Spring Library

Pionpill [1]

本文档为作者归纳的 Spring 笔记。

2022 年 11 月 7 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已有 Java 开发经验，基础知识不再赘述。

这篇文章记录 Spring 大家族以及一些相关组件的常见用法 (主要是注解形式)，如果看原理请找另一篇笔记: 《Spring Principle》。

本人的编写及开发环境如下：

- Java: Java11
- SpringBoot: 2.7.3
- OS: Windows11
- MySQL: 8.0.3

2022 年 11 月 7 日

目录

第一部分

Spring Library

1 Spring Framework 注解

1.1 Spring Bean 基础注解

这节的前四个个注解功能基本都是一样的，都是将类作为 bean 注入到 spring 容器中进行管理，只不过它们使用的场景不同。后几个注解是对前几个注解的修饰。

选择上，如果确定是 MVC 的哪一层，就选择对应的具体注解 (@Controller, @Service, @Repository) 标识，如果不确定，但又知道它是各组件，就用 @Component 标识。这四个注解除了可以明确层次关系，没有其他区别。

1.1.1 @Component

@Component 标注一个普通的组件类，通知 Spring 将类纳入到 Spring Bean 容器中并进行管理。默认定义的 bean 是单例的。可以通过 @Component(“beanName”) @Scope(“prototype”) 改变，下面同样。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {
    String value() default "";
}
```

1.1.2 @Controller

对应 Spring MVC 控制层，主要用户接受用户 http 请求并调用 Service 层返回数据给前端页面。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    @AliasFor(annotation = Component.class)
    String value() default "";
}
```

1.1.3 @Service

对应 Spring MVC 业务层。主要用于获取 pojo 层的的数据并进行业务处理，

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

```
@Component
public @interface Service {
    @AliasFor(annotation = Component.class)
    String value() default "";
}
```

1.1.4 @Repository

@Repository 对应持久层 (pojo)。主要用于直接和数据库交互。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {
    @AliasFor(annotation = Component.class)
    String value() default "";
}
```

1.1.5 @Scope

@Scope 注解用于指定 Bean 的作用范围，也即采用的设计模式。

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Scope {
    @AliasFor("scopeName")
  String value() default "";
  @AliasFor("value")
  String scopeName() default "";
    ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;
}
```

一般的有两个选项，”singleton” 与”prototype”，分别代表单例设计模式与原型设计模式。默认采用单例模式。

1.1.6 @PostConstrucor 与 @PreDestroy

顾名思义，这两个注解作用的生命周期分别是: 构造方法后和销毁前。对应了 Bean 生命周期的 init-method 和 destroy-method。

```
@Documented
@Retention (RUNTIME)
@Target(METHOD)
public @interface PostConstruct

@Documented
```

```
@Retention (RUNTIME)
@Target(METHOD)
public @interface PreDestroy
```

1.1.7 @Bean

@Bean 主要用于将一个方法地返返回值注册为 Bean:

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Bean
```

为什么要这样设计呢，主要是为了管理外部的 Bean:

```
@Bean
public DataSource getDataSource() {
    DruidDataSource di = new DruidDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ...
    return ds;
}
```

同时，@Bean 也是自动装配的，如果有形参，容器会自动注入对应类型的形参。

1.2 Spring Bean 配置注解

1.2.1 @Configuration

声明当前类为配置类，相当于 xml 形式的 Spring 配置。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
    @AliasFor(annotation = Component.class)
    String value() default "";
    boolean proxyBeanMethods() default true;
}
```

1.2.2 @ComponentScan

顾名思义，用来扫描 Component 并批量注册 Bean，默认情况下扫描当前包及子包的 Component，可以自定义扫描位置[1]:

[1]延申文献: https://zhuanlan.zhihu.com/p/520827986

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
    @AliasFor("basePackages")
    String[] value() default {};
    ......
}
```

1.2.3 @PropertySource

@PropertySource 注解用于绑定 properties 文件:

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(PropertySources.class)
public @interface PropertySource {
    String[] value();
    ......
}
```

一般的 @PropertySource 作用在 Config 文件上[2]。

1.2.4 @Import

@import 注解用于导入别的注解，当有多个 Config 时，一般在 Spring 的 Config 中导入其他配置文件。

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Import {
    Class<?>[] value();
}
```

举个例子:

```java
@Configuration
@Import
public class SpringConfig
```

[2]延申文献: https://blog.csdn.net/tenghu8888/article/details/119791417

1.3 Spring DI 注解

前面 Spring 已经完成了 Bean 的标识与扫描，但这仅仅是将 Bean 装入了 IoC 容器，下面注解实现了 DI。

1.3.1 @Autowired

@Autowired 注解顾名思义，进行自动装配，是一个功能十分强大，实现比较复杂，有争议的注解。

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER,
    ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {
    boolean required() default true;
}
```

