浅析 JVM 中的符号引用与直接引用

转载 这瓜保熟么 2018-09-11 18:10:30 2304 收藏 8

分类专栏： jvm

前言

在 JVM 的学习过程中，一直会遇到符号引用和直接引用这两个概念。最近我也查阅了一些资料，有了一些初步的认识，记录在此与大家分享。文中的内容，主要参考自 JVM里的符号引用如何存储？ 与 自己动手写Java虚拟机。

关于符号引用与直接引用，我们还是用一个实例来分析吧。看下面的 Java 代码：

```
package test;
public class Test {
    public static void main(String[] args) {
        Sub sub = new Sub();
        int a = 100;
        int d = sub.inc(a);
    }
}
class Sub {
    public int inc(int a) {
        return a + 2;
    }
}
```

编译后使用 javap 分析工具，会得到下面的 Class 文件内容：

```
Constant pool:
   #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
   #2 = Class              #16            // test/Sub
   #3 = Methodref          #2.#15         // test/Sub."<init>":()V
   #4 = Methodref          #2.#17         // test/Sub.inc:(I)I
   #5 = Class              #18            // test/Test
   #6 = Class              #19            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               main
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               SourceFile
  #14 = Utf8               Test.java
  #15 = NameAndType        #7:#8          // "<init>":()V
  #16 = Utf8               test/Sub
  #17 = NameAndType        #20:#21        // inc:(I)I
  #18 = Utf8               test/Test
  #19 = Utf8               java/lang/Object
```

```
   #20 = Utf8               inc
   #21 = Utf8               (I)I
{
  public test.Test();
    descriptor: ()V
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    Code:
      stack=2, locals=4, args_size=1
         0: new           #2                  // class test/Sub
         3: dup
         4: invokespecial #3                  // Method test/Sub."<init>":()V
         7: astore_1
         8: bipush        100
        10: istore_2
        11: aload_1
        12: iload_2
        13: invokevirtual #4                  // Method test/Sub.inc:(I)I
        16: istore_3
        17: return
}
```

因为篇幅有限，上面的内容只保留了常量池，和 Code 部分。下面我们主要对 inc 方法的调用来进行说明。

符号引用

在 main 方法的字节码中，调用 inc 方法的指令如下：

```
13: invokevirtual #4                  // Method test/Sub.inc:(I)I
```

invokevirtual 指令就是调用实例方法的指令，后面的操作数 4 是 Class 文件中常量池的下标，表示用来指定要调用的目标方法。我们再来看常量池在这个位置上的内容：

```
#4 = Methodref          #2.#17
```

这是一个 Methodref 类型的数据，我们再来看看虚拟机规范中对该类型的说明：

```
CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

这实际上就是一种引用类型，tag 表示了常量池数据类型，这里固定是 10。class_index 表示了类的索引，name_and_type_index 表示了名称与类型的索引，这两个也都是常量池的下标。在 javap 的输出中，已经将对应的关系打印了出来，我们可以直接的观察到它都引用了哪些类型：

```
#4 = Methodref          #2.#17         // test/Sub.inc:(I)I
|--#2 = Class              #16            // test/Sub
|  |--#16 = Utf8               test/Sub
|--#17 = NameAndType        #20:#21        // inc:(I)I
|  |--#20 = Utf8               inc
|  |--#21 = Utf8               (I)I
```

这里我们将其表现为树的形式。可以看到，我们可以得到该方法所在的类，以及方法的名称和描述符。于是我们根据 invokevirtual 的操作数，找到了常量池中方法对应的 Methodref，进而找到了方法所在的类以及方法的名称和描述符，当然这些内容最终都是字符串形式。

实际上这就是一个符号引用的例子，符号引用也可以理解为像这样使用文字形式来描述引用关系。

直接引用

符号引用在上面说完了，我们知道符号引用大概就是文字形式表示的引用关系。但是在方法的执行中，只有这样一串字符串，有什么用呢？方法的本体在哪里？下面这就是直接引用的概念了，这里我用自己目前的理解总结一下，直接引用就是通过对符号引用进行解析，来获得真正的函数入口地址，也就是在运行的内存区域找到该方法字节码的起始位置，从而真正的调用方法。

那么将符号引用解析为直接引用的过程是什么样的呢？我这个小渣渣目前也给不出确定的答案，在 JVM 里的符号引用如何存储？ 里，RednaxelaFX 大大给出了一个 Sun JDK 1.0.2 的实现；在 自己动手写Java虚拟机 中，作者给出了一种用 Go 的简单实现，下面这里就来看一下这个简单一些的实现。在 HotSpot VM 中的实现肯定要复杂得多，这里还是以大致的学习了解为主，以后如果有时间有精力，再去研究一下 OpenJDK 中 HotSpot VM 的实现。

不过不管是哪种实现，肯定要先读取 Class 文件，然后将其以某种格式保存在内存中，类的数据会记录在某个结构体内，方法的数据也会记录在另外的结构体中，然后将结构体之间相互组合、关联起来。比如，我们用下面的形式来表达 Class 的数据在内存中的保存形式：

