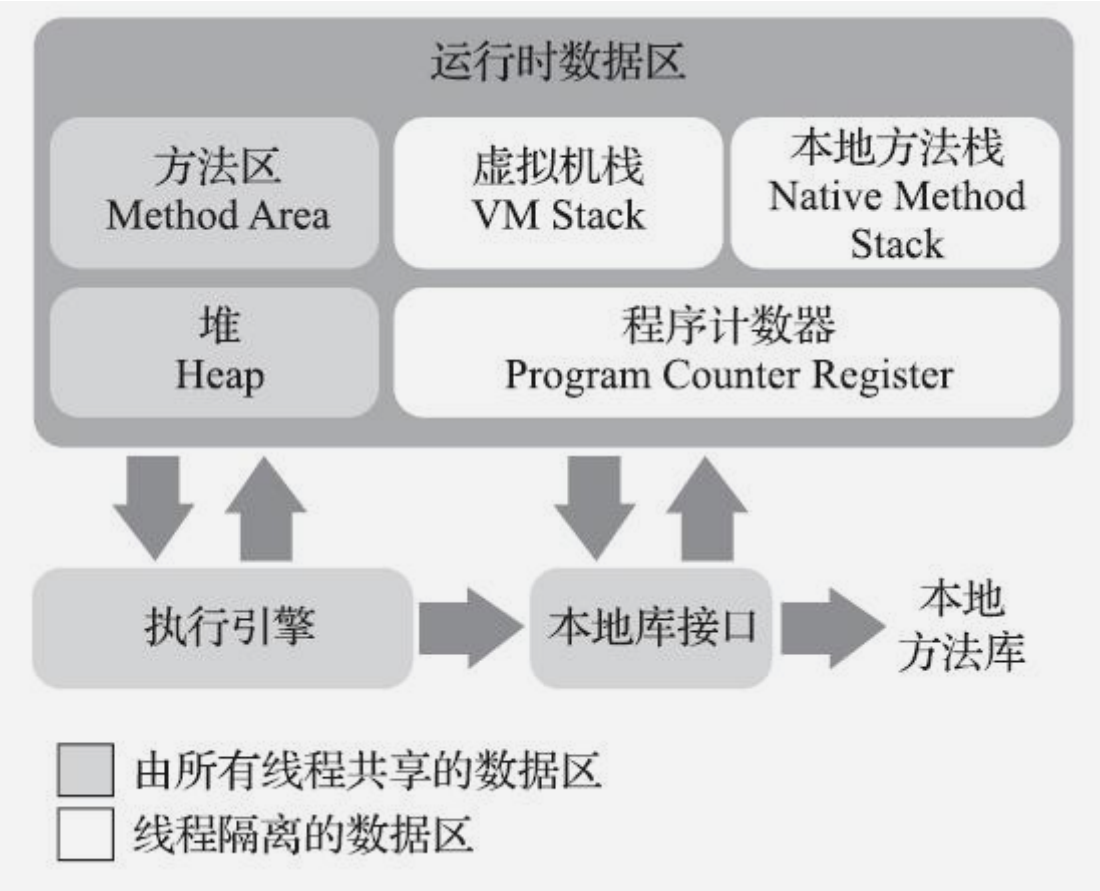
2 Java内存区域与内存溢出异常

2.1 概述

在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去写配对的delete/free代码，不容易出现内存泄漏和溢出方面的问题。

2.2 运行时数据区域

Java虚拟机的运行时数据区：



2.2.1 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。在Java虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换、分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都会执行一条程序中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，即“线程私有”的内存。

如果正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的位置；如果正在执行的是本地（Native）方法，这个计数器值则应为空（Undefined）。

此区域是唯一一个在《Java虚拟机规范》中没有规定任何OutOfMemoryError情况的区域。

2.2.2 Java虚拟机栈

Java Virtual Machine Stack也是线程私有的，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**：每个方法被执行时，虚拟机都会同步创建一个栈帧（Stack Frame）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。

局部变量表存放了编译期可知的各种Java虚拟机基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用类型和returnAddress类型（指向了一条字节码指令的地址）。

这些数据类型在局部变量表中的存储空间以局部变量槽（Slot）来表示，64位长度的long和double类型的数据会占用两个变量槽，其余只占用一个。

在《Java虚拟机规范》中，对Java虚拟机栈规定了两类异常状况：若线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；若Java虚拟机栈无法申请到足够的内存会抛出OutOfMemoeyError异常。

2.2.3 本地方法栈

Native Method Stack与虚拟机栈所发挥的作用是非常相似的，其区别只是虚拟机栈为虚拟机执行Java方法（也即字节码）服务，而本地方法栈则是为虚拟机使用到的本地方法服务。