默认情况下，@Autowired 根据类型实现自动装配，构造方法注入和 setter 注入都可以实现。@Autowired 可以直接作用在属性上，而不直接使用 setter 方法注入，它将使用暴力反射的方式将对应的类型注入到属性中[3]。注意，既然使用了暴力反射，就要提供 bean 的无参构造方法。

@Autowired 默认使用也推荐使用类型注入，但类型注入必然会带来问题: 同类型二义性，不知道用哪个。

1.3.2 @Qualifier

@Qualifier 用于按名称注入，他必须依赖于 @Autowired 注解。

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE,
    ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier {
    String value() default "";
}
```

使用时，必须先给出 @Autowired 注解:

```
@Autowired
@Qualifier("bookDao") // 对应某个 bean 名为 bookDao
private BookDao bookDao;
```

当然这样有个问题，耦合度高!!!

[3]延申文献: https://blog.csdn.net/Weixiaohuai/article/details/123005140

1.3.3 @Value

注意这里是 Spring Framework 里的 @Value 不是 lombok 的 @Value。

@Autowired 有一个缺陷，他只能注入引用类型，注入基本类型[4]要依靠 @Value:

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
    ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Value {
    String value();
}
```

@Value 一般用于注入 properties 文件的内容。需要结合 @PropertySource 注解使用。

```
@PropertySource("classpath:jdbc.properties") // classpath: 可以不加
public class JdbcProperties {
    ......
}

@Repository
public class BookDao {
    @Value(${name})   // jdbc.properties 存在字段和 ${} 内容相同
    private String name;
}
```

1.4 Spring AOP 注解

这小节的注解需要下面依赖:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

1.4.1 @EnableAspectJAutoProxy

该注解用于启动对应配置类下的 AOP，具体表现为识别 @Aspect 注解[5]。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {
    boolean proxyTargetClass() default false;
    boolean exposeProxy() default false;
```

[4]想念 Python

[5]拓展: https://blog.csdn.net/yuan882696yan/article/details/115359291

```
}
```

1.4.2 @Aspect

把当前类标识为一个切面供容器读取。注意在只用它之前要确保类已经被 @Component 标注为 Bean。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Aspect {
    String value() default "";
}
```

1.4.3 @Pointcut

该注解用于指定切入点。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Pointcut {
    String value() default "";
    String argNames() default "";
}
```

切入点定义依托于一个不具有实际意义的方法进行，及无参数，无返回值，方法体无实际逻辑。可以理解为，运行到该方法时，需要加功能 (Advice)。

切入点表达式[6]: 要进行增强的方法的描述方式。一个切入点表达式的标准格式如下:

动作关键字 (访问修饰符返回值包名. 类/接口名. 方法名 (参数) 异常名)

例如: `execution(public User com.pionpill.UserService.findById(int))`

一般的，动作关键字都是 `execution`，访问修饰符都是 `public` 可以省略，大多数情况下没有异常，可以省略。

切入点表达式中可以使用通配符简化书写，但是这样会降低性能和可读性，应该减少或者不使用，这里仅作说明:

- *: 表示任意类型返回值/包名/类名/方法名，但至少一个。
- ..: 表示任意多个参数，任意包地址，可以没有。
- +: 代表子类类型。

同时的，在切入点表达式中可以使用 “与或非” 三种运算符来组合切入点表达式。

```
@Pointcut("@execution(org.com.common.aspect.annotation.PermissionData)")
public void pointCut() { }
```

[6]这里只讲最常用的语法。

1.4.4 Advice

Advice 也即要进行增强处理的方法，它包括五个注解。这五个注解的使用方式类似。都需要加入切入点方法作为参数。假设切入点方法如下:

```
@Pointcut("execution (void ...)")
public void pt() { }
```

@Before, @After 在切入点之前/后执行

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Before {
    String value();
    String argNames() default "";
}
```

这两个注解使用起来非常简单:

```
@Before("pt()")
public void before () {
    System.out.println("Before execution ...");
}
```

@Around 这是最重要也最常用的 Advice 注解。它对应的方法需要一个 ProceedingJoinPoint 类型的参数用于标记切入点方法执行的位置。

```
@Around("pt()")
public void around(ProceedingJoinPoint pjp) throws Throwable{
    System.out.println("Before execution ...");
    pjp.proceed(); // 对原始方法的调用
    System.out.println("After execution ...");
}
```

这里需要抛出异常，因为不确定原始方法究竟有没有异常，所以强制抛出异常。

这里有个注意点，如果在 around 中返回了其他值，将覆盖原始方法的返回值。

@AfterReturning, @AfterThrowing 分别用于在返回值之后和抛出异常后执行，不是很常用。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AfterReturning {
    String value() default "";
    String pointcut() default "";
    String returning() default "";
    String argNames() default "";
}
```

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AfterThrowing {
    String value() default "";
    String pointcut() default "";
    String throwing() default "";
    String argNames() default "";
}
```