```
type Class struct {
        accessFlags uint16         // 访问控制
        name string                // 类名
        superClassName string      // 父类名
        interfaceNames []string    // 接口名列表
        constantPool *ConstantPool // 该类对应的常量池
        fields []*Field            // 字段列表
        methods []*Method          // 方法列表
        loader *ClassLoader        // 加载该类的类加载器
        superClass *Class          // 父类结构体的引用
        interfaces []*Class        // 各个接口结构体的引用
        instanceSlotCount uint     // 类中的实例变量数量
        staticSlotCount uint       // 类中的静态变量数量
        staticVars Slots           // 类中的静态变量的引用列表
        initStarted bool           // 类是否被初始化
}
```

类似的，常量池中的方法引用，也要有类似的结构来表示：

```
type MethodRef struct {
    cp *ConstantPool  // 常量池
    className string  // 所在的类名
    class *Class      // 所在的类的结构体引用
    name string       // 方法名
    descriptor string // 描述符
    method *Method    // 方法数据的引用
}
```

回到上面符号解析的例子。当遇到 invokevirtual 指令时，根据后面的操作数，可以去常量池中指定位置取到方法引用的结构体。实际上这个结构体中已经包含了上面看到的各种符号引用，最下面的 method 就是真正的方法数据。类加载到内存中时，method 的值为空，当方法第一次调用时，会根据符号引用，找到方法的直接引用，并将值赋予 method。从而后面再次调用该方法时，只需要返回 method 即可。下面我们看方法的解析过程：

```
func (self *MethodRef) resolveMethodRef() {
        c := self.ResolvedClass()
        method := lookupMethod(c, self.name, self.descriptor)
        if method == nil {
                panic("java.lang.NoSuchMethodError")
        }
        self.method = method
}
```

这里面省略了验证的部分，包括检查解析后的方法是否为空、检查当前类是否可以访问该方法，等等。首先我们看到，第一步是找到方法对应的类：

```
func (self *SymRef) ResolvedClass() *Class {
        if self.class == nil {
        d := self.cp.class
        c := d.loader.LoadClass(self.className)
        self.class = c
        }
        return self.class
}
```

在 MethodRef 结构体中包含对应 class 的引用，如果 class 不为空，则可以直接返回；否则会根据类名，使用当前类的类加载器去尝试加载这个类。最后将加载好的类引用赋给 MethodRef.class。找到了方法所在的类，下一步就是从类中找到这个方法，也就是方法数据在内存中的地址，对应上面的 lookupMethod 方法。查找时，会遍历类中的方法列表，这块在类加载的过程中已经完成，下面是方法数据的结构体：

```
type Method struct {
        accessFlags uint16
        name string
        descriptor string
        class *Class
```

```
        maxStack uint
        maxLocals uint
        code []byte
        argSlotCount uint
}
```

这个其实就和 Class 文件中的 Code 属性类似，这里面省略了异常和其他的一些信息。类加载过程中，会将各个方法的 Code 属性按照上面的结构保存在内存中，然后将类中所有方法的地址列表保存在 Class 结构体中。当在 Class 结构体中查找指定方法时，只需要遍历方法列表，然后比较方法名和描述符即可：

```
for c := class; c != nil; c = c.superClass {
    for _, method := range c.methods {
        if method.name == name && method.descriptor == descriptor {
                return method
        }
    }
}
```

可以看到，查找方法会从当前方法查找，如果找不到，会继续从父类中查找。除此以外，还会从实现的接口列表中查找，代码中省略了这部分，还有一些判断的条件。

最终，如果成功找到了指定方法，就会将方法数据的地址赋给 MethodRef.method，后面对该方法的调用只需要直接返回 MethodRef.method 即可。

以上便是 自己动手写Java虚拟机 一书中，符号引用解析为直接引用的实现。