Java注解类型(@Annotation)

Java注解是JDK5时引入的新特性，鉴于目前大部分框架（如Spring）都使用了注解简化代码并提高编码的效率，因此掌握并深入理解注解对于一个Java工程师来说是件很有必要的事情。

1. 理解Java注解

实际上Java注解与普通修饰符(`pubic, static void`等)的使用方式没有多大区别，下面的例子是常见的注解：

```
/**
 * @author zn
 * @date 2020/3/30
 **/
public interface Animal {
    void eat();
}

/**
 * @author zn
 * @date 2020/3/30
 **/
public class Dog implements Animal {

    @Deprecated
    @Override
    public void eat() {

    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
    }
}

```

当然我们已经见过了`@Test`注解，这个注解不表明是一个单元测试，能直接执行。

本例子当中的 @Deprecated 代表本方法已经过时，下个版本可能就废除了 @Override 表示了这个方法是继承或实现的方法，如果方法名写的不对就会报错。

注解可以再类上，里也能

而对于`@Deprecated`和`@SuppressWarnings("uncheck")`，则是Java本身内置的注解，在代码中可以经常见到，但这并不是一件好事，毕竟当方法或是类上面有`@Deprected`注解时，说明该方法或是类都已经过期且不建议使用，`@SuppressWarnings`则表示忽略指定警告，比如`@SuppressWarnings("uncheck")`，这就是注解的最简单的使用方式，下面我们来看看注解定义的基本语法。

2. 基本语法

(1)声明注解与元注解

我们先来看看前面的`Test`注解是如何声明的：

```
//声明Test注解
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test{

}
```

我们使用了`@interface`声明了`Test`注解，并使用`@Target`注解传入`ElementType.METHOD`参数来标明`@Test`只能用于方法上，`@Retention(RetentionPolicy.RUNTIME)`则用来表示该注解生存期是运行时，从代码上看，注解的定义很像接口的定义，确实如此，毕竟在编译后也会生成`Test.class`文件。对于`@Target`和`@Retention`是由Java提供的元注解，所谓的元注解就是标记其他注解的注解，下面分别介绍：

- `@Target`用来约束注解可以应用的地方（如方法、类或字段），其中`ElementType`是枚举类型，其定义如下，也代表可能的取值范围

```
public enum ElementType{
    //标明该注解可以用于类、接口（包括注解类型）或enum声明
    TYPE,

    //标明该注解可以用于字段（域）声明，包括enum实例
    FIELD,

    //标明该注解可以用于方法声明
    METHOD,

    //标明该注解可以用于参数声明
    PARAMETER,

    //标明注解可以用于构造函数声明
    CONSTRUCTOR,

    //标明注解可以用于局部变量声明
    LOCAL_VARIABLE,

    //标明注解可用于注解声明（应用于另一个注解上）
    ANNOTATION_TYPE,

    //标明注解可以用于包声明
    PACKAGE,

    /**
    *标明注解可以用于类型参数声明（1.8新加入）
    *@since 1.8
    */
    TYPE_PARAMETER,
```

```
31
32      /**
33      *类型使用声明（1.8新加入）
34      *@since1.8
35      */
36      TYPE_USE
37  }
```

注意，当注解未指定`Target`值时，则此注解可以用于任何元素之上，多个值使用`{}`包含并用逗号隔开，如下：

```
1  @Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE,
   PARAMETER, TYPE})
```

- ```
  1  @Retention
  ```

  用来约束注解的生命周期，分别有三个值，源码级别

  ```
  1  source
  ```

  ，类文件级别

  ```
  1  class
  ```

  或者运行时级别

  ```
  1  runtime
  ```

  ，其含义如下：

  - `SOURCE`:注解将被编译器丢弃（该类型的注解信息只会保留在源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的`.class`文件里）
  - `CLASS`：注解在`.class`文件中可用，但会被VM丢弃（该类型的注解信息会保留在源码里和`.class`文件里，在执行的时候，不会加载到虚拟机中），请注意，当注解未定义`Retention`值时，默认值是`CLASS`，如Java内置注解，`@Override,@Deprecated,@SuppressWarning`等
  - `RUNTIME`：注解信息将在运行期（JVM）保留，因此可以通过反射机制读取注解的信息（源码、`.class`文件和执行的时候都有注解信息），如`SpringMVC`中的`@Controller, @Autowired, @RequestMapping`等。

(2) 注解元素及其数据类型

通过上述对`@Test`注解的定义，我们了解了注解定义的过程，由于`@Test`内部没有定义其他元素，所以`@Test`也称为**标记注解（marker annotation）**，但在自定义注解中，一般都会包含一些元素以表示某些值，方便处理器使用，这点在下面的例子将会看到：
  ```

```java
@Target(ElementType.TYPE)//只能应用于类上
@Retention(RetentionRolicy.RUNTIME)//保存到运行时
public @interface DBTable{
    String name() default "";
}
```

上述定义一个名为`DBTable`的注解，该注解主要用于数据库表域Bean类的映射（稍后会有完整案例分析），与前面`Test`注解不同的是，我们声明一个`String`类型的`name`元素，其默认值为空字符串，但是必须注意到对应任何元素的声明应采用方法的声明方式，同时可以选择使用`default`提供默认值，`@DBTable`使用方式如下：

