

## 1. 纲要

- a) 反射的基本概念
- b) Java 中的类反射
- c) 安全性和反射
- d) 反射的两个缺点

## 2. 内容

### 1.1 反射的基本概念

反射的概念是由 Smith 在 1982 年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。Java 中，反射是一种强大的工具。它使您能够创建灵活的代码，这些代码可以在运行时装配，无需在组件之间进行源代码链接。反射允许我们在编写与执行时，使我们的程序代码能够接入装载到 JVM 中的类的内部信息，而不是源代码中选定的类协作的代码。这使反射成为构建灵活的应用的主要工具。**但需要注意的是：如果使用不当，反射的成本很高。**

### 1.2 Java 中的类反射

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序对自身进行检查，或者说“自审”或“自省”，并能直接操作程序的内部属性。Java 的这一能力在实际应用中也许用得不是很多，但是在其它的程序设计语言中根本就不存在这一特性。例如，Pascal、C 或者 C++ 中就没有办法在程序中获得函数定义相关的信息。

#### 2.2.1 reflection 的工作机制

程序运行时，java 系统会一直对所有对象进行所谓的运行时类型识别，这项信息记录了每个对象所属的类。通过专门的类可以访问这些信息。用来保存这些信息的类是 class 类，class 类为编写可动态操纵的 java 代码程序提供了强大功能

构造 Class 对象有 3 种方式：

## 1、Class.forName();

```
try {  
    // 构造 Class 对象的第一种方法  
    Class clazz = Class.forName("java.lang.String");  
    Object obj = clazz.newInstance();  
    System.out.println(obj);  
} catch ( ClassNotFoundException e ) {  
    e.printStackTrace();  
} catch ( IllegalAccessException e ) {  
    e.printStackTrace();  
} catch ( InstantiationException e ) {  
    e.printStackTrace();  
}
```

## 2、类.class

```
try {  
    // 构造 Class 对象的第二种方法  
    Class stringClass = String.class;  
    System.out.println(stringClass);  
} catch ( ClassNotFoundException e ) {  
    e.printStackTrace();  
} catch ( IllegalAccessException e ) {  
    e.printStackTrace();  
} catch ( InstantiationException e ) {  
    e.printStackTrace();  
}
```

## 3、Object.getClass()


```
try {  
    // 构造 Class 对象的第三种方法
```

```
String s = "s";
stringClass = s.getClass();
System.out.println(stringClass);
} catch ( ClassNotFoundException e ) {
    e.printStackTrace();
} catch ( IllegalAccessException e ) {
    e.printStackTrace();
} catch ( InstantiationException e ) {
    e.printStackTrace();
}
```

类对象的比较:

相同的类:

```
Class clazz = Class.forName("java.lang.String");
Class stringClass = String.class;
System.out.println("字符串类对象的比较="+ (clazz == stringClass));
```



```
D:\share\03-J2SE\lesson\less10>java ClassRefTest
字符串类对象的比较=true
```

不同的类:

```
Class stringClass = String.class;
Class intClass = int.class;
System.out.println("字符串类对象和 Int 类对象的比较=" + (stringClass == intClass));
```



```
D:\share\03-J2SE\lesson\less10>java ClassRefTest
字符串类对象和Int类对象的比较=false
D:\share\03-J2SE\lesson\less10>
```

## 2.2.2 Java 反射中的主要类和方法

**软件包 `java.lang.reflect`**

提供类和接口，以获取关于类和对象的反射信息。

## 1、Constructor 构造函数对象

```
class A {  
    public A() {  
  
    }  
    public A( String s ) {  
  
    }  
}  
...  
  
A a = new A();  
Class aClass = a.getClass();  
//得到类对象的所有公共的构造函数对象  
Constructor[] constructors = aClass.getConstructors();  
// 得到类对象特定的公共构造函数对象  
Constructor c = aClass.getConstructor(String.class);  
// 获取全部声明的构造方法  
Constructor[] c1 = aClass.getDeclaredConstructors();  
    for ( Constructor c1 : constructors ) {  
        System.out.println( "构造方法的名称="+ c1.getName() );  
        System.out.println( "  构造 方法 的 修 饰 符 =" +  
Modifier.toString(c1.getModifiers()));  
        Class[] clazz1 = c1.getParameterTypes();  
        for ( Class cs : clazz1 ) {  
            System.out.println("参数类型:"+cs.getName());  
        }  
    }  
}
```

