## 深入理解Java反射:

　　要想理解反射的原理，首先要了解什么是类型信息。Java让我们在运行时识别对象和类的信息，主要有2种方式：一种是传统的RTTI，它假定我们在编译时已经知道了所有的类型信息；另一种是反射机制，它允许我们在运行时发现和使用类的信息。

（使用的前提条件：必须先得到代表的字节码的Class，Class类用于表示.class文件（字节码））

### 何为Reflect？

Java反射是指在程序运行时获取已知名臣的类或已有对象的相关信息的一种机制，包括累的方法、属性、父类等信息；

Java反射机制是在运行状态中，对于任意一个类，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一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制；

在JDK中主要通过以下类来实现Java反射机制，这些类都位于Java.lang.reflect包中：

one：Class类：代表一个类；

two：Constructor：代表类的构造方法；

thre：Filed类：代表类的成员变量；

four: Method类：代表类的方法；

1、Class对象

　　理解RTTI在Java中的工作原理，首先需要知道类型信息在运行时是如何表示的，这是由Class对象来完成的，它包含了与类有关的信息。Class对象就是用来创建所有“常规”对象的，Java使用Class对象来执行RTTI，即使你正在执行的是类似类型转换这样的操作。

　　每个类都会产生一个对应的Class对象，也就是保存在.class文件。所有类都是在对其第一次使用时，动态加载到JVM的，当程序创建一个对类的静态成员的引用时，就会加载这个类。Class对象仅在需要的时候才会加载，static初始化是在类加载时进行的。

复制代码

public class TestMain {

public static void main(String[] args) {

System.out.println(XYZ.name);

}

}

class XYZ {

public static String name = "luoxn28";

static {

System.out.println("xyz静态块");

}

public XYZ() {

System.out.println("xyz构造了");

}

}

类加载器首先会检查这个类的Class对象是否已被加载过，如果尚未加载，默认的类加载器就会根据类名查找对应的.class文件。

### 关于Class类：

**Class 类的实例表示正在运行的 Java 应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。（包括基本数据类型）**

**Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的defineClass 方法自动构造的。也就是这不需要我们自己去处理创建，JVM已经帮我们创建好了**

### 获取Class对象的方法：

1、获取Class对象的三种方式

1.1 Object ——> getClass();

1.2 任何数据类型（包括基本数据类型）都有一个“静态”的class属性

1.3 通过Class类的静态方法：forName（String className）(常用)

三种方式常用第三种，第一种对象都有了还要反射干什么。第二种需要导入类的包，依赖太强，不导包就抛编译错误。一般都第三种，一个字符串可以传入也可写在配置文件中等多种方法。

　　想在运行时使用类型信息，必须获取对象(比如类Base对象)的Class对象的引用，使用功能Class.forName(“Base”)可以实现该目的，或者使用base.class。注意，有一点很有趣，使用功能”.class”来创建Class对象的引用时，不会自动初始化该Class对象，使用forName()会自动初始化该Class对象。为了使用类而做的准备工作一般有以下3个步骤：

加载：由类加载器完成，找到对应的字节码，创建一个Class对象

链接：验证类中的字节码，为静态域分配空间

初始化：如果该类有超类，则对其初始化，执行静态初始化器和静态初始化块

public class Base {

static int num = 1;

static {

System.out.println("Base " + num);

}

}

public class Main {

public static void main(String[] args) {

// 不会初始化静态块

Class clazz1 = Base.class;

System.out.println("------");

// 会初始化

Class clazz2 = Class.forName("zzz.Base");

}

}

输出结果：

**------**

**Base 1**

2、类型转换前先做检查

　　编译器将检查类型向下转型是否合法，如果不合法将抛出异常。向下转换类型前，可以使用instanceof判断。

class Base { }

class Derived extends Base { }

public class Main {

public static void main(String[] args) {

Base base = new Derived();

if (base instanceof Derived) {

// 这里可以向下转换了

System.out.println("ok");

