# day33-基础加强【反射&BeanUtils&注解】

# 学习目标

01\_能够通过反射技术获取Class字节码对象

02\_能够通过反射技术获取构造方法对象，并创建对象。

03\_能够通过反射获取成员方法对象，并且调用方法。

04\_能够通过反射获取属性对象，并且能够给对象的属性赋值和取值。

05\_能够使用BeanUtils常用方法操作JavaBean对象

06\_能够说出注解的作用

07\_能够自定义注解和使用注解

08\_能够说出常用的元注解及其作用

09\_能够解析注解并获取注解中的数据

10\_能够完成注解的MyTest案例

# 项目清单

|  |
| --- |
| com.itheima.ref：反射案例  Demo01：获取Class对象的三种方式  Demo02：获取Class对象的信息  Demo03：获取Class对象的Constructor信息  Demo04：获取Class对象的Method信息  Demo05：获取Class对象的Field信息  Demo06：反射案例  com.itheima.beanu：BeanUtils案例  Demo01：BeanUtils常用方法  com.itheima.anno：注解案例  Demo01：常见注解  Book： 自定义注解  Demo02：使用自定义注解  Book： 使用元注解  Demo03：解析注解Book  Demo04：注解案例 |

# 回顾与提问

1\_以下定时服务的配置表示什么意思？

12 12 1,11,21,31 \* 2018 date >> /usr/local/my.txt

|  |
| --- |
| 2018年，每月，1，11，21，31号，的12时12分，执行date命令，  并将结果保存到/usr/local/my.txt文件中 |

2\_以下命令是什么意思？

cd /usr/local/nginx/sbin

./nginx

./nginx -s stop

|  |
| --- |
| 进行/usr/local/nginx/sbin目录  启动nginx服务器  停止nginx服务器。 |

# 问题引入

有的时候我们想知道别人写的类有哪些构造器，属性和方法，目的是调用方法。但别人出于保护的自己私有内容的情况下，他不给你看类的源代码。那么在这种情况下，我们有没有办法获取他的类中相关信息呢？

答案是肯定的：反射。

# 第01章 反射【重点】

## 1.1反射的概述

### 1.1.1什么是反射

反射是一种机制/功能，利用该机制/功能可以在程序运行过程中对类进行解剖并操作类中的构造方法，成员方法，成员属性。

|  |
| --- |
|  |

### 1.1.2反射的应用场景

1. 开发工具中写代码时的提示

|  |
| --- |
|  |

开发工具之所能够把该对象的方法和属性展示出来就使用利用了反射机制对该对象所有类进行了解剖获取到了类中的所有方法和属性信息，这是反射在IDE中的一个使用场景。

1. 各种框架的设计

|  |
| --- |
|  |

以上三个图标上面的名字就是Java的三大框架，简称SSH，SSM。

【SSH = Struts2 + Spring + Hibernate】

【SSM = Springmvc + Spring + MyBatis】

这三大框架的内部实现也大量使用到了反射机制，所以要想学好这些框架，则必须要

求对反射机制熟练了。

### 1.1.3使用反射机制解剖类的前提

必须先要获取到该类的字节码文件对象，即Class类型对象。关于Class描述字节码

文件如下图所示：

|  |
| --- |
|  |

说明：

1. Java中使用Class类表示class文件。
2. 任何一个class文件都是Class类的一个对象。

## 1.2获取Class对象的三种方式

### 1.2.1通过类名.class获取

|  |
| --- |
| **package** com.itheima.ref;  */\*\*  \* 学生  \*/* **public class** Student {   */\*\*  \* 成员属性  \*/* **private int id**;  **private** String **name**;   */\*\*  \* 构造方法  \*/* **public** Student(){  System.***out***.println(**"public Student()"**);  }  **public** Student(**int** id){  System.***out***.println(**"public Student(int id)"**);  }  **private** Student(**int** id,String name){  System.***out***.println(**"private Student(int id,String name)"**);  }   */\*\*  \* 成员方法  \*/* **public void** run(){  System.***out***.println(**"public void run()"**);  }  **public void** jump(**int** id){  System.***out***.println(**"public void jump(int id)"**);  }  **private void** sleep(**int** id,String name){  System.***out***.println(**"private void sleep(int id,String name)"**);  }  } |

|  |
| --- |
| */\*\*  \* 通过类名.class获取  \*/* **public void** test1() {  *//获取java.lang.String类在内存中的Class对象* Class clazz1 = java.lang.String.**class**;  *//获取Student类在内存中的Class对象* Class clazz2 = Student.**class**;  *//void都有对应的Class对象* Class clazz3 = **void**.**class**;  Class clazz4 = **int**[].**class**;  System.***out***.println(**"1:"**+clazz2); } |

