深入理解Java虚拟机

极客学院出版

前言

Java 虚拟机屏蔽了与具体操作系统平台相关的信息,使得 Java 语言编译程序只需生成在 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。Java 虚拟机在执行字节码时,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。本文详细的介绍了Java 语言的编译、运行、类加载机制，类文件结构、内存的分配策略、垃圾回收机制、javac 编译、JIT 编译等 JVM 相关知识。

适用人群

Java 程序开发者，对于那些想要了解动态编译与静态编译、Java 语言是如何进行编译和执行的开发者是一本不错的参考材料。

学习前提

本书是中高级教程，需要读者对 Java 语言有比较全面的了解。

目录

前言 1

第 1 章　走进 Java 3

第 2 章　Java 代码编译和执行的整个过程 7

第 3 章　Java 内存区域与内存溢出 12

第 4 章　Class 类文件结构 20

第 5 章　类初始化 33

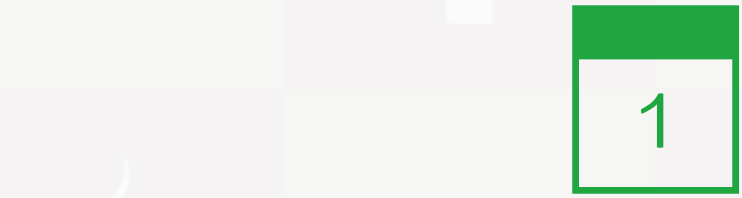
第 6 章　类加载机制 37

第 7 章　多态性实现机制——静态分派与动态分派 48

第 8 章　Java 语法糖 54

第 9 章　javac 编译与 JIT 编译 58

第 10 章　Java 垃圾收集机制 64



1 走进 Java

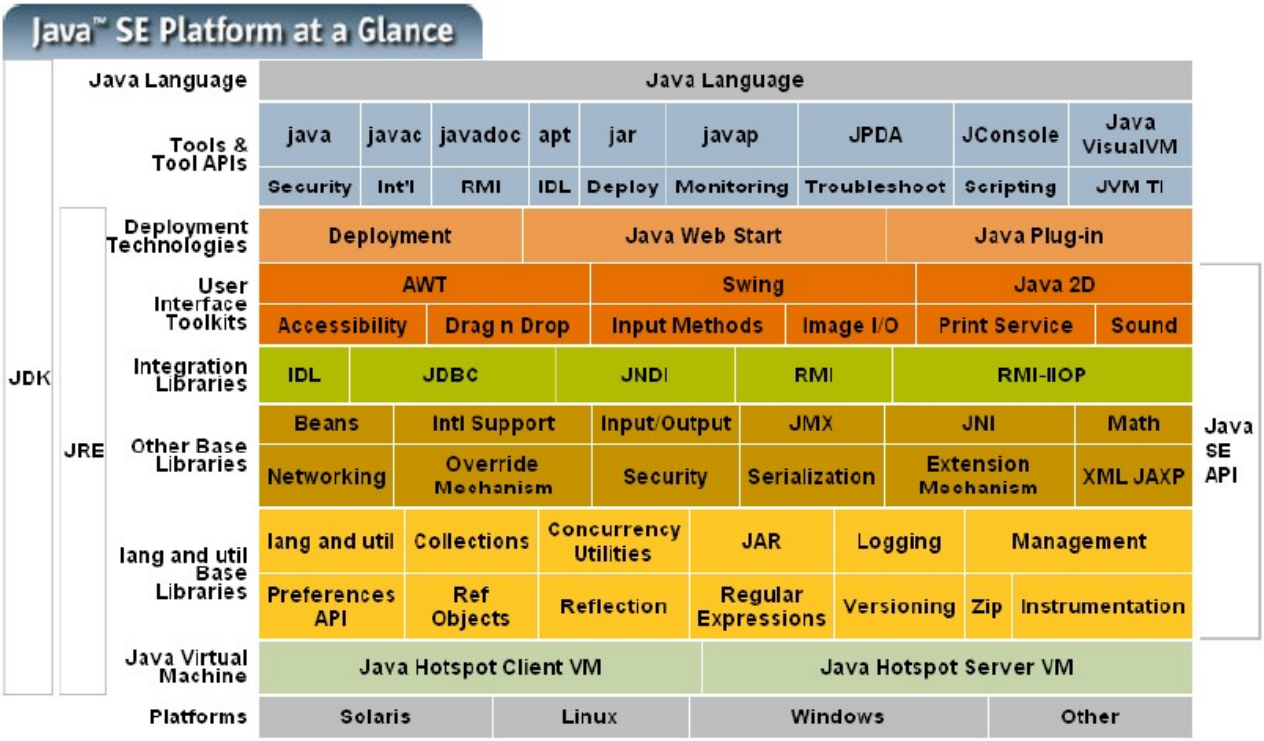
概述

Java 不仅仅是一门编程语言，它还是一个由一系列计算机软件和规范形成的技术体系，这个技术体系提供了完整的用于软件开发和跨平台部署的支持环境，并广泛应用于嵌入式系统、移动终端、企业服务器和大型机等各种场合。时至今日，Java 技术体系已经吸引了近千万软件开发者，这是全球最大的软件开发团队。使用 Java 的设备多达几十亿台，其中包括 8 亿多台个人计算机、21 亿部移动电话及其他手持设备、35 亿个智能卡，以及大量机顶盒、导航系统和其他设备。

Java 能获得如此广泛的认可，除了因为它拥有一门结构严谨、面向对象的编程语言之外，还有许多不可忽视的优点：它摆脱了硬件平台的束缚，实现了“一次编写，到处运行”的理想；它提供了一种相对安全的内存管理和访问机制，避免了绝大部分的内存泄漏和指针越界问题；它实现了热点代码检测和运行时编译及优化，这使得 Java 应用能随着运行时间的增加而获得更高的性能；它有一套完善的应用程序接口，还有无数的来自商业机构和开源社区的第三方类库来帮助实现各种各样的功能。Java 所带来的这些好处让程序的开发效率得到了很大的提升。作为一名Java 程序员，在编写程序时除了尽情发挥 Java 的各种优势外，还应该去了解和思考一下 Java 技术体系中这些技术是如何实现的。认清这些技术的运作本质，是自己思考“程序这样写好不好”的基础和前提。当我们在使用一门技术时，如果不再依赖书本和他人就能得到这个问题的答案，那才算升华到了“不惑”的境界。

Java 技术体系

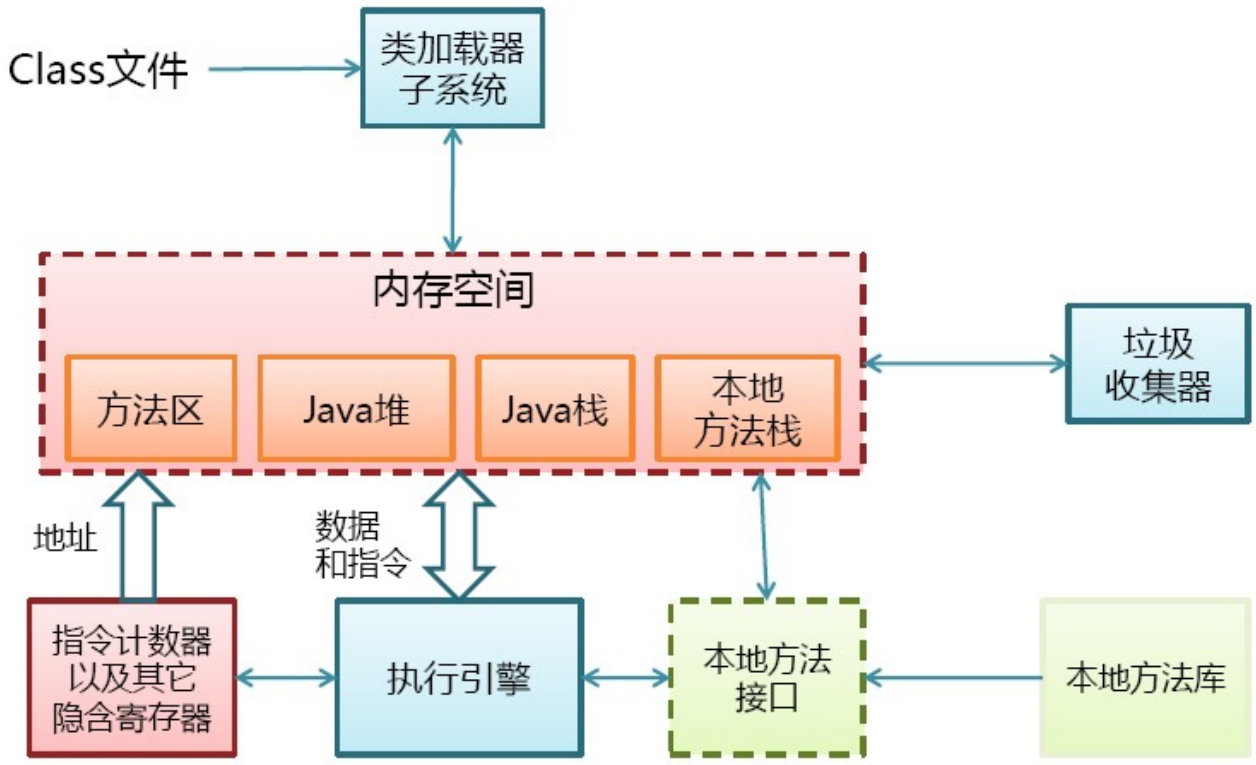
站在 Java 平台的逻辑结构上来说，我们可以从下图来了解 JVM：



以上是根据各个组成部分的功能来进行划分的，如果按照技术所服务的领域来划分，或者说按照Java技术关注的重点业务领域来划分，Java技术体系可以分为四个平台，分别为：

- 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并提供了相关的部署支持，这个版本以前称为 J2EE。

对于 JVM 自身的物理结构，我们可以从下图了解：



什么是 JVM

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。它是一种基于下层的操作系统和硬件平台并利用软件方法来实现的抽象的计算机，可以在上面执行 Java 的字节码程序。

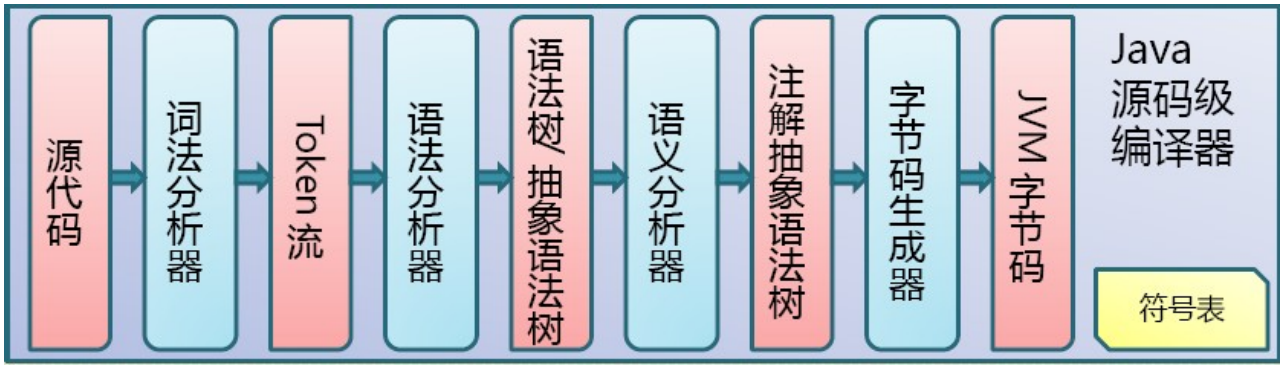
Java 编译器只需面向 JVM，生成 JVM 能理解的代码或字节码文件。Java 源文件经编译器，编译成字节码程序，通过 JVM 将每一条指令翻译成不同平台机器码，通过特定平台运行。

