JVM

一、内存模型

1. 简介

Java 虚拟机有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。

Java 虚拟机本质是就是一个程序，当它在命令行上启动的时候，就开始执行保存在某字节码文件中的指令。Java 语言的可移植性正是建立在 Java 虚拟机的基础上。任何平台只要装 有针对于该平台的 Java 虚拟机，字节码文件（.class）就可以在该平台上运行。这就是“一次编译，多次运行”。

Java 虚拟机不仅是一种跨平台的语言，而且是一种新的网络计算平台。该平台包括许多相关的技术，如符合开放接口标准的各种 API、优化技术等。Java 技术使同一种应用可以运行

在不同的平台上。Java 平台可分为两部分，即 Java 虚拟机（Java virtual machine，JVM）和 Java API 类库。

每个 Java 程序都离不开 Java 虚拟机，Java 程序的运行依靠具体的 Java 虚拟机实例。在 Java 虚拟机规范中，分别用子系统、内存区、数据类型以及指令这几个术语来描述的。这些组成部分一起展示出一个抽象化的虚拟机内部的抽象体系结构。

Java 虚拟机主要分为五大模块：类装载器子系统、运行时数据区、执行引擎、本地方法接

口和垃圾收集模块。其中垃圾收集模块在 Java 虚拟机规范中并没有要求 Java 虚拟机垃圾收

集，但是在没有发明无限的内存之前，大多数 JVM 实现都是有垃圾收集的。而运行时数据

区都会以某种形式存在于每一个 JAVA 虚拟机实例中，但是 Java 虚拟机规范对它的描述却是

相当抽象。这些运行时数据结构上的细节，大多数都由具体实现的设计者决定。

Java 虚拟机不是真实的物理机，它没有寄存器，所以指令集是使用 Java 栈来存储中间数据，

这样做的目的就是为了保持 Java 虚拟机的指令集尽量的紧凑，同时也便于 JAVA 虚拟机在那

些只有很少通用寄存器的平台上实现。另外，JAVA 虚拟机的这种基于栈的体系结构，有助

于运行时某些虚拟机实现的动态编译器和即时编译器的代码优化。

Java 虚拟机内存模型中定义的访问操作与物理计算机处理的基本一致

2. 原理

2.1 什么是JVM

（1）jvm 是一种用于计算设备的规范，它是一个虚构出来的机器，是通过在实际的计 算机上仿真模拟各种功能实现的。

