# 《深入理解Java虚拟机：JVM高级特性与最佳实践--第二版》学习日志（三）Part 3：虚拟机字节码执行引擎

2018-06-16 Java Java JVM

虚拟机是如何执行字节码的呢？

学习资料主要参考：《深入理解Java虚拟机：JVM高级特性与最佳实践(第二版)》，作者：周志明

## 运行时栈帧结构

**栈帧**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虚拟机栈的栈元素。

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。在编译程序代码时，栈帧需要多大的局部变量表、多深的操作数栈都已经确定。

每一个方法从调用开始到执行完成，都对应着一个栈帧在虚拟机栈里面入栈出栈的过程。

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）。

下面介绍栈帧中各个部分的作用和数据结构。

### 1. 局部变量表

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。

局部变量表的容量以变量槽（Varable Slot）为最小单位。它可以存储一个boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据。其中 reference 表示对一个对象实例的引用。returnAddress类型已经很少见了，它为字节码jsr、jsr_w和ret服务。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从0开始到局部变量表最大的Slot数量。

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，那局部变量表中的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法可以用this来访问。

为了尽可能地节省栈帧空间，局部变量中的Slot是可以重用的，因为方法体中定义的变量都是由作用域的，当字节码PC计数器的值已经超出了某个变量的作用域时，这个变量对于的Slot就可以交给其他变量使用。

### 2. 操作数栈

也常称为操作栈。它是一个后入先出（Last In First Out，LIFO）栈。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。

当一个方法刚刚开始执行的时候，这个方法的操作数栈时空的，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，也就是入栈、出栈操作。

### 3. 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转化为直接引用，这种称为静态解析。另外一

Content

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。

## 4. 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，第一种是执行引擎遇到任意一个方法返回的字节码指令。这种退出方法的方式称为**正常完成出口**。

另一种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处理。这种退出称为**异常完成出口**。

方法的退出过程实际上是当前栈帧的出栈。

## 5. 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧之中。

在实际开发中，一般会把动态连接、方法返回地址和其他附加信息归为一类，称为栈帧信息。

# 方法调用

方法调用并不等同于方法执行，它仅指确定被调用方法的过程。

## 1. 解析

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，这类方法的调用称为解析。

符合“编译期可知、运行期不可变”这个要求的，主要包括静态方法和私有方法。前者与类型直接关联，后者在外部不能被访问。

虚拟机中提供了5条方法调用字节码指令，分别是：

- 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：调用所有虚方法
- invokeinterface：调用接口方法，会在运行期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。

只要是能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段变为直接引用，也就是静态方法、私有方法、实例构造器、父类方法4类。这4类方法也叫做非虚方法。

除了以上4类方法之外，还有一种被final修饰的方法，它虽然是被invokevirtual调用，但它却是一个非虚方法。

## 2. 分派

### 1）静态分派

```
Human man = new Man();
```

参考以上代码，Human被称为变量的静态类型，或者叫外观类型，后面的Man称为变量的实际类型。

对于静态类型的变化，可以在编译器可知，但是对于实际类型的变化，只有在运行期才能确定。

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定依据的。

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**。典型例子就是方法重载。

### 2）动态分派

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。

与重写有这密切关系。

### 3）单分派与多分派

方法的接收者与方法的参数统称为方法的宗量。

根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。

单分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。

Java至今为止的是一门静态多分派、动态单分派的语言。

### 4）虚拟机动态分派的实现

由于动态分派是非常频繁的动作，所以为了提高性能，最常用的方法就是为类在方法区建立一个**虚方法表**。使用虚方法表索引来代替元数据查找以提高性能。

除了虚方法表之外，还有内联缓存和基于“类型继承关系分析”技术的守护内联。

## 3. 动态类型语言支持

### 1）动态类型语言

动态类型语言的关键特征是它的类型检查的主体过程是在运行期而不是编译期。

“变量无类型，而变量值有类型”，这个特点也是动态类型语言的一个重要特征。

静态类型语言的优点是：编译器可以提供严谨的类型检查，这样与类型相关的问题能在编码的时候就及时发现，利于稳定性及代码达到更大规模。

动态类型语言的优点是：可以为开发人员提供更大的灵活性，也可以提升开发效率。

### 2）JDK1.7与动态类型

JDK1.7之前，JVM层面对于动态类型语言的支持不太好，主要表现在方法调用方面。

所以JDK1.7中引入了invokedynamic指令以及java.lang.invoke包。

### 3）java.lang.invoke包

这个包提供了一种新的动态确定目标方法的机制，称为Methodhandle。

MethodHandle和Reflection有很多的相似之处，但是也有很多区别：

- 从本质上来讲，Reflection和 MethodHandle机制都是在模拟方法调用，但Reflection是在模拟 Java 代码层次的方法调用，而 MethodHandle 是在模拟字节码层次的方法调用。
- Reflection 中的 java.lang.reflect.Method对象远比MethodHandle机制中的 java.lang.invoke.Methodhandle 对象所包含的信息多。
- 由于MethodHandle 是对字节码的方法指令调用的模拟，所以理论上虚拟机在这方面做的各种优化，在MethodHandle上也应当可以采用类似思路去支持，而通过反射就不行。

