链滴

Spring Boot 参数校验 Validation 入门

作者：JayGao
原文链接：https://ld246.com/article/1650948173810
来源网站：链滴

摘要: 原创出处 <a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.iocoder.
n%2FSpring-Boot%2FValidation%2F" target="_blank" rel="nofollow ugc">http://www.iocoder
cn/Spring-Boot/Validation/ 「芋道源码」欢迎转载，保留摘要，谢谢！

概述

<ol start="2">
注解

<ol start="3">
快速入门

<ol start="4">
处理校验异常

<ol start="5">
自定义约束

<ol start="6">
分组校验

<ol start="7">
手动校验

<ol start="8">
国际化 i18n

<ol start="666">
彩蛋

<hr>
<hr>

<blockquote>
本文在提供完整代码示例，可见 https://gihub.com/YunaiV/SpringBoot-Labs 的 lab-22 目录。
原创不易，给点个 Star</> 嘿，一起冲鸭！
</blockquote>
<h2 id="1--概述">1. 概述</h2>
<blockquote>
在想标题的时候，到底应该叫数据 校验，还是参数验时，我纠结了，而且非常。
最后，考虑参数 校验更贴近我们的理解，就选择了它。实际更合适的叫法还是数据校验 。
文头艿艿瞎哔哔了一些碎碎念，嫌弃的胖友，可以跳往 「3. 快速入门」 。
</blockquote>
当我们想提供可靠的 API 接口，对参数的校验，以保证最终数据入库的正确性，是必可少 的活。例如说，用户注册时，会校验手机格式的正确性，密码非弱密码。
可惜的是，在翻开自己的项目的时候，会发现大量的 API 接口，我们并没有添加相应的参数校验而是把这个活交给调用方（例如说前端）来完成。:smiling_imp: 甚至在艿艿接触过的后端开发中，为这是前端的活，简直了！
世界比我们想象中的不安全，可能有“黑客”会绕过浏览器，直接使用 HTTP 工具，模拟请求向端 API 接口传入违法的参数，以达到它们“不可告人”的目的。
又或者前端开发小哥，不小心漏做了一些 API 接口调用时的参数校验，结果导致用户提交了大量正确的数据到后端 API 接口，并且这些数据成功 入库了。这个时候，你是会甩给前端小哥，还是怒喷测试小姐姐验收不到位呢？
我相信，很多时候并不是我们不想添加，而是没有统一方便的方式，让我们快速的添加实现参数验的功能。毕竟，比起枯燥的 CRUD 来说，它更枯燥。例如说，还是拿用户注册的接口，校验手机密码这两个参数，可能就要消耗掉小 10 行的代码。更不要说，管理后台创建商品这种参数贼多的接。
:smiling_imp: 世界上大多数碰到的困难，大多已经有了解决方案，特别是软件开发。实际上，Ja a 早在 2009 年就提出了 Bean Validation 规，并且已经历经 JSR303、JSR349、JSR380 三次标准的置顶，发展到了 2.0 。/p>
<blockquote>
FROM https://beanvalidation.org/specificati n/
Bean Validation 1.0 ： Bean Validation 1.0 (JSR 303) was the first version of Java's standard for object v lidation. It was released in 2009 and is part of Java EE 6. You can learn more about Bean Valid tion 1.0 here (specification text, API docs etc).
Bean Validation 1.1 ： Bean Validation 1.1 (JSR 349) was finished in 2013 and is part of Java EE 7. Its mai contributions are method-level validation, integration with CDI, group conversion and some more. You can learn more about Bean Validation 1.1 <a href="https://ld246.com/forward?go

o=https%3A%2F%2Fbeanvalidation.org%2F1.1%2F" target="_blank" rel="nofollow ugc">her
 (specification text, full change log, API docs etc).
Bean Validation 2.0 ： Bean Validation 2.0 (<a href="https://ld246.co
/forward?goto=https%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D380" target="
blank" rel="nofollow ugc">JSR 380) was finished in August 2017.
It's part of Java EE 8 (but can of course be used with plain Java SE as the previous releases
.
You can learn more about Bean Validation 2.0 <a href="https://ld246.com/forward?goto
https%3A%2F%2Fbeanvalidation.org%2F2.0%2F" target="_blank" rel="nofollow ugc">here</
> (specification text, full change log, API docs etc).
</blockquote>
Bean Validation 和我们很久以前学习过的 JPA 一样，只提供规范，不提供具体的实现。
<blockquote>
艿艿：对 JPA 不了的胖友，可以看看 <a href="https://ld246.com/forward?goto=http%3A%
F%2Fwww.iocoder.cn%2FSpring-Boot%2FJPA%2F%3Fself" target="_blank" rel="nofollow ugc
>《芋道 Spring Boot JPA 入门》 一文。
</blockquote>

