JVM 性能调优-JVM 内存区域划分

1.程序计数器（线程私有）

程序计数器（Program Counter Register），也有称作为 PC 寄存器。保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当 CPU 需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加 1 或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。也就是说是用来**指示执行哪条指令的**。

由于在 JVM 中，多线程是通过**线程轮流切换来获得 CPU 执行时间的**，因此，在任一具体时刻，一个 CPU 的内核只会执行一条线程中的指令，因此，为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。因此，可以这么说，程序计数器是**每个线程所私有的。**

在 JVM 规范中规定，如果线程执行的是非 native 方法，则程序计数器中保存的是当前需要执行的指令的地址；如果线程执行的是 **native 方法，**则程序计数器中的值是 **undefined。**

由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的。

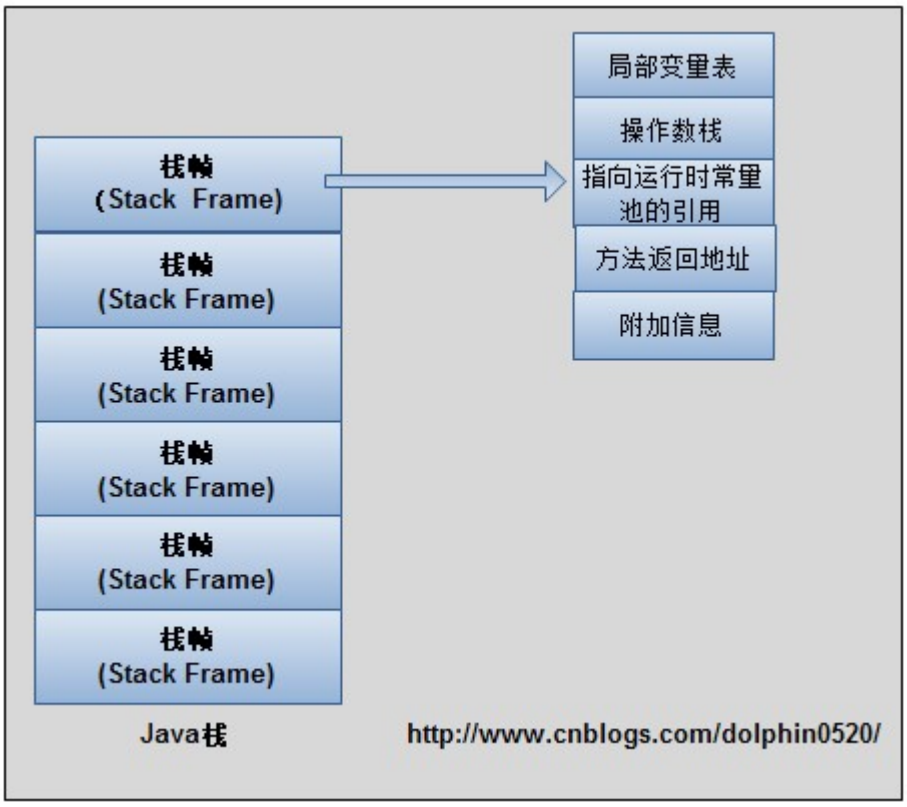
异常情况：

不存在

2.Java 栈（线程私有）

Java 栈也称作虚拟机栈（Java Vitual Machine Stack）

Java 栈中存放的是一个个的栈帧，**每个栈帧对应一个被调用的方法**，在栈帧中包括**局部变量表、操作数栈、指向当前方法所属的类的运行时常量池的引用、方法返回地址、额外的附加信息**。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程**当前执行的方法**所对应的栈帧必**定位于 Java 栈的顶部**。



局部变量表，用来**存储**方法中的局部变量（包括在方法中**声明的非静态变量以及函数形参**）。对于**基本数据类型**的变量，则**直接存储**它的值，对于**引用类型**的变量，则**存的**是指向**对象**的**引用**。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

存储内容：引用对象，returnAddress 类型。Long 和 double 类型占用 2 个局部变量空间，其余的数据类型占据一个。局部变量表空间在编译期间完成分配。

操作数栈，栈最典型的一个应用就是用**来对表达式求值**。想想一个线程执行方法的过程中，实际上就是不断执行语句的过程，而归根到底就是进行计算的过程。因此可以这么说，程序中的**所有计算过程**都是在借助于**操作数栈来完成的**。

