

# 注解和反射

## 1. 注解

注解

1.回顾我们之前所接触过的注解 @Override @Deprecated SuppressWarnings @FunctionalInterface

分析：以上注解 可以书写在不同的位置 有不同的作用 有的需要写值 有的不能写值

2.如何控制这些注解书写的位置、如何使这些注解有不同的作用 等等

通过@Target注解控制注解可以书写的位置

我们也可以通过自定义注解来实现这些效果

3.注解是由来

注解是JDK1.5新增的内容

4.注解用来解决什么问题呢？

早期的Java项目中 复杂度非常之高 所以项目中会存在很多的配置文件 阅读性差 书写性差

所以 在JDK1.5引入了注解来解决这个问题 最初的梦想是 零配置(配置文件全部消失)

目前的情况：注解 + 配置文件

注解也确实简略 省略了 大量的配置文件 但是配置文件依然是不能完全杜绝的

注解采用了一种思想：约定大于配置

5.元注解

用于修饰注解的注解 称之为元注解

@Target 用于规定注解书写的位置 不写表示此注解可以加在任何位置

@Retention 用于规定注解的保留策略

@Documented 被此注解修饰的注解可以保存在帮助文档中

@Inherited 被此注解修饰的注解 可以被子类继承

注解属性和赋值

注解属性支持的数据类型： 八种基本数据类型、String、枚举、Class类型 和 以上类型对应的数组类型

注解中的属性必须有值

注解属性的赋值：

1.如果注解中只有一个属性 并且属性名为value 则可以直接写值

- 2.如果为数组类型 一个元素直接赋值 多个元素 加上大括号
- 3.否则其他的情况都必须写为 属性名 = 属性值 这种写法
- 4.我们也可以使用default关键字给注解加上默认值

## 2. JUnit单元测试

JUnit Java Unit 单元测试框架 是一个专业的测试框架(别人写好的一些类 接口 方法 等等)

JUnit是Apache开源软件基金会维护的产品

回顾我们之前怎么测试我们写的代码 使用main方法 但是一个类只能有一个main方法 所以 不能满足我们实际的开发需求

我们可以使用JUnit单元测试框架

我们目前所使用src目录 表示源文件目录 是用于存放Java源文件的 所以规范而言 测试的代码 不应该存放在 src 目录下

应该单独创建一个测试目录

- 1.在项目下创建文件夹 取名为test
- 2.右键将此目录标记为测试资源根目录

@Test 注解 加在方法上 表示此方法可以单独执行

@Before 表示此类中的@Test修饰的方法执行之前都执行一次

@After 表示此类中的@Test修饰的方法执行之后都执行一次

@BeforeClass 本类中的方法执行之前只执行一次

@AfterClass 本类中的方法执行之后只执行一次

```
package com.atguigu.test;

import org.junit.*;

/**
 * @Test 注解 加在方法上 表示此方法可以单独执行
 * @Before 表示此类中的@Test修饰的方法执行之前都执行一次
 * @After 表示此类中的@Test修饰的方法执行之后都执行一次
 *
 * @BeforeClass 本类中的方法执行之前只执行一次
 * @AfterClass 本类中的方法执行之后只执行一次
 */
public class TestJUnit {
    @Before
    public void before(){
        System.out.println("执行前置操作");
    }

    @After
```

```

    public void after(){
        System.out.println("执行后置操作");
    }

    @BeforeClass
    public static void beforeClass(){
        System.out.println("本类中的方法执行之前只执行一次");
    }

    @AfterClass
    public static void afterClass(){
        System.out.println("本类中的方法执行之后只执行一次");
    }

    @Test
    public void m1(){
        System.out.println("m1方法执行");
    }

    @Test
    public void m2(){
        System.out.println(10 / 1);
    }

    @Test
    public void m3(){
        System.out.println("m3方法执行");
    }
}