在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fmvnrepository.com%2Fa
tifact%2Fjavax.validation%2Fvalidation-api" target="_blank" rel="nofollow ugc">Bean Validat
on API 中，定义了 Bean Validation 相关的接口，并没有具体实现。
在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Feclipse-e
4j%2Fbeanvalidation-api%2Ftree%2Fmaster%2Fsrc%2Fmain%2Fjava%2Fjavax%2Fvalidation%
Fconstraints" target="_blank" rel="nofollow ugc"><code>javax.validation.constraints</code
 包下，定义了一系列的校验注解。例如说，<a href="https://ld246.com/forward?goto=http
%3A%2F%2Fgithub.com%2Feclipse-ee4j%2Fbeanvalidation-api%2Fblob%2Fmaster%2Fsrc%2
main%2Fjava%2Fjavax%2Fvalidation%2Fconstraints%2FNotNull.java" target="_blank" rel="no
ollow ugc"><code>@NotNull</code>、<a href="https://ld246.com/forward?goto=htt
s%3A%2F%2Fgithub.com%2Feclipse-ee4j%2Fbeanvalidation-api%2Fblob%2Fmaster%2Fsrc%
Fmain%2Fjava%2Fjavax%2Fvalidation%2Fconstraints%2FNotEmpty.java" target="_blank" rel=
nofollow ugc"><code>@NotEmpty</code> 。

实现 Bean Validation 规范的数据校验框架，主要有：

<a href="https://ld246.com/forward?goto=https%3A%2F%2Fhibernate.org%2Fvalidator
2F" target="_blank" rel="nofollow ugc">Hibernate Validator
<blockquote>
不要以为 Hibernate 仅仅是一个 ORM 框架，这只是它的 <a href="https://ld246.com/forwar
?goto=https%3A%2F%2Fhibernate.org%2Form" target="_blank" rel="nofollow ugc">Hiberna
e ORM 所提供的。
Hibernate 可是打着“Everything data”口号的，它还提供了 <a href="https://ld246.com/fo
ward?goto=https%3A%2F%2Fhibernate.org%2F" target="_blank" rel="nofollow ugc">Hibern
te Search、<a href="https://ld246.com/forward?goto=https%3A%2F%2Fhibernate.org%
Fogm" target="_blank" rel="nofollow ugc">Hibernate OGM 等等解决方案的。:smiling_i
p:
所以，女朋友也是 data ，我们来 <code>new</code> 一个就好，不需要找。
</blockquote>

~~:chicken: 咳咳咳，突然想不起来还有个叫啥了，以后补充吧。啪啪打脸的疼~~ ~ <a
ref="https://ld246.com/forward?goto=https%3A%2F%2Fbval.apache.org%2F" target="_blank
rel="nofollow ugc">Apache BVal

绝大多数情况下，也就 99.99% 吧，我们采用 Hibernate Validator 。</p

但是，我们在使用 Spring 的项目中，因为 <a href="https://ld246.com/forward?goto=https
3A%2F%2Fgithub.com%2Fspring-projects%2Fspring-framework%2Ftree%2Fmaster%2Fspring
context%2Fsrc%2Fmain%2Fjava%2Forg%2Fspringframework%2Fvalidation" target="_blank" r
l="nofollow ugc">Spring Validation 提供了对 Bean Validation 的内置封装支持，可以使用
a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Fspring-projects%
Fspring-framework%2Fblob%2Fmaster%2Fspring-context%2Fsrc%2Fmain%2Fjava%2Forg%2F
pringframework%2Fvalidation%2Fannotation%2FValidated.java" target="_blank" rel="nofoll
w ugc"><code>@Validated</code> 注解，实现声明式校验 ，而无需
接调用 Bean Validation 提供的 API 方法。而在实现原理上，也是基于 Spring AOP 拦截，实现校
相关的操作。
<blockquote>
友情提示：这一点，类似 Spring Transaction 事务，通过 <code>@Transactional</code>
解，实现声明式事务。
</blockquote>
而在 Spring Validation 内部，最终还是调用不同的 Bean Validation 的实现框架。例如说，Hib
rnate Validator 。
下面，让我们开始遨游，在 Spring Boot 中，如何实现参数校验。
<h2 id="2--注解">2. 注解</h2>
在开始入门之前，我们先了解下本文可能会涉及到的注解。
<h2 id="2-1-Bean-Validation-定义的约束注解">2.1 Bean Validation 定义的约束注解</h2>
<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Feclipse-ee4
%2Fbeanvalidation-api%2Ftree%2Fmaster%2Fsrc%2Fmain%2Fjava%2Fjavax%2Fvalidation%2F
onstraints" target="_blank" rel="nofollow ugc"><code>javax.validation.constraints</code><
a> 包下，定义了一系列的约束(constraint)注解。如下：
<blockquote>
参考 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.ibm.com%2Fdeve
operworks%2Fcn%2Fjava%2Fj-lo-jsr303%2Findex.html" target="_blank" rel="nofollow ugc"
《JSR 303 - Bean Validation 介绍及最佳实践》 博客。
一共 22 个注解，快速略过即可。
</blockquote>