### 1.2.2通过Object类的成员方法getClass()方法获取

|  |
| --- |
| */\*\*  \* 通过Object类的成员方法getClass()方法获取  \*/* **public void** test2() {  Student s = **new** Student();  Class clazz = s.getClass();  System.***out***.println(**"2:"**+clazz); } |

### 1.2.3通过Class.forName("全限定类名")方法获取

|  |
| --- |
| */\*\*  \* 通过Class.forName("全限定类名")方法获取  \*/* **public void** test3() **throws** Exception{  Class clazz = Class.*forName*(**"com.itheima.ref.Student"**);  System.***out***.println(**"3:"**+clazz); } |

## 1.3获取Class对象的信息

知道怎么获取Class对象之后，接下来就介绍几个Class类中常用的方法了。

### 1.3.1获取简单类名

String getSimpleName(); 获取简单类名，只是类名，没有包

|  |
| --- |
| */\*\*  \* 获取简单类名  \* String getSimpleName(); 获取简单类名，只是类名，没有包  \* 单元测试  \* Junit  \*/* @Test **public void** test1(){  Class clazz = Student.**class**;  *//获取简单类名* String simpleName = clazz.getSimpleName();  System.***out***.println(simpleName); } |

### 1.3.2获取完整类名

String getName(); 获取完整类名，包含包名 + 类名

|  |
| --- |
| */\*\*  \* 获取完整类名  \* String getName(); 获取完整类名，包含包名 + 类名  \*/* @Test **public void** test2() {  Class clazz = Student.**class**;  *//获取全限定类名，包名+类名* String name = clazz.getName();  System.***out***.println(name); } |

### 1.3.3创建对象

T newInstance() ; 创建此 Class对象所表示的类的一个新实例。要求：类必须有

public的无参数构造方法

|  |
| --- |
| */\*\*  \* 通创建对象  \* T newInstance() ; 创建此 Class对象所表示的类的一个新实例。要求：类必须有  \* public的无参数构造方法  \*/* @Test **public void** test3() **throws** Exception{  *//在内存中的Class对象* Class clazz = Student.**class**;  *//根据Class对象创建实例，底层会调用Student类的无参构造器* Object obj = clazz.newInstance();  **if**(obj **instanceof** Student){  *//强转成Student* Student s = (Student) obj;  System.***out***.println(s.hashCode());  } } |

## 1.4获取Class对象的Constructor信息

一开始在阐述反射概念的时候，我们说到利用反射可以在程序运行过程中对类进行

解剖并操作里面的成员。而一般常操作的成员：构造方法，成员方法，成员属性，

那么接下来看看怎么利用反射来操作这些成员以及操作这些成员能干什么，先来看

看怎么操作构造方法。而要通过反射操作类的构造方法，我们需要先知道一个

Constructor类。

### 1.4.1 Constructor类概述

Constructor是构造方法类，类中的每一个构造方法都是Constructor的对象，通过

Constructor对象可以实例化对象。

|  |
| --- |
|  |

### 1.4.2 Class类中与Constructor相关方法

1. Constructor[] getConstructors()

获取所有的public修饰的构造方法

2. Constructor[] getDeclaredConstructors()

获取所有构造方法，包括privat修饰的

3. Constructor getConstructor(Class... parameterTypes)

根据参数类型获取构造方法对象，只能获得public修饰的构造方法

4. Constructor getDeclaredConstructor(Class... parameterTypes)

根据参数类型获取构造方法对象，包括private修饰的构造方法

### 1.4.3 Constructor类中常用方法

1. T newInstance(Object... initargs)

根据指定参数创建对象

2. void setAccessible(true)

