第二十三章 注解

1. 注解的概述

1. 注解是什么

代码块

```
package com.powernode.annotation01;

public class Test {
    @Override //可以理解为一个标签 @Override标签的作用就是约束该方法必须重写父类的方法
    public String toString() {
        return super.toString();
    }
}
```

2. 有什么用
   a. 可以**标注元素**
   b. 这个标注**不会改变程序的执行逻辑，被编译后会执行特定的操作**
   c. 这个标注或者称为标签，学名叫**注解，可以贴在代码的不同元素上，用于传达特定的意图**
   d. 比如：@Override用于标注该方法必须重写父类的，当编译器看到这个注解的时候
      i. 会检查该方法是否重写父类的
      ii. 如果没有重写就会报错
3. 怎么用

代码块

```
package com.powernode.annotation01;

/**
 * - 自定义注解
 *   1.修饰类的关键字是class,修饰注解的关键字是@interface
 *   2.注解也是一种数据类型，编译后也会生成单独的class文件
 *   3.声明注解的语法:
 *      [修饰符] @interface 注解名称{
 *
 *      }
 */
```

```java
@interface MyAnnotation{

}
@MyAnnotation
public class Teacher {
    @MyAnnotation
    private String name;

    @MyAnnotation
    public Teacher(@MyAnnotation String name) {
        this.name = name;
    }
    @MyAnnotation
    public String getName() {
        return name;
    }
    @MyAnnotation
    public void setName(String name) {
        this.name = name;
    }
}
```

2. 预制注解

- 预制注解：是一种定义好的注解，使用它可以为代码添加额外的语义，从而让编译器更好的理解和处理代码

2.1 @Deprecated

- @Deprecated 不推荐使用|弃用，用于标记某个类，方法，属性或者接口等元素，不推荐使用
- 向开发者传递，尽量不要使用被标记的元素

代码块

```java
package com.powernode.annotation02;
class DBUtil{
    /**
     * 1.语法：@Deprecated(属性 = 值,属性 = 值)
     * 2.@Deprecated  不推荐使用|弃用，用于标记某个类，方法，属性或者接口等元素，不推荐使用
     * 3.since 属性：指定该元素从哪个版本开始被弃用，默认为1.0
     * 4.forRemoval属性：指定该元素是否在将来的版本中被移除，默认为false
     */
    @Deprecated(since = "1.8",forRemoval = true)
    public static void getConnection(){
```

```
11            System.out.println("获得数据库连接");
12        }
13    }
14    public class Test {
15        public static void main(String[] args) {
16            DBUtil.getConnection();
17        }
18    }
```

2.2 @Override

代码块

```
1   package com.powernode.annotation03;
2
3   public class Test01 {
4       /**
5        * 用于标注该方法是被重写的方法
6        */
7       @Override
8       public String toString() {
9           return super.toString();
10      }
11  }
```

2.3 @FunctionalInterface

代码块

```
1   package com.powernode.annotation03;
2
3   /**
4    * 1.函数式接口，使用@FunctionalInterface注解修饰
5    * 2.函数式接口中只有一个抽象方法
6    * 3.函数式接口可以有默认方法，静态方法和私有方法
7    */
8   @FunctionalInterface
9   public interface Flyer {
10      void fly();
11      //函数式接口可以有默认方法，静态方法和私有方法
12      default void swim() {}
13      static void show() {}
14      private void show2() {}
15  }
```

2.4 @SuppressWarnings

代码块

```
package com.powernode.annotation03;

import java.util.ArrayList;
import java.util.List;

public class Test02 {
    public static void main(String[] args) {

        // List<Integer> list = new ArrayList<>();
         /**
          * 1.@SuppressWarnings("rawtypes"),表示当前代码中，忽略所有泛型警告
          * 2.@SuppressWarnings("unchecked"),表示当前代码中，忽略所有未检查的转换警告
          * 3.@SuppressWarnings("all"),表示当前代码中，忽略所有警告
          * 4.@SuppressWarnings("unused"),表示当前代码中，忽略所有未使用的变量警告
          * 5.@SuppressWarnings("deprecation"),表示当前代码中，忽略所有过时的API警告
          * 6.@SuppressWarnings("serial"),表示当前代码中，忽略所有序列化警告
          * ....
          */
         //抑制警告
         @SuppressWarnings("rawtypes")
         List list = new ArrayList();
    }
}
```

