31、Spring Validation（方法参数校验）

（0）参考

- https://segmentfault.com/a/1190000023471742

（1）校验标准

- validation-api接口：是JavaAPI(JSR303)定义的Bean校验的规范标准

（2）校验标准实现

- hibernate validation是对这个标准的实现，并增加了校验注解如@Email、@Length等
- spring validation是对hibernate validation的二次封装，用于支持spring mvc参数自动校验。
- 实际上，不管是RequestBody对象级别的校验、还是@RequstParam/@PathVariable方法级别的校验，最终都是调用Hibernate Validator执行校验，Spring Validation只是做了一层封装。

（3）引入依赖

- boot2.3.x以前，spring-boot-starter-web会自动传入hibernate-validator依赖
- boot2.3.x以后，需要手动引入依赖

复制代码

```java
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.1.Final</version>
</dependency>
```

- @Valid和@Validated注解

区别	@Valid	@Validated
提供者	JSR-303规范	Spring
是否支持分组	不支持	支持
标注位置	METHOD, FIELD, CONSTRUCTOR, PARAMETER, TYPE_USE	TYPE, METHOD, PARAMETER
嵌套校验	支持	不支持

（4）请求参数的形式

- POST、PUT请求，使用requestBody传递参数；
- GET请求，使用RequestParam/PathVariable传递参数。

（5）对@RequestBody修饰的参数进行校验

- 一般使用DTO对象来接收请求参数（等同于VO中的定义的接收参数的对象），只要给DTO对象加上@Validated注解就能实现自动参数校验。
- DTO对象：数据传输对象，Data Transfer Object，用于客户端和服务器之间的交互传输对象，等同于VO对象。
- 如果校验失败，会抛出MethodArgumentNotValidException异常，Spring默认会将其转为400（Bad Request）请求
- 案例演示
  - （1）DTO对象的属性上增加约束注解@NotNull，@Length，@Max，@Min，@Email等
  - （2）Controller方法参数上增加校验注解@Validated或@Valid

笔记目录

```java
@Data
public class UserDTO {

    private Long userId;

    @NotNull
    @Length(min = 2, max = 10)
    private String userName;

    @NotNull
    @Length(min = 6, max = 20)
    private String account;

    @NotNull
    @Length(min = 6, max = 20)
    private String password;
}

@RestController
class UserController{
 @PostMapping("/save")
 public Result saveUser(@RequestBody @Validated UserDTO userDTO)
{
       // 校验通过，才会执行业务逻辑处理
       return Result.ok();
  }
}
```


（6）对@RequestParam/@PathVariable修饰的参数进行校验

- GET请求时，是将一个个参数平铺到方法入参中（超过6个参数还是推荐使用DTO对象接收），这种情况必须在Controller类上增加@Validated注解，并在方法入参上增加约束注解（如@Min等），这样才能实现自动参数校验。
- 如果校验失败，会抛出ConstraintViolationException异常。
- 案例演示

复制代码

```
@RequestMapping("/api/user")
@RestController
@Validated
public class UserController {
    // 路径变量
    @GetMapping("{userId}")
    public Result detail(@PathVariable("userId") @Min(100L) Long userId) {
        // 校验通过，才会执行业务逻辑处理
        UserDTO userDTO = new UserDTO();
        userDTO.setUserId(userId);
        userDTO.setAccount("11111111111111111");
        userDTO.setUserName("xixi");
        userDTO.setAccount("11111111111111111");
        return Result.ok(userDTO);
    }

    // 查询参数
    @GetMapping("getByAccount")
    public Result getByAccount(@Length(min = 6, max = 20) @NotNull String account) {
        // 校验通过，才会执行业务逻辑处理
        UserDTO userDTO = new UserDTO();
        userDTO.setUserId(10000000000000003L);
```

```
        userDTO.setAccount(account);
        userDTO.setUserName("xixi");
        userDTO.setAccount("11111111111111111");
        return Result.ok(userDTO);
    }
}
```

java

（7）统一异常处理

- 前面的两个案例说明，如果校验失败，会抛出MethodArgumentNotValidException或者ConstraintViolationException异常，但在实际项目开发中，通常会用统一异常处理来返回一个更友好的提示，比如我们系统要求无论发送什么异常，http的状态码必须返回200，由业务码去区分系统的异常情况
- 案例演示

