

# Spring/SpringBoot 常用注解

## 1. @SpringBootApplication

这里先单独拎出 `@SpringBootApplication` 注解说一下，虽然我们一般不会主动去使用它。

Guide 哥：这个注解是 Spring Boot 项目的基石，创建 SpringBoot 项目之后会默认在主类加上。

```
1  @SpringBootApplication
2  public class SpringSecurityJwtGuideApplication {
3      public static void main(java.lang.String[] args) {
4          SpringApplication.run(SpringSecurityJwtGuideApplication.class, args);
5      }
6  }
```

我们可以把 `@SpringBootApplication` 看作是

`@Configuration`、`@EnableAutoConfiguration`、`@ComponentScan` 注解的集合。

```
1  package org.springframework.boot.autoconfigure;
2  @Target(ElementType.TYPE)
3  @Retention(RetentionPolicy.RUNTIME)
4  @Documented
5  @Inherited
6  @SpringBootConfiguration
7  @EnableAutoConfiguration
8  @ComponentScan(excludeFilters = {
9      @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
10     @Filter(type = FilterType.CUSTOM, classes =
11         AutoConfigurationExcludeFilter.class) })
12     public @interface SpringBootApplication {
13         .....
14     }
15     package org.springframework.boot;
16     @Target(ElementType.TYPE)
17     @Retention(RetentionPolicy.RUNTIME)
18     @Documented
19     @Configuration
20     public @interface SpringBootConfiguration {
21
22     }
```

根据 SpringBoot 官网，这三个注解的作用分别是：

- `@EnableAutoConfiguration`：启用 SpringBoot 的自动配置机制
- `@ComponentScan`：扫描被 `@Component` (`@Service`, `@Controller`) 注解的 bean，注解默认会

扫描该类所在的包下所有的类。

- `@Configuration`：允许在 Spring 上下文中注册额外的 bean 或导入其他配置类

## 2. Spring Bean 相关

### 2.1. @Autowired

自动导入对象到类中，被注入进的类同样要被 Spring 容器管理比如：Service 类注入到 Controller 类中。

```
1  @Service
2  public class UserService {
3      .....
4  }
5
6  @RestController
7  @RequestMapping("/users")
8  public class UserController {
9      @Autowired
10     private UserService userService;
11     .....
12 }
```

### 2.2. @Component , @Controller , @Service , @Repository

我们一般使用 `@Autowired` 注解让 Spring 容器帮我们自动装配 bean。要想把类标识成可用于

`@Autowired` 注解自动装配的 bean 的类,可以采用以下注解实现：

- `@Component`：通用的注解，可标注任意类为 `Spring` 组件。如果一个 Bean 不知道属于哪个层，可以使用 `@Component` 注解标注。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Controller`：对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

### 2.3. @RestController

`@RestController` 注解是 `@Controller`和 `@ResponseBody` 的合集,表示这是个控制器 bean,并且是将函数的返回值直接填入 HTTP 响应体中,是 REST 风格的控制器。

现在都是前后端分离，说实话我已经很久没有用过 `@Controller`。如果你的项目太老了的话，就当我没说。

单独使用 `@Controller` 不加 `@ResponseBody` 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的 Spring MVC 的应用，对应于前后端不分离的情况。`@Controller` + `@ResponseBody` 返回 JSON 或 XML 形式数据

## 2.4. @Scope

声明 Spring Bean 的作用域，使用方法：

```
1  @Bean
2  @Scope("singleton")
3  public Person personSingleton() {
4      return new Person();
5  }
```

四种常见的 Spring Bean 的作用域：

- singleton: 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype: 每次请求都会创建一个新的 bean 实例。
- request: 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效。
- session: 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效。

## 2.5. @Configuration

一般用来声明配置类，可以使用 `@Component` 注解替代，不过使用 `Configuration` 注解声明配置类更加语义化。

```
1  @Configuration
2  public class AppConfig {
3      @Bean
4      public TransferService transferService() {
5          return new TransferServiceImpl();
6      }
7
8  }
```

