

## Chp16 反射

反射是 Java 中非常重要的一个语言特性，反射的强大和完善，让 Java 语言在工程实践中的灵活性大大的增强，使得 Java 程序在运行时可以探查类的信息，动态的创建类的对象，获知对象的属性，调用对象的方法。因此，反射技术被广泛的应用在一些工具和框架的开发上。

也许，并不是每一个程序员都有机会利用反射 API 进行他们的 Java 开发，但是，学习反射是一个 Java 程序员必须要走过的道路之一，对反射的掌握能够帮助程序员更好的理解后面很多的框架和 Java 工具，毕竟这些框架和工具都是采用反射作为底层技术的。

首先，先来看看几个编程中的问题。给出下面两个需求：

1、给定一个对象，要求输出这个对象所具有的所有方法的名字。即，写出类似下面的函数：

```
public static void printMethod(Object obj)
```

2、给定一个字符串参数，这个参数表示一个类的名字。根据类名，创建该类的一个对象并返回。即写出类似下面这种定义的函数：

```
public static Object createObject(String className)
```

思考一下，用现在的知识，能做到这一点么？有一点困难吧。

## 1 类对象

### 1.1 概念

要理解反射，首先要理解的是“类对象”的概念。

Java 中有一个类，`java.lang.Class` 类。这个类的对象，就被称之为类对象。

那类对象用来干什么呢？比如，以前我们写过学生类，一个学生对象都是用来保存一个学生的信息。而一个类对象呢，则用来保存一个类的信息。所谓类的信息，包括：这个类继承自哪个类，实现了哪些接口，有哪些属性，有哪些方法，有哪些构造方法……等等。

我们之前提到过类加载的概念。当 JVM 第一次遇到某个类的时候，会通过 `CLASSPATH` 找到相应的 `.class` 文件，读入这个文件并把读到的类的信息保存起来。而类的信息在 JVM 中，则被封装在了类对象中。

这种阐述比较抽象，我们可以举一个非常形象的例子。我们在动物园中，能够看到动物，我们见到的都是活生生的对象。例如，我们在笼子中见到了三条狗，那实际上是说，我们遇到了三个狗对象。

在关着狗的笼子外面，一般会有一个牌子。方便游人更好的认识这种动物。

例如，牌子上会有这样的信息：狗，脊椎动物门，哺乳纲，食肉目，犬科。这部分信息，实际上是在说明，狗的父类是什么，介绍的是狗这个类的继承关系。

狗能当宠物，这说明的是狗实现了什么接口。

狗有四条腿，脚上有每个脚上有 4 个脚趾，有尾巴…… 这些，表明的都是狗有什么，实际上说明的是狗的属性。

狗吃肉，能看家，能拉雪橇……这说明的是狗有哪些方法。

狗什么时候繁殖，一胎生多少只小狗……这是狗的构造方法。

换句话说，在动物园的牌子上，写满了狗这个类的信息。（注意，牌子上写的不是对象的信息。对象信息是什么呢？比如，笼子里的狗叫什么名字，是公还是母，年龄多大……显然这些不会写在牌子上）

下面，我们思考这个牌子。首先，这个牌子也是一个对象；其次，这个牌子对象的作用，就是用来保存狗这个类的信息。

因此，我们所说的类对象，就非常类似于动物园里的牌子：这种对象的创建，就是为了保存类的信息。

## 1.2 获取类对象

接下来，我们来介绍一下如何获得类对象。获得类对象总共有三种方式。

### 1.2.1 类名.class

可以通过类名.class 的方法直接获得某个类的类对象。例如，如果要获得 Student 类的类对象，就可以使用 `Class c = Student.class`。

这种方法获得类对象比较直接，并且还有一个非常重要的特点。对于基本类型来说，他们也有自己的类对象，但是要获得基本类型的类对象，只能通过类型名.class 来获得类对象。例如下面的代码，就能获得 int 类型的类对象

```
Class c = int.class;
```

