Java高级工程师

JVM(一)

大厂为何如此看重JVM的知识

作为java程序员来说,不了解jvm不可能写出优质高效的程序

大厂为何如此看重JVM的知识

顶级的调优策略依赖于对jvm的深度定制

大厂为何如此看重JVM的知识

高并发、高可用的解决方案依托于jvm这个最基础的背景

之前通过java技术栈学习将大家认知提升到架构思想

从现在开始,我们理解一个技术产品就要用架构的思想来入门

JVM作为一个成熟的产品同样需要从架构入手进行分析

之前通过java技术栈学习将大家认知提升到架构思想

从现在开始,我们理解一个技术产品就要用架构的思想来入门

面试题

- 你知道jvm和jre之间有什么关系吗?
- 频度:高
- 难度:低
- 通过率:低

答案

回答jvm相关的问题,一定要简明扼要,迅速结束问题

Note:底层问题是个无底洞,不断下钻下去一定会“触礁”

JVM的架构解析

- 必会知识

- JRE(Java Runtime Environment):执行字节码

- 事实上, JRE是JVM(Java Virtual Machine)解析字节码,编译代码,执行的实现

什么是JVM

- 一个运行在物理机上软件实现的虚拟机.
- JAVA就是基于VM之上

面试题

- Java宣称的一次编译,到处运行是怎样实现的?
- 频度:中
- 难度:低
- 通过率:高

答案

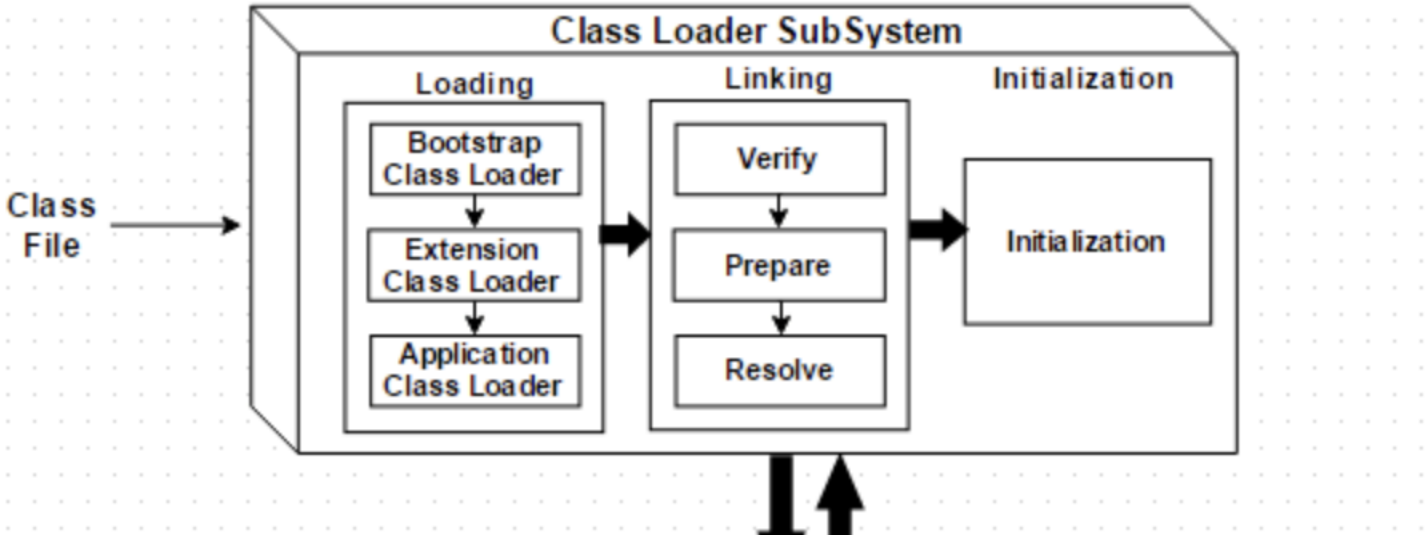
- 编译器将.java文件编译成java.class文件

- .class文件被JVM加载并执行

- Note:谁来执行?

- JRE

JVM的架构图(一)


Class Loader SubSystem
Loading
Linking
Initialization
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Class File

Class loader 解析

加载.class 文件入jvm

Class loader Sub system

一、Loading

二、Linking

三、Initialization

面试题

- JVM classloader 的主要过程/
- JVM实现类加载的主要模块/组件是些什么?