简单的说，JVM 就相当于一台柴油机,它只能用 Java (柴油)运行,JVM 就是 Java 的虚拟机,有了 JVM 才能运行 Java 程序

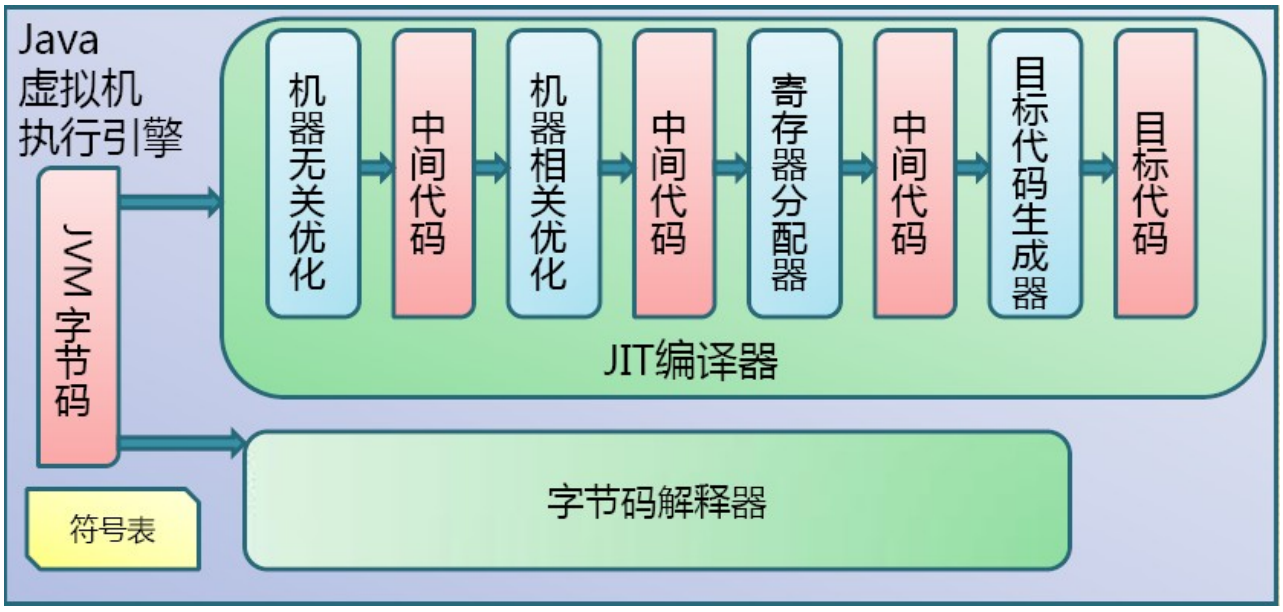
2

Java 代码编译和执行的整个过程

Java 代码编译是由 Java 源码编译器来完成，流程图如下所示：



Java 字节码的执行是由 JVM 执行引擎来完成，流程图如下所示：



Java 代码编译和执行的整个过程包含了以下三个重要的机制：

- Java 源码编译机制
- 类加载机制
- 类执行机制

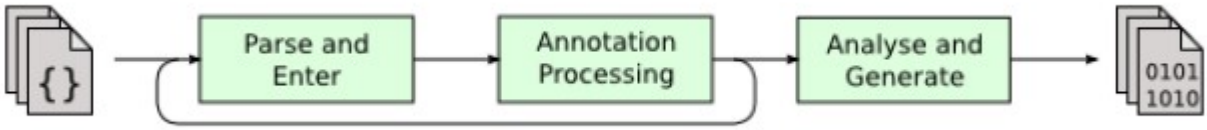
Java 源码编译机制

Java 源码编译由以下三个过程组成：

- 分析和输入到符号表
- 注解处理

- 语义分析和生成 class 文件

流程图如下所示：

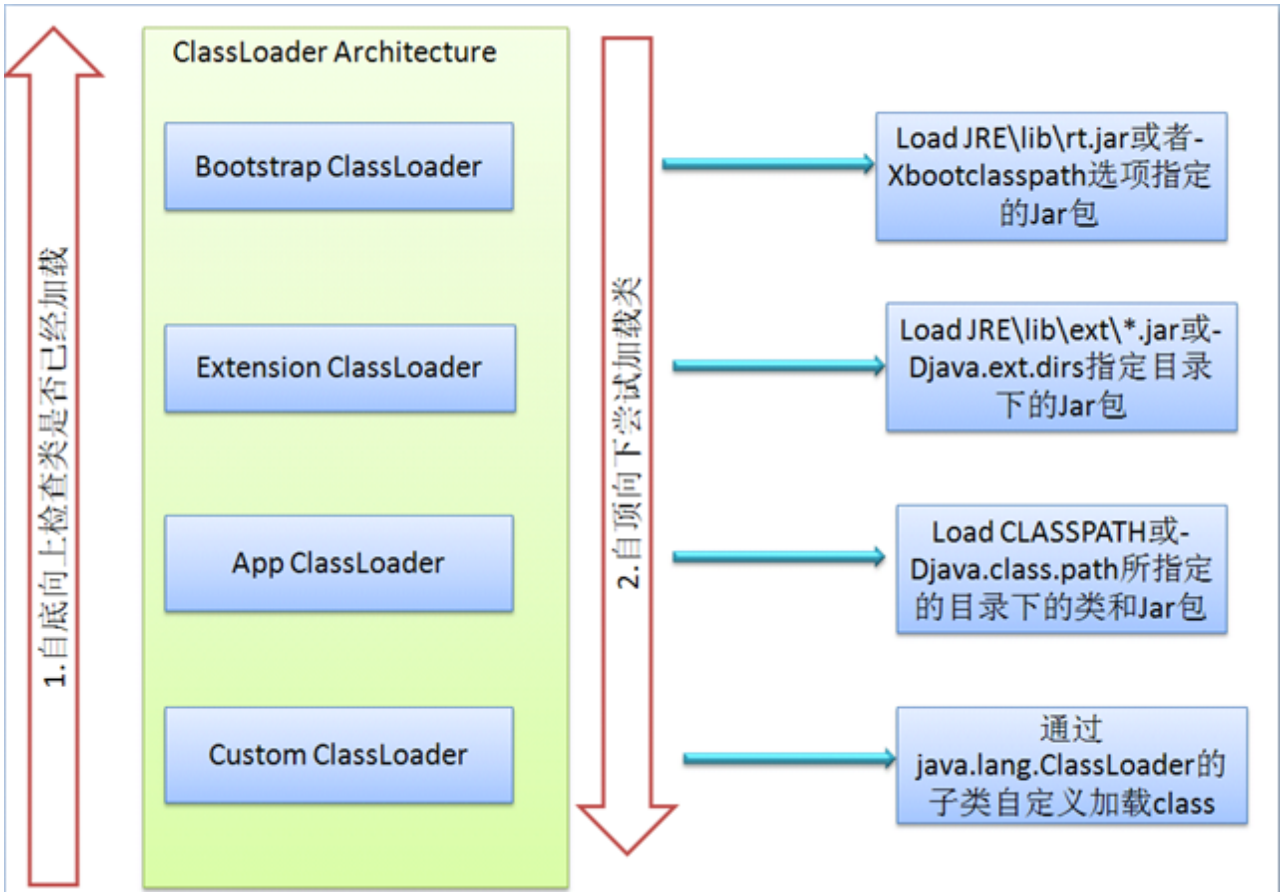


最后生成的 class 文件由以下部分组成：

- 结构信息。包括 class 文件格式版本号及各部分的数量与大小的信息。
- 元数据。对应于 Java 源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池。
- 方法信息。对应 Java 源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息。

类加载机制

JVM 的类加载是通过 ClassLoader 及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



1）Bootstrap ClassLoader

负责加载 `$JAVA_HOME中jre/lib/rt.jar` 里所有的 class，由 C++ 实现，不是 ClassLoader 子类。

2）Extension ClassLoader

负责加载Java平台中扩展功能的一些 jar 包，包括 `$JAVA_HOME中jre/lib/*.jar` 或 `-Djava.ext.dirs` 指定目录下的 jar 包。

3）App ClassLoader

负责记载 classpath 中指定的 jar 包及目录中 class。

4）Custom ClassLoader

属于应用程序根据自身需要自定义的 ClassLoader，如 Tomcat、jboss 都会根据 J2EE 规范自行实现 ClassLoader。

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从 Custom ClassLoader 到 BootStrap ClassLoader 逐层检查，只要某个 Classloader 已加载就视为已加载此类，保证此类只所有 ClassLoade r加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

类执行机制

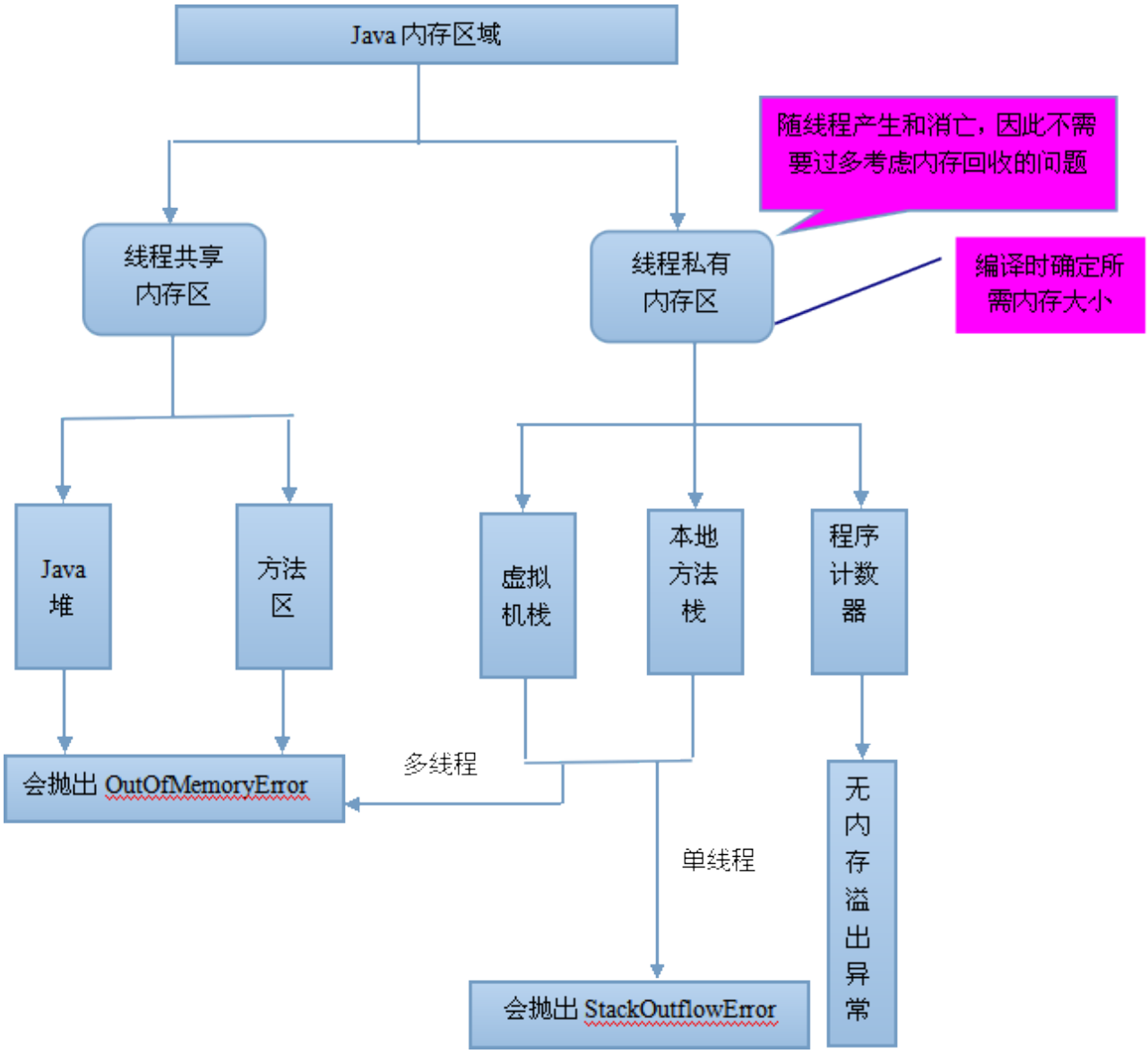
JVM 是基于栈的体系结构来执行 class 字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：

3

Java 内存区域与内存溢出

内存区域

Java 虚拟机在执行 Java 程序的过程中会把他所管理的内存划分为若干个不同的数据区域。Java 虚拟机规范将 JVM 所管理的内存分为以下几个运行时数据区：程序计数器、Java 虚拟机栈、本地方法栈、Java 堆、方法区。下面详细阐述各数据区所存储的数据类型。



程序计数器

一块较小的内存空间，它是当前线程所执行的字节码的行号指示器，字节码解释器工作时通过改变该计数器的值来选择下一条需要执行的字节码指令，分支、跳转、循环等基础功能都要依赖它来实现。每条线程都有一个独立的的程序计数器，各线程间的计数器互不影响，因此该区域是线程私有的。

当线程在执行一个 Java 方法时，该计数器记录的是正在执行的虚拟机字节码指令的地址，当线程在执行的是 Native 方法（调用本地操作系统方法）时，该计数器的值为空。另外，该内存区域是唯一一个在 Java 虚拟机规范中么有规定任何 OOM（内存溢出：OutOfMemoryError）情况的区域。

Java 虚拟机栈

该区域也是线程私有的，它的生命周期也与线程相同。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，栈它是用于支持续虚拟机进行方法调用和方法执行的数据结构。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入了方法表的 Code 属性之中。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

在 Java 虚拟机规范中，对这个区域规定了两种异常情况：

- 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。
- 如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

这两种情况存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。在单线程的操作中，无论是由于栈帧太大，还是虚拟机栈空间太小，当栈空间无法分配时，虚拟机抛出的都是 StackOverflowError 异常，而不会得到 OutOfMemoryError 异常。而在多线程环境下，则会抛出 OutOfMemoryError 异常。

下面详细说明栈帧中所存放的各部分信息的作用和数据结构。

1、局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量，其中存放的数据的类型是编译期可知的各种基本数据类型、对象引用（reference）和 returnAddress 类型（它指向了一条字节码指令的地址）。局部变量表所需的内存空间在编译期间完成分配，即在 Java 程序被编译成 Class 文件时，就确定了所需分配的最大局部变量表的容量。当进入一个方法时，这个方法需要在栈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。

局部变量表的容量以变量槽（Slot）为最小单位。在虚拟机规范中并没有明确指明一个 Slot 应占用的内存空间大小（允许其随着处理器、操作系统或虚拟机的不同而发生变化），一个 Slot 可以存放一个32位以内的数据类型：boolean、byte、char、short、int、float、reference 和 returnAddresss。reference 是对象的引用类

型，returnAddress 是为字节指令服务的，它执行了一条字节码指令的地址。对于 64 位的数据类型（long和double），虚拟机会以高位在前的方式为其分配两个连续的 Slot 空间。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始到局部变量表最大的 Slot 数量，对于 32 位数据类型的变量，索引 n 代表第 n 个 Slot，对于 64 位的，索引 n 代表第 n 和第 n+1 两个 Slot。

