# 反射机制

## 简介

**反射提供了更丰富的类设计方式。**

获得类的类型信息Class<?>实例对象，通过该对象来使用类。

获得Class<?>实例对象的三种方式：

1.类名.class;

2.object.getClass();

3.Class.forName(“类的完整路径名”);该方法使用它类无需import语句，但是可能发生ClassNotFoundException。

## 反射的应用

### 反射与工厂设计模式

通过反射和泛型的配合，可以设计不局限于接口类型的工厂设计模式。

|  |
| --- |
| **package** com.pcq.react;  **import** java.lang.reflect.InvocationTargetException;  */\*\*  \* 反射和工厂设计模式  \* 工厂设计模式应不限于具体的类，甚至于具体的接口类型。  \* 前者需要反射，后者需要泛型配合  \*/* **public class** ReactAndFactory {  **public static void** main(String[] args) {  IMessageService msgService = Factory.*getInstance*(**"com.pcq.react.NetMessage"**, IMessageService.**class**);  msgService.send(**"net message"**);  ILifeService lifeService = Factory.*getInstance*(**"com.pcq.react.HomeService"**, ILifeService.**class**);  lifeService.apply(**"home live"**);  } }  **class** Factory {  **private** Factory() {}   */\*\*  \*  \** ***@param className*** *实例化对象的类路径  \** ***@param clazz*** *实例化对象的接口类型  \** ***@param <T>*** *\** ***@return*** *\*/* **public static** <T> T getInstance(String className, Class<T> clazz) {  T instance = **null**;  **try** {  instance = (T)Class.*forName*(className).getDeclaredConstructor().newInstance();  } **catch** (Exception e) {  e.printStackTrace();  }  **return** instance;  } }  */\*\*  \* 消息服务接口  \*/* **interface** IMessageService {  **void** send(String msg); }  */\*\*  \* 生活服务接口  \*/* **interface** ILifeService {  **void** apply(String service); }  **class** NetMessage **implements** IMessageService {   @Override  **public void** send(String msg) {  System.***out***.println(**"send："** + msg);  } }  **class** HomeService **implements** ILifeService {   @Override  **public void** apply(String service) {  System.***out***.println(**"support："** + service);  } } |

### 反射与单例设计模式（懒汉式）

在多线程情况下，懒汉式并不安全。此时需要通过同步类型对象来解决单例问题。

|  |
| --- |
| **package** com.pcq.react; */\*\*  \* 反射与单例模式（懒汉式）  \*/* **public class** ReactAndSingleton {  **public static void** main(String[] args) {  **for**(**int** i = 0; i < 5; i++) {  **new** Thread(  ()->{  Singleton.*getInstance*();  }  ).start();  }  } }  */\*\*  \* 标准的多线程下饿汉式单例模式  \*/* **class** Singleton {  **private static volatile** Singleton *instance* = **null**;*//与主内存中保持一致* **private** Singleton() {  System.***out***.println(Thread.*currentThread*().getName() + **": exe getInstance"**);  }  **public static** Singleton getInstance() {  **if**(*instance* == **null**) {  **synchronized** (Singleton.**class**) {  **if**(*instance* == **null**) {*//同步逻辑中需要再次判断  instance* = **new** Singleton();  }  }  }  **return** *instance*;  } } |

## 反射与类的操作

如果拿到一个类的Class<?>对象，那可以通过该对象操作这个类。

### 反射获取类结构信息

|  |
| --- |
| Class<?> clazz = ReactClassInfo.**class**; String packageName = clazz.getPackage().getName();//获取包 System.***out***.println(packageName); Class<?> cls = clazz.getSuperclass();//获取父类 System.***out***.println(cls.getName()); Class<?>[] clss = clazz.getInterfaces();//获取实现的接口 **for**(Class<?> temp : clss) {  System.***out***.println(temp.getName());  } |

### 反射获取类的构造方法

|  |
| --- |
| Constructor<?>[] constructors = clazz.getConstructors(); **for**(Constructor<?> cons : constructors) {  System.***out***.println(cons); } Constructor con = clazz.getDeclaredConstructor(String.**class**);//能获取到私有的 System.***out***.println(con); Object obj = con.newInstance(**"abc"**); System.***out***.println(obj.toString()); |