空和非空检查

<code>@NotBlank</code> ：只能用于字符串不为 <code>null</code> ，并且字符串 <co
e>#trim()</code> 以后 length 要大于 0 。
<code>@NotEmpty</code> ：集合对象的元素不为 0 ，即集合不为空，也可以用于字符串
为 <code>null</code> 。
<code>@NotNull</code> ：不能为 <code>null</code> 。
<code>@Null</code> ：必须为 <code>null</code> 。

数值检查

<code>@DecimalMax(value)</code> ：被注释的元素必须是一个数字，其值必须小于等于
定的最大值。
<code>@DecimalMin(value)</code> ：被注释的元素必须是一个数字，其值必须大于等于指
的最小值。
<code>@Digits(integer, fraction)</code> ：被注释的元素必须是一个数字，其值必须在可
受的范围内。
<code>@Positive</code> ：判断正数。
<code>@PositiveOrZero</code> ：判断正数或 0 。
<code>@Max(value)</code> ：该字段的值只能小于或等于该值。
<code>@Min(value)</code> ：该字段的值只能大于或等于该值。
<code>@Negative</code> ：判断负数。

<code>@NegativeOrZero</code> ： 判断负数或 0 。

Boolean 值检查

<code>@AssertFalse</code> ： 被注释的元素必须为 <code>true</code> 。
<code>@AssertTrue</code> ： 被注释的元素必须为 <code>false</code> 。

长度检查

<code>@Size(max, min)</code> ： 检查该字段的 <code>size</code> 是否在 <code>mi
</code> 和 <code>max</code> 之间，可以是字符串、数组、集合、Map 等。

日期检查

<code>@Future</code> ： 被注释的元素必须是一个将来的日期。
<code>@FutureOrPresent</code> ： 判断日期是否是将来或现在日期。
<code>@Past</code> ： 检查该字段的日期是在过去。
<code>@PastOrPresent</code> ： 判断日期是否是过去或现在日期。

其它检查

<code>@Email</code> ： 被注释的元素必须是电子邮箱地址。
<code>@Pattern(value)</code> ： 被注释的元素必须符合指定的正则表达式。

<h2 id="2-2-Hibernate-Validator-附加的约束注解">2.2 Hibernate Validator 附加的约束注解</h
>
<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Fhibernate%
Fhibernate-validator%2Ftree%2Fmaster%2Fengine%2Fsrc%2Fmain%2Fjava%2Forg%2Fhibern
te%2Fvalidator%2Fconstraints" target="_blank" rel="nofollow ugc"><code>org.hibernate.val
dator.constraints</code> 包下，定义了一系列的约束(constraint)注解。如下：

<code>@Range(min=, max=)</code> ： 被注释的元素必须在合适的范围内。
<code>@Length(min=, max=)</code> ： 被注释的字符串的大小必须在指定的范围内。
<code>@URL(protocol=,host=,port=,regexp=,flags=)</code> ： 被注释的字符串必须是
个有效的 URL 。
<code>@SafeHtml</code> ： 判断提交的 HTML 是否安全。例如说，不能包含 javascript 脚
等等。
... 等等，就不一一列举了。

<h2 id="2-3--Valid-和--Validated">2.3 @Valid 和 @Validated</h2>
<a href="https://ld246.com/forward?goto=https%3A%2F%2Fdocs.oracle.com%2Fjavaee
2F7%2Fapi%2Fjavax%2Fvalidation%2FValid.html" target="_blank" rel="nofollow ugc"><code
@Valid</code> 注解，是 Bean Validation 所定义，可以添加在普通方法、构造方法、方法
数、方法返回、成员变量上，表示它们需要进行约束校验。
<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Fspring-proj
cts%2Fspring-framework%2Fblob%2Fmaster%2Fspring-context%2Fsrc%2Fmain%2Fjava%2Fo
g%2Fspringframework%2Fvalidation%2Fannotation%2FValidated.java" target="_blank" rel="
ofollow ugc"><code>@Validated</code> 注解，是 Spring Validation 锁定义，可以添加

