侯捷觀點

Java 反射機制

— Java Reflection —

北京《程序員》2004/09
台北《Run!PC》2004/09

作者簡介：侯捷，資訊工作者、專欄執筆、大學教師。常著文章自娛，頗示己志。
侯捷網站：http://www.jjhou.com（繁體）
北京鏡站：http://jjhou.csdn.net（簡體）
永久郵箱：jjhou@jjhou.com

- 讀者基礎：具備 Java 語言基礎。
- 本文適用工具：JDK1.5
- 本文程式源碼可至侯捷網站下載
  http://www.jjhou.com/javatwo-2004-reflection-and-generics-in-jdk15-sample.ZIP
- 本文同時也是 JavaTwo-2004 技術研討會同名講題之部分內容書面整理。
- 關鍵術語：
  Introspection（內省、內觀）
  Reflection（反射）

引言

Reflection 是 Java 被視爲動態（或準動態）語言的一個關鍵性質。這個機制允許程式在執行期透過 Reflection APIs 取得任何一個已知名稱的 class 的內部資訊，包括其 modifiers（諸如 public, static 等等）、superclass（例如 `Object`）、實作之 interfaces（例如 `Cloneable`），也包括 fields 和 methods 的所有資訊，並可於執行期改變 fields 內容或喚起 methods。本文藉由實例，大面積示範 Reflection APIs。

有時候我們說某個語言具有很強的動態性，有時候我們會區分動態和靜態的不同技術與作法。我們朗朗上口**動態繫結**（dynamic binding）、**動態聯結**（dynamic linking）、**動態載入**（dynamic loading）…。然而「動態」一詞其實沒有絕對而普遍適用的嚴格定義，有時候甚至像**物件導向**當初被導入編程領域一樣，一人一把號，各吹各的調。

一般而言，開發者社群說到**動態語言**，大致認同的一個定義是：「程式執行期間，允許改變程式結構或變數型別，這種語言稱爲動態語言」。從這個觀點看，Perl, Python, Ruby 是動態語言，C++, Java, C# 不是動態語言。

儘管在這樣的定義與分類下 Java 不是動態語言，它卻有著一個非常突出的動態相關機制：**Reflection**。這個字的意思是「反射、映象、倒影」，用在 Java 身上指的是我們可以於執行期載入、探知、使用編譯期間完全未知的 classes。換句話說，Java 程式可以載入一個執行期才得知名稱的 class，獲悉其完整構造（但不包括 methods 定義），並生成其物件實體、或對其 fields 設值、或喚起其 methods[1]。這種「看透 class」的能力（the ability of the program to examine itself）被稱爲 **introspection**（內省、內觀、反省）。Reflection 和 introspection 是常被並提的兩個術語。

Java 如何能夠做出上述的動態特性呢？這是一個深遠話題，本文對此只簡單介紹一些概念。整個篇幅最主要還是介紹 Reflection APIs，也就是讓讀者知道如何探索 class 的結構、如何對某個「執行期才獲知名稱的 class」生成一份實體、爲其 fields 設值、呼叫其 methods。本文將談到 `java.lang.Class`，以及 java.lang.reflect 中的 `Method`, `Field`, `Constructor` 等等 classes。

"Class" class

眾所周知 Java 有個 `Object` class，是所有 Java classes 的繼承根源，其內宣告了數個應該在所有 Java class 中被改寫的 methods：`hashCode()`、`equals()`、`clone()`、

[1]用過諸如 MFC 這類所謂 Application Framework 的程式員也許知道，MFC 有所謂的 dynamic creation。但它並不等同於 Java 的動態載入或動態辨識；所有能夠在 MFC 程式中起作用的 classes，都必須先在編譯期被編譯器「看見」。

`toString()`、`getClass()`...。其中 `getClass()`傳回一個 `Class` object。

`Class` class 十分特殊。它和一般 classes 一樣繼承自 `Object`，其實體用以表達 Java 程式執行期間的 classes 和 interfaces，也用來表達 enum、array、primitive Java types（boolean, byte, char, short, int, long, float, double）以及關鍵字 void。當一個 class 被載入，或當載入器（class loader）的 `defineClass()`被 JVM 呼叫，JVM 便自動產生一個 `Class` object。如果您想藉由「修改 Java 程式庫源碼」來觀察 `Class` object 的實際生成時機（例如在 `Class` 的 constructor 內添加一個 `println()`），不能夠！因爲 `Class` 並沒有 public constructor（見圖 1）。本文最後我會撥一小塊篇幅順帶談談 Java 程式庫源碼的改動辦法。

`Class` 是 Reflection 故事起源。針對任何您想探勘的 class，唯有先爲它產生一個 `Class` object，接下來才能經由後者喚起爲數十多個的 Reflection APIs。這些 APIs 將在稍後的探險活動中一一亮相。

```
#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     }
...
```