}

else {

System.out.println("not ok");

}

}

}

### 附注：instanceof 与 isinatance：

instanceof运算符： 只被用于对象引用变量，检查左边的被测试对象 是不是 右边类或接口的 实例化。如果被测对象是null值，则测试结果总是false。

形象地：自身实例或子类实例 instanceof 自身类 返回true

例： String s=new String("javaisland");

System.out.println(s instanceof String); //true

Class类的isInstance(Object obj)方法，obj是被测试的对象，如果obj是调用这个方法的class或接口的实例，则返回true。这个方法是instanceof运算符的动态等价。

形象地：自身类.class.isInstance(自身实例或子类实例) 返回true

例：String s=new String("javaisland");

System.out.println(String.class.isInstance(s)); //true

### 反射：运行时类信息

　　如果不知道某个对象的确切类型，RTTI可以告诉你，但是有一个前提：这个类型在编译时必须已知，这样才能使用RTTI来识别它。Class类与java.lang.reflect类库一起对反射进行了支持，该类库包含Field、Method和Constructor类，这些类的对象由JVM在启动时创建，用以表示未知类里对应的成员。这样的话就可以使用Contructor创建新的对象，用get()和set()方法获取和修改类中与Field对象关联的字段，用invoke()方法调用与Method对象关联的方法。另外，还可以调用getFields()、getMethods()和getConstructors()等许多便利的方法，以返回表示字段、方法、以及构造器对象的数组，这样，对象信息可以在运行时被完全确定下来，而在编译时不需要知道关于类的任何事情。

反射机制并没有什么神奇之处，当通过反射与一个未知类型的对象打交道时，JVM只是简单地检查这个对象，看它属于哪个特定的类。因此，那个类的.class对于JVM来说必须是可获取的，要么在本地机器上，要么从网络获取。所以对于RTTI和反射之间的真正区别只在于：RTTI，编译器在编译时打开和检查.class文件；反射，运行时打开和检查.class文件；

public class Person implements Serializable {

private String name;

private int age;

// get/set方法

}

public static void main(String[] args) {

Person person = new Person("luoxn28", 23);

Class clazz = person.getClass();

Field[] fields = clazz.getDeclaredFields();

for (Field field : fields) {

String key = field.getName();

PropertyDescriptor descriptor = new PropertyDescriptor(key, clazz);

Method method = descriptor.getReadMethod();

Object value = method.invoke(person);

System.out.println(key + ":" + value);

}

}

以上通过getReadMethod()方法调用类的get函数，可以通过getWriteMethod()方法来调用类的set方法。通常来说，我们不需要使用反射工具，但是它们在创建动态代码会更有用，反射在Java中用来支持其他特性的，例如对象的序列化和JavaBean等。

### 通过反射获取对象的实例：

方法一：通过Class类的无参getConstructors（）方法，获得Constructor对象数组，其长度等于反射类中实际构造方法的个数。然后，分别调用newInstance（）方法；

例：

1. class Demo *{*
2. *public static void main(String[] args) {*
3. *try {*
5. *//根据类名获取Class对象*
6. *Class c = Class.forName("Test");*
7. *//根据参数类型获取相应的构造函数*
8. Constructor constructor=c.getConstructors();
9. *//参数数组*
10. Object[] parameters=*{1,1}*;
11. *//根据获取的构造函数和参数，创建实例*
12. Object o=constructor[0].newInstance(parameters);
14. } catch (Exception e) *{*
15. *e.printStackTrace();*
16. *}*
17. }