## 3. 处理常见的 HTTP 请求类型

- GET: 请求从服务器获取特定资源。举个例子: `GET /users` (获取所有学生)
- POST: 在服务器上创建一个新的资源。举个例子: `POST /users` (创建学生)
- PUT: 更新服务器上的资源 (客户端提供更新后的整个资源)。举个例子: `PUT /users/12` (更新编号为 12 的学生)
- DELETE: 从服务器删除特定的资源。举个例子: `DELETE /users/12` (删除编号为 12 的学生)
- PATCH: 更新服务器上的资源 (客户端提供更改的属性，可以看做是部分更新)，使用的比较少，这里就不举例子了。

### 3.1. GET 请求

`@GetMapping("users")` 等价于

`@RequestMapping(value="/users",method=RequestMethod.GET)`

```
1  @GetMapping("/users")
2  public ResponseEntity<List<User>> getAllUsers() {
3      return userRepository.findAll();
4  }
```

### 3.2. POST 请求

`@PostMapping("users")` 等价于

`@RequestMapping(value="/users",method=RequestMethod.POST)`

关于 `@RequestBody` 注解的使用，在下面的“前后端传值”这块会讲到。

```
1  @PostMapping("/users")
2  public ResponseEntity<User> createUser(
3      @Valid @RequestBody UserCreateRequest userCreateRequest) {
4      return userRepository.save(user);
5  }
```

### 3.3. PUT 请求

`@PutMapping("/users/{userId}")` 等价于

`@RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)`

```
1  @PutMapping("/users/{userId}")
2  public ResponseEntity<User> updateUser(
3      @PathVariable(value = "userId") Long userId,
4      @Valid @RequestBody UserUpdateRequest userUpdateRequest) {
5      .....
6  }
```

### 3.4. DELETE 请求

`@DeleteMapping("/users/{userId}")` 等价于

`@RequestMapping(value="/users/{userId}",method=RequestMethod.DELETE)`

```
1  @DeleteMapping("/users/{userId}")
2  public ResponseEntity deleteUser(
3      @PathVariable(value = "userId") Long userId){
4      .....
5  }
```

### 3.5. PATCH 请求

一般实际项目中，我们都是 PUT 不够用了之后才用 PATCH 请求去更新数据。

```
1  @PatchMapping("/profile")
2  public ResponseEntity updateStudent(
3      @RequestBody StudentUpdateRequest studentUpdateRequest) {
4      studentRepository.updateDetail(studentUpdateRequest);
5      return ResponseEntity.ok().build();
6  }
```

## 4. 前后端传值

掌握前后端传值的正确姿势，是你开始 CRUD 的第一步！

### 4.1. @PathVariable 和 @RequestParam

`@PathVariable` 用于获取路径参数，`@RequestParam` 用于获取查询参数。

举个简单的例子：

```
1  @GetMapping("/klasses/{klassId}/teachers")
2  public List<Teacher> getClassRelatedTeachers(
3      @PathVariable("klassId") Long klassId,
4      @RequestParam(value = "type", required = false) String type ) {
5      ...
6  }
```

如果我们请求的 url 是： `/klasses/{123456}/teachers?type=web`

那么我们服务获取到的数据就是： `klassId=123456,type=web` 。

### 4.2. @RequestBody

用于读取 Request 请求（可能是 POST,PUT,DELETE,GET 请求）的 body 部分并且 **Content-Type** 为 **application/json** 格式的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会使用 `HttpMessageConverter` 或者自定义的 `HttpMessageConverter` 将请求的 body 中的 json 字符串转换为 java 对象。

我用一个简单的例子来给演示一下基本使用！

我们有一个注册的接口：

```

1  @PostMapping("/sign-up")
2  public ResponseEntity signUp(
3      @RequestBody @Valid UserRegisterRequest userRegisterRequest) {
4      userService.save(userRegisterRequest);
5      return ResponseEntity.ok().build();
6  }

```

`UserRegisterRequest` 对象:

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class UserRegisterRequest {
5      @NotBlank
6      private String userName;
7      @NotBlank
8      private String password;
9      @FullName
10     @NotBlank
11     private String fullName;
12 }

```

我们发送 post 请求到这个接口，并且 body 携带 JSON 数据：