1.5 Spring 事务注解

这小节的注解需要下面依赖:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
</dependency>
```

1.5.1 @EnableTransactionManagement

用于开启注解式事务驱动。在配置类上开启了这个，才能使用其他事务注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {
    boolean proxyTargetClass() default false;
    AdviceMode mode() default AdviceMode.PROXY;
    int order() default Ordered.LOWEST_PRECEDENCE;
}
```

1.5.2 @Transactional

该注解用于添加 Spring 事务管理。一般不用于开在实现类中，而是接口中。

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";
    String[] label() default {};
    Propagation propagation() default Propagation.REQUIRED;
```

```
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
    String timeoutString() default "";
    boolean readOnly() default false;
    Class<? extends Throwable>[] rollbackFor() default {};
    String[] rollbackForClassName() default {};
    Class<? extends Throwable>[] noRollbackFor() default {};
    String[] noRollbackForClassName() default {};
}
```

事务相关的配置如下:

表 1 事务相关配置

属性	作用	示例
readOnly	设置是否为只读事务	true: 只读
timeout	设置事务超时事件	timeout = -1: 永不超时
rollbackFor	设置事务回滚异常	rollbackFor = (NullPointException.class)
noRollbackFor	设置事务不回滚异常	noRollbackFor = (NullPointException.class)
propagation	设置事务传播行为	...

默认情况下，Spring 事务只对 Unchecked Exception (Error, Runtime Exception) 异常进行回滚。如果需要添加其他异常或者将某些异常剔除，需要用到 rollBackFor 和 noRollBackFor 属性。propagation 用于传播事务行为，我们将主事务 (需要执行的那个事务) 称为事务管理员，被加入的其他事务成为事务协调员。

表 2 事务传播行为

传播属性	事务管理员	事务协调员
REQUIRED(默认)	开启 无	加入 新建
REQUIRED_NEW	开启 无	新建 新建
SUPPORTS	开启 无	加入 无
NOT_SUPPOETED	开启 无	无 无
MANDATORY	开启 无	加入 ERROR
NEVER	开启 无	ERROR 无
NESTED	设置 savePoint, 事务将回滚到此处。	

2 Spring MVC 注解

2.1 MVC 控制层相关注解

2.1.1 @RequestMapping

作用: 将 Web 请求与请求处理类中的方法进行映射

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
public @interface RequestMapping {
    // 省略具体实现
}
```

它有以下的六个配置属性:

- value: 映射请求的 URL 或者别名。
- method: 兼容 HTTP 的方法名
- params: 根据 HTTP 参数的存在、缺省或值对请求进行过滤
- header: 根据 HTTP Header 的存在、缺省或值对请求进行过滤
- consume: 设定在 HTTP 请求正文中允许使用的媒体类型
- product: 在 HTTP 响应体中允许使用的媒体类型

在使用 @RequestMapping 之前，请求处理类还需要使用 @Controller 进行标记。

```
@Controller
@RequestMapping("/home")
public class HomeController {
    ......
}
```

@GetMapping 用于于处理 HTTP GET 请求，并将请求映射到具体的处理方法中。具体来说，@GetMapping 是一个组合注解，它相当于是 @RequestMapping(method=RequestMethod.GET) 的快捷方式。下面几个映射注解也类似。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {
    // 省略具体实现
}
```

给个例子，下面几个不给了:

```
@Controller
@RequestMapping("/home")
```

```
public class HomeController {
    @GetMapping("/users")
    public List<User> findAllUsers() {
        List<User> users = userService.findAll();
        return users;
    }
    ......
}
```

@PostMapping 注解用于处理 HTTP POST 请求。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.POST)
public @interface PostMapping {
    // 省略具体实现
}
```

@PutMapping 注解用于处理 HTTP PUT 请求，

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.PUT)
public @interface PutMapping {
    // 省略具体实现
}
```

@DeleteMapping 注解用于处理 HTTP Delete 请求，

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.DELETE)
public @interface DeleteMapping {
    // 省略具体实现
}
```

@PatchMapping 注解用于处理 HTTP PATCH 请求，

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.DELETE)
public @interface DeleteMapping {
    // 省略具体实现
}
```

2.1.2 @ResponseBody

用于将 Controller 的方法返回的对象，通过 springmvc 提供的 HttpMessageConverter 接口转换为指定格式的数据如：json、xml 等，通过 Response 响应给客户端。

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ResponseBody { }
```

2.1.3 @RequestParam

该注解用于将请求参数绑定到控制器的方法参数上:

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestParam {
    @AliasFor("name")
    String value() default "";
    @AliasFor("value")
    String name() default "";
    boolean required() default true;
    String defaultValue() default ValueConstants.DEFAULT_NONE;
}
```

在参数上加入该注解后有如下效果:

- 不加该注解的前端和后端参数名需要一致。
- 加该注解的参数为必传参数，不加为非必传，也可通过 `required = false` 设置为非必传。
- 可以通过 `value` 指定参数名。
- 可以通过 `defaultValue` 指定参数默认值。

2.1.4 @RequestBody

用于将请求中请求体包含的数据传递给请求参数，一个处理器方法只能使用一次:

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestBody {
    boolean required() default true;
}
```

