2020-Spring Boot Validation 看这篇就够了

作者：晨光

众所周知，在实际的开发过程中无法避免的需要对输入参数进行校验，`不为空、最大值、最小值`等，到处写校验的代码很繁琐也让代码看起来很凌乱。Jakarta有一个项目Bean Validation，正是为解决这个问题而诞生的。而Spring Boot将它很好的集成进来，香。

但是在使用过程中会有一些坑，建议在使用之前先系统地学习下，避免采坑。这篇文章就想记录下Spring Boot Validation的使用方式和注意点。

代码

文章涉及的所有代码都在GitHub上。

引入Validation

Spring为Validation提供了一个starter，`spring-boot-starter-validation`

- Maven方式引入

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

- Gradle方式引入

```
implementation('org.springframework.boot:spring-boot-starter-validation')
```

正常情况下不需要添加版本号，Spring Boot 框架会在parent pom中指定版本，如果需要添加指定版本的jar包，可以在这里查找。

当然你也可以使用其他的Validation实现，比如`[hibernate validator](https://search.maven.org/search?q=g:org.hibernate.validator%20AND%20a:hibernate-validator&core=gav)`，听说他是目前使用最广泛的validation实现。如果引入了`hibernate validator`，就可以不用引入`spring-boot-starter-validation`实现。

Bean Validation 在Spring MVC Controller上的应用

`Bean Validation`通过Annotation的方式来使用，如：`@NotEmpty @Max @Min`等，所有的注解可以在这里找到。直接看例子：

验证RequestParam和PathVariables

```
@RestController
@Validated
@RequestMapping("/validation")
```

```
public class ValidationTestController {

    @GetMapping("/example1")
    String validateExample1(@NotEmpty(message = "不可以为空") @RequestParam String param) {
        return "valid";
    }

    @GetMapping("/example2/{id}")
    String validateExample2(@Min(value = 10, message = "ID不能小于10") @PathVariable String id) {
        return "valid";
    }
}
```

- 这里`@NotEmpty`要求`param`参数不能为空，包括null和空字符串。`message`为报错的时候的提示信息。
- `@Min`要求id不能小于10，提示信息`message`如果不定义则使用默认提示信息，默认提示信息是什么样的，在文章的后面会介绍。
- 我们必须在Controller的类级别添加`@Validated`，告诉Spring框架触发参数校验。
- 如果校验失败，Spring框架会抛出异常：`ConstraintViolationException`Spring没有默认对这个类型的异常处理器，所以会导致请求返回500错误。如果想要修改异常是的Http staus，我们可以添加一个自定义异常处理：

```
@RestController
@Validated
@RequestMapping("/validation")
public class ValidationTestController {

    // ...

    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @GetMapping("/example3/{id}")
    String validateExample3(@Min(value = 10, message = "ID不能小于10") @PathVariable String id) {
        return "valid";
    }
}
```

- 文章的后面我们会讨论如何更优雅地处理错误，并放回给用户友好的提示。

验证RequestBody合法性

当Controller方法需要接收的参数很多时，我们需要通过一个Pojo来接收，Spring Boot Validation也提供了这种情况下的验证方式。比如我们的Pojo如下：

```
@Data
public class Student {

    private int id;

    @NotEmpty(message = "姓名不能为空")
    private String name;

```

```
    @Min(value = 3, message = "3岁以下小朋友就别过来")
    @Max(value = 60, message = "60岁以上老爷爷也别过来")
    private int age;

    @Size(min = 1, message = "至少要有一个喜好")
    private List<String> favorite;
}
```

- 姓名不能为空
- 年龄在3-60岁
- 至少要有一个喜好

下面是使用这个内容来接收数据的示例：

```
@RestController
@Validated
@RequestMapping("/validation")
public class ValidationTestController {

    // ....

    @PostMapping("/example3/")
    String validateExample2(@RequestBody @Valid Student student) {
        return "valid";
    }

    // ...
}
```

我们需要在这种参数前面添加`@RequestBody`注解来要求Spring接收`student`的数据，同时添加`@Valid`注解要求Spring验证`student`中的数据。

> 如果这里的student中有组合对象，比如学校school，而学校又是一个对象，包含地址，名称等属性，要验证这种组合对象时，也要在学校的对象上添加@Valid参数。

```
@Data
public class Student {

    // ...

    @Valid
    private School school;

    @NotEmpty(message = "姓名不能为空")
    private String name;

        // ...
}
```

测试：

```
### Send POST request with json body
POST http://localhost:8080/validation/example3/
Content-Type: application/json

{
  "name": "",
  "age": "1",
  "favorite": ["reading"]
}
```

