[**JAVA深入研究——Method的Invoke方法。**](http://www.cnblogs.com/onlywujun/p/3519037.html)

**http://www.cnblogs.com/onlywujun/p/3519037.html**

　　在写代码的时候，发现Method可以调用子类的对象，但子类即使是改写了的Method，方法名一样，去调用父类的对象也会报错，虽然这是很符合多态的现象，也符合java的动态绑定规范，但还是想弄懂java是如何实现的，就学习了下Method的源代码。

Method的invoke方法

1.先检查 AccessibleObject的override属性是否为true。

AccessibleObject是Method,Field,Constructor的父类，override属性默认为false,可调用setAccessible方法改变，如果设置为true,则表示可以忽略访问权限的限制，直接调用。

2.如果不是ture,则要进行访问权限检测。用Reflection的quickCheckMemberAccess方法先检查是不是public的，如果不是再用Reflection.getCallerClass(1)方法获

得到调用这个方法的Class,然后做是否有权限访问的校验，校验之后缓存一次，以便下次如果还是这个类来调用就不用去做校验了，直接用上次的结果，（很奇怪用这种方式缓存，因为这种方式如果下次换个类来调用的话，就不用会缓存了，而再验证一遍，把这次的结果做为缓存，但上一次的缓存结果就被冲掉了。这是一个很简单的缓冲机制，只适用于一个类的重复调用)。

3.调用MethodAccessor的invoke方法。每个Method对象包含一个root对象，root对象里持有一个MethodAccessor对象。我们获得的Method独享相当于一个root对象的镜像，所有这类Method共享root里的MethodAccessor对象,(这个对象由ReflectionFactory方法生成,ReflectionFactory对象在Method类中是static final的由native方法实例化)。

ReflectionFactory生成MethodAccessor：如果noInflation的属性为true则直接返回MethodAccessorGenerator创建的一个MethodAccessor。

否则返回DelegatingMethodAccessorImpl，并将他与一个NativeMethodAccessorImpl互相引用。但DelegatingMethodAccessorImpl执行invoke方法的时候又委托给NativeMethodAccessorImpl了。

再一步深入

4.NativeMethodAccessorImpl的invkoe方法:

调用natiave方法invoke0执行方法调用.

注意这里有一个计数器numInvocations，每调用一次方法+1,当比 ReflectionFactory.inflationThreshold(15)大的时候,用MethodAccessorGenerator创建一个MethodAccessor,并把之前的DelegatingMethodAccessorImpl引用替换为现在新创建的。下一次DelegatingMethodAccessorImpl就不会再交给NativeMethodAccessorImpl执行了，而是交给新生成的java字节码的MethodAccessor。

MethodAccessorGenerator使用了asm字节码动态加载技术，暂不深入研究。

总结 一个方法可以生成多个Method对象，但只有一个root对象，主要用于持有一个MethodAccessor对象，这个对象也可以认为一个方法只有一个，相当于是static的。因为Method的invoke是交给MethodAccessor执行的，所以我所想要知道的答案在MethodAccessor的invoke中，深入MethodAccessor：

------------------------------------------MethodAccessor---------------------------------

假如有这么一个类A：

public class A {  
　　public void foo(String name) {  
　　　　System.out.println("Hello, " + name);  
　　}  
}

可以编写另外一个类来反射调用A上的方法：

import java.lang.reflect.Method;

public class TestClassLoad {  
　　public static void main(String[] args) throws Exception {  
　　　　Class<?> clz = Class.forName("A");  
　　　　Object o = clz.newInstance();  
　　　　Method m = clz.getMethod("foo", String.class);  
　　　　for (int i = 0; i < 16; i++) {  
　　　　　　m.invoke(o, Integer.toString(i));  
　　　　}  
　　}  
}

