## 深入理解JVM虚拟机

### 二、自动内存管理

#### 1、Java内存区域及内存溢出异常

##### 1）运行时的数据区域

注意：HotSpot不分虚拟机栈和本地方法栈



###### A）程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器

###### B）Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stack）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的线程内存模型：每个方法被执行的时候，Java虚拟机都会同步创建一个**栈帧**（StackFrame）用于**存储局部变量表、操作数栈、动态连接、方法出口**等信息。每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

###### C）本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别只是拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用的本地（Native）方法服务。

###### D）Java堆

Java堆（Java Heap）是虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是**存放对象实例**，Java世界里“几乎”所有的对象实例都在这里分配内存。在《Java虚拟机规范》中对Java堆的描述是：“**所有的对象实例以及数组都应当在堆上分配**”，而这里笔者写的“几乎”是指从实现角度来看，随着Java语言的发展，现在已经能看到些许迹象表明日后可能出现值类型的支持，即使只考虑现在，由于即时编译技术的进步，尤其是逃逸分析技术的日渐强大，栈上分配、标量替换优化手段已经导致一些微妙的变化悄然发生，所以说Java对象实例都分配在堆上也渐渐变得不是那么绝对了。

从回收内存的角度看，由于**现代垃圾收集器大部分都是基于分代收集理论设计**的，所以Java堆中经常会出现“新生代”“老年代”“永久代”“Eden空间”“From Survivor空间”“To Survivor空间”等名词

（以G1收集器的出现为分界），作为业界绝对主流的HotSpot虚拟机，它内部的垃圾收集器全部都基于“经典分代”来设计，需要新生代、老年代收集器搭配才能工作，在这种背景下，上述说法还算是不会产生太大歧义

###### E）方法区

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已**被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据**。虽然《Java虚拟机规范》中把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫作“非堆”（Non-Heap），目的是与Java堆区分开来。

在JDK 6的时候HotSpot开发团队就有放弃永久代，逐步改为采用本地内存（Native Memory）来实现方法区的计划了，到了**JDK 7的HotSpot**，已经把原本放在永久代的**字符串常量池、静态变量等移出**，而**到了JDK 8，终于完全废弃了永久代的概念**，改用与JRockit、J9一样在本地内存中实现的**元空间**（Meta-space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

###### F）运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class文件中除了有**类的版本、字段、方法、接口等描述信息**外，还有一项信息是**常量池表**（Constant Pool Table），用于存放编译期生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

###### G）直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是《Java虚拟机规范》中定义的内存区域。但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现，所以我们放到这里一起讲解。显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括物理内存、SWAP分区或者分页文件）大小以及处理器寻址空间的限制，一般服务器管理员配置虚拟机参数时，会根据实际内存去设置-Xmx等参数信息，但经常忽略掉直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常。

