## 1 HotSpot VM的历史

SUN/Oracle JDK 中使用的 JVM 是 HotSpot VM.

SUN JDK 从 1.3.1 版本开始采用 HotSpot 虚拟机, 并于 2006 年底开源, 主要使用C++实现, JNI 接口部分用C实现。

HotSpot是较新的JVM，用来替代JIT (Just in Time,即时编译)，可以大大提高Java的运行性能，即:

Java起初是将源代码编译为class字节码在虚拟机上执行，速度较慢；

HotSpot将常用的部分代码编译为本地(native)代码，显著提高了性能。

HotSpot JVM参数分为标准参数(standard options)、非标准参数(non-standard options)以及非稳定参数，可参考: GC学习入门看这一篇就够了(基于Oracle JDK 8)

## 2 HotSpot VM 概述

HotSpot 包括一个解释器和两个编译器(client 和 server, 实际运行中选择其一即可, 多选 server), 解释与编译混合执行模式, 默认启动解释执行。

server 模式下应用程序启动较慢, 占用内存多, 但执行效率高, 其适用于服务器端需要长期执行的应用;

client 模式下应用程序启动较快, 占用内存小, 但执行效率较低, 默认情况下不进行动态编译, 适用于桌面应用程序.

查看 JVM 的启动模式:

# 使用解释与编译混合的模式

java -version

java version "1.8.0\_162"

Java(TM) SE Runtime Environment (build 1.8.0\_162-b12)

Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12, mixed mode)

# mixed mode 解释与编译混合的模式, 是默认使用的模式.

1

2

3

4

5

6

# 使用纯解释模式

java -Xint -version

java version "1.8.0\_162"

Java(TM) SE Runtime Environment (build 1.8.0\_162-b12)

# interpreted mode 纯解释模式, 即禁用JIT编译.

1

2

3

4

5

# 使用纯编译模式

java -Xcomp -version

java version "1.8.0\_162"

Java(TM) SE Runtime Environment (build 1.8.0\_162-b12)

Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12, compiled mode)

# compiled mode 纯编译模式, 如果方法无法编译, 则回退到解释模式执行无法编译的方法.

1

2

3

4

5

6

### 2.1 编译器

Java 源代码被编译器编译成 class 字节码文件, 字节码在运行时可以被动态编译(JIT)成本地代码, 前提是解释与编译混合执行模式且虚拟机不是刚启动.

### 2.2 解释器

解释器用来解释 class 字节码文件, Java是解释型语言(区别于编译型语言).

### 2.3 解释型语言 VS 编译型语言

计算机不能直接理解执行高级语言, 只能直接执行机器语言, 所以必须将高级语言翻译成机器语言再执行.

翻译的方式有两种: 编译和解释, 两者只是翻译的时间不同:

① 解释型语言: 不需要提前翻译 —— 例如 Java 语言, 在运行时才翻译, 再由专门的解释器解释执行 —— 每执行一次就要翻译一次, 效率较低.

优点: 可移植性好, 只要有解释环境, 就可以在不同的操作系统上运行. 比如在解释执行时可以动态改变变量的类型、修改程序以及在程序中插入良好的调试诊断等信息, 将解释器移植到不同的系统上，则程序不用改动就可以在移植了解释器的系统上运行。

缺点: 运行需要解释环境, 运行速度较慢, 占用的资源更多 —— 因为不仅要为用户程序分配空间, 解释器本身也要占用一定的系统资源. 其封装了底层代码, 程序严重依赖平台, 不能同 C++ 那样直接操作底层.

代表: 一些网页脚本、服务器脚本等对速度要求不高, 却对不同系统平台间的兼容性有要求的程序通常使用解释性语言, 如 Java、JavaScript、Perl、Python、Ruby、MATLAB 等.

② 编译型语言: 编写的程序在运行前, 需要通过编译器将代码编译成机器语言(二进制文件), 然后直接由相应的操作系统执行.

编译和执行是分开的, 但不能跨平台. 例如 C++ 编译成的 exe 文件, 运行时不需再次编译, 直接使用编译的结果即可 —— 效率较高.

因为不同的操作系统识别的二进制文件是不同的, 所以编译型语言的程序在移植后, 需要重新编译(如 Windows 下的 C 程序编译成 exe 文件, Linux 下要编译成 erp 文件).

优点: 运行速度快, 代码效率高, 同等条件下对系统要求较低.

缺点: 代码需要经过编译才能运行, 可移植性差 —— 只能在兼容的操作系统上运行.

代表: 像开发操作系统、大型应用程序、数据库系统等时都使用编译型语言, 如 C、C++、Pascal 等.

## 3 动态编译

### 3.1 概述

动态编译 (compile during run-time), 英文称 Dynamic compilation, Just In Time (即时编译) 也有此意.

HotSpot VM对字节码的编译不是在程序运行前, 而是在程序运行过程中编译的.

HotSpot VM 中有一个监视器 (Profile Monitor), 用来监视程序的运行状况.

Java字节码 (class文件) 是以解释的方式被加载到虚拟机中 (默认启动时就解释执行), 程序运行过程中, 使用频率高、对程序的性能影响重要的代码, 称之为热点 (hotspot), HotSpot VM 会将这些热点动态地编译成机器码 (native code), 同时对机器码进行优化, 从而提高运行效率. 对使用频率较低的代码, HotSpot VM 就不会编译.

比如指令的重排序, 使用 volatile 关键字可以禁止重排序等都是在这个编译阶段进行的.

### 3.2 HotSpot VM 对字节码的处理有三种

① 不编译: 字节码加载到虚拟机中时的状态, 也就是虚拟机执行时再编译;

② 编译: 把字节码编译成本地代码, 虚拟机执行时已经编译好了, 无需再次编译;

③ 编译并优化: 不但把字节码编译成本地代码, 而且还进行了优化.

具体如何处理, 都是由监视器 (Profile Monitor) 决定的.

### 3.3 为什么不静态编译

为什么不采取将字节码直接变异成本地代码, 然后再装载到虚拟机中?

① 动态编译器在许多方面比静态编译器优越 —— 静态编译器通常很难预知程序运行过程中最需要优化的代码.

② 函数调用是很浪费系统时间的, 因为有许多进出栈操作. 因此有一种优化办法, 就是把原来的函数调用通过编译, 改为非函数调用 —— 将函数代码直接嵌到调用的地方, 变为顺序执行.

③ 面向对象的语言支持多态, 静态编译无法确定程序调用的到底是超类的哪些实现子类的方法, 因为多态是在程序运行中才能具体确定的.