侯捷观点

Java反射机制

摘要

Reflection 是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。本文借由实例，大面积示范Reflection APIs。

关于本文：
读者基础：具备Java 语言基础。
本文适用工具：JDK1.5

关键词：
Introspection（内省、内观）
Reflection（反射）

有时候我们说某个语言具有很强的动态性，有时候我们会区分动态和静态的不同技术与作法。我们朗朗上口动态绑定（dynamic binding）、动态链接（dynamic linking）、动态加载（dynamic loading）等。然而“动态”一词其实没有绝对而普遍适用的严格定义，有时候甚至像对象导向当初被导入编程领域一样，一人一把号，各吹各的调。

一般而言，开发者社群说到动态语言，大致认同的一个定义是：“程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。

尽管在这样的定义与分类下Java不是动态语言，它却有着一个非常突出的动态相关机制：**Reflection**。这个字的意思是“反射、映象、倒影”，用在Java身上指的是我们可以于运行时加载、探知、使用编译期间完全未知的classes。换句话说，Java程序可以加载一个运行时才得知名称的class，获悉其完整构造（但不包括methods定义），并生成其对象实体、或对其fields设值、或唤起其methods[1]。这种“看透class”的能力（the ability of the program to examine itself）被称为introspection（内省、内观、反省）。Reflection和introspection是常被并提的两个术语。

Java如何能够做出上述的动态特性呢？这是一个深远话题，本文对此只简单介绍一些概念。整个篇幅最主要还是介绍Reflection APIs，也就是让读者知道如何探索class的结构、如何对某个“运行时才获知名称的class”生成一份实体、为其fields设值、调用其methods。本文将谈到

java.lang.Class，以及java.lang.reflect中的Method、Field、Constructor等等classes。

“Class”class

众所周知Java有个Object class，是所有Java classes的继承根源，其内声明了数个应该在所有Java class中被改写的methods：hashCode()、equals()、clone()、toString()、getClass()等。其中getClass()返回一个Class object。

Class class十分特殊。它和一般classes一样继承自Object，其实体用以表达Java程序运行时的classes和interfaces，也用来表达enum、array、primitive Java types（boolean, byte, char, short, int, long, float, double）以及关键词void。当一个class被加载，或当加载器（class loader）的defineClass()被JVM调用，JVM 便自动产生一个Class object。如果您想借由“修改Java标准库源码”来观察Class object的实际生成时机（例如在Class的constructor内添加一个println()），不能够！因为Class并没有public constructor（见图1）。本文最后我会拨一小块篇幅顺带谈谈Java标准库源码的改动办法。

Class是Reflection故事起源。针对任何您想探勘的class，唯有先为它产生一个Class object，接下来才能经由后者唤起为数十多个的Reflection APIs。这些APIs将在稍后的探险活动中一一亮相。

```
#001 public final
#002 class Class<T> implements java.io.Serializable,
#003 java.lang.reflect.GenericDeclaration,
#004 java.lang.reflect.Type,
#005 java.lang.reflect.AnnotatedElement {
#006    private Class() {}
#007    public String toString() {
#008        return ( isInterface() ? "interface " :
#009        (isPrimitive() ? "" : "class "))
#010    + getName();
#011 }
...
```

图1：Class class片段。注意它的private empty ctor，意指不允许任何人经由编程方式产生Class object。是的，其object 只能由JVM 产生。

“Class” object的取得途径

Java允许我们从多种管道为一个class生成对应的Class object。图2是一份整理。

Class object 诞生管道	示例
运用getClass() 注：每个class 都有此函数	String str = "abc"; Class c1 = str.getClass();

运用 Class.getSuperclass()[2]	Button b = new Button(); Class c1 = b.getClass(); Class c2 = c1.getSuperclass();
运用static method Class.forName() （最常被使用）	Class c1 = Class.forName ("java.lang. String"); Class c2 = Class.forName ("java.awt.Button"); Class c3 = Class.forName ("java.util. LinkedList$Entry"); Class c4 = Class.forName ("I"); Class c5 = Class.forName ("[I");
运用 .class 语法	Class c1 = String.class; Class c2 = java.awt.Button.class; Class c3 = Main.InnerClass.class; Class c4 = int.class; Class c5 = int[].class;
运用 primitive wrapper classes 的TYPE 语法	Class c1 = Boolean.TYPE; Class c2 = Byte.TYPE; Class c3 = Character.TYPE; Class c4 = Short.TYPE; Class c5 = Integer.TYPE; Class c6 = Long.TYPE; Class c7 = Float.TYPE; Class c8 = Double.TYPE; Class c9 = Void.TYPE;