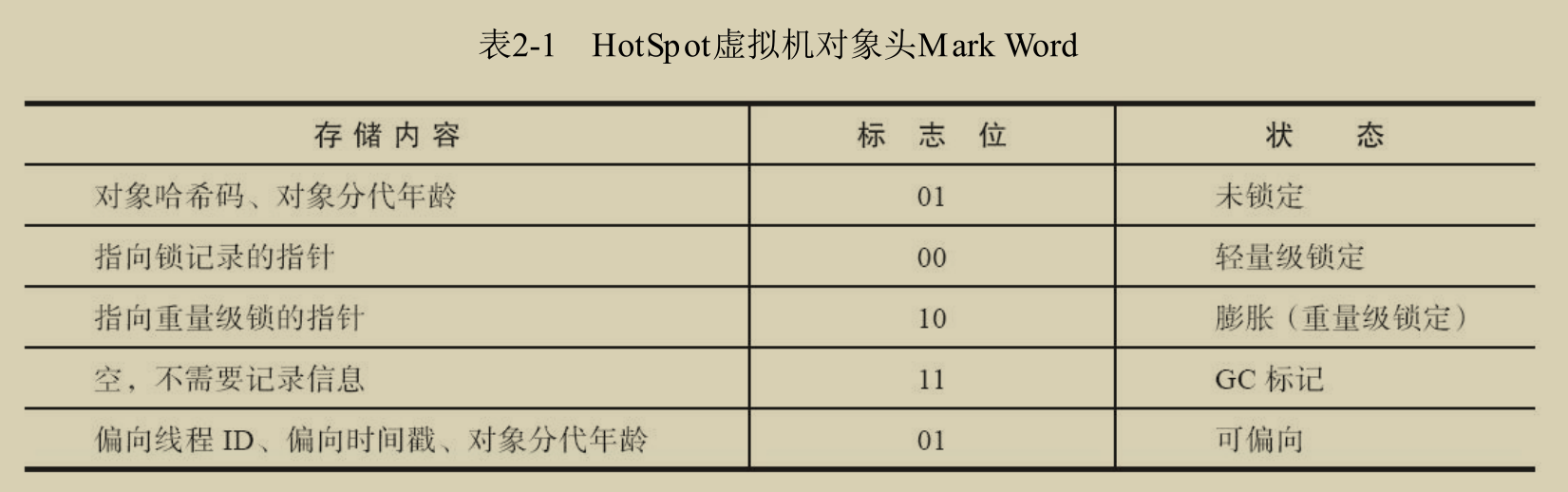
##### 2）对象的内存布局

在HotSpot虚拟机里，对象在堆内存中的存储布局可以划分为三个部分：

对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

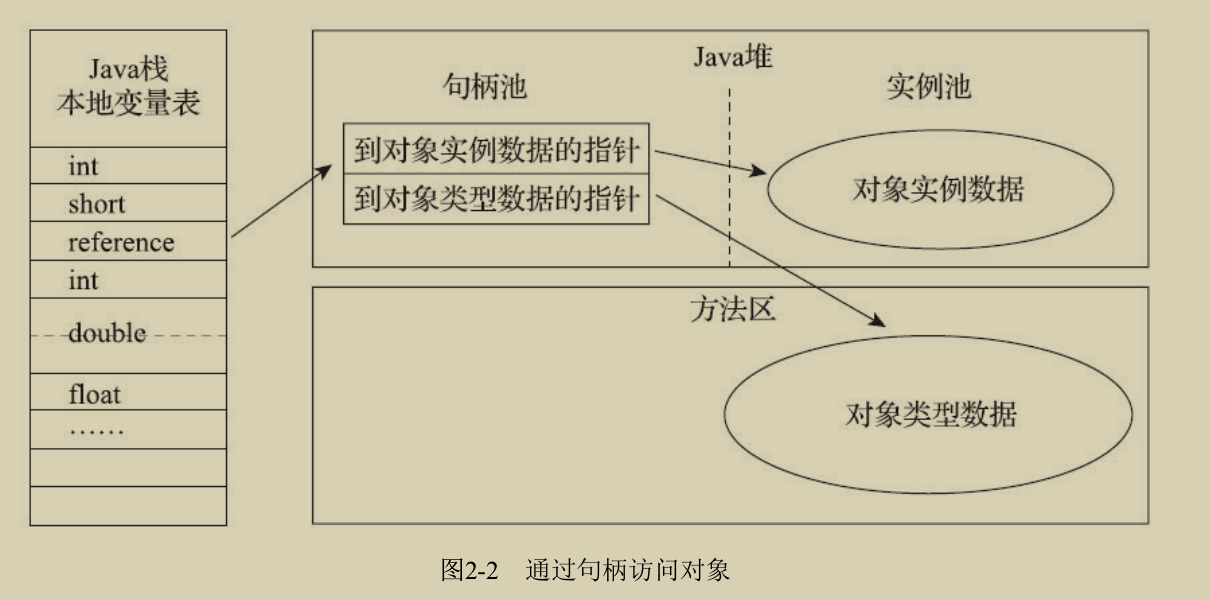
HotSpot虚拟机对象的对象头部分包括两类信息。

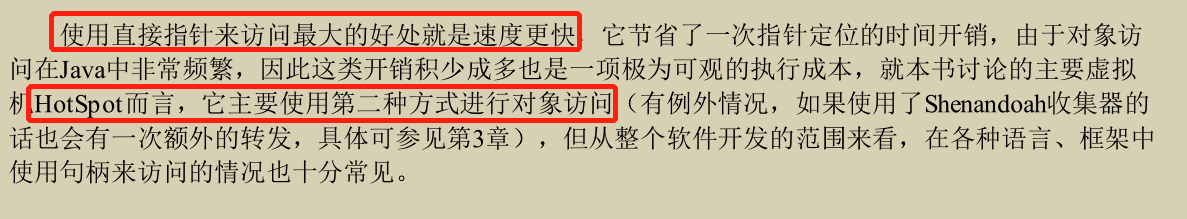
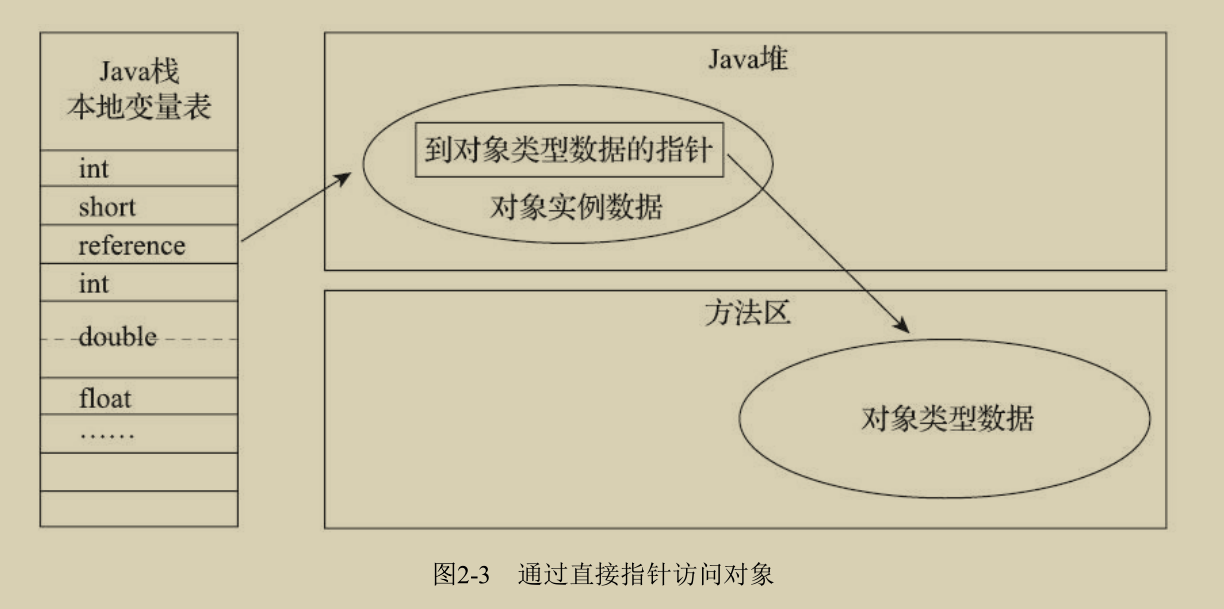
第一类是用于**存储对象自身的运行时数据**，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等



对象头的另外一部分是**类型指针**，即对象指向它的类型元数据的指针，Java虚拟机通过这个指针来确定该对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针

##### 3）对象的访问定位





##### ★★★★★4）OOM实战

###### HotSpot：参数

内存溢出：

-Xms 初始堆大小 物理内存的1/64(<1GB) 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制.

-Xmx 最大堆大小 物理内存的1/4(<1GB) 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制

-Xmn 年轻代大小(1.4or lator)

注意：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。

整个堆大小=年轻代大小 + 年老代大小 + 持久代大小.增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8

栈溢出：

·-XX：MaxMetaspaceSize：设置元空间最大值，默认是-1，即不限制，或者说只受限于本地内存大小。

·-XX：MetaspaceSize：指定元空间的初始空间大小，以字节为单位，**达到该值就会触发垃圾收集进行类型卸载**，同时收集器会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过-XX：MaxMetaspaceSize（如果设置了的话）的情况下，适当提高该值。

·-XX：MinMetaspaceFreeRatio：作用是在垃圾收集之后控制最小的元空间剩余容量的百分比，可减少因为元空间不足导致的垃圾收集的频率。类似的还有-XX：Max-MetaspaceFreeRatio，用于控制最大的元空间剩余容量的百分比

#### 2、垃圾收集器与内存分配策略

#### 3、虚拟机性能监控及故障处理工具

#### 4、调优案例分析实战

##### 1）Java技术体系