AOP+自定义注解

> 前言

上班后就看见公司有一些骚操作，特别来学习一下，此篇文章是作为 有一些基础学习AOP + 自定义注解

😉 开始学习自定义注解

注解定义和处理

自定义注解

Java的注解可以分为三类：

第一类是由编译器使用的注解，例如：

- `@Override`：让编译器检查该方法是否正确地实现了覆写；
- `@SuppressWarnings`：告诉编译器忽略此处代码产生的警告。

这类注解不会被编译进入`.class`文件，它们在编译后就被编译器扔掉了。

第二类是由工具处理`.class`文件使用的注解，比如有些工具会在加载class的时候，对class做动态修改，实现一些特殊的功能。这类注解会被编译进入`.class`文件，但加载结束后并不会存在于内存中。这类注解只被一些底层库使用，一般我们不必自己处理。

第三类是在程序运行期能够读取的注解，它们在加载后一直存在于JVM中，这也是最常用的注解。例如，一个配置了`@PostConstruct`的方法会在调用构造方法后自动被调用（这是Java代码读取该注解实现的功能，JVM并不会识别该注解）。

自定义注解参数配置

定义一个注解时，还可以定义配置参数。配置参数可以包括：

- 所有基本类型；
- String；
- 枚举类型；
- 基本类型、String、Class以及枚举的数组。

因为配置参数必须是常量，所以，上述限制保证了注解在定义时就已经确定了每个参数的值。

注解的配置参数可以有默认值，缺少某个配置参数时将使用默认值。

此外，大部分注解会有一个名为`value`的配置参数，对此参数赋值，可以只写常量，相当于省略了value参数。

如果只写注解，相当于全部使用默认值。

定义注解

Java语言使用`@interface`语法来定义注解（`Annotation`），它的格式如下：

```
/**
 * @author wmt
```

```
 * @Title:
 * @Package
 * @Description: 自定义注解
 * @date 2023/11/418:30
 */
// 下面的两个注解被称为元注解
@Target(ElementType.METHOD) //表示使用在方法上
@Retention(RetentionPolicy.RUNTIME) // 表示在运行时使用
public @interface MyAnnotation {
    int code();
    String msg() default "默认消息";
}
```

注解的参数类似无参数方法，可以用`default`设定一个默认值（强烈推荐）。最常用的参数应当命名为`value`。

元注解

有一些注解可以修饰其他注解，这些注解就称为元注解（meta annotation）。Java标准库已经定义了一些元注解，我们只需要使用元注解，通常不需要自己去编写元注解。

@Target

最常用的元注解是`@Target`。使用`@Target`可以定义`Annotation`能够被应用于源码的哪些位置：

- 类或接口：`ElementType.TYPE`；
- 字段：`ElementType.FIELD`；
- 方法：`ElementType.METHOD`；
- 构造方法：`ElementType.CONSTRUCTOR`；
- 方法参数：`ElementType.PARAMETER`。

实际上`@Target`定义的`value`是`ElementType[]`数组，只有一个元素时，可以省略数组的写法。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {

    ElementType[] value();
}
```

@Retention

另一个重要的元注解`@Retention`定义了`Annotation`的生命周期：

- 仅编译期：`RetentionPolicy.SOURCE`；
- 仅class文件：`RetentionPolicy.CLASS`；
- 运行期：`RetentionPolicy.RUNTIME`。

如果`@Retention`不存在，则该`Annotation`默认为`CLASS`。因为通常我们自定义的`Annotation`都是`RUNTIME`，所以，务必要加上`@Retention(RetentionPolicy.RUNTIME)`这个元注解：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

@Repeatable

使用`@Repeatable`这个元注解可以定义`Annotation`是否可重复。这个注解应用不是特别广泛，并且不能和Inherited重叠使用。