```

1  {"userName":"coder","fullName":"shuangkou","password":"123456"}

```

这样我们的后端就可以直接把 json 格式的数据映射到我们的 `UserRegisterRequest` 类上。



需要注意的是：一个请求方法只可以有一个 `@RequestBody`，但是可以有多个 `@RequestParam` 和 `@PathVariable`。如果你的方法必须要用两个 `@RequestBody` 来接受数据的话，大概率是你的数据库设计或者系统设计出问题了！

## 5. 读取配置信息

很多时候我们需要将一些常用的配置信息比如阿里云 oss、发送短信、微信认证的相关配置信息等等放到配置文件中。

下面我们来看一下 Spring 为我们提供了哪些方式帮助我们从配置文件中读取这些配置信息。

我们的数据源 `application.yml` 内容如下：

```
1  wuhan2020: 2020年初武汉爆发了新型冠状病毒，疫情严重，但是，我相信一切都会过去！武汉加油！中国加油！
2
3  my-profile:
4      name: Guide哥
5      email: koushuangbwcx@163.com
6
7  library:
8      location: 湖北武汉加油中国加油
9      books:
10         - name: 天才基本法
11             description: 二十二岁的林朝夕在父亲确诊阿尔茨海默病这天，得知自己暗恋多年的校园男神裴之即将出国深造的消息——对方考取的学校，恰是父亲当年为她放弃的那所。
12         - name: 时间的秩序
13             description: 为什么我们记得过去，而非未来？时间“流逝”意味着什么？是我们存在于时间之内，还是时间存在于我们之中？卡洛·罗韦利用诗意的文字，邀请我们思考这一亘古难题——时间的本质。
14         - name: 了不起的我
15             description: 如何养成一个新习惯？如何让心智变得更成熟？如何拥有高质量的关系？ 如何走出人生的艰难时刻？
```

## 5.1. @Value（常用）

使用 `@Value("${property}")` 读取比较简单的配置信息：

```
1  @Value("${wuhan2020}")
2  String wuhan2020;
```

## 5.2. @ConfigurationProperties（常用）

通过 `@ConfigurationProperties` 读取配置信息并与 bean 绑定。

```
1  @Component
2  @ConfigurationProperties(prefix = "library")
3  class LibraryProperties {
4      @NotEmpty
5      private String location;
6      private List<Book> books;
7
8      @Setter
9      @Getter
10     @ToString
11     static class Book {
12         String name;
13         String description;
14     }
15     省略getter/setter
16     .....
17 }
```

你可以像使用普通的 Spring bean 一样，将其注入到类中使用。

### 5.3. @PropertySource（不常用）

**@PropertySource** 读取指定 properties 文件

```
1  @Component
2  @PropertySource("classpath:website.properties")
3
4  class WebSite {
5      @Value("${url}")
6      private String url;
7
8      省略getter/setter
9      .....
10 }
```

## 6. 参数校验

数据的校验的重要性就不用说了，即使在前端对数据进行校验的情况下，我们还是要对传入后端的数据再进行一遍校验，避免用户绕过浏览器直接通过一些 HTTP 工具直接向后端请求一些违法数据。

**JSR(Java Specification Requests)** 是一套 JavaBean 参数校验的标准，它定义了很多常用的校验注解，我们可以直接将这些注解加在我们 JavaBean 的属性上面，这样就可以在需要校验的时候进行校验了，非常方便！

校验的时候我们实际用的是 **Hibernate Validator** 框架。Hibernate Validator 是 Hibernate 团队最初的数据校验框架，Hibernate Validator 4.x 是 Bean Validation 1.0（JSR 303）的参考实现，Hibernate Validator 5.x 是 Bean Validation 1.1（JSR 349）的参考实现，目前最新版的 Hibernate Validator 6.x 是 Bean Validation 2.0（JSR 380）的参考实现。

SpringBoot 项目的 spring-boot-starter-web 依赖中已经有 hibernate-validator 包，不需要引用相关依赖。如下图所示（通过 idea 插件—Maven Helper 生成）：