注意到TestClassLoad类上不会有对类A的符号依赖——也就是说在加载并初始化TestClassLoad类时不需要关心类A的存在与否，而是等到main()方法执行到调用Class.forName()时才试图对类A做动态加载；这里用的是一个参数版的forName()，也就是使用当前方法所在类的ClassLoader来加载，并且初始化新加载的类。……好吧这个细节跟主题没啥关系。   
  
回到主题。这次我的测试环境是Sun的JDK 1.6.0 update 13 build 03。编译上述代码，并在执行TestClassLoad时加入-XX:+TraceClassLoading参数（或者-verbose:class或者直接-verbose都行），如下：

控制台命令

java -XX:+TraTestClassLoad  ceClassLoading

可以看到输出了一大堆log，把其中相关的部分截取出来如下：

[复制代码](javascript:void(0);)

[Loaded TestClassLoad from file:/D:/temp\_code/test\_java\_classload/]

[Loaded A from file:/D:/temp\_code/test\_java\_classload/]

[Loaded sun.reflect.NativeMethodAccessorImpl from shared objects file]

[Loaded sun.reflect.DelegatingMethodAccessorImpl from shared objects file]

Hello, 0

Hello, 1

Hello, 2

Hello, 3

Hello, 4

Hello, 5

Hello, 6

Hello, 7

Hello, 8

Hello, 9

Hello, 10

Hello, 11

Hello, 12

Hello, 13

Hello, 14

[Loaded sun.reflect.ClassFileConstants from shared objects file]

[Loaded sun.reflect.AccessorGenerator from shared objects file]

[Loaded sun.reflect.MethodAccessorGenerator from shared objects file]

[Loaded sun.reflect.ByteVectorFactory from shared objects file]

[Loaded sun.reflect.ByteVector from shared objects file]

[Loaded sun.reflect.ByteVectorImpl from shared objects file]

[Loaded sun.reflect.ClassFileAssembler from shared objects file]

[Loaded sun.reflect.UTF8 from shared objects file]

[Loaded java.lang.Void from shared objects file]

[Loaded sun.reflect.Label from shared objects file]

[Loaded sun.reflect.Label$PatchInfo from shared objects file]

[Loaded java.util.AbstractList$Itr from shared objects file]

[Loaded sun.reflect.MethodAccessorGenerator$1 from shared objects file]

[Loaded sun.reflect.ClassDefiner from shared objects file]

[Loaded sun.reflect.ClassDefiner$1 from shared objects file]

[Loaded sun.reflect.GeneratedMethodAccessor1 from \_\_JVM\_DefineClass\_\_]

Hello, 15

[复制代码](javascript:void(0);)

可以看到前15次反射调用A.foo()方法并没有什么稀奇的地方，但在第16次反射调用时似乎有什么东西被触发了，导致JVM新加载了一堆类，其中就包括[Loaded sun.reflect.GeneratedMethodAccessor1 from \_\_JVM\_DefineClass\_\_]这么一行。这是哪里来的呢？

先来看看JDK里Method.invoke()是怎么实现的。   
java.lang.reflect.Method：

[复制代码](javascript:void(0);)

public final

class Method extends AccessibleObject implements GenericDeclaration,

Member {

// ...

private volatile MethodAccessor methodAccessor;

// For sharing of MethodAccessors. This branching structure is

// currently only two levels deep (i.e., one root Method and

// potentially many Method objects pointing to it.)

private Method root;

// ...

public Object invoke(Object obj, Object... args)

throws IllegalAccessException, IllegalArgumentException,

InvocationTargetException

{

if (!override) {

if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {

Class caller = Reflection.getCallerClass(1);

Class targetClass = ((obj == null || !Modifier.isProtected(modifiers))

? clazz

: obj.getClass());

boolean cached;

synchronized (this) {

cached = (securityCheckCache == caller)

&& (securityCheckTargetClassCache == targetClass);

}

if (!cached) {

Reflection.ensureMemberAccess(caller, clazz, obj, modifiers);

synchronized (this) {

securityCheckCache = caller;

securityCheckTargetClassCache = targetClass;

