

33 | Java Agent与字节码注入

2018-10-05 郑雨迪

深入拆解Java虚拟机

[进入课程 >](#)




讲述：郑雨迪

时长 10:26 大小 4.79M




关于 Java agent，大家可能都听过大名鼎鼎的`premain`方法。顾名思义，这个方法指的就是在`main`方法之前执行的方法。

 复制代码

```
1 package org.example;
2
3 public class MyAgent {
4     public static void premain(String args) {
5         System.out.println("premain");
6     }
7 }
```

我在上面这段代码中定义了一个`premain`方法。这里需要注意的是，Java 虚拟机所能识别的`premain`方法接收的是字符串类型的参数，而并非类似于`main`方法的字符串数组。

为了能够以 Java agent 的方式运行该`premain`方法，我们需要将其打包成 jar 包，并在其中的 MANIFEST.MF 配置文件中，指定所谓的`Premain-class`。具体的命令如下所示：

 复制代码

```
1 # 注意第一条命令会向 manifest.txt 文件写入两行数据，其中包括一行空行
2 $ echo 'Premain-Class: org.example.MyAgent
3 ' > manifest.txt
4 $ jar cvmf manifest.txt myagent.jar org/
5 $ java -javaagent:myagent.jar HelloWorld
6 premain
7 Hello, World
```

除了在命令行中指定 Java agent 之外，我们还可以通过 Attach API 远程加载。具体用法如下面的代码所示：

 复制代码

```
1 import java.io.IOException;
2
3 import com.sun.tools.attach.*;
4
5 public class AttachTest {
6     public static void main(String[] args)
7         throws AttachNotSupportedException, IOException, AgentLoadException, AgentInitial:
8     if (args.length <= 1) {
9         System.out.println("Usage: java AttachTest <PID> /PATH/TO/AGENT.jar");
10        return;
11    }
12    VirtualMachine vm = VirtualMachine.attach(args[0]);
13    vm.loadAgent(args[1]);
14 }
15 }
16
```

使用 Attach API 远程加载的 Java agent 不会再先于`main`方法执行，这取决于另一虚拟机调用 Attach API 的时机。并且，它运行的也不再是`premain`方法，而是名为`agentmain`的方法。

```
1 public class MyAgent {
2     public static void agentmain(String args) {
3         System.out.println("agentmain");
4     }
5 }
```

相应的，我们需要更新 jar 包中的 manifest 文件，使其包含 Agent-Class 的配置，例如 Agent-Class: org.example.MyAgent。

```
1 $ echo 'Agent-Class: org.example.MyAgent
2 ' > manifest.txt
3 $ jar cvmf manifest.txt myagent.jar org/
4 $ java HelloWorld
5 Hello, World
6 $ jps
7 $ java AttachTest <pid> myagent.jar
8 agentmain
9 // 最后一句输出来自于运行 HelloWorld 的 Java 进程
```

Java 虚拟机并不限制 Java agent 的数量。你可以在 java 命令后附上多个 -javaagent 参数，或者远程 attach 多个 Java agent，Java 虚拟机会按照定义顺序，或者 attach 的顺序逐个执行这些 Java agent。

在 premain 方法或者 agentmain 方法中打印一些字符串并不出奇，我们完全可以将其中的逻辑并入 main 方法，或者其他监听端口的线程中。除此之外，Java agent 还提供了一套 instrumentation 机制，允许应用程序拦截类加载事件，并且更改该类的字节码。

接下来，我们来了解一下基于这一机制的字节码注入。

字节码注入

```
1 package org.example;
2
3 import java.lang.instrument.*;
```

```

4 import java.security.ProtectionDomain;
5
6 public class MyAgent {
7     public static void premain(String args, Instrumentation instrumentation) {
8         instrumentation.addTransformer(new MyTransformer());
9     }
10
11     static class MyTransformer implements ClassFileTransformer {
12         public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRec
13             ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassF
14             System.out.printf("Loaded %s: 0x%X%X%X%X\n", className, classfileBuffer[0], class
15                 classfileBuffer[2], classfileBuffer[3]);
16         return null;
17     }
18 }
19 }