```

▼ spring-boot-starter-web : 2.1.8.RELEASE [compile]
  ▼ hibernate-validator : 6.0.17.Final [compile]
    classmate : 1.4.0 [compile]
    jboss-logging : 3.3.3.Final [compile]
    validation-api : 2.0.1.Final [compile]
  spring-boot-starter : 2.1.8.RELEASE [compile]
  ▼ spring-boot-starter-json : 2.1.8.RELEASE [compile]
    jackson-databind : 2.9.9.3 [compile]
    ▼ jackson-datatype-jdk8 : 2.9.9 [compile]
      jackson-core : 2.9.9 [compile]
      jackson-databind : 2.9.9.3 [compile]
    ▼ jackson-datatype-jsr310 : 2.9.9 [compile]
      jackson-annotations : 2.9.0 [compile]
      jackson-core : 2.9.9 [compile]
      jackson-databind : 2.9.9.3 [compile]
    ▼ jackson-module-parameter-names : 2.9.9 [compile]
      jackson-core : 2.9.9 [compile]
      jackson-databind : 2.9.9.3 [compile]
    spring-boot-starter : 2.1.8.RELEASE [compile]
    spring-web : 5.1.9.RELEASE [compile]
  ▼ spring-boot-starter-tomcat : 2.1.8.RELEASE [compile]
    javax.annotation-api : 1.3.2 [compile]
    tomcat-embed-core : 9.0.24 [compile]

```

知乎 @Java领域佼佼者

非 SpringBoot 项目需要自行引入相关依赖包，这里不多做讲解

需要注意的是：所有的注解，推荐使用 JSR 注解，即 `javax.validation.constraints`，而不是 `org.hibernate.validator.constraints`

## 6.1. 一些常用的字段验证的注解

- `@NotEmpty` 被注释的字符串的不能为 null 也不能为空
- `@NotBlank` 被注释的字符串非 null，并且必须包含一个非空白字符
- `@Null` 被注释的元素必须为 null
- `@NotNull` 被注释的元素必须不为 null
- `@AssertTrue` 被注释的元素必须为 true
- `@AssertFalse` 被注释的元素必须为 false
- `@Pattern(regex=, flag=)` 被注释的元素必须符合指定的正则表达式
- `@Email` 被注释的元素必须是 Email 格式。
- `@Min(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@Max(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@DecimalMin(value)` 被注释的元素必须是一个数字，其值必须大于等于指定的最小值
- `@DecimalMax(value)` 被注释的元素必须是一个数字，其值必须小于等于指定的最大值
- `@Size(max=, min=)` 被注释的元素的大小必须在指定的范围内
- `@Digits(integer, fraction)` 被注释的元素必须是一个数字，其值必须在可接受的范围内

- `@Past` 被注释的元素必须是一个过去的日期
- `@Future` 被注释的元素必须是一个将来的日期
- .....

## 6.2. 验证请求体 (RequestBody)

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class Person {
5
6      @NotNull(message = "classId 不能为空")
7      private String classId;
8
9      @Size(max = 33)
10     @NotNull(message = "name 不能为空")
11     private String name;
12
13     @Pattern(regexp = "(^Man$|^Woman$|^UGM$)", message = "sex 值不在可选范围")
14     @NotNull(message = "sex 不能为空")
15     private String sex;
16
17     @Email(message = "email 格式不正确")
18     @NotNull(message = "email 不能为空")
19     private String email;
20
21 }
```

我们在需要验证的参数上加上了 `@Valid` 注解，如果验证失败，它将抛出 `MethodArgumentNotValidException`。

```

1  @RestController
2  @RequestMapping("/api")
3  public class PersonController {
4
5      @PostMapping("/person")
6      public ResponseEntity<Person> getPerson(@RequestBody @Valid Person person) {
7          return ResponseEntity.ok().body(person);
8      }
9  }
```

## 6.3. 验证请求参数 (Path Variables 和 Request Parameters)

一定一定不要忘记在类上加上 `Validated` 注解了，这个参数可以告诉 Spring 去校验方法参数。

```

1  @RestController
2  @RequestMapping("/api")
3  @Validated
4  public class PersonController {
5
6      @GetMapping("/person/{id}")
7      public ResponseEntity<Integer> getPersonByID(@Valid @PathVariable("id")
8      @Max(value = 5,message = "超过 id 的范围了") Integer id) {
9          return ResponseEntity.ok().body(id);
10     }
11 }

```

## 7. 全局处理 Controller 层异常

介绍一下我们 Spring 项目必备的全局处理 Controller 层异常。

相关注解：

1. `@ControllerAdvice` :注解定义全局异常处理类
2. `@ExceptionHandler` :注解声明异常处理方法

如何使用呢？拿我们在第 5 节参数校验这块来举例子。如果方法参数不对的话就会抛出 `MethodArgumentNotValidException`，我们来处理这个异常。

```

1  @ControllerAdvice
2  @ResponseBody
3  public class GlobalExceptionHandler {
4
5      /**
6       * 请求参数异常处理
7       */
8      @ExceptionHandler(MethodArgumentNotValidException.class)
9      public ResponseEntity<?>
10     handleMethodArgumentNotValidException(MethodArgumentNotValidException ex,
11     HttpServletRequest request) {
12         .....
13     }
14 }

