

**1.1 JAVA优点：**

* “一次编写，到处运行”；
* 提供了相对安全的内存管理和访问机制，避免了绝大部分内存泄露和指针越界问题；
* 实现了热点代码检测和运行时编译及优化，使得Java应用能随着运行时间的增加而获得更高的性能；
* 有一套完善的应用程序接口，还有无数来自商业机构和开源社区的第三方类库

**1.2　Java技术体系**

1. Java技术体系：

Java程序设计语言

Java虚拟机

Java API类库

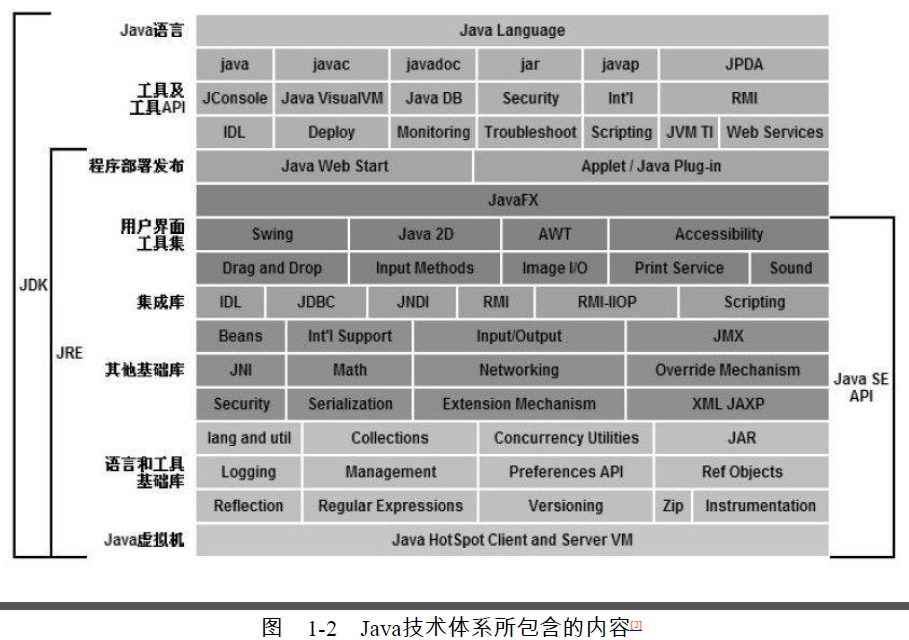
Class文件格式

来自商业机构和开源社区的第三方Java类库

2. JDK（Java Development Kit）:Java程序设计语言、Java虚拟机、Java API类库.是用于支持Java程序开发的最小环境

3. 可把Java虚拟机和Java API类库中的JavaSE API子集[1]统称为JRE（Java Runtime Environment），支持Java程序运行的标准环境。

4. 根据组成部分的功能划分:



5.按技术所服务的领域划分，或者说按照Java技术关注的重点业务领域来划分:

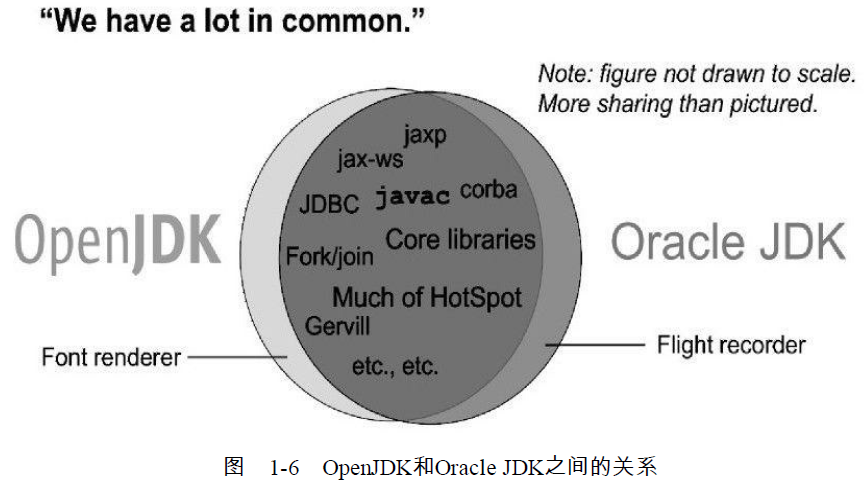
* Java Card：支持一些Java小程序（Applets）运行在小内存设备（如智能卡）上的平台。
* Java ME（Micro Edition）：支持Java程序运行在移动终端（手机、PDA）上的平台，对Java API有所精简，并加入了针对移动终端的支持，这个版本以前称为J2ME。
* Java SE（Standard Edition）：支持面向桌面级应用（如Windows下的应用程序）的Java平台，提供了完整的Java核心API，这个版本以前称为J2SE。
* Java EE（Enterprise Edition）：支持使用多层架构的企业应用（如ERP、CRM应用）的Java平台，除了提供Java SE API外，还对其做了大量的扩充[3]并提供了相关的部署支持，以前称为J2EE。

|  |
| --- |
| [3]这些扩展一般以javax.\*作为包名，而以java.\*为包名的包都是Java SE API的核心包，但由于历史原因，一部分曾经是扩展包的API后来进入了核心包，因此核心包中也包含了不少javax.\*的包名。 |