2.1.5 @EnableWebMvc

该注解可以开启 SpringMVC 多项辅助功能:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc { }
```

2.1.6 @DateTimeFormat

该注解用于将传入的 String 类型参数转换为 Data 类型，用于设定日期时间型数据格式。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER,
    ElementType.ANNOTATION_TYPE})
public @interface DateTimeFormat {
    String style() default "SS";
    ISO iso() default ISO.NONE;
    String pattern() default "";
    String[] fallbackPatterns() default {};
    enum ISO { DATE, TIME, DATE_TIME, NONE }
}
```

举个例子:

```
@DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")
private Date date;
```

2.2 MVC 视图相关注解

2.2.1 @ModelAttribute

@ModelAttribute 主要的作用是将数据添加到模型对象中，用于视图页面显示。@ModelAttribute 注释的位置不同, 和其他注解一起使用时有很多种用法。[7]

```
@Target({ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ModelAttribute {
    @AliasFor("name")
    String value() default "";
    @AliasFor("value")
    String name() default "";
    boolean binding() default true;
```

[7]参考文献: https://blog.csdn.net/yue_xx/article/details/105740360

```
}
```

主要用途可分为注解在方法上和注解在参数上:

- 用在方法上，@ModelAttribute 注解的方法会在 Controller 每个方法被执行前调用。根据返回值不同也有区分。
  - void 方法: 一般会在方法的参数中使用 Model 参数，在方法体内将模型数据添加到模型对象中。

```
@ModelAttribute
public void NoneReturn(@RequestParam String data, Model model) {
    model.addAttribute("指定一个名称",data);
}
```

  - 具体类型方法: 一般用 @ModelAttribute 的 value 属性指定 model 属性的名称。model 属性对应的对象就是方法的返回值。不指定名称，则 model 属性名就会默认是返回类型的首字母小写。

```
@ModelAttribute(name = "tacoOrder")
public TacoOrder order() {
    return new TacoOrder();
}
```

  - @ModddelAttribute 和 @RequestMapping 共同注解一个方法时: 此时方法的返回值并不是表示一个视图名称，而是 model 属性的值，此时的视图名称就是 @RequestMapping 中指定的访问路径的最后一层去掉扩展名。

```
@RequestMapping(value = "指定一个访问路径")
@ModelAttribute("指定一个名称")
public String Fix() {
    return "猛虎蔷薇";
}
```

- 标记在方法的参数上，会将客户端传递过来的参数按名称注入到指定对象中，并且会将这个对象自动加入 ModelMap 中，便于 View 层使用。

2.2.2 @SessionAttributes

若希望在多个请求之间共用数据，则可以在控制器类上标注一个 @SessionAttributes，配置需要在 session 中存放的数据范围，Spring MVC 将存放在 model 中对应的数据暂存到 HttpSession 中。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface SessionAttributes {
    @AliasFor("names")
    String[] value() default {};
    @AliasFor("value")
```

```
    String[] names() default {};
    Class<?>[] types() default {};
}
```

3 SpringBoot 注解

3.1 SpringBoot 常见注解

3.1.1 @SpringBootApplication

@SpringBootApplication 表明这是一个 Spring 引导应用程序，默认 SpringBoot 项目的启动类会被该注解修饰。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM,classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
    // ... 此处省略源码
}
```

可以看出，该注解是由三个注解组成，这里做简单说明，细节请查询相应的注解:

- @ComponentScan: 自动扫描并加载符合条件的组件。
- @EnableAutoConfiguration: 借助 @Import 的支持，收集和注册特定场景相关的 bean 定义。
- @SpringBootConfiguration: 标注当前类是配置类，并会将当前类内声明的一个或多个以 @Bean 注解标记的方法的实例纳入到 spring 容器中，并且实例名就是方法名。

参考文献:

- CSDN: https://blog.csdn.net/qq_28289405/article/details/81302498

第二部分

Spring Data Library

4 Spring Data JPA 注解

导入依赖如下:

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

4.1 表相关注解

4.1.1 @Entity

被该注解标注的实体类将会被 JPA 管理控制，在程序运行时，JPA 会识别并映射到指定的数据库表。

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
  String name() default "";
}
```

属性 name 用于指定实体类名称，默认为实体类的非限定名。

4.1.2 @Table

若表名与实体类名称不同时，使用 @Table(name=" 表名")，与 @Entity 标注并列使用，置于实体类声明语句之前。如果表名和实体类名相同，那么 @Table 可以省略。

注意是 javax.persistence.Table 下的 @Table:

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}
```

@Table 的属性如下:

- name: 对应表名。
- catalog: 表所属的数据库目录。通常为数据库名。
- schema: 表所属的数据库模式。通常为数据库名。
- uniqueConstraints: 设置约束条件。