```

### 3.反射

反射 在程序运行期间 动态的获取某个类的信息(属性、方法、构造方法) 并且访问

也就是不通过new对象的方式 依然可以访问类中的属性 方法 和 构造方法

生活中的反射：倒车镜 拍X光片 IDE的自动提示功能 等等

综合生活中常见的反射的操作 我们发现 反射在有些情况下是必不可少的

JUnit框架

如果在程序运行过程中 不知道需要什么类型的对象 以及 不知道需要获取多少个对象 该如何创建对象呢？

Spring IOC Inversion Of Control 容器 就是使用反射技术帮我们创建对象的

万物皆对象：类也是对象 方法也是对象 属性也是对象 构造器也是对象

java.lang.Class 用于表示类类型

java.lang.reflect.Field 字段类 类型

java.lang.reflect.Method 方法类 类型

java.lang.reflect.Constructor 构造器类 类型

### 3.1 获取属性

Field类 所有的字段都属于此类的对象

通过Class类提供的如下方法获取Field字段

Field getField(String fieldName) 根据指定的名称获取public修饰的属性

Field [] getFields() 获取本类中所有的public修饰的属性

```
package com.atguigu.test6;

import java.lang.reflect.Field;

/**
 * Field类 所有的字段都属于此类的对象
 * 通过Class类提供的如下方法获取Field字段
 *
 * Field getField(String fieldName) 根据指定的名称获取public修饰的属性
 * Field [] getFields() 获取本类中所有的public修饰的属性
 */
public class TestFiled1 {
    public static void main(String[] args) {
        try {
            // 根据全限定名获取Class对象
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            // 根据具体的名称获取单个字段对象
            Field heightField = stuClass.getField("height");

            // 打印字段对象的名称 和 类型
            System.out.println(heightField.getName() + "===" + heightField.getType());

            System.out.println("-----");

            // 获取所有的public修饰的字段对象
            Field[] fields = stuClass.getFields();

            // 遍历数组
            for(Field f : fields){
                // 打印字段对象的名称 和 类型
                System.out.println(f.getName() + "===" + f.getType());
            }

            System.out.println("-----");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

获取非公开的属性

Field getDeclaredField(String fieldName) : 根据属性名获取一个属性对象 可以是任何访问修饰符修饰

Field [] getDeclaredFields() : 获取所有属性对象 可以是任何访问修饰符修饰

Field类方法

第一个参数 表示给哪个对象的name属性赋值

第二个参数 具体的属性值

set(Object obj, Object value)

取值传参 表示声明 访问哪个对象的此属性值

get(Object obj)

```

package com.atguigu.test6;

import java.lang.reflect.Field;

/**
 * 获取非公开的属性
 * Field getDeclaredField(String fieldName) : 根据属性名获取一个属性对象 可以是任何访问修饰符修饰
 * Field [] getDeclaredFields() : 获取所有属性对象 可以是任何访问修饰符修饰
 *
 * Field类方法
 * 第一个参数 表示给哪个对象的name属性赋值
 * 第二个参数 具体的属性值
 * set(Object obj, Object value)
 *
 * 取值传参 表示声明 访问哪个对象的此属性值
 * get(Object obj)
 */
public class TestField2 {
    public static void main(String[] args) {
        try {
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            Field[] declaredFields = stuClass.getDeclaredFields();

            Object obj = stuClass.newInstance();

```

```

        for (Field declaredField : declaredFields) {
            System.out.println(declaredField.getName() + "===" + declaredField.getType());
        }

        System.out.println("-----");

        Field nameField = stuClass.getDeclaredField("name");

        // 调用此方法表示忽略JVM安全检查 即不再抛出异常 即可以访问了
        nameField.setAccessible(true);

        // 第一个参数 表示给哪个对象的名字属性赋值
        // 第二个参数 具体的属性值
        nameField.set(obj, "赵四");

        // 取值传参 表示声明 访问哪个学生对象的名字属性值
        System.out.println(nameField.get(obj));

    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    } catch (InstantiationException e) {
        throw new RuntimeException(e);
    } catch (IllegalAccessException e) {
        throw new RuntimeException(e);
    }
}
}

```

### 3.2 获取方法

Class类提供的两个方法

Method getMethod(String name, Class<?>... parameterType)

根据方法名和形参列表获取一个public或者继承自父类的方法

Method [] getMethods() 获取本类中所有public修饰的 和 继承自父类的方法

```

package com.atguigu.test7;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * Class类提供的两个方法
 * Method getMethod(String name, Class<?>... parameterType)
 * 根据方法名和形参列表获取一个public或者继承自父类的方法

```

```

*
* Method [] getMethods() 获取本类中所有public修饰的 和 继承自父类的方法
*/
public class TestMethod1 {
    public static void main(String[] args) {
        try {
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            Method[] methods = stuClass.getMethods();

            for (Method method : methods) {
                System.out.println(method.getName() + "-----" + method.getParameterCount());
            }

            System.out.println("-----");