### 4）invokedynamic指令

每一处含有invokedynamic指令的位置都称做“动态调用点”。

### 5）掌控方法分派规则

可以使用它来调用祖父类方法。

```
import static java.lang.invoke.MethodHandles.lookup;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodType;

class Test {

    class GrandFather {
        void thinking() {
            System.out.println("i am grandfather");
        }
    }

    class Father extends GrandFather {
        void thinking() {
            System.out.println("i am father");
        }
    }

    class Son extends Father {
         void thinking() {
            try {
                MethodType mt = MethodType.methodType(void.class);
                MethodHandle mh = lookup().findSpecial(GrandFather.class, "thinking", mt, getClass());
                mh.invoke(this);
            } catch (Throwable e) {
            }
        }
    }

    public static void main(String[] args) {
        (new Test().new Son()).thinking();
    }
}
```

**注意**： JDK1.8环境下MethodHandles.lookup方法是调用者敏感的，所以1.8下的打印结果是“i am father”。

## 基于栈的字节码解释执行引擎

### 1. 解释执行

将代码进行词法分析和语法分析处理，再转化为抽象语法树，然后遍历语法树生成闲心的字节码指令流的过程。

### 2. 基于栈的指令集与基于寄存器的指令集

基于栈的指令集主要的优点是可移植。缺点是会稍慢一些。

## 类加载及执行子系统的案例与实战

有4个例子，关于类加载和字节码的案例各有两个。

### 1. Tomcat：正统的类加载器架构

#### 1）Web服务器需要解决的问题

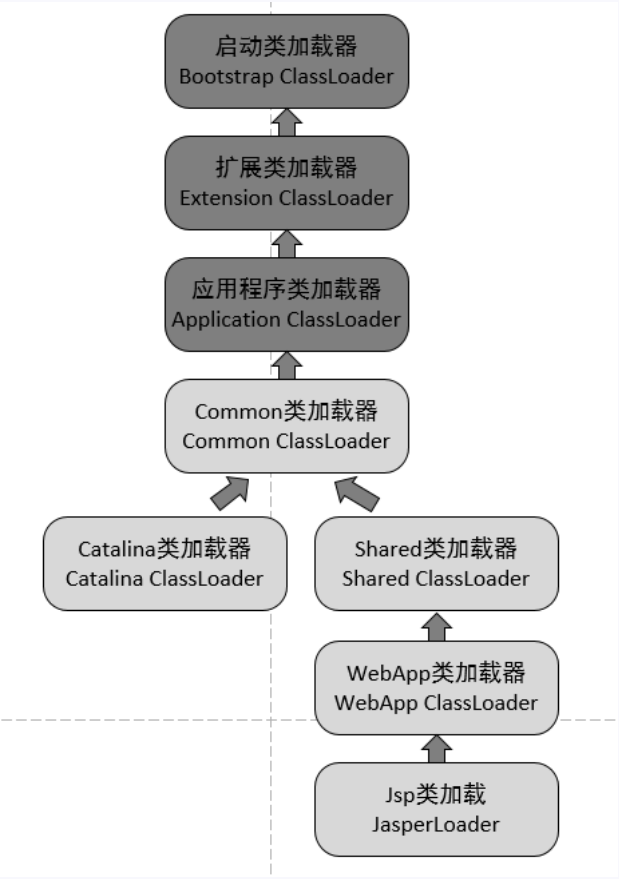
- 部署在同一个服务器上的两个web应用程序锁使用的Java类库可以实现相互隔离
- 部署在同一个服务器上的两个web应用程序锁使用的Java类库可以互相共享
- 服务器需要尽可能地保证自身的安全不受部署的web应用程序影响
- 支持JSP应用的web服务器，大多数都需要支持HotSwap功能

为了解决上述问题，所以各种web服务器都提供了好几个ClassPath路径供用户存放第三方类库，通常以lib或者classes命名。被放置在不同路径中的类库，具备不同的访问范围和服务对象。通常，每一个目录都会有一个相应的自定义类加载器去加载本目录下的Java类库。