}

}

}

}

if (methodAccessor == null) acquireMethodAccessor();

return methodAccessor.invoke(obj, args);

}

// NOTE that there is no synchronization used here. It is correct

// (though not efficient) to generate more than one MethodAccessor

// for a given Method. However, avoiding synchronization will

// probably make the implementation more scalable.

private void acquireMethodAccessor() {

// First check to see if one has been created yet, and take it

// if so

MethodAccessor tmp = null;

if (root != null) tmp = root.getMethodAccessor();

if (tmp != null) {

methodAccessor = tmp;

return;

}

// Otherwise fabricate one and propagate it up to the root

tmp = reflectionFactory.newMethodAccessor(this);

setMethodAccessor(tmp);

}

// ...

}

[复制代码](javascript:void(0);)

　　可以看到Method.invoke()实际上并不是自己实现的反射调用逻辑，而是委托给sun.reflect.MethodAccessor来处理。   
每个实际的Java方法只有一个对应的Method对象作为root，。这个root是不会暴露给用户的，而是每次在通过反射获取Method对象时新创建Method对象把root包装起来再给用户。在第一次调用一个实际Java方法对应得Method对象的invoke()方法之前，实现调用逻辑的MethodAccessor对象还没创建；等第一次调用时才新创建MethodAccessor并更新给root，然后调用MethodAccessor.invoke()真正完成反射调用。   
  
那么MethodAccessor是啥呢？

sun.reflect.MethodAccessor：

public interface MethodAccessor {

/\*\* Matches specification in {@link java.lang.reflect.Method} \*/

public Object invoke(Object obj, Object[] args)

throws IllegalArgumentException, InvocationTargetException;

}

　　可以看到它只是一个单方法接口，其invoke()方法与Method.invoke()的对应。

创建MethodAccessor实例的是ReflectionFactory。

sun.reflect.ReflectionFactory：

[复制代码](javascript:void(0);)

public class ReflectionFactory {

private static boolean initted = false;

// ...

//

// "Inflation" mechanism. Loading bytecodes to implement

// Method.invoke() and Constructor.newInstance() currently costs

// 3-4x more than an invocation via native code for the first

// invocation (though subsequent invocations have been benchmarked

// to be over 20x faster). Unfortunately this cost increases

// startup time for certain applications that use reflection

// intensively (but only once per class) to bootstrap themselves.

// To avoid this penalty we reuse the existing JVM entry points

// for the first few invocations of Methods and Constructors and

// then switch to the bytecode-based implementations.

//

// Package-private to be accessible to NativeMethodAccessorImpl

// and NativeConstructorAccessorImpl

private static boolean noInflation = false;

private static int inflationThreshold = 15;

// ...

/\*\* We have to defer full initialization of this class until after

the static initializer is run since java.lang.reflect.Method's

static initializer (more properly, that for

java.lang.reflect.AccessibleObject) causes this class's to be

run, before the system properties are set up. \*/

private static void checkInitted() {

if (initted) return;

AccessController.doPrivileged(new PrivilegedAction() {

public Object run() {

// Tests to ensure the system properties table is fully

// initialized. This is needed because reflection code is

// called very early in the initialization process (before

// command-line arguments have been parsed and therefore

// these user-settable properties installed.) We assume that

// if System.out is non-null then the System class has been

// fully initialized and that the bulk of the startup code

// has been run.

if (System.out == null) {

// java.lang.System not yet fully initialized

return null;

}

String val = System.getProperty("sun.reflect.noInflation");

if (val != null && val.equals("true")) {

noInflation = true;

}

val = System.getProperty("sun.reflect.inflationThreshold");

if (val != null) {

try {

inflationThreshold = Integer.parseInt(val);

} catch (NumberFormatException e) {

throw (RuntimeException)

new RuntimeException("Unable to parse property sun.reflect.inflationThreshold").

initCause(e);

}

}

initted = true;

return null;