在方法执行时，虚拟机是使用局部变量表来完成参数值到参数变量列表的传递过程的，如果是实例方法（非static），则局部变量表中的第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问这个隐含的参数。其余参数则按照参数表的顺序来排列，占用从1开始的局部变量 Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的 Slot。

局部变量表中的 Slot 是可重用的，方法体中定义的变量，作用域并不一定会覆盖整个方法体，如果当前字节码PC计数器的值已经超过了某个变量的作用域，那么这个变量对应的 Slot 就可以交给其他变量使用。这样的设计不仅仅是为了节省空间，在某些情况下 Slot 的复用会直接影响到系统的而垃圾收集行为。

2、操作数栈

操作数栈又常被称为操作栈，操作数栈的最大深度也是在编译的时候就确定了。32 位数据类型所占的栈容量为1,64 位数据类型所占的栈容量为 2。当一个方法开始执行时，它的操作栈是空的，在方法的执行过程中，会有各种字节码指令（比如：加操作、赋值元算等）向操作栈中写入和提取内容，也就是入栈和出栈操作。

Java 虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。因此我们也称 Java 虚拟机是基于栈的，这点不同于 Android 虚拟机，Android 虚拟机是基于寄存器的。

基于栈的指令集最主要的优点是可移植性强，主要的缺点是执行速度相对会慢些；而由于寄存器由硬件直接提供，所以基于寄存器指令集最主要的优点是执行速度快，主要的缺点是可移植性差。

3、动态连接

每个栈帧都包含一个指向运行时常量池（在方法区中，后面介绍）中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。Class 文件的常量池中存在有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用，一部分会在类加载阶段或第一次使用的时候转化为直接引用（如 final、static 域等），称为静态解析，另一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。

4、方法返回地址

当一个方法被执行后，有两种方式退出该方法：执行引擎遇到了任意一个方法返回的字节码指令或遇到了异常，并且该异常没有在方法体内得到处理。无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行。方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状

态。一般来说，方法正常退出时，调用者的 PC 计数器的值就可以作为返回地址，栈帧中很可能保存了这个计数器值，而方法异常退出时，返回地址是要通过异常处理器来确定的，栈帧中一般不会保存这部分信息。

方法退出的过程实际上等同于把当前栈帧出站，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，如果有返回值，则把它压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令。

本地方法栈

该区域与虚拟机栈所发挥的作用非常相似，只是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈则为使用到的本地操作系统（Native）方法服务。

Java 堆

Java Heap 是 Java 虚拟机所管理的内存中最大的一块，它是所有线程共享的一块内存区域。几乎所有的对象实例和数组都在这类分配内存。Java Heap 是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。

根据 Java 虚拟机规范的规定，Java 堆可以处在物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有内存可分配时，并且堆也无法扩展时，将会抛出 OutOfMemoryError 异常。

方法区

方法区也是各个线程共享的内存区域，它用于存储已经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。方法区域又被称为“永久代”，但这仅仅对于 Sun HotSpot 来讲，JRockit 和 IBM J9 虚拟机中并不存在永久代的概念。Java 虚拟机规范把方法区描述为 Java 堆的一个逻辑部分，而且它和 Java Heap 一样不需要连续的内存，可以选择固定大小或可扩展，另外，虚拟机规范允许该区域可以选择不实现垃圾回收。相对而言，垃圾收集行为在这个区域比较少出现。该区域的内存回收目标主要针是对废弃常量的和无用类的回收。运行时常量池是方法区的一部分，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（Class文件常量池），用于存放编译器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。运行时常量池相对于 Class 文件常量池的另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 文件中的常量池的内容才能进入方法区的运行时常量池，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比较多的是 String 类的 intern（）方法。

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。

直接内存

直接内存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，它直接从操作系统中分配，因此不受 Java 堆大小的限制，但是会受到本机总内存的大小及处理器寻址空间的限制，因此它也可能导致 OutOfMemoryError 异常出现。在 JDK1.4 中新引入了 NIO 机制，它是一种基于通道与缓冲区的新 I/O 方式，可以直接从操作系统中分配直接内存，即在堆外分配内存，这样能在一些场景中提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

内存溢出

下面给出个内存区域内存溢出的简单测试方法。

<table>
<tr><th>内存区域</th><th colspan="2">内存溢出的测试方法</th></tr>
<tr><td>Java 堆</td><td colspan="2">无限循环地 new 对象出来，在 List 中保存引用，以不被垃圾收集器回收。另外，该区域也有可能会发生内存泄露（Memory Leak），出现问题时，要注意区别。</td></tr>
<tr><td>方法区</td><td colspan="2">生成大量的动态类，或无线循环调用 String 的 intern（）方法产生不同的 String 对象实例，并在 List 中保存其引用，以不被垃圾收集器回收。后者测试常量池，前者测试方法区的非常量池部分。</td></tr>
<tr><td rowspan="2">虚拟机栈和本地方法栈</td><td>单线程</td><td>多线程</td></tr>
<tr><td>递归调用一个简单的方法：如不断累积的方法。
会抛出 StackOverflowError</td><td>无线循环地创建线程，并未每个线程无限循环地增加内存。
会抛出 OutOfMemoryError</td></tr>
</table>

这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制 Java 堆和方法区这两部分内存的最大值，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，剩下的内存便给了虚拟机栈和本地方法栈，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过减少最大堆和每个线程的栈容量来换取更多的线程。

另外，由于 Java 堆内也可能发生内存泄露（Memory Leak），这里简要说明一下内存泄露和内存溢出的区别：

内存泄露是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java 中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们 new 了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露，

内存溢出是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限。

对象实例化分析

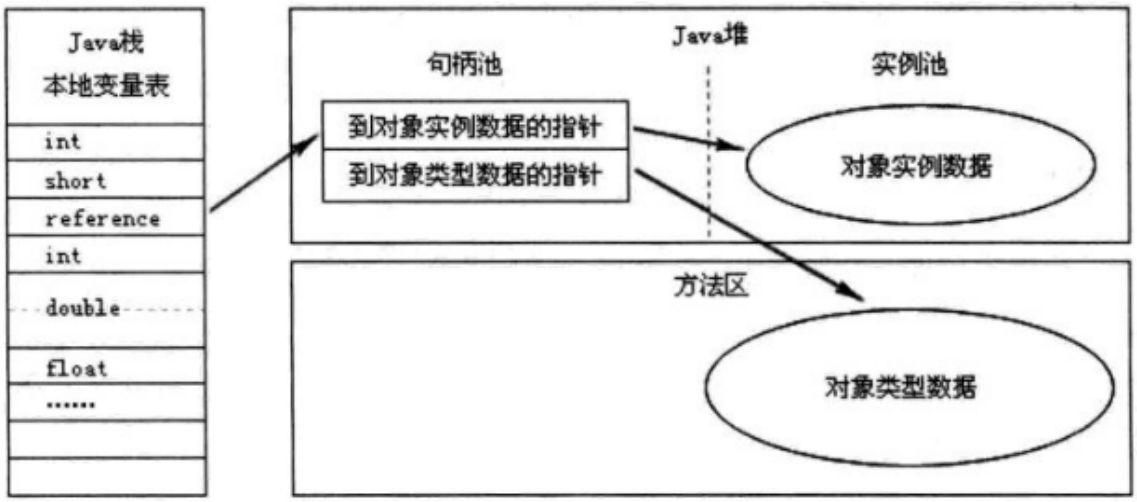
对内存分配情况分析最常见的示例便是对象实例化:

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

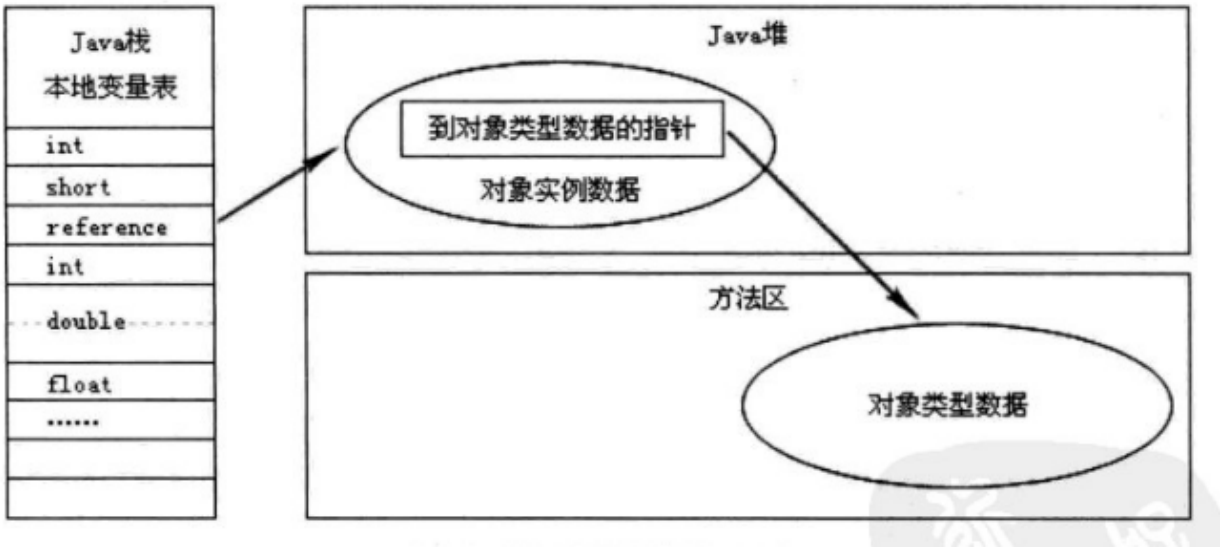
这段代码的执行会涉及 Java 栈、Java 堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对 JVM 虚拟机不了解的 Java 使用这，应该也知道 obj 会作为引用类型（reference）的数据保存在 Java 栈的本地变量表中，而会在 Java 堆中保存该引用的实例化对象，但可能并不知道，Java 堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。

另外，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：使用句柄池和直接使用指针。

通过句柄池访问的方式如下:



通过直接指针访问的方式如下:



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 中存放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 本身不需要修改。使用直接指针访问方式的最大好处是速度快，它节省了一次指针定位的时间开销。目前 Java 默认使用的 HotSpot 虚拟机采用的便是是第二种方式进行对象访问的。

4

Class 类文件结构

平台无关性

Java 是与平台无关的语言，这得益于 Java 源代码编译后生成的存储字节码的文件，即 Class 文件，以及 Java 虚拟机的实现。不仅使用 Java 编译器可以把 Java 代码编译成存储字节码的 Class 文件，使用 JRuby 等其他语言的编译器也可以把程序代码编译成 Class 文件，虚拟机并不关心 Class 的来源是什么语言，只要它符合一定的结构，就可以在 Java 中运行。Java 语言中的各种变量、关键字和运算符的语义最终都是由多条字节码命令组合而成的，因此字节码命令所能提供的语义描述能力肯定会比 Java 语言本身更强大，这便为其他语言实现一些有别于 Java 的语言特性提供了基础，而且这也正是在类加载时要进行安全验证的原因。

类文件结构

Class 文件是一组以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在 Class 文件中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部都是程序运行的必要数据。根据 Java 虚拟机规范的规定，Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储，这种伪结构中只有两种数据类型：无符号数和表。无符号数属于基本数据类型，以 u1、u2、u4、u8 来分别代表 1、2、4、8 个字节的无符号数。表是由多个无符号数或其他表作为数据项构成的符合数据类型，所有的表都习惯性地以“_info”结尾。

整个 Class 文件本质上就是一张表，它由如下所示的数据项构成。

从表中可以看出，无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会使用一个前置的容量计数器加若干个连续的该数据项的形式，称这一系列连续的摸一个类型的数据为某一类型的集合，比如，fields_count 个 field_info 表数据构成了字段表集合。这里需要说明的是：Class 文件中的数据项，都是严格按照上表中的顺序和数量被严格限定的，每个字节代表的含义，长度，先后顺序等都不允许改变。

下表列出了 Class 文件中各个数据项的具体含义：

类型	名称	数量
U4	magic	1
U2	minor_version	1
U2	major_version	1
U2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
U2	access_flags	1
U2	this_class	1
U2	super_class	1
U2	interfaces_count	1
U2	interfaces	interfaces_count
U2	fields_count	1
field_info	fields	fields_count
U2	methods_count	1
method_info	methods	methods_count
U2	atrtributes_count	1
attribute_info	atrtributes	atrtributes_count

从表中可以看出，无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会在其前面使用一个前置的容量计数器来记录其数量，而便跟着若干个连续的数据项，称这一系列连续的某一类型的数据为某一类型的集合，如：fields_count 个 field_info 表数据便组成了方法表集合。这里需要注意的是：Class 文件中各数据项是按照上表的顺序和数量被严格限定的，每个字节代表的含义、长度、先后顺序都不允许改变。

magic 与 version

每个 Class 文件的头 4 个字节称为魔数（magic），它的唯一作用是判断该文件是否为一个能被虚拟机接受的 Class 文件。它的值固定为 0xCAFEBABE。紧接着 magic 的 4 个字节存储的是 Class 文件的次版本号和主版本号，高版本的 JDK 能向下兼容低版本的 Class 文件，但不能运行更高版本的 Class 文件。

constant_pool

major_version 之后是常量池（constant_pool）的入口，它是 Class 文件中与其他项目关联最多的数据类型，也是占用 Class 文件空间最大的数据项目之一。