```java
@DBTable(name = "MEMBER")
public class Member{
    //...
}
```

关于注解支持的元素数据类型出来上述的`String`，还支持如下数据类型：

- 所有基本类型（`int, float, boolean, byte, double, char, long, short`）
- `String`
- `Class`
- `enum`
- `Annotation`
- 上述类型的数组

倘若使用了其他数据类型，编译器将会丢出一个编译错误，注意，声明注解元素时可以使用基本类型，但不允许使用任何包装类型，同时还应该注意到注解也可以作为元素的类型，也就是嵌套注解，下面的代码演示了上述类型的使用过程：

```java
package com.guirunxiang.annotationdemo;

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

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Reference{
    boolean next() default false;
}

public @interface AnnotationElementDemo {
    //枚举类型
    enum Status {FIXED, NORMAL};

    //声明枚举
    Status status() default Status.FIXED;

    //布尔类型
    boolean showSupport() default false;
```

```
23
24      //String类型
25      String name() default "";
26
27      //class类型
28      Class<?> testCase() default Void.class;
29
30      //注解嵌套
31      Reference reference() default @Reference(next = true);
32
33      //数组类型
34      long[] value();
35  }
```

(3) 编译器对默认值的限制

编译器对元素的默认值有些过分挑剔。首先，元素不能有不确定的值。也就是说，元素必须要么具有默认值，要么在使用注解时提供元素的值。其次，对于非基本类型的元素，无论在源代码中声明还是在注解接口中定义默认值，都不能以`null`为值。

(4)注解不支持继承

注解是不支持继承的，因此不能使用关键字`extends`来继承某个`@interface`，但注解在编译后，编译器会自动继承`java.lang.annotation.Annotation`接口，这里我们反编译前面定义的`DBTable`注解：

```
1  package com.guirunxiang.annotationdemo;
2
3  import java.lang.annotation.Annotation;
4  //反编译后的代码
5  public interface DBTable extends Annotation{
6      public abstract String name();
7  }
```

虽然反编译后发现`DBTable`注解继承了`Annotation`接口，但定义注解时依然无法使用`extends`关键字继承`@interface`。

(5) 快捷方式

所谓的快捷方式就是注解中定义了名为`value`的元素，并且在使用该注解时，如果该元素是唯一需要复制的元素，那么此时无需使用`key = value`语法，而只需在括号内给出`value`元素所需的值即可。这可以应用于任何合法类型的元素，但这限制了元素名必须为`value`，简单案例如下：

```
1  package com.guirunxiang.annotationdemo;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  //定义注解
9  @Target(ElementType.FIELD)
```

```java
@Retention(RetentionPolicy.RUNTIME)
@interface IntegerValue{
    int value() default 0;
    String name() default "";
}

//使用注解
public class QuicklyWay {

    //当只想给value赋值时，可以使用一下快捷方式
    @IntegerValue(20)
    public int age;

    //当name也需要赋值是必须采用key = value的方式赋值
    @IntegerValue(value = 1000, name = "MONEY")
    public int money;
}
```

3. Java内置注解与其它元注解

接着看看Java提供的内置注解，主要有3个，如下：

- `@Override`：用于标明此方法覆盖率弗雷德方法，源码如下：

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

}
```

- `Deprecated`:用于标记已经过时的方法或类，源码如下，关于`@Documented`稍后分析：

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated{

}
```

- `@SuppressWarnnings`:用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告，其实现源码如下：

```java
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

其内部有一个`String`数组，主要接收值如下：

参数名称	说明
`"deprecation"`	使用了不赞成使用的类或方法时的警告
`"unchecked"`	执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型
`"fallthrough"`	当 `switch` 程序块直接通往下一种情况而没有 `break` 时的警告
`"path"`	在类路径、源文件路径等中有不存在的路径时的警告
`"serial"`	当在可序列化的类上缺少 `serialVersionUID` 定义时的警告
`"finally"`	任何 `finally` 子句不能正常完成时的警告
`"all"`	关于以上所有情况的警告

这三个注解比较简单，看个简单的案例：

```
//注明该类已过时，不建议使用
@Deprecated
Class A{
    public void A(){ }

    //注明该方法已过时，不建议使用
    @Deprecated
    public void B(){ }
}

class B extends A{
    @Override //标明覆盖父类A的A方法
    public void A(){
        super.A();
    }

    //去掉检测警告
    @SuppressWarnings({"uncheck", "deprecation"})
    public void C(){ }

    //去掉检测警告
    @SuppressWarnings("uncheck")
    public void D(){ }
}
```

前面我们分析了两种元注解， `@Target` 和`@Retention`， 除了这两种元注解，Java还提供了另外两种元注解，`@Documented`和`@Inherited`，下面分别介绍：

