# 框架基础

# 反射

Java反射机制是在运行状态中，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，都能调用它的任意一个方法和属性，这种动态获取的信息以及调用对象的方法的功能成为java语言的反射机制.

# Class类

Class这个类用于操纵一个类的属性，方法，构造器。

# 获得Class对象的三种方式

1,方法1：通过Class.forName(“类的全路径”);

2,方法2：通过类名打点class,如：person.class;

3,方法3：通过调用该类的对象调用getClass()方法；

|  |
| --- |
| 1, Class<?> clazz = Class.*forName*("java.util.HashMap");  2,Class<?> clazz= obj.getClass();  3,Class<?> clazz= Person.**class**; |

# Class对象的属性，方法，构造器的获取

1，获得类的所有属性

Field[] fields= clazz.getDeclaredFields();

获得类的单个属性

Field field = class1.getDeclaredField("name");

2,获得类的所有方法

Method[] Methods = clazz.getDeclaredMethods();

获得单个方法

没参数的方法

Method method = class1.getDeclaredMethod("getName",**new** Class[]{});

有参数的方法

Method method1 = class1.getDeclaredMethod("setName",new Class[]{String.**class**});

3,获得类的所有构造器

Constructor<?>[]constructors=clazz.getDeclaredConstructors()

获取单个构造器

没带参数的构造器（默认）

Constructor<?> constructor = class1.getDeclaredConstructor(**new** Class[]{});

带参数的构造器

Constructor<?> constructor2=class1.getDeclaredConstructor(**new**Class[]{String.**class**,Integer.**class**,String.**class**});

# 反射赋值

1，根据构造器创建对象

newInstance(object..)

默认构造器获得对象

Object instance = constructor.newInstance(**new** object[]{});

带参数的构造器获得对象(参数要一构造器中一致)

Object instance2 = constructor1.newInstance(**new** Object[]{"张三",1,"广州"});

|  |
| --- |
| Class<?> class1 = Class.*forName*("com.mj.reflect.Person");  Constructor<?> constructor = class1.getDeclaredConstructor(**new** Class[]{});  //根据类的默认构造器来获得对象  Object instance = constructor.newInstance(**new** Object[]{});  System.*out*.println(instance);  //获得类的所有方法  Method[] methods = class1.getDeclaredMethods();  **for**(Method m:methods){  //获得方法的名字  String mname=m.getName();  //获得是否以set开头  **boolean** startsWith = mname.startsWith("set");  **if**(startsWith){  //获得到set字符串后面的值  String fieldname=mname.substring(3);  //截取第一个字符把它变成小写，加上原有字符除这个大写外的其他字符,获得属性的名字  fieldname=fieldname.substring(0,1).toLowerCase()+fieldname.substring(1); //获得方法所对应的属性 Field field = class1.getDeclaredField(fieldname);  //获得属性的具体类型  Class<?> type = field.getType();  **if**(type==Integer.**class**){  //反射调用赋值  m.invoke(instance, **new** Object[]{1});  }  //判断类型是否相等  **if**(type==String.**class**&&"address".equals(fieldname)){    m.invoke(instance, **new** Object[]{"广州"});  }  **if**(type==String.**class**&&"name".equals(fieldname)){    m.invoke(instance, **new** Object[]{"李四"});  }    }  } |

# getDeclaredMethod(获取方法)

## Invoke设置属性和获取返回值的

Method method = class1.getDeclaredMethod("setName", **new** Class[]{String.**class**});

Object obj= method.invoke(instance, **new** Object[]{"张三"});

Method mget=class1.getDeclaredMethod("getName", **new** Class[]{});

Object invoke = mget.invoke(instance, **new** Object[]{});

静态方法

Method method2 = class1.getDeclaredMethod("print", **null**);

Object invoke2 = method2.invoke(**null**, **null**);

# 破坏私有属性或方法的权限

field.setAccessible(**true**);