返回结果：


```
{
  "timestamp": "2020-08-21T02:48:11.945+00:00",
  "status": 400,
  "error": "Bad Request",
  "trace": "org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument ...",
  "message": "Validation failed for object='student'. Error count: 3",
  "errors": [
    {
      "codes": [
        "Min.student.age",
        "Min.age",
        "Min.int",
        "Min"
      ],
      "arguments": [
        {
          "codes": [
            "student.age",
            "age"
          ],
          "arguments": null,
          "defaultMessage": "age",
          "code": "age"
        },
        3
      ],
      "defaultMessage": "3岁以下小朋友就别过来",
      "objectName": "student",
      "field": "age",
      "rejectedValue": 1,
      "bindingFailure": false,
      "code": "Min"
    },
    {
      "codes": [
        "NotEmpty.student.name",
        "NotEmpty.name",
        "NotEmpty.java.lang.String",
        "NotEmpty"
      ],
      "arguments": [
        {
          "codes": [
```

```
            "student.name",
            "name"
          ],
          "arguments": null,
          "defaultMessage": "name",
          "code": "name"
        }
      ],
      "defaultMessage": "姓名不能为空",
      "objectName": "student",
      "field": "name",
      "rejectedValue": "",
      "bindingFailure": false,
      "code": "NotEmpty"
    },
    {
      "codes": [
        "Size.student.favorite",
        "Size.favorite",
        "Size.java.util.List",
        "Size"
      ],
      "arguments": [
        {
          "codes": [
            "student.favorite",
            "favorite"
          ],
          "arguments": null,
          "defaultMessage": "favorite",
          "code": "favorite"
        },
        2147483647,
        2
      ],
      "defaultMessage": "至少要有二个喜好",
      "objectName": "student",
      "field": "favorite",
      "rejectedValue": [
        "reading"
      ],
      "bindingFailure": false,
      "code": "Size"
    }
  ],
  "path": "/validation/example3/"
}
```


我们可以看到验证失败的情况下回返回`MethodArgumentNotValidException`异常，但是Http状态码是400，缩影Spring框架默认处理了这种异常，告诉用户这是一个`Bad Request`而不是一个系统错误。

在Service的方法上应用Validation

在Service中应用需要两个注解配合`@Validated`和`@Valid`，看例子：

```
@Service
@Validated
public class StudentService {

    public void addStudent(@Valid Student student) {
        // add student
    }

}
```

```
@RestController
@Validated
@RequestMapping("/validation")
public class ValidationTestController {

    private final StudentService studentService;

    // ...

    @GetMapping("/example4/")
    String validateExample4() {
        Student student =
Student.builder().name("").age(2).favorite(List.of("reading")).build();
        studentService.addStudent(student);
        return "valid";
    }
}
```

验证：

```
### Service方法的参数验证
GET http://localhost:8080/validation/example4/
Accept: application/json
```

后台会抛出异常：

```
addStudent.student.age: 3岁以下小朋友就别过来, addStudent.student.name: 姓名不能为空, addStudent.student.favorite: 至少要有二个喜好
javax.validation.ConstraintViolationException: addStudent.student.age: 3岁以下小朋友就别过来, addStudent.student.name: 姓名不能为空,
addStudent.student.favorite: 至少要有二个喜好
...
```

我们看到这里也抛出了`ConstraintViolationException`异常，并给出了自定义的提示信息。

持久层的参数验证

我们不建议等到持久层才去做参数验证，应该将参数验证尽量前移保证后续逻辑的正确性和程序的健壮，但是`Validation`也支持在这一层去验证你的参数。使用方式与前面提到的POJO的方式类似。同样也抛出`ConstraintViolationException`异常，这里不做赘述，请小伙伴们自己去试一试。

实现自定义的验证器

当`Bean Validation`提供的默认验证器不能满足我们需求的时候，我们就会考虑自己写验证器，恰好`Validation`提供了这个扩展可以优雅地实现自己的验证逻辑并在项目的各个地方使用，我们来看一下。

我们给学生实体添加一个属性头像地址`avataUrl`它是一个URL，假设必须以`http`或者`https`开头，结尾必须包含`jpg`或者`png`。我们来定义个自定义的校验器：

```
@Target({FIELD})
@Retention(RUNTIME)
@Constraint(validatedBy = AvatarValidator.class)
@Documented
public @interface AvatarUrl {

    String message() default "{AvatarUrl.invalid}";

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

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

- `message`：用来定义验证不通过以后的消息，但是真正的消息，而是指向一个配置文件`ValidationMessages.properties`的key，这样就可以解决国际化问题，也可以实现默认提示消息的自定义。
- `group`：用来定义在什么场景下触发验证，后面我们会具体介绍。
- `payload`：可以通过此属性来给约束条件指定严重级别，这个属性并不被API自身所使用，如果指定了`payload`，在校验完成后可以通过调用`ConstraintViolation.getConstraintDescriptor().getPayload()`来得到之前指定到错误级别了,并且可以根据这个信息来决定接下来到行为。
- 其中`@Constraint`注解，指定了这个验证注解的验证器实现。

验证器实现示例：

