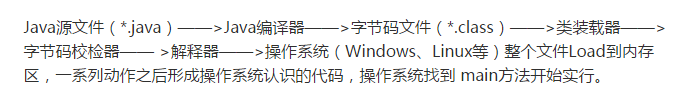
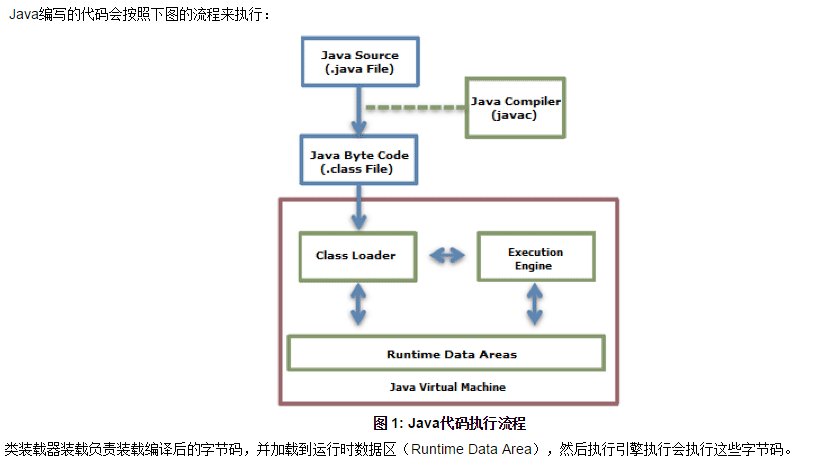
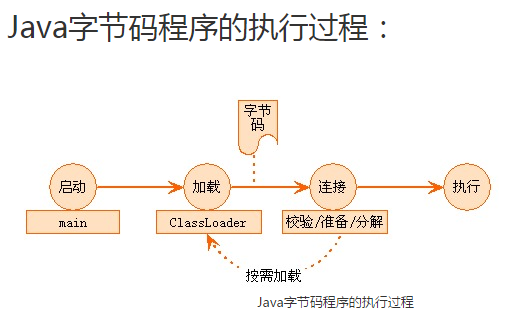
**JVM类的编译、加载、和执行机制及GC机制**

