崔子寒
MG20330012
zihancui@smail.nju.edu.cn

Assignment 2: jmtrace
ISER,20

2020 年 12 月 3 日

1 Java 字节码插桩

为了获得更好的跨平台特性，Java 的源代码会被编译成特定的字节码，再由 JVM 来解释执行。JVM 的输入就是 Java 源代码编译产生的字节码，每一个类都会被编译成一个.class 文件，当我们执行 Java 代码时，JVM 会将所需的.class 文件中类的信息加载入内存，并解析生成对应的 class 对象。这个过程被称为 Java 的类加载，实际上 JVM 要做的事情非常复杂，但是我们只需要知道我们可以捕捉这个加载过程，并修改即将加载入内存中的类的字节码，在我们关心的指令前后插入一些新的指令，这就是 Java 字节码插桩的原理。

2 插桩实现

2.1 Java Instrument 和 Java ASM

为了方便开发者捕捉类的加载过程，java 提供了 `java.lang.instrument` 包，其中的 `Instrumentation` 类提供了在类加载过程中修改类的字节码的方法。当我们运行一个 jar 包时，如果设定了它的 javaagent，那么 JVM 就会在加载 jar 之前先加载 javaagent，并执行它的 `preMain` 方法。在 `preMain` 方法中，我们会调用 `Instrumentation` 类的 `addTransformer` 方法，传入一个实现了 `ClassFileTransformer` 接口的对象。然后会通过运行该对象的 `transform` 方法来完成插桩,所以我们的任务主要是编写一个实现了 `ClassFileTransformer` 接口的类。

直接在 `transform` 方法中解析字节码工作量比较大，我们可以利用一些开源的包，比如 Java ASM。Java ASM 提供了两套不同风格的 API 用于插桩，分别是基于事件的 Core API 和基于对象的 Tree API。两套 API 的功能是等价的，我们采用 Core API 来实现插桩。

2.2 理解相关 JVM 指令

在插桩之前，首先需要理解 JVM 中的一些特性，最重要的一点是 JVM 的指令集是基于栈而不是基于寄存器的。对于需要参数的指令，都是在执行指令之前将参数放入栈中，指令执行时从栈中取出参数，指令执行结束后，将结果再压入栈中，函数的调用也是类似。

JVM 中的数据类型再栈中有两种。category 1 在栈中的长度是 1 个单位，category 2 在栈中的长度是 2 个单位，只有 `double` 和 `long` 是 category 2。在插桩的实现中，我们需要针对不同的数据类型，使用不同的指令。

2.3 实现插桩

在 `transform` 方法中，我们读取类的字节码，然后用一个 visitor 按照指令的顺序遍历字节码，当读到目标的指令时，就进行插桩，最后将插桩完的字节码返回。

```
ClassReader reader = new ClassReader(classfileBuffer);
ClassWriter writer = new ClassWriter(reader, ClassWriter.COMPUTE_MAXS);
ClassVisitor visitor = new MemoryTraceClassVisitor(writer);
reader.accept(visitor, 0);
return writer.toByteArray();
```

在 visitor 中，我们根据读到的指令类型以及 opcode 来检查是否读到了我们的目标指令，如果读到，就调用相关的方法进行插桩。我们的目标指令是 `*aload/*astore/getfield/putfield/getstatic/putstatic`，我们需要获取这些指令执行时有关的对象的信息，有关线程的信息等，一个比较好的方法是获取对象的引用，以及相关的变量，并将它们作为参数放入栈中，然后插入调用我们自己写的打印信息的方法的指令。1展示了如何对 `lastore` 指令实现上面的过程。

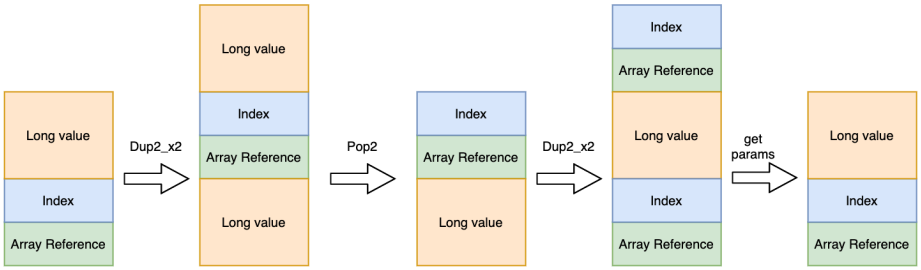

图 1: lastore instrument

需要注意的一点是，对于 `putfield` 指令，我们直接通过指令是无法判断栈中的 value 是 category 1 还是 category 2，所以需要通过反射获取相关的对象的类型，然后再分情况进行操作：

```
final Type fieldType = Type.getType(descriptor);
if(fieldType.getSize() == 1) {
    // value is category 1
    mv.visitInsn(Opcodes.DUP2);
    // [...,objRef, value, objRef, value] <-
    mv.visitInsn(Opcodes.POP);
    // [...,objRef, value, objRef] <-
} else {
    // value is category 2
    mv.visitInsn(Opcodes.DUP2_X1);
    // [...,value, objRef, value] <-
    mv.visitInsn(Opcodes.POP2);
    // [...,value, objRef] <-
    mv.visitInsn(Opcodes.DUP_X2);
    // [...,objRef, value, objRef] <-
}
```

在准备好了参数之后，只需要调用我们的方法，就可以完成插桩：

```
// [..., objRef, value, objRef] <-
mv.visitLdcInsn(owner);
// [..., objRef, value, objRef, ownerStringRef] <-
mv.visitLdcInsn(name);
// [..., objRef, value, objRef, ownerStringRef, nameStringRef] <-
mv.visitMethodInsn(Opcodes.INVOKESTATIC, Type.getInternalName(MemoryTraceUtils.class),
    "tracePutField", "(Ljava/lang/Object;Ljava/lang/String;Ljava/lang/String;)V", false);
// [..., objRef, value] <-
```