java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methods1。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。  
       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。  
  
       目前好多框架都会用到java的反射机制。比如struts2,sping,hibernate。  
如果我们不用struts2，自己写一个类似的功能也是可以实现的，比如浏览器通过HTTP发送数据，而这些数据都是字符串，我们接受到这些字符串时， 可以通过反射去构造一个对象(通过拦截器做成框架的功能)，这样就可以用对象的get和set方法了，而不用原始的getPeremter方法。事实上， 在struts2出来之前，我们又不想用struts1的ActionForm就做过这样项目。  
一、Class object 的产生方式有以下几种。  
  
1、运用getClass()  
  
注：每个class 都有此函数  
      
String str = "abc";  
Class c1 = str.getClass();  
  
2、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");  
  
3、运用.class 语法  
      
Class c1 = String.class;  
Class c2 = java.awt.Button.class;  
  
4、运用primitive wrapper classes的TYPE 语法  
  
Class c1 = Integer.TYPE;  
Class c2 = Long.TYPE;  
  
  
二、Java类反射中的主要方法  
  
对于以下三类组件中的任何一类来说 -- 构造函数、字段和方法 -- java.lang.Class 提供四种独立的反射调用，以不同的方式来获得信息。调用都遵循一种标准格式。以下是用于查找构造函数的一组反射调用：  
  
Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的公共构造函数，  
Constructor[] getConstructors() -- 获得类的所有公共构造函数  
Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(与接入级别无关)  
Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数(与接入级别无关)   
  
获得字段信息的Class 反射调用不同于那些用于接入构造函数的调用，在参数类型数组中使用了字段名：  
  
Field getField(String name) -- 获得命名的公共字段  
Field[] getFields() -- 获得类的所有公共字段  
Field getDeclaredField(String name) -- 获得类声明的命名的字段  
Field[] getDeclaredFields() -- 获得类声明的所有字段   
  
用于获得方法信息函数：  
  
Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法  
Method[] getMethods() -- 获得类的所有公共方法  
Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法  
Method[] getDeclaredMethods() -- 获得类声明的所有方法   
  
  
三、以下是代码实例：  
  
import java.lang.reflect.Constructor;  
import java.lang.reflect.Field;  
import java.lang.reflect.Method;  
public class ReflectTester {  
    /\*\*  
    \* 通过java的反射机制获取类的所有属性和方法  
    \*/  
    public void test1() {  
        try {  
            Class c = Class.forName("demo1.client.Customer");  
            System.out.println("属性：");  
            Field f[] = c.getDeclaredFields();  
            for (int i = 0; i < f.length; i++) {  
                System.out.println(f[i].getName());  
            }  
            System.out.println("方法：");  
            Method m[] = c.getDeclaredMethods();  
            for (int i = 0; i < m.length; i++) {  
                System.out.println(m[i].toString());  
            }  
  
        } catch (Throwable e) {  
            System.err.println(e);  
        }  
    }  
  
    /\*\*  
    \* 通过java的反射机制动态修改对象的属性  
    \* @param o  
    \*/  
    public void test2(Customer o) {  
  
        try {  
            Class c = o.getClass();  
            //getMethod方法第一个参数指定一个需要调用的方法名称，第二个参数是需要调用方法的参数类型列表，如无参数可以指定null，该方法返回一个方法对象　  
            Method sAge = c.getMethod("setAge", new Class[] { int.class });  
            Method gAge = c.getMethod("getAge", null);  
            Method sName = c.getMethod("setName", new Class[] { String.class });  
            //动态修改Customer对象的age  
            Object[] args1 = { new Integer(25) };  
            sAge.invoke(o, args1);  
            //动态取得Customer对象的age  
            Integer AGE = (Integer) gAge.invoke(o, null);  
            System.out.println("the Customer age is: " + AGE.intValue());  
            //动态修改Customer对象的name  
            Object[] args2 = { new String("李四") };  
            sName.invoke(o, args2);  
  
        } catch (Throwable e) {  
            System.err.println(e);  
        }  
    }  
    /\*\*  
    \* 通过java的反射机制做一个简单对象的克隆  
    \* @param o  
    \* @return  
    \*/  
    public Object test3(Customer o) {  
        Object o2 = null;  
        try {  
            Class c = o.getClass();  
            //通过默认构造方法创建一个新的对象   
            o2 = c.getConstructor(new Class[] {}).newInstance(  
                    new Object[] {});  
            Field fields[] = c.getDeclaredFields();  
            for (int i = 0; i < fields.length; i++) {  
                Field field = fields[i];  
                String fieldName = field.getName();  
                String firstLetter = fieldName.substring(0, 1).toUpperCase();  
                // 获得和属性对应的getXXX()方法的名字   
                String getMethodName = "get" + firstLetter + fieldName.substring(1);  
                // 获得和属性对应的setXXX()方法的名字   
                String setMethodName = "set" + firstLetter + fieldName.substring(1);   
                // 获得和属性对应的getXXX()方法   
                Method getMethod = c.getMethod(getMethodName, new Class[] {});   
                // 获得和属性对应的setXXX()方法   
                Method setMethod = c.getMethod(setMethodName, new Class[] { field.getType() });   
                // 调用原对象的getXXX()方法   
                Object value = getMethod.invoke(o, new Object[] {});  
               // 调用拷贝对象的setXXX()方法   
                setMethod.invoke(o2, new Object[] { value });  
            }  
        } catch (Throwable e) {  
            System.err.println(e);  
        }  
        return o2;  
    }  
  