}

});

}

// ...

public MethodAccessor newMethodAccessor(Method method) {

checkInitted();

if (noInflation) {

return new MethodAccessorGenerator().

generateMethod(method.getDeclaringClass(),

method.getName(),

method.getParameterTypes(),

method.getReturnType(),

method.getExceptionTypes(),

method.getModifiers());

} else {

NativeMethodAccessorImpl acc =

new NativeMethodAccessorImpl(method);

DelegatingMethodAccessorImpl res =

new DelegatingMethodAccessorImpl(acc);

acc.setParent(res);

return res;

}

}

}

[复制代码](javascript:void(0);)

　　这里就可以看到有趣的地方了。如注释所述，实际的MethodAccessor实现有两个版本，一个是Java实现的，另一个是native code实现的。Java实现的版本在初始化时需要较多时间，但长久来说性能较好；native版本正好相反，启动时相对较快，但运行时间长了之后速度就比不过Java版了。这是HotSpot的优化方式带来的性能特性，同时也是许多虚拟机的共同点：跨越native边界会对优化有阻碍作用，它就像个黑箱一样让虚拟机难以分析也将其内联，于是运行时间长了之后反而是托管版本的代码更快些。   
　　为了权衡两个版本的性能，Sun的JDK使用了“inflation”的技巧：让Java方法在被反射调用时，开头若干次使用native版，等反射调用次数超过阈值时则生成一个专用的MethodAccessor实现类，生成其中的invoke()方法的字节码，以后对该Java方法的反射调用就会使用Java版。   
Sun的JDK是从1.4系开始采用这种优化的。

　　PS.可以在启动命令里加上-D*sun.reflect.noInflation*=true，就会RefactionFactory的noInflation属性就变成true了，这样不用等到15调用后，程序一开始就会用java版的MethodAccessor了。

　　上面看到了ReflectionFactory.newMethodAccessor()生产MethodAccessor的逻辑，在“开头若干次”时用到的DelegatingMethodAccessorImpl代码如下：

sun.reflect.DelegatingMethodAccessorImpl：

[复制代码](javascript:void(0);)

/\*\* Delegates its invocation to another MethodAccessorImpl and can

change its delegate at run time. \*/

class DelegatingMethodAccessorImpl extends MethodAccessorImpl {

private MethodAccessorImpl delegate;

DelegatingMethodAccessorImpl(MethodAccessorImpl delegate) {

setDelegate(delegate);

}

public Object invoke(Object obj, Object[] args)

throws IllegalArgumentException, InvocationTargetException

{

return delegate.invoke(obj, args);

}

void setDelegate(MethodAccessorImpl delegate) {

this.delegate = delegate;

}

}

[复制代码](javascript:void(0);)

　　这是一个间接层，方便在native与Java版的MethodAccessor之间实现切换。

然后下面就是native版MethodAccessor的Java一侧的声明:

sun.reflect.NativeMethodAccessorImpl：

[复制代码](javascript:void(0);)

/\*\* Used only for the first few invocations of a Method; afterward,

switches to bytecode-based implementation \*/

class NativeMethodAccessorImpl extends MethodAccessorImpl {

private Method method;

private DelegatingMethodAccessorImpl parent;

private int numInvocations;

NativeMethodAccessorImpl(Method method) {

this.method = method;

}

public Object invoke(Object obj, Object[] args)

throws IllegalArgumentException, InvocationTargetException

{

if (++numInvocations > ReflectionFactory.inflationThreshold()) {

MethodAccessorImpl acc = (MethodAccessorImpl)

new MethodAccessorGenerator().

generateMethod(method.getDeclaringClass(),

method.getName(),

method.getParameterTypes(),

method.getReturnType(),

method.getExceptionTypes(),

method.getModifiers());

parent.setDelegate(acc);

}

return invoke0(method, obj, args);

}

void setParent(DelegatingMethodAccessorImpl parent) {

this.parent = parent;