### 2）Tomcat的目录结构

在Tomcat下，有3组目录可以存放Java类库：/common，/server，/shared。 另外还有web应用程序自身的目录/WEB-INF，一共4组，把Java类库放置在这些目录中的含义分别如下：

- /common： 类库可以被Tomcat和所有的Web应用程序共同使用
- /server：类库只能被Tomcat使用
- /shared：类库可以被所有web应用程序共同使用，但对tomcat自己不可见
- /WebApp/WEB—INF：类库仅仅可以被此web应用程序使用，对tomcat和其他web应用都不可见




从上图的委派关系中可以看出，CommonClassLoader能加载的类都可以被CatalinaClassLoader和SharedClassLoader使用，而 CatalinaClassLoader 和 SharedClassLoader 加载的类相互隔离。

Tomcat 6.x版本，默认把common、server和shared三个目录合并到了一起成为lib目录。同时也没有建立CatalinaClassLoader和SharedClassLoader的实例。如果需要，可以修改配置文件指定server.loader和share.loader的方式呈现启用Tomcat 5的加载器结构。

## 2. OSGi：灵活的类加载器架构

OSGi：Open Service Gateway Initiative，是一个基于Java引用的动态模块化规范。最常见的应用案例，就是Eclipse IDE。

### 1）模块 Bundle

OSGi中模块（Bundle）与普通的Java类库区别并不太大，两者一般都为JAR格式进行封装，并且内部存储的是Java Package和Class。

但是一个Bundle可以声明它所依赖的Java Package（Import-Package），也可以声明它允许导出发布的Java Package（Export-Package）。

这样子，Bundle之间的依赖关系从传统的上层依赖底层，编程了平级模块之间的依赖。另外类库的可见性得到了非常精确的控制。除此之外，可以实现模块级别的热插拔功能。

### 2）类加载器架构

OSGi的Bundle类加载器直接只有规则，没有固定的委派关系。

比如，某个Bundle A声明了一个它依赖Package 1，如果有其他的Bundle B声明发布了Package 1，那么所有对Package 1的类加载动作都会委派给Bundle B的类加载器去完成。

另外，一个 Bundle 类加载器为其他 Bundle 通过服务时，会根据Export-Package列表严格控制访问范围。

如果一个类存在于Bundle A 的类库中但是没有被Export，那么Bundle A的类加载器能找到这个类，但不会提供给其他Bundle使用。而且OSGi平台也不会把其他Bundle的类加载器请求分配给Bundle A来处理。

## 3. 字节码生成技术与动态代理的实现

字节码生成的例子有很多，最基本的就是javac命令，还有Web中JSP编译器，还有常用的动态代理技术。

动态代理中所谓的“动态”，是相对于Java代码中实际编写代理类的静态代理而言的。它的优势不在于省去了编写代理类的工作量，而是实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为。当代理类与原始类脱离直接联系后，就可能很灵活地重用在不同的应用场景之中。

```
public class DynamicProxyTest {

    interface IHello {
        void sayHello();
    }

    static class Hello implements IHello {
        @Override
        public void sayHello() {
            System.out.println("hello world");
        }
    }

    static class DynamicProxy implements InvocationHandler {

        Object originalObj;

        Object bind(Object originalObj) {
            this.originalObj = originalObj;
            return Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),
originalObj.getClass().getInterfaces(), this);
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            System.out.println("welcome");
            return method.invoke(originalObj, args);
        }
    }

    public static void main(String[] args) {
        IHello hello = (IHello) new DynamicProxy().bind(new Hello());
        hello.sayHello();
    }
}
```

## 4. Retrotranslator：跨越JDK版本

在有些情况下，需要把JDK 1.5 中编译的代码，放到JDK 1.4、1.3的环境中去运行。为了解决这个问题，一种名为“Java逆向移植”的工具（Java Backporting Tools）应运而生，Retrotranslator是其中较出色的一个。

## 5. 自己动手实现远程执行功能

有时候我们需要在运行的服务中，执行一段代码，用来定位或者排除问题。

### 1）目标

为了实现“在服务端执行临时代码”这个需求，我们希望我们的目标产品是这样子的：

- 不依赖JDK版本
- 不改变原有服务端程序的部署
- 不依赖任何第三方类库
- 不侵入原有程序
- 需要直接支持java语言
- 具有足够的自由度，不需要依赖特定的类或者实现特定的接口
- 执行结果能返回客户端

