参数校验 Validation

背景介绍

JSR303 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(面向注解编程的时代)，就可以在需要校验的时候进行校验了。Java API 规范 (JSR303) 定义了 Bean 校验的标准 validation-api，但没有提供实现。hibernate validation 是对这个规范的实现，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。

> 官方文档:
>
> Hibernate Validator
> https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#preface
>
> Spring Validator
> https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/core.html#validation

引入依赖

非`Springboot`项目中可以引用依赖，并自行调整版本，使用hibernate validation来验证参数。

```
<!--jsr 303-->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
<!-- hibernate validator-->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.2.0.Final</version>
</dependency>
```

`Springboot`项目中可以引用以下依赖。

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

实战操作

对于 web 服务来说，为防止非法参数对业务造成影响，在 Controller 层一定要做参数校验的!

大部分情况下，请求参数分为如下两种形式：

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

requestBody 参数校验

POST、PUT 请求一般会使用 requestBody 传递参数，这种情况下，后端使用 DTO 对象进行接收。只要给 DTO 对象加上 @Validated 注解就能实现自动参数校验。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。

- 在DTO字段上声明约束注解

```
@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;
}
```

- 在方法参数上声明校验注解

```
@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();
}
```

这种情况下，使用 @Valid 和 @Validated 都可以。

requestParam/PathVariable 参数校验

GET 请求一般会使用 requestParam/PathVariable 传参。如果参数比较多 (比如超过 6 个)，还是推荐使用 DTO 对象接收。否则，推荐将一个个参数平铺到方法入参中。在这种情况下，必须在 Controller 类上标注 @Validated 注解，并在入参上声明约束注解 (如 @Min 等)。如果校验失败，会抛出 ConstraintViolationException 异常。

```
@RequestMapping("/api/user")
@RestController
@Validated
public class UserController {
    // 路径变量
    @GetMapping("{userId}")
    public Result detail(@PathVariable("userId") @Min(10000000000000000L) Long
userId) {
        // 校验通过，才会执行业务逻辑处理
```

```
        return Result.ok();
    }

    // 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
        return Result.ok();
    }
}
```

适用@Valid校验，并将校验结果放到BindingResult对象中

@valid校验可以通过BindingResult来获取校验结果，个性化处理，如无特殊处理可以用统一异常处理去对这些异常做处理。

```
@PostMapping("/save")
public String saveUser(@RequestBody @Valid PlanDto planDto, BindingResult
bindingResult) {
    if(bindingResult.hasErrors()){
        for (ObjectError allError : bindingResult.getAllErrors()) {
            return allError.getDefaultMessage();
        }
    }
    // 校验通过，才会执行业务逻辑处理
    return "ok";
}
```

统一异常处理

前面说过，如果无特殊处理可以用统一异常去处理。由于在默认情况下，校验失败会抛出 **MethodArgumentNotValidException** 或者 **ConstraintViolationException** 异常，所以实际项目开发中，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。

```
@RestControllerAdvice
public class CommonExceptionHandler {
    @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()) {

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

    @ExceptionHandler({ConstraintViolationException.class})
    @ResponseStatus(HttpStatus.OK)
```

```
    @ResponseBody
    public Result
handleConstraintViolationException(ConstraintViolationException ex) {
        return Result.fail(BusinessCode.参数校验失败, ex.getMessage());
    }
}
```

进阶使用

分组校验

在实际项目中，可能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。这个时候，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，spring-validation 支持了分组校验的功能，专门用来解决这类问题。

- 约束注解上声明适用的分组信息 groups

```
@Data
public class UserDTO {
    @Min(value = 10000000000000000L, groups = Update.class)
    private Long userId;

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

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

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

- @Validated 注解上指定校验分组

```
@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")
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();
}
```

嵌套校验

前面的示例中，DTO 类里面的字段都是基本数据类型和 String 类型。但是实际场景中，有可能某个字段也是一个对象，这种情况下，可以使用嵌套校验。需要注意的是，此时 DTO 类的对应字段必须标记 **@Valid** 注解。

```
@Data
public class UserDTO {
    @Min(value = 10000000000000000L, groups = Update.class)
    private Long userId;

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

    @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 {
    }
}
```

集合校验

如果请求体直接传递了 json 数组给后台，并希望对数组中的每一项都进行参数校验。此时，如果我们直接使用 java.util.Collection 下的 list 或者 set 来接收数据，参数校验并不会生效！我们可以使用自定义 list 集合来接收参数：

- 包装 List 类型，并声明 @Valid 注解

