# 1. 注解

## 1.1 注解的使用

实际上我们之前遇到过注解，使用注解时以“@”开头，例如@Override等。先学习使用JDK中三个简单的注解。

（1）@Override注解：写在方法上，用于说明该方法覆盖了父类的方法。如果父类无该方法，则报错，因此也可用于覆盖方法的检测，建议重写方法时加上此注解。

（2）@Deprecated注解：写在方法上，用于说明该方法过时了，不建议再使用了。

（3）@SuppressWarnings注解：可写在方法和语句上，用于抑制Java代码中的警告。Java代码中的警告是会被IDE检测并显示的。@SuppressWarnings用法为：@SuppressWarnings(需要抑制的警告类型)，其中的类型可以为：

unused：抑制“未使用变量”的警告

deprecation：抑制“使用过时方法”的警告

unchecked：抑制“使用未检查的转换”的警告，例如集合没有使用泛型

rawtypes：抑制“方法参数是集合时没有使用泛型”的警告

all：抑制所有警告

上述注解使用示例：

|  |
| --- |
| **import** java.util.ArrayList; **import** java.util.List;  **public class** AnnoDemo {  @Override  **public** String toString() {  **return super**.toString();  }  *// 可抑制多种警告* @SuppressWarnings({**"unused"**, **"unchecked"**})  **public void** useList() {  List list = **new** ArrayList();  }   *// 抑制当方法参数是集合时，没有使用泛型的警告* @SuppressWarnings(**"rawtypes"**)  **public void** listParam(List list) {  }   **public void** lang() {  @SuppressWarnings(**"all"**)  **int** a; *// 这条语句会被抑制警告* **int** b; *// 这条语句不会被抑制警告* } } |

上面的三个注解实际上是给编译器检查使用的，能使代码中提供更多的信息，看上去作用不是那么明显，我们学完注解和以后的Java EE框架后会有更深的认识。

## 1.2 自定义注解

### 1.2.1 基本注解的定义

通过自定义注解，可以在类、字段和方法上添加描述信息。

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。

|  |
| --- |
| **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 { } |

总之，定义注解属性使用的格式为“属性类型 属性名() [default 默认值]”，其中中括号中的默认值是可省略的。需要注意的是，注解属性的类型只能是：基本类型、String、Class、注解、枚举及以上类型的一维数组。其他类型例如Date等不能作为注解属性的类型。

所有的自定义注解类型都是java.lang.annotation.Annotation的子类，自定义注解会继承Annotation注解中的属性（虽然不怎么用）。

### 1.2.2 特殊的属性value

如果注解中只定义了一个名为value的属性，那么使用此注解时，可直接在括号中写属性值，这时设置的就是value属性。例如：

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

如果注解中除了value外还有别的属性，那么使用该注解时就不能省略写属性名称了，除非这些别的属性都是有默认值的。

### 1.2.3 数组属性

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 { } |

如果大括号中什么都没有，则该属性值就是空数组。

### 1.2.4 注解的嵌套

有时，某些注解中的属性类型又是注解，这就是注解的嵌套，例如下面的MyAnno注解中嵌套了Param注解：

|  |
| --- |
| @**interface** Param {  String name();  String value(); }  @**interface** MyAnno {  Param[] params(); *// 属性类型是Param[]* } |

那么如何使用MyAnno进行注解的嵌套呢？按照正常书写即可，params属性值为数组，数组中再写@Param注解和其中属性即可：

|  |
| --- |
| @MyAnno(params = {  @Param(name=**"encoding"**, value = **"UTF-8"**),  @Param(name=**"baseDir"**, value = **"d:/files/"**) }) **public class** AnnoDemo { } |