圖 1 / `Class` class 片段。注意它的 private empty ctor，意指不允許任何人經由編程方式產生 `Class` object。是的，其 object 只能由 JVM 產生。

"Class" object 的取得途徑

Java 允許我們從多種管道爲一個 class 生成對應的 `Class` object。圖 2 是一份整理。

Class object 誕生管道	示例
運用 getClass() 註：每個 class 都有此函式	String str = "abc"; Class c1 = str.getClass();
運用 Class.getSuperclass()[2]	Button b = new Button(); Class c1 = b.getClass(); Class c2 = c1.getSuperclass();
運用 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");
運用 .class 語法	Class c1 = String.class; Class c2 = java.awt.Button.class; Class c3 = Main.InnerClass.class; Class c4 = int.class; Class c5 = int[].class;
運用 primitive wrapper classes 的 TYPE 語法	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;

圖 2 / Java 允許多種管道生成 Class object。

Java classes 組成分析

首先容我以圖 3 的 java.util.LinkedList 爲例，將 Java class 的定義大卸八塊，每一塊分別對應圖 4 所示的 Reflection API。圖 5 則是「獲得 class 各區塊資訊」的程式示例及執行結果，它們都取自本文示例程式的對應片段。

```
package java.util;     //(1)
import java.lang.*;    //(2)

public class LinkedList<E>                          //(3)(4)(5)
    extends AbstractSequentialList<E>               //(6)
```

[2] 如果操作對象是 Object，Class.getSuperClass()會傳回 null。

```
    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;
}
```

圖 3 / 將一個 Java class 大卸八塊，每塊相應於一個或一組 Reflection APIs（圖 4）。

Java classes 各 ㄓㄨˋ份所對應的 Reflection APIs

圖 3 的各個 Java class 成份，分別對應於圖 4 的 Reflection API，其中出現的 `Package`, `Method`, `Constructor`, `Field` 等等 classes，都定義於 `java.lang.reflect`。

Java class 內部模塊（參見圖 3）	Java class 內部模塊說明	相應之 Reflection API. 多半爲 `Class` methods.	傳回值型別 (return type)
(1) package	class 隸屬哪個 package	getPackage()	`Package`
(2) import	class 匯入哪些 classes	無直接對應之 API。 解決辦法見圖 5-2。	

(3) modifier	class（或 methods, fields）的屬性	int getModifiers()
		Modifier.toString(int)
		Modifier.isInterface(int)
			`int`
			`String`
			`bool`

(4) class name or interface name	class/interface 名稱	getName()	`String`
(5) type parameters	參數化型別的名稱	getTypeParameters()	`TypeVariable <Class>[]`
(6) base class	base class（只可能一個）	getSuperClass()	`Class`
(7) implemented interfaces	實作有哪些 interfaces	getInterfaces()	`Class[]`
(8) inner classes	內隱式 classes	getDeclaredClasses()	`Class[]`

(8') outer class	如果我們觀察的 class 本身是 inner classes，那麼相對它就會有個 outer class。	getDeclaringClass()	`Class`
(9) constructors	建構式	getDeclaredConstructors() 不論 public 或 private 或其他 access level，皆可獲得。 另有功能近似之取得函式。	`Constructor[]`
(10) methods	操作函式	getDeclaredMethods() 不論 public 或 private 或其他 access level，皆可獲得。 另有功能近似之取得函式。	`Method[]`
(11) fields	欄位（成員變數）	getDeclaredFields() 不論 public 或 private 或其他 access level，皆可獲得。 另有功能近似之取得函式。	`Field[]`

圖 4 / Java class 大卸八塊後（如圖 3），每一塊所對應的 Reflection API。本表並非 Reflection APIs 的全部。

Java Reflection API 運用示例

圖 5 示範圖 4 提過的每一個 Reflection API，及其執行結果。程式中出現的 `tName()` 是個輔助函式，可將其第一引數所代表的「Java class 完整路徑字串」剝除路徑部分，留下 class 名稱，儲存到第二引數所代表的一個 hashtable 去並傳回（如果第二引數爲 null，就不儲存而只是傳回）。

```
#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()+";");

執行結果（例）：
package java.util;
```

圖 5-1 / 找出 class 隸屬的 package。其中的 c 將繼續沿用於以下各程式片段。

```
#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()); //不必記錄自己（不需 import 自己）

執行結果（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;
```

圖 5-2 / 找出匯入的 classes，動作細節詳見內文說明。

```
#001 int mod = c.getModifiers();
#002 System.out.print(Modifier.toString(mod));   //整個 modifier
#003
#004 if (Modifier.isInterface(mod))
#005     System.out.print(" ");       //關鍵字 "interface" 已含於 modifier
#006 else
#007     System.out.print(" class "); //關鍵字 "class"
#008 System.out.print(tName(c.getName(), null)); //class 名稱

執行結果（例）：
public class LinkedList
```

