# 深入探索 Java 热部署

简介

在 Java 开发领域，热部署一直是一个难以解决的问题，目前的 Java 虚拟机只能实现方法体的修改热部署，对于整个类的结构修改，仍然需要重启虚拟机，对类重新加载才能完成更新操作。对于某些大型的应用来说，每次的重启都需要花费大量的时间成本。虽然 osgi 架构的出现，让模块重启成为可能，但是如果模块之间有调用关系的话，这样的操作依然会让应用出现短暂的功能性休克。本文将探索如何在不破坏 Java 虚拟机现有行为的前提下，实现某个单一类的热部署，让系统无需重启就完成某个类的更新。

类加载的探索

首先谈一下何为热部署（hotswap），热部署是在不重启 Java 虚拟机的前提下，能自动侦测到 class 文件的变化，更新运行时 class 的行为。Java 类是通过 Java 虚拟机加载的，某个类的 class 文件在被 classloader 加载后，会生成对应的 Class 对象，之后就可以创建该类的实例。默认的虚拟机行为只会在启动时加载类，如果后期有一个类需要更新的话，单纯替换编译的 class 文件，Java 虚拟机是不会更新正在运行的 class。如果要实现热部署，最根本的方式是修改虚拟机的源代码，改变 classloader 的加载行为，使虚拟机能监听 class 文件的更新，重新加载 class 文件，这样的行为破坏性很大，为后续的 JVM 升级埋下了一个大坑。

另一种友好的方法是创建自己的 classloader 来加载需要监听的 class，这样就能控制类加载的时机，从而实现热部署。本文将具体探索如何实现这个方案。首先需要了解一下 Java 虚拟机现有的加载机制。目前的加载机制，称为双亲委派，系统在使用一个 classloader 来加载类时，会先询问当前 classloader 的父类是否有能力加载，如果父类无法实现加载操作，才会将任务下放到该 classloader 来加载。这种自上而下的加载方式的好处是，让每个 classloader 执行自己的加载任务，不会重复加载类。但是这种方式却使加载顺序非常难改变，让自定义 classloader 抢先加载需要监听改变的类成为了一个难题。

不过我们可以换一个思路：虽然无法抢先加载该类，但是仍然可以用自定义 classloader 创建一个功能相同的类，让每次实例化的对象都指向这个新的类。当这个类的 class 文件发生改变的时候，再次创建一个更新的类，之后如果系统再次发出实例化请求，创建的对象讲指向这个全新的类。

下面来简单列举一下需要做的工作。

* 创建自定义的 classloader，加载需要监听改变的类，在 class 文件发生改变的时候，重新加载该类。
* 改变创建对象的行为，使他们在创建时使用自定义 classloader 加载的 class。

自定义加载器的实现

自定义加载器仍然需要执行类加载的功能。这里却存在一个问题，同一个类加载器无法同时加载两个相同名称的类，由于不论类的结构如何发生变化，生成的类名不会变，而 classloader 只能在虚拟机停止前销毁已经加载的类，这样 classloader 就无法加载更新后的类了。这里有一个小技巧，让每次加载的类都保存成一个带有版本信息的 class，比如加载 Test.class 时，保存在内存中的类是 Test\_v1.class，当类发生改变时，重新加载的类名是 Test\_v2.class。但是真正执行加载 class 文件创建 class 的 defineClass 方法是一个 native 的方法，修改起来又变得很困难。所以面前还剩一条路，那就是直接修改编译生成的 class 文件。

利用 ASM 修改 class 文件

可以修改字节码的框架有很多，比如 ASM，CGLIB。本文使用的是 ASM。先来介绍一下 class 文件的结构，class 文件包含了以下几类信息，一个是类的基本信息，包含了访问权限信息，类名信息，父类信息，接口信息。第二个是类的变量信息。第三个是方法的信息。ASM 会先加载一个 class 文件，然后严格顺序读取类的各项信息，用户可以按照自己的意愿定义增强组件修改这些信息，最后输出成一个新的 class。

首先看一下如何利用 ASM 修改类信息。

清单 1. 利用 ASM 修改字节码

|  |
| --- |
| ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE\_MAXS);  ClassReader cr = null;  String enhancedClassName = classSource.getEnhancedName();  try {      cr = new ClassReader(new FileInputStream(classSource.getFile()));  } catch (IOException e) {      e.printStackTrace();      return null;  }  ClassVisitor cv = new EnhancedModifier(cw,className.replace(".", "/"),          enhancedClassName.replace(".", "/"));  cr.accept(cv, 0); |

ASM 修改字节码文件的流程是一个责任链模式，首先使用一个 ClassReader 读入字节码，然后利用 ClassVisitor 做个性化的修改，最后利用 ClassWriter 输出修改后的字节码。