备注：在构造方法的第一次调用时，构造方法编译时类似栈结构，先进后出。所以无参构造函数在数组的最后一个，前面是一个参数的，等等；

方法二：通过Class类的有参getConstructor（）方法，来获得一个构造方法信息，然后调用newINstance（）方法，完成实例的创建；

例：

1. class Demo *{*
2. *public static void main(String[] args) {*
3. *try {*
5. *//根据类名获取Class对象*
6. *Class c = Class.forName("Test");*
7. *//参数类型数组*
8. *Class[] parameterTypes={int.class,int.class}*;
9. *//根据参数类型获取相应的构造函数*
10. Constructor constructor=c.getConstructor(parameterTypes);
11. *//参数数组*
12. Object[] parameters=*{1,1}*;
13. *//根据获取的构造函数和参数，创建实例*
14. Object o=constructor.newInstance(parameters);
16. } catch (Exception e) *{*
17. *e.printStackTrace();*
18. *}*
19. }

**反射方法获取与调用：**

**A：获取所有方法 数组**

**1、getMethods 获取该类包括其父类的公共成员方法**

**2、getDeclaredMethods 获取该类的所有的成员方法**

**B：获取单个方法**

**1、getMethod　获取该类所有的公共类单个成员方法**

**2、getDeclaredMethod　获取该类所有类型的单个成员方法**

1 //获取字节码文件

2 Class c = Class.forName("zl\_Relfect\_01.Person");

3 　　　　//无参构造创建对象

4 Constructor con = c.getConstructor();

5 Object obj = con.newInstance();

6

7 //获取该类单个无参无返回值公共类型的成员方法

8 //无参的成员方法，只需写成员方法的名称

9 Method m1 = c.getMethod("show");

10 // public Object invoke(Object obj,Object... args) 返回值是Object接收,第一个参数表示对象是谁，第二参数表示调用该方法的实际参数

11 //show是无参，无返回值构造

12 m1.invoke(obj);//调用Person类的show方法

13

14

15 //获取该类单个有参有返回值公共类型的成员方法

16 Method m2 = c.getMethod("method", String.class);

17 //没有返回值

18 m2.invoke(obj, "哈喽");////调用Person类的method方法

19

20 //获取该类单个有参有返回值公共类型的成员方法

21 Method m3 = c.getMethod("getString", String.class,int.class);

22 //有返回值， public Object invoke(Object obj,Object... args) 返回值是Object接收

23 Object ob1 = m3.invoke(obj, "哈喽",123);

24 System.out.println(ob1);

25

26 //获取该类单个 无参无返回值的私有成员方法

27 Method m4 = c.getDeclaredMethod("function");

28 //取消语言访问检查

29 m4.setAccessible(true);

30 //调用Person类的function方法

31 m4.invoke(obj);

Filed常用方法：

1 package com.testReflect;

2

3 import java.lang.reflect.Field;

4 import java.lang.reflect.Modifier;

5

6 public class FieldTest {

7 public static void main(String[] args) throws Exception {

8 //使用反射第一步:获取操作类FieldDemo所对应的Class对象

9 Class<?> cls = Class.forName("com.testReflect.FieldDemo");

10 //使用FieldDemo类的class对象生成 实例

11 Object obj = cls.newInstance();

12

13 //通过Class类中getField(String name)： 获取类特定的方法，name参数指定了属性的名称

14 Field field = cls.getField("num1");

15

16 //拿到了Field类的实例后就可以调用其中的方法了

17

18 //方法:getModifiers() 以整数形式返回由此 Field 对象表示的字段的 Java 语言修饰符

19 System.out.println("修饰符: " + Modifier.toString(field.getModifiers()));

20

21 //方法:getType() 返回一个 Class 对象，它标识了此 Field 对象所表示字段的声明类型

22 System.out.println("类型: "+field.getType());

23

24 //方法:get(Object obj) 返回指定对象obj上此 Field 表示的字段的值

25 System.out.println("属性值: "+field.get(obj));

26

27 //方法: set(Object obj, Object value) 将指定对象变量上此 Field 对象表示的字段设置为指定的新值

28 field.set(obj, 55);

29 System.out.println("修改属性值后 --> get(Object obj): "+field.get(obj));

30 }

31 }

Method常用方法：

1 package com.testReflect;

2 import java.lang.reflect.Method;

3 import java.lang.reflect.Modifier;