### 反射获取类的普通方法

通过反射能拿到类的方法所有信息，包括方法的修饰符，返回类型，参数，抛出的异常

|  |
| --- |
| Class<?> clazz = ReactClassInfo.**class**; Method[] methods = clazz.getMethods(); **for**(Method m : methods) {*//遍历循环类的方法对象* System.***out***.print(Modifier.*toString*(m.getModifiers()) + **" "**);  System.***out***.print(m.getReturnType() + **" "**);  System.***out***.print(m.getName());  System.***out***.print(**"("**);  Class<?>[] params = m.getParameterTypes();  **for**(**int** i = 0; i < params.**length**; i++) {  System.***out***.print(params[i].getName() + **" "** + **"arg-"** + i);  **if**(i < params.**length** - 1) {  System.***out***.print(**","**);  }  }  System.***out***.print(**")"**);  Class<?>[] exceptions = m.getExceptionTypes();  **if**(exceptions.**length** > 0) {  System.***out***.print(**" throws "**);  }  **for**(**int** j = 0; j < exceptions.**length**; j++) {  System.***out***.print( exceptions[j].getName());  **if**(j < exceptions.**length** - 1) {  System.***out***.print(**","**);  }   }  System.***out***.println(); } |

### 反射获取成员信息

|  |
| --- |
| Class<?> clazz = ReactClassInfo.**class**; Field[] fields = clazz.getFields(); **for**(Field field : fields) {  System.***out***.println(field.getName()); } |

字段对象有两个常用的重要方法：

field.setAccessible(**true**);*//对私有的属性解除封装*

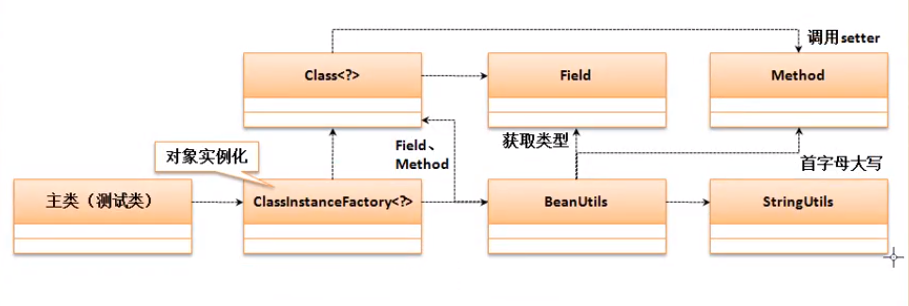
getType();//获取字段类型

## Unsafe类

除了JVM提供了反射处理之外，Java还提供了底层C++代码代替JVM执行，即绕过了JVM对对象的管理机制，包括内存管理和垃圾回收处理。

|  |
| --- |
| Field field = Unsafe.**class**.getDeclaredField(**"theUnsafe"**); field.setAccessible(**true**);*//解除封装* Unsafe u = (Unsafe) field.get(**null**); Singleton singleton = (Singleton)u.allocateInstance(Singleton.**class**); System.***out***.println(singleton); |

## 反射与简单java类



|  |
| --- |
| **package** com.pcq.react.simplebean;  **import** com.pcq.util.StringUtil;  **import** java.lang.reflect.Field; **import** java.lang.reflect.InvocationTargetException; **import** java.lang.reflect.Method;  **public class** ClassInstanceFactory {  **public static** <T> T create(Class<?> clazz, String value) {  Object object = **null**;  **try** {  object = clazz.getDeclaredConstructor().newInstance();  BeanUtils.*setValue*(object, value);  **return** (T)object;  } **catch** (InstantiationException e) {  e.printStackTrace();  } **catch** (IllegalAccessException e) {  e.printStackTrace();  } **catch** (InvocationTargetException e) {  e.printStackTrace();  } **catch** (NoSuchMethodException | NoSuchFieldException e) {  e.printStackTrace();  }  **return null**;  } }   */\*\*  \* 设置bean对象的属性值  \* 格式"param1:val1|param2:val2..."  \*/* **class** BeanUtils {  **public static void** setValue(Object obj, String value) **throws** NoSuchMethodException, InvocationTargetException, IllegalAccessException, NoSuchFieldException {  String[] keyVals = value.split(**"\\|"**);*//获得属性名：属性值的键值对* **for**(String temp : keyVals) {  String[] args = temp.split(**":"**);  Field field = obj.getClass().getDeclaredField(args[0]);  Method setMethod = obj.getClass().getMethod(**"set"** + StringUtil.*initCap*(args[0]), field.getType());*//获得set方法对象* Object val = *convertAttributeValueType*(field.getType().getName(), args[1]);  setMethod.invoke(obj, val);  }  }   */\*\*  \* 转换数据类型值  \** ***@return*** *\*/* **private static** Object convertAttributeValueType(String type, String value) {  *//System.out.println("type = " + type + ", value = " + value);* **if**(**"long"**.equals(type) || **"java.lang.Long"**.equals(type)) {  **return** Long.*parseLong*(value);  } **else if**(**"double"**.equals(type) || **"java.lang.Double"**.equals(type)) {  **return** Double.*parseDouble*(value);  } **else if**(**"int"**.equals(type) || **"java.lang.Integer"**.equals(type)) {  **return** Integer.*parseInt*(value);  }  **else** {  **return** value;  }  } } |