# 通过反射拷贝对象

|  |
| --- |
| **public** **static** Object copyobj(Object obj) **throws** Exception {  // 获得传递过来的对象的属性和构造器  Class<? **extends** Object> class1 = obj.getClass();  Field[] fields = class1.getDeclaredFields();  Constructor<? **extends** Object> constructor = class1  .getDeclaredConstructor(**new** Class[] {});  //创建对象  Object instance = constructor.newInstance(**new** Object[]{});  **for**(Field f:fields){  //获得属性的name  String fname = f.getName();  //获得属性的类型  Class<?> type = f.getType();  //获取的属性对应的set方法名字  String setMethodName= "set"+fname.substring(0,1).toUpperCase()+fname.substring(1);  String getMethodName= "get"+fname.substring(0,1).toUpperCase()+fname.substring(1);  //获得get方法  Method gmethod = class1.getDeclaredMethod(getMethodName, **null**);  //调用get方法获得被拷贝对象的一个属性值  Object gresult = gmethod.invoke(obj, **null**);  //  Method smethod = class1.getDeclaredMethod(setMethodName, **new** Class[]{gmethod.getReturnType()});  smethod.invoke(instance, **new** Object[]{gresult});    }  **return** instance;  } |

# 代理模式

代理模式的作用是：为其他对象提供一种以控制对方的访问。

在某种情况下，一个客户不想或者不能直接引用另一个对象可以在客户端和目标对象直接起到中介的作用。

代理的角色：

第一种：抽象角色，声明真实对象和代理对象的共同接口。

第二种：代理角色，代理对象角色内部含有对真实对象的引用，从而可以操纵真实的对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象，同时代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

第三种：真实角色，代理所代理的真实角色，是我们要引用的对象

# 静态代理

## 抽象角色

|  |
| --- |
| **public** **interface** HireHose {  **public** **void** hire();  } |

## 具体角色

|  |
| --- |
| **public** **class** HireHouseImpl **implements** HireHose {    **public** **void** hire() {  System.*out*.println("租房");  } |

## 代理角色

|  |
| --- |
| //真实角色  **private** HireHose hh;    **public** HireHouseProxy(HireHose hh) {  **super**();  **this**.hh = hh;  }  **public** **void** hire() {  System.*out*.println("收费");    hh.hire();  System.*out*.println("扣押金");    } |

## 测试

|  |
| --- |
| HireHose hh=**new** HireHouseProxy(**new** HireHouseImpl());  hh.hire(); |

# jdk动态代理可以为所有带有接口实现类提供代理。

# 动态代理开发步骤

# 1.创建一个实现接口InvocationHandler的类，它必须实现invoke方法

# 2.创建被代理的类和接口

# 3.通过Proxy.newProxyInstance来创建代理类

# 4.通过代理调用

## 接口

|  |
| --- |
| **public** **interface** HireHose {    **public** **void** hire();  } |

## 接口实现类

|  |
| --- |
| **public** **class** HireHouseImpl **implements** HireHose {    **public** **void** hire() {  System.*out*.println("租房");  } |

## 接口实现类

|  |
| --- |
| **public** **class** HireProxy **implements** InvocationHandler{  //被代理的真实角色  **private** Object obj;  **public** HireProxy(Object obj) {  **super**();  **this**.obj = obj;  }  //第二个参数method是被代理对象的接口方法  //第三个参数method是被代理对象的接口方法参数  //jdk的动态代理，被代理的对象必须实现接口  @Override  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.*out*.println("前置工作");  **for**(Object obj1:args){  System.*out*.println(obj1);  }  Object invoke = method.invoke(obj, args);  System.*out*.println("后置工作");  **return** invoke;  } |

## 调用

|  |
| --- |
| //创建被代理的接口的实现类对象  HireHouseImpl hh=new HireHouseImpl();  //创建代理对象,第一个参数是获得被代理对象的类加载器，第二个参数获得被代理对象的类的所有接口，第三个参数是自定义代理类对象    HireHose hhs = (HireHose) Proxy.*newProxyInstance*(hh.getClass().getClassLoader(), hh.getClass().getInterfaces(), **new** HireProxy(hh));  hhs.hire(); |

## 步奏

1,创建具体接口

2，创建接口的实现类（具体的角色）

3,创建一个实现接口InvocationHandler的类，

3-1,创建object对象，提供构造器

3-2，实现接口中的方法，

3-3，实现invoke方法，

4，调用

4-1,创建被代理角色;

4-2，Proxy.*newProxyInstance(*被代理对象.getClass().getClassLoader(),被代理对象.getClass().getInterfaces(),new 实现InvocationHandler的类(传入被代理对象)*);*

*4-3,*使用接口来接收返回值

*4-4,*调用接口方法