```
// 下面的注解被称为元注解
@Target(ElementType.METHOD) //表示使用在方法上
@Retention(RetentionPolicy.RUNTIME) //表示在运行时使用
@Repeatable(MyAnnotations.class) //表示可以重复注解
public @interface MyAnnotation {
    int code();
    String msg() default "默认消息";
}

@Retention(RetentionPolicy.RUNTIME) //表示在运行时使用  生命周期要相同
@Target(ElementType.METHOD)
public @interface MyAnnotations{
    MyAnnotation[] value();
}
```

经过`@Repeatable`修饰后，在某个类型声明处，就可以添加多个`@MyAnnotation`注解：

```
    @Test
    @MyAnnotation(code = 200)
    @MyAnnotation(code = 300, msg = "测试")
    void testAnnotation() {
    }
```

@Inherited

Inherited 翻译 遗传

使用`@Inherited`定义子类是否可继承父类定义的`Annotation`。`@Inherited`仅针对`@Target(ElementType.TYPE)`类型的`annotation`有效，并且仅针对`class`的继承，对`interface`的继承无效：

```
@Inherited // 父类继承注解  注意不能和@Repeatable重叠
@Target(ElementType.TYPE) //表示使用在方法上
public @interface MyAnnotation {
    int code();
    String msg() default "默认消息";
}
```

在使用的时候，如果一个类用到了`@MyAnnotation`：

```
@MyAnnotation(code = 300, msg = "测试")
public class Person {
}
```

则它的子类默认也定义了该注解：

```
public class Student extends Person {
}
```

处理注解

Java的注解本身对代码逻辑没有任何影响。根据`@Retention`的配置：

- `SOURCE`类型的注解在编译期就被丢掉了；
- `CLASS`类型的注解仅保存在class文件中，它们不会被加载进JVM；
- `RUNTIME`类型的注解会被加载进JVM，并且在运行期可以被程序读取。

如何使用注解完全由工具决定。`SOURCE`类型的注解主要由编译器使用，因此我们一般只使用，不编写。`CLASS`类型的注解主要由底层工具库使用，涉及到class的加载，一般我们很少用到。只有`RUNTIME`类型的注解不但要使用，还经常需要编写。

因此，我们只讨论如何读取`RUNTIME`类型的注解。

因为注解定义后也是一种`class`，所有的注解都继承自`java.lang.annotation.Annotation`，因此，读取注解，需要使用反射API。

Java提供的使用反射API读取`Annotation`的方法包括：

判断某个注解是否存在于`Class`、`Field`、`Method`或`Constructor`：

- `Class.isAnnotationPresent(Class)`
- `Field.isAnnotationPresent(Class)`
- `Method.isAnnotationPresent(Class)`
- `Constructor.isAnnotationPresent(Class)`

例如：

`ResultResp`类上面的自定义注解

```
@Data
@MyAnnotation(code = 200, msg = "测试的默认msg")
public class ResultResp {
    private Integer code;
    private String msg;
}
```

`@MyAnnotation` 自定义注解的定义

```
 */
// 下面的注解被称为元注解
@Inherited // 父类继承注解
@Target(ElementType.TYPE) //表示使用在方法上
@Retention(RetentionPolicy.RUNTIME) //表示在运行时使用
public @interface MyAnnotation {
    int code();
    String msg() default "默认消息";
}
```

使用反射API读取Annotation：

- `Class.getAnnotation(Class)`
- `Field.getAnnotation(Class)`
- `Method.getAnnotation(Class)`
- `Constructor.getAnnotation(Class)`

测试

```
@Test
    void testAnnotation() {
        Class<ResultResp> resultRespClass = ResultResp.class;
        // 通过反射判断这个类是否在这个类上面
        boolean annotationPresent =
resultRespClass.isAnnotationPresent(MyAnnotation.class);
        log.info("annotationPresent ={}", annotationPresent); //
annotationPresent =true
//         `Class.getAnnotation(Class)` 通过类去获取注解
        MyAnnotation annotation =
resultRespClass.getAnnotation(MyAnnotation.class);
        // code = 200, msg = 测试的默认msg,
        log.info("code = {}, msg = {}, ", annotation.code(), annotation.msg());
//         `Field.getAnnotation(Class)` 通过字段