4 public class MethodTest {

5 public static void main(String[] args) throws Exception {

6 //使用反射第一步:获取操作类MethodDemoFieldDemo所对应的Class对象

7 Class<?> cls = Class.forName("com.testReflect.MethodDemo");

8 //使用MethodDemo类的class对象生成 实例

9 Object obj = cls.newInstance();

10

11 //获取public int addResult(int addNum)方法

12 Method addMethod = cls.getMethod("addResult", new Class[]{int.class});

13 System.out.println("修饰符: " + Modifier.toString(addMethod.getModifiers()));

14 System.out.println("返回值: " + addMethod.getReturnType());

15 System.out.println("方法名称: " + addMethod.getName());

16 System.out.println("参数列表: " + addMethod.getParameterTypes());

17 int result = (int)addMethod.invoke(obj, 2);

18 System.out.println("调用addResult后的运行结果:" + result);

19

20 System.out.println("--------------------------------");

21

22 //获取public String toString() 方法

23 Method toStringMethod = cls.getMethod("toString", new Class[]{});

24 System.out.println("修饰符: " + Modifier.toString(toStringMethod.getModifiers()));

25 System.out.println("返回值: " + toStringMethod.getReturnType());

26 System.out.println("方法名称: " + toStringMethod.getName());

27 System.out.println("参数列表: " + toStringMethod.getParameterTypes());

28 String msg = (String)toStringMethod.invoke(obj);

29 System.out.println("调用toString后的运行结果:" + msg);

30 }

31 }

复制代码

**方法总结一：**

1、方法总结：

1.1获取class的三种方法：

第一种：Class clazz = Class.forName(XX.class的全包名)

第二种：Class clazz = XX xx = new XX();xx.getClass();

第三种：Class clazz = XX.class;

见实例代码test0()

1.2如何通过类class文件，拿到该文件里面的资源？这种方式获取的路径方法，永远是固定的，不会随着工程的路径改变而改变，始终都能获取到。

比如，TestClass.class文件（被编译后）所在的classes文件夹下，有prop.properties属性文件，如何拿到这个文件的绝对路径？如何把该文件加载到流中？通过在clazz中有个getClassLoader()方法，得到类加载器ClassLoader，然后再通过getSource()的方法，得到URL，然后再getPath()得到路径名

两种固定写法：

第一种：

Class clazz = TestClass.class;

ClassLoader classLoader = clazz.getClassLoader();

URL url = classLoader.getResource("prop.properties");

String path = url.getPath();

InputStream ins = new FileInputStream(path);

链式写法为：String path = TestClass.class.getClassLoader().getResource("prop.properties").getPath();

InputStream ins = new FileInputStream(path);

第二种[直接将资源加载成流]：InputStream ins = TestClass.class.getClassLoader().getSourceAsStream("prop.properties");

见实例代码test()

1.3如何用该class文件，创建一个对象？

创建对象有两种方法，直接用xx.newInstance()或使用xx.constructor()获取构造器然后再newInstance()。

第一种：

Class xx = TestClass.class;

Object obj = xx.newInstance();//无参构造方法

TestClass tc = (TestClass)obj;//转换为该类的对象

第二种：

Class xx = TestClass.class;

Constructor cst = xx.getConstructor();//无参构造方法

Object obj = cst.newInstance();//这个也不能有参数

TestClass tc = (TestClass)obj;//转换为该类的对象

见实例代码test1()、test2()

注意：这两种方法，只能使用无参的构造方法来创建无参默认的对象，如果想创建带参的对象呢？

使用构造器，然后指定参数类型，再创建实例对象时，将实参数传递

Class xx = TestClass.class;

Constructor cst = xx.getConstructor(String.class,int.class);//指定参数类型的带参构造方法

Object obj = cst.newInstance("李四",18);//顺便指定好实参

TestClass tc = (TestClass)obj;//转换为该类的对象

见实例代码test2()

1.4如何获取到该类创建的对象中的公有及私有属性值呢？需要用到xx.getField().get()或xx.getDeclaredField().setAccessible().get();

