Java内存管理：深入Java内存区域 - zero516cn - 博

星期一, 七月 11, 2016　　11:50 上午

Java内存管理：深入内存区域

Java内存管理：深入Java内存区域

本文引用自：深入理解Java虚拟机的第2章内容

Java与C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

概述：

对于从事C和C++程序开发的开发人员来说，在内存管理领域，他们既是拥有最高权力的皇帝，又是从事最基础工作的劳动人民—既拥有每

一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于Java程序员来说，在虚拟机的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，而且不容易出现

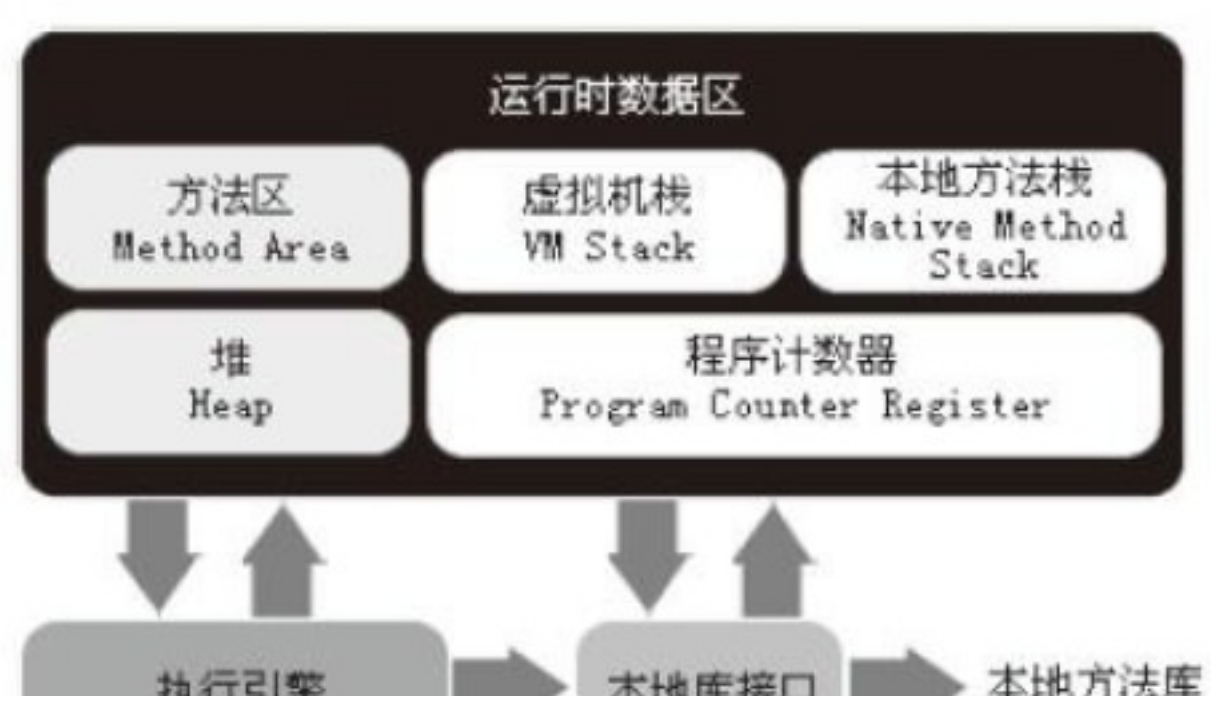
内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。不过，也正是因为Java程序员把内存控制的权力交给了Java虚拟机，一旦

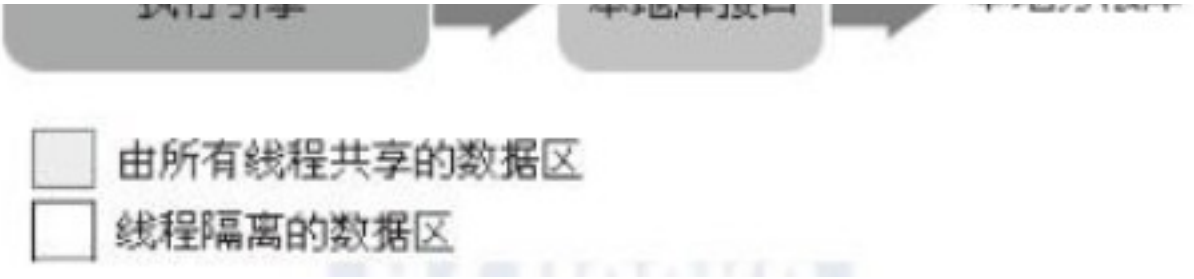
博客园

出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误将会成为一项异常艰难的工作。

运行时数据区域

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。根据《Java虚拟机规范（第2版）》的规定，Java虚拟机所管理的内存将会包括以下几个运行时数据区域，如下图所示：





