**第8章 虚拟机字节码执行引擎**

字节码执行引擎是Java虚拟机最核心的组成部分之一。在Java虚拟机规范中制定了虚拟机字节码执行引擎的**概念模型**，这个概念模型成为**各种虚拟机执行引擎的统一外观（Facade）**。

在不同虚拟机实现里面，执行引擎在执行Java代码的时候可能会有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择，也可能两者兼备，甚至还可能会包含几个不同级别的编译器执行引擎。但从外观上看起来，所有的Java虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。本章主要从概念模型的角度来讲解虚拟机的**方法调用**和**字节码执行**。

运行时栈帧结构

**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（VM Stack）的栈元素。每一个栈帧存储了每一个方法的**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**在虚拟机栈里从**入栈到出栈**的过程。

在编译程序代码时，每一个栈帧（方法）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，并写入到对应方法表的Code属性中。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。

一个线程中的方法调用链可能会很长，所以会有很多方法都同时处于执行状态。而对于执行引擎来说，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 Stack Frame），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。概念模型的栈帧结构如下所示：

**活动线程（当前线程）** 线程n 线程n-1

**当前栈帧**

局部变量表

Local Variable Table

**……**

操作数栈

Operand Stack

动态连接

Dynamic Linking

返回地址

Return Address

**……**

**……**

**栈帧2**

**栈帧1**

栈帧的概念结构

局部变量表

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。

局部变量表的容量以变量槽（Variable Slot）为最小单位。一个Slot可以存放一个32位以内的数据类型，Java中占用32位以内的数据类型有boolean、byte、char、short、int、float、reference（表示对一个对象实例的引用，在64位虚拟机中则实际长度为64位。可从此引用中直接或间接地查找到对象在Java堆中的数据存放的起始地址索引，也可从此引用中直接或间接地查找到对象所属数据类型在方法区中的存储的类型信息）和returnAddress（为字节码指令jsr、jsr\_w、ret服务，指向了一条字节码指令的地址）。

对于64位的数据类型，虚拟机会以高位对齐的方式为其分配两个连续的Slot空间。64位的数据类型只有long和double（以及64位虚拟机中的reference）。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围从0开始至局部变量表最大的Slot数量。如果访问的是32位数据类型的变量，索引n就代表使用第n个Slot，如果是64位数据类型的变量，则说明会同时使用n和n+1两个Slot。JVM不允许用户采用任何方式单独访问存放64位数据的两个Slot中的某一个，否则在类加载的校验阶段会抛出异常。

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的。如果执行的是**实例方法**（非static），则**局部变量表中第0位索引**的Slot默认是用于传递**方法所属对象实例的引用**，在方法中可以通过关键字**“this”**来访问到这个隐含的参数。**其余参数则按照参数表顺序排列**，占用从1开始的局部变量Slot，参数值分配完毕后，再根据方法体内部定义的**局部变量**顺序和作用域分配其余的Slot。

为节省栈帧空间，局部变量表中的Slot是可以重用的。方法体中定义的变量，其作用域并不一定会覆盖到整个方法体，如果当前字节码PC计数器的值已经超出了某个变量的作用域，那这个变量对应的Slot就可以交给其他变量使用。

操作数栈

操作数栈（Operand Stack）也称为操作栈。操作数栈的最大深度也在编译时写入到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。

当一个方法刚开始执行的时候，该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，即入栈/出栈操作。

另外，在概念模型中，两个栈帧作为虚拟机栈的元素，是完全相互独立的。但在大多虚拟机实现里都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，无须进行额外的参数复制传递。

Java虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。

操作数栈

----------------------------------其他栈帧信息

----------------------------------局部变量表

**局部变量表共享区域**

**操作数栈共享区域**

操作数栈

----------------------------------其他栈帧信息

----------------------------------局部变量表

两个栈帧之间的数据共享

动态连接

每个栈帧都包含一个指向运行时常量池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）。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**。

方法返回地址

当一个方法开始执行后，只有两种方式可以退出这个方法。

第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者（调用当前方法的方法称为调用者），是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为正常完成出口。

另一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为异常完成出口。一个方法使用异常完成出口的方式退出，是不会给它的上层调用者产生任何返回值的。

无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的PC计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中不会保存这部分信息。

方法退出的过程实际上等于把当前栈帧出栈，因此退出时可能执行的操作有：

（1）恢复上层方法的局部变量表和操作数栈；

（2）把返回值压入调用者栈帧的操作数栈中；

（3）调整PC计数器的值以指向方法调用指令后的下一条指令。

附加信息

取决于具体虚拟机实现，比如与调试相关的信息等等。略。

方法调用

注意，方法调用并不等同于方法执行，方法调用阶段唯一任务就是确定被调用方法的版本（即调用哪一个方法），不涉及方法内部的具体运行过程。另外一切方法调用在Class文件里存储的只有符号引用，而并非方法在实际运行时内存布局中的入口地址（即直接引用）。只有在类加载期间，甚至到运行期才能确定目标方法的直接引用。

解析

所有目标方法在Class文件里面都是一个常量池中的符号引用，在**类加载的解析阶段**，会将其中的**一部分符号引用转化为直接引用**，这种解析能成立的条件是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。也就是说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法称为解析。