常量池中主要存放两大类常量：字面量和符号引用。字面量比较接近于 Java 层面的常量概念，如文本字符串、被声明为 final 的常量值等。而符号引用总结起来则包括了下面三类常量：

- 类和接口的全限定名（即带有包名的 Class 名，如：org.lxh.test.TestClass）
- 字段的名称和描述符（private、static 等描述符）

- 方法的名称和描述符（private、static 等描述符）

虚拟机在加载 Class 文件时才会进行动态连接，也就是说，Class 文件中不会保存各个方法和字段的最终内存布局信息，因此，这些字段和方法的符号引用不经过转换是无法直接被虚拟机使用的。当虚拟机运行时，需要从常量池中获得对应的符号引用，再在类加载过程中的解析阶段将其替换为直接引用，并翻译到具体的内存地址中。

这里说明下符号引用和直接引用的区别与关联：

- 符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到了内存中。
- 直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那说明引用的目标必定已经存在于内存之中了。

常量池中的每一项常量都是一个表，共有 11 种（JDK1.7 之前）结构各不相同的表结构数据，没中表开始的第一位是一个 u1 类型的标志位（1-12，缺少 2），代表当前这个常量属于的常量类型。11 种常量类型所代表的具体含义如下表所示：

类型	标志	描述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整性字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整形字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类和接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用

这 11 种常量类型各自均有自己的结构。在 CONSTANT_Class_info 型常量的结构中有一项 name_index 属性，该常属性中存放一个索引值，指向常量池中一个 CONSTANT_Utf8_info 类型的常量，该常量中即保存了该类的全限定名字符串。而 CONSTANT_Fieldref_info、CONSTANT_Methodref_info、CONSTANT_InterfaceMethodref_info 型常量的结构中都有一项index属性，存放该字段或方法所属的类或接口的描述符 CONSTANT_Class_info 的索引项。另外，最终保存的诸如 Class 名、字段名、方法名、修饰符等字符串都是一个 CONSTANT_Utf8_info 类型的常量，也因此，Java 中方法和字段名的最大长度也即是CONSTANT_Utf8_info 型常量的最大长度，在 CONSTANT_Utf8_info 型常量的结构中有一项 length 属性，它是 u2 类型的，即占

用 2 个字节，那么它的最大的 length 即为 65535。因此，Java 程序中如果定义了超过 64KB 英文字符的变量或方法名，将会无法编译。

下表给出了常量池中 11 种数据类型的结构：

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	u1	值为1
	length	u2	UF-8????????????
	bytes	u1	长度为length?UTF-8??????
CONSTANT_Integer_info	tag	u1	值为3
	bytes	u4	按照高位在前存储的int?
CONSTANT_Float_info	tag	u1	值为4
	bytes	u4	按照高位在前存储的float?
CONSTANT_Long_info	tag	u1	值为5
	bytes	u8	按照高位在前存储的long?
	tag	u1	值为6

CONSTANT_Double_info	bytes	u8	按照高位在前存储的double?
CONSTANT_Class_info	tag	u1	值为7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info	tag	u1	值为8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	u1	值为9
	index	u2	指向声明字段的类或接口描述符CONSTANT_Class_info????
	index	u2	指向字段名称及类型描述符CONSTANT_NameAndType_info????
CONSTANT_Methodref_info	tag	u1	值为10
	index	u2	指向声明方法的类描述符CONSTANT_Class_info????
	index	u2	指向方法名称及类型描述符CONSTANT_NameAndType_info????
	tag	u1	值为11

CONSTANT_InrerfaceMethodref_info	index	u2	指向声明方法的接口描述符CONSTANT_Class_info????
	index	u2	指向方法名称及类型描述符CONSTANT_NameAndType_info????
CONSTANT_NameAndType_info	tag	u1	值为12
	index	u2	指向字段或方法名称常量项目的索引
	index	u2	指向该字段或方法描述符常量项的索引

access_flag

在常量池结束之后，紧接着的 2 个字节代表访问标志（access_flag），这个标志用于识别一些类或接口层次的访问信息，包括：这个 Class 是类还是接口，是否定义为 public 类型，abstract 类型，如果是类的话，是否声明为 final，等等。每种访问信息都由一个十六进制的标志值表示，如果同时具有多种访问信息，则得到的标志值为这几种访问信息的标志值的逻辑或。

this_class、super_class、interfaces

类索引（this_class）和父类索引（super_class）都是一个 u2 类型的数据，而接口索引集合（interfaces）则是一组 u2 类型的数据集合，Class 文件中由这三项数据来确定这个类的继承关系。类索引、父类索引和接口索引集合都按照顺序排列在访问标志之后，类索引和父类索引两个 u2 类型的索引值表示，它们各自指向一个类型为 COMNSTANT_Class_info 的类描述符常量，通过该常量中的索引值找到定义在 COMNSTANT_Utf8_info 类型的常量中的全限定名字符串。而接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 implements 语句（如果这个类本身是个接口，则应当是 extend 语句）后的接口顺序从左到右排列在接口的索引集合中。

fields

字段表（field_info）用于描述接口或类中声明的变量。字段包括了类级变量或实例级变量，但不包括在方法内声明的变量。字段的名字、数据类型、修饰符等都是无法固定的，只能引用常量池中的常量来描述。下面是字段表的最种格式：

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
Attribute_info	attributes	attributes_count

其中的 access_flags 与类中的 access_flagsfei 类似，是表示数据类型的修饰符，如 public、static、volatile 等。后面的 name_index 和 descriptor_index 都是对常量池的引用，分别代表字段的简单名称及字段和方法的描述符。这里简单解释下“简单名称”、“描述符”和“全限定名”这三种特殊字符串的概念。

前面有所提及，全限定名即指一个事物的完整的名称，如在 org.lxh.test 包下的 TestClass 类的全限定名为：`org/lxh/test/TestClass`，即把包名中的“.”改为“/”，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会加入一个“；”来表示全限定名结束。简单名称则是指没有类型或参数修饰的方法或字段名称，如果一个类中有这样一个方法 boolean get(int name)和一个变量 private final static int m，则他们的简单名称则分别为 get()和 m。

而描述符的作用则是用来描述字段的数据类型、方法的参数列表（包括数量、类型以及顺序等）和返回值的。根据描述符规则，详细的描述符标示字的含义如下表所示：

标示字符	含义
B	基本类型 byte
C	基本类型 char
S	基本类型 shot
I	基本类型 int
F	基本类型 float
D	基本类型 double
J	基本类型 long
Z	基本类型 boolean
V	无返回值的 void
L，如 Lorg/lxh/test/TestClass	对象类型

对于数组类型，每一维度将使用一个前置的“[”字符来描述，如一个整数数组“int [][]”将为记录为“[[I”，而一个 String 类型的数组“String[]”将被记录为“[Ljava/lang/String”。

用方法描述符描述方法时，按照先参数后返回值的顺序描述，参数要按照严格的顺序放在一组小括号内，如方法 int getIndex(String name,char[] tgc,int start,int end,char target) 的描述符为“（Ljava/lang/String[CIIC）I”。

字段表包含的固定数据项目到 descriptor_index 为止就结束了，但是在它之后还紧跟着一个属性表集合用于存储一些额外的信息。比如，如果在类中有如下字段的声明：staticfinalint m = 2；那就可能会存在一项名为ConstantValue 的属性，它指向常量 2。关于 attribute_info 的详细内容，在后面关于属性表的项目中会有详细介绍。

最后需要注意一点：字段表集合中不会列出从父类或接口中继承而来的字段，但有可能列出原本 Java 代码中不存在的字段。比如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

methods

方法表（method_info）的结构与属性表的结构相同，不过多赘述。方法里的 Java 代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为“Code”的属性里，关于属性表的项目，同样会在后面详细介绍。

与字段表集合相对应，如果父类方法在子类中没有被覆写，方法表集合中就不会出现来自父类的方法信息。但同样，有可能会出现由编译器自动添加的方法，最典型的便是类构造器 “” 方法和实例构造器 “” 方法。

在 Java 语言中，要重载一个方法，除了要与原方法具有相同的简单名称外，还要求必须拥有一个与原方法不同的特征签名，特征签名就是一个方法中各个参数在常量池中的字段符号引用的集合，也就是因为返回值不会包含在特征签名之中，因此 Java 语言里无法仅仅依靠返回值的不同来对一个已有方法进行重载。

ttributes

属性表（attribute_info）在前面已经出现过多系，在 Class 文件、字段表、方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。

属性表集合的限制没有那么严格，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，但 Java 虚拟机运行时会忽略掉它不认识的属性。Java 虚拟机规范中预定义了 9 项虚拟机应当能识别的属性（JDK1.5 后又增加了一些新的特性，因此不止下面 9 项，但下面 9 项是最基本也是必要，出现频率最高的），如下表所示：

属性名称	使用位置	含义
Code	方法表	Java 代码编译成的字节码指令
ConstantValue	字段表	fnal 关键字定义的常量值
Deprecated	类、方法表、字段表	被声明为 deprecated 的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTable	Code 属性	Java 源码的行号与字节码指令间的对应关系
LocalVariableTable	Code 属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类、方法表、字段表	标示类、方法或字段等是编译器自动生成的

对于每个属性，它的名称都需要从常量池中引用一个 CONSTANT_Utf8_info 类型的常量来表示，每个属性值的结构是完全可以自定义的，只需说明属性值所占用的位数长度即可。一个符合规则的属性表至少应具有 “attribute_name_info”、“attribute_length” 和至少一项信息属性。

Code 属性

前面已经说过，Java 程序方法体中的代码讲过 javac 编译后，生成的字节码指令便会存储在 Code 属性中，但并非所有的方法表都必须存在这个属性，比如接口或抽象类中的方法就不存在 Code 属性。如果方法表有 Code 属性存在，那么它的结构将如下表所示：

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
arrtibute_info	attributes	attributes_count

attribute_name_index 是一项指向 CONSTANT_Utf8_info 型常量的索引，常量值固定为 “Code”，它代表了该属性的名称。attribute_length 指示了属性值的长度，由于属性名称索引与属性长度一共是 6 个字节，所以属性值的长度固定为整个属性表的长度减去 6 个字节。

max_stack 代表了操作数栈深度的最大值，max_locals 代表了局部变量表所需的存储空间，它的单位是Slot，并不是在方法中用到了多少个局部变量，就把这些局部变量所占 Slot 之和作为 max_locals 的值，原因是局部变量表中的 Slot 可以重用。

code_length 和 code 用来存储 Java 源程序编译后生成的字节码指令。code 用于存储字节码指令的一系列字节流，它是 u1 类型的单字节，因此取值范围为 0x00 到 0xFF，那么一共可以表达 256 条指令，目前，Java 虚拟机规范已经定义了其中 200 条编码值对应的指令含义。code_length 虽然是一个 u4 类型的长度值，理论上可以达到 2^32-1，但是虚拟机规范中限制了一个方法不允许超过 65535 条字节码指令，如果超过了这个限制，Javac 编译器将会拒绝编译。

字节码指令之后是这个方法的显式异常处理表集合（exception_table），它对于 Code 属性来说并不是必须存在的。它的格式如下表所示：

类型	名称	数量
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_pc	1

它包含四个字段，这些字段的含义为：如果字节码从第 start_pc 行到第 end_pc 行之间（不含 end_pc 行）出现了类型为 catch_type 或其子类的异常（catch_type为指向一个 CONSTANT_Class_info 型常量的索引），则转到第 handler_pc 行继续处理，当 catch_pc 的值为 0 时，代表人和的异常情况都要转到 handler_pc 处进行处理。异常表实际上是 Java 代码的一部分，编译器使用异常表而不是简单的跳转命令来实现 Java 异常即 finally 处理机制，也因此，finally 中的内容会在 try 或 catch 中的 return 语句之前执行，并且在 try 或 catch 跳转到 finally 之前，会将其内部需要返回的变量的值复制一份副本到最后一个本地表量表的 Slot中，也因此便有了http://blog.csdn.net/ns_code/article/details/17485221这篇文章中出现的情况。

Code 属性是 Class 文件中最重要的一个属性，如果把一个 Java 程序中的信息分为代码和元数据两部分，那么在整个 Class 文件里，Code 属性用于描述代码，所有的其他数据项目都用于描述元数据。

Exception 属性

这里的 Exception 属性的作用是列举出方法中可能抛出的受查异常，也就是方法描述时在 throws 关键字后面列举的异常。它的结构很简单，只有 attribute_name_index、attribute_length、number_of_exceptions、exception_index_table 四项，从字面上便很容易理解，这里不再详述。

LineNumberTable 属性

它用于描述 Java 源码行号与字节码行号之间的对应关系。

LocalVariableTable 属性

它用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的对应关系。

SourceFile 属性

它用于记录生成这个 Class 文件的源码文件名称。

ConstantValue 属性

ConstantValue 属性的作用是通知虚拟机自动为静态变量赋值，只有被 static 修饰的变量才可以使用这项属性。在 Java 中，对非 static 类型的变量（也就是实例变量）的赋值是在实例构造器方法中进行的；而对于类变量（static 变量），则有两种方式可以选择：在类构造其中赋值，或使用 ConstantValue 属性赋值。

目前 Sun Javac 编译器的选择是：如果同时使用 final 和 static 修饰一个变量（即全局常量），并且这个变量的数据类型是基本类型或 String 的话，就生成 ConstantValue 属性来进行初始化（编译时 Javac 将会为该常量生成 ConstantValue 属性，在类加载的准备阶段虚拟机便会根据 ConstantValue 为常量设置相应的值），如果该变量没有被 final 修饰，或者并非基本类型及字符串，则选择在方法中进行初始化。