图**2**：**Java** 允许多种管道生成**Class object**。

Java classes 组成分析

首先容我以图3的java.util.LinkedList为例，将Java class的定义大卸八块，每一块分别对应图4所示的Reflection API。图5则是“获得class各区块信息”的程序示例及执行结果，它们都取自本文示例程序的对应片段。

```
package java.util;                          //(1)
import java.lang.*;                         //(2)
public class LinkedList<E>                  //(3)(4)(5)
extends AbstractSequentialList<E>           //(6)
implements List<E>, Queue<E>,
Cloneable, java.io.Serializable             //(7)
{
private static class Entry<E> { … }//(8)
public LinkedList() { … }                   //(9)
public LinkedList(Collection<? extends E> c) { … }
public E getFirst() { … }                   //(10)
public E getLast() { … }
private transient Entry<E> header = …;  //(11)
private transient int size = 0;
}
```

图3：将一个**Java class** 大卸八块，每块相应于一个或一组**Reflection APIs**（图4）。

Java classes 各成份所对应的Reflection APIs

图3的各个Java class成份，分别对应于图4的Reflection API，其中出现的Package、Method、Constructor、Field等等classes，都定义于java.lang.reflect。

Java class 内部模块（参见图3）	Java class 内部模块说明	相应之Reflection API，多半为Class methods。	返回值类型 (return type)
(1) package	class隶属哪个package	getPackage()	Package
(2) import	class导入哪些classes	无直接对应之API。 解决办法见图5-2。	
(3) modifier	class（或methods, fields）的属性	int getModifiers() Modifier.toString (int) Modifier. isInterface(int)	int String bool
(4) class name or interface name	class/interface	名称getName()	String
(5) type parameters	参数化类型的名称	getTypeParameters ()	TypeVariable <Class>[]
(6) base class	base class（只可能一个）	getSuperClass()	Class
(7) implemented interfaces	实现有哪些interfaces	getInterfaces()	Class[]
(8) inner classes	内部classes	getDeclaredClasses ()	Class[]
(8') outer class	如果我们观察的class 本身是inner classes，那么相对它就会有个outer class。	getDeclaringClass()	Class
(9) constructors	构造函数 getDeclaredConstructors ()	不论 public 或 private 或其它 access level，皆可获得。另有功能近似之取得函数。	Constructor[]

(10) methods	操作函数 getDeclaredMethods()	不论 public 或 private 或其它 access level，皆可获得。另有功能近似之取得函数。	Method[]
(11) fields	字段（成员变量）	getDeclaredFields() 不论 public 或 private 或其它 access level，皆可获得。另有功能近似之取得函数。	Field[]

图**4**：**Java class**大卸八块后（如图**3**），每一块所对应的**Reflection API**。本表并非 **Reflection APIs** 的全部。

Java Reflection API 运用示例

图5示范图4提过的每一个Reflection API，及其执行结果。程序中出现的tName()是个辅助函数，可将其第一自变量所代表的“Java class完整路径字符串”剥除路径部分，留下class名称，储存到第二自变量所代表的一个hashtable去并返回（如果第二自变量为null，就不储存而只是返回）。

```
#001 Class c = null;
#002 c = Class.forName(args[0]);
#003
#004 Package p;
#005 p = c.getPackage();
#006
#007 if (p != null)
#008    System.out.println("package "+p.getName()+";");

执行结果（例）：
package java.util;
```

图**5-1**：找出**class** 隶属的**package**。其中的**c**将继续沿用于以下各程序片段。

```
#001 ff = c.getDeclaredFields();
#002 for (int i = 0; i < ff.length; i++)
#003    x = tName(ff[i].getType().getName(), classRef);
#004
#005 cn = c.getDeclaredConstructors();
#006 for (int i = 0; i < cn.length; i++) {
#007    Class cx[] = cn[i].getParameterTypes();
#008    for (int j = 0; j < cx.length; j++)
#009        x = tName(cx[j].getName(), classRef);
#010 }
#011
#012 mm = c.getDeclaredMethods();
#013 for (int i = 0; i < mm.length; i++) {
#014    x = tName(mm[i].getReturnType().getName(), classRef);
#015    Class cx[] = mm[i].getParameterTypes();
```

```
#016    for (int j = 0; j < cx.length; j++)
#017        x = tName(cx[j].getName(), classRef);
#018 }
#019 classRef.remove(c.getName()); //不必记录自己（不需import 自己）

执行结果（例）：
import java.util.ListIterator;
import java.lang.Object;
import java.util.LinkedList$Entry;
import java.util.Collection;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
```

