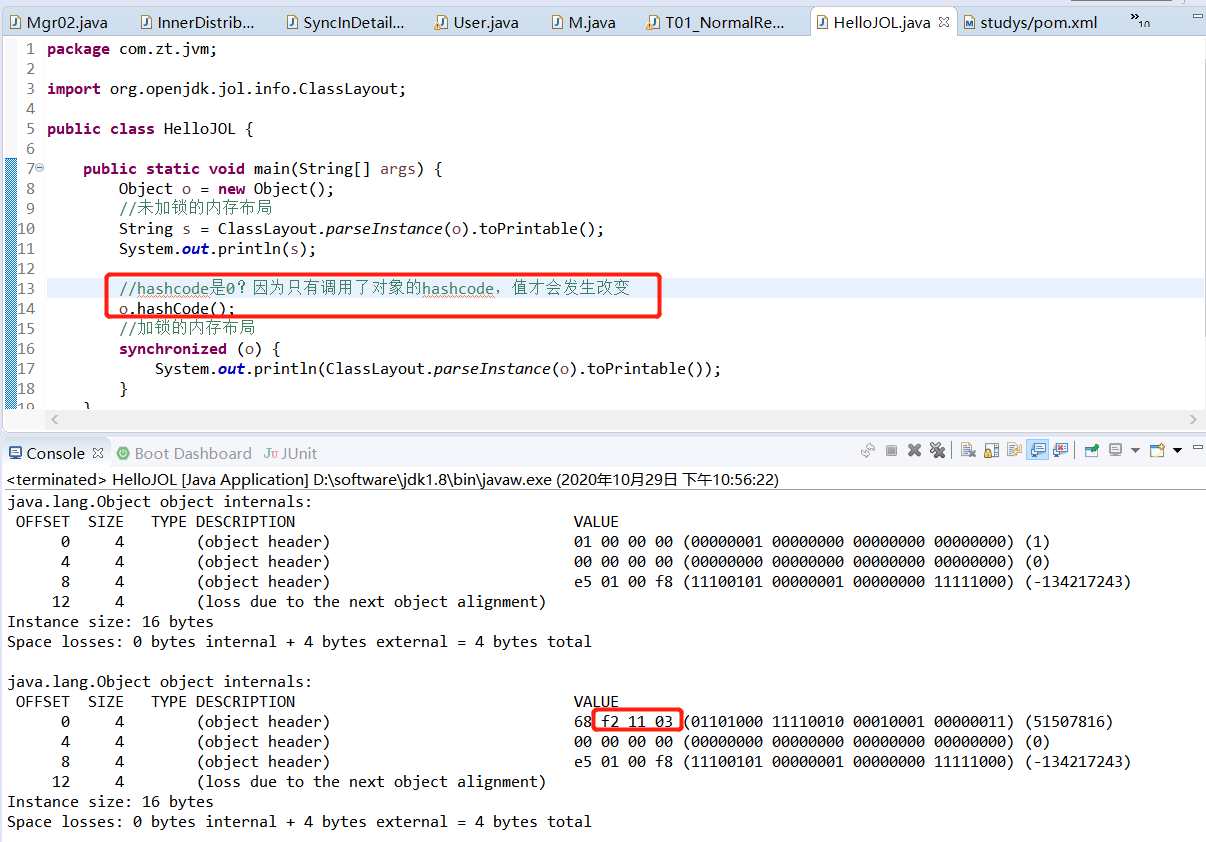


|  |
| --- |
| <dependency>  <groupId>org.openjdk.jol</groupId>  <artifactId>jol-core</artifactId>  <version>0.10</version>  </dependency> |

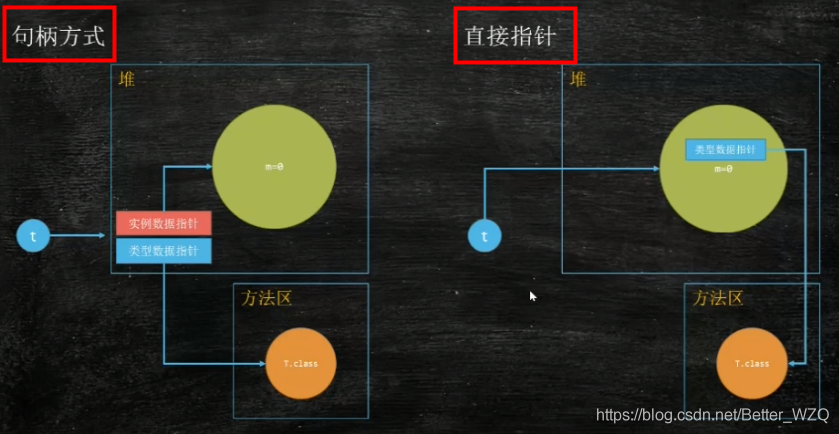
|  |
| --- |
| **package** com.zt.jvm;  **import** org.openjdk.jol.info.ClassLayout;  **public** **class** HelloJOL {  **public** **static** **void** main(String[] args) {  Object o = **new** Object();  //未加锁的内存布局  String s = ClassLayout.*parseInstance*(o).toPrintable();  System.***out***.println(s);  //加锁的内存布局  **synchronized** (o) {  System.***out***.println(ClassLayout.*parseInstance*(o).toPrintable());  }  }  } |