```
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();
    }
}
```

@Delegate 注解受 lombok 版本限制，1.18.6 以上版本可支持。如果校验不通过，会抛出 NotReadablePropertyException，同样可以使用统一异常进行处理。

- 比如，我们需要一次性保存多个 User 对象，Controller 层的方法可以这么写：

```
@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();
}
```

自定义校验

业务需求总是比框架提供的这些简单校验要复杂的多，我们可以自定义校验来满足我们的需求。

自定义 spring validation 非常简单，假设我们自定义加密 id（由数字或者 a-f 的字母组成，32-256 长度）校验，主要分为两步：

- 自定义约束注解

```
@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 {};
}
```

- 实现 ConstraintValidator 接口编写约束校验器

```
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;
    }
}
```

这样我们就可以使用 @EncryptId 进行参数校验了！

编程式校验

上面的示例都是基于注解来实现自动校验的，在某些情况下，我们可能希望以编程方式调用验证。这个时候可以注入 javax.validation.Validator 对象，然后再调用其 api。

```
@Autowired
private javax.validation.Validator globalValidator;

// 编程式校验
@PostMapping("/saveWithCodingValidate")
public Result saveWithCodingValidate(@RequestBody UserDTO userDTO) {
    Set<ConstraintViolation<UserDTO>> validate =
globalValidator.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> userDTOConstraintViolation : validate)
{
            // 校验失败，做其它逻辑
            System.out.println(userDTOConstraintViolation);
        }
    }
    return Result.ok();
}
```

快速失败 (Fail Fast)

Spring Validation 默认会校验完所有字段，然后才抛出异常。可以通过一些简单的配置，开启 Fali Fast 模式，一旦校验失败就立即返回。

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

补充

@Valid 和 @Validated 区别

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

常用注解说明

验证注解	验证的数据类型	说明
@AssertFalse	Boolean,boolean	验证注解的元素值是false
@AssertTrue	Boolean,boolean	验证注解的元素值是true
@NotNull	任意类型	验证注解的元素值不是null
@Null	任意类型	验证注解的元素值是null
@Min(value=值)	BigDecimal，BigInteger, byte,short, int, long，等任何Number或CharSequence（存储的是数字）子类型	验证注解的元素值大于等于@Min指定的value值
@Max（value=值)	和@Min要求一样	验证注解的元素值小于等于@Max指定的value值
@DecimalMin(value=值)	和@Min要求一样	验证注解的元素值大于等于@ DecimalMin指定的value值
@DecimalMax(value=值)	和@Min要求一样	验证注解的元素值小于等于@ DecimalMax指定的value值
@Digits(integer=整数位数, fraction=小数位数)	和@Min要求一样	验证注解的元素值的整数位数和小数位数上限
@Size(min=下限, max=上限)	字符串、Collection、Map、数组等	验证注解的元素值的在min和max（包含）指定区间之内，如字符长度、集合大小
@Past	java.util.Date,java.util.Calendar;Joda Time类库的日期类型	验证注解的元素值（日期类型）比当前时间早
@Future	与@Past要求一样	验证注解的元素值（日期类型）比当前时间晚
@NotBlank	CharSequence子类型	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的首位空格
@Length(min=下限, max=上限)	CharSequence子类型	验证注解的元素值长度在min和max区间内

验证注解	验证的数据类型	说明
@NotEmpty	CharSequence子类型、Collection、Map、数组	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）
@Range(min=最小值,max=最大值)	BigDecimal,BigInteger,CharSequence,byte, short, int, long等原子类型和包装类型	验证注解的元素值在最小值和最大值之间
@Email(regexp=正则表达式,flag=标志的模式)	CharSequence子类型（如String）	验证注解的元素值是Email，也可以通过regexp和flag指定自定义的email格式
@Pattern(regexp=正则表达式,flag=标志的模式)	String，任何CharSequence的子类型	验证注解的元素值与指定的正则表达式匹配
@Valid	任何非原子类型	指定递归验证关联的对象如用户对象中有个地址对象属性，如果想在验证用户对象时一起验证地址对象的话，在地址对象上加@Valid注解即可级联验证

> 注意：1.@NotNull：不能为**null**，但可以为empty，如 ""," "," "。
> 2.@NotEmpty：不能为**null**，而且长度必须大于0，如 " "," "。
> 3.@NotBlank：只能作用在String上，不能为**null**，而且调用trim()后，长度必须大于0，如"test"。