（2）jvm 包含一套字节码指令集，一组寄存器，一个栈，一个垃圾回收堆和一个存储方法域。

（3）JVM 屏蔽了与具体操作系统平台相关的信息，使 Java 程序只需生成在 Java 虚拟

机上运行的目标代码（字节码）,就可以在多种平台上不加修改地运行。JVM 在执行字

节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

2.2 jdk、jre、jvm 是什么关系？

（1）JRE(Java Runtime Environment)，也就是 java 平台。所有的 java 程序都要在 JRE环境下才能运行。

（2）JDK(Java Development Kit)，是开发者用来编译、调试程序用的开发包。JDK 也是JAVA 程序需要在 JRE 上运行。

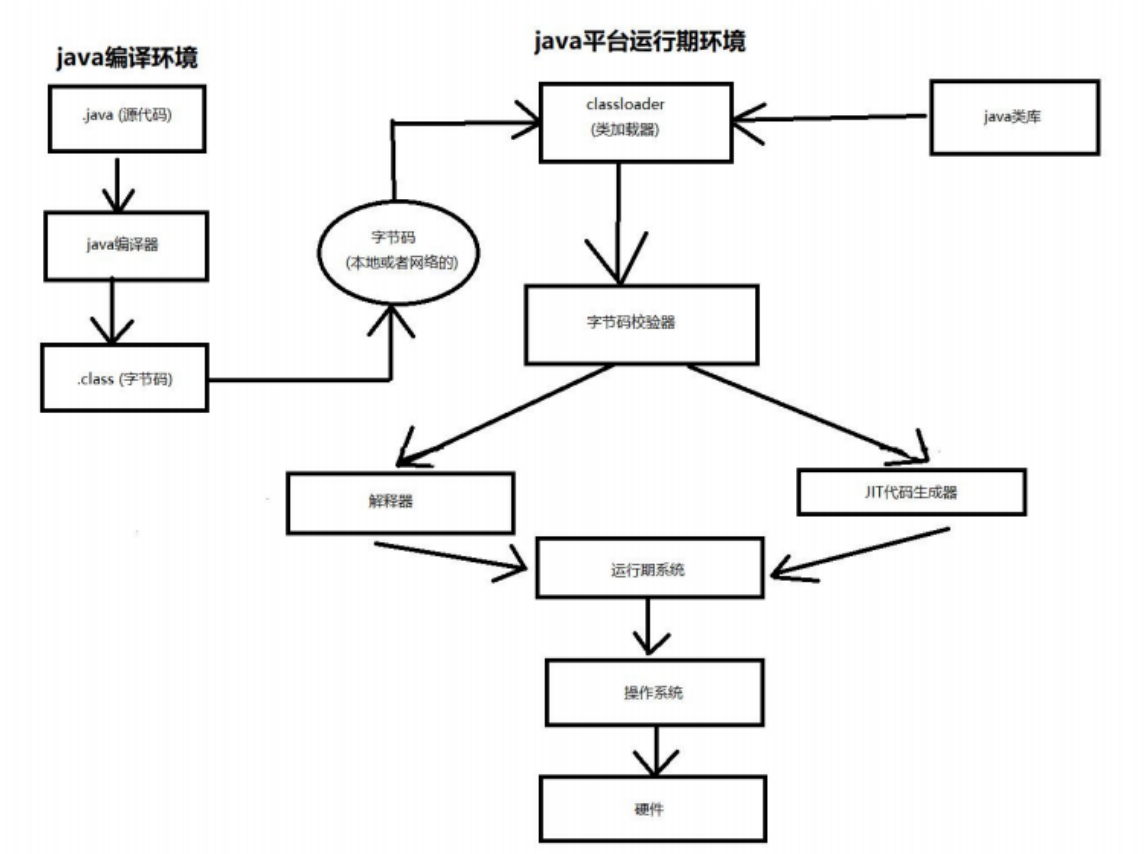
（3）JVM(Java Virtual Machine)，是 JRE 的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

JVM 有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。使用 JVM 就是为了支持与操作系统无关，实现跨平台。

2.3 JVM原理

（1）jvm 是 java 的核心和基础，在 java 编译器和 os 平台之间的虚拟处理器，可在上面执行字节码程序。

（2）java 编译器只要面向 jvm，生成 jvm 能理解的字节码文件。java 源文件经编译成字节码程序，通过 jvm 将每条指令翻译成不同的机器码，通过特定平台运行。



2.4 JVM执行程序的过程

1) 加载

.class 文件

2) 管理并分配内存

3) 执行垃圾收集

JRE（java 运行时环境）由 JVM 构造的 java 程序的运行环，也是 Java 程序运行的环境，

是他同时一个操作系统的一个应用程序一个进程，因此他也有他自己的运行的生命周期，也有自己的代码和数据空间。JVM 在整个 jdk 中处于最底层，负责于操作系统的交互，用来屏蔽操作系统环境，提供一个完整的 Java 运行环境，因此也就虚拟计算机。操作系统装入 JVM 是通过 jdk 中 Java.exe 来完成，通过下面 4 步来完成 JVM 环境：

