技术专家

技术都是解决问题的。

昵称：技术专家
园龄：1年4个月
粉丝：11
关注：0
+加关注

java注解的自定义和使用

小伙伴们。今天我们来说说注解、标志@ 。针对java不同版本来说，注解的出现是在jdk1.5 但是在jdk1.5版本使用注解必须继续类的方法的重写，不能用于实现的接口中的方法实现，在jdk1.6环境下对于继续和实现都是用。

jdk1.5版本内置了三种标准的注解：

@Override，表示当前的方法定义将覆盖超类中的方法。

@Deprecated，使用了注解为它的元素编译器将发出警告，因为注解@Deprecated是不赞成使用的代码，被弃用的代码。

@SuppressWarnings,关闭不当编辑器警告信息。

Java还提供了4中注解，专门负责新注解的创建：

@Target：

表示该注解可以用于什么地方，可能的ElementType参数有：
CONSTRUCTOR：构造器的声明
FIELD：域声明（包括enum实例）
LOCAL_VARIABLE：局部变量声明
METHOD：方法声明
PACKAGE：包声明
PARAMETER：参数声明
TYPE：类、接口（包括注解类型）或enum声明

@Retention

表示需要在什么级别保存该注解信息。可选的RetentionPolicy参数包括：
SOURCE：注解将被编译器丢弃
CLASS：注解在class文件中可用，但会被VM丢弃
RUNTIME：VM将在运行期间保留注解，因此可以通过反射机制读取注解的信息

@Document

将注解包含在Javadoc中

@Inherited

允许子类继承父类中的注解

下面我们自己来新建一个注解。在我们开发中。经常会使用自己设置的注解

首先我们创建一个注解类

```
package com.java.api;
```

最新评论

1. Re:java中的多线程高并发与负载均衡的用途
说了等于没说

--一片番薯

2. Re:浅谈Java数据结构和算法

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

/**定义注解
 * @Target:

表示该注解可以用于什么地方，可能的ElementType参数有:
CONSTRUCTOR:构造器的声明
FIELD:域声明（包括enum实例）
LOCAL_VARIABLE:局部变量声明
METHOD:方法声明
PACKAGE:包声明
PARAMETER:参数声明
TYPE:类、接口（包括注解类型）或enum声明
@Retention

表示需要在什么级别保存该注解信息。可选的RetentionPolicy参数包括:
SOURCE:注解将被编译器丢弃
CLASS:注解在class文件中可用，但会被VM丢弃
RUNTIME:VM将在运行期间保留注解，因此可以通过反射机制读取注解的信息
 *
 * */
public class UseCase{
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCases{
    public String id();
    public String description() default "no description";
}
}
```

然后我那使用注解

```
package com.java.api;

import com.java.api.UseCase.UseCases;

/**
 * 使用注解:
 *
 * */
public class PasswordUtils {
    @UseCases(id="47",description="Passwords must contain at least one numeric")
     public boolean validatePassword(String password) {
         return (password.matches("\\w*\\d\\w*"));
     }

     @UseCases(id ="48")
     public String encryptPassword(String password) {
         return new StringBuilder(password).reverse().toString();
     }
}
```

最后我们来测试我们写的注解：

```
package com.java.api;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import com.java.api.UseCase.UseCases;

