JavaSE6新特性:I trumentation PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/252/2021_2022_JavaSE6_E6_96_B0_c104_252403.htm 简介 利用 Java 代码，即 java.lang.instrument 做动态 Instrumentation 是 Java SE 5 的新特性，它把 Java 的 instrument 功能从本地代码中解放出来，使之可以用 Java 代码的方式解决问题。使用 Instrumentation，开发者可以构建一个独立于应用程序的代理程序（Agent），用来监测和协助运行在 JVM 上的程序，甚至能够替换和修改某些类的定义。有了这样的功能，开发者就可以实现更为灵活的运行时虚拟机监控和 Java 类操作了，这样的特性实际上提供了一种虚拟机级别支持的 AOP 实现方式，使得开发者无需对 JDK 做任何升级和改动，就可以实现某些 AOP 的功能了。在 Java SE 6 里面，instrumentation 包被赋予了更强大的功能：启动后的 instrument、本地代码（native code）instrument，以及动态改变 classpath 等等。这些改变，意味着 Java 具有了更强的动态控制、解释能力，它使得 Java 语言变得更加灵活多变。 在 Java SE6 里面，最大的改变使运行时的 Instrumentation 成为可能。在 Java SE 5 中，Instrument 要求在运行前利用命令行参数或者系统参数来设置代理类，在实际的运行之中，虚拟机在初始化之时（在绝大多数的 Java 类库被载入之前），instrumentation 的设置已经启动，并在虚拟机中设置了回调函数，检测特定类的加载情况，并完成实际工作。但是在实际的很多的情况下，我们没有办法在虚拟机启动之时就为其设定代理，这样实际上限制了 instrument 的应用。而 Java SE 6 的新特性改变了这种情况，通过 Java Tool API 中的 attach 方式

，我们可以很方便地在运行过程中动态地设置加载代理类，以达到 instrumentation 的目的。 另外，对 native 的 Instrumentation 也是 Java SE 6 的一个崭新的功能，这使以前无法完成的功能 对 native 接口的 instrumentation 可以在 Java SE 6 中，通过一个或者一系列的 prefix 添加而得以完成。 最后，Java SE 6 里的 Instrumentation 也增加了动态添加 class path 的功能。所有这些新的功能，都使得 instrument 包的功能更加丰富，从而使 Java 语言本身更加强大。 Instrumentation 的基本功能和用法“ java.lang.instrument ”包的具体实现，依赖于 JVMTI。JVMTI（Java Virtual Machine Tool Interface）是一套由 Java 虚拟机提供的，为 JVM 相关的工具提供的本地编程接口集合。JVMTI 是从 Java SE 5 开始引入，整合和取代了以前使用的 Java Virtual Machine Profiler Interface (JVMPI) 和 the Java Virtual Machine Debug Interface (JVMDI)，而在 Java SE 6 中，JVMPI 和 JVMDI 已经消失了。JVMTI 提供了一套 ”代理 ” 程序机制，可以支持第三方工具程序以代理的方式连接和访问 JVM，并利用 JVMTI 提供的丰富的编程接口，完成很多跟 JVM 相关的功能。事实上，java.lang.instrument 包的实现，也就是基于这种机制的：在 Instrumentation 的实现当中，存在一个 JVMTI 的代理程序，通过调用 JVMTI 当中 Java 类相关的函数来完成 Java 类的动态操作。除开 Instrumentation 功能外，JVMTI 还在虚拟机内存管理，线程控制，方法和变量操作等等方面提供了大量有价值的函数。 Instrumentation 的最大作用，就是类定义动态改变和操作。在 Java SE 5 及其后续版本当中，开发者可以在一个普通 Java 程序（带有 main 函数的 Java 类）运行时，通过 javaagent 参数指定一个特定的 jar 文件