1) 创建 JVM 装载环境和配置

2) 装载 JVM.dll

3) 初始化 JVM.dll 并挂界到 JNIENV(JNI 调用接口)实例

4) 调用 JNIEnv 实例装载并处理 class 类。

2.5 JVM生命周期

1) JVM 实例对应了一个独立运行的 java 程序它是进程级别

a) 启动。启动一个 Java 程序时，一个JVM 实例就产生了，任何一个拥有 public static void main(String[] args)函数的 class 都可以作为 JVM 实例运行的起点

b) 运行。main()作为该程序初始线程的起点，任何其他线程均由该线程启动。JVM 内部

有两种线程：守护线程和非守护线程，main()属于非守护线程，守护线程通常由 JVM 自己

使用，java 程序也可以表明自己创建的线程是守护线程

c) 消亡。当程序中的所有非守护线程都终止时，JVM 才退出；若安全管理器允许，程序

也可以使用 Runtime 类或者 System.exit()来退出

2) JVM 执行引擎实例则对应了属于用户运行程序的线程它是线程级别的

2.6 各个内存空间的介绍

2.6.1 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。

2.6.2 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 ①）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（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。

2.6.3 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。

2.6.4 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③，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②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（ThreadLocal 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。

2.6.5 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。

2.6.6 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。

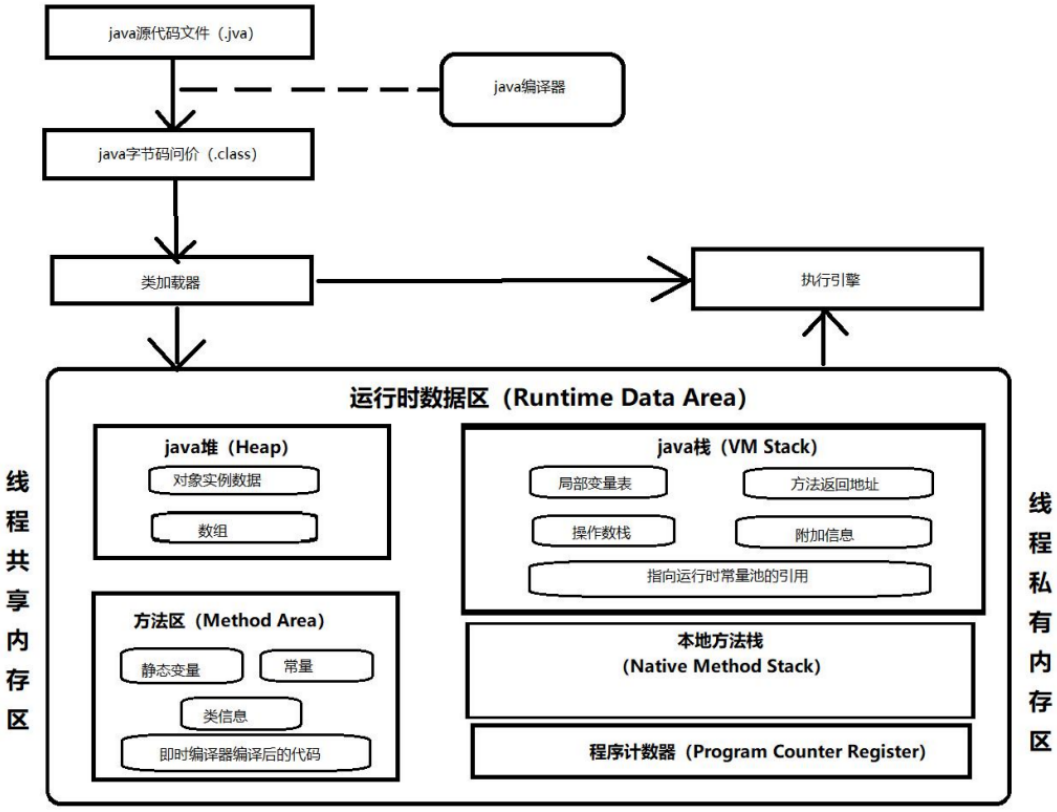
既然运行时常量池是方法区的一部分，自然会受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常

2.6.7 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。

在 JDK 1.4 中新加入了 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，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

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 RAM 及 SWAP 区或者分页文件）的大小及处理器寻址 空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置-Xmx 等参数 信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现 OutOfMemoryError 异常。