该区域规定的异常情况与虚拟机栈一样。

2.2.4 Java堆

对于Java应用来说。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。此内存区域唯一的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。

Java堆是垃圾收集管理的内存区域，因此一些资料中它也被称为GC堆。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”等名词。

Java堆既可以被实现成固定大小的，也可以是可扩展的，不过当前主流的Java虚拟机都是按照可扩展来实现的（通过参数-Xmx和-Xms设定）。若Java队中没有内存完成实例分配，且堆也无法再扩展时，Java虚拟机将会抛出OutOfMemory异常。

2.2.5 方法区

Method Area与Java堆一样，是线程共享的内存区域，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。

《Java虚拟机规范》对方法区的约束是非常宽松的，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，甚至还可以选择不实现垃圾收集。垃圾收集行为在这个区域相对而言较少出

现。该区域的内存回收目标主要是针对常量池的回收和对类型的卸载。
根据《Java虚拟机规范》的规定，方法区无法满足新的内存分配需求时，将抛出OutOfMemory异常。

2.2.6 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，用于存放编译期生成的各种字面量与符号引用，这部分内容加载类加载后存放到方法区的运行时常量池中。
当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

2.2.7 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。这部分区域也可能会出现OOM异常。
JDK 1.4中加入的NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储再Java堆里面的DirectByteBuffer对象作为这块内存的引用操作。在某些场景中能显著提升性能，因为避免了在Java堆与Native堆中来回复制数据。动态拓展时可能会出现OOM异常。

2.3 HotSpot虚拟机对象探秘

2.3.2 对象的内存布局

在Hotspot虚拟机中，对象在堆内存中的存储布局可划分为对象头、实例数据和对齐填充三个部分。
Hotspot虚拟机的对象头部分包含两类信息。一是用于存储对象自身的运行时数据，如哈希码、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。一是类型指针，即对象指向它的类型元数据的指针。
实例数据部分是对象真正存储的有效信息，即各字段内容。
对齐填充部分数据并非必然存在的，也没有特别的含义。