//         `Method.getAnnotation(Class)` 通过方法

//         `Constructor.getAnnotation(Class)` 通过构造方法
    }
```

AOP

> 当前案例 基于spring boot

需求通过切面完成 方法增强获取方法中的参数 和操作时间 完成日志保存

切面常识就不简绍了默认懂

导入依赖

boot依赖

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
```

spring依赖

```
       <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.2.5.RELEASE<ersion>
        </dependency>
```

业务编写

> 首先先定义一个枚举用于操作类型的规定

```

/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 操作枚举
 * @date 2023/11/512:55
 */
@Getter
public enum OpEnum {
    SELECT_COMMODITY(20001, "查看商品"),
    UPDATE_COMMODITY(20002, "更新商品")
    ;

    /**
     * 操作code
     */
    private final Integer code;

    /**
     * 操作消息
     */
    private final String msg;

    OpEnum(Integer code, String msg) {
        this.code = code;
        this.msg = msg;
```

```
    }
}
```

> 编写一个 日志对象 用于方法操作后的对象保留

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 系统日志
 * @date 2023/11/512:57
 */
@Data
@ToString
public class SysLog {
    /**
     * 操作code
     */
    private  Integer code;

    /**
     * 操作消息
     */
    private  String msg;

    /**
     * 操作时间
     */
    private  LocalDateTime opTime;

    /**
     * 操作对象id
     */
    private Integer opObjectId;

    /**
     * SysLog对象构造器
     *
     * @param opEnum 操作枚举
     * @param opTime 操作时间
     * @param opObjectId 操作对象
     */
    public SysLog(OpEnum opEnum, LocalDateTime opTime, Integer opObjectId) {
        this.code = opEnum.getCode();
        this.msg = opEnum.getMsg();
        this.opTime = opTime;
        this.opObjectId = opObjectId;
    }

    SysLog(Integer code, String msg, LocalDateTime opTime, Integer opObjectId) {
        this.code = code;
        this.msg = msg;
        this.opTime = opTime;
```

```
        this.opObjectId = opObjectId;
    }
}
```

> 编写商品BO

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 商品信息
 * @date 2023/11/512:52
 */
@Data
@AllArgsConstructor
@EqualsAndHashCode
public class Commodity {
    /**
     * id
     */
    private Integer id;

    /**
     * 商品名称
     */
    private String name;

    /**
     * 商品价格
     */
    private Integer price;

    /**
     * 商品数量
     */
    private Integer count;
}
```

> 切面编写

注解的使用

@Before 前置通知 在目标方法前

@AfterReturning 后置通知 注解有retuning属性

@Around 环绕通知

@AfterThrowing 异常通知

@After 最终通知不管怎么都要被执行

@Aspect 表示这是一个切面

@Pointcut 辅助定义切入点，如果项目有多个切入点重复，就可以使用

切面的切入表达式（execution执行）

execution（访问权限 **方法返回值 方法声明 （参数）** 异常类型）

切入表达式

- 所有类上的所有方法：`execution(* *.*(..))`
- 特定包中的所有方法：`execution(* com.example.demo.*.*(..))`
- 特定类中的所有方法：`execution(* com.example.demo.MyClass.*(..))`
- 特定类中以get开头的所有方法：`execution(* com.example.demo.MyClass.get*(..))`
- 特定类中以set开头的所有方法，参数为String类型：`execution(* com.example.demo.MyClass.set*(String))`
- 特定注释标记的所有方法：`@annotation(org.springframework.transaction.annotation.Transactional)`