图**5-2**：找出导入的**classes**，动作细节详见内文说明。

```
#001 int mod = c.getModifiers();
#002 System.out.print(Modifier.toString(mod)); //整个modifier
#003
#004 if (Modifier.isInterface(mod))
#005    System.out.print(" "); //关键词 "interface" 已含于modifier
#006 else
#007    System.out.print(" class "); //关键词 "class"
#008 System.out.print(tName(c.getName(), null)); //class 名称

执行结果（例）：
public class LinkedList
```

图**5-3**：找出**class**或**interface** 的名称，及其属性（**modifiers**）。

```
#001 TypeVariable<Class>[] tv;
#002 tv = c.getTypeParameters(); //warning: unchecked conversion
#003 for (int i = 0; i < tv.length; i++) {
#004    x = tName(tv[i].getName(), null); //例如 E,K,V...
#005    if (i == 0) //第一个
#006        System.out.print("<" + x);
#007    else //非第一个
#008        System.out.print("," + x);
#009    if (i == tv.length-1) //最后一个
#010        System.out.println(">");
#011 }

执行结果（例）：
public abstract interface Map<K,V>
或 public class LinkedList<E>
```

图**5-4**：找出**parameterized types** 的名称

```
#001 Class supClass;
#002 supClass = c.getSuperclass();
#003 if (supClass != null) //如果有super class
#004    System.out.print(" extends" +
#005 tName(supClass.getName(),classRef));

执行结果（例）：
public class LinkedList<E>
extends AbstractSequentialList,
```

图**5-5**：找出**base class**。执行结果多出一个不该有的逗号于尾端。此非本处重点，为简化计，不多做处理。

```
#001 Class cc[];
#002 Class ctmp;
#003 //找出所有被实现的interfaces
#004 cc = c.getInterfaces();
#005 if (cc.length != 0)
#006    System.out.print(", \r\n" + " implements "); //关键词
#007 for (Class cite : cc) //JDK1.5 新式循环写法
#008    System.out.print(tName(cite.getName(), null)+", ");

执行结果（例）：
public class LinkedList<E>
extends AbstractSequentialList,
implements List, Queue, Cloneable, Serializable,
```

图**5-6**：找出**implemented interfaces**。执行结果多出一个不该有的逗号于尾端。此非本处重点，为简化计，不多做处理。

```
#001 cc = c.getDeclaredClasses(); //找出inner classes
#002 for (Class cite : cc)
#003    System.out.println(tName(cite.getName(), null));
#004
#005 ctmp = c.getDeclaringClass(); //找出outer classes
#006 if (ctmp != null)
#007    System.out.println(ctmp.getName());

执行结果（例）：
LinkedList$Entry
LinkedList$ListItr
```

图**5-7**：找出**inner classes** 和**outer class**

```
#001 Constructor cn[];
#002 cn = c.getDeclaredConstructors();
#003 for (int i = 0; i < cn.length; i++) {
#004    int md = cn[i].getModifiers();
#005    System.out.print(" " + Modifier.toString(md) + " " +
#006    cn[i].getName());
#007    Class cx[] = cn[i].getParameterTypes();
#008    System.out.print("(");
#009    for (int j = 0; j < cx.length; j++) {
#010        System.out.print(tName(cx[j].getName(), null));
#011        if (j < (cx.length - 1)) System.out.print(", ");
#012    }
#013    System.out.print(")");
#014 }

执行结果（例）：
public java.util.LinkedList(Collection)
public java.util.LinkedList()
```

图**5-8a**：找出所有**constructors**

```
#004 System.out.println(cn[i].toGenericString());

执行结果（例）：
```

```
public java.util.LinkedList(java.util.Collection<? extends E>)
public java.util.LinkedList()
```

图**5-8b**：找出所有**constructors**。本例在**for** 循环内使用**toGenericString()**，省事。

```
#001 Method mm[];
#002 mm = c.getDeclaredMethods();
#003 for (int i = 0; i < mm.length; i++) {
#004    int md = mm[i].getModifiers();
#005    System.out.print(" "+Modifier.toString(md)+" "+
#006    tName(mm[i].getReturnType().getName(), null)+" "+
#007    mm[i].getName());
#008    Class cx[] = mm[i].getParameterTypes();
#009    System.out.print("(");
#010    for (int j = 0; j < cx.length; j++) {
#011        System.out.print(tName(cx[j].getName(), null));
#012    if (j < (cx.length - 1)) System.out.print(", ");
#013    }
#014    System.out.print(")");
#015 }

执行结果（例）：
public Object get(int)
public int size()
```

