概述

本文重点讲述javaagent的具体实现，因为它面向的是我们Java程序员，而且agent都是用Java编写的，不需要太多的C/C++编程基础，不过这篇文章里也会讲到JVMTIAgent(C实现的)，因为javaagent的运行还是依赖于一个特殊的JVMTIAgent。

对于javaagent，或许大家都听过，甚至使用过，常见的用法大致如下：

java -javaagent:myagent.jar=mode=test Test

**相关厂商内容**

[**不可错过的智能时代的大前端**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=4267&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**如何通过使用 AWS对IT资源实现高级别管控，并大规模实现更高级别的安全性？**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=3006&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**解析微信朋友圈的lookalike算法**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=4259&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**百度人脸识别技术在行业应用中有哪些突出表现**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=4309&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**中国技术开放日3月北京站：电商，变革的不仅仅是买卖**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=4305&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

我们通过-javaagent来指定我们编写的agent的jar路径（./myagent.jar），以及要传给agent的参数（mode=test），在启动的时候这个agent就可以做一些我们希望的事了。

javaagent的主要功能如下：

* 可以在加载class文件之前做拦截，对字节码做修改
* 可以在运行期对已加载类的字节码做变更，但是这种情况下会有很多的限制，后面会详细说
* 还有其他一些小众的功能
  + 获取所有已经加载过的类
  + 获取所有已经初始化过的类（执行过clinit方法，是上面的一个子集）
  + 获取某个对象的大小
  + 将某个jar加入到bootstrap classpath里作为高优先级被bootstrapClassloader加载
  + 将某个jar加入到classpath里供AppClassloard去加载
  + 设置某些native方法的前缀，主要在查找native方法的时候做规则匹配

想象一下可以让程序按照我们预期的逻辑去执行，听起来是不是挺酷的。

**JVMTI**

[JVMTI](http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html)全称JVM Tool Interface，是JVM暴露出来的一些供用户扩展的接口集合。JVMTI是基于事件驱动的，JVM每执行到一定的逻辑就会调用一些事件的回调接口（如果有的话），这些接口可以供开发者扩展自己的逻辑。

比如最常见的，我们想在某个类的字节码文件读取之后、类定义之前修改相关的字节码，从而使创建的class对象是我们修改之后的字节码内容，那就可以实现一个回调函数赋给jvmtiEnv（JVMTI的运行时，通常一个JVMTIAgent对应一个jvmtiEnv，但是也可以对应多个）的回调方法集合里的ClassFileLoadHook，这样在接下来的类文件加载过程中都会调用到这个函数中，大致实现如下:，

jvmtiEventCallbacks callbacks;

jvmtiEnv \* jvmtienv = jvmti(agent);

jvmtiError jvmtierror;

memset(&callbacks, 0, sizeof(callbacks));

callbacks.ClassFileLoadHook = &eventHandlerClassFileLoadHook;

jvmtierror = (\*jvmtienv)->SetEventCallbacks( jvmtienv,

&callbacks,

sizeof(callbacks));

**JVMTIAgent**

JVMTIAgent其实就是一个动态库，利用JVMTI暴露出来的一些接口来干一些我们想做、但是正常情况下又做不到的事情，不过为了和普通的动态库进行区分，它一般会实现如下的一个或者多个函数：

JNIEXPORT jint JNICALL

Agent\_OnLoad(JavaVM \*vm, char \*options, void \*reserved);

JNIEXPORT jint JNICALL

Agent\_OnAttach(JavaVM\* vm, char\* options, void\* reserved);

JNIEXPORT void JNICALL

Agent\_OnUnload(JavaVM \*vm);

* Agent\_OnLoad函数，如果agent是在启动时加载的，也就是在vm参数里通过-agentlib来指定的，那在启动过程中就会去执行这个agent里的Agent\_OnLoad函数。
* Agent\_OnAttach函数，如果agent不是在启动时加载的，而是我们先attach到目标进程上，然后给对应的目标进程发送load命令来加载，则在加载过程中会调用Agent\_OnAttach函数。
* Agent\_OnUnload函数，在agent卸载时调用，不过貌似基本上很少实现它。