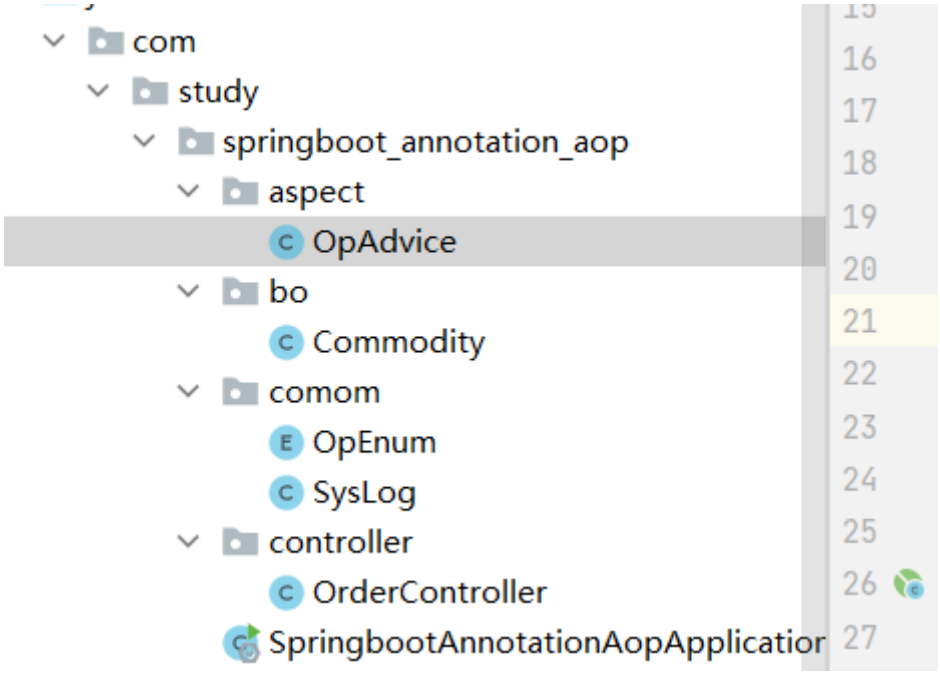
其中，`*`表示任意返回类型，`..`表示任意数量的参数，而`*`后面的点号表示任意方法名（或许可以使用通配符）。在实际应用中，您可以根据需要创建自己的特定表达式，以匹配要拦截的方法。

增强器是切面中定义的通知（advice），该通知在指定的连接点执行。通知类型包括@Before、@After、@Around、@AfterReturning和@AfterThrowing等。

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 操作切面
 * @date 2023/11/513:03
 */
@Component
@Aspect
@Slf4j
public class OpAdvice {
    /**
     * 在controller下的所有方法前切入    不灵活
     */
    @Before("execution(* com.study.springboot_annotation_aop.controller.*.*
(..))")
    public void registerLog(JoinPoint joinPoint){
        Object[] args = joinPoint.getArgs();
        // 获取到日志的详细详细 操作类型是写死的 不灵活
        SysLog sysLog = new SysLog(OpEnum.SELECT_COMMODITY, LocalDateTime.now(),
(Integer) args[0]);
        // 可以对这个日志进行保存到数据库 这是个简短的demo
        log.info("sysLog ={}", sysLog); //sysLog =SysLog(code=20001, msg=查看商品,
opTime=2023-11-05T15:45:46.222306800, opObjectId=0)
    }
}
```

> 上面的编写方式 不够灵活

项目结构：



> controller 层

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 订单查询
 * @date 2023/11/513:05
 */
@RestController
@RequestMapping("/order")

public class OrderController {

/**
 * 查看商品详细
 */
    @GetMapping("selectCommodity/{id}")
    public Commodity selectCommodity(@PathVariable("id") Integer id){
        Commodity commodity1 = new Commodity(0, "华为mete60", 9999, 10001);
        Commodity commodity2 = new Commodity(1, "华为mete50", 5999, 2001);
        List<Commodity> list = new ArrayList<>();
        list.add(commodity1);
        list.add(commodity2);
        return list.get(id);
    }
}
```

