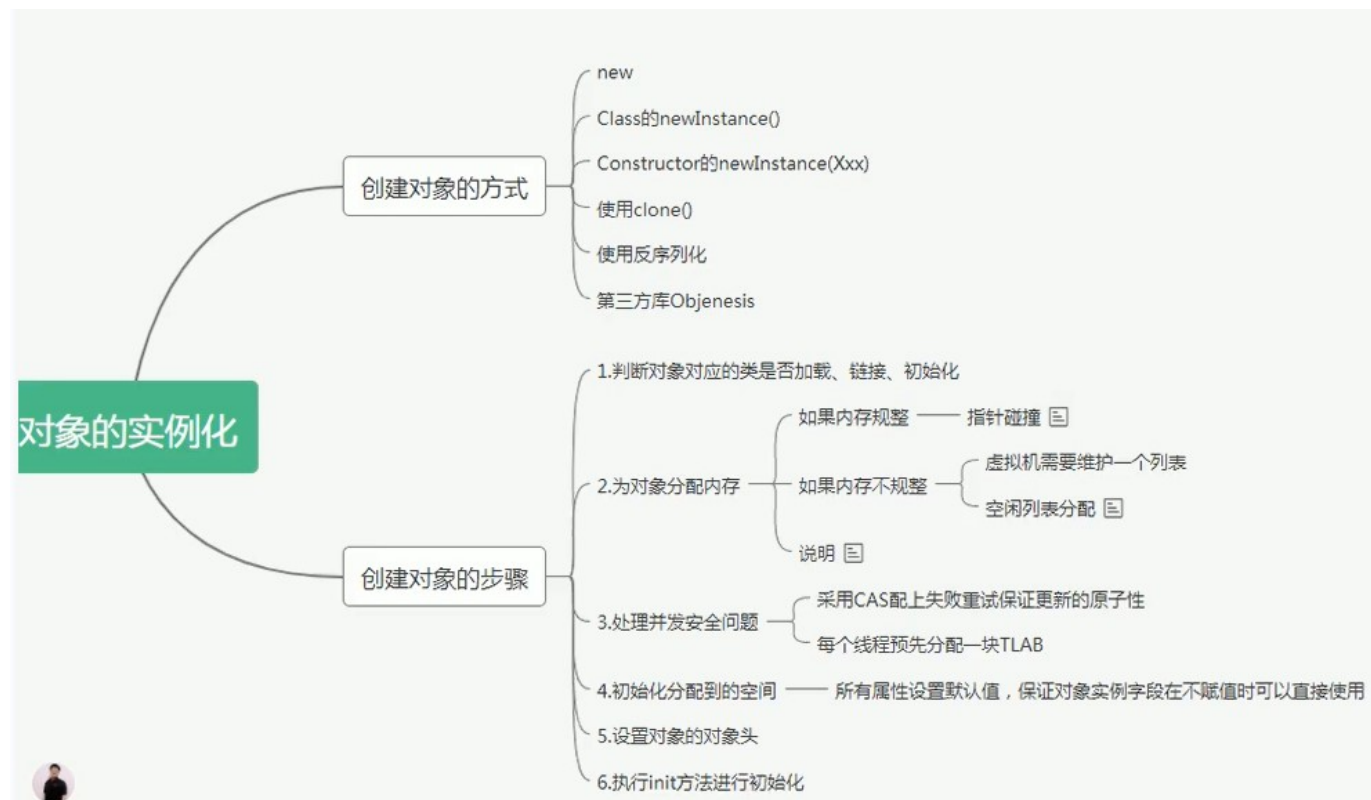
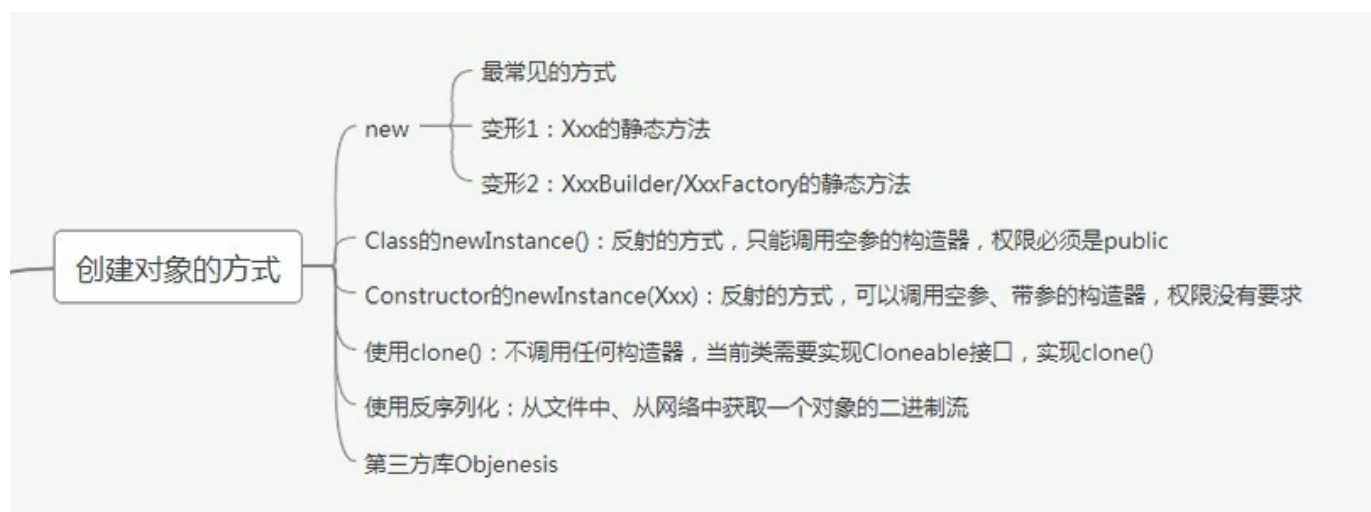