/**
 *解析注解:
 *
 * */
```

```
public class UserCaseTest {
public static void main(String[] args) {
     List<Integer> useCases = new ArrayList<Integer>();
     Collections.addAll(useCases, 47, 48, 49, 50);
     trackUseCases(useCases, PasswordUtils.class);
}
public static void trackUseCases(List<Integer> useCases, Class<?> cl) {
    for (Method m : cl.getDeclaredMethods()) {
        //获得注解的对象
        UseCases uc = m.getAnnotation(UseCases.class);
        if (uc != null) {
            System.out.println("Found Use Case:" + uc.id() + " "
                    + uc.description());
            useCases.remove(new Integer(uc.id()));
        }
    }
    for (int i : useCases) {
        System.out.println("Warning: Missing use case-" + i);
    }
}
}
```

总结：

java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。包含在 java.lang.annotation 包中

java自定义注解和运行时靠反射获取注解。

转

1、Annotation的工作原理：
JDK5.0中提供了注解的功能，允许开发者定义和使用自己的注解类型。该功能由一个定义注解类型的语法和描述一个注解声明的语法，读取注解的API，一个使用注解修饰的class文件和一个注解处理工具组成。

Annotation并不直接影响代码的语义，但是他可以被看做是程序的工具或者类库。它会反过来对正在运行的程序语义有所影响。

Annotation可以冲源文件、class文件或者在运行时通过反射机制多种方式被读取。

2、@Override注解：

```
java.lang
注释类型 Override
@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override
```

表示一个方法声明打算重写超类中的另一个方法声明。如果方法利用此注释类型进行注解但没有重写超类方法，则编译器会生成一条错误消息。

@Override注解表示子类要重写父类的对应方法。

Override是一个Marker annotation，用于标识的Annotation，Annotation名称本身表示了要给工具程序的信息。

下面是一个使用@Override注解的例子：

```
class A {
    private String id;
    A(String id){
        this.id = id;
    }
    @Override
    public String toString() {
        return id;
    }
}
```

3、@Deprecated注解：

```
java.lang
注释类型 Deprecated
@Documented
@Retention(value=RUNTIME)
public @interface Deprecated
```

用 @Deprecated 注释的程序元素，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。在使用不被赞成的程序元素或在不被赞成的代码中执行重写时，编译器会发出警告。

@Deprecated注解表示方法是不被建议使用的。

Deprecated是一个Marker annotation。

下面是一个使用@Deprecated注解的例子：

```
class A {
    private String id;
    A(String id){
        this.id = id;
    }
    @Deprecated
    public void execute(){
        System.out.println(id);
    }
    public static void main(String[] args) {
        A a = new A("a123");
        a.execute();
    }
}
```

4、@SuppressWarnings注解：

java.lang

注释类型 SuppressWarnings

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

指示应该在注释元素（以及包含在该注释元素中的所有程序元素）中取消显示指定的编译器警告。注意，在给定元素中取消显示的警告集是所有包含元素中取消显示的警告的超集。例如，如果注释一个类来取消显示某个警告，同时注释一个方法来取消显示另一个警告，那么将在此方法中同时取消显示这两个警告。

根据风格不同，程序员应该始终在最里层的嵌套元素上使用此注释，在那里使用才有效。如果要在特定的方法中取消显示某个警告，则应该注释该方法而不是注释它的类。

@SuppressWarnings注解表示抑制警告。

下面是一个使用@SuppressWarnings注解的例子：

```
@SuppressWarnings("unchecked")
public static void main(String[] args) {
    List list = new ArrayList();
    list.add("abc");
```

```
}
```

5、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，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。

自定义最简单的注解：

```
public @interface MyAnnotation {

}
```

使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation
    public void execute(){
        System.out.println("method");
    }
}
```

5.1、添加变量：

```
public @interface MyAnnotation {

    String value1();
}
```

使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation(value1="abc")
    public void execute(){
        System.out.println("method");
    }
}
```

当注解中使用的属性名为value时，对其赋值时可以不指定属性的名称而直接写上属性值接口；除了value意外的变量名都需要使用name=value的方式赋值。

5.2、添加默认值：

```
public @interface MyAnnotation {
```

```
    String value1() default "abc";
}
```

5.3、多变量使用枚举：

```
public @interface MyAnnotation {

    String value1() default "abc";
    MyEnum value2() default MyEnum.Sunny;
}
enum MyEnum{
    Sunny,Rainy
}
```

使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation(value1="a", value2=MyEnum.Sunny)
    public void execute(){
        System.out.println("method");
    }
}
```