### 1.2.2 getClass()方法

Object 类中定义了一个 getClass()方法，这个方法也能获得类对象。我们之前曾经介绍过这个方法，前文中，把这个方法称之为获得对象的实际类型。而现在我们可以知道，这个方法实际上是返回某个对象的类对象。

例如，我们可以对一条狗（狗对象）调用它的 getClass()方法，此时，它会叼着那块牌子返回给你。

另外，由于多条狗公用一个牌子，也就是说，同一类型的对象公用一个类对象，因此，对同一类型的任何一个对象调用 getClass()方法，返回的应该是同一个类对象。

正因为如此，对类对象的比较，可以使用“==”。可以回顾一下 equals 方法的写法，在比较实际类型时，使用的是 getClass()方法，用“==”比较两个类对象。

当我们有一个对象，而想获得这个对象的类对象时，应当调用这个对象的 getClass()方法。因此，getClass()方法主要用于：通过类的对象获得类对象。

### 1.2.3 Class.forName()方法

在 Class 类中有一个静态方法，这个静态方法叫做 forName。方法的签名如下：

```
public static Class forName(String className) throws ClassNotFoundException
```

这个方法接受一个字符串作为参数，这个字符串参数表示一个类的类名。这个静态方法能够根据类名返回一个类对象。

需要注意的是两点：

- 1、当 className 所代表的类不存在时，这个方法会抛出一个已检查异常 ClassNotFoundException。

- 2、这个方法接受的字符串参数，必须带包名。举例来说，如果想要利用 Class.forName 获得 ArrayList 这个类的类对象的话，必须使用 `Class.forName("java.util.ArrayList")` 这样的方式获得类对象，而不是 `Class.forName("ArrayList")`。

这种获得类对象的方式还有其他的作用。考虑下面的代码：

```
public static void main(String args[]) throws Exception{
    Class c = Class.forName("p1.p2.TestClass");
}
```

在上面的代码中，主方法中利用 `Class.forName()` 获得 `p1.p2.TestClass` 类的类对象。而我们知道，类对象是在类加载之后才会产生的。在调用 `Class.forName()` 方法的时候，`p1.p2.TestClass` 这个类并没有被加载。因此，要获得这个类的类对象，必须要先加载这个类。`Class.forName()` 就会触发类加载的动作。

有些时候，我们可以用 `Class.forName()` 这个方法，来进行“强制类加载”的操作。

获得类对象之后，接下来要做的事情就是使用类对象。获得的类对象究竟有什么用处呢？应该如何使用呢？这就是我们接下来要介绍的。

为了介绍类对象的使用，我们首先创建一个 **Student** 类。代码如下：

```
public class Student {
    public String name;
    private int age;

    public Student() {}

    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;
    }

    public void study(){
        System.out.println(name + " study");
    }

    public void study(int h){
        System.out.println(name + " study for " + h + " hours");
    }

    public double study(int a, double b){
```

```

        System.out.println(name + " study " + a + " " + b);
        return a * b;
    }

    private void play(){
        System.out.println(name + " play");
    }

}

```

这个 `Student` 类中包括两个属性，一个公开的 `name` 属性和一个私有的 `age` 属性，并对这两个私有属性提供了相应的 `get/set` 方法。

此外，`Student` 类还定义了三个重载的 `study` 方法，并定义了一个私有的 `play` 方法。

接下来，用 `Student` 类的类对象来演示如何使用类对象。

### 1.3 使用类对象获取类的信息

当我们获得了类对象，当然就可以调用类对象中的方法。例如：

- `getName()`: 获得类的名称，包括包名
- `getSimpleName()`: 获得类的名称，不包括包名
- `getSuperClass()`: 获得本类的父类的类对象
- `getInterfaces()`: 获得本类所实现的所有接口的类对象，返回值类型为 `Class[]`，当然，这是对的，一个类可以实现多个接口。

