**Java 反射**

**Class类**

Java程序在运行时，Java运行时系统一直对所有的对象进行所谓的运行时类型标识，即所谓的**RTTI**(多态是基于RTTI实现的)。这项信息纪录了每个对象所属的类。虚拟机通常使用运行时类型信息选准正确方法去执行，用来保存这些类型信息的类是Class类。Class类封装一个对象和接口运行时的状态，当装载类时，Class类型的对象自动创建。

Class类没有公共的构造方法，Class对象是在类加载的时候由Java虚拟机以及通过调用类加载器中的 defineClass 方法自动构造的，因此不能显式地声明一个Class对象。一个类被加载到内存并供我们使用需要经历如下三个阶段：

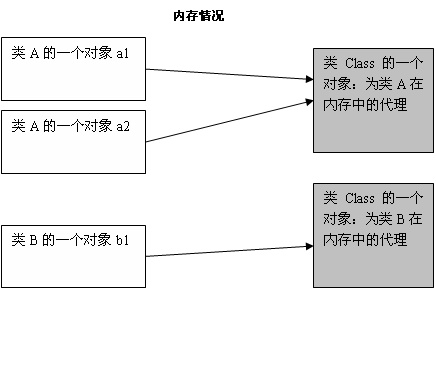
1. 加载，这是由类加载器（ClassLoader）执行的。通过一个类的全限定名来获取其定义的二进制字节流（Class字节码），将这个字节流所代表的静态存储结构转化为方法去的运行时数据接口，根据字节码在java堆中生成一个代表这个类的java.lang.Class对象。
2. 链接。在链接阶段将验证Class文件中的字节流包含的信息是否符合当前虚拟机的要求，为静态域分配存储空间并设置类变量的初始值（默认的零值），并且如果必需的话，将常量池中的符号引用转化为直接引用。
3. 初始化。到了此阶段，才真正开始执行类中定义的java程序代码。用于执行该类的静态初始器和静态初始块，如果该类有父类的话，则优先对其父类进行初始化。

所有的类都是在对其第一次使用时，动态加载到JVM中的（懒加载）。当程序创建第一个对类的静态成员的引用时，就会加载这个类。使用new创建类对象的时候也会被当作对类的静态成员的引用。因此java程序程序在它开始运行之前并非被完全加载，其各个类都是在必需时才加载的。这一点与许多传统语言都不同。动态加载使能的行为，在诸如C++这样的静态加载语言中是很难或者根本不可能复制的。

在类加载阶段，类加载器首先检查这个类的Class对象是否已经被加载。如果尚未加载，默认的类加载器就会根据类的全限定名查找.class文件。在这个类的字节码被加载时，它们会接受验证，以确保其没有被破坏，并且不包含不良java代码。一旦某个类的Class对象被载入内存，我们就可以它来创建这个类的所有对象。

**原理**

在object这个类中有一个方法：getclass().这个方法是用来取得该类已经被实例化了的对象的该类的引用，这个引用指向的是Class类的对象。我们自己无法生成一个Class对象（构造函数为private)，而 这个Class类的对象是在当各类被调入时，由 Java 虚拟机自动创建 Class 对象，或通过类装载器中的 defineClass 方法生成。我们生成的对象都会有个字段记录该对象所属类在CLass类的对象的所在位置。



**获取一个Class类对象**

**forName**

public class shapes{}

Class obj = Class.forName(“shapes”;)

**getClass**

public class shapes{}

shapes s1= new shapes();

Class obj = s1.getClass();

Class obj1 = s1.getSuperClass(); //获取shapes 类的父类的类型

**使用类的字面常量**

在编译时就会受到检查(因此不需要置于try语句块中)。 该方式不会自动地初始化该Class对象（这点和Class.forName方法不同）。类对象的初始化阶段被延迟到了对静态方法或者非常数静态域首次引用时才执行。

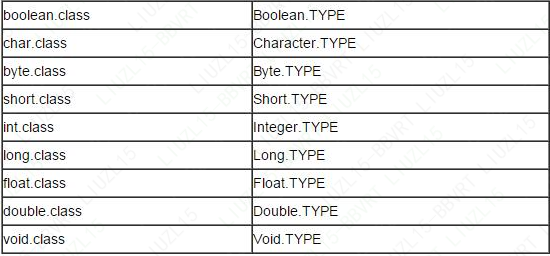
Class obj = String.class;

Class obj1 = int.class;

**注意**

使用这种办法生成Class类对象时，不会使JVM自动加载该类(如String类)。而其他办法会使得JVM初始化该类。

包装类中有一个字段TYPE，TYPE字段是一个引用，指向对应的基本数据类型的Class对象。如



说明：

如果一个字段被 static final 修饰，称为“编译时常量”，在调用这个字段的时候是不会对该类进行初始化的。因为被 static 和 final 修饰的字段，在编译时期就把结果放入了常量池中。