类、方法参数、普通方法上，表示它们需要进行约束校验。同时，<code>@Validated</code> 有 code>value</code> 属性，支持分组校验。属性如下：
|
<pre><code class="highlight-chroma">// Validated.java

Class<?>[] va ue() default {};
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
对于初学的胖友来说，很容易搞混 <code>@Valid</code> 和 <code>@Validated</code> 注解。
① 声明式校验
Spring Validation 仅 对 <code>@Validated</code> 注解，实现声明校验。
② 分组校验
Bean Validation 提供的 <code>@Valid</code> 注解，因为没有分组校验的属性，所以无法供分组校验。此时，我们只能使用 ``@Validated` 注解。
③ 嵌套校验
相比来说，<code>@Valid</code> 注解的地方，多了【成员变量】。这就导致，如果有嵌套象的时候，只能使用 <code>@Valid</code> 注解。例如说：
|
<pre><code class="highlight-chroma">// User.java
public class User {

    private String id

    @Valid
    private UserProf le profile;

}

// UserProfile.java
public class UserP ofile {

    @NotBlank
    private String n ckname;

}
</code></pre>
<table>
<thead>
<tr>

<th></th>
</tr>
</thead>
</table>

如果不在 <code>User.profile</code> 属性上，添加 <code>@Valid</code> 注解，就会导 <code>UserProfile.nickname</code> 属性，不会进行校验。

当然，<code>@Valid</code> 注解的地方，也多了【构造方法】和【方法返回】，所以在有方面的诉求的时候，也只能使用 <code>@Valid</code> 注解。
:fire: 总结
总的来说，绝大多数场景下，我们使用 <code>@Validated</code> 注解即可。
而在有嵌套校验的场景，我们使用 <code>@Valid</code> 注解添加到成员属性上。
<h2 id="3--快速入门">3. 快速入门</h2>
<blockquote>
示例代码对应仓库：lab-22-validation-01 。
</blockquote>
本小节，我们会实现在 Spring Boot 中，对 SpringMVC 的 Controller 的 API 接口参数，实现数校验。
同时，因为我们在 Service 也会有参数校验的诉求，所以我们也会提供示例。
<h2 id="3-1-引入依赖">3.1 引入依赖</h2>
在 ><code>pom.xml</code> 文件中，引入相关依赖。
|
<pre><code class="highlight-chroma"><?xml version="1.0" encoding="UTF-8"?>
<project xmlns "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="h tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaL cation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" gt;
  <parent>
    <groupId gt;org.springframework.boot</groupId>
    <artifactId gt;spring-boot-starter-parent</artifactId>
    <version& t;2.1.3.RELEASE</version>
    <relativePa h/> <!-- lookup parent from repository -->
  </parent>
  <modelVersi n>4.0.0</modelVersion>

  <artifactId&g ;lab-22-validation-01</artifactId>

  <dependenci s>

        <!-- 实现对 Spring MVC 的自动化配置 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- 保证 Spring AOP 相关的依赖包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
        </dependency>

        <!-- 方便会写单元测试 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
</code></pre>

- 具体每个依赖的作用，胖友自己认真看下艿艿添加的所有注释噢。
- [spring-boot-starter-web](https://ld246.com/forward?goto=https%3A%2F%2Fmvnrepository.com%2Fartifact%2Forg.springframework.boot%2Fspring-boot-starter-web) 依赖里，已经默认引入 [hibernate-validator](https://ld246.com/forward?goto=https%3A%2F%2Fmvnrepository.com%2Fartifact%2Forg.hibernate.validator%2Fhibernate-validator) 依赖，所以本示例使用的是 Hibernate Validator 作为 Bean Validation 的实现框架。

/li>

在 Spring Boot 体系中，也提供了 <code>spring-boot-starter-validation</code></> 依赖。在这里，我们并没有引入。为什么呢？该依赖的目的，重点也是引入 <code>hibernate-validator</code> 依赖，这在 <code>spring-boot-starter-web</code> 已经引入，所以无需重复入。
<h2 id="3-2-Application">3.2 Application</h2>
创建 <code>Application.java</code> 类，配置 <code>@SpringBotApplication</code> 注解即可。代码如下：
|
<pre><code class="highlight-chroma">@SpringBootApplication
@EnableAspectJAtoProxy(exposeProxy = true) // http://www.voidcn.com/article/p-zddcuyii-bpt.html
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

添加 <code>@EnableAspectJAutoProxy</code> 注解，重点是配置 <code>exposeProxy = true</code> ，因为我们希望 Spring AOP 能将当前代理对象设置到 AopContext 中。具用途，我们会在下文看到。想要提前看的胖友，可以看看 《Spring AOP 通过获取代理对象实现事务切换》 文章。