我们来看如下代码：

```

import java.util.ArrayList;
public class TestClass1 {
    public static void main(String[] args) {
        Class c = ArrayList.class;

        String className = c.getName();
        System.out.println("类名: "+className);

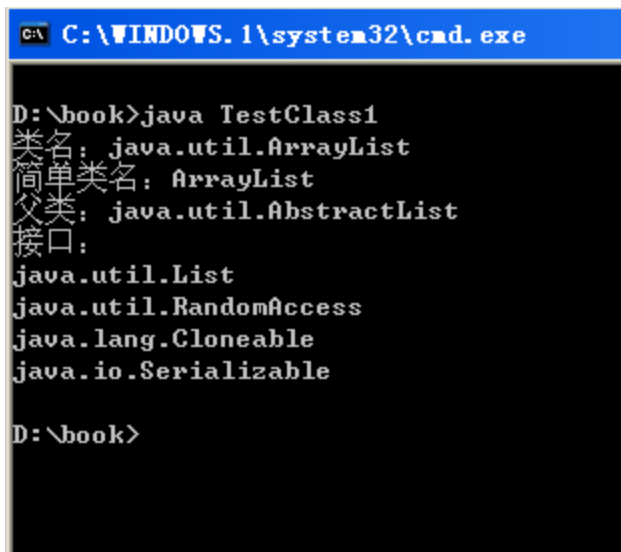
        String simpleName = c.getSimpleName();
        System.out.println("简单类名: "+simpleName);

        Class superClass = c.getSuperclass();
        System.out.println("父类: "+superClass.getName());

        Class[] interfaces = c.getInterfaces();
        System.out.println("接口: ");
        for(int i =0 ; i < interfaces.length ; i++){
            System.out.println(interfaces[i].getName());
        }
    }
}

```

运行结果：



```
C:\WINDOWS.1\system32\cmd.exe

D:\book>java TestClass1
类名: java.util.ArrayList
简单类名: ArrayList
父类: java.util.AbstractList
接口:
java.util.List
java.util.RandomAccess
java.lang.Cloneable
java.io.Serializable

D:\book>
```

该程序通过分析类对象，打印出了 ArrayList 类的父类以及所实现的接口，和 API 文档中提示的是一致的。

## 1.4 使用类对象获取类中方法的信息

在 Class 类中，有两个方法，这两个方法签名如下：

```
public Method[] getDeclaredMethods() throws SecurityException
```

```
public Method[] getMethods() throws SecurityException
```

这两个方法都抛出 SecurityException 异常，这个异常是一个未检查异常，可处理可不处理。

这两个方法都返回一个 Method 类型的数组。Method 类是在 java.lang.reflect 包中定义的一类。Method 类用来表示方法，一个 Method 对象封装了一个方法的信息。我们可以调用 Method 对象的 getName() 方法获得方法名，也可以直接调用 Method 对象的 toString() 方法直接返回方法的签名。

以上所述的两个方法，都可以返回类中所有方法的信息，由于一个方法的信息会封装在一个 Method 对象中，因此，返回值类型均为 Method[]。

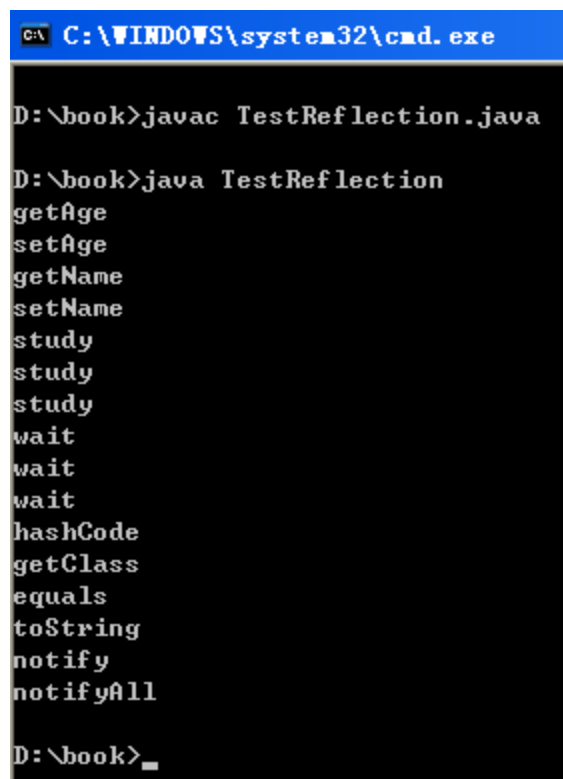
同样是返回 Method 数组，那 getMethods() 和 getDeclaredMethods() 有什么区别呢？

