第八章：反射与代理机制

8.1 Java 反射机制

1.Java 类型信息

（1）获取 Java 运行时的类型信息有两种方法：

①RTTI。在运行时，需要识别一个对象的类型。当从数组中取出元素时，会自动将结果转型回 Shape，这是 RTTI 最基本的使用形式，因为在 Java 中，所有的类型转换都是在运行时进行正确性检查的。

在写代码时，大部分代码应尽可能少地了解对象的具体类型，而是只与对象家族中的一个通用表示打交道。

②Java 反射机制。它指在运行状态中，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方法和属性；这种动态获取信息以及动态调用对象的方法称为 Java 的反射机制。

2.类 Class

Class 类是 Java 一个基础类，每装载一个新类，java 虚拟机就会在 java 堆中，创建一个 Class 实例，这个实例就代表 Class 这个类型，通过实例获取类型信息。该类中一些方法如下：

（1）Method[] getMethods()：获得 Class 类中所有的方法，这个是方法类的数组。

（2）Field[] getFields()：获得 Class 类中所有的成员变量，这是一个 Field 类的数组。

（3）Constructor<?>[] getDeclaredConstructors()：获得 Class 类中所有的构造方法。

例：利用 Class 类创建一个实例

1.创建 Class 类的一个对象，返回一个类的引用

```
Class cls = Class.forName("Airplane");//返回一个类型
```

forName 是 Class 的一个静态方法，实际上 Airplane 是一个类的名字，cls 实际上是 Class 的一个实例，这个 cls 实际上是 Airplane 这个类型。

2.通过类的引用创建实例

```
cls.newInstace();//通过 newinstace 创建实例，一般调用默认构造函数
```

这样就可以用一个字符串“Airplane”创建出一个类的实例。

完整版：

```
Class Airplane{
    public String toString(){
        return("in airplane");
    }
}
public class CreateInstance{
    public static void main(String[] args)throws Exception{
        Class c1=null;
        Object ap;
        c1=Class.forName("Airplane");//创建 Class 类的一个对象
        System.out.println(c1);
```

```
            ap=c1.newInstance();//创建实例的另外一种方法
            System.out.println(ap.toString());
        }
}
```

传统创建实例的方法：

```
Airplane ap = new Airplane();
```

Java 反射的例子——Method 类的 invoke

如图 1 所示，首先要引出 reflect 这个包。然后定义 add 方法，add 方法的功能是把两个参数转换成整数值，然后再把它们相加。再定义第二个方法 StringAdd 方法，输出“out”和 abc 的值。

接下来看主方法。第一个定义了一个 Method 类的实例 mth，这句话代表获得了 ClassA 中 add 的这个方法，逗号后面的语句为提供的参数。接下来通过 invoke 方法来执行 add 方法。后面括号里是指一样的，前面生成了一个 Class 类的对象，后面就要用 newInstance 方法创建实例，然后两个参数里面的数字分别是 1 和 2。后面获取 StringAdd 与前面同理。

图 1

8.2 Java 静态代理

1.代理模式

在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介作用。

代理模式的作用是，为其他对象提供一种代理以控制这个对象的访问。如图 2 所示，Client 要调用我们目标对象的 request 方法。左下角的代理和右下角的真实对象拥有同样的接口，只不过它可能还要做一些其它的事情。

图 2

2.代理模式一般涉及到的角色

（1）抽象角色：声明真是对象和代理对象共同的接口。

（2）代理角色：代理对象角色内部含有对真是对象的引用，从而可以操作真是对象，同事代理对象提供与真是对象相同的接口以便在任何时刻能够代替真实对象。同事，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真是对象进行封装。

（3）真实角色：代理角色所代表的的真实对象，是我们最终要引用的对象。

例：静态代理的例子

```
//真实对象和代理对象的共同接口
abstract class Subject{
    public abstract void request();
}
//真实对象和代理对象的共同接口
class RealSubject extends Subject{
    public void request(){
        System.out.println("From Real Subject!");
    }
}
```

```
//客户端
public class Client{
    public static void main(String[] args){
        Subject subject = new ProxySubject();
        subject.request();
    }
}
//代理角色
class ProxySubject extends Subject{
    //代理角色对象内部含有对真是对象的引用
    private RealSubject realSubject;
    @Overridepublic void request(){
        //在真实角色操作之前所附加的操作
        preRequest();
        if(null == realSubject){
            realSubject=new RealSubject();
        }//真实角色所完成的事情
        realSubject.request();
        //在真是角色操作之后附加操作
        postRequest();
    }
    private void preRequest(){
        System.out.println("Pre");
    }
    private void postRequest(){
        System.out.println("post");
    }
```

这里就看出来了，代理角色可以实现真实角色的方法，也可以自己附加执行自己的一些操作。

3.静态代理的优缺点

优点：

业务类只需要关注业务逻辑本身，保证了业务类的可重用性，这是代理的共有优点。

缺点：

（1）代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理，静态代理在程序规模稍大时就无法胜任。

（2）如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法，增加了代码维护的复杂度。

8.3 Java 动态代理

1.Java 动态代理

使用到的类：

（1）java.lang.reflect.Proxy

这是 java 动态代理机制的主类，它提供了一组静态方法来为一组接口动态地生成代理

类及其对象。常用方法有以下几种：

①用于获取指定代理对象所关联的调用处理器：

static InvocationHandler getInvocationHandler(Object proxy)

②获取关联于指定类装载器和一组接口的动态代理类的类对象

static Class getProxyClass(ClassLoader loader, Class[] interfaces)