```
public class AvatarValidator implements ConstraintValidator<AvatarUrl,
String> {

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
{
        if (StringUtils.isEmpty(value)) {
            return false;
        }
        if (!value.startsWith("http://") && !value.startsWith("https://")) {
            return false;
        }
        if (!value.endsWith(".jpg") && !value.endsWith(".png")) {
            return false;
        }
        return true;
    }
}
```

之后我们就可以使用`@AvaterUrl`了：

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    // ...

        @AvatarUrl(message = "头像链接格式不正确")
    private String avatar;
}
```

通过编程的方式来调用验证器

上面我们都是通过注解的方式实现验证的，我们也可以通过代码来主动调用验证器做验证：

```java
class ProgrammaticallyValidatingService {
    void validateStudent(Student student) {
        ValidatorFactory factory =
Validation.buildDefaultValidatorFactory();
        Validator validator = factory.getValidator();
        Set<ConstraintViolation<Student>> violations =
validator.validate(student);
        if (!violations.isEmpty()) {
            throw new ConstraintViolationException(violations);
        }
    }
}
```

以上是不依赖Spring容器，自己创建`validator`对象的方式，但其实Spring 容器给我们已经准备好了内置的`validator`对象，我们可以直接注入使用：

```java
@Service
public class SpringBasedProgrammaticallyValidatingService {

    private Validator validator;

    SpringBasedProgrammaticallyValidatingService(Validator validator) {
        this.validator = validator;
    }

    void validateStudent(Student student) {
        Set<ConstraintViolation<Student>> violations =
validator.validate(student);
        if (!violations.isEmpty()) {
            throw new ConstraintViolationException(violations);
        }
    }
}
```

一个对象在不同的场景使用不同的校验规则

之前我们提到`group`这个属性，就是用来解决这种问题的，比如在插入数据和更新数据时，我们对数据的校验试不一样的，相当于两个场景：

- 插入场景，我们定义一个Group：`interface OnCreate {}`
- 更新场景，我们第一一个Group：`interface OnUpdate {}`

```
public interface Views {
    interface OnCreate {};
    interface OnUpdate {};
}
```

那么我们就可以这么用：

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    @Null(groups = Views.OnCreate.class)
    @NotNull(groups = Views.OnUpdate.class)
    private int id;

    // ...
}
```

在Service中也要添加上场景的`Group`

```
@Service
@Validated
public class StudentService {

    @Validated(Views.OnCreate.class)
    public void addStudent(@Valid Student student) {
        // add student
    }

    @Validated(Views.OnUpdate.class)
    public void updateStudent(@Valid Student student) {
        // update student
    }
}
```

优雅地处理验证失败以后的错误

前文提到校验失败后应用会抛出400或者500异常，我们就得在一个地方接住他们，并且 友好地返回给用户。

- 首先我们定义一个返回用户时的结构对象：

```java
@Data
public class ValidateResult {
    private final String fieldName;
    private final String message;
}

@Data
public class ValidationErrorResponse {
    private List<ValidateResult> validateResults = new ArrayList<>();
}
```

- 再创建一个`ControllerAdvice`类来全局处理校验错误。

```java
@ControllerAdvice
class ErrorHandlingControllerAdvice {

    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    ValidationErrorResponse onConstraintValidationException(ConstraintViolationException e) {
        ValidationErrorResponse error = new ValidationErrorResponse();
        for (ConstraintViolation violation : e.getConstraintViolations()) {
            error.getValidateResults()
                .add(new ValidateResult(violation.getPropertyPath().toString(), violation.getMessage()));
        }
        return error;
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    ValidationErrorResponse onMethodArgumentNotValidException(MethodArgumentNotValidException e) {
        ValidationErrorResponse error = new ValidationErrorResponse();
        for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {
            error.getValidateResults().add(new ValidateResult(fieldError.getField(), fieldError.getDefaultMessage()));
        }
        return error;
    }
}
```

我们在来测试下之前的`example3`,看我们得到了什么：

```http
POST http://localhost:8080/validation/example3/

HTTP/1.1 400
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 21 Aug 2020 12:23:55 GMT
Connection: close

```

```
{
  "validateResults": [
    {
      "fieldName": "age",
      "message": "3岁以下小朋友就别过来"
    },
    {
      "fieldName": "name",
      "message": "姓名不能为空"
    },
    {
      "fieldName": "favorite",
      "message": "至少要有二个喜好"
    },
    {
      "fieldName": "avatar",
      "message": "头像链接格式不正确"
    }
  ]
}
```

总结

我们一起学习了`Spring Boot`提供的`Validation`的功能，包括在`Controller`层，`Service`层，`DAO`层的使用方式，也学习了通过编程的方式使用验证功能，还学习了基于场景（`Group`）的校验需要怎么做，最后我们通过统一的错误处理给用户以友好的校验结果提醒，怎么样，你学会了么。