其实我们每天都在和JVMTIAgent打交道，只是你可能没有意识到而已，比如我们经常使用Eclipse等工具调试Java代码，其实就是利用JRE自带的jdwp agent实现的，只是Eclipse等工具在没让你察觉的情况下将相关参数(类似-agentlib:jdwp=transport=dt\_socket,suspend=y,address=localhost:61349)自动加到程序启动参数列表里了，其中agentlib参数就用来跟要加载的agent的名字，比如这里的jdwp(不过这不是动态库的名字，JVM会做一些名称上的扩展，比如在Linux下会去找libjdwp.so的动态库进行加载，也就是在名字的基础上加前缀lib，再加后缀.so)，接下来会跟一堆相关的参数，将这些参数传给Agent\_OnLoad或者Agent\_OnAttach函数里对应的options。

**javaagent**

说到javaagent，必须要讲的是一个叫做instrument的JVMTIAgent（Linux下对应的动态库是libinstrument.so），因为javaagent功能就是它来实现的，另外instrument agent还有个别名叫JPLISAgent(Java Programming Language Instrumentation Services Agent)，这个名字也完全体现了其最本质的功能：就是专门为Java语言编写的插桩服务提供支持的。

**instrument agent**

instrument agent实现了Agent\_OnLoad和Agent\_OnAttach两方法，也就是说在使用时，agent既可以在启动时加载，也可以在运行时动态加载。其中启动时加载还可以通过类似-javaagent:myagent.jar的方式来间接加载instrument agent，运行时动态加载依赖的是JVM的attach机制（[JVM Attach机制实现](http://lovestblog.cn/blog/2014/06/18/jvm-attach/)），通过发送load命令来加载agent。

instrument agent的核心数据结构如下：

struct \_JPLISAgent {

JavaVM \* mJVM; /\* handle to the JVM \*/

JPLISEnvironment mNormalEnvironment; /\* for every thing but retransform stuff \*/

JPLISEnvironment mRetransformEnvironment;/\* for retransform stuff only \*/

jobject mInstrumentationImpl; /\* handle to the Instrumentation instance \*/

jmethodID mPremainCaller; /\* method on the InstrumentationImpl that does the premain stuff (cached to save lots of lookups) \*/

jmethodID mAgentmainCaller; /\* method on the InstrumentationImpl for agents loaded via attach mechanism \*/

jmethodID mTransform; /\* method on the InstrumentationImpl that does the class file transform \*/

jboolean mRedefineAvailable; /\* cached answer to "does this agent support redefine" \*/

jboolean mRedefineAdded; /\* indicates if can\_redefine\_classes capability has been added \*/

jboolean mNativeMethodPrefixAvailable; /\* cached answer to "does this agent support prefixing" \*/

jboolean mNativeMethodPrefixAdded; /\* indicates if can\_set\_native\_method\_prefix capability has been added \*/

char const \* mAgentClassName; /\* agent class name \*/

char const \* mOptionsString; /\* -javaagent options string \*/

};

struct \_JPLISEnvironment {

jvmtiEnv \* mJVMTIEnv; /\* the JVM TI environment \*/

JPLISAgent \* mAgent; /\* corresponding agent \*/

jboolean mIsRetransformer; /\* indicates if special environment \*/

};

这里解释一下几个重要项：

* mNormalEnvironment：主要提供正常的类transform及redefine功能。
* mRetransformEnvironment：主要提供类retransform功能。
* mInstrumentationImpl：这个对象非常重要，也是我们Java agent和JVM进行交互的入口，或许写过javaagent的人在写`premain`以及`agentmain`方法的时候注意到了有个Instrumentation参数，该参数其实就是这里的对象。
* mPremainCaller：指向`sun.instrument.InstrumentationImpl.loadClassAndCallPremain`方法，如果agent是在启动时加载的，则该方法会被调用。
* mAgentmainCaller：指向`sun.instrument.InstrumentationImpl.loadClassAndCallAgentmain`方法，该方法在通过attach的方式动态加载agent的时候调用。
* mTransform：指向`sun.instrument.InstrumentationImpl.transform`方法。
* mAgentClassName：在我们javaagent的MANIFEST.MF里指定的`Agent-Class`。
* mOptionsString：传给agent的一些参数。
* mRedefineAvailable：是否开启了redefine功能，在javaagent的MANIFEST.MF里设置`Can-Redefine-Classes:true`。
* mNativeMethodPrefixAvailable：是否支持native方法前缀设置，同样在javaagent的MANIFEST.MF里设置`Can-Set-Native-Method-Prefix:true`。
* mIsRetransformer：如果在javaagent的MANIFEST.MF文件里定义了`Can-Retransform-Classes:true`，将会设置mRetransformEnvironment的mIsRetransformer为true。