注意：获得公有私有的属性和方法时，必须指定获取哪个对象的，必须创建好一个对象。

Class xx = TestClass.class;

Constructor cst = xx.getConstructor(String.class,int.class);//指定参数类型的带参构造方法

Object obj = cst.newInstance("李四",18);//顺便指定好实参

TestClass tc = (TestClass)obj;//转换为该类的对象

//获取公有属性name：

Field fieldName = xx.getField("name");//指定哪个字段

Object name = fieldName.get(tc);//获取tc对象的name值

String strname = (String)name;

//获取私有属性age：

Field fieldAge = xx.getDeclaredField("age");//获取所有声明的一个叫age的字段

fieldAge.setAccessible(true);//使得该私有age可以被获取

Object age = fieldAge.get(tc)//获取tc的私有age

int intage = (int)age;

见实例代码test3()

1.5如何获取到该类创建的对象中的公有及私有的方法呢？xx.getMethod().invoke()或者xx.getDeclaredMethod().setAccessible().invoke();

注意：获得公有私有的属性和方法时，必须指定获取哪个对象的，必须创建好一个对象。

Class xx = TestClass.class;

Constructor cst = xx.getConstructor();//无参构造方法

Object obj = cst.newInstance();//这个也不能有参数

TestClass tc = (TestClass)obj;//转换为该类的对象

//获得公有的无参的方法:

Method mt1 = xx.getMethod("showPublic");//指定获取哪个方法（showPublic()方法）

mt1.invoke(tc);//获取对象tc中的showPublic()方法

//获得公有的带参的方法:

Method mt2 = xx.getMethod("showPublicParams", String.class,int.class);//指定获取哪个方法，并指定带哪种参数（showPublicParams(String,int)方法）

mt2.invoke(tc1, "刘二麻子",23);//获取对象tc中的showPublicParams并指定传进两个实参

//获得私有的无参方法【两步：1.获得声明的方法；2.让该方法可视化】

Method mt3 = xx.getDeclaredMethod("showPrivate");//获取哪个私有方法(showPrivate()方法)

mt3.setAccessible(true);//该方法可被获取，即可视化

mt3.invoke(tc);//获取对象tc中的showPrivate()方法

见实例代码test4()

1.6如果一个方法中的参数带了泛型，怎么获取该参数以及泛型参数的类型？如参数为(List,int)

见实例代码test5()。其中，getGenericParameterTypes只能获取方法的参数列表中的类型，返回的是一个类型数组[list类型,int类型];ParameterizedType是Type的子接口,需要向下转型为ParameterizedType，然后再使用方法getActualTypeArguments获取泛型的参数类型，返回的是个类型数组[People类型,String类型]。

见实例代码test5()

**其他反射常用方法：**

package reflect;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

import java.lang.reflect.Modifier;

import org.omg.CORBA.PUBLIC\_MEMBER;

/\*\*

\* 五个函数：（public static void）

\* ClassUtil.printClassMessage(Object obj);

\* ClassUtil.printFieldMessage(Object obj);

\* ClassUtil.printConstructorMessage(Object obj);

\* ClassUtil.printInterfacesMessage(Object obj);

\* ClassUtil.printSuperclassMessage(Object obj);

\*

\* @author sargeles

\*/

public class ClassUtil {

/\*\*

\* 打印类的信息，包括五个函数。

\*

\* @param obj

\*

\*/

public static void printClassMessage(Object obj) {

// 要获取类的信息，首先要获取类的类类型

Class c = obj.getClass();// 传递的是哪个子类的对象,c就是该子类的类类型

// 获取类的名称

System.out.println("类的名称是：" + c.getName());

System.out.println("类的成员方法有：");

/\*

\* Method类，方法对象 一个成员方法就是一个Method对象

\* getMethods()方法获取的是所有的public的方法，包括从父类继承而来的

\* getDeclareMethods()获取的是所有该类自己声明的方法，不问访问权限

\*/

Method[] ms = c.getMethods();

for (Method m : ms) {

// public int getModifiers()方法是公有的，可以获取修饰符信息，将其包装成一个int。

int mo = m.getModifiers();