圖 5-3 / 找出 class 或 interface 的名稱，及其屬性（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)   //第一個
#006        System.out.print("<" + x);
#007     else          //非第一個
#008        System.out.print("," + x);
#009     if (i == tv.length-1) //最後一個
#010        System.out.println(">");
#011  }

執行結果（例）：
   public abstract interface Map<K,V>
或 public class LinkedList<E>
```

圖 5-4 / 找出 parameterized types 的名稱

```
#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));

執行結果（例）：
public class LinkedList<E>
  extends AbstractSequentialList,
```

圖 5-5 / 找出 base class。執行結果多出一個不該有的逗號於尾端。此非本處重點，爲簡化計，不多做處理。

```
#001 Class cc[];
#002 Class ctmp;
#003 //找出所有被實作的 interfaces
#004 cc = c.getInterfaces();
#005 if (cc.length != 0)
#006    System.out.print(", \r\n" + "  implements ");   //關鍵字
#007 for (Class cite : cc)   //JDK1.5 新式迴圈寫法
#008    System.out.print(tName(cite.getName(), null)+", ");

執行結果（例）：
public class LinkedList<E>
  extends AbstractSequentialList,
    implements List, Queue, Cloneable, Serializable,
```

圖 5-6 / 找出 implemented interfaces。執行結果多出一個不該有的逗號於尾端。此非本處重點，爲簡化計，不多做處理。

```
#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());

執行結果（例）：
LinkedList$Entry
LinkedList$ListItr
```

圖 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 }

執行結果（例）：
    public java.util.LinkedList(Collection)
    public java.util.LinkedList()
```

圖 5-8a / 找出所有 constructors

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

執行結果（例）：
    public java.util.LinkedList(java.util.Collection<? extends E>)
    public java.util.LinkedList()
```

圖 5-8b / 找出所有 constructors。本例在 for 迴圈內使用 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 }

執行結果（例）：
   public Object get(int)
   public int size()
```

圖 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()
```

圖 5-9b / 找出所有 methods。本例在 for 迴圈內使用 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 }

執行結果（例）：
   private transient LinkedList$Entry header;
   private transient int size;
```

圖 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
```

圖 5-10b / 找出所有 fields。本例在 for 迴圈內使用 toGenericString()，省事。

找出 class 參用（用）的所有 classes

沒有直接可用的 Reflection API 可以爲我們找出某個 class 參用的所有其他 classes。要獲得這項資訊，必須做苦工，一步一腳印逐一記錄。我們必須觀察所有 fields 的型別、所有 methods（包括 constructors）的參數型別和回返型別，剔除重複，留下唯一。這正是爲什麼圖 5-2 程式碼要爲 `tName()` 指定一個 hashtable（而非一個 null）做爲第二引數的緣故：hashtable 可爲我們儲存元素（本例爲字串），又保證不重複。

本文討論至此，幾乎可以還原一個 class 的原貌（唯有 methods 和 ctors 的定義無法取得）。接下來討論 Reflection 的另三個動態性質：(1) 執行期生成 instances，(2) 執行期喚起 methods，(3) 執行期改動 fields。

執行期生成 instances

欲生成物件實體，在 Reflection 動態機制中有兩種作法，一個針對「無引數 ctor」，一個針對「帶參數 ctor」。圖 6 是面對「無引數 ctor」的例子。如果欲叫起的是「帶參數 ctor」就比較麻煩些，圖 7 是個例子，其中不再呼叫 `Class` 的 `newInstance()`，而是呼叫 `Constructor` 的 `newInstance()`。圖 7 首先準備一個 `Class[]`做爲 ctor 的參數型別（本例指定爲一個 `double` 和一個 `int`），然後以此爲引數呼叫 `getConstructor()`，獲得一個專屬 ctor。接下來再準備一個 `Object[]` 做爲 ctor 實際引數值（本例指定 3.14159 和 125），呼叫上述專屬 ctor 的 `newInstance()`。

```
#001 Class c = Class.forName("DynTest");
#002 Object obj = null;
#003 obj = c.newInstance();  //不帶引數
#004 System.out.println(obj);
```

圖 6 / 動態生成「`Class` object 所對應之 class」的物件實體；無引數。

```
#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，便可獲得特定之 ctor
#005
#006 Object obj = null;
```

