前言

本来计划要写Android内存优化的，觉得有必要在此之前介绍一下Java虚拟机的相关知识，Java虚拟机也并不是三言两语能够介绍完的，因此开了Java虚拟机系列，这一篇文章我们来学习Java虚拟机的结构原理与运行时数据区域。

**1.Java**虚拟机概述

Oracle官方定义的Java技术体系主要包括以下几个部分：

• Java程序设计语言

• 各种平台的Java虚拟机

• Class文件格式

• Java API类库

• 第三方Java类库

可以把Java程序设计语言、Java虚拟机和Java API类库这三部分统称为JDK（Java Development Kit），它是Java程序开发的最小环境。另外，Java API中的Java SE API子集和Java虚拟机这两部分统称为JRE（Java Runtime Environment），它是Java程序运行的标准环境。  
从上面可以看出Java虚拟机及其重要，它是整个Java平台的基石，是Java语言编译代码的运行平台。你可以把Java虚拟机看做一个抽象的计算机，它有各种指令集和各种运行时数据区域。

**1.1 Java**虚拟机家族

很多同学可能认为Java虚拟机，就是一个虚拟机而已，它还有家族？或者认为Java虚拟机指的就是Oracle的HotSpot虚拟机。这里来简单介绍Java虚拟机家族，自从1996年Sun公司发布的JDK1.0中包含的Sun Classic VM到今天，出现和消亡了很多种虚拟机，我们这里只简单介绍目前存活的相对主流Java虚拟机。

**HotSpot VM**  
Oracle JDK和OpenJDK中自带的虚拟机，是最主流的和使用范围最广的Java虚拟机。介绍Java虚拟机的技术文章，如果不做特殊说明，大部分都是介绍HotSpot VM的。HotSpot VM并非是Sun公司开发的，而是由Longview Technologies这家小公司设计的，它在1997年被Sun公司收购，Sun公司又在2009年被Oracle收购。  
**J9 VM**  
J9 VM 是IBM开发的VM，目前是其主力发展的Java虚拟机。J9 VM的市场定位和HotSpot VM接近，它是一款设计上从服务端到桌面应用再到嵌入式都考虑到的多用途虚拟机，目前J9 VM的性能水平大致跟HotSpot VM是一个档次的。  
**Zing VM**  
以Oracle的HotSpot VM为基础，改进了许多会影响延迟的细节。最大的三个卖点是：

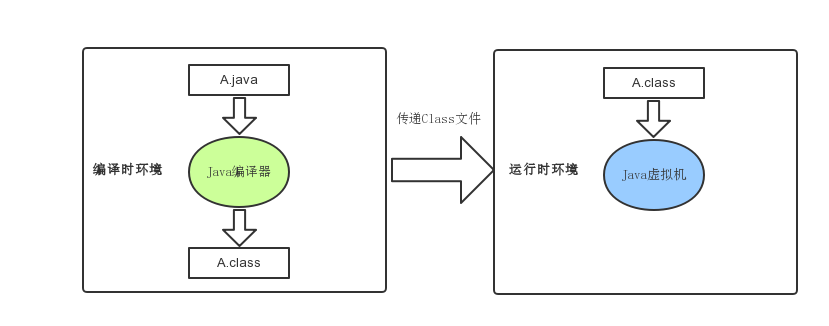
• 1.低延迟，“无暂停”的C4 GC，GC带来的暂停可以控制在10ms以下的级别，支持的Java堆大小可以到1TB；