之前提过，需要将读取的 class 文件的类名做一些修改，加载成一个全新名字的派生类。这里将之分为了2个步骤。

第一步，先将原来的类变成接口。

清单 2. 重定义的原始类

|  |
| --- |
| public Class<?> redefineClass(String className){         ClassWriter cw = new ClassWriter(ClassWriter. COMPUTE\_MAXS);         ClassReader cr = null;         ClassSource cs = classFiles.get(className);         if(cs==null){             return null;         }         try {             cr = new ClassReader(new FileInputStream(cs.getFile()));         } catch (IOException e) {             e.printStackTrace();             return null;         }         ClassModifier cm = new ClassModifier(cw);         cr.accept(cm, 0);         byte[] code = cw.toByteArray();         return defineClass(className, code, 0, code.length);  } |

首先 load 原始类的 class 文件，此处定义了一个增强组件 ClassModifier，作用是修改原始类的类型，将它转换成接口。原始类的所有方法逻辑都会被去掉。

第二步，生成的派生类都实现这个接口，即原始类，并且复制原始类中的所有方法逻辑。之后如果该类需要更新，会生成一个新的派生类，也会实现这个接口。这样做的目的是不论如何修改，同一个 class 的派生类都有一个共同的接口，他们之间的转换变得对外不透明。

清单 3. 定义一个派生类

   // 在 class 文件发生改变时重新定义这个类

   private Class<?> redefineClass(String className, ClassSource classSource){

       ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE\_MAXS);

       ClassReader cr = null;

       classSource.update();

       String enhancedClassName = classSource.getEnhancedName();

       try {

           cr = new ClassReader(new FileInputStream(classSource.getFile()));

       } catch (IOException e) {

           e.printStackTrace();

           return null;

       }

       EnhancedModifier em = new EnhancedModifier(cw, className.replace(".", "/"),

               enhancedClassName.replace(".", "/"));

       ExtendModifier exm = new ExtendModifier(em, className.replace(".", "/"),

               enhancedClassName.replace(".", "/"));

       cr.accept(exm, 0);

       byte[] code = cw.toByteArray();

       classSource.setByteCopy(code);

       Class<?> clazz = defineClass(enhancedClassName, code, 0, code.length);

       classSource.setClassCopy(clazz);

       return clazz;

}

再次 load 原始类的 class 文件，此处定义了两个增强组件，一个是 EnhancedModifier，这个增强组件的作用是改变原有的类名。第二个增强组件是 ExtendModifier，这个增强组件的作用是改变原有类的父类，让这个修改后的派生类能够实现同一个原始类（此时原始类已经转成接口了）。

自定义 classloader 还有一个作用是监听会发生改变的 class 文件，classloader 会管理一个定时器，定时依次扫描这些 class 文件是否改变。

改变创建对象的行为

Java 虚拟机常见的创建对象的方法有两种，一种是静态创建，直接 new 一个对象，一种是动态创建，通过反射的方法，创建对象。

由于已经在自定义加载器中更改了原有类的类型，把它从类改成了接口，所以这两种创建方法都无法成立。我们要做的是将实例化原始类的行为变成实例化派生类。

对于第一种方法，需要做的是将静态创建，变为通过 classloader 获取 class，然后动态创建该对象。

清单 4. 替换后的指令集所对应的逻辑

|  |
| --- |
| // 原始逻辑    Greeter p = new Greeter();  // 改变后的逻辑    IGreeter p = (IGreeter)MyClassLoader.getInstance().    findClass("com.example.Greeter").newInstance(); |

这里又需要用到 ASM 来修改 class 文件了。查找到所有 new 对象的语句，替换成通过 classloader 的形式来获取对象的形式。

清单 5. 利用 ASM 修改方法体

@Override

public void visitTypeInsn(int opcode, String type) {

    if(opcode==Opcodes.NEW && type.equals(className)){

        List<LocalVariableNode> variables = node.localVariables;

        String compileType = null;

        for(int i=0;i<variables.size();i++){

            LocalVariableNode localVariable = variables.get(i);

            compileType = formType(localVariable.desc);

            if(matchType(compileType)&&!valiableIndexUsed[i]){

                valiableIndexUsed[i] = true;

                break;

            }

        }

    mv.visitMethodInsn(Opcodes.INVOKESTATIC, CLASSLOAD\_TYPE,

        "getInstance", "()L"+CLASSLOAD\_TYPE+";");

    mv.visitLdcInsn(type.replace("/", "."));

    mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, CLASSLOAD\_TYPE,

        "findClass", "(Ljava/lang/String;)Ljava/lang/Class;");

    mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/lang/Class",

        "newInstance", "()Ljava/lang/Object;");

    mv.visitTypeInsn(Opcodes.CHECKCAST, compileType);

    flag = true;