暴力反射，设置为可以直接访问私有类型的构造方法

### 1.4.4示例代码

|  |
| --- |
| **package** com.itheima.ref;  **import** org.junit.Test; **import** java.lang.reflect.Constructor;  */\*\*  \* 获取Class对象的Constructor信息  \*/* **public class** Demo03 {   */\*\*  \* 根据参数类型获取构造方法对象，只能获得public修饰的构造方法  \* Constructor getConstructor(Class... parameterTypes)  \*  \* 根据指定参数创建对象  \* T newInstance(Object... initargs)  \*/* @Test  **public void** test1() **throws** Exception {  Class clazz = Student.**class**;  *//Constructor[] cs1 = clazz.getConstructors();//public  //Constructor[] cs2 = clazz.getDeclaredConstructors();//public+private  //System.out.println(cs1.length);  //System.out.println(cs2.length);* Constructor c = clazz.getConstructor(**int**.**class**);  Object obj = c.newInstance(10);  System.***out***.println(obj);  }   */\*\*  \* 根据参数类型获取构造方法对象，包括private修饰的构造方法  \* Constructor getDeclaredConstructor(Class... parameterTypes)  \*  \* 暴力反射，设置为可以直接访问私有类型的构造方法  \* void setAccessible(true)  \*  \* 根据指定参数创建对象  \* T newInstance(Object... initargs)  \*/* @Test  **public void** test2() **throws** Exception{  Class clazz = Student.**class**;  Constructor c = clazz.getDeclaredConstructor(**int**.**class**,String.**class**);  *//如果构造器是private的，默认情况下不能调用，得用暴力反射  //顺序很重是* c.setAccessible(**true**);  Object obj = c.newInstance(10,**"哈哈"**);  System.***out***.println(obj);  }  } |

## 1.5获取Class对象的Method信息

操作完构造方法之后，就来看看反射怎么操作成员方法了。同样在操作成员方法

之前我们需要学习一个类：Method类。

### 1.5.1 Method类概述

Method是方法类，类中的每一个方法都是Method的对象，通过Method对象可以

调用方法。

|  |
| --- |
|  |

### 1.5.2 Class类中与Method相关方法

1. Method[] getMethods()

获取所有的public修饰的成员方法，包括父类中

2. Method[] getDeclaredMethods()

获取当前类中所有的方法，包含私有的，不包括父类中

3. Method getMethod("方法名", 方法的参数类型... 类型)

根据方法名和参数类型获得一个方法对象，只能是获取public修饰的

4. Method getDeclaredMethod("方法名", 方法的参数类型... 类型)

根据方法名和参数类型获得一个方法对象，包括private修饰的

### 1.5.3 Method类中常用方法

1. Object invoke(Object obj, Object... args)

根据参数args调用对象obj的该成员方法

如果obj=null，则表示该方法是静态方法

2. void setAccessible(true)

暴力反射，设置为可以直接调用私有修饰的成员方法

### 1.5.4示例代码

|  |
| --- |
| **package** com.itheima.ref;  **import** org.junit.Test; **import** java.lang.reflect.Method;  */\*\*  \* 获取Class对象的Method信息  \*/* **public class** Demo04 {   */\*\*  \* 根据方法名和参数类型获得一个方法对象，只能是获取public修饰的  \* Method getMethod("方法名", 方法的参数类型... 类型)  \*  \* 根据参数args调用对象obj的该成员方法  \* Object invoke(Object obj, Object... args)  \*/* @Test  **public void** test1() **throws** Exception{  */\*  Class clazz = Student.class;  //Method[] ms = clazz.getMethods();  Method[] ms = clazz.getDeclaredMethods();  for (Method m : ms) {  //获取方法的名字  System.out.println(m.getName());  }  \*/* Class clazz = Student.**class**;  Object obj = clazz.newInstance();  *//参数一：方法名  //参数二：方法参数的类型* Method m = clazz.getMethod(**"jump"**,**int**.**class**);  *//执行方法  //参数一：对象  //参数二：方法的实参* m.invoke(obj,22);  }   */\*\*  \* 根据方法名和参数类型获得一个方法对象，包括private修饰的  \* Method getDeclaredMethod("方法名", 方法的参数类型... 类型)  \*  \* 暴力反射，设置为可以直接调用私有修饰的成员方法  \* void setAccessible(true)  \*  \* 根据参数args调用对象obj的该成员方法  \* Object invoke(Object obj, Object... args)  \*/* @Test  **public void** test2() **throws** Exception{  Class clazz = Student.**class**;  Object obj = clazz.newInstance();  *//参数一：方法名  //参数二：方法参数的类型* Method m = clazz.getDeclaredMethod(**"sleep"**,**int**.**class**,String.**class**);  *//暴力反射* m.setAccessible(**true**);  *//执行方法  //参数一：对象  //参数二：方法的实参* m.invoke(obj,22,**"呵呵"**);   *//如果是static方法，在调用时obj换成null* }   */\*\*  \* 反射调用static的方法  \*/* @Test  **public void** test3() **throws** Exception{  Class clazz = Student.**class**;  Method m = clazz.getMethod(**"eat"**,**null**);  m.invoke(**null**,**null**);  }  } |