4.1.3 @Id

@Id 用于实体类的一个属性或者属性对应的 getter 方法上，被标注的的属性将映射为数据库主键。

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {}
```

4.1.4 @Basic

@Basic 是实体类与数据库字段映射时最简单的类型。它可以用于持久类属性或实例变量，类型包含基本类型，包装类，枚举类，实现了 Serializable 接口的类型。

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

两个属性意义如下:

- fetch: 加载机制：默认 EAGER，即时加载，可以改为 LAZY 懒加载。
- optional: 判断属性是否能为空，默认可以。

简言之，与数据库对应的属性都要加上 @Basic，如果你在实体类属性上不加 @Basic 注解，它也会自动加上 @Basic，并使用默认值。一般不需要显示书写，除非要改属性。

4.1.5 @GeneratedValue

与 @Id 一同使用，用于标注主键的生成策略，通过 strategy 属性指定。

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}
```

在 javax.persistence.GenerationType 中定义了以下几种可供选择的策略:

- AUTO: 默认方式，JPA 自动选择合适的策略。
- IDENTITY: 由数据库生成，采用数据库自增长，Oracle 不支持。
- SEQUENCE: 通过数据库序列生成，MySQL 不支持。
- TABLE：通过表产生主键，框架借由表模拟序列产生主键。

generator 属性的值是一个字符串, 默认为"", 其声明了主键生成器的名称。使用非 AUTO 的策略需要结合其他注解使用，不是很常用，不讲了。

4.1.6 @Transient

表示该属性并非一个数据库表的字段的映射，ORM 框架将忽略该属性，不会对其持久化。

4.1.7 @Cloumn

用于定义实体属性:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```

4.1.8 @Embedded @Embeddable

通过此注解可以在 Entity 模型中使用一般的 java 对象, 不过此对象还需要用 @Embeddable 注解标注。

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Embedded { }
```

例如：User 包括 id,name,city,street,zip 属性，我们希望 city,street,zip 属性映射为 Address 对象，这样，User 对象将具有 id,name 和 address 这三个属性，Address 对象要定义为 @Embededable。

参考例子: https://blog.csdn.net/je_ge/article/details/53678238

5 MyBatis Plus 注解

注意 5.1. *MyBatis 和 MyBatis Plus 不是 Spring 框架内的东西，只是开发中比较常用 SSM 框架。MP 的源码有中文注释，所以这里写的很简略。*

MP 坐标如下:

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>最新版本</version>
</dependency>
```

5.1 表相关注解

5.1.1 @TableName

表名注解，用于关联一个数据表:

```
    @Documented
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.TYPE, ElementType.ANNOTATION_TYPE})
    public @interface TableName {
        String value() default "";
        String schema() default "";
        boolean keepGlobalPrefix() default false;
        String resultMap() default "";
        boolean autoResultMap() default false;
        String[] excludeProperty() default {};
    }
```

主要属性如下:

- value: 表名;
- resultMap: XML 中 resultMap 的 id;
- autoResultMap: 是否自动构建 resultMap 并使用。

5.1.2 @TableId

标识主键

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE})
public @interface TableId {
    String value() default "";
    IdType type() default IdType.NONE;
}
```

主要属性如下:

- value: 数据库主键字段名;
- type: 主键类型，默认为 idType.NONE。

枚举 idType 如下:

- AUTO: 数据库 ID 自增;
- NONE: 无状态, 该类型为未设置主键类型 (注解里等于跟随全局, 全局里约等于 INPUT);
- INPUT: insert 前自行 set 主键值;
- ASSIGN_ID: 分配 ID(主键类型为 Number(Long 和 Integer) 或 String)，使用雪花算法。
- ASSIGN_UUID: 分配 UUID, 主键类型为 String(since 3.3.0)。

5.1.3 @TableField

非主键的字段注解:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE})
public @interface TableField { ... }
```

属性太多不写了。

5.1.4 @TableLogic

逻辑处理注解:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE})
public @interface TableLogic {
    String value() default "";
    String delval() default "";
}
```

两个属性意义如下:

- value: 逻辑未删除值;
- delval: 逻辑删除值。

5.1.5 @EnumValue

标识是个枚举类型:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE})
public @interface EnumValue { }
```

5.1.6 @Version

乐观锁标记。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.ANNOTATION_TYPE})
public @interface Version { }
```

第三部分

Plugins Library

6 Validation 注解

导入依赖如下:

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

可以发现 validation 以及被集成到 spring 中了，为什么要这样做呢? 简言之，validation 的一部分注解会在 mvc 的参数解析器中进行解析。

6.1 约束性注解

6.1.1 @NotNull,@NotEmpty,@NotBlank

这三个注解均用来哦按段是否为空，分别为:

- @NotNull: 属性不可以为 null，但可以为空串。
- @NotEmpty: 属性不可以为 null, 且不可以为空串 (长度大于 0)。
- @NotBlank: 只能作用于 String 类型的属性上，属性不可以为 null，且 trim() 后不可以为空串。

其中 @NotNull 源代码如下:

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface NotNull {
    String message() default "{javax.validation.constraints.NotNull.message}";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotNull[] value();
    }
}
```