对于 getMethods() 来说，返回的 Method 类型的数组中，包括所有的公开方法，也包括父类中定义的公开方法。对于 Student 类来说，既包括在 Student 类中定义的 study 以及一系列 get/set 方法，也包括在 Object 类中定义而被 Student 类继承的方法，例如 toString、equals 方法等。但是，私有方法不会被获取。演示代码如下：

```
import java.lang.reflect.*;

public class TestReflection {
    public static void main(String[] args) {
        Class c = Student.class;
        Method[] ms = c.getMethods();
        for(int i = 0; i<ms.length; i++){
            System.out.println(ms[i]);
        }
    }
}
```

运行结果如下：



```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestReflection.java

D:\book>java TestReflection
getAge
setAge
getName
setName
study
study
study
wait
wait
wait
hashCode
getClass
equals
toString
notify
notifyAll

D:\book>
```

可以看出，包括了父类的中的公开方法，但是不包括任何的私有方法（例如 play 方法）。

而 `getDeclaredMethods` 方法，则会返回在本类中定义的所有方法，包括私有方法。也就是说，对于 `Student` 类的类对象而言，调用 `getDeclaredMethods` 方法会获得在 `Student` 类中定义的所有方法，包括私有的 `play` 方法。但是，不能获得父类中的任何方法。演示代码如下：

```
import java.lang.reflect.*;

public class TestReflection {

    public static void main(String[] args) {
        Class c = Student.class;
        Method[] ms = c.getDeclaredMethods();
        for(int i = 0; i<ms.length; i++){
            System.out.println(ms[i].toString());
        }
    }
}
```

运行结果如下：

```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestReflection.java

D:\book>java TestReflection
public int Student.getAge()
public void Student.setAge(int)
public java.lang.String Student.getName()
public void Student.setName(java.lang.String)
public void Student.study(int)
public double Student.study(int,double)
public void Student.study()
private void Student.play()

D:\book>
```

可以看到，相比前一个程序，结果中多出了 play 方法，但是少了很多 Object 类中的方法。并且，相对于 getName 只能获得方法名，Method 对象的 toString 方法能够获得对象的完整签名。

至此，我们可以完成本章开始时提出的需求 1：给定一个对象，输出这个对象的所有方法的名字。示例代码如下：

```
public static void printMethod(Object obj){
    //获取obj对象所对应的类对象
    Class c = obj.getClass();

    //通过类对象，获取其中的所有方法对象
    Method[] ms = c.getMethods();

    //打印每个方法的方法名
    for(int i = 0 ; i < ms.length ; i++){
        System.out.println(ms.getName());
    }
}
```

## 1.5 使用类对象创建类的对象

类对象除了能够获知类中有哪些方法之外，还有着很多其他很有价值的功能。例如，在 Class 类中有一个方法：newInstance()，这个方法能够通过类的无参构造方法，创建一个对象。也就是说，我们可以通过类对象创建一个类的对象。（当然，这有些不符合刚刚我们说的例子，我们不能拿着一个写满狗的信息的牌子，就创建一条狗吧……）

例如，如果要创建一个 Student 对象，除了可以直接 new 之外，还能够用下面的方法：

```
Class c = Student.class;
Student stu = (Student) c.newInstance();
```