### 2）思路

在实现程序之前，需要解决三个问题：

- 如何编译提交到服务器的java代码？
- 如何执行编译之后的java代码？
- 如何收集java代码的执行结果？

#### 第一个问题

有两种思路。

一种是使用tools.jar包中的com.sun.tools.javac.Main类来编译Java文件，这其实和使用Javac命令编译时一样的。缺点是引入了额外的JAR包，而且把程序“绑死”在Sun的JDK上了。

另一种是直接在客户端编译好，把字节码而不是java代码传到服务器。

#### 第二个问题

让类加载器加载这个类生成的一个Class对象，然后反射调用一下某个方法就可以了。可以直接是main方法。还要注意类在执行完后，可以能够卸载和回收。

#### 第三个问题

因为System.out 和 System.err 是整个虚拟机进程全局共享的资源，所以如果java代码使用了它们把输出流重定向到自己定义的PrintStream对象上，有可能会对原有程序产生影响。

为了解决这个问题，可以在执行类中把对 System.out 的符号引用替换为我们准备的PrintStream的符号引用。

### 3）实现

共有4个类。

第一个类用于实现“同一个类的代码可以被多次加载”。

```
/**
 * 为了多次载入执行类而加入的加载器<br>
 * 把defineClass方法开放出来，只有外部显式调用的时候才会使用到loadByte方法
 * 由虚拟机调用时，仍然按照原有的双亲委派规则使用loadClass方法进行类加载
 *
 * @author zzm
 */
public class HotSwapClassLoader extends ClassLoader {

    public HotSwapClassLoader() {
        super(HotSwapClassLoader.class.getClassLoader());
    }

    public Class loadByte(byte[] classByte) {
        return defineClass(null, classByte, 0, classByte.length);
    }

}
```