（永久代是 hotspot 虚拟机说法，其他虚拟机并不存在，用方法区代替）

JVM 内存结构主要有三大块：堆内存、方法区和栈。堆内存是 JVM 中最大的一块由年轻代和老年代组成，而年轻代内存又被分成三部分，Eden 空间、From Survivor 空间、ToSurvivor 空间,默认情况下年轻代按照 8:1:1 的比例来分配；

方法区存储类信息、常量、静态变量等数据，是线程共享的区域，为与 Java 堆区分，方法区还有一个别名 Non-Heap(非堆)；栈又分为 java 虚拟机栈和本地方法栈主要用于方法的执行。

控制参数

-Xms 设置堆的最小空间大小。

-Xmx 设置堆的最大空间大小。

-XX:NewSize 设置新生代最小空间大小。

-XX:MaxNewSize 设置新生代最大空间大小。

-XX:PermSize 设置永久代最小空间大小。

-XX:MaxPermSize 设置永久代最大空间大小。

-Xss 设置每个线程的堆栈大小。

3. 对象的访问

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：

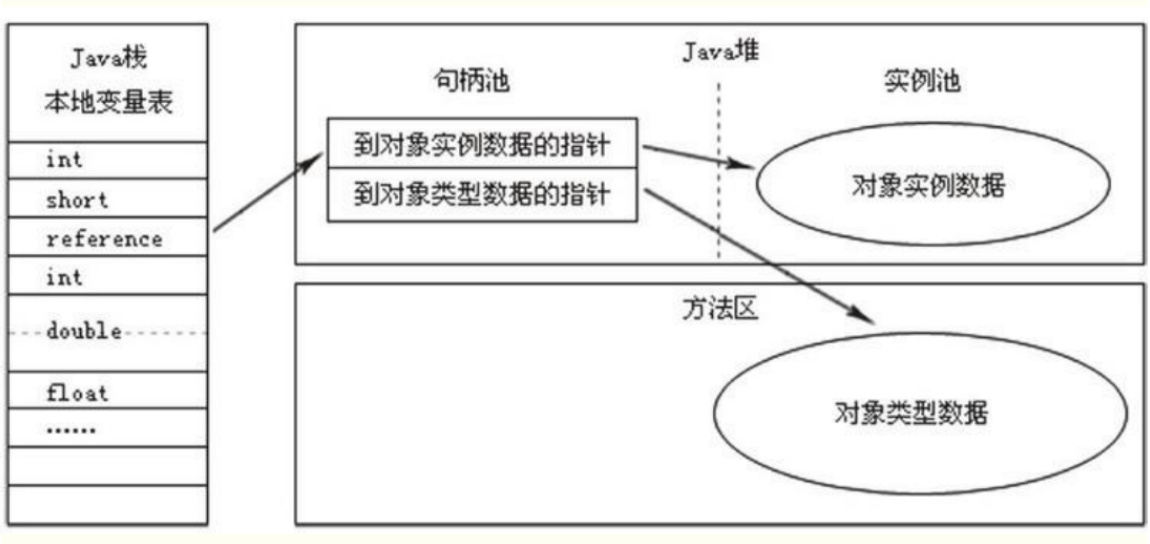
```
Object obj = new Object();
```

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

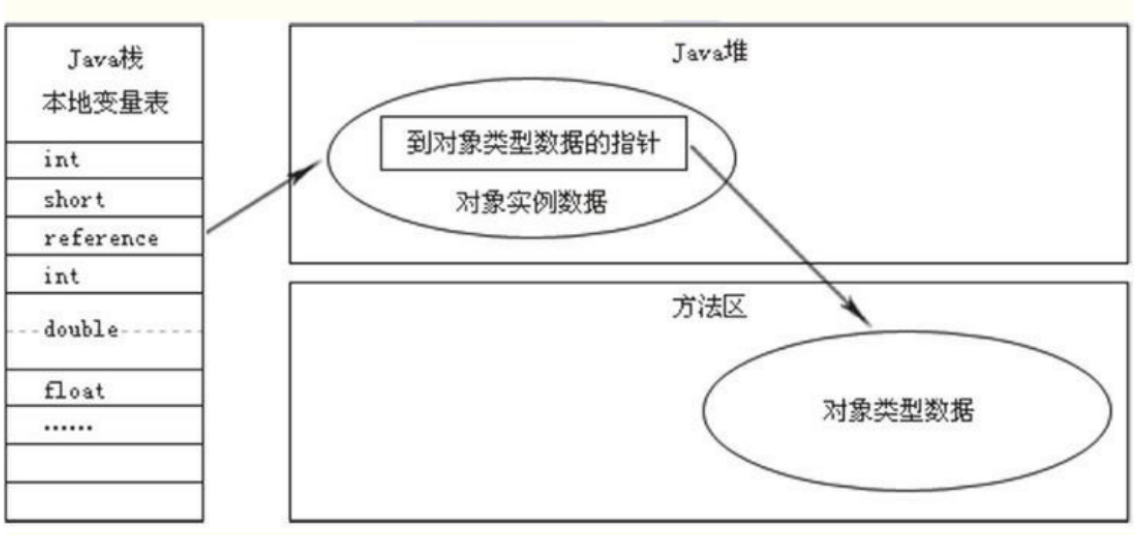
由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现 的对象访问方式会有所不同，主流的访问方式有两种：

