java自定义注解

1.Java注解（Annotation）简介

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***，注解相关类都包含在java.lang.annotation包中。

（ps:元信息：在编程和代码开发中，**元信息（Metadata）通常指的是描述数据、代码或其他资源属性的信息。**这些属性可以包括各种特征、属性、分类、关系等，有助于更好地理解和使用这些数据或代码。）

- 用@interface关键字来声明注解，注解也会生成.class文件
- 注解可以有成员变量，在使用注解时可以给注解的成员变量赋值，可以使用default来定义默认值，可以这样声明：成员类型 成员名();
- 成员类型可以为基础数据类型、String、Class、enum、Annotation，以及相应的数组
- 如果注解只用一个成员变量，成员名通常用value
- 使用注解时必须指定参数值，如果使用时只有一个参数成员，且名称为value，可以直接写"参数值"
- **注解是接口类，都继承自Annotation接口类**

```java
//先来个例子
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE, ElementType.CONSTRUCTOR, ElementType.PARAMETER, ElementType.TYPE_USE})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(List.class)//允许在同一个位置多次使用相同的注解。这里，@Min注解被定义为可以通过@Min.List来多次使用。
@Documented
@Constraint(
 //@Constraint 这是Jakarta Bean Validation的一部分，它表示这是一个验证注解。但在这个例子中，validatedBy属性是空的，这意味着你需要提供一个或多个验证器类来实际执行验证。（这里写相应验证器的class文件）
    validatedBy = {}
)
public @interface Min {
    String message() default "{jakarta.validation.constraints.Min.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    long value();

    @Target({ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE, ElementType.CONSTRUCTOR, ElementType.PARAMETER, ElementType.TYPE_USE})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    public @interface List {
        Min[] value();
    }
    //内部注解 @Min.List
```

```
    //这是一个容器注解，允许你在同一个位置多次使用@Min注解。它有一个value属性，这是一个@Min注解的数组.
}
```

```
//下面这是文心一言给的例子
public class Example {
    @Min.List({
        @Min(value = 10, message = "Value must be at least 10"),
        @Min(value = 20, message = "Value must be at least 20") // 注意：这里可能不符合逻辑，只是示例
    })
    private Integer myField;

    // ... getters, setters, etc.
    /*
    注意：在上面的示例中，@Min.List注解在逻辑上可能并不合理，因为通常一个字段只有一个最小值。但是，从技术的角度来看，这样的　　用法是允许的。如果你真的需要这样的功能，你可能需要自定义一个更复杂的验证逻辑，并在验证器类中实现它。
    */
}
```

关于@Constraint自定义参数校验注解

（写springboot项目的时候用这个）

[@Constraint自定义参数校验注解_java_LBL_lin-华为开发者联盟HarmonyOS专区 (csdn.net)]

2.Java注解分类

- JDK基本注解
- JDK元注解
- 自定义注解

1）JDK基本注解

```
@Override
定义在java.lang.Override
方法重写

@Deprecated
定义在java.lang.Deprecated
标识内容不建议被使用，但可以使用

@SuppressWarnings(value = "unchecked")
定义在java.lang.SuppressWarnings
抑制编译时的警告信息
```

2) JDK 元注解

ps：在Java中，**元注解主要用于修饰其他注解的定义**，从而定义该注解的类型、使用范围、生命周期等信息。

```
@Retention：定义注解的保留策略（注解的生命周期）
@Retention(RetentionPolicy.SOURCE)
//SOURCE 源码级注解。注解信息只会保留在 java 源码中，源码在编译后注解信息被丢弃，不会保留在 class 文件

@Retention(RetentionPolicy.CLASS)
//默认的保留策略，CLASS 编译时注解。注解信息会保留在 java 源码以及 class 文件中。当运行 java 程序时，JVM 会丢弃该注解信息，不会保留在 JVM 中

@Retention(RetentionPolicy.RUNTIME)
//RUNTIME 运行时注解，当运行 java 程序时，JVM 也会保留该注解信息，可以通过反射获取该注解信息。
//(一般自定义注解都用RUNTIME)

@Target：指定被修饰的Annotation可以放置的位置(被修饰的目标)
（如果没加注解没加@Target注解，默认所有位置都可以）
@Target(ElementType.TYPE)                   //接口、类
@Target(ElementType.FIELD)                  //属性
@Target(ElementType.METHOD)                 //方法
@Target(ElementType.PARAMETER)              //方法参数
@Target(ElementType.CONSTRUCTOR)            //构造函数
@Target(ElementType.LOCAL_VARIABLE)         //局部变量
@Target(ElementType.ANNOTATION_TYPE)        //注解
@Target(ElementType.PACKAGE)                //包
注：可以指定多个位置，例如：
@Target({ElementType.METHOD, ElementType.TYPE})，也就是此注解可以在方法和类上面使用

@Inherited：指定被修饰的Annotation将具有继承性
    JDK中@Inherited的说明文档很清楚的阐述了继承性：
当用户在一个程序元素类上，使用AnnotatedElement的相关注解查询方法，查询元注解
@Inherited修饰的其他注解类型A时，如果这个类本身并没有被注解A修饰，那么会自动查询这个类的父类是否被注解A修饰。查询过程会沿着类继承链一直向上查找，直到注解A被找到，或者到达继承链顶层（Object）。
如果元注解Inherited修饰的其他注解，修饰了除类之外的其他程序元素，那么继承性将会失效。

@Documented：指定被修饰的该Annotation可以被javadoc工具提取成文档.（ps：javadoc：Java API 文档生成器）
```