## 1.6获取Class对象的Field信息

### 1.6.1 Field类概述

Field是属性类，类中的每一个属性都是Field的对象，通过Field对象可以

给对应的属性赋值和取值。

|  |
| --- |
|  |

### 1.6.2 Class类中与Field相关方法

1. Field[] getFields()

获取所有的public修饰的属性对象，返回数组

2. Field[] getDeclaredFields()

获取所有的属性对象，包括private修饰的，返回数组

3. Field getField(String name)

根据属性名获得属性对象，只能获取public修饰的

4. Field getDeclaredField(String name)

根据属性名获得属性对象，包括private修饰的

### 1.6.3 Field类中常用方法

set通用方法都是给对象obj的属性设置使用  
get通用方法是获取对象obj对应的属性值的

void setAccessible(true);暴力反射，设置为可以直接访问私有类型的属性

### 1.6.4 示例代码

|  |
| --- |
| **package** com.itheima.ref;  **import** org.junit.Test; **import** java.lang.reflect.Field;  */\*\*  \* 获取Class对象的Field信息  \*/* **public class** Demo05 {   */\*\*  \* 获取所有的属性对象，包括private修饰的，返回数组  \* Field[] getDeclaredFields()  \*/* @Test  **public void** test1() **throws** Exception{  Class clazz = Class.*forName*(**"com.itheima.ref.Student"**);  Field[] fs = clazz.getDeclaredFields();  **for**(Field f : fs){  *//获取Field的名字和类型* System.***out***.println(f.getName()+**"#"**+f.getType());  }  }   */\*\*  \* 根据属性名获得属性对象，包括private修饰的  \* Field getDeclaredField(String name)  \*  \* 暴力反射，设置为可以直接访问私有类型的属性  \* void setAccessible(true);  \*  \* set通用方法都是给对象obj的属性设置使用  \* get通用方法是获取对象obj对应的属性值的  \*/* @Test  **public void** test2() **throws** Exception {  Class clazz = Class.*forName*(**"com.itheima.ref.Student"**);  Object obj = clazz.newInstance();  *//获取id属性* Field idField = clazz.getDeclaredField(**"id"**);  idField.setAccessible(**true**);  *//为id属性设置值  //参数一：对象  //参数二：实际值  //即id=2018* idField.setInt(obj,2018);    *//获取name属性* Field nameField = clazz.getDeclaredField(**"name"**);  nameField.setAccessible(**true**);  *//为name属性设置值  //参数一：对象  //参数二：实际值  //即name=黄克* nameField.set(obj,**"黄克"**);   *//获取id和name的值  //参数一：对象  //对象.getId()* System.***out***.println(idField.get(obj));  System.***out***.println(nameField.get(obj));  }  } |

## 1.7反射案例

### 1.7.1案例需求

编写一个方法可以根据配置文件产任意类型的对象，

并读取属性文件中的值，设置到这个类型的对象中。

### 1.7.2案例分析

1\_已知：在src/obj.properties创建一个属性文件。

|  |
| --- |
| obj.properties  **class**=**com.itheima.ref.User id**=**2018 name**=**赵君 sal**=**5555.55** |

2\_已知：创建User.java类。

|  |  |
| --- | --- |
| User.java  */\*\*  \* 用户  \*/* **public class** User {   **private int id**;  **private** String **name**;  **private double sal**;   **public** User(){}   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  System.***out***.println(**"setId()"**);  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  System.***out***.println(**"setName()"**);  }   **public double** getSal() {  **return sal**;  }   **public void** setSal(**double** sal) {  **this**.**sal** = sal;  System.***out***.println(**"setSal()"**);  } } |  |

