

bytebuddy动态加载原理解析



作者 浮云1o__o1 (/u/09e5b9581d54) +关注

2016.11.26 11:36 字数 772 阅读 220 评论 0 喜欢 0

(/u/09e5b9581d54)

前言

bytebuddy是一个提供了一个API用于生成任意的Java类工具包，给需要编写javaagent的代码用户提供了一个很方便的工具。

一般来说，如果要编写agent代码一般都是从premain函数开始，然后在启动的时候通过javaagent命令进行启动，bytebuddy提供了一个动态加载agent方式的api，本文主要分析这个api的工作原理，本文使用的bytebuddy版本是1.4.21

开始

动态加载javaagent主要是在程序运行过程中通过 `ByteBuddyAgent.install()`；获得 `Instrumentation inst` 对象，而不是在启动的时候通过加入 `-javaagent` 来获得 `Instrumentation inst` 对象。

`ByteBuddyAgent.install()`；源码里面刚开始就是设置一些默认的配置，接下来就是最关键的代码

```
public static synchronized Instrumentation install(AttachmentProvider attachmentProvider, Pr
    Instrumentation instrumentation = doGetInstrumentation();
    if (instrumentation != null) {
        return instrumentation;
    }
    install(attachmentProvider, processProvider.resolve(), AgentProvider.ForByteBuddyAgent.
    return doGetInstrumentation();
}
```

刚开始尝试获得 `Instrumentation`，如果是第一次启动，就会返回null，然后进入到函数 `install`

```
private static void install(AttachmentProvider attachmentProvider, String processId, AgentPr
    AttachmentProvider.Accessor attachmentAccessor = attachmentProvider.attempt();
    if (!attachmentAccessor.isAvailable()) {
        throw new IllegalStateException();
    }
    try {
        Object virtualMachineInstance = attachmentAccessor.getVirtualMachineType()
            .getDeclaredMethod(ATTACH_METHOD_NAME, String.class)
            .invoke(STATIC_MEMBER, processId);
        try {
            attachmentAccessor.getVirtualMachineType()
                .getDeclaredMethod(LOAD_AGENT_METHOD_NAME, String.class, String.class)
                .invoke(virtualMachineInstance, agentProvider.resolve().getAbsolutePath()
            } finally {
                attachmentAccessor.getVirtualMachineType().getDeclaredMethod(DETACH_METHOD_N
            }
        } catch (RuntimeException exception) {
            throw exception;
        } catch (Exception exception) {
            throw new IllegalStateException("Error during attachment using: " + attachmentPr
        }
    }
}
```



attachmentAccessor 对象里面就是刚才上面加入的一些默认的设置，里面主要是设置了一个类型为 `Class<?>` `clazz = com.sun.tools.attach.VirtualMachine.getClass()` 成员，这个作用是什么可以参考：

<https://docs.oracle.com/javase/7/docs/jdk/api/attach/spec/com/sun/tools/attach/VirtualMachine.html>

(<https://docs.oracle.com/javase/7/docs/jdk/api/attach/spec/com/sun/tools/attach/VirtualMachine.html>)

然后通过反射 `VirtualMachine` 类里面的 `attach` 方法，然后传入当前进程号，来attach到当前进程的vm上面，然后通过 `loadAgent` 方法，把 `bytebuddyAgent.jar` 加载到进程中

那么 `bytebuddyAgent.jar` 是从哪里来的呢？就是 `agentProvider.resolve().getAbsolutePath()` 来得到的，他是在代码运行时产生的一个临时文件，产生的代码是

```
public File resolve() throws IOException {
    File agentJar;
    InputStream inputStream = Installer.class.getResourceAsStream('/') + Installer.class.getName().replace('.', '/') + ".jar";
    if (inputStream == null) {
        throw new IllegalStateException("Cannot locate class file for Byte Buddy");
    }
    try {
        agentJar = File.createTempFile(AGENT_FILE_NAME, JAR_FILE_EXTENSION);
        agentJar.deleteOnExit(); // Agent jar is required until VM shutdown due to instrumentation
        Manifest manifest = new Manifest();
        manifest.getMainAttributes().put(Attributes.Name.MANIFEST_VERSION, MANIFEST_VERSION);
        manifest.getMainAttributes().put(new Attributes.Name(AGENT_CLASS_PROPERTY), Installer.class.getName());
        manifest.getMainAttributes().put(new Attributes.Name(CAN_REDEFINE_CLASSES), "true");
        manifest.getMainAttributes().put(new Attributes.Name(CAN_RETRANSFORM_CLASSES), "true");
        manifest.getMainAttributes().put(new Attributes.Name(CAN_SET_NATIVE_METHOD_ACCESS_FLAGS), "true");
        JarOutputStream jarOutputStream = new JarOutputStream(new FileOutputStream(agentJar));
        try {
            jarOutputStream.putNextEntry(new JarEntry(Installer.class.getName().replace('.', '/') + ".jar"));
            byte[] buffer = new byte[BUFFER_SIZE];
            int index;
            while ((index = inputStream.read(buffer)) != END_OF_FILE) {
                jarOutputStream.write(buffer, START_INDEX, index);
            }
            jarOutputStream.closeEntry();
        } finally {
            jarOutputStream.close();
        }
    } finally {
        inputStream.close();
    }
    return agentJar;
}
```

这份代码主要就是生成了2个文件，一个是Manifest文件，一个是Installer.class文件，为什么要产生这2个文件？

Manifest文件是 `loadAgent` 方法加载agent.jar的时候，需要去读的一个配置文件，里面有一个属性 `Agent-Class: net.bytebuddy.agent.Installer` 指定了一个类，就是表示要执行这个类的 `agentmain` 方法，这个时候 `loadAgent` 会传入 `Instrumentation inst` 对象，这样在当前的 `ClassLoader` 里面就会得到并且保存这个对象了，用于以后class类文件的改造。

Installer.class文件就是Manifest文件里面指定的class，用于启动agentmain方法的入口类文件，里面就一个 `Instrumentation inst` 成员和一个 `agentmain` 方法

生成了上面的2个文件以后，通过zip方法压缩成一个jar包，保存到临时目录，然后把这个临时目录的绝对地址传入到 `loadAgent` 方法，就算完成了agent的启动。

最后再通过

```
attachmentAccessor.getVirtualMachineType().getDeclaredMethod(DETACH_METHOD_NAME).invoke(virtualMachineInstance);
```

把attach上的jvm进行detach



所以通过上面的一系列步骤，就把 `inst` 对象加入到了当前的 `classloader` 里面，以后再通
过这个 `inst` 对象，就可以对 `ClassFileTransformer` 对象进行操作，完成class文件的修改工
作了。

jvm (/nb/7956989)

举报文章 © 著作权归作者所有



浮云1o__o1 (/u/09e5b9581d54)

写了 14733 字，被 22 人关注，获得了 25 个喜欢 (/u/09e5b9581d54)

+ 关注

长沙java码农

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持

♡ 喜欢 (/sign_in) | 0







更多分享

(http://cwb.assets.jianshu.io/notes/images/7223266



登录 (/sign_in) 后发表评论

评论

智慧如你，不想发表一点想法 (/sign_in)咩~