```

我们先来看一个例子。在上面这段代码中，`premain`方法多出了一个`Instrumentation`类型的参数，我们可以通过它来注册类加载事件的拦截器。该拦截器需要实现`ClassFileTransformer`接口，并重写其中的`transform`方法。

`transform`方法将接收一个 `byte` 数组类型的参数，它代表的是正在被加载的类的字节码。在上面这段代码中，我将打印该数组的前四个字节，也就是 Java class 文件的魔数（magic number）`0xCAFEBAFE`。

`transform`方法将返回一个 `byte` 数组，代表更新过后的类的字节码。当方法返回之后，Java 虚拟机会使用所返回的 `byte` 数组，来完成接下来的类加载工作。不过，如果 `transform`方法返回 `null` 或者抛出异常，那么 Java 虚拟机将使用原来的 `byte` 数组完成类加载工作。

基于这一类加载事件的拦截功能，我们可以实现字节码注入（bytecode instrumentation），往正在被加载的类中插入额外的字节码。

在工具篇中我曾经介绍过字节码工程框架 ASM 的用法。下面我将演示它的[tree 包](#)（依赖于[基础包](#)），用面向对象的方式注入字节码。

 复制代码

```

1 package org.example;
2

```

```


3 import java.lang.instrument.*;
4 import java.security.ProtectionDomain;
5 import org.objectweb.asm.*;
6 import org.objectweb.asm.tree.*;
7
8 public class MyAgent {
9     public static void premain(String args, Instrumentation instrumentation) {
10         instrumentation.addTransformer(new MyTransformer());
11     }
12
13     static class MyTransformer implements ClassFileTransformer, Opcodes {
14         public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRe
15             ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassF
16             ClassReader cr = new ClassReader(classfileBuffer);
17             ClassNode classNode = new ClassNode(ASM7);
18             cr.accept(classNode, ClassReader.SKIP_FRAMES);
19
20             for (MethodNode methodNode : classNode.methods) {
21                 if ("main".equals(methodNode.name)) {
22                     InstrList instrumentation = new InstrList();
23                     instrumentation.add(new FieldInsnNode(GETSTATIC, "java/lang/System", "out", "
24                     instrumentation.add(new LdcInsnNode("Hello, Instrumentation!"));
25                     instrumentation
26                         .add(new MethodInsnNode(INVOKEVIRTUAL, "java/io/PrintStream", "println",
27
28                     methodNode.instructions.insert(instrumentation);
29                 }
30             }
31
32             ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES | ClassWriter.COMPUTE_
33             classNode.accept(cw);
34             return cw.toByteArray();
35         }
36     }
37 }

```

上面这段代码不难理解。我们将使用ClassReader读取所传入的 byte 数组，并将其转换成ClassNode。然后我们将遍历ClassNode中的MethodNode节点，也就是该类中的构造器和方法。

当遇到名字为"main"的方法时，我们会在方法的入口处注入

System.out.println("Hello, Instrumentation!");。运行结果如下所示：

 复制代码

```
1 $ java -javaagent:myagent.jar -cp ./PATH/TO/asm-7.0-beta.jar:/PATH/TO/asm-tree-7.0-beta
```

```
2 Hello, Instrumentation!
3 Hello, World!
```

Java agent 还提供了另外两个功能 `redefine` 和 `retransform`。这两个功能针对的是已加载的类，并要求用户传入所要 `redefine` 或者 `retransform` 的类实例。


其中，`redefine` 指的是舍弃原本的字节码，并替换成由用户提供的 `byte` 数组。该功能比较危险，一般用于修复出错了的字节码。

`retransform` 则将针对所传入的类，重新调用所有已注册的 `ClassFileTransformer` 的 `transform` 方法。它的应用场景主要有如下两个。

