📖 青空的b站课程

注解

注意： 注解跟我们之前讲解的注释完全不是一个概念，不要搞混了。

其实我们在之前就接触到注解了，比如`@Override`表示重写父类方法（当然不加效果也是一样的，此注解在编译时会被自动丢弃）注解本质上也是一个类，只不过它的用法比较特殊。

注解可以被标注在任意地方，包括方法上、类名上、参数上、成员属性上、注解定义上等，就像注释一样，它相当于我们对某样东西的一个标记。而与注释不同的是，注解可以通过反射在运行时获取，注解也可以选择是否保留到运行时。

预设注解

JDK预设了以下注解，作用于代码：

- @Override - 检查（仅仅是检查，不保留到运行时）该方法是否是重写方法。如果发现其父类，或者是引用的接口中并没有该方法时，会报编译错误。
- @Deprecated - 标记过时方法。如果使用该方法，会报编译警告。
- @SuppressWarnings - 指示编译器去忽略注解中声明的警告（仅仅编译器阶段，不保留到运行时）
- @FunctionalInterface - Java 8 开始支持，标识一个匿名函数或函数式接口。
- @SafeVarargs - Java 7 开始支持，忽略任何使用参数为泛型变量的方法或构造函数调用产生的警告。

元注解

元注解是作用于注解上的注解，用于我们编写自定义的注解：

- @Retention - 标识这个注解怎么保存，是只在代码中，还是编入class文件中，或者是在运行时可以通过反射访问。
- @Documented - 标记这些注解是否包含在用户文档中。
- @Target - 标记这个注解应该是哪种 Java 成员。
- @Inherited - 标记这个注解是继承于哪个注解类(默认 注解并没有继承于任何子类)
- @Repeatable - Java 8 开始支持，标识某注解可以在同一个声明上使用多次。

看了这么多预设的注解，你们肯定眼花缭乱了，那我们来看看`@Override`是如何定义的：

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

该注解由`@Target`限定为只能作用于方法上，ElementType是一个枚举类型，用于表示此枚举的作用域，一个注解可以有很多个作用域。`@Retention`表示此注解的保留策略，包括三种策略，在上述中有写到，而这里定义为只在代码中。一般情况下，自定义的注解需要定义1个`@Retention`和1-n个`@Target`。

既然了解了元注解的使用和注解的定义方式，我们就来尝试定义一个自己的注解：

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

这里我们定义一个Test注解，并将其保留到运行时，同时此注解可以作用于方法或是类上：

```java
@Test
public class Main {
    @Test
    public static void main(String[] args) {

    }
}
```

这样，一个最简单的注解就被我们创建了。

注解的使用

我们还可以在注解中定义一些属性，注解的属性也叫做成员变量，注解只有成员变量，没有方法。注解的成员变量在注解的定义中以“无形参的方法”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型：

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    String value();
}
```

默认只有一个属性时，我们可以将其名字设定为value，否则，我们需要在使用时手动指定注解的属性名称，使用value则无需填入：

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    String test();
}
public class Main {
    @Test(test = "")
    public static void main(String[] args) {

    }
}
```

我们也可以使用default关键字来为这些属性指定默认值：

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    String value() default "都看到这里了，给个三连吧！";
}
```

当属性存在默认值时，使用注解的时候可以不用传入属性值。当属性为数组时呢？

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    String[] value();
}
```

当属性为数组，我们在使用注解传参时，如果数组里面只有一个内容，我们可以直接传入一个值，而不是创建一个数组：

```java
@Test("关注点了吗")
public static void main(String[] args) {

}
public class Main {
    @Test({"value1", "value2"})   //多个值时就使用花括号括起来
    public static void main(String[] args) {

    }
}
```

反射获取注解

既然我们的注解可以保留到运行时，那么我们来看看，如何获取我们编写的注解，我们需要用到反射机制：

```java
public static void main(String[] args) {
    Class<Student> clazz = Student.class;
    for (Annotation annotation : clazz.getAnnotations()) {
        System.out.println(annotation.annotationType());   //获取类型
        System.out.println(annotation instanceof Test);   //直接判断是否为Test
        Test test = (Test) annotation;
        System.out.println(test.value());   //获取我们在注解中写入的内容
    }
}
```

通过反射机制，我们可以快速获取到我们标记的注解，同时还能获取到注解中填入的值，那么我们来看看，方法上的标记是不是也可以通过这种方式获取注解：

```java
public static void main(String[] args) throws NoSuchMethodException {
    Class<Student> clazz = Student.class;
    for (Annotation annotation : clazz.getMethod("test").getAnnotations()) {
        System.out.println(annotation.annotationType());    //获取类型
        System.out.println(annotation instanceof Test);    //直接判断是否为Test
        Test test = (Test) annotation;
        System.out.println(test.value());    //获取我们在注解中写入的内容
    }
}
```

无论是方法、类、还是字段，都可以使用`getAnnotations()`方法（还有几个同名的）来快速获取我们标记的注解。

所以说呢，这玩意学来有啥用？丝毫get不到这玩意的用处。其实不是，现阶段作为初学者，还体会不到注解带来的快乐，在接触到Spring和SpringBoot等大型框架后，相信各位就能感受到注解带来的魅力了。