# 1. 注解

实际上我们之前遇到过注解，比如重写方法时出现的@Override。注解的作用基本上有两个：一是告诉编译器如何运行程序，二是代替或者简化配置文件，即可用于对程序的配置。

常用的注解：

@Override表示重写父类方法；@SuppressWarnings表示抑制编译器警告；@Deprecated表示标记此方法已经过时。例如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.List;  **public class** Demo {  @Override  **public** String toString() {  **return super**.toString();  }  *// unused表示抑制变量未使用警告，unchecked表示抑制Java语言检查警告* @SuppressWarnings({**"unused"**, **"unchecked"**})  **public void** useList() {  List list = **new** ArrayList();  }  @Deprecated  **public void** testm() {  } } |

## 1.1 自定义注解

通过自定义注解，可以在类、字段和方法上添加描述信息。说明：注解和类、接口是同一个级别的。

首先，用@interface定义一个注解：

|  |
| --- |
| **package** com.zhang.test;  *// 定义一个Author注解，用于描述一个作者* **public** @**interface** Author {  *// 其中写注解的属性。属性的修饰符是public，不能显式声明  // 注解的“属性”写法看上去像方法，但是不能有主体内容* String name(); *// 定义作者姓名属性* **int** age(); *// 定义作者年龄属性* } |

然后，在类、方法和属性上就能使用此注解描述信息。这里必须写全注解性

|  |
| --- |
| **package** com.zhang.test;  @Author(name = **"张三"**, age = 12) **public class** Demo {  @Author(name = **"张三"**, age = 13)  **private** String **addr**;  @Author(name = **"李四"**, age = 14)  **public void** test() {} } |

如果定义注解时，给某属性设置了默认值，那么使用此注解时，该值就可以不写而使用默认值。比如：

|  |
| --- |
| **public** @**interface** Author {  String name();  **int** age() **default** 20; *// 设置了默认值0* } |

使用时（默认使用了age为20的属性值）：

|  |
| --- |
| @Author(name = **"张三"**) **public class** Test { } |

如果注解中只定义了一个属性，且该属性名为value，那么使用此注解时，可直接在括号中写属性值。例如：

|  |
| --- |
| **public** @**interface** Author {  String value(); }  @Author(**"张三"**) *//直接写值* **public class** Test { } |

若注解中某属性值为数组类型，则使用时，该属性值应该用大括号包起来，各项之间用逗号分隔：

|  |
| --- |
| **public** @**interface** Author {  String[] value() **default** {**"test1"**, **"test2"**}; }  @Author({**"张三"**, **"李四"**}) **public class** Test { } |

## 1.2 元注解

元注解用于指明注解可用的范围和注解的生命周期，元注解是“注解的注解”。

（1）可用@Target注解来指定注解的可用范围。Target注解只有一个value属性，类型是数组，可选用下述枚举类型作为此属性的枚举值：

|  |  |
| --- | --- |
| ElementType.TYPE | 可用范围为类、接口和枚举 |
| ElementType.FIELD | 可用范围为字段 |
| ElementType.METHOD | 可用范围为方法 |
| ElementType.PARAMETER | 可用范围为参数 |
| ElementType.CONSTRUCTOR | 可用范围为构造器 |
| ElementType.LOCAL\_VARIABLE | 可用范围为局部变量 |
| ElementType.ANNOTATION\_TYPE | 可用范围为注解 |
| ElementType.PACKAGE | 可用范围为包 |

例如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.lang.annotation.ElementType; **import** java.lang.annotation.Target;  *// 说明此注解只能在类、接口、枚举和方法上使用* @Target({ElementType.***TYPE***, ElementType.***METHOD***}) **public** @**interface** Author {  String value(); } |

使用时：

|  |
| --- |
| **package** com.zhang.test;  **import** java.lang.annotation.Target;  @Author(**"张三"**) **public class** Demo {  @Author(**"李四"**)  **public void** test() {}  @Author(**"王五"**) *// 这个就提示错误了* **private int age**; } |

（2）可用@Retention注解指定注解的生命周期。此注解只能有一个value属性，属性类型是RetentionPolicy，不再是数组了，即不能“多选”。RetentionPolicy枚举类中有如下枚举可供选择：

|  |  |
| --- | --- |
| SOURCE | 注解只在源码级别有效，一般这样的注解是给编译器看的 |
| CLASS | 默认值，注解在字节码级别被保留，但是运行时无效 |
| RUNTIME | 注解在字节码级别被保留，运行时有效。可被注解反射读取。大多会用此值 |