先暂时不启动项目。等我们添加好 Controller 。
<h2 id="3-3-UserAddDTO">3.3 UserAddDTO</h2>
在 <code>cn.iocoder.springboot.lab22.validation.dto</code> 包路径下，创建 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2FSpringB

ot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fjava%2Fcn2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fdto%2FUserAddDTO.java" target="_blan" rel="nofollow ugc">UserAddDTO 类，为用户添加 DTO 类。代码如下：
|
<pre><code class="highlight-chroma">// UserAddDTO.java

public class UserAdDTO {

    /**
     * 账号
     */
    @NotEmpty(message = "登录账号不能为空")
    @Length(min = 5, max = 16, message = "账号长度为 5-16 位")
    @Pattern(regep = "^[A-Za-z0-9]+$", message = "账号格式为数字以及字母")
    private String uername;
    /**
     * 密码
     */
    @NotEmpty(message = "密码不能为空")
    @Length(min = 4, max = 16, message = "密码长度为 4-16 位")
    private String pssword;

    // ... 省略 settin/getting 方法
}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
每个字段上的约束注解，胖友仔细瞅瞅。
<h2 id="3-4-UserController">3.4 UserController</h2>
在 <code>cn.iocoder.springboot.lab22.validation.controller</code> 包径下，创建 UserController 类，提供用户 API 接口。代如下：
|

<pre><code class="highlight-chroma"><span class="highlight
cl">// UserController.java

@RestController
@RequestMappin
("/users")
@Validated
public class UserC
ntroller {

    private Logger
ogger = LoggerFactory.getLogger(getClass());

    @GetMapping(
/get")
    public void get
@RequestParam("id") @Min(value = 1L, message = "编号必须大于 0") Integer id) {
        logger.info("
get][id: {}]", id);
    }

    @PostMapping
"/add")
    public void add
@Valid UserAddDTO addDTO) {
        logger.info("[
dd][addDTO: {}]", addDTO);
    }

}
</code></pre>

在类上，添加 <code>@Validated</code> 注解，表示 UserController 是所有接口都需要进参数校验。
对于 <code>#get(id)</code> 方法，我们在 <code>id</code> 参数上，添加了 <code>@in</code> 注解，校验 <code>id</code> 必须大于 0 。校验不通过示例如下图：

<img src="https
//ld246.com/images/img-loading.svg" alt="不通过示例 1" data-src="https://b3logfile.com/file
2022/04/solo-fetchupload-4278002953257287855-4ffd6adb.jpeg?imageView2/2/interlace/1/
ormat/jpg">

对于 <code>#add(addDTO)</code> 方法，我们在 <code>addDTO</code> 参数上，添了 <code>@Valid</code> 注解，实现对该参数的校验。校验不通过示例如下图：

<img src="https
//ld246.com/images/img-loading.svg" alt="不通过示例 2" data-src="https://b3logfile.com/file
2022/04/solo-fetchupload-8782562183278777062-78312573.jpeg?imageView2/2/interlace/1/
ormat/jpg">

<code>errors</code> 字段，参数错误明细数组 。每一个数组元素，对一个参数错误明细。这里，<code>username</code> 违背了长度不满足 <code>[5, 16]</code>

。

示例我们是已经成功跑通了，但是呢，这里有几点差异性，我们要来理解下。
<blockquote>
艿艿：解释起来，信息量有点大，胖友保持耐心。
也可以不理解，就按照这么使用即可。
</blockquote>
第一点 ，<code>#get(id)</code> 方法上，我们并没有给 <code>id</ode> 添加 <code>@Valid</code> 注解，而 <code>#add(addDTO)</code> 方法上，我们给 <ode>addDTO</code> 添加 <code>@Valid</code> 注解。这个差异，是为什么呢？
因为 UserController 使用了 <code>@Validated</code> 注解，那么 Spring Validation 就使用 AOP 进行切面，进行参数校验。而该切面的拦截器，使用的是 MethodValidationInterceptor 。

对于 <code>#get(id)</code> 方法，需要校验的参数 <code>id</code> ，是平 开的，所以无需添加 <code>@Valid</code> 注解。
对于 <code>#add(addDTO)</code> 方法，需要校验的参数 <code>addDTO</code> ，际相当于嵌套校验 ，要校验的参数的都在 <code>addDTO</code> 里面，以需要添加 <code>@Valid</code> 注解。

第二点 ，<code>#get(id)</code> 方法的返回的结果是 <code>status 500</code> ，而 <code>#add(addDTO)</code> 方法的返回的结果是 <code>status = 400</ode> 。