3. 自定义注解

3.1 自定义注解的属性和默认值

代码块

```
package com.powernode.annotation04;

/**
 * 1.修饰类的关键字是class,修饰注解的关键字是@interface
 * 2.注解也是一种数据类型，编译后也会生成单独的.class文件
 * 3.声明注解的语法：[修饰符] @interface 注解名称 {}
 */
@interface MyAnnotation01 {
   //注解的属性
    String name();
}
```

```java
@interface MyAnnotation02 {
    //注解的属性
    int num() default 10;
}

public class Test {
    /**
     * - 使用注解时
     *   1.如果注解没有默认值
     *   2.添加注解的时候，必须指定属性值
     */
    @MyAnnotation01(name ="zs")
    String uname ;
    //如果添加了默认值，那么添加注解的时候可以不指定属性值
    @MyAnnotation02
    int age;

}
```

3.2 属性名为value

代码块

```java
package com.powernode.annotation05;

@interface MyAnnotation {
     String value();
}
public class Test {
    //@MyAnnotation(value = "张三")
    //如果属性名为value，那么value可以省略，直接写值
    @MyAnnotation("张三")
    private String name;
}
```

3.3 属性为数组类型

代码块

```java
package com.powernode.annotation06;

import java.util.List;

@interface MyAnnotation {
    // value属性，该属性是一个字符串数组
```

```
    String[] value();
}

public class Test {

    //@MyAnnotation(value = {"张三", "李四"})
    //@MyAnnotation({"张三", "李四"}) // value属性可以省略
    @MyAnnotation("张三") // 如果数组中只有一个元素，可以省略{}
    String name;

    @SuppressWarnings({"unused", "rawtypes"})
    List list;

}
```

3.4 注解可以使用哪些数据类型

代码块

```
package com.powernode.annotation07;
enum MyEnum{
    ONE,TWO,THREE
}
@interface MyInnotation02 {
    String name();
}
public @interface MyInnotation01 {
    /**
     * - 注解可以使用的数据类型
     *    1.基本类型
     *    2.引用类型：String,Class,enum,注解
     *    3.以上类型的数组
     */
    //1.基本类型
    byte b();
    short s();
    int i();
    long l();

    float f();
    double d();

    char c();

    boolean bool();

```

```
28      //2.引用类型
29      String str();
30      Class<?> clazz();
31      MyEnum myEnum();
32      MyInnotation02 myInnotation02();
33
34      //3.以上类型的数组
35      String[] strs();
36      Class<?>[] clazzs();
37      MyEnum[] myEnums();
38      MyInnotation02[] myInnotation02s();
39
40  }
41  class Test{
42      @MyInnotation01(
43              b = 10,s = 20,i = 30,l = 40,f = 50.5f,d = 60.6,c = 'a',bool = true,
44              str = "hello",clazz = String.class,myEnum = MyEnum.ONE,
    myInnotation02 = @MyInnotation02(name = "张三"),
45              strs = {"hello","world"},clazzs =
    {String.class,Integer.class},myEnums = {MyEnum.ONE,MyEnum.TWO},
46              myInnotation02s = {@MyInnotation02(name = "张
    三"),@MyInnotation02(name = "张三")}
47      )
48      public void method(){
49
50      }
51  }
```

4. 元注解

- 元注解：标注注解的注解

代码块

```
1   @Target(ElementType.METHOD)
2   @Retention(RetentionPolicy.SOURCE)
3   public @interface Override {
4   }
```

