执行引擎是 Java 虚拟机最核心的组成部分之一。

1. 栈帧的概念。

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）[1]的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。
每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。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[2]，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。

2. 为当前栈帧的概念。

一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态。对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧（Current StackFrame），与这个栈帧相关联的方法称为当前方法（Current Method）。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。

3. 局部变量表的概念。

局部变量表（Local Variable Table）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。在 Java 程序编译为 Class 文件时，就在方法的 Code 属性的 max_locals 数据项中确定了该方法所需要分配的局部变量表的最大容量。

4. 局部变量的使用。

一个局部变量定义了但没有赋初始值是不能使用的，不要认为 Java 中任何情况下都存在诸如整型变量默认为 0，布尔型变量默认为 false 等这样的默认值。

5. 操作数栈的概念。

操作数栈（Operand Stack）也常称为操作栈，它是一个后入先出（Last In FirstOut,LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候写入到 Code 属性的 max_stacks 数据项中。操作数栈的每一个元素可以是任意的 Java 数据类型，包括 long 和 double。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。在方法执行的任何时候，操作数栈的深度都不会超过在 max_stacks 数据项中设定的最大值。
当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈/入栈操作。

6. 动态连接的概念。

每个栈帧都包含一个指向运行时常量池[1]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（Dynamic Linking）。通过第 6 章的讲解，我们知道 Class 文件的常量池中存有大量的符号引用，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作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就转化为直接引用，这种转化称为静态解析。另外一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。

7. 方法调用的概念。

方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。在程序运行时，进行方法调用是最普遍、最频繁的操作，但前面已经讲过，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（相当于之前说的直接引用）。这个特性给 Java 带来了更强大的动态扩展能力，但也使得 Java 方法调用过程变得相对复杂起来，需要在类加载期间，甚至到运行期间才能确定目标方法的直接引用。

8. 静态分派的概念。

所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。静态分派的典型应用是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。自动转型还能继续发生多次，按照 char- > int- > long- > float- > double 的顺序转型进行匹配。

9. invokevirtual 指令的运行时解析过程。

1）找到操作数栈顶的第一个元素所指向的对象的实际类型，记作 C。
2）如果在类型 C 中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 java.lang.IllegalAccessError 异常。
3）否则，按照继承关系从下往上依次对 C 的各个父类进行第 2 步的搜索和验证过程。
4）如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。
由于 invokevirtual 指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的 invokevirtual 指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是 Java 语言中方法重写的本质。我们把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。今天（直至还未发布的 Java 1.8）的 Java 语言是一门静态多分派、动态单分派的语言。

10. 虚拟机动态分派的实现。

是为类在方法区中建立一个虚方法表（Vritual Method Table，也称为 vtable，与此对应的，在 invokeinterface 执行时也会用到接口方法表——Inteface Method Table，简称 itable），使用虚方法表索引来代替元数据查找以提高性能。
方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。