2.3.3 对象的访问定位

《Java虚拟机规范》里面只规定了rederence类型是一个指向对象的引用，对象访问方式也是由虚拟机实现而定的，主流的访问方式主要有使用句柄和直接指针两种：

- 句柄访问时，Java堆中可能会划分出一块内存来作为句柄池，reference中存储的是对象的句柄地址，句柄中包含了对象实例数据与类型数据各自具体的地址信息。

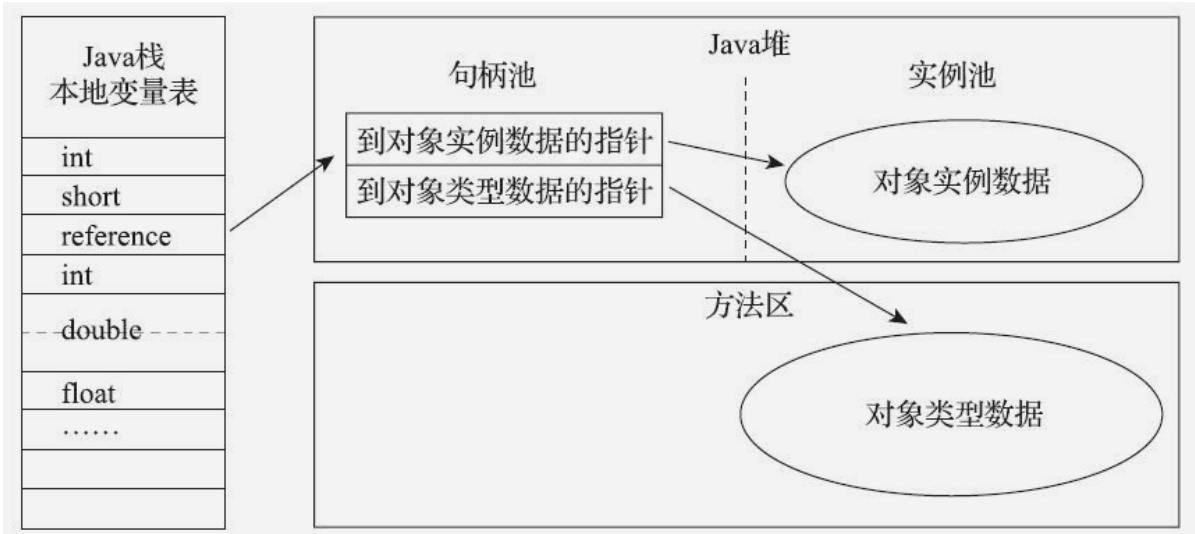


图2-2　通过句柄访问对象

- 直接指针访问时，Java堆中对象的内存布局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。

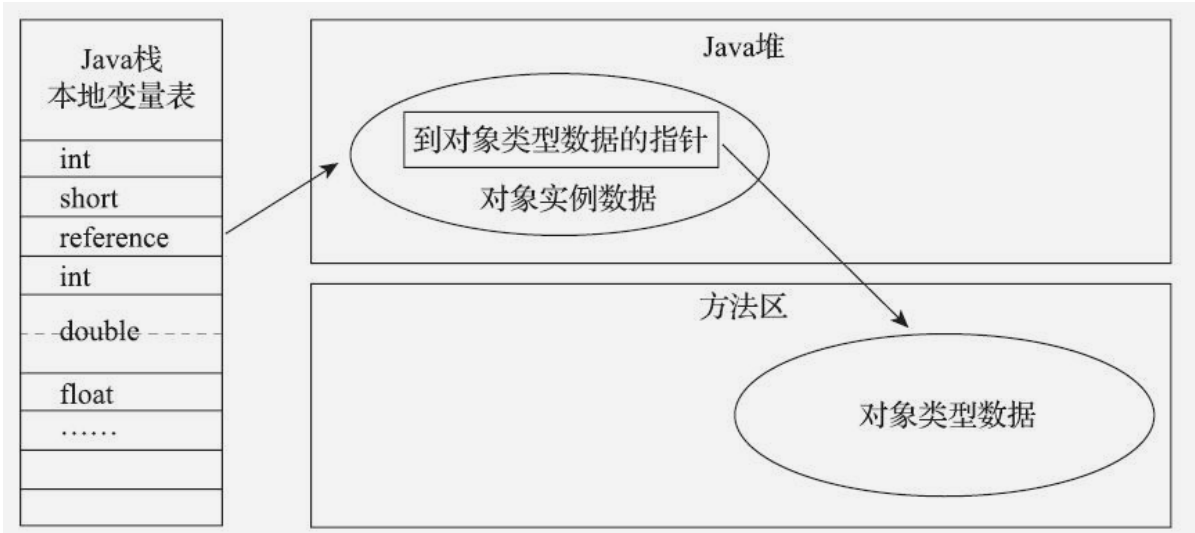


图2-3　通过直接指针访问对象

3 垃圾收集器与内存分配策略

3.1 概述

垃圾收集需要完成的三件事情：

- 哪些内存需要回收？
- 什么时候回收？
- 如何回收？

3.2 对象已死？

3.2.1 引用计数算法

在对象中添加一个引用计数器，每有一处引用它时，计数器值加一；每有引用失效时，计数器值减一。任何时刻计数器为零的对象就是不可能再被使用的。

引用计数算法原理简单，判定效率也高，多数情况下是一个不错的算法。但主流的java虚拟机里面都没有选用引用计数算法来管理内存，主要原因是，这个看似简单的算法有很多例外情况要考虑，必须配合大量额外的工作才能保证正确工作，如单纯的引用计数就很难解决对象之间相互循环引用的问题。

3.2.2 可达性分析算法

当前主流的商用程序语言的内存管理子系统，都是通过可达性分析（Reachablility Analysis）算法来判断对象是否存活的。基本思路是通过一系列称为“GC Roots”的跟对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程中所走过的路径被称为“引用链”，若某对象到GC Roots间没有任何引用链相连则证明此对象是不可能再被使用的。

在Java计数体系里，固定可作为GC Roots的对象包括以下几种：

- 在虚拟机栈（栈帧中的本地变量表）中引用的对象，如线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等。
- 在方法区中类静态属性引用的对象，如Java类的引用类型静态变量。
- 在方法区中常量引用的对象，如字符串常量池里的引用。
- 在本地方法栈中Native方法引用的对象。
- Java虚拟机内部的引用，如基本数据类型对应的Class对象，一些常驻的异常对象（如NullPointerException、OutOfMemoryError）等，还有系统类加载器。
- 所有被同步锁持有的对象。
- 反应Java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等。

3.2.3 再谈引用

在JDK 1.2版之前，Java里面的引用是很传统的定义：若reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称该reference数据是代表某块内存、某个对象的引用。

在JDK 1.2版之后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用4种，这4种引用强度依次逐渐减弱。

- 强引用是最传统的“引用”的定义，是指代码中普遍存在的引用赋值，即类似Object obj = new Object()这种引用关系。无论何种情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。
- 软引用是用来描述一些还有用，但非必须的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，若这次回收还没有足够的内存，才会抛出内存溢出异常。
- 弱引用也是用来描述哪些非必须对象，但其强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。
- 虚引用也称为”幽灵引用“或者“幻影引用”，它是最弱的一种引用关系。一个对象是否有虚引用存在，完全不会对

其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的只是为了能在这个对象被收集器回收时收到一个系统通知。