}

private static native Object invoke0(Method m, Object obj, Object[] args);

}

[复制代码](javascript:void(0);)

　　每次NativeMethodAccessorImpl.invoke()方法被调用时，都会增加一个调用次数计数器，看超过阈值没有；一旦超过，则调用MethodAccessorGenerator.generateMethod()来生成Java版的MethodAccessor的实现类，并且改变DelegatingMethodAccessorImpl所引用的MethodAccessor为Java版。后续经由DelegatingMethodAccessorImpl.invoke()调用到的就是Java版的实现了。

注意到关键的invoke0()方法是个native方法。它在HotSpot VM里是由JVM\_InvokeMethod()函数所支持的：

由C编写

JNIEXPORT jobject JNICALL Java\_sun\_reflect\_NativeMethodAccessorImpl\_invoke0

(JNIEnv \*env, jclass unused, jobject m, jobject obj, jobjectArray args)

{

return JVM\_InvokeMethod(env, m, obj, args);

}

[复制代码](javascript:void(0);)

JVM\_ENTRY(jobject, JVM\_InvokeMethod(JNIEnv \*env, jobject method, jobject obj, jobjectArray args0))

JVMWrapper("JVM\_InvokeMethod");

Handle method\_handle;

if (thread->stack\_available((address) &method\_handle) >= JVMInvokeMethodSlack) {

method\_handle = Handle(THREAD, JNIHandles::resolve(method));

Handle receiver(THREAD, JNIHandles::resolve(obj));

objArrayHandle args(THREAD, objArrayOop(JNIHandles::resolve(args0)));

oop result = Reflection::invoke\_method(method\_handle(), receiver, args, CHECK\_NULL);

jobject res = JNIHandles::make\_local(env, result);

if (JvmtiExport::should\_post\_vm\_object\_alloc()) {

oop ret\_type = java\_lang\_reflect\_Method::return\_type(method\_handle());

assert(ret\_type != NULL, "sanity check: ret\_type oop must not be NULL!");

if (java\_lang\_Class::is\_primitive(ret\_type)) {

// Only for primitive type vm allocates memory for java object.

// See box() method.

JvmtiExport::post\_vm\_object\_alloc(JavaThread::current(), result);

}

}

return res;

} else {

THROW\_0(vmSymbols::java\_lang\_StackOverflowError());

}

JVM\_END

[复制代码](javascript:void(0);)

其中的关键又是Reflection::invoke\_method()：

[复制代码](javascript:void(0);)

// This would be nicer if, say, java.lang.reflect.Method was a subclass

// of java.lang.reflect.Constructor

oop Reflection::invoke\_method(oop method\_mirror, Handle receiver, objArrayHandle args, TRAPS) {

oop mirror = java\_lang\_reflect\_Method::clazz(method\_mirror);

int slot = java\_lang\_reflect\_Method::slot(method\_mirror);

bool override = java\_lang\_reflect\_Method::override(method\_mirror) != 0;

objArrayHandle ptypes(THREAD, objArrayOop(java\_lang\_reflect\_Method::parameter\_types(method\_mirror)));

oop return\_type\_mirror = java\_lang\_reflect\_Method::return\_type(method\_mirror);

BasicType rtype;

if (java\_lang\_Class::is\_primitive(return\_type\_mirror)) {

rtype = basic\_type\_mirror\_to\_basic\_type(return\_type\_mirror, CHECK\_NULL);

} else {

rtype = T\_OBJECT;

}

instanceKlassHandle klass(THREAD, java\_lang\_Class::as\_klassOop(mirror));

methodOop m = klass->method\_with\_idnum(slot);

if (m == NULL) {

THROW\_MSG\_0(vmSymbols::java\_lang\_InternalError(), "invoke");

}

methodHandle method(THREAD, m);

return invoke(klass, method, receiver, override, ptypes, rtype, args, true, THREAD);

}

[复制代码](javascript:void(0);)