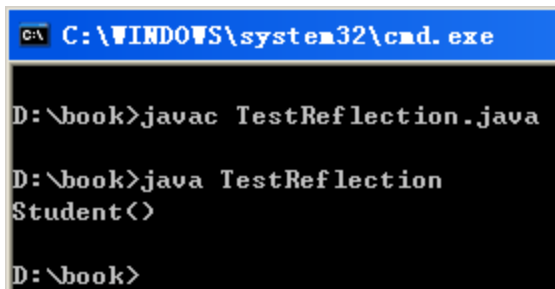
这样创建对象的时候，会调用对象的无参构造方法。为了证明这一点，我们修改 Student 类的无参构造方法如下：

```
public Student() {
    System.out.println("Student()");
}
```

然后，运行示例代码：

```
import java.lang.reflect.*;
public class TestReflection {
    public static void main(String[] args) throws Exception {
        Class c = Student.class;
        Student stu = (Student) c.newInstance();
    }
}
```

结果如下：



```
C:\WINDOWS\system32\cmd.exe

D:\book>javac TestReflection.java

D:\book>java TestReflection
Student()
```

输出 Student(), 说明无参构造方法被调用，创建了一个 Student 对象。

至此，开头的需求 2 也可以完成。示例代码如下：

```
public static Object createObject(String className){
    Object result = null;
    try{
        Class c = Class.forName(className);
        result = c.newInstance();
    }catch(Exception e){
        e.printStackTrace();
    }
    return result;
}
```

## 2 反射包

上一部分介绍了反射的一些基本内容。这一节将在上一节的基础上，进一步学习反射的使用，利用反射完成更多的事情。

接下来要学习的这几个类，都在 java.lang.reflect 这个反射包下面。

### 2.1 Field 类

Field 类封装了属性信息，一个 Field 对象封装了一个属性的信息。

#### 2.1.1 获取特定属性

首先，学习 Field 类第一部分，就是如何获得 Field 对象。在 Class 类中，有以下两个方法：



`Field getDeclaredField(String name)`

`Field getField(String name)`

顾名思义，这两个方法能够根据属性名，获得相应的 `Field` 对象。`getField` 方法可以获得本类的公开属性以及从父类继承到的公开属性，但是无法获得非公开属性；而 `getDeclaredField` 方法只能获得本类属性，但这包括本类的非公开属性。这点区别类似于 `getMethods` 方法和 `getDeclaredMethods` 方法，不是吗？

例如，如果要获得代表 `Student` 类的 `name` 属性的 `Field` 对象，则可以使用下面的代码：

```
Class c = Student.class;
Field nameField = c.getField("name");
这样，就可以获得相应的 Field 对象。
```

### 2.1.2 修改、读取属性

有了 `Field` 对象之后，我们还可以使用反射来获取、修改属性的值。

首先，我们分析一下，如果不使用反射的话，应当如何对属性的值进行读取的修改。代码如下：

```
Student stu = new Student();
stu.name = "tom"; //修改属性值
String data = stu.name; //获取属性值
```

要注意的是，修改属性值，有三个要素：

- 1、`stu`。这个要素说明的是，我们要修改哪一个对象的属性；
- 2、`.name`。这个要素说明的是，我们希望修改的是对象的哪一个属性。
- 3、“tom”。这个要素说明的是，我们希望把对象的属性值修改成什么。

分析清楚了这三个要素之后，我们就可以学习利用反射来修改属性。

首先我们必须获得即将被修改的属性所对应的 `Field` 对象，然后对 `Field` 对象调用 `set` 方法。`set` 方法签名如下（不包括抛出的异常）

```
public void set(Object obj, Object value)
```

这个方法有两个参数，第一个参数表示要修改属性的对象，第二个参数表示属性值要修改成什么。

我们可以用代码来表示：

```
Student stu = new Student();
Class c = stu.getClass();
Field nameField = c.getField("name"); //1
nameField.set(stu, //2
    "tom"); //3
```

上面的代码，同样把 `stu` 对象的 `name` 属性设为了 `tom`。我们来分析一下反射的这部分代码。

//1 的位置，是获取相应的 `Field` 对象。我们可以认为这是对应着上面所说的要素 2：要修改对象的哪一个属性。我们通过调用 `getField` 方法，说明要修改的是名字为“name”的属性。