- 难度:低

- 频度:高

- 通过率:低

答案

- 三个过程
-
-
- Loading——>Linking——>Initializaton
- 加载 →连接→ 初始化

.Boot Strap class Loader (引导类加载器)

负责加载bootstrap classpath 的类

.Boot Strap class Loader (引导类加载器)

最高级别的类加载权利

. Extension ClassLoader (扩展类加载器)

负载加载 ext 文件夹下(jre\lib)下的包

Application class Loader (应用类加载器)

按应用级别的类路径,路径涉及的环境变量加载应用类

学习要求

一般来说,知道有哪些类加载器
各自承担哪些功能即可

学习要求

但是真正的架构师岗位还需要进一步了解类加载相关的算法和逻辑

欲进一步了解,请大家捧场中阶以上的课程

Linking(连接器)

- 这里重点掌握三个名词:

  验证

  准备

  解析

验证

确保class文件的字节码是否满足规范,如果验证失败直接返回验证错误

准备

为所有的静态变量分配内存,并赋**默认值**

解析

常量池内的符号引用替换为直接引用

符号引用(Symbolic References)

- 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的定位到目标即可。

- 在.class文件中,用来标示一个具体引用目标(值、对象)的标识符就是符号引用.

字面量（literal）

用于表达源代码中一个固定值的表示法（notation）

整数、浮点数以及字符串

直接引用(Direct References)

- 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。直接引用是和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，那引用的目标必定已经在内存中存在

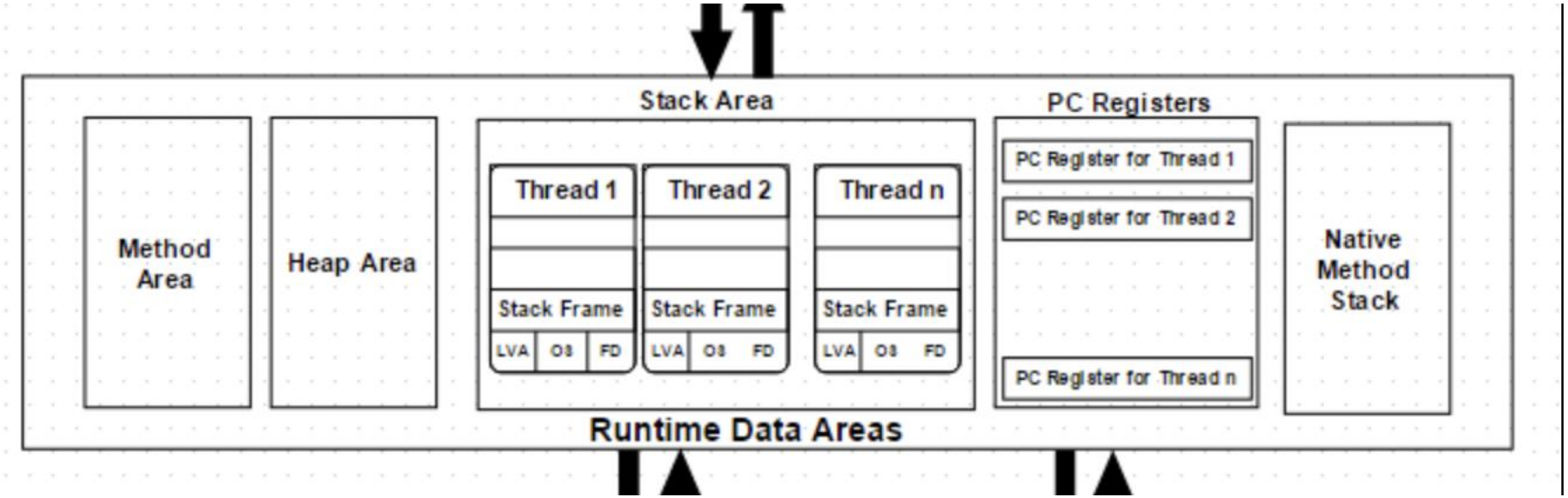
几个名词解释

- 指针: 地址,如0x0f79af
- 偏移量: 指针(指向一个类)+0x4f(偏移到一个属性)
- 句柄: 在不同年代的程序语言有不同的定义:
  面向对象时代,可以认为是对指针的一个封装

初始化

类加载的最后一个阶段,所有的静态变量分配原始值,静态代码块会被执行

内存结构


Stack Area
PC Registers
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas

五个主要模块

- 方法区
- 堆区
- 栈区
- PC寄数器
- 本地方法栈

方法区

所有类级别的数据被储存在这里,包括静态变量.每个JVM只有一个方法区,它是共享资源.

堆区

所有对象和它们对应的实例变量数组都被存储在这.每个JVM堆区只有一个.因为方法区和堆区在内存里可以被多个线程共享,所以存储数据不是线程安全的.

栈区

每个线程都会创建一个独立运行的栈区.对于每个方法的调用,会在栈内存中创建一个栈帧.所有的本地变量都会在栈内存中创建.栈区是安全的,因为它不是共享的,栈帧被分成3个子实体.