3\_定义一个方法：createObject()，方法返回值类型为：Object。

4\_创建Properties对象，读取user.properties文件。

5\_根据class获得类全名，并通过反射技术创建对象并获得Class对象。

6\_用Properties对象的stringPropertyNames()方法，读取user.properties文件中的

所有属性到集合中。

7\_遍历集合，根据属性值找到Class对象的Field对象，并设置暴力反射。

8\_根据Field的类型，调用setXxx方法为该对象设置值。

9\_最后，返回这个新创建的对象。

### 1.7.3案例代码

|  |
| --- |
| **package** com.itheima.ref;  **import** java.io.FileInputStream; **import** java.io.InputStream; **import** java.lang.reflect.Field; **import** java.lang.reflect.Type; **import** java.util.Properties; **import** java.util.Set;  */\*\*  \* 反射案例  \* 已知：  \* 1\_src/user.properties  \* 2\_com.itheima.ref.User  \* 求：  \* public Object createObject(){}如何写？  \*/* **public class** Demo06 {    **public** Object createObject() **throws** Exception {   *//读取user.properties属性文件，/表示src目录* String path = Demo06.**class**.getResource(**"/user.properties"**).getPath();  InputStream is = **new** FileInputStream(path);  Properties props = **new** Properties();  props.load(is);   *//获取属性文件中的class的值* String className = props.getProperty(**"class"**);  *//获取Class对象* Class clazz = Class.*forName*(className);  *//创建对象* Object obj = clazz.newInstance();   *//一次性读取属性文件的所有数据，是一个集合，里面只有属性名* Set<String> set = props.stringPropertyNames();  **for** (String name : set) {  *//排除class* **if**(!name.equals(**"class"**)){  *//获取属性文件的名  //System.out.println(name+"#");  //获取属性文件的值* String value = props.getProperty(name);  *//System.out.println(value);   //获取User对象的private属性，获取到id/name/sal* Field field = clazz.getDeclaredField(name);  *//暴力反射* field.setAccessible(**true**);   *//获取属性的类型，int/double/String* Type type = field.getType();  *//进行判断，根据不同类型，进行设置值* **if**(**int**.**class** == type){  *//将String转成不同的类型* field.setInt(obj,Integer.*parseInt*(value));  }**else if**(**double**.**class** == type){  field.setDouble(obj,Double.*parseDouble*(value));  }**else if**(String.**class** == type){  field.set(obj,value);  }   }  }*//for end* **return** obj;  }      **public static void** main(String[] args) **throws** Exception {  Demo06 test = **new** Demo06();  User user = (User)test.createObject();  System.***out***.println(**"编号:"**+user.getId());  System.***out***.println(**"姓名:"**+user.getName());  System.***out***.println(**"工资:"**+user.getSal());  } } |

# 第02章 BeanUtils【重点】

## 2.1 BeanUtils的概述

### 2.1.1什么是BeanUtils

BeanUtils是Apache Commons组件的成员之一，主要用于简化JavaBean封装数据

的操作。常用的操作有以下三个：

1. 对JavaBean的属性进行赋值和取值。
2. 将一个JavaBean所有属性赋值给另一个JavaBean对象中。
3. 将一个Map集合的数据封装到一个JavaBean对象中。

|  |
| --- |
|  |

### 2.1.2什么是JavaBean

JavaBean就是一个类，但该类需要满足以下三个条件：

1. 类必须使用public修饰。
2. 提供无参数的构造器。
3. 提供getter和setter方法访问属性。

JavaBean的两个重要概念

字段：就是成员变量，字段名就是成员变量名。

属性：属性名通过setter/getter方法去掉set/get前缀，首字母小写获得。

比如：setName()--> Name--> name

一般情况下，字段名和属性名是一致的。

|  |
| --- |
|  |

### 2.1.3 BeanUtils的应用场景

1. 快速将一个JavaBean各个属性的值，赋值给具有相同结构的另一个JavaBean中。

2. 快速收集表单中的所有数据到JavaBean中。

### 2.1.4 BeanUtils相关JAR包

BeanUtils是第三方组织写的工具类，要使用它，得先下载对应的工具JAR包。

1. 下载地址：http://commons.apache.org/



1. 相关的jar包

commons‐beanutils‐1.9.3.jar // 工具核心包

commons‐logging‐1.2.jar // 日志记录包

commons‐collections‐3.2.2.jar // 增强的集合包

## 2.2 BeanUtils的基本使用

### 2.2.1导入相关JAR包

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

### 2.2.2 BeanUtils常用方法

1. public static void setProperty(Object bean, String name, Object value)

给指定对象bean的指定name属性赋值为指定值value。

如果指定的属性不存在，则什么也不发生。

2.public static String getProperty(Object bean, String name)