4.1 @Target

代码块

```
1   package com.powernode.annotation08;
2
```

```java
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * Target元注解：可以标注自定义注解，告诉JVM这个注解可以放在哪个位置
 *      TYPE,
 *      FIELD,
 *      METHOD,
 *      PARAMETER,
 *      CONSTRUCTOR,
 *      LOCAL_VARIABLE,
 *      ANNOTATION_TYPE,
 *      PACKAGE,
 *      TYPE_PARAMETER,
 *      TYPE_USE,
 *      MODULE,
 *      RECORD_COMPONENT
 *
 *      1.ElementType.TYPE：可以放在类上(重点)
 *      2.ElementType.FIELD：可以放在属性上(重点)
 *      3.ElementType.METHOD：可以放在方法上(重点)
 *      4.ElementType.CONSTRUCTOR：可以放在构造方法上(重点)
 *      5.ElementType.PARAMETER：可以放在参数上(重点)
 *      6.ElementType.ANNOTATION_TYPE：可以放在注解上(重点)
 *      7.ElementType.LOCAL_VARIABLE：可以放在局部变量上(重点)
 *
 *      8.ElementType.PACKAGE：可以放在包上
 *      9.ElementType.TYPE_PARAMETER：可以放在类型参数上
 *      10.ElementType.TYPE_USE：可以放在类型使用上
 *      11.ElementType.MODULE：可以放在模块上
 *      12.ElementType.RECORD_COMPONENT：可以放在记录组件上
 *
 *
 */
@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD,ElementType.CONSTRUCTOR})
@interface MyAnnotation {

}

@MyAnnotation
public class Test {
    @MyAnnotation
    String name;
    @MyAnnotation
    public Test(String name) {
        this.name = name;
```

```
    }
    @MyAnnotation
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

4.2 @Retention

- 配置注解的保留策略

代码块

```
package com.powernode.annotation09;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
 * 1.@Retention 配置注解的保留策略
 * 2.RetentionPolicy，保留策略的取值：
 *    1.SOURCE：在源文件中保留，编译时注解丢失
 *    2.CLASS：在class文件中保留，运行时注解丢失
 *    3.RUNTIME：在运行时保留，可通过反射获取注解信息
 *    4.注解默认的保留策略是CLASS
 */
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {

}
public class Test {
    @Override
    public String toString() {
        return super.toString();
    }
}
```

4.3 @Documented@Inherited

代码块

```java
package com.powernode.annotation10;

import java.lang.annotation.*;

@Documented //生成文档时，该注解会保留在文档中
@Inherited //子类会继承父类的注解
@Retention(RetentionPolicy.RUNTIME) //运行时保留
@interface MyAnnotation {
}

@MyAnnotation
class Person{}

class Student extends Person{

}
public class Test {
    public static void main(String[] args) {
        Class<Student> studentClass = Student.class;//获得Student的Class对象
        //获得Student类上的所有注解
        Annotation[] annotations = studentClass.getAnnotations();
        for (Annotation annotation : annotations) {
            //判断子类是否有MyAnnotation注解
            if (annotation instanceof MyAnnotation) {
                System.out.println("子类继承了父类的注解");
            }
        }
    }
}
```

5. 体验使用注解+反射生成动态SQL

代码块

```java
package com.powernode.annotation11;

import java.lang.annotation.*;
@Documented// 使用 @Documented 注解，表示该注解被记录在文档中
@Retention(RetentionPolicy.RUNTIME)// 使用 @Retention 注解，表示该注解在运行时可以被获取
@Target(ElementType.TYPE)// 使用 @Target 注解，表示该注解可以被应用在类上
public @interface Table {
   String value();
}
```

代码块

```
package com.powernode.annotation11;

import java.lang.annotation.*;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)// 表示该注解可以被应用在字段上
public @interface Column {
    String value();
}
```

代码块

```
package com.powernode.annotation11;

@Table("person")
public class Person
{
    @Column("name")
    private String name;

    @Column("sex")
    private String sex;
    @Column("id")
    private int id;

    private int age;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getSex()
    {
        return sex;
    }

```