在启动时加载instrument agent

正如前面“概述”里提到的方式，就是启动时加载instrument agent，具体过程都在`InvocationAdapter.c`的`Agent\_OnLoad`方法里，这里简单描述下过程：

* 创建并初始化JPLISAgent
* 监听VMInit事件，在vm初始化完成之后做下面的事情：
  + 创建InstrumentationImpl对象
  + 监听ClassFileLoadHook事件
  + 调用InstrumentationImpl的`loadClassAndCallPremain`方法，在这个方法里会调用javaagent里MANIFEST.MF里指定的`Premain-Class`类的premain方法
* 解析javaagent里MANIFEST.MF里的参数，并根据这些参数来设置JPLISAgent里的一些内容

在运行时加载instrument agent

在运行时加载的方式，大致按照下面的方式来操作：

VirtualMachine vm = VirtualMachine.attach(pid);

vm.loadAgent(agentPath, agentArgs);

上面会通过JVM的attach机制来请求目标JVM加载对应的agent，过程大致如下：

* 创建并初始化JPLISAgent
* 解析javaagent里MANIFEST.MF里的参数
* 创建InstrumentationImpl对象
* 监听ClassFileLoadHook事件
* 调用InstrumentationImpl的loadClassAndCallAgentmain方法，在这个方法里会调用javaagent里MANIFEST.MF里指定的Agent-Class类的agentmain方法

**instrument agent的ClassFileLoadHook回调实现**

不管是启动时还是运行时加载的instrument agent，都关注着同一个jvmti事件——ClassFileLoadHook，这个事件是在读取字节码文件之后回调时用的，这样可以对原来的字节码做修改，那这里面究竟是怎样实现的呢？

void JNICALL

eventHandlerClassFileLoadHook( jvmtiEnv \* jvmtienv,

JNIEnv \* jnienv,

jclass class\_being\_redefined,

jobject loader,

const char\* name,

jobject protectionDomain,

jint class\_data\_len,

const unsigned char\* class\_data,

jint\* new\_class\_data\_len,

unsigned char\*\* new\_class\_data) {

JPLISEnvironment \* environment = NULL;

environment = getJPLISEnvironment(jvmtienv);

/\* if something is internally inconsistent (no agent), just silently return without touching the buffer \*/

if ( environment != NULL ) {

jthrowable outstandingException = preserveThrowable(jnienv);

transformClassFile( environment->mAgent,

jnienv,

loader,

name,

class\_being\_redefined,

protectionDomain,

class\_data\_len,

class\_data,

new\_class\_data\_len,

new\_class\_data,

environment->mIsRetransformer);

restoreThrowable(jnienv, outstandingException);

}

}

先根据jvmtiEnv取得对应的JPLISEnvironment，因为上面我已经说到其实有两个JPLISEnvironment（并且有两个jvmtiEnv），其中一个是专门做retransform的，而另外一个用来做其他事情，根据不同的用途，在注册具体的ClassFileTransformer时也是分开的，对于作为retransform用的ClassFileTransformer，我们会注册到一个单独的TransformerManager里。

接着调用transformClassFile方法，由于函数实现比较长，这里就不贴代码了，大致意思就是调用InstrumentationImpl对象的transform方法，根据最后那个参数来决定选哪个TransformerManager里的ClassFileTransformer对象们做transform操作。

private byte[]

transform( ClassLoader loader,

String classname,

Class classBeingRedefined,

ProtectionDomain protectionDomain,

byte[] classfileBuffer,

boolean isRetransformer) {

TransformerManager mgr = isRetransformer?

mRetransfomableTransformerManager :

mTransformerManager;

if (mgr == null) {

return null; // no manager, no transform

} else {

return mgr.transform( loader,

classname,

classBeingRedefined,

protectionDomain,

classfileBuffer);

