# 为什么要了解虚拟机

JVM不单单只支持Java语言，也支持其他语言（Scala、Kotlin、Groovy等等）

区块链2.0--以太坊(比特币是区块链1.0) 中提供了EVM的虚拟机，它的实现和JVM类似，基于栈、生成脚本编译成字节码来执行。知识通用。（理论大于实际）

# 虚拟机历史

了解即可，无需关注

解释执行和编译执行（针对字节码的执行）

解释执行就是边翻译为机器码边执行、即时编译（编译执行）就是先将一个方法中的所有字节码全部编译成机器码之后再执行。

Hotspot采用的是先解释执行，到了一定时机后热点代码（多次执行、循环等）再翻译成机器码

热点代码探测技术（通过执行计数器找到最有编译价值的代码，如果代码用得非常频繁，就会把这些代码编译成本地代码）。

JRockit采取的方法是在执行class时直接编译为机器码（Java程序启动速度会比较慢）

J9和Hotspot比较接近，主要是用在IBM产品（IBM WebSphere和IBM的AIX平台上），华为有的项目用的J9。

谷歌：Google Android Dalivk VM：使用的寄存器架构，执行dex（Dalvik Executable）通过class转化而来。

# 未来的Java技术

**模块化:**OSGI（动态化、模块化），应用层面就是微服务，互联网的发展方向

**混合语言**：多个语言都可以运行在JVM中，google的Kotlin 成为了 Android 的官方语言。Scala(Kafka)

**多核并行**：CPU从高频次转变为多核心，多核时代。JDK1.7引入了Fork/Join，JDK1.8提出lambda表达式(函数式编程天生适合并行运行)

**丰富语法：**JDK5提出自动装箱、泛型(并发编程讲到)、动态注解等语法。JDK7二进制原生支持。try-catch-finally 至try-with-resource

**64位：**虽然同样的程序64位内存消耗比32位要多一点，但是支持内存大，所以虚拟机都会完全过渡到64位，32位的JVM有4G的堆大小限制。

**更强的垃圾回收器（现在主流CMS、G1）：**JDK11 –ZGC（暂停时间不超过10毫秒，且不会随着堆的增加而增加，TB级别的堆回收））：有色指针、加载屏障。JDK12支持并发类卸载，进一步缩短暂停时间 JDK13(计划于2019年9月)将最大堆大小从4TB增加到16TB

# Java SE体系架构

JavaSE，Java平台标准版，为Java EE和Java ME提供了基础。

JDK：Java开发工具包，JDK是JRE的超集，包含JRE中的所有内容，以及开发程序所需的编译器和调试程序等工具。

JRE：Java SE运行时环境 ，提供库、Java虚拟机和其他组件来运行用Java编程语言编写的程序。主要类库，包括：程序部署发布、用户界面工具类、继承库、其他基础库，语言和工具基础库

JVM：java虚拟机，负责JavaSE平台的硬件和操作系统无关性、编译执行代码（字节码）和平台安全性

# 运行时数据区域

这个是抽象概念，内部实现依赖寄存器、高速缓存、主内存（具体要分析JVM源码 C++语言实现，没必要看）

计算机的运行=指令+数据，指令用于执行方法的，数据用于存放数据和对象的。

虚拟机栈----执行java方法、本地方法栈---执行本地方法、程序计数器---程序执行的计数器

Java中的数据：变量、常量、对象、数组相关。

# 程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响（面试可能问到为什么需要）

如果线程正在执行的是一个Java方法，则指明当前线程执行的代字节码行数

如果正在执行的是Natvie方法，这个计数器值则为空（Undefined）

此内存区域是唯一一个不会出现OutOfMemoryError情况的区域。

# 虚拟机栈（JVM后续的执行子程序有详细的见解）

|  |  |
| --- | --- |
| **iload\_1** | 第二个int型局部变量进栈 |
| bipush | 将一个byte型常量值推送至栈顶 |
| **isub** | 栈顶两int型数值相减，并且结果进栈 |
| istore\_1 | 将栈顶int型数值存入第二个局部变量 |
|  |  |

**栈：**

数据结构的特点和java中方法中调用方法的特性一致。（为什么JVM使用栈 –演示代码StackFilo）

虚拟机栈：

**异常：**

线程请求的栈深度大于虚拟机所允许的深度：StackOverflowError

JVM动态扩展时无法申请到足够的内存时：OutOfMemoryError

**虚拟机栈：**

每个线程私有的，线程在运行时，在执行每个方法的时候都会打包成一个栈帧，存储了局部变量表，操作数栈，动态链接，方法出口等信息，然后放入栈。每个时刻正在执行的当前方法就是虚拟机栈顶的栈桢。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程。