• 2.启动后快速预热功能。

• 3.可管理性：零开销、可在生产环境全时开启的、整合在JVM内的监控工具Zing Vision。

**1.2 Java**虚拟机执行流程

当我们执行一个Java程序时，它的执行流程是怎样的呢？如下图所示。

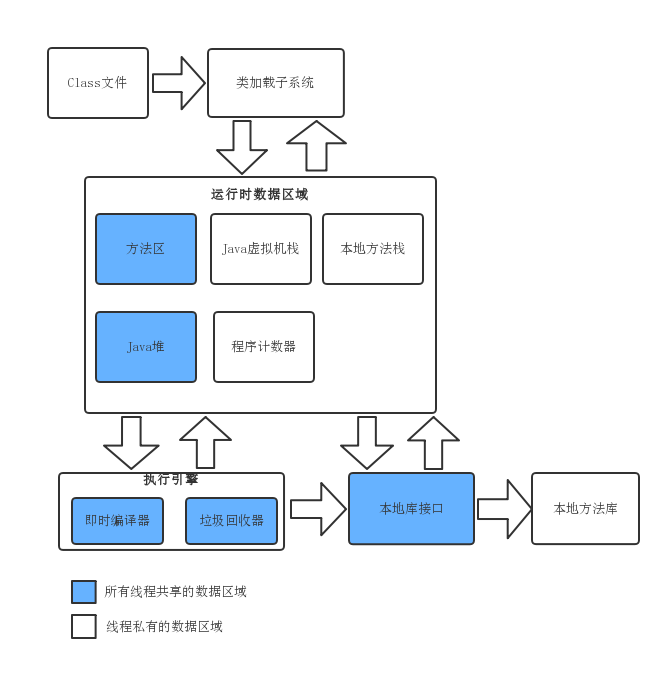


Java程序执行流程(2).png

从上图可以看到Java虚拟机与java语言没有什么必然联系，它只与特定的二进制文件：Class文件有关。

**2.Java**虚拟机结构

这里所讲的体系结构，是指的Java虚拟机的抽象行为，而不是具体的比如HotSpot VM的实现。按照Java虚拟机规范，抽象的Java虚拟机如下图所示。



Java虚拟机体系结构(3).png

**2.1 Class**文件格式

Java文件被编译后生成了Class文件，这种二进制格式文件不依赖于特定的硬件和操作系统。每一个Class文件中都对应着唯一一个类或者接口的的定义信息，但是类或者接口并不一定定义在文件中，比如类和接口可以通过类加载器来直接生成。

ClassFile的文件结构如下所示。

|  |
| --- |
| ClassFile {  u4 magic; //魔数，固定值为0xCAFEBABE，用来判断当前文件是能被Java虚拟机处理的Class文件  u2 minor\_version; //副版本号  u2 major\_version; //主版本号  u2 constant\_pool\_count; //常量池计数器  cp\_info constant\_pool[constant\_pool\_count-1]; //常量池  u2 access\_flags; //类和接口层次的访问标志  u2 this\_class; //类索引  u2 super\_class; //父类索引  u2 interfaces\_count; //接口计数器  u2 interfaces[interfaces\_count]; //接口表  u2 fields\_count; //字段计数器  field\_info fields[fields\_count]; //字段表  u2 methods\_count; //方法计数器  method\_info methods[methods\_count]; //方法表  u2 attributes\_count; //属性计数器  attribute\_info attributes[attributes\_count]; //属性表  } |

**2.2** 类加载器子系统

类加载器子系统通过多种类加载器来查找和加载Class文件到 Java 虚拟机中。Java虚拟机有两种类加载器：系统加载器和用户自定义加载器。其中系统加载器包括以下三种：  
**Bootstrap ClassLoader**（引导类加载器）  
用C/C++代码实现的加载器，用于加载指定的JDK的核心类库，比如java.lang.、*java.uti.*等这些系统类。它用来加载以下目录中的类库：

• $JAVA\_HOME/jre/lib目录

• -Xbootclasspath参数指定的目录

Java虚拟机的启动就是通过引导类加载器创建一个初始类来完成的。由于类加载器是使用平台相关的底层C/C++语言实现的， 所以该加载器不能被Java代码访问到。但是我们可以查询某个类是否被引导类加载器加载过。

**Extensions ClassLoader**（拓展类加载器）  
用于加载 Java 的拓展类 ，用来提供除了系统类之外的额外功能。它用来加载以下目录中的类库：

• 加载$JAVA\_HOME/jre/lib/ext目录

• 系统属性java.ext.dir所指定的目录。

**Application ClassLoader**（应用程序类加载器）

又称作System ClassLoader(系统类加载器)，这是因为这个类加载器可以通过ClassLoader的getSystemClassLoader方法获取到。它用来加载以下目录中的类库：

• 当前应用程序Classpath目录

• 系统属性java.class.path指定的目录

用户自定义加载器，则是通过继承 java.lang.ClassLoader类的方式来实现自己的类加载器。  
类加载器子系统除了要加载Class文件类到 Java 虚拟机中，还必须负责验证被导入的Class类的正确性，为类变量分配并初始化内存，以及帮助解析符号引用。这些动作必须严格按以下顺序进行：

1.加载：查找并加载Class文件。  
2.链接：验证、准备、以及解析。

• 验证：确保被导入类型的正确性。

• 准备：为类的静态字段分配字段，并用默认值初始化这些字段。

• 解析：根据运行时常量池的符号引用来动态决定具体值得过程。

3.初始化：将类变量初始化为正确初始值。

**2.3** 数据类型

Java虚拟机与Java语言的数据类型相似，可以分为两类：基本类型和引用类型。Java虚拟机希望编译器在编译期间尽可能的完成类型检查，使得虚拟机在运行期间无需进行类型检查操作。

**2.4** 运行时数据区域

很多人将Java的内存分为堆内存（heap）和栈内存（Stack），这种分发不够准确，Java的内存区域划分实际上远比这复杂。  
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不同的数据区域，根据《Java虚拟机规范（Java SE7版）》的规定，这些数据区域分别为程序计数器、Java虚拟机栈、本地方法栈、Java堆和方法区，下面我们来一一的对它们进行介绍。