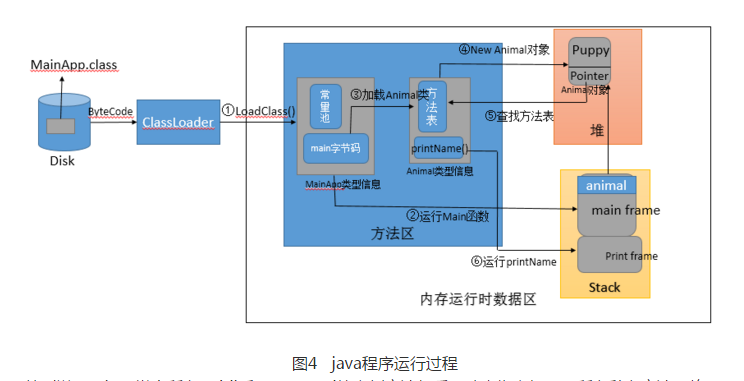
Java类执行总过程：

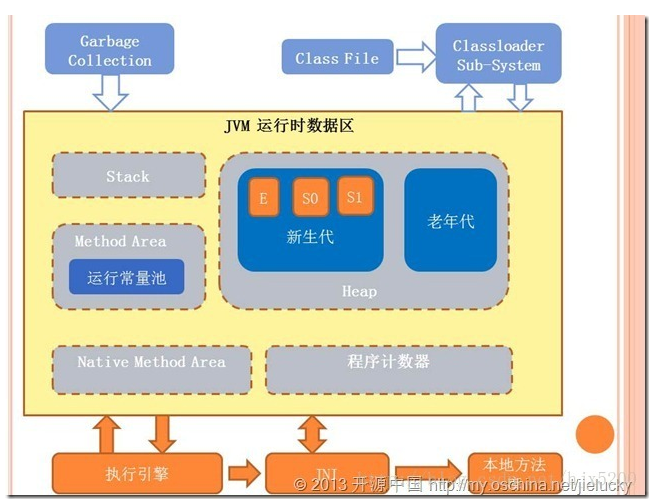
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 1 | Java源码文件（.java文件） |  |  |
| 2 | Java源码级编译器 |  |  |
| 3 | Java二进制字节码文件（.class文件） |  |  |
|  |  |  |  |
|  | 源于本地或者网络的字节码 |  |  |
|  |  |  |  |
| 4 | 加载（loading） | 加载到JVM方法区 |  |
| 5 | 连接（linking）：验证（verifying） |  |  |
| 6 | 连接（linking）：准备（preparing） |  |  |
| 7 | 连接（linking）：解析（resolving） |  |  |
| 8 | 初始化（initializing） |  |  |
|  |  |  |  |
|  |  |  |  |
| 7 | 使用（using）：执行引擎 |  |  |
|  |  | Java字节码解释器 或 |  |
|  |  | JIT（Just In Time）编译器 |  |
|  |  | 操作系统 |  |
|  |  | 硬件 |  |
| 8 | 卸载（unloading） |  |  |
| 9 |  |  |  |

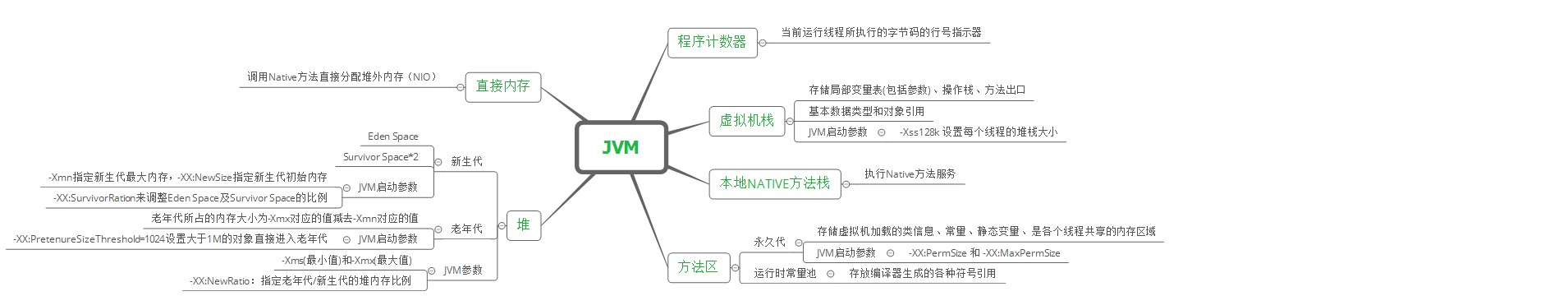












<http://static.haonanji.cn/wp-content/uploads/2016/04/JVM%E5%86%85%E5%AD%98%E6%A8%A1%E5%9E%8B%E5%92%8C%E5%90%AF%E5%8A%A8%E5%8F%82%E6%95%B0%E7%9A%84%E5%85%B3%E7%B3%BB.jpg>

JVM理解主要是两大方面内容：

Java类的编译、加载和执行。JVM的内存管理和垃圾回收机制。

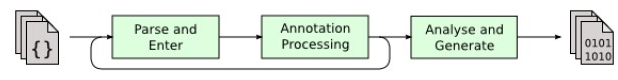
1. Java类的编译过程

由\*.java源码文件转为 \*.class二进制字节码文件的过程。

我们编写好的源代码，就是\*.java文件。使用“javac test.java”就可以编译test.java文件。

编译过程主要有三步：

1. 分析和输入到符号表
2. 注解处理
3. 语义分析和生成字节码



详细过程：

源代码文件\*.java->语法分析器->tokens流->语法分析器->语法树/抽象语法树->语义分析器->注解抽象语法树->字节码生成器->JVM字节码文件\*.class