程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟

机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），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，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并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）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条

线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。 如果线程正在执行的是一个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地址；如果正在执行的是Natvie方法，这个计数器值则为空（Undefined）。**此内存区域是唯一**

一个

在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）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

方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）**用于存储局部变量表、操作栈、动态链接、方法出口**

等信息。**每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。**

经常有人把Java内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java内存区域的划分实际上远比这复杂。这种划分

方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的“堆”在后面会专门讲述，而所指

的“栈”就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用

（reference类型），它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的

句柄或者其他与此对象相关的位置）和returnAddress类型（指向了一条字节码指令的地址）。

其中64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1个。**局部变量表所需的内存**

空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改

变局部变量表的大小。 在Java虚拟机规范中，对这个区域规定了两种异常状况：如果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；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的

虚拟机栈），当扩展时无法申请到足够的内存时会抛出OutOfMemoryError异常。

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，其区别不过是虚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到的Native方法服务。虚

们的*服务，而本地方法栈则是为虚拟机使用到的Native方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没

有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与

虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

Java堆

对于大多数应用来说，Java堆（Java Heap）是Java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，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，

但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝

对”了。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”（Garbage Collected Heap，幸好国内没翻译成“垃圾堆”）。如果从内存回收的角度看，

由于现在收集器基本都是采用的**分代收集算法**，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有Eden空间、From Survivor空间、To Survivor空间

等。如果**从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区**（Thread Local Allocation Buffer，TLAB）。不过，无论如何划

分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。在本章中，我们仅仅针对内存区

域的作用进行讨论，Java堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定

大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展

时，将会抛出OutOfMemoryError异常。

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信息、常量、静态变量、即时编**

区域，也称为“非堆”，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等

数据。虽然Java虚拟机规范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做Non-Heap（非堆），目的应该是与Java堆区分开来。

对于习惯在HotSpot虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久代”（Permanent Generation），本质上两者并不等价，仅仅是因

为HotSpot虚拟机的设计团队选择把GC分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如BEA JRockit、IBM J9等）来说是不存在

永久代的概念的。即使是HotSpot虚拟机本身，根据官方发布的路线图信息，现在也有放弃永久代并“搬家”至Native Memory来实现方法区的规划了。

Java虚拟机规范对这个区域的限制非常宽松，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，还可以选择不实现垃圾收集。相对而言，

垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个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，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在Sun公司的BUG列表中，

曾出现过的若干个严重的BUG就是由于低版本的HotSpot虚拟机对此区域未完全回收而导致内存泄漏。 根据Java虚拟机规范的规定，当方法区无法满足内存分配

需求时，将抛出OutOfMemoryError异常。

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、接口等描述等信息外，还有一项信息是常量池

（Constant Pool Table），**用于存放编译期生成的各种字面量和符号引用，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。**Java虚拟机对Class

文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于

运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class

文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。 运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言

并不要求常量一定只能在编译期产生，也就是并非预置入Class文件中

常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被

开发人员利用得比较多的便是String类的intern()方法。 既然运行时常量池是方法区的一部分，自然会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。

对象访问

介绍完Java虚拟机的运行时数据区之后，我们就可以来探讨一个问题：在Java语言中，对象访问是如何进行的？对象访问在Java语言中无处不在，是最普通的程

序行为，但即使是最简单的访问，也会却涉及Java栈、Java堆、方法区这三个最重要内存区域之间的关联关系，如下面的这句代码：

```
Object obj = new Object();
```

假设这句代码出现在方法体中，那“Object obj”这部分的语义将会反映到Java栈的本地变量表中，作为一个reference类型数据出现。而“new Object()”这部分的语

义将会反映到Java堆中，形成一块存储了Object类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现

的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在Java堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实

现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于reference类型在Java虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到Java堆中的对

象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄和直接指针。 如果使用句柄访问方式，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和类型数据各自的具体地址信息，如下图所示：