## 2、 Method

```
Method[] ms = aClass.getDeclaredMethods();
for ( Method ms1 : ms ) {
    System.out.println();
    System.out.println( "方法的名称=" + ms1.getName() );
    System.out.println( "方法的修饰符=" + ms1.getModifiers() + ":"
+ Modifier.toString(ms1.getModifiers()) );
    System.out.println( "方法的修饰符是否为 public=" +
Modifier.isPublic(ms1.getModifiers()) );
    Class[] clazz1 = ms1.getParameterTypes();
    for ( Class cs : clazz1 ) {
        System.out.println("参数类型:"+cs.getName());
    }
    System.out.println( "方法是否带有可变数量的参数=" +
ms1.isVarArgs() );
    System.out.println( "方法的返回类型 :
"+ms1.getReturnType().getName());
}
```

## 3、 Field

```
Field[] f = aClass.getDeclaredFields();
B b = new B();
for ( Field f1 : f ) {
    System.out.println();
    System.out.println( "变量的名称=" + f1.getName() );
    System.out.println( "变量的修饰符 =" +
Modifier.toString(f1.getModifiers()) );
    System.out.println( "变量的类型=" + f1.getType().getName() );
    System.out.println( "变量的值=" + f1.get(b) );
}
```

#### 4、Class

```
Class[] classes = aClass.getInterfaces();  
for ( Class c3 : classes ) {  
    System.out.println("接口名称: " + c3.getName());  
}  
// 获取类对象的父类  
Class c4 = aClass.getSuperclass();  
System.out.println("父类名称: " + c4);  
// 获取类对象的包对象  
String pName = String.class.getPackage().getName();  
System.out.println("String 所在的包名称: " + pName);  
  
System.out.println("aClass 是否为接口: " + aClass.isInterface());  
System.out.println("C 是否为接口: " + C.class.isInterface());  
  
System.out.println("类名: " + String.class.getName());  
System.out.println("类名的简称: " + String.class.getSimpleName());
```