**1.3　Java发展史**

自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach循环）等语法特性都是在JDK1.5中加入的。在虚拟机和API层面上，这个版本改进了Java的内存模型（Java MemoryModel,JMM）、提供了java.util.concurrent并发包等。

**1.4　Java虚拟机发展史**



获取OpenJDK源码: 从Source Bundle Releases页面（地址：http://jdk7.java.net/source.html）取得打包好的源码

**1.5　@(补)展望Java技术的未来**

**1.6　@(补)实战：自己编译JDK**

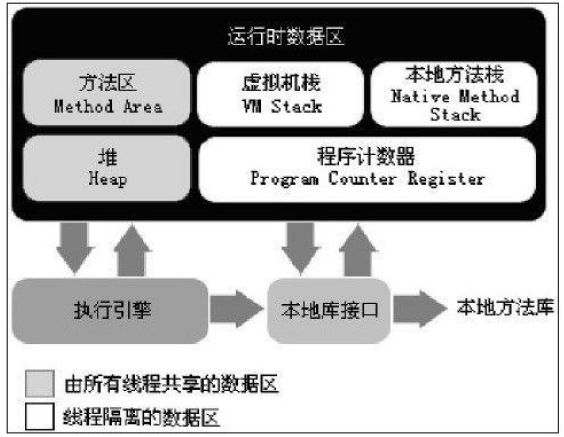
**第二部分　自动内存管理机制**

**第2章　Java内存区域与内存溢出异常**

**2.1 概述**

对Java程序员来说，在虚拟机自动内存管理机制帮助下，不再需要为每一new操作写配对的delete/free代码，不易出现内存泄漏和内存溢出问题.不过，也正是因此，一旦出现内存泄漏和溢出方面的问题，排查错误艰难。

**2.2　运行时数据区域(java虚拟机内存)**



**2.2.1　程序计数器**

较小的内存空间，可看作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，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。

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，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，各条线程之间计数器互不影响，独立存储，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Native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。

**2.2.2　Java虚拟机栈**

线程私有，生命周期与线程相同。

描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame[1]）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

每个方法从调用至执行完,对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

常把Java内存区分为堆内存（Heap）和栈内存（Stack），分法较粗糙.

所指的“栈”是现在讲的虚拟机栈中局部变量表部分。

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，不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或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），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

**2.2.3　本地方法栈**

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，它们之间

的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，而本地方法栈则为虚

拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言、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。

**2.2.4　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[1]，但是随着JIT编译器的发展与逃逸分析技

术逐渐成熟，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”）。从内存回收的角度来看，由于现在收集器基

本都采用分代收集算法，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。从内存分配的角度来看，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、回收等细节将是第3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。

**2.2.5　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、部署程序的开发者来说，很多人都更愿意把方法区

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，这样

HotSpot的垃圾收集器可以像管理Java堆一样管理这部分内存，能够省去专门为方法区编写内

存管理代码的工作。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。原则上，如何实现方法区属于虚拟机实现细节，不受虚拟机规范约束，但使用永久代

来实现方法区，现在看来并不是一个好主意，因为这样更容易遇到内存溢出问题（永久代

有-XX：MaxPermSize的上限，J9和JRockit只要没有触碰到进程可用内存的上限，例如32位系

统中的4GB，就不会出现问题），而且有极少数方法（例如String.intern（））会因这个原因

导致不同虚拟机下有不同的表现。因此，对于HotSpot虚拟机，根据官方发布的路线图信

息，现在也有放弃永久代并逐步改为采用Native Memory来实现方法区的规划了[1]，在目前已

经发布的JDK 1.7的HotSpot中，已经把原本放在永久代的字符串常量池移出。

Java虚拟机规范对方法区的限制非常宽松，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。

**2.2.6　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进入方法区的运行时常

量池中存放。

Java虚拟机对Class文件每一部分（自然也包括常量池）的格式都有严格规定，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[1]。

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

要求常量一定只有编译期才能产生，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。

**2.2.7　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堆中的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，肯定还是

会受到本机总内存（包括RAM以及SWAP区或者分页文件）大小以及处理器寻址空间的限

制。服务器管理员在配置虚拟机参数时，会根据实际内存设置-Xmx等参数信息，但经常忽略

直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），

从而导致动态扩展时出现OutOfMemoryError异常。

**第3章　垃圾收集器与内存分配策略**

**第4章　虚拟机性能监控与故障处理工具**

**第5章　调优案例分析与实战**