第一，在执行 `premain` 或者 `agentmain` 方法前，Java 虚拟机早已加载了不少类，而这些类的加载事件并没有被拦截，因此也没有被注入。使用 `retransform` 功能可以注入这些已加载但未注入的类。

第二，在定义了多个 Java agent，多个注入的情况下，我们可能需要移除其中的部分注入。当调用 `Instrumentation.removeTransformer` 去除某个注入类后，我们可以调用 `retransform` 功能，重新从原始 `byte` 数组开始进行注入。

Java agent 的这些功能都是通过 JVMTI agent，也就是 C agent 来实现的。JVMTI 是一个事件驱动的工具实现接口，通常，我们会在 C agent 加载后的入口方法 `Agent_OnLoad` 处注册各个事件的钩子（hook）方法。当 Java 虚拟机触发了这些事件时，便会调用对应的钩子方法。

 复制代码


```
1 JNIEXPORT jint JNICALL
2 Agent_OnLoad(JavaVM *vm, char *options, void *reserved);
```

举个例子，我们可以为 JVMTI 中的 `ClassFileLoadHook` 事件设置钩子，从而在 C 层面拦截所有的类加载事件。关于 JVMTI 的其他事件，你可以参考该[链接](#)。

基于字节码注入的 profiler

我们可以利用字节码注入来实现代码覆盖工具（例如[JaCoCo](#)），或者各式各样的 profiler。

通常，我们会定义一个运行时类，并在某一程序行为的周围，注入对该运行时类中方法的调用，以表示该程序行为正要发生或者已经发生。

 复制代码

```
1 package org.example;
2
3 import java.util.concurrent.ConcurrentHashMap;
4 import java.util.concurrent.atomic.AtomicInteger;
5
6 public class MyProfiler {
7     public static ConcurrentHashMap<Class<?>, AtomicInteger> data = new ConcurrentHashMap<>();
8
9     public static void fireAllocationEvent(Class<?> klass) {
10         data.computeIfAbsent(klass, kls -> new AtomicInteger())
11             .incrementAndGet();
12     }
13
14     public static void dump() {
15         data.forEach((kls, counter) -> {
16             System.err.printf("%s: %d\n", kls.getName(), counter.get());
17         });
18     }
19
20     static {
21         Runtime.getRuntime().addShutdownHook(new Thread(MyProfiler::dump));
22     }
23 }
```

举个例子，上面这段代码便是一个运行时类。该类维护了一个 `HashMap`，用来统计每个类所新建实例的数目。当程序退出时，我们将逐个打印出每个类的名字，以及其新建实例的数目。

在 `Java agent` 中，我们会截获正在加载的类，并且在每条 `new` 字节码之后插入对 `fireAllocationEvent` 方法的调用，以表示当前正在新建某个类的实例。具体的注入代码如下所示：

```
1 package org.example;
2
3 import java.lang.instrument.*;
4 import java.security.ProtectionDomain;
5
6 import org.objectweb.asm.*;
7 import org.objectweb.asm.tree.*;
8
9 public class MyAgent {
10
11     public static void premain(String args, Instrumentation instrumentation) {
12         instrumentation.addTransformer(new MyTransformer());
13     }
14
15     static class MyTransformer implements ClassFileTransformer, Opcodes {
16         public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRec
17             ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassF
18             if (className.startsWith("java") ||
19                 className.startsWith("javax") ||
20                 className.startsWith("jdk") ||
21                 className.startsWith("sun") ||
22                 className.startsWith("com/sun") ||
23                 className.startsWith("org/example")) {
24                 // Skip JDK classes and profiler classes
25                 return null;
26             }
27
28             ClassReader cr = new ClassReader(classfileBuffer);
29             ClassNode classNode = new ClassNode(ASM7);
30             cr.accept(classNode, ClassReader.SKIP_FRAMES);
31
32             for (MethodNode methodNode : classNode.methods) {
33                 for (AbstractInsnNode node : methodNode.instructions.toArray()) {
34                     if (node.getOpcode() == NEW) {
35                         TypeInsnNode typeInsnNode = (TypeInsnNode) node;
36
37                         InsnList instrumentation = new InsnList();
38                         instrumentation.add(new LdcInsnNode(Type.getObjectType(typeInsnNode.desc)))
39                         instrumentation.add(new MethodInsnNode(INVOKESTATIC, "org/example/MyProfile
40                             "(Ljava/lang/Class;)V", false));
41
42                         methodNode.instructions.insert(node, instrumentation);
43                     }
44                 }
45             }
46
47             ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES | ClassWriter.COMPUTE_
48             classNode.accept(cw);
49             return cw.toByteArray();
50         }
51     }
```