　　再下去就深入到HotSpot VM的内部了，本文就在这里打住吧。有同学有兴趣深究的话以后可以再写一篇讨论native版的实现。

　　回到Java的一侧。MethodAccessorGenerator长啥样呢？由于代码太长，这里就不完整贴了，有兴趣的可以到OpenJDK 6的Mercurial仓库看：[OpenJDK 6 build 17的MethodAccessorGenerator](http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/b202c79afde3/src/share/classes/sun/reflect/MethodAccessorGenerator.java)。它的基本工作就是在内存里生成新的专用Java类，并将其加载。就贴这么一个方法：

[复制代码](javascript:void(0);)

private static synchronized String generateName(boolean isConstructor,

boolean forSerialization)

{

if (isConstructor) {

if (forSerialization) {

int num = ++serializationConstructorSymnum;

return "sun/reflect/GeneratedSerializationConstructorAccessor" + num;

} else {

int num = ++constructorSymnum;

return "sun/reflect/GeneratedConstructorAccessor" + num;

}

} else {

int num = ++methodSymnum;

return "sun/reflect/GeneratedMethodAccessor" + num;

}

}

[复制代码](javascript:void(0);)

　　去阅读源码的话，可以看到MethodAccessorGenerator是如何一点点把Java版的MethodAccessor实现类生产出来的。也可以看到GeneratedMethodAccessor+数字这种名字是从哪里来的了，就在上面的generateName()方法里。   
对本文开头的例子的A.foo()，生成的Java版MethodAccessor大致如下：

[复制代码](javascript:void(0);)

package sun.reflect;

public class GeneratedMethodAccessor1 extends MethodAccessorImpl {

public GeneratedMethodAccessor1() {

super();

}

public Object invoke(Object obj, Object[] args)

throws IllegalArgumentException, InvocationTargetException {

// prepare the target and parameters

if (obj == null) throw new NullPointerException();

try {

A target = (A) obj;

if (args.length != 1) throw new IllegalArgumentException();

String arg0 = (String) args[0];

} catch (ClassCastException e) {

throw new IllegalArgumentException(e.toString());

} catch (NullPointerException e) {

throw new IllegalArgumentException(e.toString());

}

// make the invocation

try {

target.foo(arg0);

} catch (Throwable t) {

throw new InvocationTargetException(t);

}

}

}

[复制代码](javascript:void(0);)

　　就反射调用而言，这个invoke()方法非常干净（然而就“正常调用”而言这额外开销还是明显的）。注意到参数数组被拆开了，把每个参数都恢复到原本没有被Object[]包装前的样子，然后对目标方法做正常的invokevirtual调用。由于在生成代码时已经循环遍历过参数类型的数组，生成出来的代码里就不再包含循环了。

　　至此找到我的答案了，因为MethodAccessor会做强制类型转换再进行方法调用，但父类强制转化成子类的的时候就会报错类型不匹配错误了，所以如果变量的引用声明是父但实际指向的对象是子，那么这种调用也是可以的。

----------------------------------------------------------题外话------------------------------------------  
　　当该反射调用成为热点时，它甚至可以被内联到靠近Method.invoke()的一侧，大大降低了反射调用的开销。而native版的反射调用则无法被有效内联，因而调用开销无法随程序的运行而降低。   
　　虽说Sun的JDK这种实现方式使得反射调用方法成本比以前降低了很多，但Method.invoke()本身要用数组包装参数；而且每次调用都必须检查方法的可见性（在Method.invoke()里），也必须检查每个实际参数与形式参数的类型匹配性（在NativeMethodAccessorImpl.invoke0()里或者生成的Java版MethodAccessor.invoke()里）；而且Method.invoke()就像是个独木桥一样，各处的反射调用都要挤过去，在调用点上收集到的类型信息就会很乱，影响内联程序的判断，使得Method.invoke()自身难以被内联到调用方。   
　　相比之下JDK7里新的MethodHandler则更有潜力，在其功能完全实现后能达到比普通反射调用方法更高的性能。在使用MethodHandle来做反射调用时，MethodHandle.invoke()的形式参数与返回值类型都是准确的，所以只需要在链接方法的时候才需要检查类型的匹配性，而不必在每次调用时都检查。而且MethodHandle是不可变值，在创建后其内部状态就不会再改变了；JVM可以利用这个知识而放心的对它做激进优化，例如将实际的调用目标内联到做反射调用的一侧。