图片显示hashcode是0？因为只有调用了对象的hashcode，值才会发生改变。



## **5：对象怎么定位（直接间接）**

直接：直接指针（Hotspot使用）  
间接：句柄方式  
  
句柄方式：  
优点：对象小，垃圾回收时不用频繁改动t  
缺点：两次访问

直接指针：

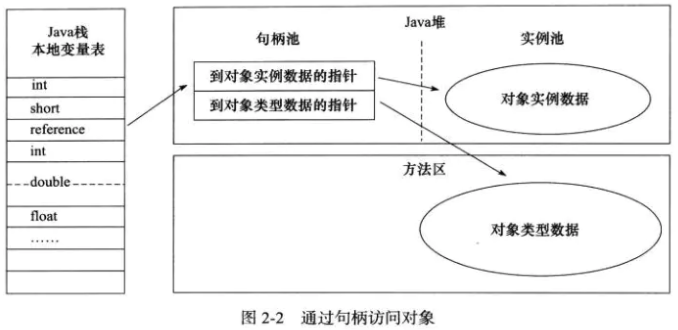
优点：直接访问

缺点：GC需要移动对象是稍麻烦

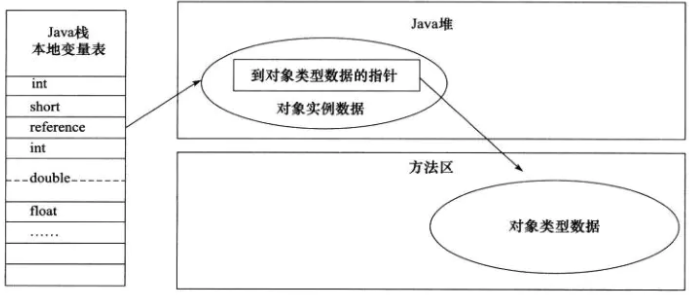
<对象的访问定位>

建立对象就是为了使用对象，我们的Java程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有**①使用句柄**和**②直接指针**两种：

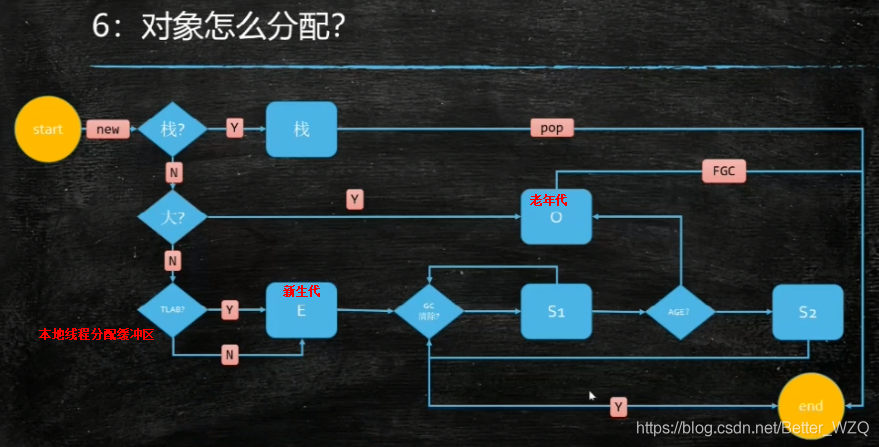
**句柄：** 如果使用句柄的话，那么Java堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

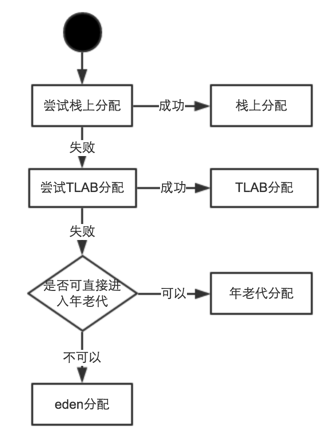
对象实例就是new出来的对象！存在于JAVA堆中！   而class的一些信息则存在于方法区中！就是对象类型数据！！详见方法区。 句柄池就存放这两个指针！

**直接指针：** 如果使用直接指针访问，那么 Java 堆对像的布局中就必须考虑如何防止访问类型数据的相关信息，reference 中存储的直接就是对象的地址。

**这两种对象访问方式各有优势。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。**

## **6：对象怎么分配（栈上-线程本地-Eden -Old）**



  
栈上分配原则：没有逸出（别的方法不用这个对象），标量替换（直接存两个成员变量）  
如果在栈上分配，结束时，直接出栈，不需要GC的介入，效率高（通过设置Xss）  
TLAB（Threadlocal Allocation Buffer 线程本地分配缓冲区,每个线层都在Eden区有自己专属的空间，当自己的线程要分配对象时，优先往自己的TLAB里分配，不用和别的线程竞争--因为多个线程竞争同一块资源<碰撞指针>，JVM需要协调多个指针，损耗资源）

PS/PO不适合分布式。

## **7: Object o = new Object()在内存中占多少字节**

主要是要看是否开启压缩：  
压缩有两种:1 压缩classpointer 2:压缩普通对象指针  
o为普通对象指针（OOPS）：4个字节  
16+4 = 20个字节