```

## 8. JPA 相关

## 8.1. 创建表

`@Entity` 声明一个类对应一个数据库实体。

`@Table` 设置表明

```
1  @Entity
2  @Table(name = "role")
3  public class Role {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7      private String name;
8      private String description;
9      省略getter/setter.....
10 }
```

## 8.2. 创建主键

`@Id` : 声明一个字段为主键。

使用 `@Id` 声明之后，我们还需要定义主键的生成策略。我们可以使用 `@GeneratedValue` 指定主键生成策略。

1.通过 `@GeneratedValue` 直接使用 JPA 内置提供的四种主键生成策略来指定主键生成策略。

```
1  @Id
2  @GeneratedValue(strategy = GenerationType.IDENTITY)
3  private Long id;
```

JPA 使用枚举定义了 4 中常见的主键生成策略，如下：

枚举替代常量的一种用法

```
1  public enum GenerationType {
2
3      /**
4       * 使用一个特定的数据库表格来保存主键
5       * 持久化引擎通过关系数据库的一张特定的表格来生成主键，
6       */
7      TABLE,
8
9      /**
10     *在某些数据库中,不支持主键自增长,比如Oracle、PostgreSQL其提供了一种叫做"序列
11     (sequence)"的机制生成主键
12     */
13     SEQUENCE,
14
15     /**
16     * 主键自增长
17     */
18     IDENTITY,
```

```

18
19     /**
20      *把主键生成策略交给持久化引擎(persistence engine),
21      *持久化引擎会根据数据库在以上三种主键生成 策略中选择其中一种
22      */
23     AUTO
24 }
25 @GeneratedValue`注解默认使用的策略是`GenerationType.AUTO
26 public @interface GeneratedValue {
27
28     GenerationType strategy() default AUTO;
29     String generator() default "";
30 }

```

一般使用 MySQL 数据库的话，使用 `GenerationType.IDENTITY` 策略比较普遍一点（分布式系统的话需要另外考虑使用分布式 ID）。

2.通过 `@GenericGenerator` 声明一个主键策略，然后 `@GeneratedValue` 使用这个策略

```

1  @Id
2  @GeneratedValue(generator = "IdentityIdGenerator")
3  @GenericGenerator(name = "IdentityIdGenerator", strategy = "identity")
4  private Long id;

```

等价于：

```

1  @Id
2  @GeneratedValue(strategy = GenerationType.IDENTITY)
3  private Long id;

```

jpa 提供的主键生成策略有如下几种：

```

1  public class DefaultIdentifierGeneratorFactory
2      implements MutableIdentifierGeneratorFactory, Serializable,
3      ServiceRegistryAwareService {
4
5      @SuppressWarnings("deprecation")
6      public DefaultIdentifierGeneratorFactory() {
7          register( "uuid2", UUIDGenerator.class );
8          register( "guid", GUIDGenerator.class );    // can be done with UUIDGenerator +
9          strategy
10         register( "uuid", UUIDHexGenerator.class );    // "deprecated" for new use
11         register( "uuid.hex", UUIDHexGenerator.class ); // uuid.hex is deprecated
12         register( "assigned", Assigned.class );
13         register( "identity", IdentityGenerator.class );
14         register( "select", SelectGenerator.class );
15         register( "sequence", SequenceStyleGenerator.class );
16         register( "seqhilo", SequenceHiLoGenerator.class );
17         register( "increment", IncrementGenerator.class );
18         register( "foreign", ForeignGenerator.class );
19         register( "sequence-identity", SequenceIdentityGenerator.class );
20         register( "enhanced-sequence", SequenceStyleGenerator.class );

```

```

19     register( "enhanced-table", TableGenerator.class );
20 }
21
22 public void register(String strategy, Class generatorClass) {
23     LOG.debugf( "Registering IdentifierGenerator strategy [%s] -> [%s]", strategy,
generatorClass.getName() );
24     final Class previous = generatorStrategyToClassNameMap.put( strategy,
generatorClass );
25     if ( previous != null ) {
26         LOG.debugf( "    - overriding [%s]", previous.getName() );
27     }
28 }
29
30 }

