北京圣思园科技有限公司
http://www.shengsiyuan.com

主讲人：张龙

Java 语言的反射机制

- 在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法？答案是肯定的。这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。
- Java 反射机制主要提供了以下功能

Java 语言的反射机制

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法

Java 语言的反射机制

- 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Serializable），也包括fields和methods的所有信息，并可于运行时改变fields内容或调用methods

Java 语言的反射机制

- 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

Java 语言的反射机制

- 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。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 Reflection API 简介

- 在JDK中，主要由以下类来实现Java反射机制，这些类都位于java.lang.reflect包中
  - Class类：代表一个类。
  - Field 类：代表类的成员变量（成员变量也称为类的属性）。
  - Method类：代表类的方法。
  - Constructor 类：代表类的构造方法。
  - Array类：提供了动态创建数组，以及访问数组的元素的静态方法

Java Reflection API 简介

- 例程DumpMethods类演示了Reflection API的基本作用，它读取命令行参数指定的类名，然后打印这个类所具有的方法信息

Java Reflection API 简介

- 例程ReflectTester 类进一步演示了Reflection API的基本使用方法。ReflectTester类有一个copy(Object object)方法，这个方法能够创建一个和参数object 同样类型的对象，然后把object对象中的所有属性拷贝到新建的对象中，并将它返回
- 这个例子只能复制简单的JavaBean，假定JavaBean 的每个属性都有public 类型的getXXX()和setXXX()方法。

Java Reflection API 简介

- ReflectTester 类的copy(Object object)方法依次执行以下步骤
- （1）获得对象的类型：
  - Class classType=object.getClass();
  - System.out.println("Class:"+classType.getNam e());

Java Reflection API 简介

- 在java.lang.Object 类中定义了getClass()方法，因此对于任意一个Java对象，都可以通过此方法获得对象的类型。Class类是Reflection API中的核心类，它有以下方法
  - getName()：获得类的完整名字。
  - getFields()：获得类的public类型的属性。
  - getDeclaredFields()：获得类的所有属性。
  - getMethods()：获得类的public类型的方法。
  - getDeclaredMethods()：获得类的所有方法。