```
#007 Object[] arg = new Object[] {3.14159, 125}; //引數
#008 obj = ctor.newInstance(arg);
#009 System.out.println(obj);
```

圖 7 / 動態生成「Class object 對應之 class」的物件實體；引數以 Object[]表示。

執行期喚起 methods

這個動作和上述呼叫「帶參數之 ctor」相當類似。首先準備一個 Class[]做爲 ctor 的參數型別（本例指定其中一個是 String，另一個是 Hashtable），然後以此爲引數呼叫 getMethod()，獲得特定的 Method object。接下來準備一個 Object[] 放置引數，然後呼叫上述所得之特定 Method object 的 invoke()，如圖 8。知道爲什麼索取 Method object 時不需指定回返型別嗎？因爲 method overloading 機制要求 signature（署名式）必須唯一，而回返型別並非 signature 的一個成份。換句話說，只要指定了 method 名稱和參數列，就一定指出了一個獨一無二的 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 }
```

圖 8 / 動態喚起 method

執行期變更 fields 內容

與先前兩個動作相比，「變更 field 內容」輕鬆多了，因爲它不需要參數和引數。首先呼叫 Class 的 getField()並指定 field 名稱。獲得特定的 Field object 之後

便可直接呼叫 Field 的 get()和 set()，如圖 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 名稱
#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 }
```

圖 9 / 動態變更 field 內容

Java 原碼改動辦法

先前我曾提到，原本想藉由「改動 Java 程式庫源碼」來測知 Class object 的生成，但由於其 ctor 原始設計爲 private，也就是說不可能透過這個管道生成 Class object（而是由 class loader 負責生成），因此「在 ctor 中秀出某種訊息」的企圖也就失去了意義。

這裡我要談點題外話：如何修改 Java 程式庫源碼並讓它反應到我們的應用程式來。假設我想修改 java.lang.Class，讓它在某些情況下列印某種訊息。首先必須找出標的源碼！當你下載 JDK 套件並安裝妥當，你會發現 jdk150\src\java\lang 目錄（見圖 10）之中有 Class.java，這就是我們此次行動的標的源碼。備份後加以修改，編譯獲得 Class.class。接下來準備將.class 搬移到 jdk150\jre\lib\endorsed（見圖 10）。這是一個十分特別的目錄，class loader 將優先從該處讀取內含 classes 的.jar 檔 ——成功的條件是.jar 內的 classes 壓縮路徑必須和 Java 程式庫的路徑完全相同。爲此，我們可以將剛才做出的 Class.class 先搬到一個爲此目的而刻意做出來的 \java\lang 目錄中，壓縮爲 foo.zip（任意命名，唯需夾帶路徑 java\lang），再將這個 foo.zip 搬到 jdk150\jre\lib\endorsed 並改名爲 foo.jar。此後你的應用程式便會優先用上這裡的 java.lang.Class。整個過程可寫成一個批次檔（batch file），如圖 11，在 DOS Box 中使用。

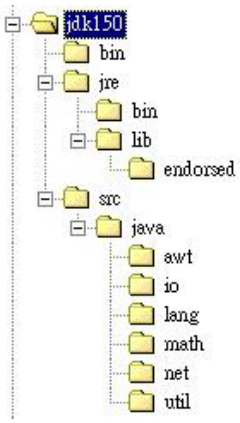

圖 10 / JDK1.5 安裝後的目錄組織。其中的 endorsed 是我新建。

```
del e:\java\lang\*.class                          //清理乾淨
del c:\jdk150\jre\lib\endorsed\foo.jar            //清理乾淨
c:
cd c:\jdk150\src\java\lang
javac -Xlint:unchecked Class.java                 //編譯源碼
javac -Xlint:unchecked ClassLoader.java           //編譯另一個源碼（如有必要）
move *.class e:\java\lang                         //搬移至刻意製造的目錄中
e:
cd e:\java\lang                                   //以下壓縮至適當目錄
pkzipc -add -path=root c:\jdk150\jre\lib\endorsed\foo.jar *.class
cd e:\test                                        //進入測試目錄
javac -Xlint:unchecked Test.java                  //編譯測試程式
java Test                                         //執行測試程式
```

圖 11 / 一個可在 DOS Box 中使用的批次檔（batch file），用以自動化 java.lang.Class 的修改動作。Pkzipc(.exe)是個命令列壓縮工具，add 和 path 都是其命令。

更多資訊

以下是視野所及與本文主題相關的更多討論。這些資訊可以彌補因文章篇幅限制而帶來的不足，或帶給您更多視野。

- "*Take an in-depth look at the Java Reflection API – Learn about the new Java 1.1 tools for finding out information about classes*", by Chuck McManis。此篇文章所附程式碼是

本文示例程式的主要依據（本文示例程式示範了更多 Reflection APIs，並採用 JDK1.5 新式的 for-loop 寫法）。

- "*Take a look inside Java classes -- Learn to deduce properties of a Java class from inside a Java program*", by Chuck McManis。
- "*The basics of Java class loaders -- The fundamentals of this key component of the Java architecture*", by Chuck McManis。
- 《The Java Tutorial Continued》, Sun microsystems. Lesson58-61, "Reflection".