虽然有 final 关键字才更符合”ConstantValue“的含义，但在虚拟机规范中并没有强制要求字段必须用 final 修饰，只要求了字段必须用 static 修饰，对 final 关键字的要求是 Javac 编译器自己加入的限制。因此，在实际的程序中，只有同时被 final 和 static 修饰的字段才有 ConstantValue 属性。而且 ConstantValue 的属性值只限于基本类型和 String，很明显这是因为它从常量池中也只能够引用到基本类型和 String 类型的字面量。

ConstantValue 属性的作用是通知虚拟机自动为静态变量赋值，只有被 static 修饰的变量才可以使用这项属性。在 Java 中，对非 static 类型的变量（也就是实例变量）的赋值是在实例构造器方法中进行的；而对于类变量（static 变量），则有两种方式可以选择：在类构造其中赋值，或使用 ConstantValue 属性赋值。

目前 Sun Javac 编译器的选择是：如果同时使用 final 和 static 修饰一个变量（即全局常量），并且这个变量的数据类型是基本类型或 String 的话，就生成 ConstantValue 属性来进行初始化（编译时 Javac 将会为该常量生成 ConstantValue 属性，在类加载的准备阶段虚拟机便会根据 ConstantValue 为常量设置相应的值），如果该变量没有被 final 修饰，或者并非基本类型及字符串，则选择在方法中进行初始化。

虽然有 final 关键字才更符合”ConstantValue“的含义，但在虚拟机规范中并没有强制要求字段必须用 final 修饰，只要求了字段必须用 static 修饰，对 final 关键字的要求是 Javac 编译器自己加入的限制。因此，在实际的程序中，只有同时被 final 和 static 修饰的字段才有 ConstantValue 属性。而且 ConstantValue 的属性值只限于基本类型和 String，很明显这是因为它从常量池中也只能够引用到基本类型和 String 类型的字面量。

下面简要说明下 final、static、static final 修饰的字段赋值的区别：

- static 修饰的字段在类加载过程中的准备阶段被初始化为 0 或 null 等默认值，而后在初始化阶段（触发类构造器）才会被赋予代码中设定的值，如果没有设定值，那么它的值就为默认值。
- final 修饰的字段在运行时被初始化（可以直接赋值，也可以在实例构造器中赋值），一旦赋值便不可更改；
- static final 修饰的字段在 Javac 时生成 ConstantValue 属性，在类加载的准备阶段根据ConstantValue 的值为该字段赋值，它没有默认值，必须显式地赋值，否则 Javac 时会报错。可以理解为在编译期即把结果放入了常量池中。

InnerClasses 属性

该属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那么编译器将会为它及它所包含的内部类生成 InnerClasses 属性。

Deprecated 属性和 Synthetic 属性

该属性用于表示某个类、字段和方法，已经被程序作者定为不再推荐使用，它可以通过在代码中使用 @Deprecated 注释进行设置。

Synthetic 属性

该属性代表此字段或方法并不是 Java 源代码直接生成的，而是由编译器自行添加的，如 this 字段和实例构造器、类构造器等。

5

类初始化

类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的 Java 程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

- 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的 Java 代码场景是：使用 new 关键字实例化对象时、读取或设置一个类的静态字段（ static ）时（ 被 static 修饰又被 final 修饰的，已在编译期把结果放入常量池的静态字段除外 ）、以及调用一个类的静态方法时。
- 使用 Java.lang.refect 包的方法对类进行反射调用时，如果类还没有进行过初始化，则需要先触发其初始化。
- 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
- 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

虚拟机规定只有这四种情况才会触发类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。下面举一些例子来说明被动引用。

通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类：

```
class Father{
    public static int m = 33;
    static{
        System.out.println("父类被初始化");
    }
}

class Child extends Father{
    static{
        System.out.println("子类被初始化");
    }
}

public class StaticTest{
    public static void main(String[] args){
        System.out.println(Child.m);
    }
}
```

执行后输出的结果如下：

```
父类被初始化
    33
```

对于静态字段，只有直接定义这个字段的类才会被初始化，因此，通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

常量在编译阶段会存入调用它的类的常量池中，本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化：

```
class Const{
    public static final String NAME = "我是常量";
    static{
        System.out.println("初始化Const类");
    }
}

public class FinalTest{
    public static void main(String[] args){
        System.out.println(Const.NAME);
    }
}
```

执行后输出的结果如下：

```
我是常量
```

虽然程序中引用了 const 类的常量 NAME，但是在编译阶段将此常量的值“我是常量”存储到了调用它的类 FinalTest 的常量池中，对常量 Const.NAME 的引用实际上转化为了 FinalTest 类对自身常量池的引用。也就是说，实际上 FinalTest 的 Class 文件之中并没有 Const 类的符号引用入口，这两个类在编译成 Class 文件后就不存在任何联系了。

通过数组定义来引用类，不会触发类的初始化：

```
class Const{
    static{
        System.out.println("初始化Const类");
    }
}

public class ArrayTest{
    public static void main(String[] args){
        Const[] con = new Const[5];
    }
}
```

执行后不输出任何信息，说明 Const 类并没有被初始化。

但这段代码里触发了另一个名为“LLConst”的类的初始化，它是一个由虚拟机自动生成的、直接继承于java.lang.Object 的子类，创建动作由字节码指令 newarray 触发，很明显，这是一个对数组引用类型的初初始化，而该数组中的元素仅仅包含一个对 Const 类的引用，并没有对其进行初始化。如果我们加入对 con 数组中各个 Const 类元素的实例化代码，便会触发 Const 类的初始化，如下：

```
class Const{
    static{
        System.out.println("初始化Const类");
    }
}

public class ArrayTest{
    public static void main(String[] args){
        Const[] con = new Const[5];
        for(Const a:con)
            a = new Const();
    }
}
```

这样便会得到如下输出结果：

```
初始化Const类
```

根据四条规则的第一条，这里的 new 触发了 Const 类。

最后看一下接口的初始化过程与类初始化过程的不同。

接口也有初始化过程，上面的代码中我们都是用静态语句块来输出初始化信息的，而在接口中不能使用“static{}”语句块，但编译器仍然会为接口生成类构造器，用于初始化接口中定义的成员变量（实际上是 static final 修饰的全局常量）。

二者在初始化时最主要的区别是：当一个类在初始化时，要求其父类全部已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量），才会初始化该父接口。这点也与类初始化的情况很不同，回过头来看第 2 个例子就知道，调用类中的 static final 常量时并不会 触发该类的初始化，但是调用接口中的 static final 常量时便会触发该接口的初始化。

6

类加载机制

类加载过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。它们开始的顺序如下图所示：

其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持 Java 语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

这里简要说明下 Java 中的绑定：绑定指的是把一个方法的调用与方法所在的类(方法主体)关联起来，对 Java 来说，绑定分为静态绑定和动态绑定：

- 静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对 Java，简单的可以理解为程序编译期的绑定。Java 当中的方法只有 final，static，private 和构造方法是前期绑定的。
- 动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在 Java 中，几乎所有的方法都是后期绑定的。

下面详细讲述类加载过程中每个阶段所做的工作。

加载

加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

- 通过一个类的全限定名来获取其定义的二进制字节流。
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为对方法区中这些数据的访问入口。

注意，这里第 1 条中的二进制字节流并不只是单纯地从 Class 文件中获取，比如它还可以从 Jar 包中获取、从网络中获取（最典型的应用便是 Applet）、由其他文件生成（JSP 应用）等。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

加载阶段完成后，虚拟机外部的 二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在 Java 堆中也创建一个 java.lang.Class 类的对象，这样便可以通过该对象访问方法区中的这些数据。

说到加载，不得不提到类加载器，下面就具体讲述下类加载器。

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远远不限于类的加载阶段。对于任意一个类，都需要由它的类加载器和这个类本身一同确定其在就 Java 虚拟机中的唯一性，也就是说，即使两个类来源于同一个 Class 文件，只要加载它们的类加载器不同，那这两个类就必定不相等。这里的“相等”包括了代表类的 Class 对象的 equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用 instanceof 关键字对对象所属关系的判定结果。

站在 Java 虚拟机的角度来讲，只存在两种不同的类加载器：

- 启动类加载器：它使用 C++ 实现（这里仅限于 Hotspot，也就是 JDK1.5 之后默认的虚拟机，有很多其他的虚拟机是用 Java 语言实现的），是虚拟机自身的一部分。
- 所有其他的类加载器：这些类加载器都由 Java 语言实现，独立于虚拟机之外，并且全部继承自抽象类 java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

站在 Java 开发人员的角度来看，类加载器可以大致划分为以下三类：

- 启动类加载器：Bootstrap ClassLoader，跟上面相同。它负责加载存放在 JDK\jre\li (JDK 代表 JDK 的安装目录，下同)下，或被 -Xbootclasspath 参数指定的路径中的，并且能被虚拟机识别的类库（如 rt.jar，所有的 java.* 开头的类均被 Bootstrap ClassLoader 加载）。启动类加载器是无法被 Java 程序直接引用的。
- 扩展类加载器：Extension ClassLoader，该加载器由 sun.misc.Launcher$ExtClassLoader 实现，它负责加载 JDK\jre\lib\ext 目录中，或者由 java.ext.dirs 系统变量指定的路径中的所有类库（如 javax.* 开头的类），开发者可以直接使用扩展类加载器。
- 应用程序类加载器：Application ClassLoader，该类加载器由 sun.misc.Launcher$AppClassLoader 来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。因为 JVM 自带的 ClassLoader 只是懂得从本地文件系统加载标准的 java class 文件，因此如果编写了自己的 ClassLoader，便可以做到如下几点：

- 在执行非置信代码之前，自动验证数字签名。
- 动态地创建符合用户特定需要的定制化构建类。
- 从特定的场所取得 java class，例如数据库中和网络中。

事实上当使用 Applet 的时候，就用到了特定的 ClassLoader，因为这时需要从网络上加载 java class，并且要检查相关的安全信息，应用服务器也大都使用了自定义的 ClassLoader 技术。

这几种类加载器的层次关系如下图所示：

这种层次关系称为类加载器的双亲委派模型。我们把每一层上面的类加载器叫做当前层类加载器的父加载器，当然，它们之间的父子关系并不是通过继承关系来实现的，而是使用组合关系来复用父加载器中的代码。该模型在 JDK1.2 期间被引入并广泛应用于之后几乎所有的 Java 程序中，但它并不是一个强制性的约束模型，而是 Java 设计者们推荐给开发者的一种类的加载器实现方式。

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

使用双亲委派模型来组织类加载器之间的关系，有一个很明显的好处，就是 Java 类随着它的类加载器（说白了，就是它所在的目录）一起具备了一种带有优先级的层次关系，这对于保证 Java 程序的稳定运作很重要。例如，类java.lang.Object 类存放在 `JDK\jre\lib` 下的 rt.jar 之中，因此无论是哪个类加载器要加载此类，最终都会委派给启动类加载器进行加载，这边保证了 Object 类在程序中的各种类加载器中都是同一个类。

验证

验证的目的是为了确保 Class 文件中的字节流包含的信息符合当前虚拟机的要求，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

- 文件格式的验证：验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。经过该阶段的验证后，字节流才会进入内存的方法区中进行存储，后面的三个验证都是基于方法区的存储结构进行的。
- 元数据验证：对类的元数据信息进行语义校验（其实就是对类中的各数据类型进行语法校验），保证不存在不符合 Java 语法规范的元数据信息。
- 字节码验证：该阶段验证的主要工作是进行数据流和控制流分析，对类的方法体进行校验分析，以保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。
- 符号引用验证：这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候（解析阶段中发生该转化，后面会有讲解），主要是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。

准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。
- 这里所设置的初始值通常情况下是数据类型默认的零值（如 0、0L、null、false 等），而不是被在 Java 代码中被显式地赋予的值。

假设一个类变量的定义为：

```
public static int value = 3;
```

那么变量 value 在准备阶段过后的初始值为 0，而不是 3，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 3 的 putstatic 指令是在程序编译后，存放于类构造器 （ ）方法之中的，所以把 value 赋值为 3 的动作将在初始化阶段才会执行。

下表列出了 Java 中所有基本数据类型以及 reference 类型的默认零值：

数据类型	默认零值
int	0
long	0L
short	(short)0
char	‘\u0000’
byte	(byte)0
boolean	false
float	0.0f
double	0.0d
reference	null

这里还需要注意如下几点：

- 对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。
- 对于同时被 static 和 final 修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被 final 修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。
- 对于引用数据类型 reference 来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。

- 如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

如果类字段的字段属性表中存在 ConstantValue 属性，即同时被 final 和 static 修饰，那么在准备阶段变量 value 就会被初始化为 ConstValue 属性所指定的值。

假设上面的类变量 value 被定义为：

```
public static final int value = 3;
```

编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 3。回忆上一篇博文中对象被动引用的第 2 个例子，便是这种情况。我们可以理解为 static final 常量在编译期就将其结果放入了调用它的类的常量池中。