最后剩成的JVM字节码文件，使用命令“javap -c test”可以查看test.class的字节码信息，主要包含三项内容：

1. 结构信息：class文件相关信息。
2. 元数据：Java源码中的声明和常量信息。
3. 方法信息：Java源码语句和表达式对应的字节码。
4. 类加载机制

类加载器其实也是Java类。有四大类：

1. 根加载器Bootstrap Class Loader
2. 扩展加载器Extension Class Loader
3. 系统应用加载器APP Class Loader
4. 用户自定义加载器Customer Class Loader

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | 中文名称 | 英文 | 语言 | 默认加载路径 | 配置加载路径 |
| 1 | 根加载器 | Bootstrap Class Loader | C++ | JAVA\_HOME/JRE/lib/rt.jar | -Xbootclasspath指定的目录 |
| 2 | 扩展加载器 | Extension Class Loader | Java | JAVA\_HOME/JRE/lib/ext/\*.jar | -Djava.ext.dirs指定的目录 |
| 3 | 系统应用加载器 | APP Class Loader | Java | 系统classpath | -Djava.class.path指定的目录 |
| 4 | 用户自定义加载器 | Customer Class Loader | Java | 自定义目录 | 自定义目录 |
|  |  |  |  |  |  |

Java程序在执行前先要检查类是否已经被加载。

检查过程：从下层往上层依次检查各个加载器已经加载的类，顺序是系统应用类加载器、扩展加载器、根加载器，一旦发现被某个加载器加载过，则马上使用该类。如果一直找到最顶层的根加载器，发现类还没有被加载进JVM运行数据区的方法区，则接下来就要加载该类。

类加载过程：加载过程和检查过程顺序相反，从上层往下层的顺序进行加载。从加载器检查自己的加载路径，找要加载的类，一旦找到类就进行加载。

注意：对每个加载器，最多只能加载一次系统绝对路径下的同一个类。对类而言，可以被不同加载器重复加载，只要你把类放到类加载器的加载路径下，就可以被那个加载器加载。

1. 类执行机制

类在被加载之后，接下来进行连接、初始化，然后才是使用，最后卸载。其中连接包括三个部分：

1. 验证verifying：验证类符合Java规范和JVM规范，和编译阶段的语法语义分析不同。
2. 准备preparing：为类的静态变量分配内存，初始化为系统的初始值。（不初始化静态代码块）。对于final static修饰的变量，直接赋值为用户的定义值。
3. 解析resolving：将符号引用（字面量描述）转为直接引用（对象和实例的地址指针、实例变量和方法的偏移量）

初始化：初始化类的静态变量和静态代码块为用户自定义的值。非静态类在实例化类，在Java堆中创建对象的时候，才会进行初始化。初始化的顺序，和Java源码的从上到下顺序一致。注意：什么时候触发初始化？在类被Java程序“第一次主动使用”的时候，才会触发初始化操作（如果还没有加载，则会顺势触发类的加载过程）。

|  |  |
| --- | --- |
| 类的主动使用（一定会发生类的初始化） | 类的被动使用（不会发生类的初始化） |
| main方法所在的类 |  |
| new一个类的对象 | 数组定义类的引用，不会触发初始化 |
| 调用类的静态成员和静态方法 | 引用常量不会触发，已加入方法区的常量池 |
| 修改类的静态成员的值 | 访问静态域，只有定义该域的类才被初始化 |
| 初始化子类时，父类若没被初始化，则先初始化父类，然后才到子类（接口则不适用） | 通过子类调用父类的静态变量，子类不会初始化，父类会初始化 |
| 使用Java.lang.reflect的包对类进行反射调用 |  |