Java语言中，符合“编译器可知，运行期不可变”这个要求的方法，只有**静态方法**和**私有方法**两大类。因为前者与类型直接关联，后者在外部不可被访问，因此这两种方法各自的特点决定了它们都无法通过继承或另外的方式重写其他版本，因此它们都适合**在类加载阶段进行解析**。

在Java虚拟机里提供了5条方法调用字节码指令，分别如下：

invokestatic：调用静态方法

invokespecial：调用实例构造器<init>方法、私有方法和父类方法

**invokevirtual：调用所有的虚方法和final方法**

invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。

invokedynamic：先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法，在此之前的4条调用指令，分派逻辑是固化在Java虚拟机内部的，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段中确定唯一的调用版本，符合这个条件的有：静态方法、私有方法、实例构造器、父类方法4类。它们在类加载的时候就会把符号引用解析为该方法的直接引用。其他方法称为虚方法（不包括final方法），final方法也属于非虚方法，因为它无法被覆盖，也没有其他版本，多态选择的结果也是唯一的。

**解析调用**一定是一个静态的过程，在编译期间就完全确定，在类装载的解析阶段就会把涉及的符号引用全部转变为可确定的直接引用，不会留到运行期再完成。而**分派（Dispatch）调用**则可能是静态的也可能是动态的，根据分派依据的宗量数可分为单分配和多分派，因此就有静态单分派、静态多分派、动态单分配、动态多分派四种分配组合情况。

分派

分派调用过程揭示了多态性特征的一些最基本的体现，比如“重载”和“重写”在Java虚拟机之中是如何实现的。

（1）静态分派

Human man = new Man();

其中，Human称为变量的静态类型（Static Type），或者叫做变量的外观类型（Apparent Type），后面的“Man”则称为变量的实际类型（Actual Type）。静态类型和实际类型在程序中都可以发生变化，区别在于静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的**静态类型是在编译期可知**的，而**实际类型变化的结果在运行期才可被真正确定**，因为编译器在编译程序的时候并不知道一个对象的实际类型是什么。

/\*\*

\* 方法静态分派

\*/

public class StaticDispatch {

static abstract class Human {}

static class Man extends Human {}

static class Woman extends Human {}

public void sayHello(Human guy) {

System.out.println("hello, guy!");

}

public void sayHello(Man guy) {

System.out.println("hello, gentleman!");

}

public void sayHello(Woman guy) {

System.out.println("hello, lady!");

}

public static void main(String[] args) {

Human man = new Man();

Human woman = new Woman();

StaticDispatch sr = new StaticDispatch();

sr.sayHello(man);//hello, guy!

sr.sayHello(woman);//hello, guy!

}

}

//实际类型变化（编译期不可知，必须到运行期可得知并真正确定）

Human man = new Man();

man = new Woman();

//静态类型变化（最终的静态类型也必须在编译期已知）

sr.sayHello((Man) man);

sr.sayHello((Woman) man);

对于方法重载，使用哪个重载版本，完全取决于传入参数的数量和数据类型。编译器在重载时是通过参数的静态类型而不是实际类型作为判断依据的。并且静态类型是编译器可知的，因此在编译阶段，Javac编译器会根据参数的静态类型决定使用哪个重载版本。

另外需要注意的是，在很多情况下，编译器确定的重载版本并不是唯一的，而往往只能确定一个当前最为适合的版本。这是由于字面量（如String str = “hello”中的“hello”）不需要定义，所以字面量没有显式的静态类型，它的静态类型只能通过语言上的规则去理解和推断。

比如当前有多个sayHello()的重载方法：

（1）sayHello(char arg)

（2）sayHello(int arg)

（3）sayHello(long arg)

（4）sayHello(float arg)

（5）sayHello(double arg)

（6）sayHello(byte arg)

（7）sayHello(short arg)

（8）sayHello(Character arg)

（9）sayHello(Serializable arg)

（10）sayHello(Object arg)

（11）sayHello(char… arg)

如果传入参数’a’,会先寻找参数类型为char的重载方法，如果没有找到，则会发生一次自动类型转换，代表数字97（字符’a’的Unicode数值为数字97），接下来会寻找参数类型为int的重载方法，如果没有找到，再次发生自动类型转换，转换为长整数97L，然后寻找参数类型为long的重载方法...

按照优先级从

char -> int -> long -> float -> double

->Character

自动装箱，’a’被包装为它的封装类型java.lang.Character，所以寻求匹配参数类型为Character的重载

->Serializable

Serializable是java.lang.Character类实现的一个接口，当自动装箱之后发现还是找不到装箱类，但是找到了装箱类实现了的接口类型，所以紧接着又发生一次自动转型。需要注意，引用数据类型的**转型只能安全地转型为它实现的接口或父类**

或

->Comparable<Character>

这是java.lang.Character类实现的另外一个接口，地位相等，优先级相同。若同时出现编译器会提示类型模糊，拒绝编译。此时程序必须在调用时显式地指定字面量的静态类型，如：sayHello((Serializable) ’a’)

->Object

接下来会转型为父类。如果有多个父类，将在继承关系中从下往上开始搜索，越接近上层的优先级越低。即使方法调用传入的参数值为null时，这个规则仍然适用。

->char…

可变参数类型的重载是优先级别最低的。此时’a’被当做一个数组元素。

的顺序转型匹配。但最终是绝对不会匹配到byte和short类型的重载，因为char到byte或short的转型是不安全的。