测试:

日志输出

```
sysLog =SysLog(code=20001, msg=查看商品, opTime=2023-11-05T15:45:46.222306800,
opObjectId=0)
```

> 以上是切面的正常使用，我觉得并不灵活

注解+AOP

上面的切面和注解结合起来，这样注解加切面的组合会更加灵活

> 先自定义注解

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 日志注解
 * @date 2023/11/516:06
 */

@Target(ElementType.METHOD) // 表示该注解用在方法上
@Retention(RetentionPolicy.RUNTIME) // 运行时
public @interface LogAnnotation {
    OpEnum value();
}
```

操作枚举

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 操作枚举
 * @date 2023/11/512:55
 */
@Getter
public enum OpEnum {
    SELECT_COMMODITY(20001, "查看商品"),
    UPDATE_COMMODITY(20002, "更新商品")
    ;

    /**
     * 操作code
     */
    private final Integer code;

    /**
     * 操作消息
     */
    private final String msg;

    OpEnum(Integer code, String msg) {
        this.code = code;
        this.msg = msg;
    }
}
```

日志对象

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 系统日志
 * @date 2023/11/512:57
 */
@Data
@ToString
@NoArgsConstructor
public class SysLog {
    /**
     * 操作code
     */
    private  Integer code;

    /**
```

```
     * 操作消息
     */
    private  String msg;

    /**
     * 操作时间
     */
    private  LocalDateTime opTime;

    /**
     * 操作对象id
     */
    private Integer opObjectId;

    /**
     * SysLog对象构造器
     *
     * @param opEnum 操作枚举
     * @param opTime 操作时间
     * @param opObjectId 操作对象
     */
    public SysLog(OpEnum opEnum, LocalDateTime opTime, Integer opObjectId) {
        this.code = opEnum.getCode();
        this.msg = opEnum.getMsg();
        this.opTime = opTime;
        this.opObjectId = opObjectId;
    }

    SysLog(Integer code, String msg, LocalDateTime opTime, Integer opObjectId) {
        this.code = code;
        this.msg = msg;
        this.opTime = opTime;
        this.opObjectId = opObjectId;
    }
}
```

> 商品Bo

```
/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 商品信息
 * @date 2023/11/512:52
 */
@Data
@AllArgsConstructor
@EqualsAndHashCode
public class Commodity {
    /**
     * id
     */
    private Integer id;
```

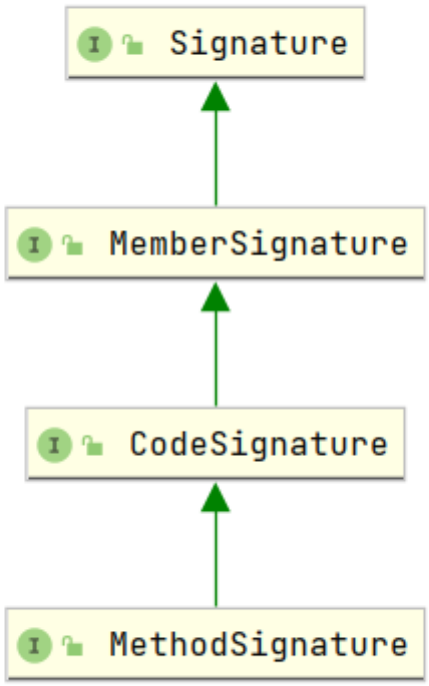
```
    /**
     * 商品名称
     */
    private String name;

    /**
     * 商品价格
     */
    private Integer price;

    /**
     * 商品数量
     */
    private Integer count;
}
```

> 切面的编写

```
package com.study.springboot_annotation_aop.aspect;

import com.study.springboot_annotation_aop.comom.GlobalException;
import com.study.springboot_annotation_aop.comom.LogAnnotation;

import com.study.springboot_annotation_aop.comom.OpEnum;
import com.study.springboot_annotation_aop.comom.SysLog;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;
import java.lang.reflect.Method;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.IdentityHashMap;
import java.util.Set;