            Method noParameterM4 = stuClass.getMethod("m4");

            Object obj = stuClass.newInstance();

            noParameterM4.invoke(obj);
            System.out.println("-----");

            Method intParameterM4 = stuClass.getMethod("m4", int.class);

            intParameterM4.invoke(obj, 100);

            System.out.println("-----");

            Method twoParameterM4 = stuClass.getMethod("m4", String.class, int.class);

            twoParameterM4.invoke(obj, "abc", 100);

        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e);
        } catch (InstantiationException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        } catch (InvocationTargetException e) {
            throw new RuntimeException(e);
        }
    }
}

```

Method getDeclaredMethod(String name, Class<?>... parameterType)

根据方法名和形参列表获取一个任意修饰符修饰的 本类中定义的方法 (不包括继承自父类的方法)

Method [] getDeclaredMethods() 获取本类中所有的任意修饰符修饰的已定义的方法(不包括继承自父类的方法)

```
package com.atguigu.test7;

import java.lang.reflect.Method;

/**
 * Method getDeclaredMethod(String name,Class<?>... parameterType)
 * 根据方法名和形参列表获取一个任意修饰符修饰的 本类中定义的方法 (不包括继承自父类的方法)
 *
 * Method [] getDeclaredMethods() 获取本类中所有的任意修饰符修饰的已定义的方法(不包括继承自父类的方法)
 */
public class TestMethod2 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

        Method[] declaredMethods = stuClass.getDeclaredMethods();

        for (Method declaredMethod : declaredMethods) {

            System.out.println(declaredMethod.getName() + "----" +
declaredMethod.getParameterCount());
        }

        System.out.println("-----");

        Method m1 = stuClass.getDeclaredMethod("m1");

        m1.setAccessible(true); // 忽略JVM安全检查 即可以访问

        Object obj = stuClass.newInstance();

        m1.invoke(obj);

    }
}
```

### 3.3 获取构造器

Class类提供如下方法获取构造器对象

Constructor getConstructor(Class<?> ...parameterType) 根据参数列表获取一个public修饰的构造器

Constructor [] getConstructors() 获取本类中所有的public修饰的构造器

Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)

注意区分



```

package com.atguigu.test8;

import com.atguigu.test6.Student;

import java.lang.reflect.Constructor;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:53
 * Class类提供如下方法获取构造器对象
 *
 * Constructor getConstructor(Class<?> ...parameterType) 根据参数列表获取一个public修饰的构造器
 * Constructor [] getConstructors() 获取本类中所有的public修饰的构造器
 *
 * Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)
 * 注意区分
 */
public class TestConstructors1 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

        Constructor<?> constructor = stuClass.getConstructor(String.class, String.class);

        Object obj = constructor.newInstance("hello", "world");

        if(obj instanceof Student){

            Student stu = (Student) obj;

            System.out.println("stu = " + stu);
        }

        System.out.println("-----");

        Constructor<?>[] constructors = stuClass.getConstructors();

        for (Constructor<?> con : constructors) {
            System.out.println(con.getName() + "=====" + con.getParameterCount());
        }

    }
}

```

Class类提供如下方法获取构造器对象

Constructor getDeclaredConstructor(Class<?> ...parameterType) 根据参数列表获取任意修饰符修饰的构造器

Constructor [] getDeclaredConstructors() 获取本类中所有任意修饰符修饰的构造器

Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)

注意区分

```
package com.atguigu.test8;

import com.atguigu.test6.Student;

import java.lang.reflect.Constructor;

/**
 * Class类提供如下方法获取构造器对象
 *
 * Constructor getDeclaredConstructor(Class<?> ...parameterType) 根据参数列表获取任意修饰符修饰的构造器
 * Constructor [] getDeclaredConstructors() 获取本类中所有任意修饰符修饰的构造器
 *
 * Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)
 * 注意区分
 *
 */
public class TestConstructors2 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

        Constructor<?> constructor = stuClass.getDeclaredConstructor();

        constructor.setAccessible(true);

        Object obj = constructor.newInstance();

        if(obj instanceof Student){

            Student stu = (Student) obj;

            System.out.println("stu = " + stu);
        }

        System.out.println("-----");

        Constructor<?>[] declaredConstructors = stuClass.getDeclaredConstructors();

        for (Constructor<?> con : declaredConstructors) {
            System.out.println(con.getName() + "----" + con.getParameterCount());
        }
    }
}
```