5.4、数组变量：

```
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```

使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation(value1={"a","b"})
    public void execute(){
        System.out.println("method");
    }
}
```

6、设置注解的作用范围：

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Retention
```

指示注释类型的注释要保留多久。如果注释类型声明中不存在 Retention 注释，则保留策略默认为 RetentionPolicy.CLASS。

只有元注释类型直接用于注释时，Target 元注释才有效。如果元注释类型用作另一种注释类型的成员，则无效。

```
public enum RetentionPolicy
extends Enum<RetentionPolicy>
```

注释保留策略。此枚举类型的常量描述保留注释的不同策略。它们与 Retention 元注释类型一起使用，以指定保留多长的注释。

CLASS
编译器将把注释记录在类文件中，但在运行时 VM 不需要保留注释。
RUNTIME
编译器将把注释记录在类文件中，在运行时 VM 将保留注释，因此可以反射性地读取。
SOURCE
编译器要丢弃的注释。
@Retention注解可以在定义注解时为编译程序提供注解的保留策略。

属于CLASS保留策略的注解有@SuppressWarnings，该注解信息不会存储于.class文件。

6.1、在自定义注解中的使用例子：

```
@Retention(RetentionPolicy.CLASS)
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```

7、使用反射读取RUNTIME保留策略的Annotation信息的例子：
java.lang.reflect
接口 AnnotatedElement
所有已知实现类：
AccessibleObject, Class, Constructor, Field, Method, Package
表示目前正在此 VM 中运行的程序的一个已注释元素。该接口允许反射性地读取注释。由此接口中的方法返回的所有注释都是不可变并且可序列化的。调用者可以修改已赋值数组枚举成员的访问器返回的数组；这不会对其他调用者返回的数组产生任何影响。

如果此接口中的方法返回的注释（直接或间接地）包含一个已赋值的 Class 成员，该成员引用了一个在此 VM 中不可访问的类，则试图通过在返回的注释上调用相关的类返回的方法来读取该类，将导致一个 TypeNotPresentException。

isAnnotationPresent

```
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)
```

如果指定类型的注释存在于此元素上，则返回 true，否则返回 false。此方法主要是为了便于访问标记注释而设计的。

参数：

`annotationClass` - 对应于注释类型的 `Class` 对象

返回：

如果指定注释类型的注释存在于此对象上，则返回 `true`，否则返回 `false`

抛出：

`NullPointerException` - 如果给定的注释类为 `null`

从以下版本开始：

`1.5`

`getAnnotation`

```
<T extends Annotation> T getAnnotation(Class<T> annotationClass)
```

如果存在该元素的指定类型的注释，则返回这些注释，否则返回 `null`。

参数：

`annotationClass` - 对应于注释类型的 `Class` 对象

返回：

如果该元素的指定注释类型的注释存在于此对象上，则返回这些注释，否则返回 `null`

抛出：

`NullPointerException` - 如果给定的注释类为 `null`

从以下版本开始：

`1.5`

`getAnnotations`

```
Annotation[] getAnnotations()
```

返回此元素上存在的所有注释。（如果此元素没有注释，则返回长度为零的数组。）该方法的调用者可以随意修改返回的数组；这不

会对其他调用者返回的数组产生任何影响。

返回：

此元素上存在的所有注释

从以下版本开始：

1.5

getDeclaredAnnotations
Annotation[] getDeclaredAnnotations()
返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注释。（如果没有注释直接存在于此元素上，则返回长度为零的一个数组。）该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响。

返回：

直接存在于此元素上的所有注释

从以下版本开始：

1.5

下面是使用反射读取RUNTIME保留策略的Annotation信息的例子：

自定义注解：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```
使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation(value1={"a","b"})
    @Deprecated
```

```
    public void execute(){
        System.out.println("method");
    }
}
```

读取注解中的信息：