指向运行时常量池的引用，因为在方法执行的过程中有可能需要用到类中的常量，所以必须要有一个引用指向运行时常量。

方法返回地址，当一个方法执行完毕之后，要返回之前调用它的地方（**参考汇编**），因此在栈帧中必须保存一个方法返回地址。

由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的 Java 栈**线程私有，互不干扰。**

异常情况：

1.栈深度大于已有深度：StackOverflowError

2.可扩展深度大于能够申请的内存：OutOfMemoryError

3.本地方法栈（线程私有）

本地方法栈与 Java 栈的作用和原理非常相似。区别只不过是 **Java 栈**是为**执行 Java 方法服务**的，而**本地方法栈**则是为**执行本地方法（Native Method）服务**的。在 JVM 规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在 **HotSopt 虚拟机中直接就把本地方法栈和 Java 栈合二为一。**

异常情况：

1.栈深度大于已有深度：StackOverflowError

2.可扩展深度大于能够申请的内存：OutOfMemoryError

4.堆（线程共享）

Java 中的堆是用来**存储对象本身**的以**及数组**（当然，数组引用是存放在 Java 栈中的），堆是被所有**线程共享**的，在 JVM 中只有一个堆。所有对象实例以及数组都要在堆上分配内存，单随着 JIT 发展，**栈上分配**，**标量替换优化技**术，在堆上分配变得不那么到绝对，只能在 **server 模式**下才能**启用逃逸分析**。

栈上分配：
一是逃逸分析：逃逸分析的目的是判断对象的作用域是否有可能逃逸出函数体。
二是标量替换：允许将对象打散分配在栈上，比如若一个对象拥有两个字段，会将这两个字段视作局部变量进行分配。

垃圾收集器管理的主要区域，很多时候被称作 **GC 堆**。现在收集器基本采用分代收集算法：新生代和老年代，再细致点 Eden 空间，From Survivor 空间，ToSurvivor 空间等。

异常情况：

1.可以处于物理上不连续的内存空间，逻辑连续即可。既可实现固定大小，也可扩展。如果堆中没有内存完成实例分配，并且堆无法再扩展是，将会抛出 **OutOfMemoryError；**

5.方法区（线程共享）

5.1.方法区中，**存储**了每个**类**的**信息**（包括类的名称、方法信息、字段信息）、**静态变量**、**常量**以及**编译器编译后的代码**等。

在 Class 文件中除了类的字段、方法、接口等描述信息外，还有一项信息是**常量池**，用来**存储**编译期间生成的**字面量**和符号**引用**。

5.2 方法区还有一块内存，**运行时常量池**，它是每一个类或接口的常量池的运行时表示形式，在**类**和接口被**加载**到 JVM 后，对应的运行时**常量池**就**被创建**出来。当然**并非**

Class 文件常量池中的内容**才能进入运行时常量池**，在**运行期间也可将新的常量放入运行时常量池**中，比如 **String** 的 **intern** 方法。

在 JVM 规范中，没有强制要求方法区必须实现垃圾回收。很多人习惯将**方法区称为"永久代"**，是因为 HotSpot 虚拟机以永久代来实现方法区，从而 JVM 的**垃圾收集器**可以像管理堆区一样**管理这部分区域**，从而不需要专门为这部分设计垃圾回收机制。不过自从 **JDK7 之后**，Hotspot 虚拟机便将**运行时常量池从永久代移除了**。

异常情况：

1.方法区调用递归，内存会溢出，报 **OutOfMemoryError**；

2.当常量池无法再申请到内存时 **OutOfMemoryError**；

6.直接内存（线程共享）

NIO,使用 native 函数库直接分配堆外内存，不经过 JVM 内存直接访问系统物理内存的类——DirectBuffer。 DirectBuffer 类继承自 ByteBuffer，但和普通的 ByteBuffer 不同，普通的 ByteBuffer 仍在 JVM 堆上分配内存，其最大内存受到最大堆内存的限制；而 DirectBuffer 直接分配在物理内存中，并不占用堆空间，其可申请的最大内存受操作系统限制。

堆内存比较：

1. 直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显
2. 直接内存 IO 读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显