对象的实例化



创建对象的方式



字节码角度看对象创建的过程

```

1 public class ObjectTest {
2     public static void main(String[] args) {
3         Object obj = new Object();
4     }
5 }

```

```
public static void main(java.lang.String[]);
```

```
descriptor: ([Ljava/lang/String;)V
```

```
flags: ACC_PUBLIC, ACC_STATIC
```

```
Code:
```

```
stack=2, locals=2, args_size=1
```

```

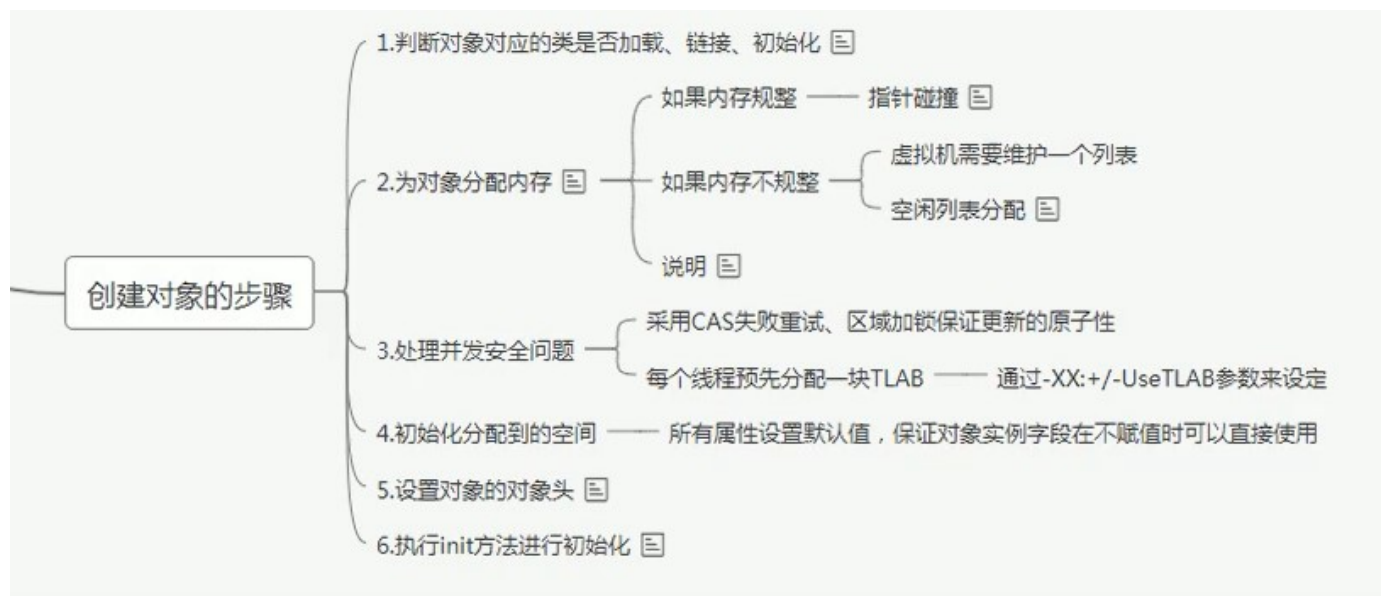
0: new          #2                // class java/lang/Object
3: dup
4: invokespecial #1                // Method java/lang/Object."<init>":()V
7: astore_1
8: return

```

```
LineNumberTable:
```

```
-1
```

从执行步骤角度分析, 创建对象



1. 判断对象对应的类是否加载、链接、初始化

虚拟机遇到一条new指令, 首先去检查这个指令的参数是否在Metaspace的常量池中定位到一个类的符号引用, 并且检查这个符号引用代表的类是否已经被加载、解析和初始化;

也就是判断类元信息是否存在;

如果没有, 那么在双亲委派模式下, 使用当前类加载器以ClassLoader+包名+类名为

Key进行查找对应的.class 文件;

如果没有找到文件, 则抛出ClassNotFoundException 异常;

如果找到, 则进行类加载, 并生成对应的Class 类对象;

2. 为对象分配内存

首先计算对象占用空间大小, 接着在堆中划分一块内存给新对象;

如果实例成员变量是引用变量, 仅分配引用变量空间即可, 即4个字节大小;

- 如果内存规整, 使用指针碰撞

如果内存是规整的,那么虚拟机将采用的是指针碰撞法(Bump The Pointer)来为对象分配内存; 意思是所有用过的内存存在一边, 空闲的内存存在另外一边, 中间放着一个指针作为分界点的指示器, 分配内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了;如果垃圾收集器选择的是Serial、ParNew这种基于压缩算法的, 虚拟机采用这种分配方式; 一般使用带有compact(整理)过程的收集器时, 使用空闲列表分配;

- 如果内存不规整, 虚拟机需要维护一个列表, 使用空闲列表分配

如果内存不是规整的, 已使用的内存和未使用的内存相互交错, 那么虚拟机将采用的是空闲列表法为对象分配内存; 意思是虚拟机维护了一个列表, 记录上哪些内存块是可用的, 再分配的时候从列表中找到一块足够大的空间划分给对象实例, 并更新列表上的内容; 这种分配方式称为 "空闲列表 (Free List)";

说明: 选择哪种分配方式, 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. 处理并发安全问题

在分配内存空间时, 另外一个问题是及时保证new对象时候的线程安全性: 创建对象是非常频繁的操作, 虚拟机需要解决并发问题; 虚拟机采用了两种方式解决并发问题:

- CAS (Compare And Swap) 失败重试, 区域加锁: 保证指针更新操作的原子性;
- TLAB 把内存分配的动作按照线程划分在不同的空间之中进行, 即每个线程在Java堆中

预先分配一小块内存, 称为本地线程分配缓冲区(TLAB, Thread Local Allocation Buffer)
虚拟机是否使用TLAB, 可以通过-XX:+/-UseTLAB参数来设定;

4. 初始化分配到空间

内存分配结束, 虚拟机将分配到的内存空间都初始化为零值(不包括对象头);

这一步保证了对象的实例字段在Java代码中可以不用赋初始值就可以直接使用, 程序能访问到这些字段的数据类型所对应的零值;

5. 设置对象的对象头

将对象的所属类 (即类的元数据信息)、对象的HashCode和对象的GC信息、锁信息等数据存储在对象的对象头中; 这个过程的具体设置方式取决于JVM实现;

6. 执行init方法进行初始化

在Java程序的视角看来, 初始化才正式开始;

初始化成员变量, 执行实例化代码块, 调用类的构造方法, 并把堆内对象的首地址赋值给引用变量;

因此一般来说 (由字节码中是否跟随有invokespecial指令所决定), new指令之后会接着就是执行方法, 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创建出来;

类创建的过程, 第4步赋初始值, 第6步初始化; 这里要和类加载过程的 prepare 和init区分, 那个初始化的是 static属性, 这里是成员变量;

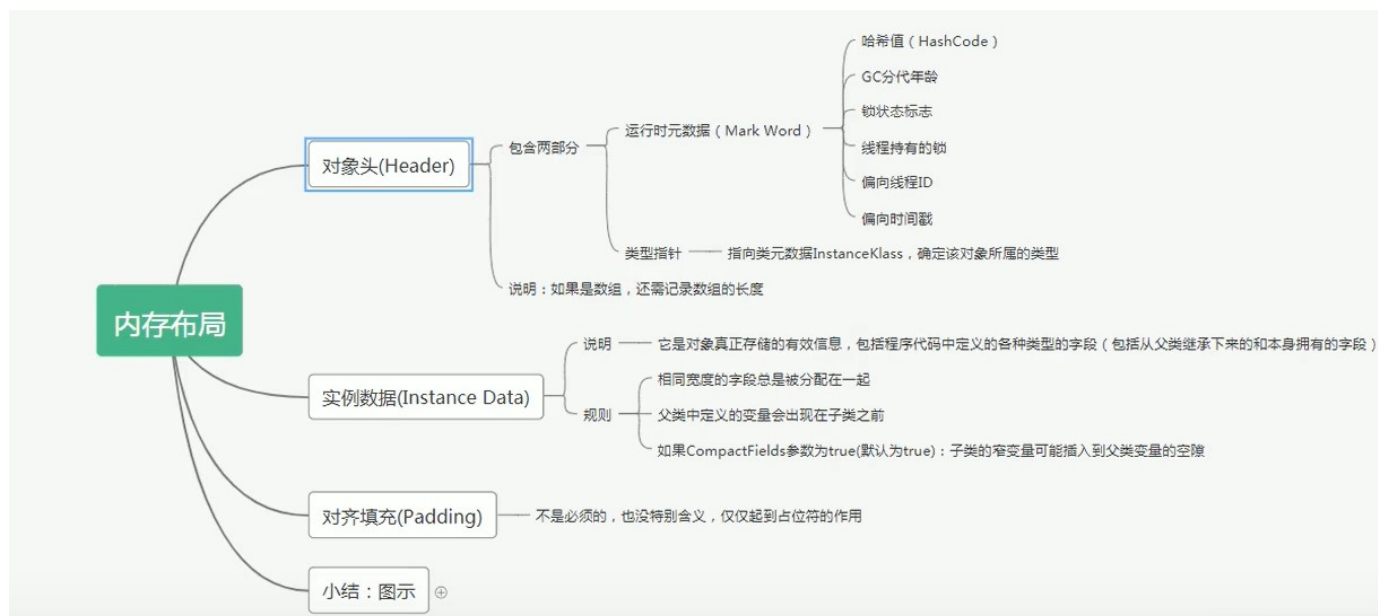
```
1  /**
2   * 测试对象实例化的过程
3   * ① 加载类元信息 - ② 为对象分配内存 - ③ 处理并发问题 - ④ 属性的默认初始化 (零值初
4   * - ⑤ 设置对象头的信息 - ⑥ 属性的显式初始化、代码块中初始化、构造器中初始化
5   *
```

```
6  *
7  * 给对象的属性赋值的操作:
8  * ① 属性的默认初始化 - ② 显式初始化 / ③ 代码块中初始化 - ④ 构造器中初始化
9  * @author shkstart shkstart@126.com
10 * @create 2020 17:58
11 */
12
13 public class Customer{
14     int id = 1001;
15     String name;
16     Account acct;
17
18     {
19         name = "匿名客户";
20     }
21     public Customer(){
22         acct = new Account();
23     }
24
25 }
26 class Account{
27
28 }
```

