Spring Validation最佳实践及其实现原理，参数校验没那么简单！


伍六七 发布于 2020-08-03

之前也写过一篇关于Spring Validation使用的文章，不过自我感觉还是浮于表面，本次打算彻底搞懂Spring Validation。本文会详细介绍Spring Validation各种场景下的最佳实践及其实现原理，死磕到底！
项目源码：spring-validation

简单使用

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。接下来，我们以spring-boot项目为例，介绍Spring Validation的使用。

引入依赖

如果spring-boot版本小于2.3.x，spring-boot-starter-web会自动传入hibernate-validator依赖。如果spring-boot版本大于2.3.x，则需要手动引入依赖：

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

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：

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

下面我们简单介绍下requestBody和requestParam/PathVariable的参数校验实战！

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。**比如，有一个保存User的接口，要求userName长度是2-10，account和password字段长度是6-20。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。

> **DTO表示数据传输对象（Data Transfer Object），用于服务器和客户端之间交互传输使用的。**在spring-web项目中可以表示用于接收请求参数的Bean对象。

- **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。代码示例如下：

```
@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
        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);
```

统一异常处理

前面说过，如果校验失败，会抛出MethodArgumentNotValidException或者ConstraintViolationException异常。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。

```
@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(": ").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。还是上面的例子，比如保存`User`的时候，`UserId`是可空的，但是更新`User`的时候，`UserId`的值必须`>=10000000000000000L`；其它字段的校验规则在两种情况下一样。这个时候使用**分组校验**的代码示例如下：

- **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;

    @NotNull(groups = {Save.class, Update.class})
    @Length(min = 6, max = 20, groups = {Save.class, Update.class})
    private String account;

    @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
```

- **`@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，这种情况先，可以使用嵌套校验。比如，上面保存User信息的时候同时还带有Job信息。需要注意的是，**此时DTO类的对应字段必须标记@Valid注解。**

```
    @NotNull(groups = {Save.class, Update.class})
    @Length(min = 6, max = 20, groups = {Save.class, Update.class})
    private String password;

    @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
     */
```

嵌套校验可以结合分组校验一起使用。还有就是嵌套集合校验会对集合里面的每一项都进行校验，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。

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();
}
```

@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

实现原理

requestBody参数校验实现原理

在spring-mvc中，RequestResponseBodyMethodProcessor是用于解析@RequestBody标注的参数以及处理@ResponseBody标注方法的返回值的。显然，执行参数校验的逻辑肯定就在解析参数的方法resolveArgument()中：

```
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodProcessor {
    @Override
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
                                  NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory)
throws Exception {

        parameter = parameter.nestedIfOptional();
        //将请求数据封装到DTO对象中
        Object arg = readWithMessageConverters(webRequest, parameter, parameter.getNestedGenericParameterType());
        String name = Conventions.getVariableNameForParameter(parameter);

        if (binderFactory != null) {
            WebDataBinder binder = binderFactory.createBinder(webRequest, arg, name);
            if (arg != null) {
                // 执行数据校验
                validateIfApplicable(binder, parameter);
                if (binder.getBindingResult().hasErrors() && isBindExceptionRequired(binder, parameter)) {
                    throw new MethodArgumentNotValidException(parameter, binder.getBindingResult());
                }
            }
            if (mavContainer != null) {
                mavContainer.addAttribute(BindingResult.MODEL_KEY_PREFIX + name, binder.getBindingResult());
            }
        }
        return adaptArgumentIfNecessary(arg, parameter);
    }
```

可以看到，`resolveArgument()`调用了`validateIfApplicable()`进行参数校验。

```
protected void validateIfApplicable(WebDataBinder binder, MethodParameter parameter) {
    // 获取参数注解，比如@RequestBody、@Valid、@Validated
    Annotation[] annotations = parameter.getParameterAnnotations();
    for (Annotation ann : annotations) {
        // 先尝试获取@Validated注解
        Validated validatedAnn = AnnotationUtils.getAnnotation(ann, Validated.class);
        //如果直接标注了@Validated，那么直接开启校验。
        //如果没有，那么判断参数前是否有Valid起头的注解。
        if (validatedAnn != null || ann.annotationType().getSimpleName().startsWith("Valid")) {
            Object hints = (validatedAnn != null ? validatedAnn.value() : AnnotationUtils.getValue(ann));
            Object[] validationHints = (hints instanceof Object[] ? (Object[]) hints : new Object[] {hints});
            //执行校验
            binder.validate(validationHints);
            break;
        }
    }
}
```

看到这里，大家应该能明白为什么这种场景下`@Validated`、`@Valid`两个注解可以混用。我们接下来继续看`WebDataBinder.validate()`实现。

```
@Override
public void validate(Object target, Errors errors, Object... validationHints) {
    if (this.targetValidator != null) {
        processConstraintViolations(
            //此处调用Hibernate Validator执行真正的校验
            this.targetValidator.validate(target, asValidationGroups(validationHints)), errors);
    }
}
```

最终发现底层最终还是调用了`Hibernate Validator`进行真正的校验处理。

方法级别的参数校验实现原理

上面提到的将参数一个个平铺到方法参数中，然后在每个参数前面声明约束注解的校验方式，就是方法级别的参数校验。实际上，这种方式可用于任何`Spring Bean`的方法上，比如`Controller`/`Service`等。**其底层实现原理就是`AOP`，具体来说是通过`MethodValidationPostProcessor`动态注册`AOP`切面，然后使用`MethodValidationInterceptor`对切点方法织入增强。**

```
public class MethodValidationPostProcessor extends AbstractBeanFactoryAwareAdvisingPostProcessorimplements
InitializingBean {
    @Override
    public void afterPropertiesSet() {
        //为所有`@Validated`标注的Bean创建切面
        Pointcut pointcut = new AnnotationMatchingPointcut(this.validatedAnnotationType, true);
        //创建Advisor进行增强
        this.advisor = new DefaultPointcutAdvisor(pointcut, createMethodValidationAdvice(this.validator));
    }

    //创建Advice，本质就是一个方法拦截器
    protected Advice createMethodValidationAdvice(@Nullable Validator validator) {
        return (validator != null ? new MethodValidationInterceptor(validator) : new MethodValidationInterceptor());
    }
}
```

接着看一下MethodValidationInterceptor：

```
public class MethodValidationInterceptor implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        //无需增强的方法，直接跳过
        if (isFactoryBeanMetadataMethod(invocation.getMethod())) {
            return invocation.proceed();
        }
        //获取分组信息
        Class<?>[] groups = determineValidationGroups(invocation);
        ExecutableValidator execVal = this.validator.forExecutables();
        Method methodToValidate = invocation.getMethod();
        Set<ConstraintViolation<Object>> result;
        try {
            //方法入参校验，最终还是委托给Hibernate Validator来校验
            result = execVal.validateParameters(
                invocation.getThis(), methodToValidate, invocation.getArguments(), groups);
        }
        catch (IllegalArgumentException ex) {
            ...
        }
        //有异常直接抛出
        if (!result.isEmpty()) {
            throw new ConstraintViolationException(result);
        }
        //真正的方法调用
        Object returnValue = invocation.proceed();
```

实际上，不管是requestBody参数校验还是方法级别的校验，最终都是调用Hibernate Validator执行校验，Spring Validation只是做了一层封装。

java spring springboot

伍六七

热爱技术，分享技术

23 声望 **1** 粉丝

关注作者