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：

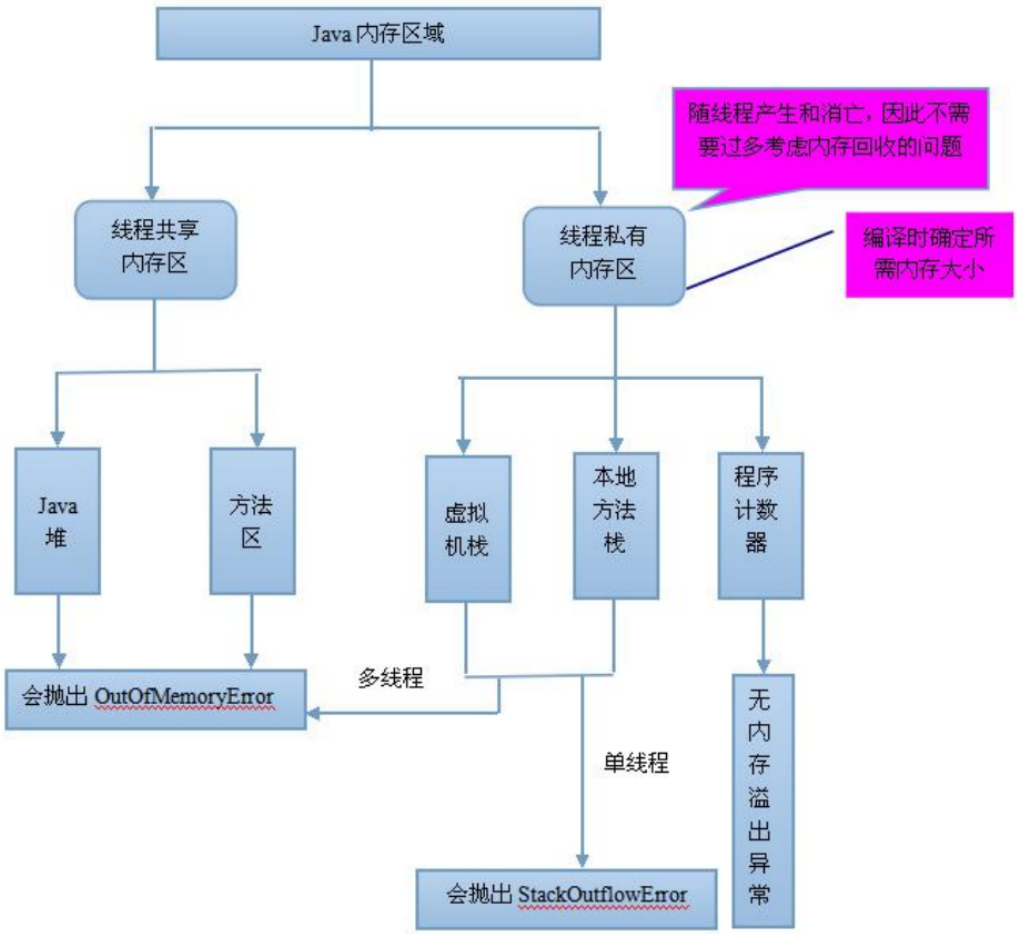
①句柄访问



如果使用直接指针访问方式，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



4．内存溢出



二、GC

1.如何确定垃圾

1.1 引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收对象。

1.2 可达性分析

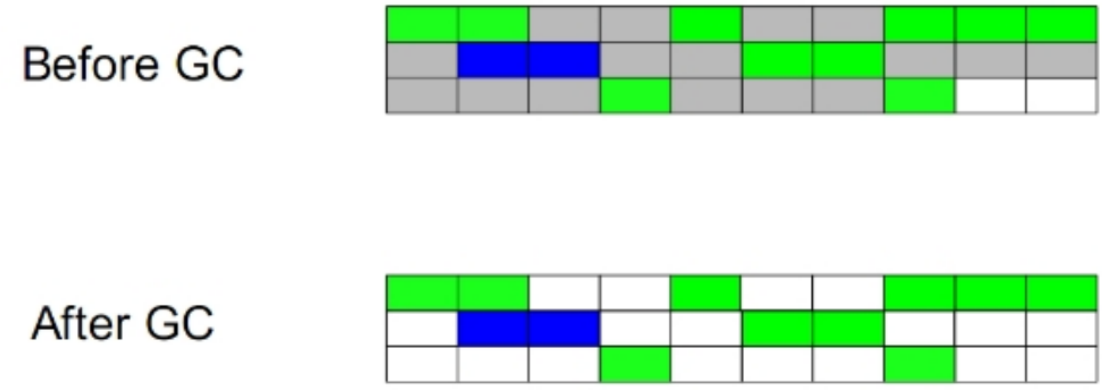
为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots”对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。

要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

2．垃圾回收的基本算法

2.1 标记清除算法（Mark-Sweep）

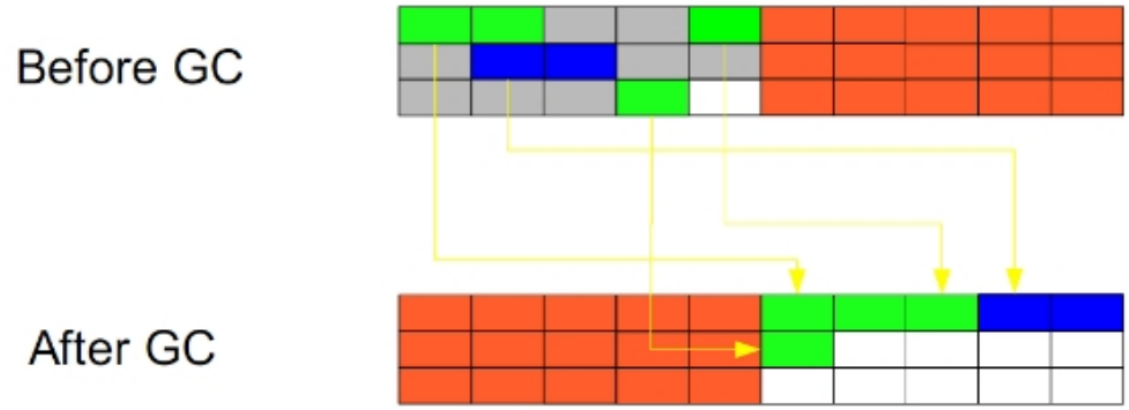
最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图