获取指定对象bean指定name属性的值。

如果指定的属性不存在，则会抛出异常。

注意；当属性的类型是数组类型时，获取到的值数组中的第一个值。

3.public static void copyProperties(Object dest, Object orig)

将对象orig的属性值赋值给对象dest对象对应的属性

注意：只有属性名名相同且类型一致的才会赋值成功。

4. public static void populate(Object bean, Map<String, ? extends Object>

properties)

将一个Map集合中的数据封装到指定对象bean中

注意：对象bean的属性名和Map集合中键要相同。

### 2.2.3演示BeanUtils常用方法

#### 2.2.3.1对JavaBean的属性进行赋值和取值

|  |
| --- |
| **package** com.itheima.beanu;  */\*\*  \* 用户  \*/* **public class** User {   **private int id**;  **private** String **name**;  **private double sal**;   **public** User(){}   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public double** getSal() {  **return sal**;  }   **public void** setSal(**double** sal) {  **this**.**sal** = sal;  }  } |

|  |
| --- |
| */\*\*  \* 对JavaBean的属性进行赋值和取值，如：User对象  \*/* @Test **public void** test1() **throws** Exception{  User user = **new** User();  BeanUtils.*setProperty*(user,**"id"**,**"2018"**);  BeanUtils.*setProperty*(user,**"name"**,**"哈哈"**);  BeanUtils.*setProperty*(user,**"sal"**,**"5555"**);   System.***out***.println(BeanUtils.*getProperty*(user,**"id"**));  System.***out***.println(BeanUtils.*getProperty*(user,**"name"**));  System.***out***.println(BeanUtils.*getProperty*(user,**"sal"**)); } |

#### 2.2.3.2将一个JavaBean对象的属性赋值给另一个JavaBean对象

|  |
| --- |
| **package** com.itheima.beanu;  */\*\*  \* 管理员  \*/* **public class** Admin {   **private int id**;  **private** String **name**;  **private** String **tel**;   **public** Admin(){}   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getTel() {  **return tel**;  }   **public void** setTel(String tel) {  **this**.**tel** = tel;  }  } |

|  |
| --- |
| */\*\*  \* 将一个JavaBean对象的属性赋值给另一个JavaBean对象，如：User对象的属性赋给Admin对象  \*/* @Test **public void** test2() **throws** Exception{  User user = **new** User(2018,**"张三"**,6666);  User otherUser = **new** User();  *//将user对象的属性值一一设置到otherUser对象中属性去* BeanUtils.*copyProperties*(otherUser,user);  System.***out***.println(otherUser.getId());  System.***out***.println(otherUser.getName());  System.***out***.println(otherUser.getSal()); } |

#### 2.2.3.3将一个Map集合的数据封装到一个JavaBean对象中

|  |
| --- |
| */\*\*  \* 将一个Map集合的数据封装到一个JavaBean对象中，如：Map<String,Object>的值赋给User对象  \*/* @Test **public void** test3() **throws** Exception{  *//准备好集合* Map<String,Object> map = **new** LinkedHashMap<String,Object>();  map.put(**"id"**,2018);  map.put(**"name"**,**"李四"**);  map.put(**"sal"**,9999.99);  *//准备好User对象* User user = **new** User();  *//将Map的值，根据key，一一封装到User对象中去* BeanUtils.*populate*(user,map);  *//显示* System.***out***.println(user.getId());  System.***out***.println(user.getName());  System.***out***.println(user.getSal()); } |

# 第03章 注解

## 3.1注解的概述

### 3.1.1什么是注解

注解（Annotation）相当于一种标记，在程序中加入注解就等于为程序打上某种标

记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种

元素上有无标记，看你的程序有什么标记，就去干相应的事，标记可以加在

包、类，属性、方法，方法的参数以及局部变量上定义。

### 3.1.2注解的应用场景

1. 重写父类方法时。

|  |
| --- |
| **public class** Demo01 {  @Override  **public** String toString(){  **return "123"**;  } } |