③判断指定类对象是否是一个动态代理类

static Boolean isProxyClass(Class cl)

④为指定类装载器、一组接口及调用处理器生成动态代理类实例

static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)

（2）java.lang.reflect.InvocationHandler

这是调用处理器接口，它自定义了一个 Invoke 方法，用于集中处理在动态代理类对象上的方法嗲用，通常在该方法中实现对委托类的代理访问。

Object invoke(Object proxy, Method method, Object[] args)

该方法负责集中处理动态代理类上的所有方法调用。第一个参数是代理类实例，第二个参数是被调用的方法对象，第三个方法是调用参数。调用处理器根据这三个参数进行预处理或分配到委托类实例上执行。

例：Java 动态代理实例

```
//抽象角色
interface Subject{
    public void request();
}
//真实角色，实现了 Subject 的 request()方法
class RealSubject implements Subject{
    public RealSubject(){}
    public void request(){
        System.out.println("From real subject");
    }
}
//代理角色，必须继承 InvocationHandler
import java.lang.reflect.Method;
import java.lang.reflect.InvocationHandler;
class DynamicSubject implements InvocationHandler{
    private Object sub;
    public DynamicSubject(){}
    public DynamicSubject(Object obj){
        sub = obj;
    }
    public Object invoke(Object proxy, Method method,
  Object[] args)throws Throwable{
        System.out.println("before calling"+method);
        method.invoke(sub,args);
        System.out.println("after calling" + method);
        return null;
    }
```

```
}
//客户端调用
import java.lang.reflect.Proxy;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationHandler;
public class Client{
    public static void main(String[] args)throws Throwable{
        RealSubject rs = new RealSubject();//在这里指定被代理类
        InvocationHandler ds = new DynamicSubject(rs);
        Class cls = rs.getClass();//以下是一次性生成代理
        Subject subject = (Subject)Proxy.newProxyInstance
        (cls.getClassLoader(),cls.getInterfaces(),ds);
        subject.request();
    }
}
```

运行结果：

```
before calling public abstract void Subject.request()
From realsubject
before calling public abstract void Subject.request()
```

2.动态代理的特点

（1）包：如果所代理的接口都是 public 的，那么它将被定义在顶层包（即包路径为空），如果所代理的接口有非 public 的接口，那么它就被定义在该接口所在的包，这样设计的目的是为了最大程度的保证动态代理类不会因为包管理的问题而无法被成功定义并访问；

（2）类修饰符：该代理类具有 final 和 public 修饰符，意味着它可以被所有的类访问，但是不能被再度继承；

（3）类名：格式是“$ProxyN”，其中 N 是一个逐一递增的阿拉伯数字，代表 Proxy 类第 N 此生成的动态代理类。注意，并不是每次调用 Proxy 的惊天方法创建动态代理都会使 N 增加，原因是对同一组接口（包括接口排序的顺序相同）试图重复创建动态代理类，它会很聪明地返回先前已经创建好的代理类的类对象，而不会尝试去创建一个全新的代理类，这样可以节省不必要重复的代码生成，提高代理类的创建效率。

（4）类继承关系：该类的继承关系如图 3 所示。

图 3

3.动态代理的优点和缺点

优点：接口中声明的所有方法都被转移到调用处理器一个集中的方法中进行处理（InvocationHandler.invoke）。这样，在借口方法数量较多的时候，可以进行灵活处理。

缺点：始终无法摆脱仅支持 interface 代理的桎梏。

8.4 Java 反射扩展-JVM 加载类原理

1.JVM 类加载的种类

（1）JVM 自带的默认加载器：

①跟类加载器：bootstrap，由 C++编写，所有 Java 程序无法获得。

②扩展类加载器：Java 编写。

③系统类、应用类加载器：Java 编写。

（2）用户自定义加载器

Java.lang.ClassLoader 的子类，用户可以定制类的加载方式。每一个类都包含了加载它的 ClassLoader 的一个引用——getClassLoader()。

2.类的加载方式

（1）本地编译好的 class 中直接加载。

（2）网络加载：java.net.URLClassLoader 可以加载 url 指定的类。

（3）从 jar、zip 等压缩文件加载类，自动解析 jar 文件找到 class 文件去加载。

（4）从 java 源代码文件动态编译成 class 文件。

3.类加载的步骤

（1）加载。把 class 文件放到虚拟机。

（2）连接。包括以下小步骤：验证，验证 class 文件是否满足 Java 虚拟机对于自解码文件的规范；准备，把静态成员做内存分配；解析，把符号性引用转换成直接引用。

（3）类的初始化。

4.类加载器的加载顺序

根加载器->扩展类加载器->应用类加载器->用户自定义类加载器

注意：如果到最后一层都加载不了，就出现 ClassNotFoundException 异常。

5.类加载器加载 Class 的过程

第一步：检测此 Class 是否载入过，如果有跳到第 8 步，否则前进到第 2 步。

第二步：如果 parent classloader 不存在（没有 parent，那 parent 一定是 bootstrap classloader），则跳到第 4 步。

第三步：请求 parent classloader 载入，如果成功到底 8 步，否则第 5 步。

第四步：请求 JVM 从 bootstrap classloader 中载入，若成功到第 8 步。

第五步：寻找 Class 文件（从榆次 classloader 相关的路径查找），如果找不到则到第 7 步。

第六步：从文件载入 Class，到第 8 步。

第七步：抛出 ClassNotFoundException。

第八步，返回 Class。