```

### 8.3. 设置字段类型

`@Column` 声明字段。

示例：

设置属性 `userName` 对应的数据库字段名为 `user_name`，长度为 32，非空

```

1  @Column(name = "user_name", nullable = false, length=32)
2  private String userName;

```

设置字段类型并且加默认值，这个还是挺常用的。

```

1  Column(columnDefinition = "tinyint(1) default 1")
2  private Boolean enabled;

```

### 8.4. 指定不持久化特定字段

`@Transient`：声明不需要与数据库映射的字段，在保存的时候不需要保存进数据库。

如果我们想让 `secret` 这个字段不被持久化，可以使用 `@Transient` 关键字声明。

```

1  Entity(name="USER")
2  public class User {
3
4      .....
5      @Transient
6      private String secret; // not persistent because of @Transient
7
8  }

```

除了 `@Transient` 关键字声明，还可以采用下面几种方法：

```

1  static String secret; // not persistent because of static
2  final String secret = "Satish"; // not persistent because of final
3  transient String secret; // not persistent because of transient

```

一般使用注解的方式比较多。

## 8.5. 声明大字段

`@Lob` :声明某个字段为大字段。

```
1  @Lob
2  private String content;
```

更详细的声明：

```
1  @Lob
2  //指定 Lob 类型数据的获取策略， FetchType.EAGER 表示非延迟 加载，而 FetchType.LAZY 表示延
   迟加载；
3  @Basic(fetch = FetchType.EAGER)
4  //columnDefinition 属性指定数据表对应的 Lob 字段类型
5  @Column(name = "content", columnDefinition = "LONGTEXT NOT NULL")
6  private String content;
```

## 8.6. 创建枚举类型的字段

可以使用枚举类型的字段，不过枚举字段要用 `@Enumerated` 注解修饰。

```
1  public enum Gender {
2      MALE("男性"),
3      FEMALE("女性");
4
5      private String value;
6      Gender(String str){
7          value=str;
8      }
9  }
10
11  @Entity
12  @Table(name = "role")
13  public class Role {
14      @Id
15      @GeneratedValue(strategy = GenerationType.IDENTITY)
16      private Long id;
17      private String name;
18      private String description;
19      @Enumerated(EnumType.STRING)
20      private Gender gender;
21      省略getter/setter.....
22  }
```

数据库里面对应存储的是 MAIL/FEMAIL。

## 8.7. 增加审计功能

只要继承了 `AbstractAuditBase` 的类都会默认加上下面四个字段。

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @MappedSuperclass
5  @EntityListeners(value = AuditingEntityListener.class)
6  public abstract class AbstractAuditBase {
7
8      @CreatedDate
9      @Column(updatable = false)
10     @JsonIgnore
11     private Instant createdAt;
12
13     @LastModifiedDate
14     @JsonIgnore
15     private Instant updatedAt;
16
17     @CreatedBy
18     @Column(updatable = false)
19     @JsonIgnore
20     private String createdBy;
21
22     @LastModifiedBy
23     @JsonIgnore
24     private String updatedBy;
25 }
```

我们对应的审计功能对应地配置类可能是下面这样的（Spring Security 项目）：

```
1  @Configuration
2  @EnableJpaAuditing
3  public class AuditSecurityConfiguration {
4      @Bean
5      AuditorAware<String> auditorAware() {
6          return () -> Optional.ofNullable(SecurityContextHolder.getContext())
7              .map(SecurityContext::getAuthentication)
8              .filter(Authentication::isAuthenticated)
9              .map(Authentication::getName);
10     }
11 }
```

简单介绍一下上面设计到的一些注解：

1. `@CreatedDate`：表示该字段为创建时间时间字段，在这个实体被 insert 的时候，会设置值
2. `@CreatedBy`：表示该字段为创建人，在这个实体被 insert 的时候，会设置值  
`@LastModifiedDate`、`@LastModifiedBy` 同理。

`@EnableJpaAuditing`：开启 JPA 审计功能。



## 8.8. 删除/修改数据

`@Modifying` 注解提示 JPA 该操作是修改操作,注意还要配合 `@Transactional` 注解使用。

```
1  @Repository
2  public interface UserRepository extends JpaRepository<User, Integer> {
3
4      @Modifying
5      @Transactional(rollbackFor = Exception.class)
6      void deleteByUsername(String userName);
7  }
```

## 8.9. 关联关系

- `@OneToOne` 声明一对一关系
- `@OneToMany` 声明一对多关系
- `@ManyToOne` 声明多对一关系
- `@ManyToMany` 声明多对多关系

## 9. 事务 @Transactional

在要开启事务的方法上使用 `@Transactional` 注解即可!