对于 <code>#get(id)</code> 方法，在 MethodValidationInterceptor 拦截器中，校验到参不正确，会抛出 ConstraintViolationException 异常。
对于 <code>#add(addDTO)</code> 方法，因为 <code>addDTO</code> 是个 POJO 对，所以会走 SpringMVC 的 DataBinder 机制，它会调用 <code>DataBinder#validate(Object... validationHints)</code> 方法，进行校验。在校验不通过，会抛出 BindException 。

在 SpringMVC 中，默认使用 <a href="https://ld246.com/forward?goto=https%3A%2F%2hyrepo.com%2Ftech%2Fspring-mvc-error-handling%2F" target="_blank" rel="nofollow ugc" DefaultHandlerExceptionResolver 处理异常。

对于 BindException 异常，处理成 400 的状态码。
对于 ConstraintViolationException 异常，没有特殊处理，所以处理成 500 的状态码。

这里，我们在抛个问题，如果 <code>#add(addDTO</code> 方法，如果参数正确，在走完 DtaBinder 中的参数校验后，会不会在走一遍 MethodValidationInterceptor 的拦截器呢？思考 100 秒...
答案是会。这样，就会导致浪费。所以 Controller 类里，如果只有 类似的

<code>#add(addDTO)</code> 方法的嵌套校验 ，那么我可以不在 Controller 类上添加 <code>@Validated</code> 注解。从而实现，仅使用 DataBinder 中来做参数校验。
第三点 ，无论是 <code>#get(id)</code> 方法，还是 <code>#add(addDTO)</code> 方法，它们的返回提示都非常不友好，那么该怎么办呢？
参考 《芋道 Spring Boot SpringMVC 入门》 的 「5. 全局异常处理」 ，使用 <code>@ExceptionHandler</code> 注解，实现自定义的异常处理。这，我们在本文的 4. 处理校验异常 小节中，来提供具体示例。
<h2 id="3-5-UserService">3.5 UserService</h2>
相比在 Controller 添加参数校验来说，在 Service 进行参数校验，会更加安全可靠。艿艿个人建议的话，Controller 的参数校验可以不做，Service 的参数校验一定要做 。</p

在 <code>cn.iocoder.springboot.lab22.validation.service</code> 包路径下创建 UserService 类，提供用户 Service 逻辑。代码如下：
|
<pre><code class="highlight-chroma">// UserService.java

@Service
@Validated
public class UserService {

    private Logger logger = LoggerFactory.getLogger(getClass());

    public void get(@Min(value = 1L, message = "编号必须大于 0") Integer id) {
        logger.info("[get][id: {}]", id);
    }

    public void add(@Valid UserAddDTO addDTO) {
        logger.info("[add][addDTO: {}]", addDTO);
    }

    public void add1(UserAddDTO addDTO) {
        this.add(addDTO);
    }

原文链接：Spring Boot 参数校验 Validation 入门

    public void add
2(UserAddDTO addDTO) {
        self().add(ad
DTO);
    }

    private UserServ
ce self() {
        return (UserS
rvice) AopContext.currentProxy();
    }

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

和 UserController 的方法是一致的，包括注解。
额外添加了 <code>#add01(addDTO)</code> 和 <code>#add02(addDTO)</code> 方法用于演示方法内部调用。

创建 UserServiceTest 测试类，我们来测试一下简单的serService 的每个操作。代码如下：
|
<pre><code class="highlight-chroma">
// UserService.java

@RunWith(Sprin
Runner.class)
@SpringBootTest(
lasses = Application.class)
public class UserS
rviceTest {

    @Autowired
    private UserServ
ce userService;

    @Test
    public void test
et() {
        userService.g
t(-1);
    }

    @Test
    public void testdd() {
        UserAddDTO addDTO = new UserAddDTO();
        userService.ad(addDTO);
    }

    @Test
    public void testdd01() {
        UserAddDTO addDTO = new UserAddDTO();
        userService.ad01(addDTO);
    }

    @Test
    public void testdd02() {
        UserAddDTO addDTO = new UserAddDTO();
        userService.ad02(addDTO);
    }

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
① <code>#testGet()</code> 测试方法
执行，抛出 ConstraintViolationException 异常。日志如下：
|
<pre><code class="highlight-chroma">javax.validation.ConstraintViolationException: get.id: 编号必须大于 0

    at org.springframework.validation.beanvalidation.MethodValidationInterceptor.invoke(MethodValidationInteceptor.java:116)
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