3）自定义注解

注解分类

根据Annotation是否包含成员变量,可以把Annotation分为两类

- 标记Annotation:

  没有成员变量的Annotation; 这种Annotation仅利用自身的存在与否来提供信息

  例如：@Override 检查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。

- 元数据Annotation:

  包含成员变量的Annotation; 它们可以接受(和提供)更多的元数据;

  例如：@SuppressWarnings

3.注解的实现与使用

RUNTIME级别（用反射来获取注释信息并处理）

java注解的功能实现基本是通过定义属性实现的（真正实现功能有相关的处理类，处置这些属性，我们先来定义属性）。注解处理器类库(java.lang.reflect.AnnotatedElement)：

AnnotatedElement 接口是所有程序元素（Class、Method和Constructor等）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象（相应的程序元素对象就可以）之后，程序就可以调用该对象的如下四个个方法来访问Annotation信息：

注解处理的一个基础：

```
方法1：<T extends Annotation> T getAnnotation(Class<T> annotationClass)：返回改程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

方法2：Annotation[] getAnnotations()：返回该程序元素上存在的所有注解。

方法3：boolean is AnnotationPresent(Class<?extends Annotation> annotationClass)：判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false.

方法4：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。
```

例子如下：

```
package com.jiang.AnnotationPackage;

import java.lang.annotation.*;
/**
 * 声明一个自定义注解
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyTestAnnotation2 {
    int result() default 50;
}

```

```

```

```java
package com.jiang.AnnotationPackage;

import java.lang.reflect.Method;

/**
 * 自定义注解在方法上的使用
 */
public class MyTestAnnotationDemo2 {

    /**
     * @param number 猜数的大小
     */
    @MyTestAnnotation2(result = 85)
    public static void guess(int number){
        System.out.println(processGuess(number));
    }

    private static String processGuess(int number){
        try {
            Method guessnumber = MyTestAnnotationDemo2.class.getDeclaredMethod("guess",int.class);
            boolean annotationPresent = guessnumber.isAnnotationPresent(MyTestAnnotation2.class);
            if(annotationPresent){
                MyTestAnnotation2 annotation2 = guessnumber.getAnnotation(MyTestAnnotation2.class);
                if(number>annotation2.result()){
                    return "猜的数字大于指定数字";
                }else if (number==annotation2.result()){
                    return "猜的数字等于指定数字";
                }else{
                    return "猜的数字小于指定数字";
                }
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
        return "猜测程序有误";
    }

    public static void main(String[] args) {
        guess(85);
        //guess(84);
        //guess(86);
    }
}

```

关于@Valid注解的实现

简单来说就是在给一个接口发送请求时，在方法执行前，会解析当前调用方法的参数，之后会获得方法参数前的所有注解，再去遍历这些注解来判断是否有@Valid注解，如果有则会继续往下验证。

下面这个是一个讲@Validated和@Valid注解区别的视频（里面结合代码讲了一下@Vaild注解的实现）

别再乱用了，这才是Spring @Validated 和 @Valid 的真正区别！！！*哔哩哔哩*bilibili

下面的两个级别的注解（ps:没找到讲解视频，相关博客很少而且也看太不明白。。。）

CLASS级别（通过注解处理器（Annotation Processors））

处理 `@Retention(RetentionPolicy.CLASS)` 的注解通常涉及编译时处理，而不是运行时处理。这是因为这些注解在类加载到JVM时不会被保留，所以无法通过反射API在运行时读取它们。

在Java中，处理这种注解的一种常见方式是通过注解处理器（Annotation Processors）。注解处理器是Java编译器的一个插件，可以在编译时检查源代码中的注解，并根据注解的内容生成额外的源代码、文件或其他类型的输出。

Java 注解与注解处理器_java 注册注解处理器-CSDN博客

1.
    - 使用场景
        - **语法检查**：例如，在Android开发中，SOURCE级别的注解可以用于IDE（如IntelliJ IDEA）的语法检查。这些注解可以帮助确保某些参数或字段的使用符合特定的规则或模式。
        - **APT（Annotation Processing Tool）技术**：APT允许在编译时处理注解，生成额外的源代码、资源文件等。SOURCE级别的注解通常与APT结合使用，以在编译阶段生成或检查代码。
    - **示例**：假设有一个名为 `@IntDef` 的注解，它用于限制某个字段或参数只能接受特定的整数值。这个注解被设置为SOURCE级别，因为它主要用于编译时的语法检查和类型检查，而不需要在运行时保留这些信息。

SOURCE级别

- 使用场景
    - **类库和框架内部**：很多类库和框架使用CLASS级别的注解来提供额外的元数据信息，这些信息在类加载时被使用，但在运行时对应用程序本身是不可见的。
    - **代码生成和分析工具**：例如，一些工具可能会在类加载时扫描CLASS级别的注解，并基于这些注解生成额外的代码或执行某些分析。
- **示例**：在Hibernate ORM框架中，`@Entity`、`@Table`、`@Column` 等注解都是CLASS级别的。这些注解在编译时被嵌入到字节码中，并在类加载时被Hibernate使用来解析ORM映射信息。