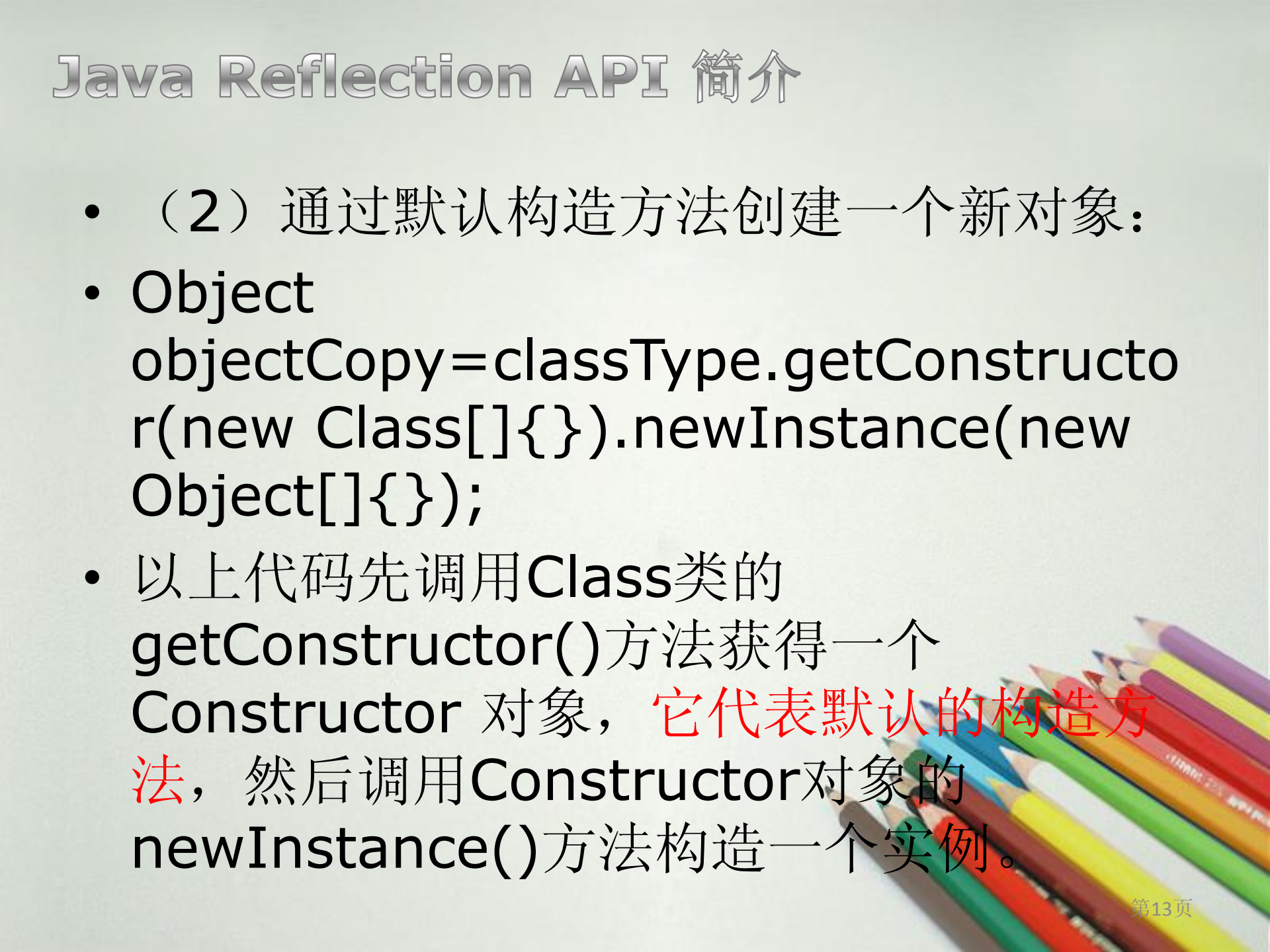
Java Reflection API 简介

- getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name 参数指定方法的名字，parameterTypes 参数指定方法的参数类型。

- getConstructors()：获得类的public类型的构造方法。
- getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes 参数指定构造方法的参数类型。
- newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。

Java Reflection API 简介

- （2）通过默认构造方法创建一个新对象：
- Object objectCopy=classType.getConstructor(new Class[]{}).newInstance(new Object[]{});
- 以上代码先调用Class类的getConstructor()方法获得一个Constructor 对象，它代表默认的构造方法，然后调用Constructor对象的newInstance()方法构造一个实例。



Java Reflection API 简介

- （3）获得对象的所有属性：
- Field fields[]=classType.getDeclaredFields();
- Class 类的getDeclaredFields()方法返回类的所有属性，包括public、protected、默认和private访问级别的属性

Java Reflection API 简介

- （4）获得每个属性相应的getXXX()和setXXX()方法，然后执行这些方法，把原来对象的属性拷贝到新的对象中

Java Reflection API 简介

- 在例程InvokeTester类的main()方法中，运用反射机制调用一个InvokeTester对象的add()和echo()方法

Java Reflection API 简介

- add()方法的两个参数为int 类型，获得表示add()方法的Method对象的代码如下：
- Method addMethod=classType.getMethod("add",new Class[]{int.class,int.class});
- Method类的invoke(Object obj,Object args[])方法接收的参数必须为对象，如果参数为基本类型数据，必须转换为相应的包装类型的对象。invoke()方法的返回值总是对象，如果实际被调用的方法的返回类型是基本类型数据，那么invoke()方法会把它转换为相应的包装类型的对象，再将其返回

Java Reflection API 简介

- 在本例中，尽管InvokeTester 类的add()方法的两个参数以及返回值都是int类型，调用add Method 对象的invoke()方法时，只能传递Integer 类型的参数，并且invoke()方法的返回类型也是Integer 类型，Integer类是int 基本类型的包装类：

- Object result=addMethod.invoke(invokeTester,
- new Object[]{new Integer(100),new Integer(200)});
- System.out.println((Integer)result); //result 为Integer类型

Java Reflection API 简介

- java.lang.Array 类提供了动态创建和访问数组元素的各种静态方法。例程
- ArrayTester1 类的main()方法创建了一个长度为10 的字符串数组，接着把索引位置为5 的元素设为 “hello”，然后再读取索引位置为5 的元素的值

Java Reflection API 简介

- 例程ArrayTester2 类的main()方法创建了一个 5 x 10 x 15 的整型数组，并把索引位置为[3][5][10] 的元素的值为设37

“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

- 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

“Class” class

- Class是Reflection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

“Class” object的取得途径

- Java允许我们从多种途径为一个class生成对应的Class object

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");

“Class” object的取得途径

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;

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”。如果欲调用的是“带参数ctor“就比较麻烦些，不再调用Class的newInstance()，而是调用Constructor 的newInstance()。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()。

运行时生成instances

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

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

运行时生成instances

```
#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()。
- 为什么获得Method object时不需指定回返类型？

运行时调用methods

- 因为method overloading机制要求signature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。

运行时调用methods

```
#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 }
```

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()，

运行时变更fields内容

```
#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