//2 的位置，是 `set` 方法的第一个参数，这个参数对应着要素 1：要修改哪一个对象的属性。这里说的很明确，要修改 `stu` 对象的属性。

//3 的位置，是 `set` 方法的第二个参数。这个参数对应着要素 3：要把属性值修改成什么。

我们可以看到，使用反射设置属性，与直接使用代码设置属性，所需要的信息是一致的。

理解了怎么设置属性之后，再学习怎么获取属性就变得比较简单了。`Field` 类中有一个 `get` 方法，签名如下（不包括抛出的异常）

```
public Object get(Object obj)
```

`get` 方法的参数，表明了要读取哪一个对象的属性。而 `get` 方法的返回值，则表明了读取到的属性值。例如，要获得 `stu` 对象的 `name` 属性值，代码如下：

```
Class c = stu.getClass();
Field f = c.getDeclaredField("name");
Object value = f.get(stu);
```

### 2.1.3 私有属性

除了能够获得 `Student` 类中的公开属性之外，利用反射还能获得并修改对象的私有属性，如下面的代码：

```
Class c = Student.class;
```

```
Field ageField = c.getDeclaredField("age");
```

由于 `age` 属性是私有的，因此只能用 `getDeclaredField` 方法。

对于一般的途径来说，是不能直接读取、修改私有属性的。然而，反射却可以突破属性私有的限制，只需要在读取和修改之前调用一个方法：

```
public void setAccessible(boolean flag)
```

为这个方法传递一个参数 `true` 就可以解决问题。参考代码如下：

```
Student stu = new Student();
// stu.age = 18; 不能直接修改 age 属性，这句代码将无法编译通过
Field f = stu.getClass().getDeclaredField("age");
f.setAccessible(true);
f.set(stu, new Integer(18));
```

需要注意的是，虽然 `age` 属性是一个 `int` 类型，但是由于 `set` 方法第二个参数是 `Object` 类型，因此必须要把 `18` 这个整数值封装成 `Integer` 对象。

从上面的例子中我们可以看到，反射可以获取一个对象的私有属性，并且可以读取和修改私有属性。这也是我们不使用反射做不到的。

那么，这样算不算破坏封装呢？严格的说，算。但是，这种对封装的破坏并不可怕。要明确的是，反射是一种非常底层的技术，而封装相对来说是一个比较高级的概念。例如，一台服务器，要防止外部的破坏，有可能会假设一道网络防火墙。防火墙这个概念就是一个相对比较高级的概念。而这个防火墙设计的再合理，如果服务器机房的钥匙被人偷走了，让人能够进入机房偷走服务器，那么防火墙设计的再好也拦不住。防火墙防止的是高层的攻击，而底层的破坏，不需要防火墙处理。

封装也一样。封装防止的是程序员直接访问和操作一些私有的数据；而反射是一个非常底层的技术，利用反射，完全可以打破封装。

## 2.2 Method 类

`Method` 类在之前已经接触过，因此关于 `Method` 类的基本概念不再赘述。

### 2.2.1 获取特定方法

除了之前我们说的 `getMethods` 和 `getDeclaredMethods` 方法之外，还有两个方法能够获得

特定的方法对象：

```
public Method getMethod(String name, Class[] parameterTypes)
```

```
public Method getDeclaredMethod(String name, Class[] parameterTypes)
```

两个方法的区别同样与 `getMethods` 方法和 `getDeclaredMethods` 方法类似。`getMethod` 可以获得公开方法，包括父类的；`getDeclaredMethod` 只能获得本类的方法，但不限于公开方法。