- `@Documented` 被修饰的注解会生成到`javadoc`中

```
//使用@Documented
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentA {
```

```java
}

//没有使用@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentB {
}

//使用注解
@DocumentA
@DocumentB
public class DocumentDemo {
    public void A(){
    }
}
```

使用`javadoc`命令生成文档:

```
javadoc DocumentDemo.java DocumentA.java DocumentB.java
```

如下， 可以发现使用`@Documented`元注解定义的注解`@DocumentA`将会生成到javadoc中，而`@DocumentB`则没有在doc文档中出现。这就是元注解`@Documented`的作用。

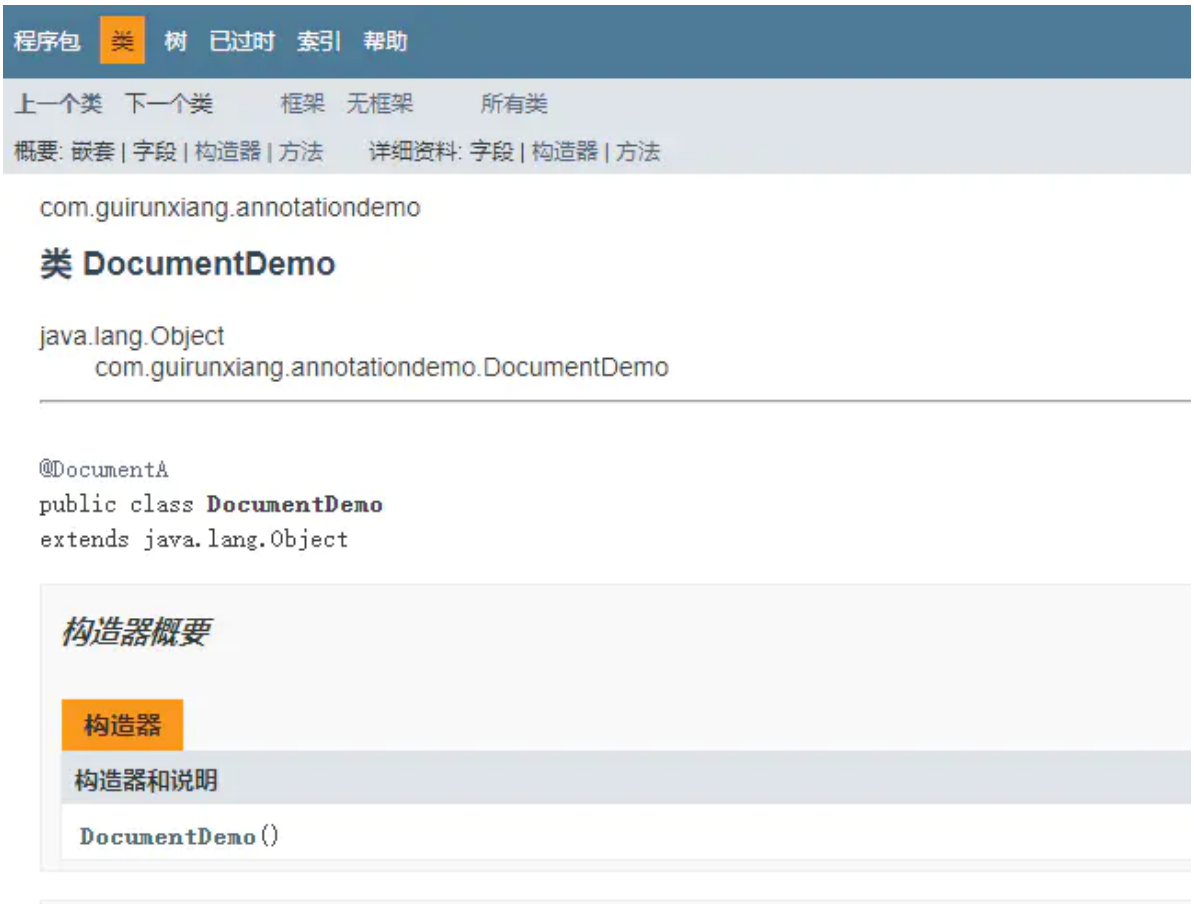

image-20200323120949046.png

- `@Inherited`可以让注解被继承，但这并不是真的继承，只是通过使用`@Inherited`， 可以让子类`Class`对象使用`getAnnotations()`获取父类被`@Inherited`修饰的注解， 如下:

```java
@Inherited
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentA {
}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentB {
}

@DocumentA
class A{ }

class B extends A{ }

@DocumentB
class C{ }

class D extends C{ }

//测试
public class DocumentDemo {

    public static void main(String... args){
        A instanceA=new B();
        System.out.println("已使用的@Inherited注解:"+Arrays.toString(instanceA.getClass().getAnnotations()));

        C instanceC = new D();

        System.out.println("没有使用的@Inherited注解:"+Arrays.toString(instanceC.getClass().getAnnotations()));
    }

    /**
     * 运行结果:
     已使用的@Inherited注解:[@com.zejian.annotationdemo.DocumentA()]
     没有使用的@Inherited注解:[]
     */
}
```

运行结果如下: