java内存管理（堆、栈、方法区）：https://blog.csdn.net/Liveor\_Die/article/details/77895631

## 对象创建

创建一个对象通常是需要new关键字，当虚拟机遇到一条new指令时，首先检查这个指令的参数是否在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被加载、解析和初始化过。如果那么执行相应的类加载过程。

　　类加载检查通过后，虚拟机将为新生对象分配内存。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。

**分配的方式**有两种：

　　　一种叫 **指针碰撞** ，假设Java堆中内存是绝对规整的，用过的和空闲的内存各在一边，中间放着一个指针作为分界点的指示器，分配内存就是把那个指针向空闲空间的那边挪动一段与对象大小相等的距离。

　　　另一种叫 **空闲列表** ：如果Java堆中的内存不是规整的，虚拟机就需要维护一个列表，记录哪个内存块是可用的，在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，并更新列表上的记录。

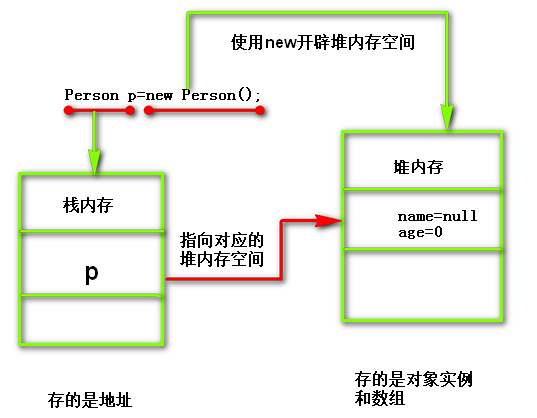
　　采用哪种分配方式是由**\*Java堆是否规整决定的，而Java堆是否规整是由所采用的垃圾收集器是否带有压缩整理功能决定的。** 另 外一个需要考虑的问题就是对象创建时的线程安全问题\*，有两种解决方案：一是对分配内存空间的动作进行同步处理；另一种是吧内存分配的动作按照线程划分在不 同的空间之中进行，即每个线程在Java堆中预先分配一小块内存(TLAB)，哪个线程要分配内存就在哪个线程的TLAB上分配，只有TLAB用完并分配 新的TLAB时才需要同步锁定。

　　内存分配完成后，虚拟机需要将分配到的内存空间初始化为零值。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可以直接使用。

　　接下来虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息等，这些信息存放在对象的对象头中。

　　上面的工作都完成以后，从虚拟机的角度来看一个新的对象已经产生了。但是从Java程序的角度，还需要执行init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。

执行过程如图：



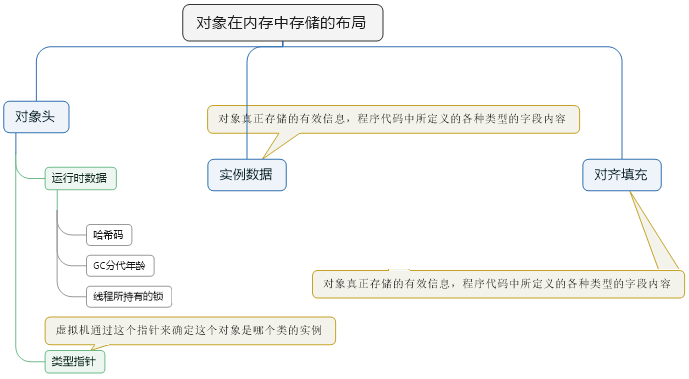
## 对象的内存布局

在HotSpot虚拟机中，对象在内存中存储的布局可分为三个部分： 对象头、实例数据和对齐填充。

　　对象头包括两个部分：第一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄、线程所持有的锁等。官方称之为“Mark Word”。第二个部分为是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

　　实例数据是对象真正存储的有效信息，也是程序代码中所定义的各种类型的字段内容。

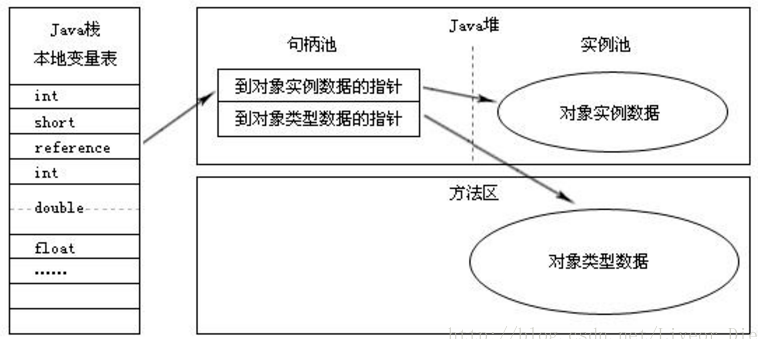
　　对齐填充并不是必然存在的，仅仅起着占位符的作用。、Hotpot VM要求对象起始地址必须是8字节的整数倍，对象头部分正好是8字节的倍数，所以当实例数据部分没有对齐时，需要通过对齐填充来对齐。



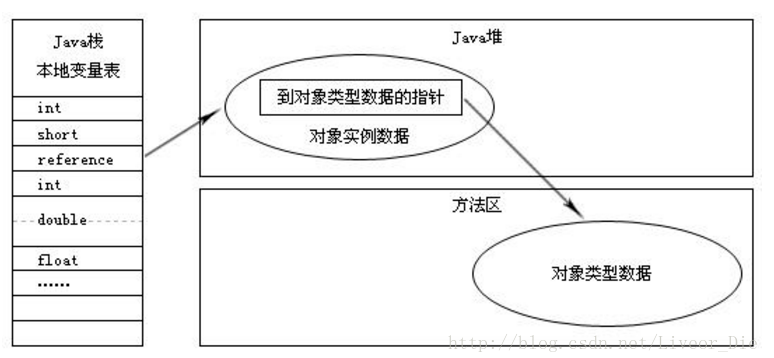
## 对象的访问定位

ava程序通过栈上的reference数据来操作堆上的具体对象。目前主流的访问方式由“使用句柄”和“直接指针”。

　　如果使用句柄访问的话，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的具体各自的地址信息。如下图：



如果使用**直接指针访问**的话，Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference中存储的直接就是**对象地址**，如下图：



对比

　　使用句柄来访问的最大好处就是reference中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要被修改。

　　使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问的在Java中非常频繁，因此这类开销积小成多也是一项非常可观的执行成本。