<https://www.bilibili.com/video/BV1PJ411n7xZ?p=104>

对象的内存布局

new的对象, 有哪几部分结构:



小结

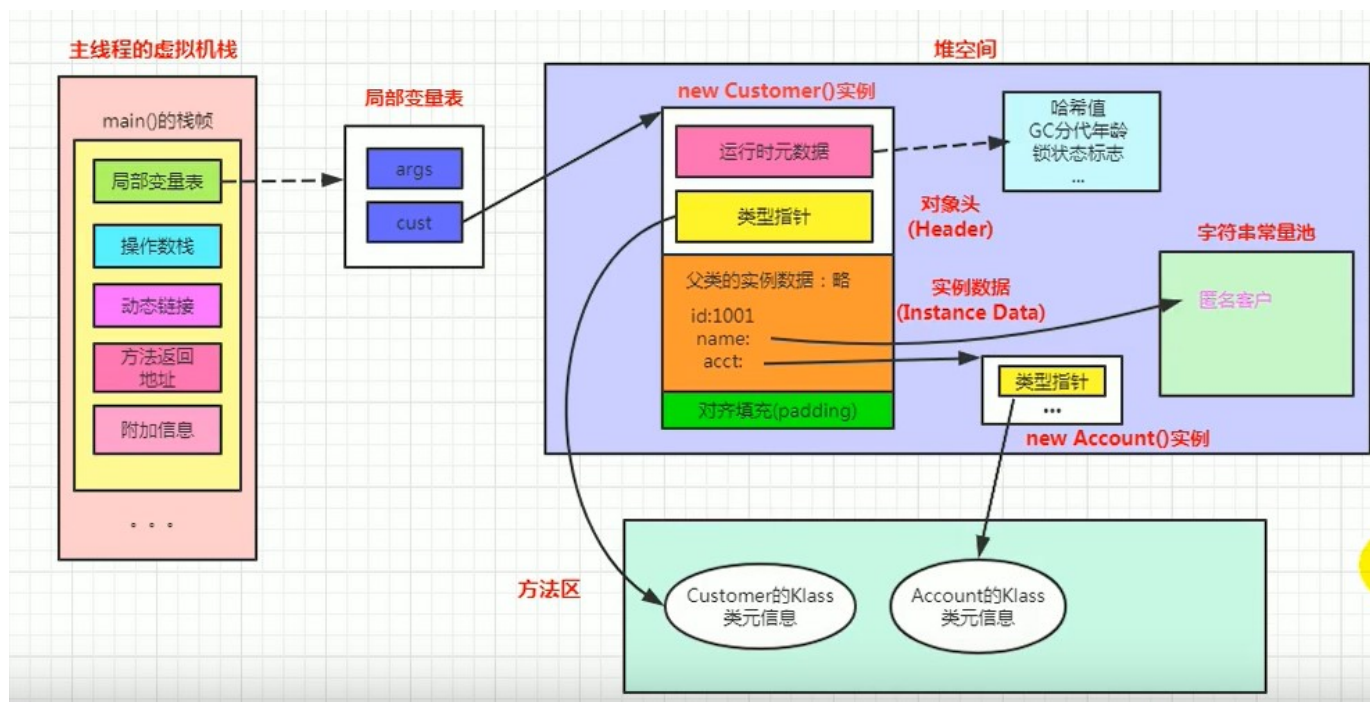
代码示例

```

1 public class CustomerTest {
2     public static void main(String[] args) {
3         Customer cust = new Customer(); // 上面一个代码
4     }
5 }

```

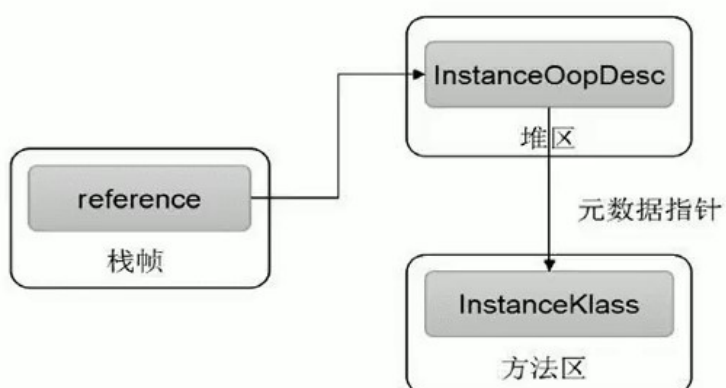
图示



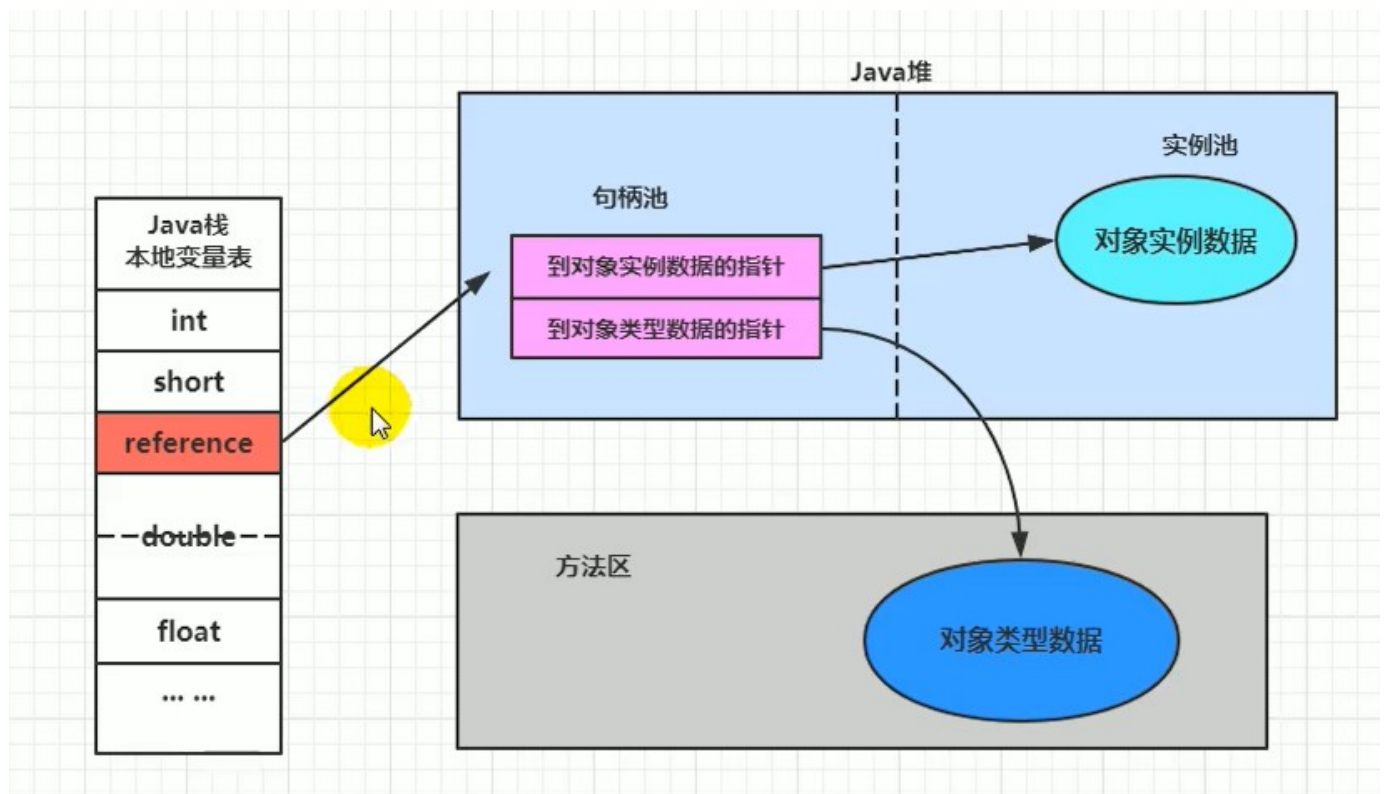
对象的访问定位



JVM是如何通过栈帧中的对象引用访问到其内部的对象实例的呢？



句柄访问

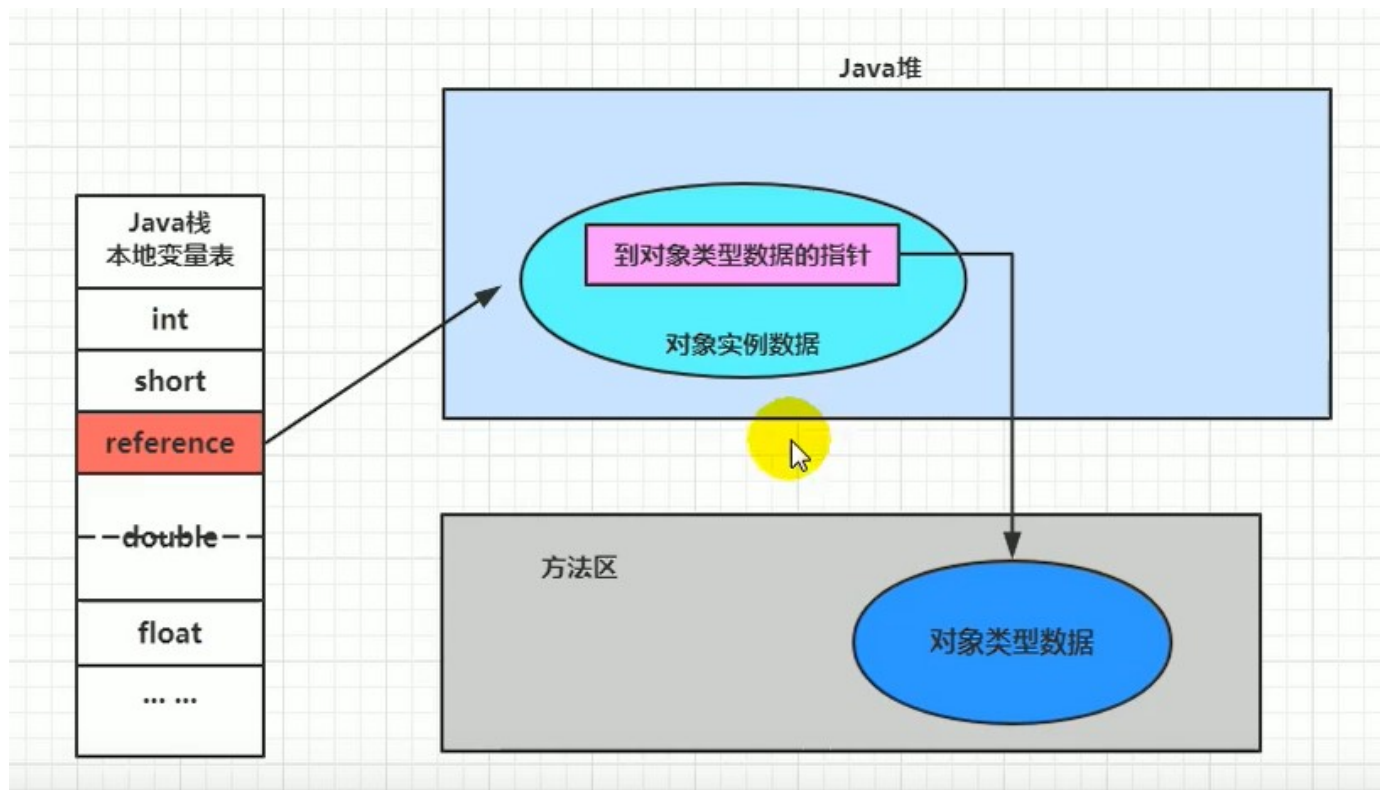


java堆 开辟了一个 句柄池

优点:

reference中存储稳定的句柄地址, 对象被移动(垃圾收集时移动对象很普遍)时, 只会改变句柄中实例数据指针即可, reference本身不需要被修改

直接指针 (HotSpot采用)



直接指向 对象实例

优点:

省空间, 直接查询, 速度快