```
public static void main(String[] args) throws SecurityException, NoSuchMethodException,
IllegalArgumentException, IllegalAccessException, InvocationTargetException {
    AnnotationTest2 annotationTest2 = new AnnotationTest2();
    //获取AnnotationTest2的Class实例
    Class<AnnotationTest2> c = AnnotationTest2.class;
    //获取需要处理的方法Method实例
    Method method = c.getMethod("execute", new Class[]{});
    //判断该方法是否包含MyAnnotation注解
    if(method.isAnnotationPresent(MyAnnotation.class)){
        //获取该方法的MyAnnotation注解实例
        MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);
        //执行该方法
        method.invoke(annotationTest2, new Object[]{});
        //获取myAnnotation
        String[] value1 = myAnnotation.value1();
        System.out.println(value1[0]);
    }
    //获取方法上的所有注解
    Annotation[] annotations = method.getAnnotations();
    for(Annotation annotation : annotations){
        System.out.println(annotation);
    }
}
```

8、限定注解的使用：

限定注解使用`@Target`。

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Target
```

指示注释类型所适用的程序元素的种类。如果注释类型声明中不存在 `Target` 元注释，则声明的类型可以用在任一程序元素上。如果存在这样的元注释，则编译器强制实施指定的使用限制。 例如，此元注释指示该声明类型是其自身，即元注释类型。它只能用在注释类型声明上：

```
@Target(ElementType.ANNOTATION_TYPE)
    public @interface MetaAnnotationType {
        ...
    }
```

此元注释指示该声明类型只可作为复杂注释类型声明中的成员类型使用。它不能直接用于注释：

```
@Target({})
    public @interface MemberType {
        ...
    }
```

这是一个编译时错误，它表明一个 `ElementType` 常量在 `Target` 注释中出现了不只一次。例如，以下元注释是非法的：

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.FIELD})
    public @interface Bogus {
        ...
    }
```

```
public enum ElementType
extends Enum<ElementType>
```

程序元素类型。此枚举类型的常量提供了 `Java` 程序中声明的元素的简单分类。

这些常量与 `Target` 元注释类型一起使用，以指定在什么情况下使用注释类型是合法的。

`ANNOTATION_TYPE`
注释类型声明
`CONSTRUCTOR`
构造方法声明
`FIELD`
字段声明（包括枚举常量）
`LOCAL_VARIABLE`
局部变量声明
`METHOD`
方法声明
`PACKAGE`
包声明
`PARAMETER`
参数声明
`TYPE`
类、接口（包括注释类型）或枚举声明

注解的使用限定的例子：

```
@Target(ElementType.METHOD)
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```

9、在帮助文档中加入注解：

要想在制作JavaDoc文件的同时将注解信息加入到API文件中，可以使用java.lang.annotation.Documented。

在自定义注解中声明构建注解文档：

```
@Documented
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```

使用自定义注解：

```
public class AnnotationTest2 {

    @MyAnnotation(value1={"a","b"})
    public void execute(){
        System.out.println("method");
    }
}
```

10、在注解中使用继承：

默认情况下注解并不会被继承到子类中，可以在自定义注解时加上java.lang.annotation.Inherited注解声明使用继承。

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Inherited
```

指示注释类型被自动继承。如果在注释类型声明中存在 Inherited 元注释，并且用户在某一类声明中查询该注释类型，同时该类声明中没有此类型的注释，则将在该类的超类中自动查询该注释类型。此过程会重复进行，直到找到此类型的注释或到达了该类层次结构的顶层 (Object) 为止。如果没有超类具有该类型的注释，则查询将指示当前类没有这样的注释。

注意，如果使用注释类型注释类以外的任何事物，此元注释类型都是无效的。还要注意，此元注释仅促成从超类继承注释；对已实现接口的注释无效。

技术专家
关注 - 0
粉丝 - 11
+加关注

1 0

« 上一篇：测试工具类汇总
» 下一篇：浅谈Java数据结构和算法

posted @ 2017-07-26 15:14 技术专家 阅读(11130) 评论(0) 编辑 收藏