从图中我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

2.2 复制（Copying）

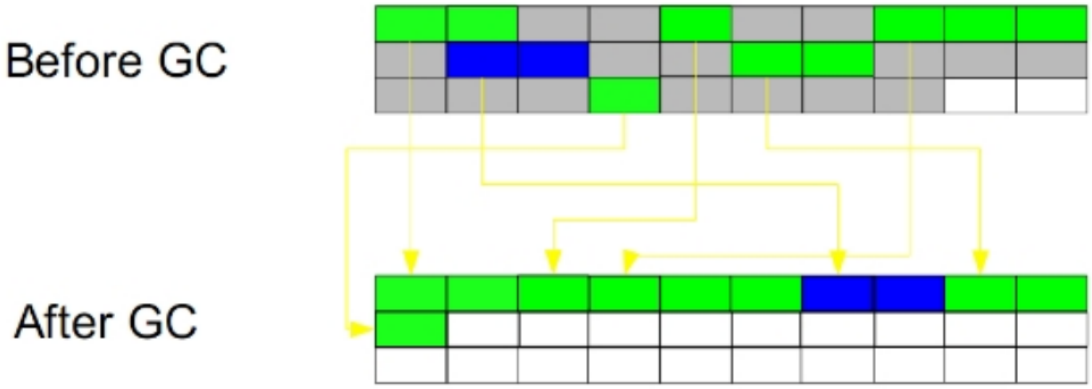
为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如图：



这种算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话，Copying 算法的效率会大大降低。

2.3 标记整理算法（Mark-Compact）

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同，标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：



此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，清除标记对象，并未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

3.按分区对待方式分类

3.1 增量收集

实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因 JDK5.0 中的收集器没有使用这种算法的。

3.2 分带收集

基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

4. 按系统线程分类

4.1 串行收集

串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。

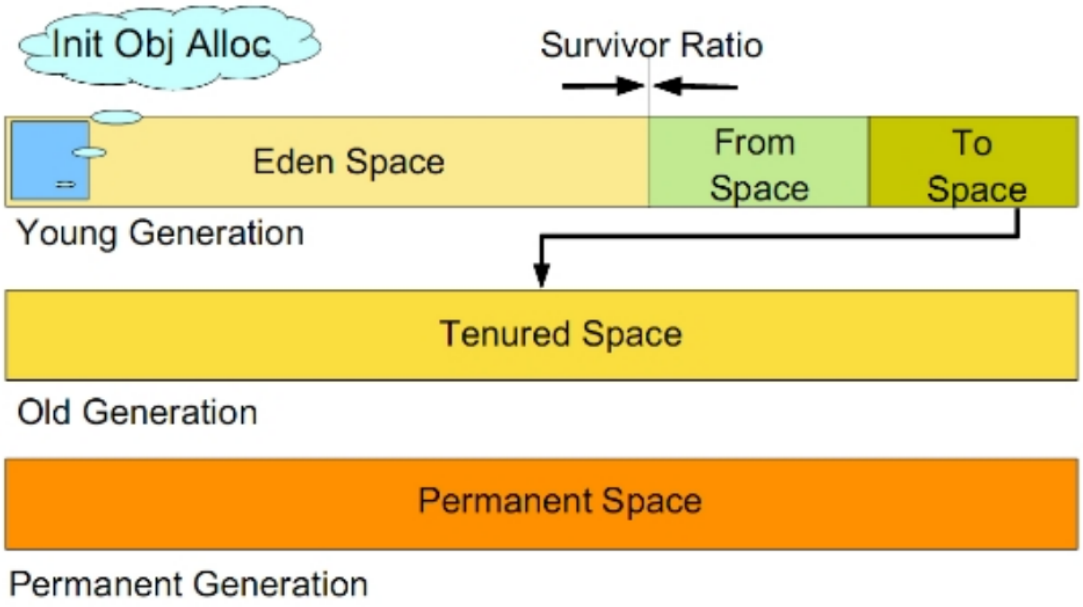
4.2 并行收集

并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上 CPU 数目越多，越能体现出并行收集器的优势。

4.3 并发收集

相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

5. 分代垃圾收集



试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

虚拟机中的共划分为三个代：年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

年轻代**:**所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来 对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代**:在年轻代中经历了 N** 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

持久代**:**用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置

一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=进行设置。