符合期望。

② <code>#testAdd()</code> 测试方法
执行，抛出 ConstraintViolationException 异常。日志如下：
|
<pre><code class="highlight-chroma"><span class="highlight
cl">javax.validation.ConstraintViolationException: add.addDTO.username: 登录账号不能为空, a
d.addDTO.password: 密码不能为空

	at org.springfr
mework.validation.beanvalidation.MethodValidationInterceptor.invoke(MethodValidationInte
ceptor.java:116)
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

符合期望。不同于我们在调用 <code>UserController#add(addDTO)</code> 方法，这里被
ethodValidationInterceptor 拦截，进行参数校验，而不是 DataBinder 当中。

③ <code>#testAdd01()</code> 测试方法
执行，正常结束。因为进行 <code>this.add(addDTO)</code> 调用时，<code>this</code>
并不是 Spring AOP 代理对象，所以并不会被 MethodValidationInterceptor 所拦截。
④ <code>#testAdd02()</code> 测试方法
执行，抛出 IllegalStateException 异常。日志如下：
|
<pre><code class="highlight-chroma"><span class="highlight
cl">java.lang.IllegalStateException: Cannot find current proxy: Set 'exposeProxy' property on
dvised to 'true' to make it available.

	at org.springfr
mework.aop.framework.AopContext.currentProxy(AopContext.java:69)
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

理论来说，因为我们配置了 <code>@EnableAspectJAutoProxy(exposeProxy = true)</code
注解，在 Spring AOP 拦截时，通过调用 <code>AopContext.currentProxy()</code> 方法，是
以获取到当前的代理对象。结果，此处抛出 IllegalStateException 异常。
显然，这里并没有将当前的代理对象，设置到 AopContext 中，所以抛出 IllegalStateException
异常。目前猜测，可能是 BUG 。:smiling_imp: 暂时木有心情去调试，嘿嘿。

<h2 id="4--处理校验异常">4. 处理校验异常</h2>
<blockquote>
示例代码对应仓库：<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.c

m%2FYunaiV%2FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01" target="_blank" rel="nofollow ugc">lab-22-validation-01 。
</blockquote>
在 「3. 快速入门」 中我们可以看到，如果直接将校验的结果返回给前端，提示内容的可阅读性是比较差的，所以我们需要校验抛出的异常进行处理。
在 《芋道 Spring Boot SpringMVC 入门》 的 「5. 全局异常处理」 小节中，使用 <code>@ExceptionHandler</code> 注解，实现自定义的异常处理。所以本小节，我们在 「3. 快速入门」 小节的 lab-22-validation-01 示例，进一步处理校验异常。
<h2 id="4-1-复制粘贴">4.1 复制粘贴</h2>
我们先把 《芋道 Spring Boot SpringMVC 入门》 的 「5. 全局异常处理」 小节中，需要用到的类，全部复制过来。

在 <code>cn.iocoder.springboot.lab22.validation.constants</code> 包路径下，复制 ServiceExceptionEnum 类。
在
<code>cn.iocoder.springboot.lab22.validation.core.exception</code> 包路径下，复制 ServiceException 类

在
<code>cn.iocoder.springboot.lab22.validation.core.vo</code> 包路下，复制 CommonResult 类。
在
<code>cn.iocoder.springboot.lab22.validation.core.web</code> 路径下，复制 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYu

aiV%2FSpringBoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fman%2Fjava%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcore%2Fweb%2FGlobaExceptionHandler.java" target="_blank" rel="nofollow ugc">GlobalExceptionHandler 和 a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2FSpringoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fjava%2Fc%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcore%2Fweb%2FGlobalResponseBodyandler.java" target="_blank" rel="nofollow ugc">GlobalResponseBodyHandler 类。

<h2 id="4-2-ServiceExceptionEnum">4.2 ServiceExceptionEnum</h2>
修改 ServiceExceptionEnum 枚举类，增校验参数不通过的错误码枚举。代码如下：
|
<pre><code class="highlight-chroma">// ServiceExceptionEnum.java

INVALID_REQUES_PARAM_ERROR(2001001002, "请求参数不合法"),
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
<h2 id="4-3-GlobalExceptionHandler">4.3 GlobalExceptionHandler</h2>
修改 GlobalExceptionHandler 类，加 <code>#constraintViolationExceptionHandler(...)</code> 方法，处理 ConstraintViolationException 异常。代码如下：
|
<pre><code class="highlight-chroma">// GlobalExceptionHandler.java
@ResponseBody
@ExceptionHandlr(value = ConstraintViolationException.class)
public CommonResult constraintViolationExceptionHandler(HttpServletRequest req, ConstraintViolationExceptin ex) {
    logger.debug("[onstraintViolationExceptionHandler]", ex);
    // 拼接错误
    StringBuilder detailMessage = new StringBuilder();
    for (ConstraintVolation<?> constraintViolation : ex.getConstraintViolations()) {
        // 使用 ; 分多个错误</code>