1. 运行一个无main的方法时，带有@Test的成员方法可以直接运行，无需main方法。

|  |
| --- |
| **public class** Demo01 {  @Test  **public void** test1(){  System.***out***.println(**"test1"**);  }  @Test  **public void** test2(){  System.***out***.println(**"test2"**);  } } |

|  |
| --- |
|  |

1. 框架中的配置。

现在框架都是基于注解的配置，简单方便。

### 3.1.3常见注解

JDK中内置了很多注解，以下例举一些：

1. @author：用来标识作者名。

2. @version：用于标识对象的版本号，适用范围：文件、类、方法。

3. @Override ：用来修饰方法声明，告诉编译器该方法是重写父类中的方法，

如果父类不存在该方法，则编译失败

|  |
| --- |
| */\*\*  \** ***@author*** *赵君  \** ***@version*** *1.0   \*/* **public class** Demo01 {  @Override  **public** String toString(){  **return "123"**;  }  @Test  **public void** test1(){  System.***out***.println(**"test1"**);  }  @Test  **public void** test2(){  System.***out***.println(**"test2"**);  } } |

## 3.2自定义注解

就算JDK中内置了很多注解，但也不一定能满足所有企业的需求，所以不排除我们需要

根据需求自定义注解。

### 3.2.1定义格式

public @interface 注解名{

}

如：定义一个名为Book的注解

public @interface Book{

}

以上定义出来的注解就是一个最简单的注解了，但这样的注解意义不大，因为注解中

没有任何内容，就好像我们定义一个类而这个类中没有任何成员变量和方法一样，这

样的类意义也是不大的，所以在定义注解时会在里面添加一些成员来让注解功能更加

强大，这些成员就是属性。接下来就看看怎么给注解添加属性。

### 3.2.2注解的属性

1. 属性的作用

可以让用户在使用注解时传递参数，让注解的功能更加强大。

2. 属性的格式

格式1：数据类型 属性名();

格式2：数据类型 属性名() default 默认值;

3. 属性定义示例

|  |
| --- |
| Book.java  */\*\*  \* 自定义注解  \*/* **public** @**interface** Book { } |

4. 属性适用的数据类型

八种基本数据类型（int,float,boolean,byte,double,char,long,short)

String类型，Class类型，枚举类型，注解类型

以上所有类型的一维数组

## 3.3使用自定义注解

### 3.3.1在成员方法上使用自定义注解

|  |
| --- |
| Demo02.java  @Book **public void** test1(){ } |

1. 使用注意事项

如果属性有默认值，则使用注解的时候，这个属性可以不用赋值。

如果属性有默认值，则使用注解的时候，这个属性如果赋值了，以赋值为准。

如果属性没有默认值，那么在使用注解时一定要给属性赋值。

### 3.3.2 一个特殊的注解属性value

1. 当注解中只有一个属性且名称是value，在使用注解时给value属性赋值可以直接给

属性值，无论value是单值元素还是数组类型。

|  |
| --- |
| Book.java  */\*\*  \* 自定义一个注解  \*/* **public** @**interface** Student {  String value(); } |

1. 如果注解中除了value属性还有其他属性，且至少有一个属性没有默认值，则在使

用注解给属性赋值时，value属性名不能省略

|  |
| --- |
| Book.java  */\*\*  \* 自定义一个注解  \*/* **public** @**interface** Student {  String value();  **int** age() **default** 44; } |

## 3.4元注解

### 3.4.1什么是元注解

用来定义注解的注解。

|  |
| --- |
| @Target(ElementType.***METHOD***) @Retention(RetentionPolicy.***SOURCE***) **public** @**interface** Override { } |

### 3.4.2元注解的作用

用来说明自定义注解的在类中出现的位置和生命周期。

### 3.4.3常用元注解

#### 3.4.3.1 @Target

作用：定义该注解用在哪个位置，如果不写，默认是类中任何地方都可以使用。

可选的参数值在枚举类ElemenetType中包括：

TYPE： 用在类,接口上

FIELD：用在成员变量上

METHOD： 用在方法上

PARAMETER：用在参数上

CONSTRUCTOR：用在构造方法上

LOCAL\_VARIABLE：用在局部变量上

#### 3.4.3.2 @Retention

作用：定义该注解的生命周期。

可选的参数值在枚举类型RetentionPolicy中包括：

SOURCE：注解只存在于Java源代码中，编译生成的字节码文件中就不存在了。

CLASS：注解存在于Java源代码、编译以后的字节码文件中，运行的时候内存中没

有，这个是默认值。

RUNTIME：注解存在于Java源代码中、编译以后的字节码文件中、运行时内存中，

程序可以通过反射获取该注解。

### 3.4.4使用元注解

|  |
| --- |
| Book.java  */\*\*  \* 自定义注解  \*/* @Retention(value=RetentionPolicy.***RUNTIME***) @Target(value={ElementType.***FIELD***,ElementType.***METHOD***}) **public** @**interface** Book {  **int** id() **default** 10;*//图书编号* String author() **default "赵君"**;*//作者* **double** price() **default** 11.11;*//单价* } |