@NotNull 对应的还有一个 @Null 注解。

6.1.2 @Size,@Length,@Max,@Min

这三个注解均用于判断数值大小/字符串长度:

- @Min: 验证 Number 和 String 对象是否大于等于指定的值。
- @Max: 验证 Number 和 String 对象是否小于等于指定的值。
- @Size(min=,max=): 验证对象（Array,Collection,Map,String）长度是否在给定范围内。
- @Length(min=,max=): 验证字符串长度是否在给定范围内。

其中 @Size 源代码如下:

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface Size {
    String message() default "{javax.validation.constraints.Size.message}";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };
    int min() default 0;
    int max() default Integer.MAX_VALUE;
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Size[] value();
    }
}
```

6.1.3 @Digits,@DecimalMax,@DecimalMin

被注解的元素必须为一个数字，其值必须在可接受的范围内。主要参数有两个:

- integer: 整数部分位数。
- fraction: 小数部分位数。

```
@Digits(integer = 3,fraction = 0)
private String ccCVV;
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface Digits {
    ...
}
```

@DecimalMax 与 @DecimalMin 分别表示数字的最大值与最小值。

6.1.4 @AssertFalse,@AssertTrue

被注解的对象必须是 True 或 False，否则抛错。

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface AssertTrue {
    ...
}
```

6.1.5 @Future,@Past

用于限定日期必须在当前日期的未来或者过期，注意源码中并没有 value，也就是说这里的日期是指程序运行时的日期。

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface Future {
    ...
}
```

6.1.6 @Pattern

限定字符串必须满足正则表达式:

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint(validatedBy = { })
public @interface Pattern {
    ...
}
```

6.1.7 @Email,@URL

用于限定被修饰的属性必须是邮箱或者 URL。

6.2 验证性注解

6.2.1 @Valid,@Validated

@Valid 注解用于校验。[8]

```
@Target({ METHOD, FIELD, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Documented
public @interface Valid {
}
```

使用校验，首先要在实体类相应字段上添加用于校验的注解。如: @Min。其次在 Controller 层的方法的要校验的参数上添加 @Valid 注解，并且需要传入 BindingResult 对象，用于获取校验失败情况下的反馈信息，代码如下:

```
@PostMapping("/girls")
public Girl addGirl(@Valid Girl girl, BindingResult bindingResult) {
    if(bindingResult.hasErrors()){
        System.out.println(bindingResult.getFieldError().getDefaultMessage());
        return null;
    }
    return girlResposity.save(girl);
}
```

@Validated 是 @Valid 的一次封装，是 Spring 提供的校验机制使用。@Valid 不提供分组功能。当一个实体类需要多种验证方式时，例：对于一个实体类的 id 来说，新增的时候是不需要的，对于更新时是必须的。可以通过 groups 对验证进行分组。

不分配 groups 时，默认每次都进行验证，对参数需要多种验证方式时，也可通过分配不同的组达到目的:

```
@NotEmpty(groups={First.class})
@Size(min=3,max=8,groups={Second.class})
private String name;
```

注意 Validated 是在 springframework 包下，不是 javax 包下:

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Validated {
    Class<?>[] value() default {};
}
```

[8]参考: https://blog.csdn.net/gaojp008/article/details/80583301

7 Lombok 注解

Lombok 依赖坐标如下:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

和 Validation，Lombok 没有被集成到 spring 中，为什么呢？因为没必要，Lombok 是在编译阶段改动语法树达到效果 (所以几乎所有注解 Retention 都是 SOURCE，CLASS)，相当于改变了 Java 源码，和 spring 没什么关系。

7.1 验证性注解

7.1.1 @NonNull

用于限定被注解的属性不能为 Null。

```
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
    ElementType.LOCAL_VARIABLE, ElementType.TYPE_USE})
@Retention(RetentionPolicy.CLASS)
@Documented
public @interface NonNull { }
```

等效代码:

```
// Lombok
public NonNullExample(@NonNull Person person) {
    this.name = person.getName();
}
// Java
public NonNullExample(@NonNull Person person) {
    if (person == null) {
      throw new NullPointerException("person is marked non-null but is null");
    }
    this.name = person.getName();
}
```

7.2 辅助性注解

7.2.1 @Cleanup

该注解用于自动调用 close() 方法释放资源，和 try-with-resource 语句有异曲同工之处。

```
@Target(ElementType.LOCAL_VARIABLE)
@Retention(RetentionPolicy.SOURCE)
```

```
public @interface Cleanup {
  String value() default "close";
}
```

等效代码:

```
// Lombok
@Cleanup InputStream in = new FileInputStream(args[0]);
@Cleanup OutputStream out = new FileOutputStream(args[1]);
byte[] b = new byte[10000];
while (true) {
    int r = in.read(b);
    if (r == -1) break;
    out.write(b, 0, r);
}
// Java
InputStream in = new FileInputStream(args[0]);
    try {
        OutputStream out = new FileOutputStream(args[1]);
        try {
            byte[] b = new byte[10000];
            while (true) {
                int r = in.read(b);
                if (r == -1) break;
                out.write(b, 0, r);
            }
        } finally {
            if (out != null) {
                out.close();
            }
        }
    } finally {
        if (in != null) {
            in.close();
        }
    }
```

7.2.2 @SneakyThrows

这个注解是用来解决这样的代码的:

```
try{
    ...
}catch(Exception e){
    throw new RuntimeException(e);
}
```

使用了该注解后可以自动向上抛错误:

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.SOURCE)
public @interface SneakyThrows {
```

```
    Class<? extends Throwable>[] value() default java.lang.Throwable.class;
}
```

等价效果如下:

```
// Lombok
@SneakyThrows
public void run() {
    throw new Throwable();
}
// Java
public void run() {
    try {
        throw new Throwable();
    } catch (Throwable t) {
        throw Lombok.sneakyThrow(t);
    }
}
```

7.3 增强性注解

7.3.1 @Getter @Setter

这些注解用于自动生成对应的 getXXX setXXX 方法。

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Getter {
    boolean lazy() default false;
    ...
}
```

等效代码:

```
// Lombok
@Getter @Setter private int age = 10;
// Java
private int age = 10;
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
```

默认的 setXXX 方法返回 void，链式编程参考接下来的注解。

getXXX 有一个重要的参数 lazy，默认关闭，如果启用，会缓存 getXXX 返回的对象，创建一个 private final 变量，以方便接下来使用。

```
// Lombok
@Getter(lazy=true) private final double[] cached = expensive();
// Java
private final java.util.concurrent.AtomicReference<java.lang.Object> cached = new
    java.util.concurrent.AtomicReference<java.lang.Object>();
public double[] getCached() {
    java.lang.Object value = this.cached.get();
    if (value == null) {
        synchronized(this.cached) {
            value = this.cached.get();
            if (value == null) {
                final double[] actualValue = expensive();
                value = actualValue == null ? this.cached : actualValue;
                this.cached.set(value);
            }
        }
    }
    return (double[])(value == this.cached ? null : value);
}
```

7.3.2 @Accessors

@Accessors 不是 stable 的注解，是一个 experimental 注解，比较常用所以也写一下。

该注解主要作用是：当属性字段在生成 getter 和 setter 方法时，做一些相关的设置。

```
@Target({ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.SOURCE)
public @interface Accessors {
  boolean fluent() default false;
  boolean chain() default false;
  boolean makeFinal() default false;
  String[] prefix() default {};
}
```

它的几个属性作用如下：

- fluent: 如果改为 true: getter 方法不会有 get 前缀，setter 方法不会有 set 前缀。
- chain: 链式编程，如果改为 true，setter 方法将会返回当前对象。
- prefix: 当该数组有值时，表示忽略字段中对应的前缀，生成对应的 getter 和 setter 方法。
- makeFinal: 如果改为 true，生成的方法会被标记为 final。

7.3.3 @ToString

这个注解用于自动生成 toString 方法。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ToString {
```

```
    ...
}
```

默认情形下，它生成的 toString 格式如下:

```
private String name;
private int age;
@Override
public String toString() {
    return "Student{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
}
```

他还有很多参数,提供自定义输出格式,另外有两个相关注解 @ToString.Exclude @ToString.Include。其他几个增强注解基本上都有类似的相关注解。

7.3.4 @EqualsAndHashCode

该注解用于生成 equals 和 hashCode 方法。该注解参数很多，实现也比较复杂。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface EqualsAndHashCode {
    boolean callSuper() default false;
    ...
}
```

有个 callSuper 和生成 hashCode 有关:

- callSuper = true: 根据子类和从父类继承的字段生成 hashCode。
- callSuper = false: 根据子类本身字段生成 hashCode。

该注解生成的 equals 方法逻辑和 String 类型的 equals 逻辑类似。下面仅给出官方实例生成的两个方法。

```
@Override public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof EqualsAndHashCodeExample)) return false;
    EqualsAndHashCodeExample other = (EqualsAndHashCodeExample) o;
    if (!other.canEqual((Object)this)) return false;
    if (this.getName() == null ? other.getName() != null :
        !this.getName().equals(other.getName())) return false;
    if (Double.compare(this.score, other.score) != 0) return false;
    if (!Arrays.deepEquals(this.tags, other.tags)) return false;
    return true;
}
@Override public int hashCode() {
    final int PRIME = 59;
    int result = 1;
    final long temp1 = Double.doubleToLongBits(this.score);
```

```
    result = (result*PRIME) + (this.name == null ? 43 : this.name.hashCode());
    result = (result*PRIME) + (int)(temp1 ^ (temp1 >>> 32));
    result = (result*PRIME) + Arrays.deepHashCode(this.tags);
    return result;
}
```

7.3.5 Constructor

Lombok 有三个用于生成构造函数的注解: @NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor。

首先看一下源代码:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface NoArgsConstructor {
    String staticName() default "";
    AnyAnnotation[] onConstructor() default {};
  AccessLevel access() default lombok.AccessLevel.PUBLIC;
  boolean force() default false;
    ...
}
```

