JAVA300集大型课程

加入www.sxt.cn 一起学习

讲师：高淇　邮箱：gaoqi110@163.com

注解 annotation

加入www.sxt.cn 一起学习
讲师：高淇　邮箱：gaoqi110@163.com

什么是注解?

- Annotation是从JDK5.0开始引入的新技术。
- Annotation的作用：
  - 不是程序本身，可以对程序作出解释。(这一点，跟注释没什么区别)
  - 可以被其他程序(比如：编译器等)读取。(注解信息处理流程，是注解和注释的重大区别。如果没有注解信息处理流程，则注解毫无意义)
- Annotation的格式：
  - 注解是以“@注释名”在代码中存在的，还可以添加一些参数值，例如：@SuppressWarnings(value="unchecked")。
- Annotation在哪里使用?
  - 可以附加在package, class, method, field等上面，相当于给它们添加了额外的辅助信息，我们可以通过反射机制编程实现对这些元数据的访问。

内置注解

- @Override
  - 定义在java.lang.Override中，此注释只适用于修辞方法，表示一个方法声明打算重写超类中的另一个方法声明。
- @Deprecated
  - 定义在java.lang.Deprecated中，此注释可用于修辞方法、属性、类，表示不鼓励程序员使用这样的元素，通常是因为它很危险或存在更好的选择。

内置注解

- @SuppressWarnings
  - 定义在java.lang.SuppressWarnings中，用来抑制编译时的警告信息。
  - 与前两个注释有所不同，你需要添加一个参数才能正确使用，这些参数值都是已经定义好了的，我们选择性的使用就好了，参数如下：

参数	说明
deprecation	使用了过时的类或方法的警告
unchecked	执行了未检查的转换时的警告，如使用集合时未指定泛型
fallthrough	当在switch语句使用时发生case穿透
path	在类路径、源文件路径等中有不存在路径的警告
serial	当在可序列化的类上缺少serialVersionUID定义时的警告
finally	任何finally子句不能完成时的警告
all	关于以上所有情况的警告

  - @SuppressWarnings("unchecked")
  - @SuppressWarnings(value={"unchecked", "deprecation"})

自定义注解

- 使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口
- 要点：
  - @interface用来声明一个注解
- 格式为：
  - public @interface 注解名 {定义体}
  - 其中的每一个方法实际上是声明了一个配置参数。
  - 方法的名称就是参数的名称
  - 返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）
  - 可以通过default来声明参数的默认值。
  - 如果只有一个参数成员，一般参数名为value

- 注意：

  注解元素必须要有值。我们定义注解元素时，经常使用空字符串、0作为默认值。
  也经常使用负数(比如：-1)表示不存在的含义

元注解

- 元注解的作用就是负责注解其他注解。 Java定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。
- 这些类型和它们所支持的类在java.lang.annotation包中可以找到
  - @Target
  - @Retention
  - @Documented
  - @Inherited

@Target

- 作用：
  - 用于描述注解的使用范围（即:被描述的注解可以用在什么地方）

所修饰范围	取值ElementType
package 包	PACKAGE
类、接口、枚举、Annotation类型	TYPE
类型成员（方法、构造方法、成员变量、枚举值）	CONSTRUCTOR:用于描述构造器 FIELD:用于描述域 METHOD:用于描述方法
方法参数和本地变量	LOCAL_VARIABLE:用于描述局部变量 PARAMETER:用于描述参数

  - @Target(value=ElementType.TYPE)

@Retention

- 作用：
  - 表示需要在什么级别保存该注释信息，用于描述注解的生命周期

取值 RetentionPolicy	作用
SOURCE	在源文件中有效（即源文件保留）
CLASS	在class文件中有效（即class保留）
RUNTIME	在运行时有效（即运行时保留） 为Runtime可以被反射机制读取

使用反射机制读取注解信息

- 如上我们只讲解了注解的定义。我们必须再将注解的读取学会才能轰然一体，彻底搞定注解。

```
try {
        Class clazz = Class.forName("com.bjsxt.test.annotation.SxtStudent");

        //获得类的所有有效注解
        Annotation[] annotations=clazz.getAnnotations();
        for (Annotation a : annotations) {
            System.out.println(a);
        }
        //获得类的指定的注解
        SxtTable st = (SxtTable) clazz.getAnnotation(SxtTable.class);
        System.out.println(st.value());

        //获得类的属性的注解
        Field f = clazz.getDeclaredField("studentName");
        SxtField sxtField = f.getAnnotation(SxtField.class);
        System.out.println(sxtField.columnName()+"--"+sxtField.type()+"--"+sxtField.length());

        //根据获得的表名、字段的信息，拼出DDL语句，然后，使用JDBC执行这个SQL，在数据库中生成相关的表

    } catch (Exception e) {
        e.printStackTrace();
    }
```

注解作业

- 什么是ORM(Object Relationship Mapping)?

```
public class SxtStudent {
    int id;
    String sname;
    int age;
}
```

```
-- Table "tb_student" DDL

CREATE TABLE `tb_student` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `sname` varchar(10) DEFAULT NULL,
  `age` int(3) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

  - 类和表结构对应
  - 属性和字段对应
  - 对象和记录对应

ID	sname	age
1001	高淇	18
1002	裴新	19

- 使用注解完成类和表结构的映射关系
  - 学习了反射机制后，我们可以定义注解处理流程读取这些注解，实现更加复杂的功能。