第二个类是实现将java.lang.System替换为我们自己定义的HackSystem类的过程，它直接修改符合Class文件格式的byte[]数组中的常量池部分，将常量池中指定内容的CONSTANT_Utf8_info 常量替换为新的字符串。

```java
/**
 * 修改Class文件，暂时只提供修改常量池常量的功能
 * @author zzm
 */
public class ClassModifier {

    /**
     * Class文件中常量池的起始偏移
     */
    private static final int CONSTANT_POOL_COUNT_INDEX = 8;

    /**
     * CONSTANT_Utf8_info常量的tag标志
     */
    private static final int CONSTANT_Utf8_info = 1;

    /**
     * 常量池中11种常量所占的长度，CONSTANT_Utf8_info型常量除外，因为它不是定长的
     */
    private static final int[] CONSTANT_ITEM_LENGTH = { -1, -1, -1, 5, 5, 9, 9, 3, 3, 5, 5, 5, 5 };

    private static final int u1 = 1;
    private static final int u2 = 2;

    private byte[] classByte;

    public ClassModifier(byte[] classByte) {
        this.classByte = classByte;
    }

    /**
     * 修改常量池中CONSTANT_Utf8_info常量的内容
     * @param oldStr 修改前的字符串
     * @param newStr 修改后的字符串
     * @return 修改结果
     */
    public byte[] modifyUTF8Constant(String oldStr, String newStr) {
        int cpc = getConstantPoolCount();
        int offset = CONSTANT_POOL_COUNT_INDEX + u2;
        for (int i = 0; i < cpc; i++) {
            int tag = ByteUtils.bytes2Int(classByte, offset, u1);
            if (tag == CONSTANT_Utf8_info) {
                int len = ByteUtils.bytes2Int(classByte, offset + u1, u2);
                offset += (u1 + u2);
                String str = ByteUtils.bytes2String(classByte, offset, len);
                if (str.equalsIgnoreCase(oldStr)) {
                    byte[] strBytes = ByteUtils.string2Bytes(newStr);
                    byte[] strLen = ByteUtils.int2Bytes(newStr.length(), u2);
                    classByte = ByteUtils.bytesReplace(classByte, offset - u2, u2, strLen);
                    classByte = ByteUtils.bytesReplace(classByte, offset, len, strBytes);
                    return classByte;
                } else {
                    offset += len;
                }
            } else {
                offset += CONSTANT_ITEM_LENGTH[tag];
            }
        }
        return classByte;
    }

    /**
     * 获取常量池中常量的数量
     * @return 常量池数量
     */
    public int getConstantPoolCount() {
        return ByteUtils.bytes2Int(classByte, CONSTANT_POOL_COUNT_INDEX, u2);
    }
}
```

```
/**
 * Bytes数组处理工具
 * @author
 */
public class ByteUtils {

    public static int bytes2Int(byte[] b, int start, int len) {
        int sum = 0;
        int end = start + len;
        for (int i = start; i < end; i++) {
            int n = ((int) b[i]) & 0xff;
            n <<= (--len) * 8;
            sum = n + sum;
        }
        return sum;
    }

    public static byte[] int2Bytes(int value, int len) {
        byte[] b = new byte[len];
        for (int i = 0; i < len; i++) {
            b[len - i - 1] = (byte) ((value >> 8 * i) & 0xff);
        }
        return b;
    }

    public static String bytes2String(byte[] b, int start, int len) {
        return new String(b, start, len);
    }

    public static byte[] string2Bytes(String str) {
        return str.getBytes();
    }

    public static byte[] bytesReplace(byte[] originalBytes, int offset, int len, byte[] replaceBytes) {
        byte[] newBytes = new byte[originalBytes.length + (replaceBytes.length - len)];
        System.arraycopy(originalBytes, 0, newBytes, 0, offset);
        System.arraycopy(replaceBytes, 0, newBytes, offset, replaceBytes.length);
        System.arraycopy(originalBytes, offset + len, newBytes, offset + replaceBytes.length,
originalBytes.length - offset - len);
        return newBytes;
    }
}
```

```java
/**
 * 为JavaClass劫持java.lang.System提供支持
 * 除了out和err外，其余的都直接转发给System处理
 *
 * @author zzm
 */
public class HackSystem {

    public final static InputStream in = System.in;

    private static ByteArrayOutputStream buffer = new ByteArrayOutputStream();

    public final static PrintStream out = new PrintStream(buffer);

    public final static PrintStream err = out;

    public static String getBufferString() {
        return buffer.toString();
    }

    public static void clearBuffer() {
        buffer.reset();
    }

    public static void setSecurityManager(final SecurityManager s) {
        System.setSecurityManager(s);
    }

    public static SecurityManager getSecurityManager() {
        return System.getSecurityManager();
    }

    public static long currentTimeMillis() {
        return System.currentTimeMillis();
    }

    public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) {
        System.arraycopy(src, srcPos, dest, destPos, length);
    }

    public static int identityHashCode(Object x) {
        return System.identityHashCode(x);
    }

    // 下面所有的方法都与java.lang.System的名称一样
    // 实现都是字节转调System的对应方法
    // 因版面原因，省略了其他方法
}
```

```
/**
 * JavaClass执行工具
 *
 * @author zzm
 */
public class JavaClassExecuter {

    /**
     * 执行外部传过来的代表一个Java类的Byte数组<br>
     * 将输入类的byte数组中代表java.lang.System的CONSTANT_Utf8_info常量修改为劫持后的HackSystem类
     * 执行方法为该类的static main(String[] args)方法，输出结果为该类向System.out/err输出的信息
     * @param classByte 代表一个Java类的Byte数组
     * @return 执行结果
     */
    public static String execute(byte[] classByte) {
        HackSystem.clearBuffer();
        ClassModifier cm = new ClassModifier(classByte);
        byte[] modiBytes = cm.modifyUTF8Constant("java/lang/System",
"org/fenixsoft/classloading/execute/HackSystem");
        HotSwapClassLoader loader = new HotSwapClassLoader();
        Class clazz = loader.loadByte(modiBytes);
        try {
            Method method = clazz.getMethod("main", new Class[] { String[].class });
            method.invoke(null, new String[] { null });
        } catch (Throwable e) {
            e.printStackTrace(HackSystem.out);
        }
        return HackSystem.getBufferString();
    }
}
```

4 ) 验证

```
<%@ page import="java.lang.*" %>
<%@ page import="java.io.*" %>
<%@ page import="org.fenixsoft.classloading.execute.*" %>
<%
        InputStream is = new FileInputStream("c:/TestClass.class");
        byte[] b = new byte[is.available()];
        is.read(b);
        is.close();

        out.println("<textarea style='width:1000;height=800'>");
        out.println(JavaClassExecuter.execute(b));
        out.println("</textarea>");
%>
```

## Similar Posts


- Java 并发编程实战-学习日志 ( 二 ) 2 : 取消与关闭
- Java 并发编程实战-学习日志 ( 二 ) 1 : 任务执行
- Java 并发编程实战-学习日志 ( 一 ) 4 : 基础构建模块
- Java 并发编程实战-学习日志 ( 一 ) 3 : 对象的组合
- Java 并发编程实战-学习日志 ( 一 ) 2 : 对象的共享
- Java 并发编程实战-学习日志 ( 一 ) 1 : 线程安全

**上一篇** SparkContext 学习

**下一篇** Spark中RDD的介绍