**特别说明**

其实对于任意一个Class对象，都需要由它的**类加载器**和这个**类本身**一同确定其在就Java虚拟机中的唯一性，也就是说，即使两个Class对象来源于同一个Class文件，只要加载它们的类加载器不同，那这两个Class对象就必定不相等。这里的“相等”包括了代表类的Class对象的equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用instanceof关键字对对象所属关系的判定结果。所以在java虚拟机中使用双亲委派模型来组织类加载器之间的关系，来保证Class对象的唯一性。

**使用Class类的对象生成目标类的实例**

**生成不精确的object实例**

获取一个Class类的对象后，可以用 newInstance() 函数来生成目标类的一个实例。然而，该函数并不能直接生成目标类的实例，只能生成object类的实例

Class obj= Class.forName(“shapes”);

Object shapesInstance = obj.newInstance();

**使用泛化Class引用生成带类型的目标类型**

Class<shapes> obj = shapes.class;

shapes newShape = obj.newInstance();

因为有了类型限制，所以使用泛化Class语法的对象引用不能指向别的类

Class obj1 = int.class;

Class<Integer> obj2 = int.class;

obj1 = double.class;

//obj2 = double.class; //这一行代码是非法的，obj2不能改指向别的类了

**有一个灵活的用法，使得可以用Class的对象指向基类的任何子类**。

Class<?> obj = int.class;

obj = double.class;

obj = shapes.class;

**使用这种泛型语法来构建你手头有的一个Class类的对象的基类对象时，必须采用以下的特殊语法**

public class shapes{}

class round extends shapes{}

Class<round> rclass = round.class;

Class<? super round> sclass = rclass.getSuperClass();

//Class<shapes> sclass = rclass.getSuperClass(); //**错误，必须使用Class<? super round>**

**Class 的方法**

**forName()**

(1)获取Class对象的一个引用，但引用的类还没有加载(该类的第一个对象没有生成)就加载了这个类。

(2)为了产生Class引用，forName()立即就进行了初始化。

**Object-getClass()**

获取Class对象的一个引用，返回表示该对象的实际类型的Class引用。

**getName()**

取全限定的类名(包括包名)，即类的完整名字。

com.cry.Test$inner

[Ljava.lang.String;

**getSimpleName()**

获取类名(不包括包名)

**getCanonicalName()**

返回更容易理解的表示，主要用于输出(toString)或log打印，大多数情况下和getName一样，但是在内部类、数组等类型的表示形式就不同了。

com.cry.Test.inner

java.lang.String[]

**isInterface()**

判断Class对象是否是表示一个接口

**getInterfaces()**

返回Class对象数组，表示Class对象所引用的类所实现的所有接口。

**getSupercalss()**

返回Class对象，表示Class对象所引用的类所继承的直接基类。应用该方法可在运行时发现一个对象完整的继承结构。

**newInstance()**

返回一个Oject对象，是实现“虚拟构造器”的一种途径。使用该方法创建的类，必须带有无参的构造器。

**getFields()**

获得某个类的所有的公共（public）的字段，包括继承自父类的所有公共字段。 类似的还有**getMethods**和**getConstructors**。

**getDeclaredFields**

获得某个类的自己声明的字段，即包括public、private和proteced，默认但是不包括父类声明的任何字段。类似的还有**getDeclaredMethods**和**getDeclaredConstructors**。

**getConstructors()**

返回每一个声明为公有的（Public）构造方法。如果知道要访问的构造方法的方法参数类型，可以用下面的方法获取指定的构造方法。如：

Constructor constructor = aClass.getConstructor(new Class[]{String.class})；

**概念**

JAVA反射机制

JAVA反射机制是在运行状态中，对于任意一个实体类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

程序运行时，允许改变程序结构或变量类型。

**使用**

**获取类的构造方法**

1. 获取类对象
2. 获取构造方法
   1. 获取所有 public 构造方法

Constructor[] constructors = mClass.getConstructors();

* 1. 获取本类所有构造方法

Constructor constructor = mClass.getDeclaredConstructors();

1. 获取构造方法参数

Class[] parameterTypes = constructor.getParameterTypes();

1. 实例化一个类，实参和形参必须一一对应

Constructor constructor = MyObject.class.getConstructor

MyObject myObject =(MyObject) constructor.newInstance(“constructor-arg1”);

**获取类的所有变量信息**

1. 获取并输出类的名称

Class mClass = SonClass.class;

1. 获取所有变量
   1. 获取所有public访问权限得到变量

Field[] fields = mClass.getFields();

* 1. 获取所有本类声明的变量（任何访问权限）

Field[] fields = mClass.getDeclaredFields();

//获取指定变量，参数为变量名

Field field = mClass.getDeclaredField(“xxx”);

1. 遍历变量并输出变量信息

for(Field field:fields){

//获取访问权限

int modifiers = field.getModifiers();

//变量类型

field.getType();

//变量名

field.getName();