```
1  @Transactional(rollbackFor = Exception.class)
2  public void save() {
3      .....
4  }
```

我们知道 Exception 分为运行时异常 RuntimeException 和非运行时异常。在 `@Transactional` 注解中如果不配置 `rollbackFor` 属性,那么事物只会在遇到 RuntimeException 的时候才会回滚,加上 `rollbackFor=Exception.class`,可以让事物在遇到非运行时异常时也回滚。

`@Transactional` 注解一般用在可以作用在 类 或者 方法 上。

- 作用于类: 当把 `@Transactional` 注解放在类上时,表示所有该类的 public 方法都配置相同的事务属性信息。
- 作用于方法: 当类配置了 `@Transactional`,方法也配置了 `@Transactional`,方法的事务会覆盖类的事务配置信息。

## 10. json 数据处理

## 10.1. 过滤 json 数据

**@JsonIgnoreProperties** 作用在类上用于过滤掉特定字段不返回或者不解析。

```
1 //生成json时将userRoles属性过滤
2 @JsonIgnoreProperties({"userRoles"})
3 public class User {
4
5     private String userName;
6     private String fullName;
7     private String password;
8     @JsonIgnore
9     private List<UserRole> userRoles = new ArrayList<>();
10 }
```

**@JsonIgnore** 一般用于类的属性上，作用和上面的 **@JsonIgnoreProperties** 一样。

```
1 public class User {
2
3     private String userName;
4     private String fullName;
5     private String password;
6     //生成json时将userRoles属性过滤
7     @JsonIgnore
8     private List<UserRole> userRoles = new ArrayList<>();
9 }
```

## 10.2. 格式化 json 数据

**@JsonFormat** 一般用来格式化 json 数据。：

比如：

```
1 @JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-dd'T'HH:mm:ss.SSS'Z'",
2             timezone="GMT")
3 private Date date;
```

## 10.3. 扁平化对象

```
1 @Getter
2 @Setter
3 @ToString
4 public class Account {
5     @JsonUnwrapped
6     private Location location;
7     @JsonUnwrapped
8     private PersonInfo personInfo;
9
10    @Getter
11    @Setter
12    @ToString
```

```

13     public static class Location {
14         private String provinceName;
15         private String countyName;
16     }
17     @Getter
18     @Setter
19     @ToString
20     public static class PersonInfo {
21         private String userName;
22         private String fullName;
23     }
24 }

```

未扁平化之前：

```

1  {
2      "location": {
3          "provinceName": "湖北",
4          "countyName": "武汉"
5      },
6      "personInfo": {
7          "userName": "coder1234",
8          "fullName": "shaungkou"
9      }
10 }

```

使用 `@JsonUnwrapped` 扁平对象之后：

```

1  @Getter
2  @Setter
3  @ToString
4  public class Account {
5      @JsonUnwrapped
6      private Location location;
7      @JsonUnwrapped
8      private PersonInfo personInfo;
9      .....
10 }
11
12 {
13     "provinceName": "湖北",
14     "countyName": "武汉",
15     "userName": "coder1234",
16     "fullName": "shaungkou"
17 }

```

## 11. 测试相关

---

**@ActiveProfiles** 一般作用于测试类上，用于声明生效的 Spring 配置文件。

```
1  @SpringBootTest(webEnvironment = RANDOM_PORT)
2  @ActiveProfiles("test")
3  @Slf4j
4  public abstract class TestBase {
5      .....
6  }
```

**@Test** 声明一个方法为测试方法

**@Transactional** 被声明的测试方法的数据会回滚，避免污染测试数据。

**@WithMockUser** Spring Security 提供的，用来模拟一个真实用户，并且可以赋予权限。

```
1      @Test
2      @Transactional
3      @WithMockUser(username = "user-id-18163138155", authorities = "ROLE_TEACHER")
4      void should_import_student_success() throws Exception {
5          .....
6      }
```