栈的大小缺省为1M，可用参数 –Xss调整大小，例如-Xss256k

在编译程序代码的时候，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，并且写入到方法表的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。

**局部变量表:**顾名思义就是局部变量的表，用于存放我们的局部变量的。首先它是一个32位的长度，主要存放我们的Java的八大基础数据类型，一般32位就可以存放下，如果是64位的就使用高低位占用两个也可以存放下，如果是局部的一些对象，比如我们的Object对象，我们只需要存放它的一个引用地址即可。（基本数据类型、对象引用、returnAddress类型）

**操作数据栈**：存放我们方法执行的操作数的，它就是一个栈，先进后出的栈结构，操作数栈，就是用来操作的，操作的的元素可以是任意的java数据类型，所以我们知道一个方法刚刚开始的时候，这个方法的操作数栈就是空的，操作数栈运行方法是会一直运行入栈/出栈的操作

**动态连接:**Java语言特性多态（需要类加载、运行时才能确定具体的方法，后续有详细的讲解）

**返回地址:**

正常返回（调用程序计数器中的地址作为返回）

三步曲：

恢复上层方法的局部变量表和操作数栈、

把返回值（如果有的话）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、

异常的话（通过异常处理器表<非栈帧中的>来确定）

# 本地方法栈

各虚拟机自由实现，本地方法栈native方法调用 JNI到了底层的C/C++(c/c++可以触发汇编语言，然后驱动硬件)

# 线程共享的区域

类信息：

类的完整有效名、返回值类型、修饰符（public，private...）、变量名、方法名、方法代码、这个类型直接父类的完整有效名(除非这个类型是interface或是 java.lang.Object，两种情况下都没有父类)、类的直接接口的一个有序列表

## 方法区/永久代

用于存储已经被虚拟机加载的类信息，常量("zdy","123"等)，静态变量(static变量)等数据，可用以下参数调整：

jdk1.7及以前：-XX:PermSize；-XX:MaxPermSize；

jdk1.8以后：-XX:MetaspaceSize； -XX:MaxMetaspaceSize

jdk1.8以后大小就只受本机总内存的限制

如：-XX:MaxMetaspaceSize=3M

## 堆

几乎所有对象都分配在这里，也是垃圾回收发生的主要区域，可用以下参数调整：

-Xms：堆的最小值；

-Xmx：堆的最大值；

-Xmn：新生代的大小；

-XX:NewSize；新生代最小值；

-XX:MaxNewSize：新生代最大值；

例如- Xmx256m

## 运行时常量池

### 符号引用（一个概念）

一个java类（假设为People类）被编译成一个class文件时，如果People类引用了Tool类，但是在编译时People类并不知道引用类的实际内存地址，因此只能使用符号引用来代替。

而在类装载器装载People类时，此时可以通过虚拟机获取Tool类的实际内存地址，因此便可以既将符号org.simple.Tool替换为Tool类的实际内存地址，及直接引用地址。

即在编译时用符号引用来代替引用类，在加载时再通过虚拟机获取该引用类的实际地址.

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是无关的，引用的目标不一定已经加载到内存中。

### 字面量

文本字符串 String a = "abc",这个abc就是字面量

八种基本类型int a = 1; 这个1就是字面量

声明为final的常量

### 常量池的变化

# 各版本之间的变化

见课件

# 直接内存

使用Native函数库直接分配堆外内存(NIO)

并不是JVM运行时数据区域的一部分，但是会被频繁使用(可以通过-XX:MaxDirectMemorySize来设置（默认与堆内存最大值一样,也会出现OOM异常)

避免了在Java 堆和Native 堆中来回复制数据，能够提高效率

测试用例JavaStack：设置JVM参数-Xmx100m，运行异常，因为如果没设置-XX:MaxDirectMemorySize，则默认与-Xmx参数值相同，分配128M直接内存超出限制范围

# 站在线程角度来看

虚拟机栈、本地方法栈、程序计数器三个区域的生命周期和线程相同。

线程共享区域：就复杂多了，后续完善

# 深入辨析堆和栈

* **功能**
* 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char等）以及对象的引用变量，其内存分配在栈上，变量出了作用域就会自动释放；
* 而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；
* **线程独享还是共享**
* 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
* 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。
* **空间大小**

栈的内存要远远小于堆内存

## 栈溢出

参数：-Xss256k

java.lang.StackOverflowError 一般的方法调用是很难出现的，如果出现了要考虑是否有无限递归。

虚拟机栈带给我们的启示：方法的执行因为要打包成栈桢，所以天生要比实现同样功能的循环慢，所以树的遍历算法中：递归和非递归(循环来实现)都有存在的意义。递归代码简洁，非递归代码复杂但是速度较快。

OutOfMemoryError：不断建立线程。（一般演示不出，演示出来机器也死了）