// Modifier封装了解读带有修饰符信息的int的方法。这些方法都是静态的。可以直接用。

System.out.print(Modifier.toString(mo) + " ");

// 得到方法返回值类型的类类型

Class returnType = m.getReturnType();

System.out.print(returnType.getName() + " ");

// 得到方法的名称

System.out.print(m.getName() + "(");

// 获取参数类型--->得到的是参数列表的类型的类类型

Class[] paramTypes = m.getParameterTypes();

for (int i = 0; i < paramTypes.length; i++) {

if (i > 0)

System.out.print(",");

System.out.print(paramTypes[i].getName());

}

System.out.println(")");

}

}

/\*\*

\* 获取成员变量的信息（自己声明的）

\*

\* @param obj

\*/

public static void printFieldMessage(Object obj) {

// 要获取类的信息，首先要获取类的类类型

Class c = obj.getClass();// 传递的是哪个子类的对象,c就是该子类的类类型

System.out.println("类的名称是：" + c.getName());

System.out.println("类的成员变量有：");

/\*

\* 成员变量也是对象 java.lang.reflect.Field Field类封装了关于成员变量的操作

\* getFields()方法获取的是所有public的成员变量的信息

\* getDeclaredFields()获取的是该类自己声明的成员变量的信息

\*/

Field[] fs = c.getDeclaredFields();

for (Field field : fs) {

// 获得成员变量的类型的类类型

Class fieldType = field.getType();

String typeName = fieldType.getName();

// 得到成员变量的名字

String fieldName = field.getName();

System.out.println(typeName + " " + fieldName);

}

}

/\*\*

\* 获取类的构造函数信息

\*

\* @param obj

\*/

public static void printConstructorMessage(Object obj) {

Class c = obj.getClass();

System.out.println("类的名称是：" + c.getName());

System.out.println("类的构造函数有：");

/\*

\* 构造函数也是对象 java.lang.Constructor中封装了构造函数的信息

\* getConstructors获取所有的public的构造函数 getDeclaredConstructors得到所有的构造函数

\*/

Constructor[] cs = c.getDeclaredConstructors();

for (Constructor constructor : cs) {

// java.lang.Class.getModifiers()方法是公有本地的，可以获取修饰符信息，将其包装成一个int。

int mo = constructor.getModifiers();

// Modifier封装了解读带有修饰符信息的int的方法。这些方法都是静态的。可以直接用。

System.out.print(Modifier.toString(mo) + " ");

System.out.print(constructor.getName() + "(");

// 获取构造函数的参数列表--->得到的是参数列表的类类型

Class[] paramTypes = constructor.getParameterTypes();

for (int i = 0; i < paramTypes.length; i++) {

if (i > 0)

System.out.print(",");

System.out.print(paramTypes[i].getName());

}

System.out.println(")");

}

}

/\*\*

\* 类的接口信息

\*

\* @param obj

\*/

public static void printInterfacesMessage(Object obj) {

Class c = obj.getClass();

System.out.println("类的名称是：" + c.getName());

System.out.println("类实现的接口有：");

/\*

\* 接口类也是对象，因为接口也是类类型的一种，所以是Class的泛型。

\*/

Class[] intes = c.getInterfaces();

for (Class classes : intes) {

// 获取每个接口的名字，然后输出。

System.out.println(classes.getName());

}

}

/\*\*

\* 类的父类信息

\*

\* @param obj

\*/

public static void printSuperclassMessage(Object obj) {

Class c = obj.getClass();

System.out.println("类的名称是：" + c.getName());

System.out.print("类的父类是：");

/\*

\* 父类只可能有一个，所以不用遍历，单根继承。

\*/

Class superclass = c.getSuperclass();

System.out.println(superclass.getName());

}

}