异常情况：

1.DirectBuffer 分配内存溢出

1.对象访问

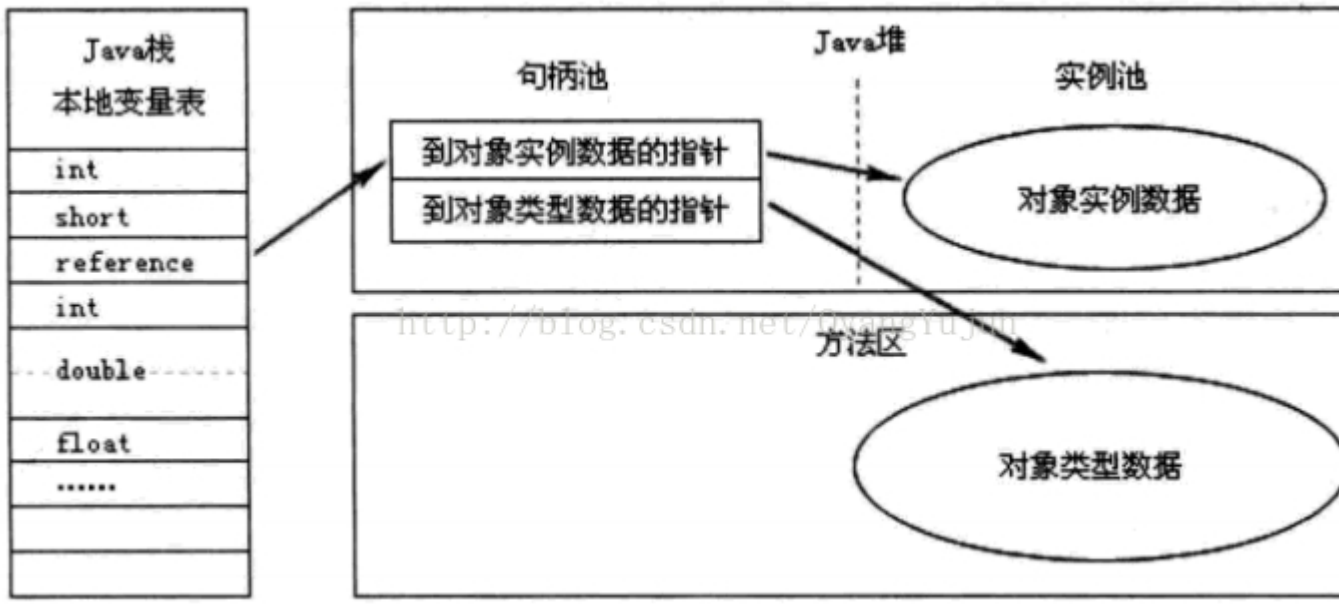
对象访问在 Java 语言中无处不在，即使是最简单的访问，也会涉及到 Java 栈，java 堆，方法区这三个最重要的内存区域之间的关联关系。如下面的代码：

Object obj = new Object();

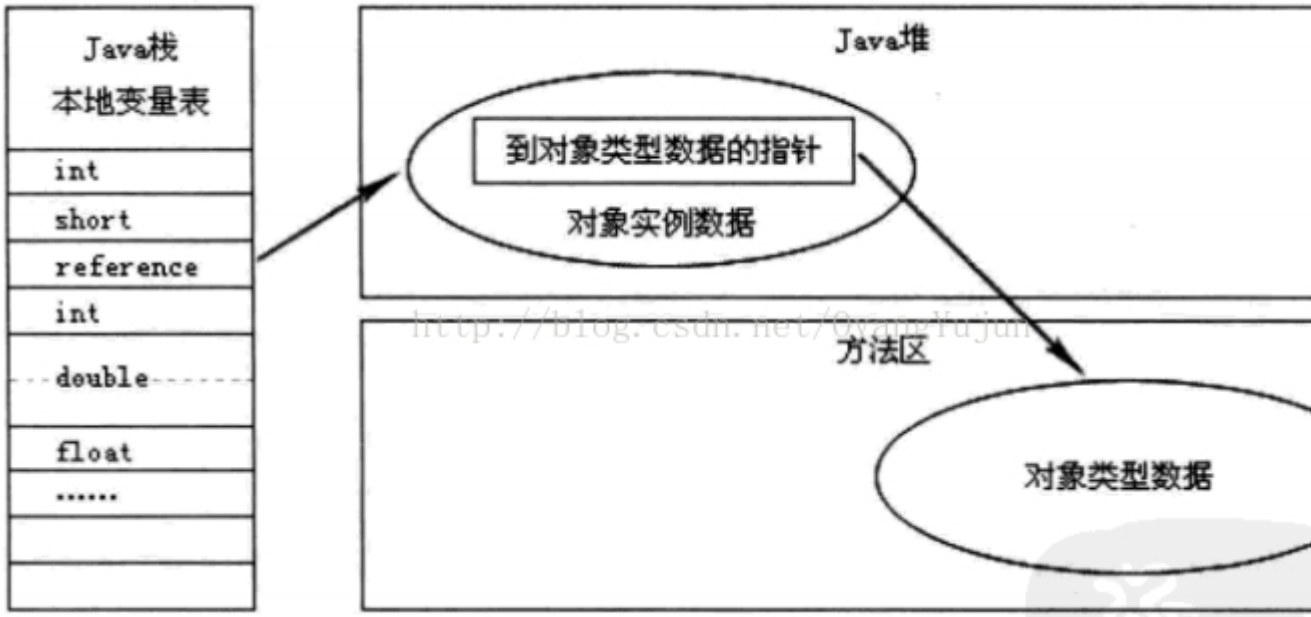
假设这段代码出现在方法体中，那么“Object　obj”部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型的数据存在。而“new　Object();”部分的语义将会反应到 Java 堆中，形成一块存储 Object 类型所有实例数据值（Instance　Data）的结构化内存，根据具体类型以及虚拟机实现的对象分布的不同，这块内存的长度是不固定的。另外，在 JAVA 堆中还必须包含能查找到此对象内存数据的地址信息，这些类型数据则存储在方法区中。