### 2.2.3 开始使用 Reflection

#### 1、打印一个类声明所有内容

```
import java.lang.reflect.*;  
class ClassRefTest1 {  
    public static void main(String[] args) {  
        try {  
            Class c = Test.class;  
            //先判断类对象的类型  
            String classType = "";  
            if ( c.isInterface() ) {
```

```
        classType = "interface ";
    } else {
        classType = "class ";
    }
    int mod = c.getModifiers();
    Class[] iClass = c.getInterfaces();
    String classInterface = "";
    if ( iClass != null && iClass.length != 0 ) {
        for ( int i = 0; i < iClass.length; i++ ) {
            classInterface+=iClass[i].getSimpleName();
            if ( i != iClass.length - 1 ) {
                classInterface+=" ";
            }
        }
    }
    Class sClass = c.getSuperclass();
    String pName = c.getPackage() == null ? "" :
c.getPackage().getName();
    String classSuper = sClass.getSimpleName();

    if ( pName != null && !"".equals( pName ) ) {
        System.out.println( "package " + pName );
    }
    String classModifier = Modifier.toString(mod);
    if ( !"".equals(classModifier) ) {
        System.out.print( classModifier + " ");
    }
    System.out.print( classTypes + c.getSimpleName() );
    if ( sClass != null ) {
        System.out.print( " extends " + classSuper );
    }
}
```

```
    }
    if ( !"".equals(classInterface) ) {
        System.out.print( " implements " + classInterface);
    }
    System.out.println( " {" );
    getConstr(c);
    getField(c);
    getMethod(c);
    System.out.println( "}" );
} catch (Exception e) {
    e.printStackTrace();
}

}

private static void getConstr(Class c) throws Exception {
    Constructor[] constructors = c.getDeclaredConstructors();
    if ( constructors != null && constructors.length != 0 ) {
        for ( Constructor con : constructors ) {
            int mod = con.getModifiers();
            String conModifier = Modifier.toString(mod);
            String conName = con.getName();
            Class[] clazzes = con.getParameterTypes();
            String conParams = "";
            if ( clazzes != null && clazzes.length != 0 ) {
                for ( int i = 0; i < clazzes.length; i++ ) {
                    conParams+=clazzes[i].getName();
                    if ( i != clazzes.length - 1 ) {
                        conParams+=" ";
                    }
                }
            }
        }
    }
}
```



```
        }

        System.out.print( "      " );
        if ( conModifier != null && !"".equals(conModifier) ) {
            System.out.print( conModifier + " " );
        }
        System.out.print(conName + "( ");
        if ( !"".equals(conParams) ) {
            System.out.print(conParams);
        }
        System.out.println(") {");
        System.out.println("    }");
    }
}

private static void getField(Class c) throws Exception {
    //TODO 留为作业
}

private static void getMethod(Class c) throws Exception {
    //TODO 留为作业
}
}

class A {

}

interface B {

}

class Test extends A implements B {
    public Test() {
```

```
}  
public Test(String s) {  
  
}  
  
public String s;  
private int i;  
  
public static void test() {  
  
}  
public void test1() {  
  
}  
public void test1(String s) {  
  
}  
private void test2() {  
  
}  
}
```

## 2、构造对象

```
Class<Test> c = Test.class;  
// 第一种构造对象的方法  
//Test t = c.newInstance();//当类没有无参构造方法时,使用此方法构造对象失败  
// 第二种构造对象的方法  
Constructor<Test> con = c.getConstructor(String.class);
```

```
Test t = con.newInstance("abc");
System.out.println("Test 对象的 S 属性值: "+t.s);

...

class Test extends A implements B {
    //public Test() {
    //
    //}
    public Test(String s) {
        this.s = s;
    }

    public String s = "s";
    private int i = 10;
```

### 3、方法

```
// 对象方法的普通调用
t.test1("abc");
// 对象方法的反射调用
Method m = c.getMethod("test3");
m.invoke(null);//调用静态方法
```

### 4、变量

```
Field f = c.getDeclaredField("i");
System.out.println("f="+f);
//System.out.println("t.i="+f.get(t));
System.out.println("是否支持 Java 语言访问检查:"+!f.isAccessible());
f.setAccessible(true);
System.out.println("t.i before="+f.get(t));
f.set(t, 20);
```

```
System.out.println("t.i after="+f.get(t));
```

### 1.3 2.3、安全性和反射

在处理反射时安全性是一个较复杂的问题。反射经常由框架型代码使用，由于这一点，我们可能希望框架能够全面介入代码，无需考虑常规的介入限制。但是，在其它情况下，不受控制的介入会带来严重的安全性风险，例如当代码在不值得信任的代码共享的环境中运行时。

### 1.4 2.4、反射的两个缺点

反射是一种强大的工具，但也存在一些不足。

- 性能问题。使用反射基本上是一种解释操作，我们可以告诉JVM，我们希望做什么并且它满足我们的要求。用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运行中相对很少涉及的部分，缓慢的性能将不会是一个问题。
- 使用反射会模糊程序内部实际要发生的事情。程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术会带来维护问题。反射代码比相应的直接代码更复杂。解决这些问题的最佳方案是保守地使用反射——仅在它可以真正增加灵活性的地方——记录其在目标类中的使用。