**2.4.1** 程序计数器

为了保证程序能够连续地执行下去，处理器必须具有某些手段来确定下一条指令的地址，而程序计数器正是起到这种作用。  
程序计数器（Program Counter Register）也叫做PC寄存器，是一块较小的内存空间。在虚拟机概念模型中，字节码解释器工作时就是通过改变程序计数器来选取下一条需要执行的字节码指令，Java虚拟机的多线程是通过轮流切换并分配处理器执行时间的方式来实现的，在一个确定的时刻只有一个处理器执行一条线程中的指令，为了在线程切换后能恢复到正确的执行位置，每个线程都会有一个独立的程序计数器，因此，程序计数器是线程私有的。如果线程执行的方法不是Native方法，则程序计数器保存正在执行的字节码指令地址，如果是Native方法则程序计数器的值则为空（Undefined）。程序计数器是Java虚拟机规范中唯一没有规定任何OutOfMemoryError情况的数据区域。

**2.4.2 Java**虚拟机栈

每一条Java虚拟机线程都有一个线程私有的Java虚拟机栈（Java Virtual Machine Stacks)。它的生命周期与线程相同，与线程是同时创建的。Java虚拟机栈存储线程中Java方法调用的状态，包括局部变量、参数、返回值以及运算的中间结果等。一个Java虚拟机栈包含了多个栈帧，一个栈帧用来存储局部变量表、操作数栈、动态链接、方法出口等信息。当线程调用一个Java方法时，虚拟机压入一个新的栈帧到该线程的Java栈中，当该方法执行完成，这个栈帧就从Java栈中弹出。我们平常所说的栈内存（Stack）指的就是Java虚拟机栈。  
Java虚拟机规范中定义了两种异常情况：

• 如果线程请求分配的栈容量超过Java虚拟机所允许的的最大容量，Java虚拟机会抛出StackOverflowError。

• 如果Java虚拟机栈可以动态扩展（大部分Java虚拟机都可以动态扩展），但是扩展时无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的Java虚拟机栈，则会抛出OutOfMemoryError异常。

**2.4.3** 本地方法栈

Java虚拟机实现可能要用到C Stacks来支持Native语言，这个C Stacks就是本地方法栈（Native Method Stack）。它与Java虚拟机栈类似，只不过本地方法栈是用来支持Native方法服务。如果Java虚拟机不支持Native方法，并且也不依赖于C Stacks，可以无需支持本地方法栈。在Java虚拟机规范中对本地方法栈的语言和数据结构等没有强制规定，因此具体的Java虚拟机可以自由实现它，比如HotSpot VM将本地方法栈和Java虚拟机栈合二为一。  
与Java虚拟机栈类似，本地方法栈也会抛出 StackOverflowError和OutOfMemoryError异常

**2.4.4 Java**堆

Java堆（Java Heap）是被所有线程共享的运行时内存区域。Java堆用来存放对象实例，几乎所有的对象实例都在这里分配内存。Java堆存储的对象被垃圾收集器管理，这些受管理的对象无需也无法显示的销毁。从内存回收的角度，Java堆可以粗略的分为新生代和老年代。从内存分配的角度Java堆中可能划分出多个线程私有的分配缓冲区。不管如何划分，Java堆存储的内容是不变的，进行划分是为了能更快的回收或者分配内存。  
Java堆的容量可以时固定的，也可以动态的扩展。Java堆的所使用的内存在物理上不需要连续，逻辑上连续即可。  
Java虚拟机规范中定义了一种异常情况：

• 如果在堆中没有足够的内存来完成实例分配，并且堆也无法进行扩展时，则会抛出OutOfMemoryError异常。

**2.4.5** 方法区

方法区（Method Area）是被所有线程共享的运行时内存区域。用来存储已经被Java虚拟机加载的类的结构信息，包括：  
运行时常量池、字段和方法信息、静态变量等数据。方法区是Java堆的逻辑组成部分，它一样在物理上不需要连续，并且可以选择在方法区中不实现垃圾收集。方法区并不等同于永久代，只是因为HotSpot VM使用永久代来实现方法区，对于其他的Java虚拟机，比如J9和JRockit等，并不存在永久代概念。  
Java虚拟机规范中定义了一种异常情况：

• 如果方法区的内存空间不满足内存分配需求时，Java虚拟机会抛出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。在2.1 Class文件格式这一小节中我们得知，Class文件不仅包含了类的版本、接口、字段和方法等信息，还包含了常量池，它用来存放编译时期生成的字面量和符号引用，这些内容会在类加载后存放在方法区的运行时常量池中。运行时常量池可以理解为是类或接口的常量池的运行时表现形式。  
Java虚拟机规范中定义了一种异常情况：  
当创建类或接口时，如果构造运行时常量池所需的内存超过了方法区所能提供的最大值，Java虚拟机会抛出OutOfMemoryError异常。

参考资料  
《深入理解Java虚拟机 第二版》  
《Java虚拟机规范（Java SE7版）》  
[理解Java虚拟机体系结构](http://www.cnblogs.com/lao-liang/p/5110710.html)  
[目前主流的 Java 虚拟机有哪些?-知乎](https://www.zhihu.com/question/29265430?sort=created)  
[jvm运行时数据区域解析](http://www.open-open.com/lib/view/open1383745340321.html)  
[Java虚拟机原理图解](http://blog.csdn.net/column/details/jvm-principle.html)  
[深入探讨 Java 类加载器](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/)