/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 操作切面
 * @date 2023/11/513:03
 */
@Component
@Aspect
@Slf4j
public class OpAdvice {
    /**
     * 定义切入点 以打注解的地方为切入点
     */
    @Pointcut(value =
"@annotation(com.study.springboot_annotation_aop.comom.LogAnnotation)")
    public void pointCut(){
    }
```

```
    /**
     * 使用注解切入
     */
    @Before(value = "pointCut()")
    public void registerLog(JoinPoint joinPoint)  {
        // 获取到方法
        MethodSignature methodSignature =(MethodSignature)
joinPoint.getSignature();
        Method method = methodSignature.getMethod();

        // 判断该方法上面是否有这个注解
        if (method.isAnnotationPresent(LogAnnotation.class)) {
            // 获取自定义注解
            LogAnnotation annotation =
method.getAnnotation(LogAnnotation.class);
            // 获取参数列表
            Object[] args = joinPoint.getArgs();
            OpEnum opEnum = annotation.value();
            // 构造日志对象
            SysLog sysLog = new SysLog();
            sysLog.setCode(opEnum.getCode());
            sysLog.setMsg(opEnum.getMsg());
            sysLog.setOpObjectId((Integer) args[0]);
            sysLog.setOpTime(LocalDateTime.now());
            // 可以对这个日志进行保存到数据库
            //  sysLog =SysLog(code=20002, msg=更新商品, opTime=2023-11-
05T17:42:17.601449400, opObjectId=0)
            log.info("sysLog ={}", sysLog);
        }

    }

    /**
     * 异常处理
     *
     * @param joinPoint
     */
    @AfterThrowing(value = "pointCut()", throwing ="ex")
    public void disposeError(JoinPoint joinPoint, Exception ex){
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        // 这里获取不到message不知道为什么
        log.error("methodName = {} error = ", method.getName(),
ex.getMessage());

    }
}
```

如图 Signature是MethodSignature的父类



> controller 类

```
package com.study.springboot_annotation_aop.controller;

import com.study.springboot_annotation_aop.bo.Commodity;

import com.study.springboot_annotation_aop.comom.LogAnnotation;
import com.study.springboot_annotation_aop.comom.OpEnum;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


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

/**
 * @author wmt
 * @Title:
 * @Package
 * @Description: 订单查询
 * @date 2023/11/513:05
 */
@RestController
@RequestMapping("/order")

/**
 * 查看商品详细
```

```
 */

public class OrderController {
    /**
     * 查询商品
     *
     * @param id 商品id
     * @return Commodity
     */
    @GetMapping("selectCommodity/{id}")
    @LogAnnotation(OpEnum.SELECT_COMMODITY)
    public Commodity selectCommodity(@PathVariable("id") Integer id){
        Commodity commodity1 = new Commodity(0, "华为mete60", 9999, 10001);
        Commodity commodity2 = new Commodity(1, "华为mete50", 5999, 2001);
        List<Commodity> list = new ArrayList<>();
        list.add(commodity1);
        list.add(commodity2);
        return list.get(id);
    }

    /**
     * 更新商品
     *
     * @param id 商品id
     * @return Commodity
     */
    @GetMapping("updateCommodity/{id}")
    @LogAnnotation(OpEnum.UPDATE_COMMODITY)
    public Commodity updateCommodity(@PathVariable("id") Integer id){
        Commodity commodity1 = new Commodity(0, "华为mete60", 9999, 10001);
        Commodity commodity2 = new Commodity(1, "华为mete50", 5999, 2001);
        List<Commodity> list = new ArrayList<>();
        list.add(commodity1);
        list.add(commodity2);
        Commodity commodity = list.get(id);
        commodity.setPrice(500);
        return commodity;
    }
}
```