我们可以看到，`getMethod` 以及 `getDeclaredMethod` 方法有两个参数。

第一个参数是一个字符串参数，表示的是方法的方法名。

但是，光有方法名还不能确定一个方法，因为类中有可能有方法重载的情况。

为了能唯一确定一个方法，除了要给一个方法名之外，还要给出这个方法的参数表。我们用一个 `Class` 数组来表示参数表。

对于无参的方法来说，参数表就是一个空数组：`new Class[]{}`

对于有一个参数的方法来说（例如一个 `int` 类型参数），则参数表是长度为 1 的数组：`new Class[]{int.class}`

对于有多个参数的方法来说，则把多个参数的类型依次罗列在数组中。例如，如果一个方法接受一个 `int` 类型与一个 `double` 类型的话，则表示参数表的 `Class` 数组为：`new Class[]{int.class, double.class}`

因此，如果想要获得两个参数的 `study` 方法，则可以用下面的代码：

```
Class c = Student.class;
Method m = c.getMethod("study",
    new Class[]{int.class, double.class});
```

### 2.2.2 利用反射，调用对象的方法

接下来，我们学习怎么用反射来调用方法。

首先，还是分析一下不用反射应当怎么来调用方法。

```
Student stu = new Student();
```

```
double result = stu.study(10, 1.5);
```

上面就是调用方法的例子。这里面有四个要素

- 1、`stu`。需要说明对哪个对象调用方法
- 2、`study`。需要说明调用的是哪个方法
- 3、`(10, 1.5)` 需要传入实参
- 4、方法可以有返回值

利用反射调用方法，首先我们必须获得即将被调用的方法所对应的 `Method` 对象，然后对 `Method` 对象调用 `invoke` 方法。`invoke` 方法签名如下（不包括抛出的异常）

```
public Object invoke(Object obj, Object[] args)
```

在这个方法中，`invoke` 方法有两个参数

- 1、第一个参数 `obj` 表示对哪一个对象调用方法
- 2、第二个参数表示调用方法时的参数表
- 3、`invoke` 方法的返回值对应于 `Method` 对象所代表的方法的返回值。

我们给出利用反射调用 `study` 方法的代码：

```
Student stu = new Student();
```

```

Class c = stu.getClass();
Method m = c.getDeclaredMethod("study", new Class[]{int.class,
double.class}); //1
Object result //2
    = m.invoke(stu, //3
        new Object[]{new Integer(10), new Double(1.5) } ); //4

```

我们来分析上述代码。

//1 的位置获得一个代表带两个参数的 study 方法的 Method 对象，表示调用哪个方法，对应于要素 2；

//2 是 invoke 方法的返回值，对应于要素 4；

//3 是 invoke 方法的第一个参数，表明对 stu 对象调用方法，对应于要素 1；

//4 的位置传入了两个参数，这两个参数形成调用方法时的实参，对应于要素 3。

上面就是利用反射调用方法。与 Field 对象类似，也可以调用一个类中的私有方法，只需要在调用 Method 对象的 invoke 方法之前，先调用 setAccessible(true)即可。

## 2.3 Constructor 类

我们简单介绍一下 Constructor 类，故名思意，这个类封装了构造函数的信息，一个 Constructor 对象代表了一个构造函数。

首先，可以通过 Class 类中的 getConstructors() / getDeclaredConstructors() 获得 Constructor 数组。

其次，可以通过 Class 类中的 getConstructor() / getDeclaredConstructor() 来获得指定的构造方法。与 getMethod 不同，这两个方法只有一个参数：一个 Class 数组。原因也很简单：构造方法的方法名与类名相同，不需要指定。

最后，可以调用 Constructor 类中的 newInstance 方法创建对象。创建对象的时候，会调用相应的构造方法。

如果创建对象只需要调用无参构造方法的话，就可以直接使用 Class 类中的 newInstance 方法，如果在创建对象的时候需要指定调用其他构造方法的话，就需要使用 Constructor 类。

下面的代码利用这两种不同的方式创建对象。

```

import java.lang.reflect.*;
class Dog{
    String name;
    int age;

    public Dog(){
        System.out.println("Dog ()");
    }
    public Dog(String name, int age) {
        System.out.println("Dog (String, int)");
        this.name = name;
        this.age = age;
    }
    public String toString(){