}

**说明**

getFields() 与 getDeclaredFields()之间的区别

1. getFields() 方法，输出 SonClass 类以及其所继承的父类( 包括 FatherClass 和 Object ) 的 public 方法。**注：Object 类中没有成员变量，所以没有输出**。
2. getDeclaredFields() ， 输出 SonClass 类的所有成员变量，不问访问权限。

**获取类的所有方法信息**

1. 获取所有 public 访问权限的方法（本类及其父类）

Method[] methods = mClass.getMethods();

1. 获取所有本类的方法（任何访问权限）

Method[] methods = mClass.getDeclaredMethods();

//获取指定方法，参数为方法名和形参类型

Method method = mClass.getDeclaredMethod(“xxxx”,new Class[]{String.class});

或

Method method = mClass.getDeclaredMethod(“xxxx”, String.class);

1. 遍历所有方法

//获取方法的访问权限

int modifiers = method.getModifiers();

//获取返回值类型和名称

Class returnType = method.getRuturnType();

String typeName = returnType.getName();

//获取方法的所有参数

Parameter[] parameters = method.getParameters();

String paraTypeName = parameter.getType().getName();

//获取方法抛出的异常

Class[] exceptionTypes = method.getExceptionTypes();

String excepTypeName = exceptionType.getName();

**访问或操作类的私有变量和方法**

**访问私有方法**

1. 获取指定的私有方法

Method privateMethod =mClass.getDeclaredMethod("privateMethod", String.class, int.class);

方法参数也可以这么写：new Class[]{String,class,int.class}，没有参数就是null。

1. 获取私有方法的访问权（只是获取访问权，并不是修改实际权限）

//**如果不加会报异常 IllegalAccessException**

privateMethod.setAccessible(true);

1. 使用invoke 反射调用私有方法

//testClass 要操作的对象

//后面两个参数传实参

privateMethod.invoke(testClass,”Java Reflect”,666);

**修改私有变量**

1. 获取私有变量

Field privateField = mClass.getDeclaredField(“XXX”);

1. 获取私有变量的访问权

privateField.setAccessible(true);

1. 获取私有变量

testClass.getXXX();

1. 修改私有变量

//调用 set(object,value) 修改变量的值为 YYY

privateField.set(testClass,”YYY”);

**修改私有常量**

常量是指使用 final 修饰符修饰的成员属性，与变量的区别就在于有无 final 关键字修饰。Java 虚拟机（JVM）在编译 .java 文件得到 .class 文件时，会优化我们的代码以提升效率。其中一个优化就是：JVM 在编译阶段会把引用常量的代码替换成具体的常量值。但是，**并不是所有常量都会优化**。经测试对于 int 、long 、boolean 以及 String 这些**基本类型 JVM 会优化**，而对于 Integer 、Long 、Boolean 这种**包装类型，或者其他诸如 Date 、Object 类型**则不会被优化。

**对于基本类型的静态常量，JVM在编译阶段会把引用此常量的代码换成具体的常量值。**

针对这种问题，进行修改。

**方法一**

java允许我们声明常量时不赋值，但必须在构造函数中赋值。如：

public class TestClass{

private final String FINAL\_VALUE;

//构造函数内为常量赋值

public TestClass(){

this.FINAL\_VALUE =”FINAL”;

}

//………

}

执行修改

Field finalField = mClass.getDeclaredField(“FINAL\_VALUE”);

finalField.setAccessible(true);

//before

finalField.get(testClass);

//修改

finalField.set(testClass,”Modified”);

//after

finalField.get(testClass);

///// **修改成功**////

**说明**：

将赋值放在构造函数中，构造函数是我们运行时 new 对象才会调用的，所以不会直接为常量赋值，在编译阶段将 getFinalValue() 方法优化为返回常量值，而是指向 FINAL\_VALUE，这样在运行阶段通过反射修改的值就有意义了。

**方法二**

将声明常量的语句改为使用三目表达式赋值：

private final String FIANL\_VALUE =null == null ?”FINAL”:null;

三目表达式是在运行时刻计算出来的，编译时刻不会计算，也就不会被优化。

**反射原理**

假如有这么一个类A：

Java代码 收藏代码

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。

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：

**java -XX:+TraceClassLoading TestClassLoed**

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：（完整的log可以从附件下载）

[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

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：

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);

}

// ...

}

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：

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;

}

}

}

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，主要作者是Ken Russell

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：

Java代码 收藏代码

/\*\* 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;

}

}

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：

Java代码 收藏代码

/\*\* 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);

}

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：

C代码 收藏代码

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);

}

**C++代码**

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

其中的关键又是Reflection::invoke\_method()：

C++代码

// 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);

}

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。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：

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;

}

}

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：

Java代码 收藏代码

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);

}

}

}

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。

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JDK 7里新的MethodHandle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。

到本来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：

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 {

}

那个"\_\_JVM\_DefineClass\_\_"的来源是这里：

src/share/vm/prims/jvm.cpp

C++代码

// 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\_\_";