}

}

public byte[]

transform( ClassLoader loader,

String classname,

Class classBeingRedefined,

ProtectionDomain protectionDomain,

byte[] classfileBuffer) {

boolean someoneTouchedTheBytecode = false;

TransformerInfo[] transformerList = getSnapshotTransformerList();

byte[] bufferToUse = classfileBuffer;

// order matters, gotta run 'em in the order they were added

for ( int x = 0; x < transformerList.length; x++ ) {

TransformerInfo transformerInfo = transformerList[x];

ClassFileTransformer transformer = transformerInfo.transformer();

byte[] transformedBytes = null;

try {

transformedBytes = transformer.transform( loader,

classname,

classBeingRedefined,

protectionDomain,

bufferToUse);

}

catch (Throwable t) {

// don't let any one transformer mess it up for the others.

// This is where we need to put some logging. What should go here? FIXME

}

if ( transformedBytes != null ) {

someoneTouchedTheBytecode = true;

bufferToUse = transformedBytes;

}

}

// if someone modified it, return the modified buffer.

// otherwise return null to mean "no transforms occurred"

byte [] result;

if ( someoneTouchedTheBytecode ) {

result = bufferToUse;

}

else {

result = null;

}

return result;

}

以上是最终调到的java代码，可以看到已经调用到我们自己编写的javaagent代码里了，我们一般是实现一个ClassFileTransformer类，然后创建一个对象注册到对应的TransformerManager里。

**Class Transform的实现**

这里说的class transform其实是狭义的，主要是针对第一次类文件加载时就要求被transform的场景，在加载类文件的时候发出ClassFileLoad事件，然后交给instrumenat agent来调用javaagent里注册的ClassFileTransformer实现字节码的修改。

**Class Redefine的实现**

类重新定义，这是Instrumentation提供的基础功能之一，主要用在已经被加载过的类上，想对其进行修改，要做这件事，我们必须要知道两个东西，一个是要修改哪个类，另外一个是想将那个类修改成怎样的结构，有了这两个信息之后就可以通过InstrumentationImpl下面的redefineClasses方法操作了：

public void redefineClasses(ClassDefinition[] definitions) throws ClassNotFoundException {

if (!isRedefineClassesSupported()) {

throw new UnsupportedOperationException("redefineClasses is not supported in this environment");

}

if (definitions == null) {

throw new NullPointerException("null passed as 'definitions' in redefineClasses");

}

for (int i = 0; i < definitions.length; ++i) {

if (definitions[i] == null) {

throw new NullPointerException("element of 'definitions' is null in redefineClasses");

}

}

if (definitions.length == 0) {

return; // short-circuit if there are no changes requested

}

redefineClasses0(mNativeAgent, definitions);

}

在JVM里对应的实现是创建一个VM\_RedefineClasses的VM\_Operation，注意执行它的时候会stop-the-world：

jvmtiError

JvmtiEnv::RedefineClasses(jint class\_count, const jvmtiClassDefinition\* class\_definitions) {

//TODO: add locking

VM\_RedefineClasses op(class\_count, class\_definitions, jvmti\_class\_load\_kind\_redefine);

VMThread::execute(&op);

return (op.check\_error());

} /\* end RedefineClasses \*/

这个过程我尽量用语言来描述清楚，不详细贴代码了，因为代码量实在有点大：

* 挨个遍历要批量重定义的jvmtiClassDefinition
* 然后读取新的字节码，如果有关注ClassFileLoadHook事件的，还会走对应的transform来对新的字节码再做修改
* 字节码解析好，创建一个klassOop对象
* 对比新老类，并要求如下：
  + 父类是同一个
  + 实现的接口数也要相同，并且是相同的接口
  + 类访问符必须一致
  + 字段数和字段名要一致
  + 新增的方法必须是private static/final的
  + 可以删除修改方法
* 对新类做字节码校验
* 合并新老类的常量池
* 如果老类上有断点，那都清除掉
* 对老类做JIT去优化
* 对新老方法匹配的方法的jmethodId做更新，将老的jmethodId更新到新的method上
* 新类的常量池的holer指向老的类
* 将新类和老类的一些属性做交换，比如常量池，methods，内部类
* 初始化新的vtable和itable
* 交换annotation的method、field、paramenter
* 遍历所有当前类的子类，修改他们的vtable及itable