## 1.3 注解反射

之前用注解描述了类的一些信息，但是并没能好好利用注解。要想利用注解简化配置，必须要在程序中读取到所配置的注解的属性值。只能通过注解反射来得到这些信息。主要注意的是，必须将注解的生命周期配置为RUNTIME，否则运行时用注解反射是读取不到注解信息的。

例子：

（1）定义Author注解

|  |
| --- |
| **package** com.zhang.test;  **import** java.lang.annotation.ElementType; **import** java.lang.annotation.Retention; **import** java.lang.annotation.RetentionPolicy; **import** java.lang.annotation.Target;  @Target({ElementType.***TYPE***, ElementType.***METHOD***, ElementType.***FIELD***}) @Retention(RetentionPolicy.RUNTIME) **public** @**interface** Author {  String name();  **int** age() **default** 20;  String remark(); *// 评论属性* } |

（2）使用注解反射

|  |
| --- |
| **package** com.zhang.test;  **import** java.lang.reflect.Method;  @Author(name = **"张三"**, age = 34, remark = **"此人写作很好"**) **public class** Demo {  @Author(name = **"李四"**, age = 12, remark = **"少年得志"**)  **public void** save() **throws** NoSuchMethodException {  *// 顺便就在此方法中获得上述两个注解的信息  // 首先得到注解所在的类 / 方法 的"对象"* Class clazz = Demo.**class**;  Method method = Demo.**class**.getMethod(**"save"**);  *// 通过对象得到Annotation对象* Author clazzAnno = (Author) clazz.getAnnotation(Author.**class**); *// 获得类的annotation对象* Author methodAnno = method.getAnnotation(Author.**class**); *// 获得方法的annotation对象  // 最后通过annotation对象可得到具体的属性值信息* System.***out***.println(clazzAnno.name());  System.***out***.println(clazzAnno.age());  System.***out***.println(clazzAnno.remark());  System.***out***.println(methodAnno.name());  System.***out***.println(methodAnno.age());  System.***out***.println(methodAnno.remark());  }   *// 顺便也在这里运行主类* **public static void** main(String[] args) **throws** NoSuchMethodException {  **new** Demo().save();  } } |

因为Field、Method和Constructor等对象中都有getAnnotation方法，所以获得注解属性值比较容易。

# 2. 单元测试

单元测试（unit testing），是指对软件中的可测试单元进行检查和验证。比如测试Java中写的函数和类能否达到预期的效果、有无出错等。Java中，通常使用JUnit工具进行单元测试，并且各种IDE工具对JUnit支持很好。

可在官网<http://junit.org/junit4/> 了解详细信息，可在github：<https://github.com/junit-team/junit4/releases> 上下载最新的jar包。目前最新的版本是JUnit4，JUnit4还依赖hamcrest-core.jar包。在项目中要一并导入才能使用JUnit。

JUnit的使用非常简单，测试的最小单元是方法。我们不需要写main()方法即可进行测试。只需要在要测试的方法上加上@Test注解即可运行测试。

|  |
| --- |
| **package** com.zhang.test;  **import** org.junit.Test;  **public class** Demo {  @Test  **public void** test() {  System.***out***.println(**"运行测试"**);  } } |

选中该方法右键点击Run Test即可。如果程序运行正常，那么IDE会显示绿色长条表示正确，否则会显示红色长条表示错误。

使用的注意点：

1. 如果选中方法名、类名、包名和工程名分别进行测试，那么测试的是方法、类、包和工程中的所有带@Test的方法。

2. @Test不能测试静态方法、私有方法，也不能测试带有形参的方法。可以在要测试的方法中调用其他的静态方法，或者给方法传递参数再测试。

3. 如果测试方法时，需要准备测试环境或者清理测试环境，则可以用@Before、 @After、@BeforeClass、@AfterClass这四个注解。

@Before、@After 是在每个测试方法测试的之前和之后调用一次，而@BeforeClass、@AfterClass是在所有的测试方法测试之前与测试之后调用一次。

4. 如果一个类需要测试，那么该类应该对应一个测试类，测试类的命名规范：被测试类的类名+Test。被测试的方法对应着一个测试的方法，测试方法的命名规范是：test + 被测试的方法名。但一般不这么规范，因为太麻烦了。可按照自己的需求进行测试，测试完后删除测试代码。