图**5-9a**：找出所有**methods**

```
#004 System.out.println(mm[i].toGenericString());

public E java.util.LinkedList.get(int)
public int java.util.LinkedList.size()
```

图**5-9b**：找出所有**methods**。本例在**for** 循环内使用**toGenericString()**，省事。

```
#001 Field ff[];
#002 ff = c.getDeclaredFields();
#003 for (int i = 0; i < ff.length; i++) {
#004    int md = ff[i].getModifiers();
#005    System.out.println(" "+Modifier.toString(md)+" "+
#006    tName(ff[i].getType().getName(), null) +" "+
#007    ff[i].getName()+";");
#008 }

执行结果（例）：
private transient LinkedList$Entry header;
private transient int size;
```

图**5-10a**：找出所有**fields**

```
#004 System.out.println("G: " + ff[i].toGenericString());

private transient java.util.LinkedList.java.util.LinkedList$Entry<E> ??
java.util.LinkedList.header
private transient int java.util.LinkedList.size
```

图**5-10b**：找出所有**fields**。本例在**for** 循环内使用**toGenericString()**，省事。

找出**class**参用（导入）的所有**classes**

没有直接可用的Reflection API可以为我们找出某个class参用的所有其它classes。要获得这项信息，必须做苦工，一步一脚印逐一记录。我们必须观察所有fields的类型、所有methods（包括constructors）的参数类型和回返类型，剔除重复，留下唯一。这正是为什么图5-2程序代码要为tName()指定一个hashtable（而非一个null）做为第二自变量的缘故：hashtable可为我们储存元素（本例为字符串），又保证不重复。

本文讨论至此，几乎可以还原一个class的原貌（唯有methods 和ctors的定义无法取得）。接下来讨论Reflection 的另三个动态性质：(1) 运行时生成instances，(2) 执行期唤起methods，(3) 运行时改动fields。

运行时生成**instances**

欲生成对象实体，在Reflection 动态机制中有两种作法，一个针对“无自变量ctor”，一个针对“带参数ctor”。图6是面对“无自变量ctor”的例子。如果欲调用的是“带参数ctor“就比较麻烦些，图7是个例子，其中不再调用Class的newInstance()，而是调用Constructor 的newInstance()。图7首先准备一个Class[]做为ctor的参数类型（本例指定为一个double和一个int），然后以此为自变量调用getConstructor()，获得一个专属ctor。接下来再准备一个Object[] 做为ctor实参值（本例指定3.14159和125），调用上述专属ctor的newInstance()。

```
#001 Class c = Class.forName("DynTest");
#002 Object obj = null;
#003 obj = c.newInstance(); //不带自变量
#004 System.out.println(obj);
```

图**6**：动态生成“**Class object** 所对应之**class**”的对象实体；无自变量。

```
#001 Class c = Class.forName("DynTest");
#002 Class[] pTypes = new Class[] { double.class, int.class };
#003 Constructor ctor = c.getConstructor(pTypes);
#004 //指定parameter list，便可获得特定之ctor
#005
#006 Object obj = null;
#007 Object[] arg = new Object[] {3.14159, 125}; //自变量
#008 obj = ctor.newInstance(arg);
#009 System.out.println(obj);
```

图**7**：动态生成“**Class object** 对应之**class**”的对象实体；自变量以**Object[]**表示。

运行时调用**methods**

这个动作和上述调用“带参数之ctor”相当类似。首先准备一个Class[]做为ctor的参数类型（本例指定其中一个是String，另一个是Hashtable），然后以此为自变量调用getMethod()，获得特定的Method object。接下来准备一个Object[]放置自变量，然后调用上述所得之特定Method object的invoke()，如图8。知道为什么索取Method object时不需指定回返类型吗？因为method overloading机制要求signature（署名式）必须唯一，而回返类型并非signature的一个成份。换句话说，只要指定了method名称和参数列，就一定指出了一个独一无二的method。

```
#001 public String func(String s, Hashtable ht)
#002 {
#003 …System.out.println("func invoked"); return s;
#004 }
#005 public static void main(String args[])
#006 {
#007 Class c = Class.forName("Test");
#008 Class ptypes[] = new Class[2];
#009 ptypes[0] = Class.forName("java.lang.String");
#010 ptypes[1] = Class.forName("java.util.Hashtable");
#011 Method m = c.getMethod("func",ptypes);
#012 Test obj = new Test();
#013 Object args[] = new Object[2];
#014 arg[0] = new String("Hello,world");
#015 arg[1] = null;
#016 Object r = m.invoke(obj, arg);
#017 Integer rval = (String)r;
#018 System.out.println(rval);
#019 }
```