栈区

- 局部变量数组 相关方法的局部变量和对相应的值都被储存在这
- 操作数栈 如果需要执行任何中间操作，则操作数堆栈充当运行时工作空间来执行操作
- 帧数据 方法的所有符号对应存储在这里,有异常时,异常捕获信息也存在这

PC计数器

每个线程有独立的PC计数器,保持当前指令的执行地址.一旦指令被执行就被更新为下一个指令.

PC计数器

注意,要区别于C语言和汇编语言的寄存器

本地方法栈

本地方法栈保存了本地方法的信息.每个线程都有独立的方法栈被创建.

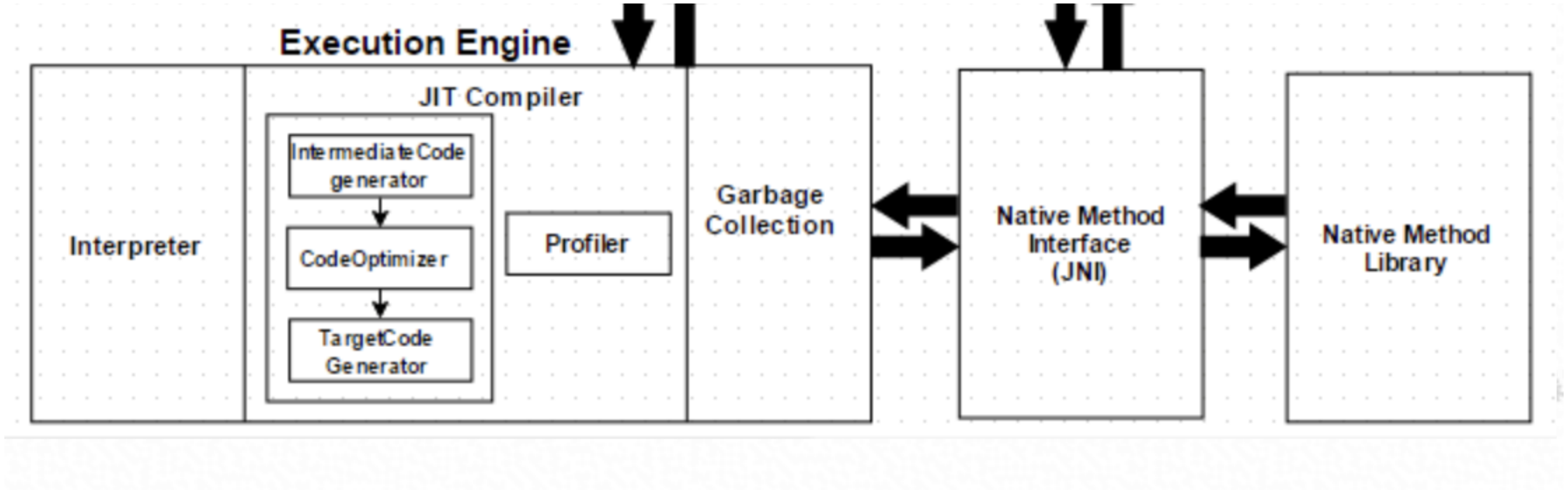
本地方法栈

所谓的本地方法,指的是jvm的原生方法

本地方法栈

用JNI实现的程序,使用的就是这个栈

执行引擎


Execution Engine
Interpreter
JIT Compiler
Intermediate Code generator
CodeOptimizer
TargetCode Generator
Profiler
Garbage Collection
Native Method Interface (JNI)
Native Method Library

执行引擎

字节码被分配到数据运行区被执行引擎执行.执行引擎一块一块的阅读字节码并执行

执行引擎

- 解析器
- JIT(Just-In-Time) 编译器
- 垃圾回收器

面试题

- 解析器已经可以实现java程序的解析运行了,为什么还需要JTL编译器呢?

- 难度:中
- 频度:中
- 通过率:低

答案

- 解析器的缺点:

- 解析器解析字节码很快,但执行比较慢.

- 当一个方法被多次调用时，每次都需要重复解析

答案

- JIT编译器优点

- 消除了解析器的缺点.执行引擎借助解析器解析字节码,一旦发现有重复的字节码就用JIT编译器,编译字节码并转化成本地代码.

- 本地代码将被直接用在重复的方法调用中,从而提升系统的性能.

JIT的具体运行机制

JIT的具体运行机制

- 中间代码生成器 生成中间代码
- 代码优化器 负责优化上面生成的中间代码
- 目标代码生成器 负责生成机器代码或本地代码
- 探查器 一个特殊的组件,负责寻找热点,例如该方法是否被多次调用

Note

上述的优化其实就是C++编译器的优化策略,注意触类旁通

Java语言技术栈其实正在c++化,这里可以看出一个趋势

更直白的说,java技术栈现在更类似产品化后的c++

本节课程结束

请大家认真复习