复制代码

```
@RestControllerAdvice
public class CommonExceptionHandler {

    //统一拦截抛出的MethodArgumentNotValidException异常
    @ExceptionHandler({MethodArgumentNotValidException.class})
    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    public Result handleMethodArgumentNotValidException(MethodArgumentNotValidException ex) {
        BindingResult bindingResult = ex.getBindingResult();
        StringBuilder sb = new StringBuilder("校验失败:");
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
```

```java
sb.append(fieldError.getField()).append(": ").append(fieldError.getDefaultMessage()).append(", ");
        }
        String msg = sb.toString();
        return Result.fail(BusinessCode.参数校验失败, msg);
    }

    //统一拦截抛出的ConstraintViolationException异常
    @ExceptionHandler({ConstraintViolationException.class})
    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    public Result handleConstraintViolationException(ConstraintViolationException ex) {
       return
           Result.fail(BusinessCode.参数校验失败, ex.getMessage());
    }
}
```

java

（8）分组校验（同一属性不同规则）

- 实际项目中，存在多个方法需要使用同一个DTO类来接收参数，而不同方法的校验规则很可能是不一样的，此时简单地在DTO类的字段上加约束注解无法解决这个问题，因此引出了spring-validation的分组校验的功能，来专门用来解决这类问题。

- 比如：保存一个用户，userId是可空的；修改这个用户，userId不为空并且必须>100L；其它字段校验规则一样，此时需要使用分组校验。
- 案例演示
  - 需要在DTO类中创建不同类型（保存，删除，修改，查询）的分组，然后在DTO类的属性上加上@Min等约束注解，同时在@Min约束注解的groups属性来声明可以使用哪些分组。
  - 在Controller的save方法的DTO参数上，增加@Validated注解并指定Update类型的校验分组
  - 在Controller的update方法的DTO参数上，增加@Validated注解并指定Update类型的校验分组

复制代码

```
@Data
public class UserDTO {
    //userId只在使用Update校验分组时，才自动校验@Min
    @Min(value = 10000000000000000L, groups = Update.class)
    private Long userId;

    //userName在使用Update、Save任一校验分组时，就会去校验@NotNull、@Length
    @NotNull(groups = {Save.class, Update.class})
    @Length(min = 2, max = 10, groups = {Save.class, Update.class})
    private String userName;


   //account在使用Update、Save任一校验分组时，就会去校验@NotNull、@Length
    @NotNull(groups = {Save.class, Update.class})
    @Length(min = 6, max = 20, groups = {Save.class, Update.class})
    private String account;

```

```
    //password在使用Update、Save任一校验分组时，就会去校验@NotNull @Length
    @NotNull(groups = {Save.class, Update.class})
    @Length(min = 6, max = 20, groups = {Save.class, Update.class})
    private String password;


    /**
     * 保存的时候校验分组
     */
    public interface Save {
    }

    /**
     * 更新的时候校验分组
     */
    public interface Update {
    }
}

////////////////////////////////////////////////////////////

class UserController{
  @PostMapping("/save")
  public Result saveUser(@RequestBody @Validated(UserDTO.Save.class) UserDTO userDTO) {
      // 校验通过，才会执行业务逻辑处理
      return Result.ok();
  }

  @PostMapping("/update")
```

```java
    public Result updateUser(@RequestBody
@Validated(UserDTO.Update.class) UserDTO userDTO) {
        // 校验通过，才会执行业务逻辑处理
        return Result.ok();
    }
}
```

java

（9）嵌套参数校验（对象套对象）

- 实际项目中，请求参数是json对象，而且对象中嵌套对象的情况，所以我们的DTO类中的属性不仅仅是基本数据类型、包装类型、String类型，还有可能是一个对象，这种情况下，可以使用嵌套校验。
- 比如：User用户信息中一般也同时携带Job信息，此时的UserDTO中的Job属性必须使用@Vaild注解标记。
- 案例演示
  - 嵌套校验可以和分组校验一块使用。
  - 嵌套的如果是一个对象的集合，嵌套校验会对嵌套集合中的每一个对象进行校验，例如List<Job>字段会对这个list里面的每一个Job对象都进行校验。