|  |
| --- |
|  |

## 3.5解析注解

### 3.5.1什么是解析注解

通过Java技术，在程序运行时，获取注解中所有属性的过程则称为解析注解。

### 3.5.2与注解解析相关的接口

Anontation：所有注解类型的公共接口，类似所有类的父类是Object。

AnnotatedElement：定义了与注解解析相关的方法，常用方法以下四个：

1. boolean isAnnotationPresent(Class annotationClass); 判断当前对象是否有

指定的注解，有则返回true，否则返回false。

1. T getAnnotation(Class<T> annotationClass); 获得当前对象上指定的注解对

象。

1. Annotation[] getAnnotations(); 获得当前对象及其从父类上继承的所有的注解对

象。

1. Annotation[] getDeclaredAnnotations();获得当前对象上所有的注解对象，不包

括父类的。

### 3.5.3获取注解数据的原理

注解作用在那个成员上，就通过反射获得该成员的对象来得到它的注解。

如：

如注解作用在方法上，就通过方法(Method)对象得到它的注解

|  |
| --- |
| // 得到方法对象  Method method = clazz.getDeclaredMethod("方法名");  // 根据注解名得到方法上的注解对象  Book book = method.getAnnotation(Book.class); |

如注解作用在类上，就通过Class对象得到它的注解

|  |
| --- |
| // 获得Class对象  Class c = 类名.class;  // 根据注解的Class获得使用在类上的注解对象  Book book = c.getAnnotation(Book.class); |

### 3.5.4使用反射获取注解的数据

|  |
| --- |
| Demo03.java  */\*\*  \* 解析注解Book  \*/* **public class** Demo03 {   */\*\*  \* 在程序运行时，获取注解Book的所有属性值  \*/* @Book  **public void** test1() **throws** Exception{  *//@Book->test1()->Class   //获取Demo03类的Class对象* Class clazz = Demo03.**class**;  *//获取Class对象的Method对象* Method m = clazz.getMethod(**"test1"**,**null**);  *//获取Method对象中的@Book注解  //参数一：注解类型的Class  //返回值：注解* Book book = m.getAnnotation(Book.**class**);  *//获取Book注解里面的所有属性值* System.***out***.println(book.id());  System.***out***.println(book.author());  System.***out***.println(book.price());  }    **public static void** main(String[] args) **throws** Exception{  Demo03 test = **new** Demo03();  test.test1();;  }  } |

## 3.6注解案例

### 3.6.1案例需求

模拟JUnit测试的@Test注释

### 3.6.2案例分析

1. 模拟JUnit测试的@Test注释，首先需要编写自定义注解@MyTest，并添加元注解，

保证自定义注解只能修饰方法，且在运行时可以获得。

2. 编写测试类，给目标方法使用 @MyTest注解，编写三个方法，其中两个加上@MyTest注解。

3. 编写调用类，使用main方法调用测试类，模拟JUnit的运行，只要有@MyTest注

释的方法都会运行。

### 3.6.3案例代码

|  |
| --- |
| */\*\*  \** ***@MyTest注解*** *\*/* @Retention(value= RetentionPolicy.***RUNTIME***) @Target(value= ElementType.***METHOD***) **public** @**interface** MyTest { } |

|  |
| --- |
| */\*\*  \* 使用MyTest注解  \*/* **public class** MyTestDemo {  @MyTest  **public void** test1(){  System.***out***.println(**"public void test1()"**);  }  **public void** test2(){  System.***out***.println(**"public void test2()"**);  }  @MyTest  **public void** test3(){  System.***out***.println(**"public void test3()"**);  } } |

|  |
| --- |
| **package** com.itheima.anno;  **import** java.lang.reflect.Method;  */\*\*  \* 注解案例  \*/* **public class** Demo04 {  **public static void** main(String[] args) **throws** Exception{  Class clazz = MyTestDemo.**class**;   Method[] ms = clazz.getMethods();   **for**(Method m : ms){  *//判断方法前是否有@MyTest注解* **boolean** flag = m.isAnnotationPresent(MyTest.**class**);  *//如果存在* **if**(flag){  *//调用方法* m.invoke(clazz.newInstance(),**null**);  }  }  } } |

# 随堂测试