解析

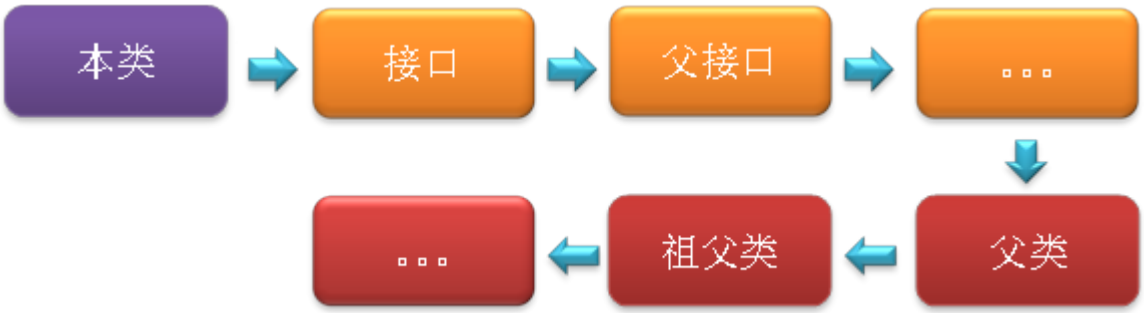
解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。在 Class 类文件结构一文中已经比较过了符号引用和直接引用的区别和关联，这里不再赘述。前面说解析阶段可能开始于初始化之前，也可能在初始化之后开始，虚拟机会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析（初始化之前），还是等到一个符号引用将要被使用前才去解析它（初始化之后）。

对同一个符号引用进行多次解析请求时很常见的事情，虚拟机实现可能会对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标示为已解析状态），从而避免解析动作重复进行。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的 CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info、CONSTANT_InterfaceMethodref_info 四种常量类型。

1、类或接口的解析：判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。

2、字段解析：对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



从下面一段代码的执行结果中很容易看出来字段解析的搜索顺序：

```
class Super{
    public static int m = 11;
    static{
        System.out.println("执行了super类静态语句块");
    }
}


class Father extends Super{
    public static int m = 33;
    static{
        System.out.println("执行了父类静态语句块");
    }
}

class Child extends Father{
    static{
        System.out.println("执行了子类静态语句块");
    }
}

public class StaticTest{
    public static void main(String[] args){
        System.out.println(Child.m);
    }
}
```

执行结果如下：

```
执行了super类静态语句块
执行了父类静态语句块
33
```

如果注释掉 Father 类中对 m 定义的那一行，则输出结果如下：

```
执行了super类静态语句块
11
```

另外，很明显这就是上篇博文中的第 1 个例子的情况，这里我们便可以分析如下：static 变量发生在静态解析阶段，也即是初始化之前，此时已经将字段的符号引用转化为了内存引用，也便将它与对应的类关联在了一起，由于在子类中没有查找到与 m 相匹配的字段，那么 m 便不会与子类关联在一起，因此并不会触发子类的初始化。

最后需要注意：理论上是按照上述顺序进行搜索解析，但在实际应用中，虚拟机的编译器实现可能要比上述规范要求的更严格一些。如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译。如果对上面的代码做些修改，将 Super 改为接口，并将 Child 类继承 Father 类且实现 Super 接口，那么在编译时会报出如下错误：

```
StaticTest.java:24: 对 m 的引用不明确，Father 中的 变量 m 和 Super 中的 变量 m
都匹配
                System.out.println(Child.m);
                                   ^
1 错误
```

3、类方法解析：对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。

4、接口方法解析：与类方法解析步骤类似，知识接口不会有父类，因此，只递归向上搜索父接口就行了。

初始化

初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的 Java 程序代码。在准备阶段，类变量已经被赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序指定的主观计划去初始化类变量和其他资源，或者可以从另一个角度来表达：初始化阶段是执行类构造器()方法的过程。

这里简单说明下()方法的执行规则:

1、()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句中可以赋值，但是不能访问。

2、（ ）方法与实例构造器()方法（类的构造函数）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的()方法执行之前，父类的()方法已经执行完毕。因此，在虚拟机中第一个被执行的()方法的类肯定是java.lang.Object。

3、()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成()方法。

4、接口中不能使用静态语句块，但仍然有类变量（final static）初始化的赋值操作，因此接口与类一样会生成()方法。但是接口鱼类不同的是：执行接口的()方法不需要先执行父接口的()方法，只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的()方法。

5、虚拟机会保证一个类的()方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的()方法，其他线程都需要阻塞等待，直到活动线程执行()方法完毕。如果在一个类的()方法中有耗时很长的操作，那就可能造成多个线程阻塞，在实际应用中这种阻塞往往是很隐蔽的。

下面给出一个简单的例子，以便更清晰地说明如上规则：

```
class Father{
    public static int a = 1;
    static{
        a = 2;
    }
}

class Child extends Father{
    public static int b = a;
}

public class ClinitTest{
    public static void main(String[] args){
        System.out.println(Child.b);
    }
}
```

执行上面的代码，会打印出 2，也就是说 b 的值被赋为了 2。

我们来看得到该结果的步骤。首先在准备阶段为类变量分配内存并设置类变量初始值，这样 A 和 B 均被赋值为默认值 0，而后再在调用()方法时给他们赋予程序中指定的值。当我们调用 Child.b 时，触发 Child 的()方法，根据规则 2，在此之前，要先执行完其父类Father的()方法，又根据规则1，在执行()方法时，需要按 static 语句或 static 变量赋值操作等在代码中出现的顺序来执行相关的 static 语句，因此当触发执行 Fathe r的()方法时，会先将 a 赋值为 1，再执行 static 语句块中语句，将 a 赋值为 2，而后再执行 Child 类的()方法，这样便会将 b 的赋值为 2。

如果我们颠倒一下 Father 类中“public static int a = 1;”语句和“static语句块”的顺序，程序执行后，则会打印出1。很明显是根据规则 1，执行 Father 的()方法时，根据顺序先执行了 static 语句块中的内容，后执行了“public static int a = 1;”语句。

另外，在颠倒二者的顺序之后，如果在 static 语句块中对 a 进行访问（比如将 a 赋给某个变量），在编译时将会报错，因为根据规则 1，它只能对 a 进行赋值，而不能访问。

总结

整个类加载过程中，除了在加载阶段用户应用程序可以自定义类加载器参与之外，其余所有的动作完全由虚拟机主导和控制。到了初始化才开始执行类中定义的 Java 程序代码（亦及字节码），但这里的执行代码只是个开端，它仅限于()方法。类加载过程中主要是将 Class 文件（准确地讲，应该是类的二进制字节流）加载到虚拟机内存中，真正执行字节码的操作，在加载完成后才真正开始。

7

多态性实现机制——静态分派与动态分派

方法解析

Class 文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址。这个特性给 Java 带来了更强大的动态扩展能力，使得可以在类运行期间才能确定某些目标方法的直接引用，称为动态连接，也有一部分方法的符号引用在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。这在前面的“Java 内存区域与内存溢出 ()”一文中有提到。

静态解析成立的前提是：方法在程序真正执行前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在编译器进行编译时就必须确定下来，这类方法的调用称为解析。

在 Java 语言中，符合“编译器可知，运行期不可变”这个要求的方法主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法都不可能通过继承或别的方式重写出其他的版本，因此它们都适合在类加载阶段进行解析。

Java 虚拟机里共提供了四条方法调用字节指令，分别是：

- invokestatic：调用静态方法。
- invokespecial：调用实例构造器方法、私有方法和父类方法。
- invokevirtual：调用所有的虚方法。
- invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法、私有方法、实例构造器和父类方法四类，它们在类加载时就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法（还包括 final 方法），与之相反，其他方法就称为虚方法（final 方法除外）。这里要特别说明下 final 方法，虽然调用 final 方法使用的是 invokevirtual 指令，但是由于它无法覆盖，没有其他版本，所以也无需对方发接收者进行多态选择。Java 语言规范中明确说明了 final 方法是一种非虚方法。

解析调用一定是个静态过程，在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用转化为可确定的直接引用，不会延迟到运行期再去完成。而分派调用则可能是静态的也可能是动态的，根据分派依据的宗量数（方法的调用者和方法的参数统称为方法的宗量）又可分为单分派和多分派。两类分派方式两两组合便构成了静态单分派、静态多分派、动态单分派、动态多分派四种分派情况。

静态分派

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派，静态分派的最典型应用就是多态性中的方法重载。静态分派发生在编译阶段，因此确定静态分配的动作实际上不是由虚拟机来执行的。下面通过一段方法重载的示例程序来更清晰地说明这种分派机制：

```
class Human{
}
class Man extends Human{
}
class Woman extends Human{
}

public class StaticPai{

   public void say(Human hum){
      System.out.println("I am human");
   }
   public void say(Man hum){
      System.out.println("I am man");
   }
   public void say(Woman hum){
      System.out.println("I am woman");
   }

   public static void main(String[] args){
      Human man = new Man();
      Human woman = new Woman();
      StaticPai sp = new StaticPai();
      sp.say(man);
      sp.say(woman);
   }
}
```

上面代码的执行结果如下：

```
I am human
I am human
```

以上结果的得出应该不难分析。在分析为什么会选择参数类型为 Human 的重载方法去执行之前，先看如下代码：

```
Human man = new Man（）;
```

我们把上面代码中的“Human”称为变量的静态类型，后面的“Man”称为变量的实际类型。静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的，而实际类型变化的结果在运行期才可确定。

回到上面的代码分析中，在调用 say()方法时，方法的调用者（回忆上面关于宗量的定义，方法的调用者属于宗量）都为 sp 的前提下，使用哪个重载版本，完全取决于传入参数的数量和数据类型（方法的参数也是数据宗量）。代码中刻意定义了两个静态类型相同、实际类型不同的变量，可见编译器（不是虚拟机，因为如果是根据静态类型做出的判断，那么在编译期就确定了）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，所以在编译阶段，javac 编译器就根据参数的静态类型决定使用哪个重载版本。这就是静态分派最典型的应用。

动态分派

动态分派与多态性的另一个重要体现——方法覆写有着很紧密的关系。向上转型后调用子类覆写的方法便是一个很好地说明动态分派的例子。这种情况很常见，因此这里不再用示例程序进行分析。很显然，在判断执行父类中的方法还是子类中覆盖的方法时，如果用静态类型来判断，那么无论怎么进行向上转型，都只会调用父类中的方法，但实际情况是，根据对父类实例化的子类的不同，调用的是不同子类中覆写的方法，很明显，这里是要根据变量的实际类型来分派方法的执行版本的。而实际类型的确定需要在程序运行时才能确定下来，这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

单分派和多分派

前面给出：方法的接受者（亦即方法的调用者）与方法的参数统称为方法的宗量。但分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。

为了方便理解，下面给出一段示例代码：

```
class Eat{
}
class Drink{
}

class Father{
    public void doSomething(Eat arg){
        System.out.println("爸爸在吃饭");
    }
    public void doSomething(Drink arg){
        System.out.println("爸爸在喝水");
```

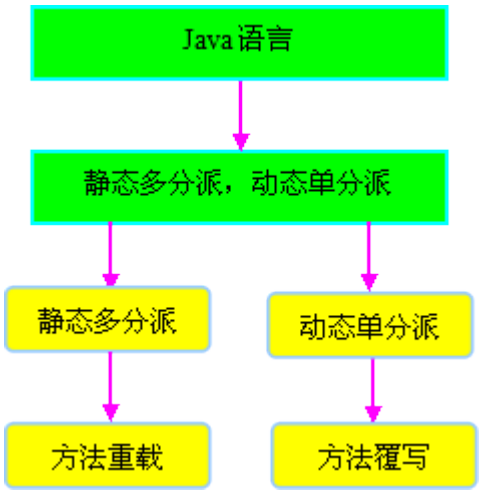
```
    }
}

class Child extends Father{
    public void doSomething(Eat arg){
        System.out.println("儿子在吃饭");
    }
    public void doSomething(Drink arg){
        System.out.println("儿子在喝水");
    }
}

public class SingleDoublePai{
    public static void main(String[] args){
        Father father = new Father();
        Father child = new Child();
        father.doSomething(new Eat());
        child.doSomething(new Drink());
    }
}
```

运行结果应该很容易预测到，如下：

```
爸爸在吃饭
儿子在喝水
```

我们首先来看编译阶段编译器的选择过程，即静态分派过程。这时候选择目标方法的依据有两点：一是方法的接受者（即调用者）的静态类型是 Father 还是 Child，二是方法参数类型是 Eat 还是 Drink。因为是根据两个宗量进行选择，所以 Java 语言的静态分派属于多分派类型。

再来看运行阶段虚拟机的选择，即动态分派过程。由于编译期已经了确定了目标方法的参数类型（编译期根据参数的静态类型进行静态分派），因此唯一可以影响到虚拟机选择的因素只有此方法的接受者的实际类型是 Father 还是 Child。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于单分派类型。



根据以上论证，我们可以总结如下：目前的 Java 语言（JDK1.6）是一门静态多分派、动态单分派的语言。

8

Java 语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。Java 中最常用的语法糖主要有泛型、变长参数、条件编译、自动拆装箱、内部类等。虚拟机并不支持这些语法，它们在编译阶段就被还原回了简单的基础语法结构，这个过程成为解语法糖。

泛型是 JDK1.5 之后引入的一项新特性，Java 语言在还没有出现泛型时，只能通过 Object 是所有类型的父类和类型强制转换这两个特点的配合来实现泛型的功能，这样实现的泛型功能要在程序运行期才能知道 Object 真正的对象类型，在 javac 编译期，编译器无法检查这个 Object 的强制转型是否成功，这便将一些风险转接到了程序运行期中。

Java 语言在 JDK1.5 之后引入的泛型实际上只在程序源码中存在，在编译后的字节码文件中，就已经被替换为了原来的原生类型，并且在相应的地方插入了强制转型代码，因此对于运行期的Java语言来说，ArrayList和 ArrayList就是同一个类。所以泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。

下面是一段简单的 Java 泛型代码：

```
Map<Integer,String> map = new HashMap<Integer,String>();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println(map.get(1));
System.out.println(map.get(2));
```

将这段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都变回了原生类型，如下面的代码所示：

```
Map map = new HashMap();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println((String)map.get(1));
System.out.println((String)map.get(2));
```

为了更详细地说明类型擦除，再看如下代码：

```
import java.util.List;
public class FanxingTest{
    public void method(List<String> list){
        System.out.println("List String");
    }
    public void method(List<Integer> list){
        System.out.println("List Int");
    }
}
```

当用 javac 编译器编译这段代码时，报出了如下错误：

```
FanxingTest.java:3: 名称冲突：method(java.util.List<java.lang.String>) 和 method

(java.util.List<java.lang.Integer>) 具有相同疑符

    public void method(List<String> list){

            ^

FanxingTest.java:6: 名称冲突：method(java.util.List<java.lang.Integer>) 和 metho

d(java.util.List<java.lang.String>) 具有相同疑符

    public void method(List<Integer> list){

            ^

2 错误
```

是因为泛型 List 和 List 编译后都被擦除了，变成了一样的原生类型 List，擦除动作导致这两个方法的特征签名变得一模一样，在 Class 类文件结构一文中讲过，Class 文件中不能存在特征签名相同的方法。

把以上代码修改如下：

```
import java.util.List;
public class FanxingTest{
   public int method(List<String> list){
      System.out.println("List String");
      return 1;
   }
   public boolean method(List<Integer> list){
      System.out.println("List Int");
      return true;
   }
}
```

发现这时编译可以通过了（注意：Java 语言中 true 和 1 没有关联，二者属于不同的类型，不能相互转换，不存在 C 语言中整数值非零即真的情况）。两个不同类型的返回值的加入，使得方法的重载成功了。这是为什么呢?