```
    public void setSex(String sex)
    {
        this.sex = sex;
    }

    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age = age;
    }

}
```

代码块

```
package com.powernode.annotation11;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
// 测试类，主要用于演示通过反射结合自定义注解来动态生成查询SQL语句的功能
public class Test {
    public static void main(String[] args) {
        // 创建一个Person对象，并设置其属性值
        Person p = new Person();
        p.setName("wanglu");
        p.setAge(25);
        p.setId(1001);
        p.setSex("男");


```

```
18          String querySQL = null;
19          try {
20              // 调用query方法尝试生成查询该Person对象对应的SQL语句，捕获可能出现的异常
21              querySQL = query(p);
22          } catch (NoSuchMethodException | SecurityException |
    IllegalAccessException | IllegalArgumentException
23                  | InvocationTargetException e) {
24              e.printStackTrace();
25          }
26          // 输出生成的查询SQL语句，如果生成过程出现异常则输出异常信息
27          System.out.println(querySQL);
28      }
29
30      private static String query(Object p) throws NoSuchMethodException,
    SecurityException, IllegalAccessException,
31              IllegalArgumentException, InvocationTargetException {
32          StringBuilder str = new StringBuilder();
33
34          // 通过反射获取传入对象的Class对象，后续将基于这个Class对象来获取相关注解信息等
35          Class<? extends Object> obj = p.getClass();
36
37          // 判断该对象的类上是否存在 @Table 注解，若不存在则无法生成对应的SQL语句，直接
    返回null
38          boolean isExistsTable = obj.isAnnotationPresent(Table.class);
39          if (!isExistsTable) {
40              return null;
41          }
42
43          // 获取对象所属类上的 @Table 注解实例，并从中获取表名（假设 @Table 注解有一个
    value属性用于指定表名）
44          Table table = (Table) obj.getAnnotation(Table.class);
45          String tableName = table.value();
46
47          // 开始拼装SQL语句的基础部分，这里先写 "select * from [表名] where 1=1 "，后
    续会根据条件拼接更多的查询条件
48          str.append("select * from ").append(tableName).append(" where 1=1 ");
49
50          // 获取该对象所属类的所有成员变量（也就是类中的属性），通过遍历这些成员变量来处理
    每个属性对应的注解及取值情况
51          Field[] fields = obj.getDeclaredFields();
52          for (Field field : fields) {
53              // 判断当前成员变量上是否存在 @Column 注解，如果不存在则跳过该成员变量，不
    将其加入查询条件中
54              Boolean isExistColumn = field.isAnnotationPresent(Column.class);
55              if (!isExistColumn) {
56                  continue;
57              }
```

```
58
59              // 获取当前成员变量上的 @Column 注解实例，并从中获取对应的列名（假设 @Column 注解有一个value属性用于指定列名）
60              Column column = field.getAnnotation(Column.class);
61              String columnName = column.value();
62
63              // 根据成员变量对应的列名来构建其对应的get方法名，遵循JavaBean的命名规范（例如属性名为 "name"，则get方法名为 "getName"）
64              String methodName = "get" + columnName.substring(0, 1).toUpperCase() + columnName.substring(1);
65
66              // 通过反射获取对象所属类中定义的上述构建好的get方法（可能会抛出 NoSuchMethodException 异常，如果方法不存在的话）
67              Method method = obj.getMethod(methodName);
68
69              // 通过反射执行获取到的get方法，传入对象本身作为参数，获取该成员变量的值（可能会抛出 IllegalAccessException、InvocationTargetException 等异常）
70              Object value = method.invoke(p);
71
72              // 过滤掉成员变量中的null值以及值为0的整数类型情况，因为这些情况通常不适合作为查询条件加入到SQL语句中，直接跳过继续处理下一个成员变量
73              if (null == value || (value instanceof Integer && (Integer) value == 0)) {
74                  continue;
75              }
76
77              // 将符合条件的成员变量对应的列名和值拼接成查询条件，添加到SQL语句中，格式为 " and [列名]=[值]"
78              str.append(" and ").append(columnName).append("=").append(value);
79          }
80
81          // 返回最终拼装好的查询SQL语句
82          return str.toString();
83      }
84  }
```