你或许已经留意到，我们不得不排除对 JDK 类以及该运行时类的注入。这是因为，对这些类的注入很可能造成死循环调用，并最终抛出 `StackOverflowException` 异常。

举个例子，假设我们在 `PrintStream.println` 方法入口处注入 `System.out.println("blahblah")`，由于 `out` 是 `PrintStream` 的实例，因此当执行注入代码时，我们又会调用 `PrintStream.println` 方法，从而造成死循环。

解决这一问题的关键在于设置一个线程私有的标识位，用以区分应用代码的上下文以及注入代码的上下文。当即将执行注入代码时，我们将根据标识位判断是否已经位于注入代码的上下文之中。如果不是，则设置标识位并正常执行注入代码；如果是，则直接返回，不再执行注入代码。

字节码注入的另一个技术难点则是命名空间。举个例子，不少应用程序都依赖于字节码工程库 ASM。当我们的注入逻辑依赖于 ASM 时，便有可能出现注入使用最新版本的 ASM，而应用程序使用较低版本的 ASM 的问题。

JDK 本身也使用了 ASM 库，如用来生成 Lambda 表达式的适配器类。JDK 的做法是重命名整个 ASM 库，为所有类的包名添加 `jdk.internal` 前缀。我们显然不好直接更改 ASM 的包名，因此需要借助自定义类加载器来隔离命名空间。

除了上述技术难点之外，基于字节码注入的工具还有另一个问题，那便是观察者效应（observer effect）对所收集的数据造成的影响。

举个利用字节码注入收集每个方法的运行时间的例子。假设某个方法调用了另一个方法，而这两个方法都被注入了，那么统计被调用者运行时间的注入代码所耗费的时间，将不可避免地被计入至调用者方法的运行时间之中。

再举一个统计新建对象数目的例子。我们知道，即时编译器中的逃逸分析可能会优化掉新建对象操作，但它不会消除相应的统计操作，比如上述例子中对 `fireAllocationEvent` 方法的调用。在这种情况下，我们将统计没有实际发生的新建对象操作。

另一种情况则是，我们所注入的对`fireAllocationEvent`方法的调用，将影响到方法内联的决策。如果该新建对象的构造器调用恰好因此没有被内联，从而造成对象逃逸。在这种情况下，原本能够被逃逸分析优化掉的新建对象操作将无法优化，我们也将统计到原本不会发生的新建对象操作。

总而言之，当使用字节码注入开发 profiler 时，需要辩证地看待所收集的数据。它仅能表示在被注入的情况下程序的执行状态，而非没有注入情况下的程序执行状态。

面向方面编程

说到字节码注入，就不得不提面向方面编程（Aspect-Oriented Programming，AOP）。面向方面编程的核心理念是定义切入点（pointcut）以及通知（advice）。程序控制流中所有匹配该切入点的连接点（joinpoint）都将执行这段通知代码。