　　本来Java的安全机制使得不同类之间不是任意信息都可见，但Sun的JDK里开了个口，有一个标记类专门用于开后门：

[复制代码](javascript:void(0);)

package sun.reflect;

/\*\* <P> MagicAccessorImpl (named for parity with FieldAccessorImpl and

others, not because it actually implements an interface) is a

marker class in the hierarchy. All subclasses of this class are

"magically" granted access by the VM to otherwise inaccessible

fields and methods of other classes. It is used to hold the code

for dynamically-generated FieldAccessorImpl and MethodAccessorImpl

subclasses. (Use of the word "unsafe" was avoided in this class's

name to avoid confusion with {@link sun.misc.Unsafe}.) </P>

<P> The bug fix for 4486457 also necessitated disabling

verification for this class and all subclasses, as opposed to just

SerializationConstructorAccessorImpl and subclasses, to avoid

having to indicate to the VM which of these dynamically-generated

stub classes were known to be able to pass the verifier. </P>

<P> Do not change the name of this class without also changing the

VM's code. </P> \*/

class MagicAccessorImpl {

}

[复制代码](javascript:void(0);)

　　那个"\_\_JVM\_DefineClass\_\_"的来源是这里：   
src/share/vm/prims/jvm.cpp

[复制代码](javascript:void(0);)

// common code for JVM\_DefineClass() and JVM\_DefineClassWithSource()

// and JVM\_DefineClassWithSourceCond()

static jclass jvm\_define\_class\_common(JNIEnv \*env, const char \*name,

jobject loader, const jbyte \*buf,

jsize len, jobject pd, const char \*source,

jboolean verify, TRAPS) {

if (source == NULL) source = "\_\_JVM\_DefineClass\_\_";

[复制代码](javascript:void(0);)

P.S. log里的"shared objects file",其实就是rt.jar,为什么要这么显示，Stack OverFlow上有这样的回答：

　　This is Class Data Sharing. When running the Sun/Oracle Client HotSpot and sharing enable (either -Xshare:auto which is the default, or -Xshare:on), the classes.jsa file is memory mapped. This file contains a number of classes (listed in the classlist file) in internal representation suitable for the exact configuration of the machine running it. The idea is that the classes can be loaded quickly, getting the the JVM up faster. Soon enough a class not covered will be hit, and rt.jar will need to be opened and classes loaded conventionally as required.

　　不能很好理解，大概理解就是所有jvm共享，并可以快速加载里面的class.有英文好的朋友可以留言帮助下。

P.S java内联函数

C++是否为内联函数由自己决定，Java由编译器决定。内联函数就是指函数在被调用的地方直接展开，编译器在调用时不用像一般函数那样，参数压栈，返回时参数出栈以及资源释放等，这样提高了程序执行速度。

Java不支持直接声明为内联函数的，如果想让他内联，则是由编译器说了算，你只能够向编译器提出请求。

final除了不能被override外，还可能实现内联。如果函数为private，则也可能是内联的。

总的来说，一般的函数都不会被当做内联函数，只有声明了final后，编译器才会考虑是不是要把你的函数变成内联函数。

内联不一定好，当被指定为内联的方法体很大时，展开的开销可能就已经超过了普通函数调用调用的时间，引入了内联反而降低了性能，因为在选择这个关键字应该慎重些，不过，在以后高版本的JVM中，在处理内联时做出了优化，它会根据方法的规模来确定是否展开调用。