    } else {

        mv.visitTypeInsn(opcode, type);

    }

 }

对于第二种创建方法，需要通过修改 Class.forName()和 ClassLoader.findClass()的行为，使他们通过自定义加载器加载类。

使用 JavaAgent 拦截默认加载器的行为

之前实现的类加载器已经解决了热部署所需要的功能，可是 JVM 启动时，并不会用自定义的加载器加载 classpath 下的所有 class 文件，取而代之的是通过应用加载器去加载。如果在其之后用自定义加载器重新加载已经加载的 class，有可能会出现 LinkageError 的 exception。所以必须在应用启动之前，重新替换已经加载的 class。如果在 jdk1.4 之前，能使用的方法只有一种，改变 jdk 中 classloader 的加载行为，使它指向自定义加载器的加载行为。好在 jdk5.0 之后，我们有了另一种侵略性更小的办法，这就是 JavaAgent 方法，JavaAgent 可以在 JVM 启动之后，应用启动之前的短暂间隙，提供空间给用户做一些特殊行为。比较常见的应用，是利用 JavaAgent 做面向方面的编程，在方法间加入监控日志等。

JavaAgent 的实现很容易，只要在一个类里面，定义一个 premain 的方法。

清单 6. 一个简单的 JavaAgent

|  |
| --- |
| public class ReloadAgent {     public static void premain(String agentArgs, Instrumentation inst){         GeneralTransformer trans = new GeneralTransformer();         inst.addTransformer(trans);     }  } |

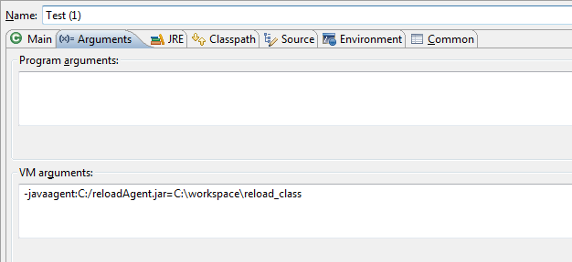
然后编写一个 manifest 文件，将 Premain-Class属性设置成定义一个拥有 premain方法的类名即可。

生成一个包含这个 manifest 文件的 jar 包。

|  |
| --- |
| manifest-Version: 1.0  Premain-Class: com.example.ReloadAgent  Can-Redefine-Classes: true |

最后需要在执行应用的参数中增加 -javaagent参数 , 加入这个 jar。同时可以为 Javaagent增加参数，下图中的参数是测试代码中 test project 的绝对路径。这样在执行应用的之前，会优先执行premain方法中的逻辑，并且预解析需要加载的 class。

图 1. 增加执行参数



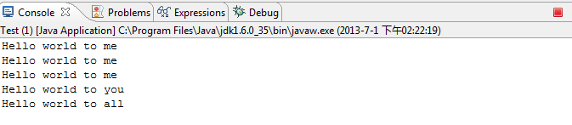
这里利用 JavaAgent替换原始字节码，阻止原始字节码被 Java 虚拟机加载。只需要实现 一个 ClassFileTransformer的接口，利用这个实现类完成 class 替换的功能。

清单 7. 替换 class

|  |
| --- |
| @Override  public byte [] transform(ClassLoader paramClassLoader, String paramString,       Class<?> paramClass, ProtectionDomain paramProtectionDomain,       byte [] paramArrayOfByte) throws IllegalClassFormatException {      String className = paramString.replace("/", ".");      if(className.equals("com.example.Test")){          MyClassLoader cl = MyClassLoader.getInstance();          cl.defineReference(className, "com.example.Greeter");          return cl.getByteCode(className);      }else if(className.equals("com.example.Greeter")){          MyClassLoader cl = MyClassLoader.getInstance();          cl.redefineClass(className);          return cl.getByteCode(className);      }      return null;   } |

至此，所有的工作大功告成，欣赏一下 hotswap 的结果吧。

图 2. Test 执行结果



结束语

解决 hotswap 是个困难的课题，本文解决的仅仅是让新实例化的对象使用新的逻辑，并不能改变已经实例化对象的行为，如果 JVM 能够重新设计 class 的生命周期，支持运行时重新更新一个 class，hotswap 就会成为 Java 的一个闪亮新特性。官方的 JVM 一直没有解决热部署这个问题，可能也是由于无法完全克服其中的诸多难点，希望未来的 Jdk 能解决这个问题，让 Java 应用对于更新更友好，避免不断重启应用浪费的时间。