举个例子，我们定义一个指代所有方法入口的切入点，并指定在该切入点执行的“打印该方法的名字”这一通知。那么每个具体的方法入口便是一个连接点。

面向方面编程的其中一种实现方式便是字节码注入，比如[AspectJ](#)。

在前面的例子中，我们也相当于使用了面向方面编程，在所有的`new`字节码之后执行了下面这样一段通知代码。

 复制代码

```
1 `MyProfiler.fireAllocationEvent(<Target>.class)`
```

我曾经参与开发过一个应用了面向方面编程思想的字节码注入框架[DiSL](#)。它支持用注解来定义切入点，用普通 Java 方法来定义通知。例如，在方法入口处打印所在的方法名，可以简单表示为如下代码：

 复制代码

```
1 @Before(marker = BodyMarker.class)
2 static void onMethodEntry(MethodStaticContext msc) {
3     System.out.println(msc.thisMethodFullName());
4 }
```

如果有同学对这个工具感兴趣，或者有什么需求或者建议，欢迎你在留言中提出。

总结与实践

今天我介绍了 Java agent 以及字节码注入。

我们可以通过 Java agent 的类加载拦截功能，修改某个类所对应的 byte 数组，并利用这个修改过后的 byte 数组完成接下来的类加载。

基于字节码注入的 profiler，可以统计程序运行过程中某些行为的出现次数。如果需要收集 Java 核心类库的数据，那么我们需要小心避免无限递归调用。另外，我们还需通过自定义类加载器来解决命名空间的问题。

由于字节码注入会产生观察者效应，因此基于该技术的 profiler 所收集到的数据并不能反映程序的真实运行状态。它所反映的是程序在被注入的情况下的执行状态。

今天的实践环节，请你思考如何注入方法出口。除了正常执行路径之外，你还需考虑异常执行路径。



深入拆解 Java 虚拟机

Oracle 高级研究员 手把手带你入门 JVM

郑雨迪 Oracle Labs 高级研究员，计算机博士



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

上一篇 32 | JNI的运行机制

下一篇 34 | Graal：用Java编译Java

精选留言 (7)

写留言



godtrue

2018-10-13

3

阅过留痕

1：Java agent 是啥玩意？

这个概念老师没有详细讲解，我的理解是Java语言的一个特性，这个特性能够实现Java字节码的注入...

展开



小美

2018-11-19

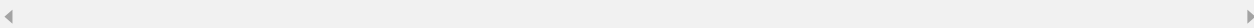
2

用attach的方式注入字节码的时候遇到了99线升高的性能问题，看一些资料说 class redefinition 的时候会阻塞线程。请问能详细讲下吗？

展开

作者回复: 你是在做redefine时出问题，还是没触发时便已出问题？

class redefinition需要爬每个线程的Java栈，检查有没有用到被redefine的类的方法，因此会stop-the-world。另外，redefine后，JIT' ed代码会被抛弃，重新解释执行



feng

2018-10-07

2

第一个实验做的不严谨，第一，木有定义HelloWord类，第二，没有执行编译操作，不知道是有意为之，还是不小心把步骤漏掉了

展开





Scott
2018-10-06



出方法时需要注入的字节码除了返回，还有几种情况，如果没有catch块，就拦截throw，如果有，但是catch块里面可能有很多层，只是遍历inst应该是不可以的

展开 ▾

作者回复: 其实不用管有没有catch块，有没有throw，直接给所有代码罩一个catch any的异常处理就行了



奇奇
2019-04-29



ASM7 GETSTATIC这些常量是哪里来的？

展开 ▾



feng
2018-10-07



还有个问题想请教下，每次启动的时候都会打印如下信息，objc[2614]: Class JavaLaunchHelper is implemented in both
/Library/Java/JavaVirtualMachines/jdk1.8.0_31.jdk/Contents/Home/bin/java
(0x102f6f4c0) and
/Library/Java/JavaVirtualMachines/jdk1.8.0_31.jdk/Contents/Home/jre/lib/libinstr...

展开 ▾



Scott
2018-10-06



我看到了jvmti可以回调异常事件，但是java.lang.instrument包下没有处理这个事件的，只能在load时回调，处理异常究竟是怎么做的？

展开 ▾

作者回复: Instrumentation包并没有所有JVMTI功能