启动JVM后，操作系统就给JVM分配了内存空间，JVM自己由把得到的内存分为几块：

|  |  |  |  |
| --- | --- | --- | --- |
| ID | 内存块 | 所属关系 | 内存块说明 |
| 1 | Java堆 | JVM实例的所有线程共享 | 存放程序创建的对象或实例。这里是垃圾回收的地方。 |
| 2 | 方法区 | JVM实例的所有线程共享 | Class被加载到的内存区域。运行常量池存放所有类和接口的常量 |
| 3 | Java栈 | 单线程独自拥有 | 由栈帧组成，栈帧又由局部变量表、操作数栈和常量池引用组成 |
| 4 | PC寄存器 | 单线程独自拥有 | 存储下一条要执行的字节码指令的地址（在方法内的偏移量） |
| 5 | 本地方法栈 | 单线程独自拥有 | 提供本地方法接口（JNI，Java Native Interface），供程序调用本地方法库 |
|  |  |  |  |

JVM是基于栈结构的体系结构来执行class字节码的，不同于windows和Linux基于寄存器结构。类的执行机制，主要是在Java栈上面完成。当一个线程被创建后，Java栈和PC寄存器就会被创建。Java栈由栈帧组成，调用一个方法，就会生成一个栈帧（可以理解为表示调用一个方法）。栈帧又由局部变量表、操作数栈和常量池引用组成。

|  |  |  |
| --- | --- | --- |
| ID | 单元 | 说明 |
| 1 | 局部变量表（局部变量区） | 本质上是数组。下标从0开始存放：方法所在的类的引用、传递给调用方法的参数、方法内部的局部变量。如果是静态方法，类的引用就没有，第0个就是传给方法的参数了。注意方法参数和局部变量，如果是long和double类型，则会占据两个槽位的空间，其他的都是一个槽位。 |
| 2 | 操作数栈 | 也是栈结构，方法执行的真正地方，存放执行过程当中的临时中间数据。 |
| 3 | 常量池引用 | 通过这个引用，可以访问方法区的常量池数据。（字面量常量池、数值常量池，整形是-128~127） |
|  |  |  |

执行的时候，每个线程都有一个Java栈，当前执行的栈称为当前栈。一个Java栈调用多个方法，则会push很多个栈帧，当前活动的栈帧称为当前栈帧。当前栈帧执行完毕之后，会把执行结果（如果有）压入到调用它的那个栈帧的操作数栈中，作为上一个栈帧的一个中间处理结果被调用，然后就会被pop出去。当所有调用的方法执行结束后，栈帧也就都pop掉没有了。

例如：执行代码

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| int x=10;  int y=4;  int c = x - y; | | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  | |  |  | | |  | |  | | |  | |  | | |  | |  | | |  |
|  |  | 开始前 | |  | iload\_0 | | |  | | iload\_1 | | |  | | 相减 | | |  | | 最后 | | |  |
| 局部变量表（数组下标） |  | 0 | 10 |  | 0 | 10 |  | | 0 | | 10 |  | | 0 | | 10 |  | | 0 | | 10 |  | |
|  | 1 | 4 |  | 1 | 4 |  | | 1 | | 4 |  | | 1 | | 4 |  | | 1 | | 4 |  | |
|  | 2 |  |  | 2 |  |  | | 2 | |  |  | | 2 | |  |  | | 2 | | 6 |  | |
|  |  |  |  |  |  |  |  | |  | |  |  | |  | |  |  | |  | |  |  | |
| 操作数栈（栈的位置） |  | 1 |  |  | 1 |  |  | | 1 | | 4 |  | | 1 | |  |  | | 1 | |  |  | |
| 栈顶 | 0 |  |  | 0 | 10 |  | | 0 | | 10 |  | | 0 | | 6 |  | | 0 | |  |  | |
|  |  |  |  |  |  |  |  | |  | |  |  | |  | |  |  | |  | |  |  | |

类具体的执行过程，由执行引擎Execute Engine来完成。执行引擎把字节码转为机器码，然后操作系统才可以真正调用，在硬件环境上执行代码。执行引擎的通过Java字节码解释器（一行一行解释字节码）和JIT（Just In Time）即时编译器（对热代码整段编译）来完成机器码的翻译工作。JIT编译器的工作流程为：

JVM字节码->机器无关优化->中间代码->机器相关优化->中间代码->寄存器分配器->中间代码->目标机器码生成器->目标机器码