说明：上述注解的使用方式都会在以后接触到，注意注解属性的类型还可以是枚举等，这里不再举例，下面有应用。

## 1.3 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”，元注解就是用在注解上的注解。

（1）使用@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ElementType枚举类型的数组。ElementType中有下述枚举值供选用：

|  |  |
| --- | --- |
| 枚举 | 标识注解可应用的位置 |
| ElementType.TYPE | 类、接口和枚举 |
| ElementType.FIELD | 字段 |
| ElementType.METHOD | 方法 |
| ElementType.PARAMETER | 参数 |
| ElementType.CONSTRUCTOR | 构造器 |
| ElementType.LOCAL\_VARIABLE | 局部变量 |
| ElementType.ANNOTATION\_TYPE | 注解 |
| ElementType.PACKAGE | 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**; } |

其实，元注解的@Target就使用了ElementType.ANNOTATION\_TYPE这个属性值，指定元注解只能在注解上使用。

（2）使用@Retention注解指定注解的生命周期（使用范围）。此注解只有一个类型为RetentionPolicy枚举的value属性。RetentionPolicy枚举类中有如下枚举可供选择：

|  |  |
| --- | --- |
| SOURCE | 表明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。如果注解不用此值的话，则无法用下面的注解反射实现特定的注解供功能。 |

（3）@Documented注解：如果某注解应用了@Documented元注解，则生成javadoc时，javadoc文档中会显示该注解是如何被类、方法等使用的。

（4）@Inherited注解：指明注解是否被继承。如果应用了某注解应用该元注解，则A类使用了该注解，那么A的子类也会自动继承使用这个注解。

一般，@Documented和@Inherited不常用。

## 1.4 注解的反射

之前定义了注解，但是并没有体会到使用注解能实现什么功能。想要使用注解实现功能，需要使用注解的反射，通过注解的反射读取注解的信息，以此实现功能。可以说，注解的定义只是注解的“形体”，而反射是注解的“灵魂”，注解的意义是通过反射控制的。

需要注意的是，用反射实现注解的功能，需要先将注解的生命周期配置为RUNTIME，使注解在运行时也有效，否则运行时用注解的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。

实际上，和注解反射相关的方法在AnnotationElement接口中，该接口提供了下面的方法：

|  |
| --- |
| （1）<T extends Annotation> T getAnnotation(Class<T> annotationType)：获取指定类型的注解实例的引用。返回的类型T需要继承Annotation类，这确保是注解类型。  （2）Annotation[] getAnnotations()：获取所有的注解，包含继承下来的。  （3）Annotation[] getDeclaredAnnotations()：获取自己直接使用的注解，不包含继承下来的。  （4）boolean isAnnotationPresent(Class<? extends Annotation> annotationType)：看看指定的注解在不在。 |

为什么Method、Class等类的实例对象能调用上述的方法分别获得方法、类等上面的注解对象信息呢？这是因为它们都实现了AnnotationElement接口。

## 1.4 注解的意义

通过学习我们知道注解有两个基本作用，一是给编译器提供源码级别的一些信息；二是可用于对程序的配置。

在开发中，一般使用XML配置程序，优点是避免了硬编码，缺点是开发不置换，配置较复杂；而使用注解配置程序的优点是开发简便快速，缺点是硬编码。因此在开发中应该按照实际选择配置的方式。现在甚至可以用注解进行程序的所有配置，实现项目的零配置文件（即项目中无配置文件）。

我们大多数情况下，只要知道怎么使用注解即可。

# 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。

# 3. IDE引入jar包

## 3.1 Eclipse项目中引入jar包

一般在项目下新建一个lib文件夹用于存放jar包。将jar包复制到此文件夹后，选中jar包，右键“Build Path”，再选择“Add to Build Path”，这样就实现在JavaSE项目中导入jar包了。

## 3.2 IDEA项目中引入jar包

首先，也在JavaSE项目下新建一个lib文件夹。然后点击“File”->“Project Structure...”，在对话框中选择“Modules”板块的“Dependencies”选项卡。点击选项卡右侧的绿色加号，选择“JARs or directories...”，选择刚才项目新建的lib文件夹即可。