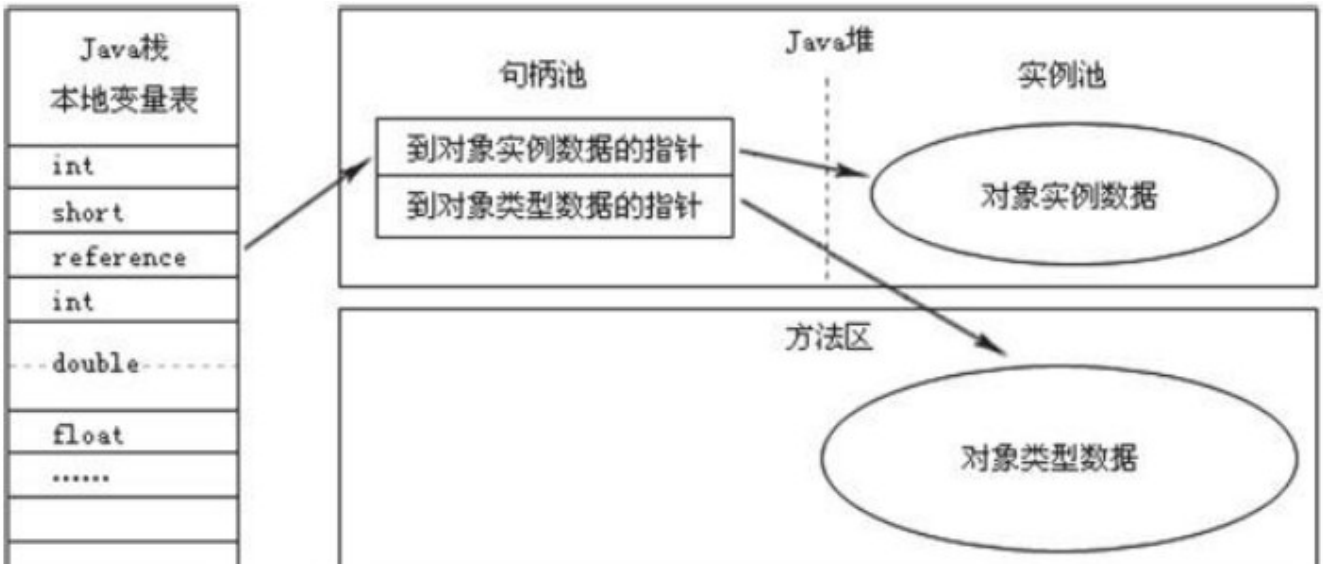


图 2-2　通过句柄访问对象

如果使用的是直接指针访问方式，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中直接存储的就是对象地址，如下

图所示：

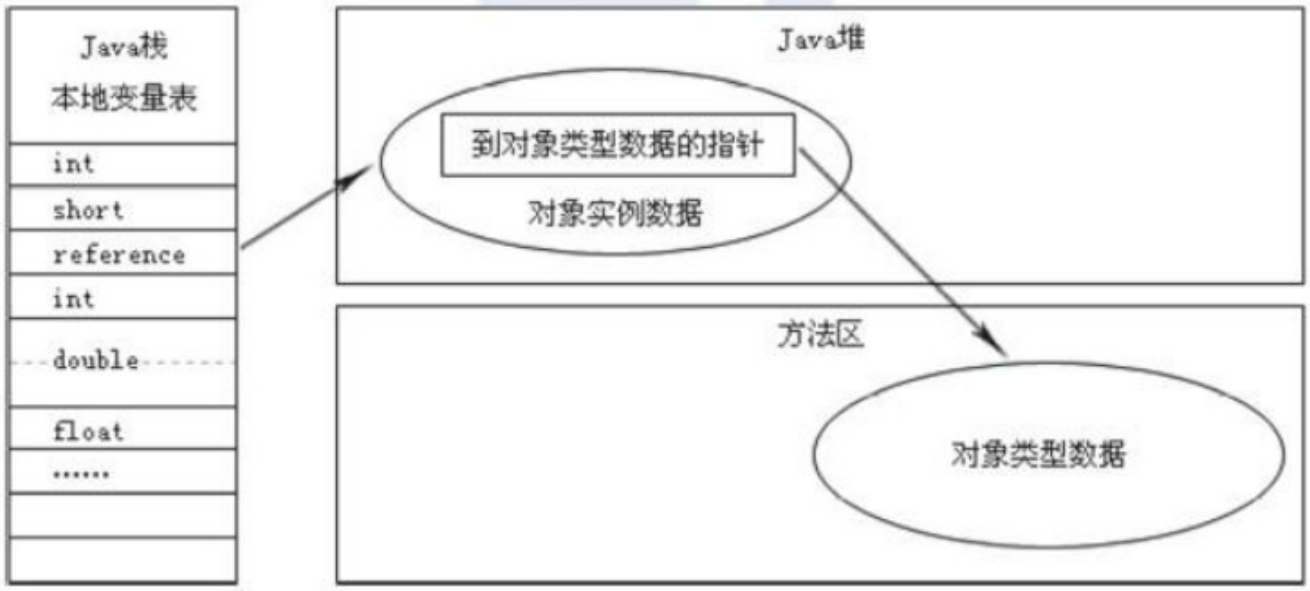


图 2-3　通过直接指针访问对象

这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是reference中存储的是稳定的句柄地址，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。使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针

定位的时间开销，由于对象的访问在Java中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。就本书讨论的主要虚拟机Sun

HotSpot而

言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

已剪辑自: http://www.cnblogs.com/gw811/archive/2012/10/18/2730117.html