复制代码

```java
@Data
public class UserDTO {

    @Min(value = 100L, groups = Update.class)
    private Long userId;


    @NotNull(groups = {Save.class, Update.class})
    @Valid
    private Job job;

```

```java
    @Data
    public static class Job {

        @Min(value = 1, groups = Update.class)
        private Long jobId;

        @NotNull(groups = {Save.class, Update.class})
        @Length(min = 2, max = 10, groups = {Save.class, Update.class})
        private String jobName;

        @NotNull(groups = {Save.class, Update.class})
        @Length(min = 2, max = 10, groups = {Save.class, Update.class})
        private String position;
    }

    /**
     * 保存的时候校验分组
     */
    public interface Save {
    }

    /**
     * 更新的时候校验分组
     */
    public interface Update {
    }
}
```

java

（10）集合参数校验（对象集合）

- 实际项目中，如果请求参数直接就是一个json数组，并希望对数组中的每一个json对象进行校验，我们直接使用java.util.Collection下的List或者Set来接收数据，参数校验并不会生效！我们需要使用自定义List集合来接收参数
- 如果校验失败，会抛出NotReadablePropertyException，同样可以使用统一异常进行处理
- 案例演示

复制代码

```java
public class ValidationList<E> implements List<E> {
    @Delegate // @Delegate是lombok注解
    @Valid // 一定要加@Valid注解
    public List<E> list = new ArrayList<>();

    // 一定要记得重写toString方法
    @Override
    public String toString() {
        return list.toString();
    }
}
```

java

复制代码

```
class UserController{
  @PostMapping("/saveList")
  public Result saveList(@RequestBody
@Validated(UserDTO.Save.class) ValidationList<UserDTO> userList) {
      // 校验通过，才会执行业务逻辑处理
      return Result.ok();
  }
```

```
}
```

java

（11）自定义校验（注解式校验）

- 业务需求复杂，框架的校验规则不满足，需要自定义spring validation的校验规则
- 案例演示：自定义加密id的校验，由数字、a-f字母、32-256长度
  - 直接使用@EncryptId注解，在需要的参数上进行标注，就实现了参数校验

复制代码

```java
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {EncryptIdValidator.class})
public @interface EncryptId {

    // 默认错误消息
    String message() default "加密id格式错误";

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

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

java

```java
public class EncryptIdValidator implements ConstraintValidator<EncryptId, String> {

    private static final Pattern PATTERN
                            = Pattern.compile("^[a-f\\d]{32,256}$");
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        // 不为null才进行校验
        if (value != null) {
            Matcher matcher = PATTERN.matcher(value);
            return matcher.find();
        }
        return true;
    }
}
```

（12）编程式校验

- 注入javax.validation.Validator对象，然后调用其api进行参数校验

```java
@Autowired
private javax.validation.Validator validator;

// 编程式校验
@PostMapping("/saveWithCodingValidate")
public Result saveWithCodingValidate(@RequestBody UserDTO userDTO) {
```

```java
    Set<ConstraintViolation<UserDTO>> validate =
                                validator.validate(userDTO,
UserDTO.Save.class);
    // 如果校验通过，validate为空；否则，validate包含未校验通过项
    if (validate.isEmpty()) {
        // 校验通过，才会执行业务逻辑处理
    } else {
        for (ConstraintViolation<UserDTO> vio : validate) {
            // 校验失败，做其它逻辑
            System.out.println(vio);
        }
    }
    return Result.ok();
}
```

java

（13）快速失败（fail fast）

- spring validation默认是校验完所有的参数，然后才抛出异常，可以通过简单的配置来开启fail fast模式，一旦校验失败就立即返回。
- 案例演示

复制代码

```java
@Bean
public Validator validator() {
    ValidatorFactory validatorFactory =
Validation.byProvider(HibernateValidator.class)
            .configure()
            // 快速失败模式
            .failFast(true)
            .buildValidatorFactory();
```

```java
    return validatorFactory.getValidator();
}
```