它的几个常用属性如下:

- access: 设置构造器的访问修饰符,如果是单例模式,可以设置为 AccessLevel.PRIVATE。
- staticName: 用于设置一个静态构造函数。
- force: 如果有未赋值的 final 字段，强制初始化默认值。

```
// Lombok
@NoArgsConstructor(staticName = "UserHa")
public class User {
    private String username;
    private String password;
}
// Java
public class User {
    private String username;
    private String password;
    private User() { }
    public static User UserHa() {
        return new User();
    }
}
```

@RequiredArgsConstructor 也是在类上使用，但是这个注解可以生成带参或者不带参的构造方法。若带参数，只能是类中所有带有 @NonNull 注解的和以 final 修饰的未经初始化的字段。

```
// Lombok
```

```
@RequiredArgsConstructor
public class User {
    private final String gender;
    @NonNull
    private String username;
    private String password;
}
// Java
public class User {
    private final String gender;
    @NonNull
    private String username;
    private String password;

    public User(String gender, @NonNull String username) {
        if (username == null) {
            throw new NullPointerException("username is marked @NonNull but is null");
        } else {
            this.gender = gender;
            this.username = username;
        }
    }
}
```

最后一个是 @AllArgsConstructor 需要注意的是，这里的全参不包括已初始化的 final 字段（主要是 final 字段，一旦被赋值不允许再被修改），没啥好说的。

7.3.6 @Builder

@Builder 注解是一个相当复杂的注解，被他注解的类可以实现 Builder 模式[9]。总而言之，它可以完成 Builder 模式所需要的所有代码。

```
@Target({TYPE, METHOD, CONSTRUCTOR})
@Retention(SOURCE)
public @interface Builder {
    ...
}
```

@Builder 的属性很多，建议查源代码了解。这个注解本身没什么好说的，对应的属性也都是和 Builder 模式相关。

7.3.7 @Synchronized

用于替换 synchronize 关键字或 lock 锁。

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.SOURCE)
public @interface SneakyThrows {
```

[9]Builder 模式不解释，可以看这篇文章: https://blog.csdn.net/qq_17678217/article/details/86507693

```
  Class<? extends Throwable>[] value() default java.lang.Throwable.class;
}
```

可以传入一个属性值用于设置锁对象。

```
// Lombok
public final String NAME = "唐嫣";

@Synchronized(value = "NAME")
public void name() {
  System.out.println(NAME);
}
// Java
public void name() {
  super.getClass();
  synchronized ("唐嫣") {
    System.out.println("唐嫣");
  }
}
```

7.3.8 @With

该注解用于“改变”final 属性，具体的方法是，通过 withXXX 函数，返回一个新的对象，这样 final 对应的属性就变相改变了。

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface With {
    AccessLevel value() default AccessLevel.PUBLIC;
    AnyAnnotation[] onMethod() default {};
    AnyAnnotation[] onParam() default {};
    ...
}
```

示例代码:

```
// Lombok
public class WithExample {
    @With(AccessLevel.PROTECTED) @NonNull private final String name;
    @With private final int age;
    public WithExample(@NonNull String name, int age) {
        this.name = name;
        this.age = age;
    }
}
// Java
public class WithExample {
    private @NonNull final String name;
    private final int age;
    public WithExample(String name, int age) {
        if (name == null) throw new NullPointerException();
```

```
        this.name = name;
        this.age = age;
    }
    protected WithExample withName(@NonNull String name) {
        if (name == null) throw new java.lang.NullPointerException("name");
        return this.name == name ? this : new WithExample(name, age);
    }
    public WithExample withAge(int age) {
        return this.age == age ? this : new WithExample(name, age);
    }
}
```

7.3.9 @Log

该注解用于创建对应的 Log 对象：

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Log {
    String topic() default "";
}
```

示例代码如下：

```
// Lombok
@Log
public class LogExample {
    public static void main(String... args) {
        log.severe("Something's wrong here");
    }
}
// Java
public class LogExample {
    private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger
        (LogExample.class.getName());

    public static void main(String... args) {
        log.severe("Something's wrong here");
    }
}
```

7.4 集成注解

7.4.1 @Data

@Data 注解是 @ToString, @EqualsAndHashCode, @Getter, @Setter, @RequiredArgsConstructor 注解的集合。

7.4.2 @Value

@Value 是针对不可变对象 (例如 String) 的 @Data。它会将被修饰的类转换为如下形式:

- 成员变量由 private final 修饰;
- 提供带参数构造函数;
- 仅为成员变量提供带参数的构造器;
- 不允许子类覆盖方法;

详细文献: https://blog.csdn.net/cauchy6317/article/details/102646009