（包含 Instrumentation 代理）来启动 Instrumentation 的代理程序。 在 Java SE 5 当中，开发者可以让 Instrumentation 代理在 main 函数运行前执行。简要说来就是如下几个步骤： 编写 premain 函数 编写一个 Java 类，包含如下两个方法当中的任何一个 public static void premain(String agentArgs, Instrumentation inst). [1] public static void premain(String agentArgs). [2] 其中，[1] 的优先级比 [2] 高，将会被优先执行（[1] 和 [2] 同时存在时，[2] 被忽略）。 在这个 premain 函数中，开发者可以进行对类的各种操作。 agentArgs 是 premain 函数得到的程序参数，随同“ javaagent ”一起传入。与 main 函数不同的是，这个参数是一个字符串而不是一个字符串数组，如果程序参数有多个，程序将自行解析这个字符串。 Inst 是一个 java.lang.instrument.Instrumentation 的实例，由 JVM 自动传入。java.lang.instrument.Instrumentation 是 instrument 包中定义的一个接口，也是这个包的核心部分，集中了其中几乎所有的功能方法，例如类定义的转换和操作等等。 jar 文件打包 将这个 Java 类打包成一个 jar 文件，并在其中的 manifest 属性当中加入 ” Premain-Class ” 来指定步骤 1 当中编写的那个带有 premain 的 Java类。（可能还需要指定其他属性以开启更多功能） 运行 用如下方式运行带有 Instrumentation 的 Java 程序： java -javaagent:jar文件的位置[=传入premain的参数] 对 Java 类文件的操作，可以理解为对一个 byte 数组的操作（将类文件的二进制字节流读入一个 byte 数组）。开发者可以在“ ClassFileTransformer ”的 transform 方法当中得到，操作并最终返回一个类的定义（一个 byte 数组）。这方面，Apache 的 BCEL 开源项目提供了强有力的支持，读者可以在参考文章

“ Java SE 5 特性 Instrumentation 实践 ” 中看到一个 BCEL 和 Instrumentation 结合的例子。具体的字节码操作并非本文的重点，所以，本文中所举的例子，只是采用简单的类文件替换的方式来演示 Instrumentation 的使用。 下面，我们通过简单的举例，来说明 Instrumentation 的基本使用方法。 首先，我们有一个简单的类，TransClass， 可以通过一个静态方法返回一个整数 1。 public class TransClass { public int getNumber() { return 1. } } 我们运行如下类，可以得到输出 ” 1 “ 。 public class TestMainInJar { public static void main(String[] args) { System.out.println(new TransClass().getNumber()). } } 然后，我们将 TransClass 的 getNumber 方法改成如下: public int getNumber() { return 2. } 再将这个返回 2 的 Java 文件编译成类文件，为了区别开原有的返回 1 的类，我们将返回 2 的这个类文件命名为 TransClass2.class.2。 接下来，我们建立一个 Transformer 类： import java.io.File. import java.io.FileInputStream. import java.io.IOException. import java.io.InputStream. import java.lang.instrument.ClassFileTransformer. import java.lang.instrument.IllegalClassFormatException. import java.security.Pro???棌?? tectionDomain. class Transformer implements ClassFileTransformer { public static final String classNumberReturns2 = "TransClass.class.2". public static byte[] getBytesFromFile(String fileName) { try { // precondition File file = new File(fileName). InputStream is = new FileInputStream(file). long length = file.length(). byte[] bytes = new byte[(int) length]. // Read in the bytes int offset = 0. int numRead = 0. while (offset amp.

(numRead = is.read(bytes, offset, bytes.length - offset)) >= 0) { offset = numRead. } if (offset throw new IOException("Could not completely read file " file.getName()). } is.close(). return bytes. } catch (Exception e) { System.out.println("error occurs in _ClassTransformer!" e.getClass().getName()). return null. } } public byte[] transform(ClassLoader l, String className, Class c, ProtectionDomain pd, byte[] b) throws IllegalClassFormatException { if (!className.equals("TransClass")) { return null. } return getBytesFromFile(classNumberReturns2). } }