上面是基本的过程，总的来说就是只更新了类里的内容，相当于只更新了指针指向的内容，并没有更新指针，避免了遍历大量已有类对象对它们进行更新所带来的开销。

**Class Retransform的实现**

retransform class可以简单理解为回滚操作，具体回滚到哪个版本，这个需要看情况而定，下面不管那种情况都有一个前提，那就是javaagent已经要求要有retransform的能力了：

* 如果类是在第一次加载的的时候就做了transform，那么做retransform的时候会将代码回滚到transform之后的代码
* 如果类是在第一次加载的的时候没有任何变化，那么做retransform的时候会将代码回滚到最原始的类文件里的字节码
* 如果类已经加载了，期间类可能做过多次redefine(比如被另外一个agent做过)，但是接下来加载一个新的agent要求有retransform的能力了，然后对类做redefine的动作，那么retransform的时候会将代码回滚到上一个agent最后一次做redefine后的字节码

我们从InstrumentationImpl的retransformClasses方法参数看猜到应该是做回滚操作，因为我们只指定了class：

public void retransformClasses(Class<?>[] classes) {

if (!isRetransformClassesSupported()) {

throw new UnsupportedOperationException( "retransformClasses is not supported in this environment");

}

retransformClasses0(mNativeAgent, classes);

}

不过retransform的实现其实也是通过redefine的功能来实现，在类加载的时候有比较小的差别，主要体现在究竟会走哪些transform上，如果当前是做retransform的话，那将忽略那些注册到正常的TransformerManager里的ClassFileTransformer，而只会走专门为retransform而准备的TransformerManager的ClassFileTransformer，不然想象一下字节码又被无声无息改成某个中间态了。

private:

void post\_all\_envs() {

if (\_load\_kind != jvmti\_class\_load\_kind\_retransform) {

// for class load and redefine,

// call the non-retransformable agents

JvmtiEnvIterator it;

for (JvmtiEnv\* env = it.first(); env != NULL; env = it.next(env)) {

if (!env->is\_retransformable() && env->is\_enabled(JVMTI\_EVENT\_CLASS\_FILE\_LOAD\_HOOK)) {

// non-retransformable agents cannot retransform back,

// so no need to cache the original class file bytes

post\_to\_env(env, false);

}

}

}

JvmtiEnvIterator it;

for (JvmtiEnv\* env = it.first(); env != NULL; env = it.next(env)) {

// retransformable agents get all events

if (env->is\_retransformable() && env->is\_enabled(JVMTI\_EVENT\_CLASS\_FILE\_LOAD\_HOOK)) {

// retransformable agents need to cache the original class file

// bytes if changes are made via the ClassFileLoadHook

post\_to\_env(env, true);

}

}

}

**javaagent的其他小众功能**

javaagent除了做字节码上面的修改之外，其实还有一些小功能，有时候还是挺有用的

* 获取所有已经被加载的类：Class[] getAllLoadedClasses();
* 获取所有已经初始化了的类： Class[] getInitiatedClasses(ClassLoader loader);
* 获取某个对象的大小： long getObjectSize(Object objectToSize);
* 将某个jar加入到bootstrap classpath里优先其他jar被加载： void appendToBootstrapClassLoaderSearch(JarFile jarfile);
* 将某个jar加入到classpath里供appclassloard去加载：void appendToSystemClassLoaderSearch(JarFile jarfile);
* 设置某些native方法的前缀，主要在找native方法的时候做规则匹配： void setNativeMethodPrefix(ClassFileTransformer transformer, String prefix)。

作者简介

李嘉鹏，花名寒泉子，使用“[你假笨](http://weibo.com/nijiaben)”的ID混迹网络，蚂蚁金服码农一枚。本科毕业四年多，一直待在支付宝，先后从事过监控系统、框架容器以及性能分析系统等研发工作，其中从事框架容器三年多，主要负责开发支付宝的统一编程框架sofa，2014年下半年开始重点从事性能分析系统的研发工作并于年底加入JVM团队。