我们知道，Java 代码中的方法特征签名只包括了方法名称、参数顺序和参数类型，并不包括方法的返回值，因此方法的返回值并不参与重载方法的选择，这样看来为重载方法加入返回值貌似是多余的。对于重载方法的选择来说，这确实是多余的，但我们现在要解决的问题是让上述代码能通过编译，让两个重载方法能够合理地共存于同一个 Class 文件之中，这就要看字节码的方法特征签名，它不仅包括了 Java 代码中方法特征签名中所包含的那

些信息，还包括方法返回值及受查异常表。为两个重载方法加入不同的返回值后，因为有了不同的字节码特征签名，它们便可以共存于一个 Class 文件之中。

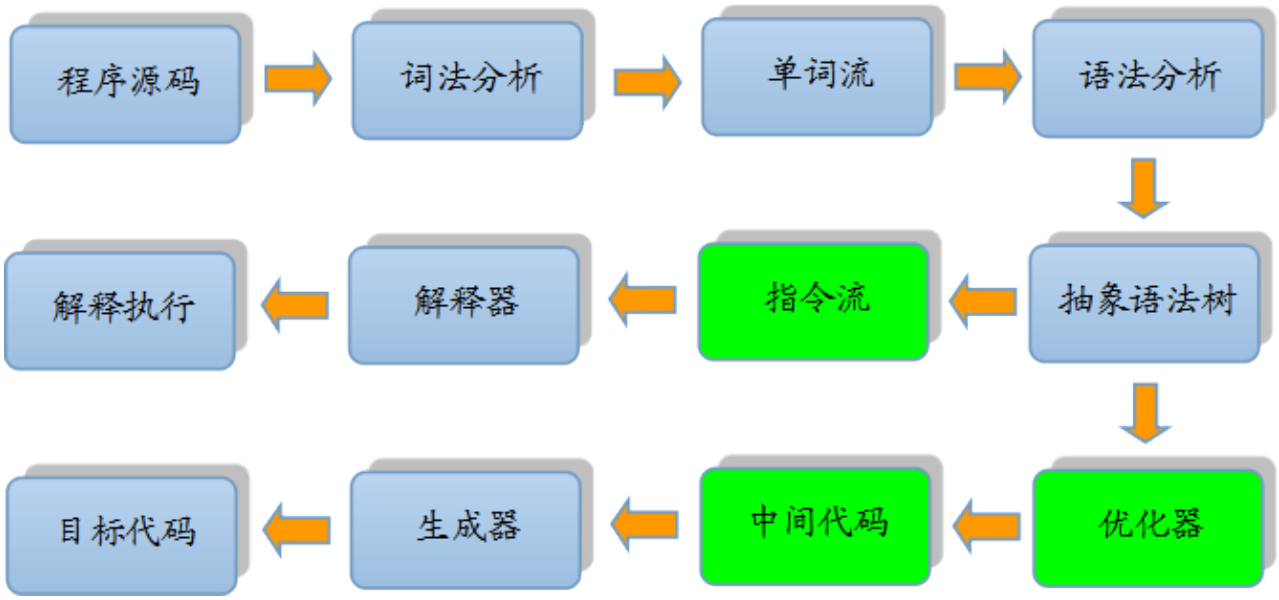
自动拆装箱、变长参数等语法糖也都是在编译阶段就把它们该语法糖结构还原为了原生的语法结构，因此在 Class 文件中也只存在其对应的原生类型，这里不再一一说明。

9

javac 编译与 JIT 编译

编译过程

不论是物理机还是虚拟机，大部分的程序代码从开始编译到最终转化成物理机的目标代码或虚拟机能执行的指令集之前，都会按照如下图所示的各个步骤进行：



其中绿色的模块可以选择性实现。很容易看出，上图中间的那条分支是解释执行的过程（即一条字节码一条字节码地解释执行，如 JavaScript），而下面的那条分支就是传统编译原理中从源代码到目标机器代码的生成过程。

如今，基于物理机、虚拟机等的语言，大多都遵循这种基于现代经典编译原理的思路，在执行前先对程序源码进行词法解析和语法解析处理，把源码转化为抽象语法树。对于一门具体语言的实现来说，词法和语法分析乃至后面的优化器和目标代码生成器都可以选择独立于执行引擎，形成一个完整意义的编译器去实现，这类代表是 C/C++语言。也可以把抽象语法树或指令流之前的步骤实现一个半独立的编译器，这类代表是 Java 语言。又或者可以把这些步骤和执行引擎全部集中在一起实现，如大多数的 JavaScript 执行器。

javac 编译

在 Java 中提到“编译”，自然很容易想到 javac 编译器将 `*.java` 文件编译成为 `*.class` 文件的过程，这里的 javac 编译器称为前端编译器，其他的前端编译器还有诸如 Eclipse JDT 中的增量式编译器 ECJ 等。相对应的还有后端编译器，它在程序运行期间将字节码转变成机器码（现在的 Java 程序在运行时基本都是解释执行加编译执行），如 HotSpot 虚拟机自带的 JIT（Just In Time Compiler）编译器（分 Client 端和 Server 端）。另外，有时候还有可能会碰到静态提前编译器（AOT，Ahead Of Time Compiler）直接把 `*.java` 文件编译成本地机器代码，如 GCJ、Excelsior JET 等，这类编译器我们应该比较少遇到。

下面简要说下 javac 编译（前端编译）的过程。

词法、语法分析

词法分析是将源代码的字符流转变为标记（Token）集合。单个字符是程序编写过程中的的最小元素，而标记则是编译过程的最小元素，关键字、变量名、字面量、运算符等都可以成为标记，比如整型标志 int 由三个字符构成，但是它只是一个标记，不可拆分。

语法分析是根据Token序列来构造抽象语法树的过程。抽象语法树是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，如 bao、类型、修饰符、运算符等。经过这个步骤后，编译器就基本不会再对源码文件进行操作了，后续的操作都建立在抽象语法树之上。

填充符号表

完成了语法分析和词法分析之后，下一步就是填充符号表的过程。符号表是由一组符号地址和符号信息构成的表格。符号表中所登记的信息在编译的不同阶段都要用到，在语义分析（后面的步骤）中，符号表所登记的内容将用于语义检查和产生中间代码，在目标代码生成阶段，党对符号名进行地址分配时，符号表是地址分配的依据。

语义分析

语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是读结构上正确的源程序进行上下文有关性质的审查。语义分析过程分为标注检查和数据及控制流分析两个步骤：

- 标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量和赋值之间的数据类型是否匹配等。
- 数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。

字节码生成

字节码生成是 javac 编译过程的最后一个阶段。字节码生成阶段不仅仅是把前面各个步骤所生成的信息转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。 实例构造器()方法和类构造器()方法就是在这个阶段添加到语法树之中的（这里的实例构造器并不是指默认的构造函数，而是指我们自己重载的构造函数，如果用户代码中没有提供任何构造函数，那编译器会自动添加一个没有参数、访问权限与当前类一致的默认构造函数，这个工作在填充符号表阶段就已经完成了）。

JIT 编译

Java 程序最初是仅仅通过解释器解释执行的，即对字节码逐条解释执行，这种方式的执行速度相对会比较慢，尤其当某个方法或代码块运行的特别频繁时，这种方式的执行效率就显得很低。于是后来在虚拟机中引入了 JIT 编译器（即时编译器），当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“Hot Spot Code”（热点代码），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，完成这项任务的正是 JIT 编译器。

现在主流的商用虚拟机（如Sun HotSpot、IBM J9）中几乎都同时包含解释器和编译器（三大商用虚拟机之一的JRockit 是个例外，它内部没有解释器，因此会有启动相应时间长之类的缺点，但它主要是面向服务端的应用，这类应用一般不会重点关注启动时间）。二者各有优势：当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即执行；当程序运行后，随着时间的推移，编译器逐渐会返回作用，把越来越多的代码编译成本地代码后，可以获取更高的执行效率。解释执行可以节约内存，而编译执行可以提升效率。

HotSpot 虚拟机中内置了两个JIT编译器：Client Complier 和 Server Complier，分别用在客户端和服务端，目前主流的 HotSpot 虚拟机中默认是采用解释器与其中一个编译器直接配合的方式工作。

运行过程中会被即时编译器编译的“热点代码”有两类：

- 被多次调用的方法。
- 被多次调用的循环体。

两种情况，编译器都是以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。要知道一段代码或方法是不是热点代码，是不是需要触发即时编译，需要进行 Hot Spot Detection（热点探测）。目前主要的热点判定方式有以下两种：

- **基于采样的热点探测**：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这段方法代码就是“热点代码”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。
- **基于计数器的热点探测**：采用这种方法的虚拟机会为每个方法，甚至是代码块建立计数器，统计方法的执行次数，如果执行次数超过一定的阀值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

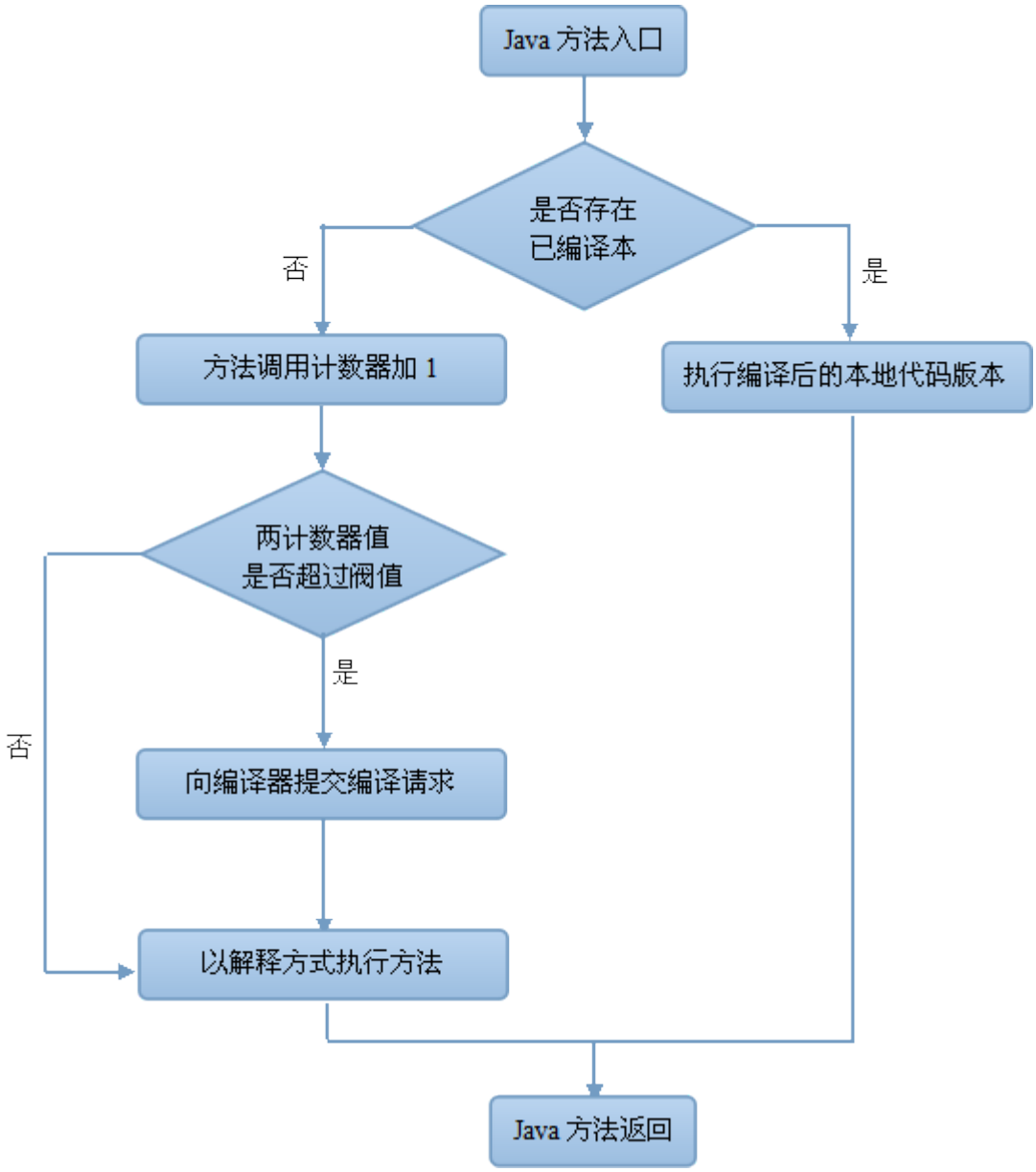
在 HotSpot 虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器和回边计数器。