图**8**：动态唤起**method**

运行时变更fields内容

与先前两个动作相比，“变更field内容”轻松多了，因为它不需要参数和自变量。首先调用Class的getField()并指定field名称。获得特定的Field object之后便可直接调用Field的get()和set()，如图9。

```
#001 public class Test {
#002 public double d;
#003
#004 public static void main(String args[])
#005 {
#006 Class c = Class.forName("Test");
#007 Field f = c.getField("d"); //指定field 名称
#008 Test obj = new Test();
#009 System.out.println("d= " + (Double)f.get(obj));
#010 f.set(obj, 12.34);
#011 System.out.println("d= " + obj.d);
#012 }
#013 }
```

图**9**：动态变更**field** 内容

Java 源码改动办法

先前我曾提到，原本想借由“改动Java标准库源码”来测知Class object的生成，但由于其ctor原始设计为private，也就是说不可能透过这个管道生成Class object（而是由class loader负责生成），因此“在ctor中打印出某种信息”的企图也就失去了意义。

这里我要谈点题外话：如何修改Java标准库源码并让它反应到我们的应用程序来。假设我想修改java.lang.Class，让它在某些情况下打印某种信息。首先必须找出标准源码！当你下载JDK 套

件并安装妥当，你会发现jdk150\src\java\lang 目录（见图10）之中有Class.java，这就是我们此次行动的标准源码。备份后加以修改，编译获得Class.class。接下来准备将.class 搬移到jdk150\jre\lib\endorsed（见图10）。

这是一个十分特别的目录，class loader将优先从该处读取内含classes的.jar文件——成功的条件是.jar内的classes压缩路径必须和Java标准库的路径完全相同。为此，我们可以将刚才做出的Class.class先搬到一个为此目的而刻意做出来的\java\lang目录中，压缩为foo.zip（任意命名，唯需夹带路径java\lang），再将这个foo.zip搬到jdk150\jre\lib\endorsed并改名为foo.jar。此后你的应用程序便会优先用上这里的java.lang.Class。整个过程可写成一个批处理文件（batch file），如图11，在DOS Box中使用。

图10

图**10**：**JDK1.5** 安装后的目录组织。其中的**endorsed** 是我新建。

```
del e:\java\lang\*.class //清理干净
del c:\jdk150\jre\lib\endorsed\foo.jar //清理干净
c:
cd c:\jdk150\src\java\lang
javac -Xlint:unchecked Class.java //编译源码
javac -Xlint:unchecked ClassLoader.java //编译另一个源码（如有必要）
move *.class e:\java\lang //搬移至刻意制造的目录中
e:
cd e:\java\lang //以下压缩至适当目录
pkzipc -add -path=root c:\jdk150\jre\lib\endorsed\foo.jar *.class
cd e:\test //进入测试目录
javac -Xlint:unchecked Test.java //编译测试程序
java Test //执行测试程序
```

图**11**：一个可在**DOS Box**中使用的批处理文件（**batch file**），用以自动化**java.lang.Class**的修改动作。**Pkzipc(.exe)**是个命令列压缩工具，**add**和**path**都是其命令。

更多信息

以下是视野所及与本文主题相关的更多讨论。这些信息可以弥补因文章篇幅限制而带来的不足，或带给您更多视野。

- "*Take an in-depth look at the Java Reflection API -- Learn about the new Java 1.1 tools forfinding out information about classes*", by Chuck McManis。此篇文章所附程序代码是本文示例程序的主要依据（本文示例程序示范了更多Reflection APIs，并采用JDK1.5 新式的for-loop 写法）。
- "*Take a look inside Java classes -- Learn to deduce properties of a Java class from inside aJava program*", by Chuck McManis。
- "*The basics of Java class loaders -- The fundamentals of this key component of the Javaarchitecture*", by Chuck McManis。
- 《The Java Tutorial Continued》, Sun microsystems. Lesson58-61, "Reflection".

注1用过诸如MFC这类所谓 Application Framework的程序员也许知道，MFC有所谓的dynamic creation。但它并不等同于Java的动态加载或动态辨识；所有能够在MFC程序中起作用的classes，都必须先在编译期被编译器“看见”。

注2如果操作对象是Object，Class.getSuperClass()会返回null。

本文程序源码可至侯捷网站下载：
http://www.jjhou.com/javatwo-2004-reflection-and-generics-in-jdk15-sample.ZIP

发表于 2004年10月27日 11:30 AM