```

```

        return name + " " + age;
    }
}

public class TestConstructor {
    public static void main(String[] args) throws Exception {
        Class c = Dog.class;

        Dog d1 = (Dog) c.newInstance();
        System.out.println(d1);
        //获得构造方法
        Constructor con = c.getConstructor(
            new Class[]{String.class, int.class});
        //创建对象时指定构造方法
        Dog d2 = (Dog) con.newInstance(
            //为构造方法传递的参数
            new Object[]{"Snoopy", new Integer(5)});
        System.out.println(d2);
    }
}

```

很明显，在上面的代码中 d1 对象是利用 Dog 类的无参构造方法创建出来的；而 d2 对象则是利用有参构造方法创建出来的，在创建的同时，name 属性被赋值为“snoopy”，age 属性被赋值为 5。

### 3 反射的作用

学到这里，你可能会感到疑惑。反射有什么用呢？我们在前面的学习中已经掌握了创建对象，调用方法的办法，利用反射做这些事情的优势在哪里呢？

我们来对比以下的代码：

```

Student s = new Student();
s.study();

```

这是最常规的创建 Student 对象，并调用 study() 方法的代码。

```

String className = "Student";
Class c = Class.forName(className);
Object o = c.newInstance();

String methodName = "study";
Method m = c.getMethod(methodName, new Class[]{});
m.invoke(o, new Object[]{});

```

这是利用反射的代码，做的是同样的事情。

很显然用反射写出的代码比较繁琐，可是除了繁琐呢？你看出别的端倪了么？

在反射代码中,创建对象所采用的类名“Student”,以及调用方法时的方法名“study”都是以字符串的形式存在的,而字符串的值完全可以不写在程序中,比如,从文本文件中读取。这样,如果需求改变了,需要创建的对象不再是 Student 对象,需要调用的方法也不再是 study 方法,那么程序有没有可能不做任何修改呢?当然可能,你需要修改的可能是那个文本文件。

反观不用反射的代码,它只能创建 Student 对象,只能调用 study 方法,如有改动则必须修改代码重新编译。明白了吧,用反射的代码,会更通用,更万能!

因此,利用反射实现的代码,可以在最大程度上实现代码的通用性,而这正是工具和框架在编写的时候所需要的。因此,反射才能在这些领域得到用武之地。

我们在前面的章节中提到过,利用多态,可以使代码通用,例如:

```
public void feed (Dog d ){
    d.eat();
}
public void feed (Animal a ){
    a.eat();
}
```

很显然,以 Animal 为参数的 feed 方法要比以 Dog 为参数的 feed 方法具有更大的通用性,因为它不仅可以传入 Dog 对象,还可以传入其他的 Animal 的子类对象。但是按照这个思路,能不能再通用一点呢?请看以下代码:

```
public void feed (Object o) throws Exception{
    Class c = o.getClass();
    Method m = c.getMethod("eat",new Class[]{});
    m.invoke(o , new Object[]{});
}
```

这个 feed 方法的参数类型为 Object,可以传入任何对象。只要这个对象具有 eat 方法,就可以通过反射来实现对 eat 方法的调用。好了,利用反射,我们已经把多态用到极致了。因为 Object 类不会再有父类了。

当然,这里并不是鼓励大家滥用反射。反射技术有着非常显著的几个缺点。

1. 运行效率与不用反射的代码相比会有明显下降。
2. 代码的复杂度大幅提升,这个从代码量上大家就能比较出来
3. 代码会变得脆弱,不易调试。使用反射,我们就在一定程度上绕开了编译器的语法检查,例如,用反射去调用一个对象的方法,而该对象没有这个方法,那么在编译时,编译器是无法发现的,只能到运行时由 JVM 抛出异常。

因此,反射作为一种底层技术,只适合于工具软件和框架程序的开发,在大部分不需要使用反射的场合,没有必要为了追求程序的通用性而随意使用反射。滥用反射绝对是一个坏的编程习惯。