方法调用计数器用来统计方法调用的次数，在默认设置下，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间内方法被调用的次数。

回边计数器用于统计一个方法中循环体代码执行的次数（准确地说，应该是回边的次数，因为并非所有的循环都是回边），在字节码中遇到控制流向后跳转的指令就称为“回边”。

在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器的值超过了阈值，就会触发JIT编译。触发了 JIT 编译后，在默认设置下，执行引擎并不会同步等待编译请求完成，而是继续进入解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成为止（编译工作在后台线程中进行）。当编译工作完成后，下一次调用该方法或代码时，就会使用已编译的版本。

由于方法计数器触发即时编译的过程与回边计数器触发即时编译的过程类似，因此这里仅给出方法调用计数器触发即时编译的流程：



javac 字节码编译器与虚拟机内的 JIT 编译器的执行过程合起来其实就等同于一个传统的编译器所执行的编译过程。

10

Java 垃圾收集机制

对象引用

Java 中的垃圾回收一般是在 Java 堆中进行，因为堆中几乎存放了 Java 中所有的对象实例。谈到 Java 堆中的垃圾回收，自然要谈到引用。在 JDK1.2 之前，Java 中的引用定义很很纯粹：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。但在 JDK1.2 之后，Java 对引用的概念进行了扩充，将其分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，引用强度依次减弱。

- 强引用：如“Object obj = new Object（）”，这类引用是 Java 程序中最普遍的。只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。
- 软引用：它用来描述一些可能还有用，但并非必须的对象。在系统内存不够用时，这类引用关联的对象将被垃圾收集器回收。JDK1.2 之后提供了 SoftReference 类来实现软引用。
- 弱引用：它也是用来描述非需对象的，但它的强度比软引用更弱些，被弱引用关联的对象只能生存岛下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK1.2 之后，提供了 WeakReference 类来实现弱引用。
- 虚引用：最弱的一种引用关系，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的是希望能在这个对象被收集器回收时收到一个系统通知。JDK1.2 之后提供了 PhantomReference 类来实现虚引用。

垃圾对象的判定

Java 堆中存放着几乎所有的对象实例，垃圾收集器对堆中的对象进行回收前，要先确定这些对象是否还有用，判定对象是否为垃圾对象有如下算法：

引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1，当引用失效时，计数器值就减1，任何时刻计数器都为 0 的对象就是不可能再被使用的。

引用计数算法的实现简单，判定效率也很高，在大部分情况下它都是一个不错的选择，当 Java 语言并没有选择这种算法来进行垃圾回收，主要原因是它很难解决对象之间的相互循环引用问题。

根搜索算法

Java 和 C# 中都是采用根搜索算法来判定对象是否存活的。这种算法的基本思路是通过一系列名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连时，就证明此对象是不可用的。在 Java 语言里，可作为 GC Roots 的兑现包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中的类静态属性引用的对象。
- 方法区中的常量引用的对象。
- 本地方法栈中 JNI（Native 方法）的引用对象。

实际上，在根搜索算法中，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法，或 finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为没有必要执行。如果该对象被判定为有必要执行 finalize()方法，那么这个对象将会被放置在一个名为 F-Queue 队列中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行 finalize()方法。finalize()方法是对象逃脱死亡命运的最后一次机会（因为一个对象的 finalize()方法最多只会被系统自动调用一次），稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果要在 finalize()方法中成功拯救自己，只要在 finalize()方法中让该对象重引用链上的任何一个对象建立关联即可。而如果对象这时还没有关联到任何链上的引用，那它就会被回收掉。

垃圾收集算法

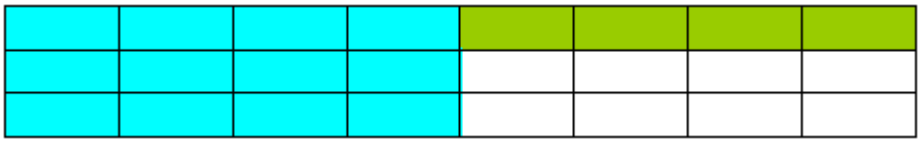
判定除了垃圾对象之后，便可以进行垃圾回收了。下面介绍一些垃圾收集算法，由于垃圾收集算法的实现涉及大量的程序细节，因此这里主要是阐明各算法的实现思想，而不去细论算法的具体实现。

标记—清除算法

标记—清除算法是最基础的收集算法，它分为“标记”和“清除”两个阶段：首先标记出所需回收的对象，在标记完成后统一回收掉所有被标记的对象，它的标记过程其实就是前面的根搜索算法中判定垃圾对象的标记过程。标记—清除算法的执行情况如下图所示：

回收前状态：

回收后状态：

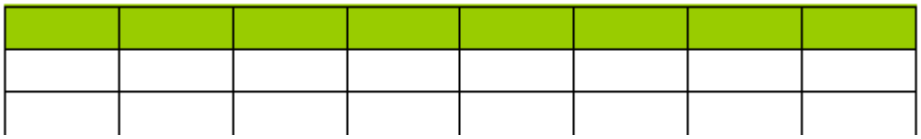


标记—整理算法

复制算法比较适合于新生代，在老年代中，对象存活率比较高，如果执行较多的复制操作，效率将会变低，所以老年代一般会选用其他算法，如标记—整理算法。该算法标记的过程与标记—清除算法中的标记过程一样，但对标记后出的垃圾对象的处理情况有所不同，它不是直接对可回收对象进行清理，而是让所有的对象都向一端移动，然后直接清理掉端边界以外的内存。标记—整理算法的回收情况如下所示：

回收前状态：

回收后状态：



分代收集

当前商业虚拟机的垃圾收集 都采用分代收集，它根据对象的存活周期的不同将内存划分为几块，一般是把 Java 堆分为新生代和老年代。在新生代中，每次垃圾收集时都会发现有大量对象死去，只有少量存活，因此可选用复制算法来完成收集，而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记—清除算法或标记—整理算法来进行回收。

垃圾收集器

垃圾收集器是内存回收算法的具体实现，Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别。Sun HotSpot 虚拟机 1.6 版包含了如下收集器：Serial、ParNew、Parallel Scavenge、CMS、Serial Old、Parallel Old。这些收集器以不同的组合形式配合工作来完成不同分代区的垃圾收集工作。

垃圾回收分析

在用代码分析之前，我们对内存的分配策略明确以下三点：

- 对象优先在 Eden 分配。
- 大对象直接进入老年代。
- 长期存活的对象将进入老年代。

对垃圾回收策略说明以下两点：

- 新生代 GC（Minor GC）：发生在新生代的垃圾收集动作，因为 Java 对象大多都具有朝生夕灭的特性，因此Minor GC 非常频繁，一般回收速度也比较快。
- 老年代 GC（Major GC/Full GC）：发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次 Minor GC。由于老年代中的对象生命周期比较长，因此 Major GC 并不频繁，一般都是等待老年代满了后才进行 Full GC，而且其速度一般会比 Minor GC 慢 10 倍以上。另外，如果分配了 Direct Memory，在老年代中进行 Full GC时，会顺便清理掉 Direct Memory 中的废弃对象。

下面我们来看如下代码：

```
public class SlotGc{
   public static void main(String[] args){
      byte[] holder = new byte[32*1024*1024];
      System.gc();
   }
}
```

代码很简单，就是向内存中填充了 32MB 的数据，然后通过虚拟机进行垃圾收集。在 javac 编译后，我们执行如下指令：java -verbose:gc SlotGc 来查看垃圾收集的结果，得到如下输出信息：

```
[GC 208K->134K(5056K), 0.0017306 secs]

[Full GC 134K->134K(5056K), 0.0121194 secs]

[Full GC 32902K->32902K(37828K), 0.0094149 sec
```

注意第三行，“->”之前的数据表示垃圾回收前堆中存活对象所占用的内存大小，“->”之后的数据表示垃圾回收堆中存活对象所占用的内存大小，括号中的数据表示堆内存的总容量，0.0094149 sec 表示垃圾回收所用的时间。

从结果中可以看出，System.gc()运行后并没有回收掉这 32MB 的内存，这应该是意料之中的结果，因为变量holder 还处在作用域内，虚拟机自然不会回收掉 holder 引用的对象所占用的内存。

我们把代码修改如下：

```
public class SlotGc{
    public static void main(String[] args){
        {
        byte[] holder = new byte[32*1024*1024];
        }
        System.gc();
    }
}
```

加入花括号后，holder 的作用域被限制在了花括号之内，因此，在执行System.gc()时，holder 引用已经不能再被访问，逻辑上来讲，这次应该会回收掉 holder 引用的对象所占的内存。但查看垃圾回收情况时，输出信息如下：

```
[GC 208K->134K(5056K), 0.0017100 secs]

[Full GC 134K->134K(5056K), 0.0125887 secs]

[Full GC 32902K->32902K(37828K), 0.0089226 secs]
```

很明显，这 32MB 的数据并没有被回收。下面我们再做如下修改：

```
public class SlotGc{
    public static void main(String[] args){
        {
        byte[] holder = new byte[32*1024*1024];
        holder = null;
        }
        System.gc();
    }
}
```

这次得到的垃圾回收信息如下：

```
[GC 208K->134K(5056K), 0.0017194 secs]

[Full GC 134K->134K(5056K), 0.0124656 secs]

[Full GC 32902K->134K(37828K), 0.0091637 secs]
```

说明这次 holder 引用的对象所占的内存被回收了。我们慢慢来分析。

首先明确一点：holder 能否被回收的根本原因是局部变量表中的 Slot 是否还存有关于 holder 数组对象的引用。

在第一次修改中，虽然在 holder 作用域之外进行回收，但是在此之后，没有对局部变量表的读写操作，holder 所占用的 Slot 还没有被其他变量所复用（回忆 Java 内存区域与内存溢出一文中关于 Slot 的讲解），所以作为

GC Roots 一部分的局部变量表仍保持者对它的关联。这种关联没有被及时打断，因此 GC 收集器不会将 holder 引用的对象内存回收掉。 在第二次修改中，在 GC 收集器工作前，手动将 holder 设置为 null 值，就把 holder 所占用的局部变量表中的 Slot 清空了，因此，这次 GC 收集器工作时将 holder 之前引用的对象内存回收掉了。

当然，我们也可以用其他方法来将 holder 引用的对象内存回收掉，只要复用 holder 所占用的 slot 即可，比如在 holder 作用域之外执行一次读写操作。

为对象赋 null 值并不是控制变量回收的最好方法，以恰当的变量作用域来控制变量回收时间才是最优雅的解决办法。另外，赋 null 值的操作在经过虚拟机 JIT 编译器优化后会被消除掉，经过 JIT 编译后，System.gc()执行时就可以正确地回收掉内存，而无需赋 null 值。

性能调优

Java 虚拟机的内存管理与垃圾收集是虚拟机结构体系中最重要的组成部分，对程序（尤其服务器端）的性能和稳定性有着非常重要的影响。性能调优需要具体情况具体分析，而且实际分析时可能需要考虑的方面很多，这里仅就一些简单常用的情况作简要介绍。

- 我们可以通过给 Java 虚拟机分配超大堆（前提是物理机的内存足够大）来提升服务器的响应速度，但分配超大堆的前提是有把握把应用程序的 Full GC 频率控制得足够低，因为一次 Full GC 的时间造成比较长时间的停顿。控制 Full GC 频率的关键是保证应用中绝大多数对象的生存周期不应太长，尤其不能产生批量的、生命周期长的大对象，这样才能保证老年代的稳定。
- Direct Memory 在堆内存外分配，而且二者均受限于物理机内存，且成负相关关系，因此分配超大堆时，如果用到了 NIO 机制分配使用了很多的 Direct Memory，则有可能导致 Direct Memory 的 OutOfMemory Error 异常，这时可以通过 -XX:MaxDirectMemorySize 参数调整 Direct Memory 的大小。
- 除了 Java 堆和永久代以及直接内存外，还要注意下面这些区域也会占用较多的内存，这些内存的总和会受到操作系统进程最大内存的限制：

1、线程堆栈：可通过 -Xss 调整大小，内存不足时抛出 StackOverflowError（纵向无法分配，即无法分配新的栈帧）或 OutOfMemoryError（横向无法分配，即无法建立新的线程）。

2、Socket 缓冲区：每个 Socket 连接都有 Receive 和 Send 两个缓冲区，分别占用大约 37KB 和 25KB 的内存。如果无法分配，可能会抛出 IOException：Too many open files 异常。关于 Socket 缓冲区的详细介绍参见我的 Java 网络编程系列中深入剖析 Socket 的几篇文章。

3、JNI 代码：如果代码中使用了JNI调用本地库，那本地库使用的内存也不在堆中。

4、虚拟机和 GC：虚拟机和 GC 的代码执行也要消耗一定的内存。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java-vm/