    public static void main(String[] args) throws Exception {  
        ReflectTester t = new ReflectTester();  
        t.test1();  
        Customer customer = new Customer();  
        customer.setAge(20);  
        customer.setName("张三");  
        System.out.println("调用前name: " + customer.getName());  
        System.out.println("调用前age: " + customer.getAge());  
        t.test2(customer);  
        System.out.println("调用后name: " + customer.getName());  
        System.out.println("调用后age: " + customer.getAge());  
        Customer customer2 = (Customer)t.test3(customer);  
        System.out.println("克隆对象的name: " + customer2.getName());  
        System.out.println("克隆对象的age: " + customer2.getAge());  
    }  
}  
  
class Customer {  
    private long id;  
    private String name;  
    private int age;  
  
    public Customer() {  
    }  
    public Customer(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public long getId() {  
        return id;  
    }  
    public void setId(long id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
运行结果：  
属性：  
id  
name  
age  
方法：  
public java.lang.String demo1.client.Customer.getName()  
public long demo1.client.Customer.getId()  
public void demo1.client.Customer.setName(java.lang.String)  
public void demo1.client.Customer.setAge(int)  
public int demo1.client.Customer.getAge()  
public void demo1.client.Customer.setId(long)  
调用前name: 张三  
调用前age: 20  
the Customer age is: 25  
调用后name: 李四  
调用后age: 25  
克隆对象的name: 李四  
克隆对象的age: 25

**扩展-**-Class还可以获得一个类的接口和继承类信息

**interface** itf1{}

**interface** itf2{}

**class** Animal{

**int** y=99;

**static** **final** **int** *yy*=10;

**final** **int** u=100;

**static** **int** *k*=90;

Animal(){

System.*out*.println("加载Animal类了");

}

/\*\*

\* static域的代码只加载一次

\*/

**static**{

System.*out*.println("Animal的static字句");

}

**public** **void** h(){

System.*out*.println("function for Animal");

}

**public** **static** **void** j(){}

}

**class** Brid **extends** Animal **implements** itf1,itf2{

**private** **int** op=10;

Brid(){

System.*out*.println("加载Brid类了");

}

**static**{

System.*out*.println("Brid的static字句");

}

**public** **void** h(){

System.*out*.println("function for Brid"+**super**.*yy*);

}

**public** **static** **void** j(){}

}

**public** **class** TestClass {

/\*\*

\* **@param** args

\* **@throws** ClassNotFoundException

\*/

**public** **static** **void** main(String[] args){

Animal cl=**new** Animal();

*showName*(cl);

System.*out*.println(**int**.**class**);

**int**[] o={};

System.*out*.println(**int**[].**class**);

Animal a=**new** Animal();

Class c=**null**;

**try** {

c=Class.*forName*("com.java.Class.Brid");

System.*out*.println("打印Brid的类型信息--------------------");

*printInfo*(c);

System.*out*.println("打印Brid实现的接口的类型信息-------------------");

**for**(Class c1:c.getInterfaces()){

*printInfo*(c1);

}

/\*\*

\* getSuperclass();得到调用Class的基类的Class

\*/

Class c2=c.getSuperclass();

System.*out*.println("打印Brid实现的父类的类型信息-------------------");

*printInfo*(c2);

Object ob=**null**;

/\*\*

\* newInstance();//创建此Class对象所表示的类的一个实例,得到的是Object类型的，需要转换成相应的类型才能使用

\*/

ob=c2.newInstance();

Animal al=(Animal)ob;

System.*out*.println("ob="+ob+" al="+al+" ");

al.h();

System.*out*.println("====打印父类的类型信息=================================");

*printInfo*(ob.getClass());

/\*\*

\* 创建实例对象的另一种方法

\*/

Animal aa=Animal.**class**.newInstance();

aa.h();

System.*out*.println(aa.y);

/\*\*

\* 创建实例对象的另一种方法

\*/

System.*out*.println("测试创建实例对象---------------------------------------------------");

Brid b=(Brid)c.newInstance();

b.h();

String s="";

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (InstantiationException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**static** **void** showName(Object o){

System.*out*.println(o.getClass().getName());

}

**public** **static** **void** printInfo(Class c){

/\*\*

\* getName():返回全限定类名（即包名和类名）

\* getSimpleName()：返回不包含包名的类名

\* getCanonicalName():全限定类名

\*/

System.*out*.println("c.getName()="+c.getName()+" c.getSimpleName()="+c.getSimpleName()+" c.getCanonicalName()="+c.getCanonicalName());

System.*out*.println("是否是接口="+c.isInterface());

}

}