由于 reference 类型在 Java 虚拟机中之规定了指向对象的引用，并没有规定这个引用要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：**句柄访问方式**和**直接指针**。

1.　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。



2.　如果通过直接指针方式访问，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的地址。



两种方式各有优势，局并访问方式最大的好处是 reference 中存放的是稳定的句柄地址，在对象被移动时，只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。而指针访问的最大优势是速度快，它节省了一次指针定位的开销，由于对象访问在 Java 中非常频繁，一次这类开销积少成多后也是一项非常可观的成本。
具体的访问方式都是有虚拟机指定的，虚拟机 Sun HotSpot 使用的是直接指针方式，不过从整个软件开发的范围来看，各种语言和框架使用句柄访问方式的情况十分常见。

2.TLAB

TLAB 的全称是 Thread Local Allocation Buffer，即线程本地分配缓存区，这是一个线程专用的内存分配区域。
由于对象一般会分配在堆上，而堆是全局共享的。因此在同一时间，可能会有多个线程在堆上申请空间。因此，每次对象分配都必须要进行同步（虚拟机采用 **CAS** 配上失败重试的方式保证更新操作的原子性），而在竞争激烈的场合分配的效率又会进一步下降。JVM 使用 TLAB 来**避免多线程冲突**，在给对象分配内存时，**每个线程使用自己的 TLAB**，这样可以**避免线程同步**，提高了对象分配的**效率**。

详情

深入虚拟机 XMIND 文件

JAVA内存区域

方法区（Method Area）：
用于存储已经被虚拟机加载的类信息（即加载类时需要加载的信息，包括版本、field、方法、接口等信息）、final常量、静态变量、编译器即时编译的代码等。

运行时常量池（Runtime Constant Pool）：
用于存储编译期就生成的字面常量、符号引用、翻译出来的直接引用

线程共享

在内存不足时抛出OutOfMemoryError:PermGen space异常

OutOfMemoryError: PermGen space

堆区（Heap）：
堆区的存在是为了存储对象实例，原则上讲，所有的对象都在堆区上分配内存（不过现代技术里，也不是这么绝对的，也有栈上直接分配的）。

堆区是理解Java GC机制最重要的区域，没有之一。在JVM所管理的内存中，堆区是最大的一块，堆区也是Java GC机制所管理的主要内存区域

线程共享

如果在执行垃圾回收之后，仍没有足够的内存分配，也不能再扩展，将会抛出OutOfMemoryError:Java heap space异常。

OutOfMemoryError: Java heap space

本地方法栈（Native Method Statck）：
本地方法栈在作用运行机制，异常类型等方面都与虚拟机栈相同
唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行native方法的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。

线程私有

OutOfMemoryError

StatckOverFlowError

jvm虚拟机
Stack
一个线程的每个
同时，都会创
（Statck Frame
储的有
局部变量表、操
接、方法

，当方法被调用
JVM栈中入栈，
成时，栈

线程私

虚拟机栈中
常，如果线程调
于虚拟机允许的
抛出
StatckOverFlo
出）；不过多数
允许动态扩展虚
(有少部分是固定
线程可以一直申
存不足，此时
OutOfMemory
出）

StatckOverFlowError