**Java反射机制及IoC原理**

**一. 反射机制概念**

　　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。在java中，只要给定类的名字， 那么就可以通过反射机制来获得类的所有信息。

　　反射是Java中一种强大的工具，能够使我们很方便的创建灵活的代码，这些代码可以再运行时装配，无需在组件之间进行源代码链接。但是反射使用不当会成本很高！

　　类中有什么信息，利用反射机制就能可以获得什么信息，不过前提是得知道类的名字。

**二. 反射机制的作用**

1.在运行时判断任意一个对象所属的类；

2.在运行时获取类的对象；

3.在运行时访问java对象的属性，方法，构造方法等。

**三. 反射机制的优点与缺点**

首先要搞清楚为什么要用反射机制？直接创建对象不就可以了吗，这就涉及到了动态与静态的概念。

静态编译：在编译时确定类型，绑定对象，即通过。

动态编译：运行时确定类型，绑定对象。动态编译最大限度发挥了java的灵活性，体现了多态的应用，有以降低类之间的藕合性。

**反射机制的优点**：可以实现动态创建对象和编译，体现出很大的灵活性（特别是在J2EE的开发中它的灵活性就表现的十分明显）。通过反射机制我们可以获得类的各种内容，进行了反编译。对于JAVA这种先编译再运行的语言来说，反射机制可以使代码更加灵活，更加容易实现面向对象。

　　比如，一个大型的软件，不可能一次就把把它设计的很完美，当这个程序编译后，发布了，当发现需要更新某些功能时，我们不可能要用户把以前的卸载，再重新安装新的版本，假如这样的话，这个软件肯定是没有多少人用的。采用静态的话，需要把整个程序重新编译一次才可以实现功能的更新，而采用反射机制的话，它就可以不用卸载，只需要在运行时才动态的创建和编译，就可以实现该功能。

**反射机制的缺点**：对性能有影响。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。这类操作总是慢于只直接执行相同的操作。

四. 反射机制的示例

1.通过一个对象获得完整的包名和类名

(添加一句：所有类的对象其实都是Class的实例。)

1. 实例化Class类对象
2. 通过Class实例化其他类的对象

编写使用Class实例化其他类的对象的时候，一定要自己定义无参的构造函数。

4.通过Class调用其他类中的构造函数 （也可以通过这种方式通过Class创建其他类的对象）

5.返回一个类实现的接口

6.取得其他类中的父类

7.获得其他类中的全部构造函数

8.取得其他类的全部属性，将这些整理在一起，也就是通过class取得一个类的全部框架

9.通过反射调用其他类中的方法

10.调用其他类的set和get方法

11.通过反射操作属性

12.通过反射取得并修改数组的信息

13.通过反射修改数组大小

14.动态代理

其实在java中有三种类类加载器。

1）Bootstrap ClassLoader 此加载器采用c++编写，一般开发中很少见。

2）Extension ClassLoader 用来进行扩展类的加载，一般对应的是jre\lib\ext目录中的类

3）AppClassLoader 加载classpath指定的类，是最常用的加载器。同时也是java中默认的加载器。

如果想要完成动态代理，首先需要定义一个InvocationHandler接口的子类，已完成代理的具体操作。

**类的生命周期**

　　在一个类编译完成之后，下一步就需要开始使用类，如果要使用一个类，肯定离不开JVM。在程序执行中JVM通过装载，链接，初始化这3个步骤完成。

　　类的装载是通过类加载器完成的，加载器将.class文件的二进制文件装入JVM的方法区，并且在堆区创建描述这个类的java.lang.Class对象。用来封装数据。 但是同一个类只会被类装载器装载以前

链接就是把二进制数据组装为可以运行的状态。

链接分为校验，准备，解析这3个阶段：

校验一般用来确认此二进制文件是否适合当前的JVM（版本），

准备就是为静态成员分配内存空间，。并设置默认值

解析指的是转换常量池中的代码作为直接引用的过程，直到所有的符号引用都可以被运行程序使用（建立完整的对应关系）。

　　完成之后，类型也就完成了初始化，初始化之后类的对象就可以正常使用了，直到一个对象不再使用之后，将被垃圾回收。释放空间。当没有任何引用指向Class对象时就会被卸载，结束类的生命周期。

**五. IoC原理**

Spring中的IoC的实现原理就是工厂模式加反射机制。

**1.我们首先看一下不用反射机制时的工厂模式：**

//工厂模式

interface fruit{

public abstract void eat();

}

class Apple implements fruit{

public void eat(){

System.out.println("Apple");

}

}

class Orange implements fruit{

public void eat(){

System.out.println("Orange");

}

}

// 构造工厂类

// 也就是说以后如果我们在添加其他的实例的时候只需要修改工厂类就行了

class Factory{

public static **fruit** getInstance(String fruitName){

fruit f=null;

if("Apple".equals(fruitName)){

f=new Apple();

}

if("Orange".equals(fruitName)){

f=new Orange();

}

return f;

}

}

class hello{

public static void main(String[] a){

**fruit f=Factory.getInstance("Orange");**

f.eat();

}

}

当我们在添加一个子类的时候，就需要修改工厂类了。如果我们添加太多的子类的时候，改的就会很多。

**2. 利用反射机制的工厂模式：**

package Reflect;

interface fruit{

public abstract void eat();

}

class Apple implements fruit{

public void eat(){

System.out.println("Apple");

}

}

class Orange implements fruit{

public void eat(){

System.out.println("Orange");

}

}

class Factory{

public static **fruit** getInstance(String ClassName){

fruit f=null;

try{

f=**(fruit)Class.forName(ClassName).newInstance();**

}catch (Exception e) {

e.printStackTrace();

}

return f;

}

}

class hello{

public static void main(String[] a){

fruit f=**Factory.getInstance("Reflect.Apple")**;

if(f!=null){

f.eat();

}

}

}

现在就算我们添加任意多个子类的时候，工厂类就不需要修改。

　　使用反射机制的工厂模式可以通过反射取得接口的实例，但是需要传入完整的包和类名。而且用户也无法知道一个接口有多少个可以使用的子类，所以我们**通过属性文件的形式配置所需要的子类**。

**3.使用反射机制并结合属性文件的工厂模式（即IoC）**

首先创建一个fruit.properties的资源文件：

1 apple=Reflect.Apple

2 orange=Reflect.Orange

然后编写主类代码：

package Reflect;

import java.io.\*;

import java.util.\*;

interface fruit{

public abstract void eat();

}

class Apple implements fruit{

public void eat(){

System.out.println("Apple");

}

}

class Orange implements fruit{

public void eat(){

System.out.println("Orange");

}

}

//操作属性文件类

class init{

public static Properties getPro() throws FileNotFoundException, IOException{

Properties pro=new Properties();

File f=new File("fruit.properties");

if(f.exists()){

pro.load(new FileInputStream(f));

}else{

pro.setProperty("apple", "Reflect.Apple");

pro.setProperty("orange", "Reflect.Orange");

pro.store(new FileOutputStream(f), "FRUIT CLASS");

}

return pro;

}

}

class Factory{

public static fruit getInstance(String ClassName){

fruit f=null;

try{

f=**(fruit)Class.forName(ClassName).newInstance();**

}catch (Exception e) {

e.printStackTrace();

}

return f;

}

}

class hello{

public static void main(String[] a) throws FileNotFoundException, IOException{

Properties pro=init.getPro();

fruit f=**Factory.getInstance(pro.getProperty("apple"))**;

if(f!=null){

f.eat();

}

}

}

//【运行结果】：Apple