原文链接：Spring Boot 参数校验 Validation 入门

        if (detailMes
age.length() > 0) {
            detailMess
ge.append(";");
        }
        // 拼接内容
其中
        detailMessag
.append(constraintViolation.getMessage());
    }
    // 包装 Commo
Result 结果
    return Commo
Result.error(ServiceExceptionEnum.INVALID_REQUEST_PARAM_ERROR.getCode(),
            ServiceExc
ptionEnum.INVALID_REQUEST_PARAM_ERROR.getMessage() + ":" + detailMessage.toString())

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

将每个约束的错误内容提示，拼接起来，使用 <code>;</code> 分隔。
重新请求 <code>UserController#get(id)</code> 对应的接口，响应结果如下：<img src="ht
ps://ld246.com/images/img-loading.svg" alt="constraintViolationExceptionHandler" data-src
"https://b3logfile.com/file/2022/04/solo-fetchupload-2696684439699133672-268004a6.png?
mageView2/2/interlace/1/format/jpg">

修改 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunai
%2FSpringBoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain
2Fjava%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcore%2Fweb%2FGlobalEx
eptionHandler.java" target="_blank" rel="nofollow ugc">GlobalExceptionHandler 类，
加 <code>#bindExceptionHandler(...)</code> 方法，处理 BindException 异常。代码如下：</p

|
<pre><code class="highlight-chroma"><span class="highlight
cl">// GlobalExceptionHandler.java

@ResponseBody
@ExceptionHandl
r(value = BindException.class)
public CommonR
sult bindExceptionHandler(HttpServletRequest req, BindException ex) {
    logger.debug("
bindExceptionHandler]", ex);
    // 拼接错误
    StringBuilder de
ailMessage = new StringBuilder();
    for (ObjectError

objectError : ex.getAllErrors()) {
        // 使用 ; 分多个错误
        if (detailMessage.length() > 0) {
            detailMessage.append(";");
        }
        // 拼接内容其中
        detailMessag.append(objectError.getDefaultMessage());
    }
    // 包装 CommoResult 结果
    return CommoResult.error(ServiceExceptionEnum.INVALID_REQUEST_PARAM_ERROR.getCode(),
            ServiceExceptionEnum.INVALID_REQUEST_PARAM_ERROR.getMessage() + ":" + detailMessage.toString())

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

将每个约束的错误内容提示，拼接起来，使用 <code>;</code> 分隔。
重新请求 <code>UserController#add(addDTO)</code> 对应的接口，响应结果如下：<img rc="https://ld246.com/images/img-loading.svg" alt="bindExceptionHandler" data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-5038843249647905972-88da93d5.png?imageView2/2/interlace/1/format/jpg">

<h2 id="5--自定义约束">5. 自定义约束</h2>
<blockquote>
示例代码对应仓库：lab-22-validation-01 。
</blockquote>
在大多数项目中，无论是 Bean Validation 定义的约束，还是 Hibernate Validator 附加的约束都是无法满足我们复杂的业务场景。所以，我们需要自定义约束。
开发自定义约束一共只要两步：1）编写自定义约束的注解；2）编写自定义的校验器 ConstraintValidator。
下面，就让我们一起来实现一个自定义约束，用于校验参数必须在枚举值的范围内。
<h2 id="5-1-IntArrayValuable">5.1 IntArrayValuable</h2>
在 `cn.iocoder.springboot.lab22.validation.core.validator` 包路径下，创建 IntArrayValuable 接口，用于返回值数组。代码如下：
|

<pre><code class="highlight-chroma">// IntArrayValuable.java

public interface IntArrayValuable {

    /**
     * @return int[] 数组
     */
    int[] array();

}
</code></pre>

因为对于一个枚举类来说，我们无法获得它具体有那些值。所以，我们会要求这个枚举类实现该接口，返回它拥有的所有枚举值。

<h2 id="5-2-GenderEnum">5.2 GenderEnum</h2>

在 <code>cn.iocoder.springboot.lab22.validation.constants</code> 包路径下，创建 GenderEnum 枚举类，枚举性别。代码如下：

|

<pre><code class="highlight-chroma">// GenderEnum.java

public enum GenderEnum implements IntArrayValuable {

    MALE(1, "男"),
    FEMALE(2, "女");

    /**
     * 值数组
     */
    public static final int[] ARRAYS = Arrays.stream(values()).mapToInt(GenderEnum::getValue).toArray();

    /**
     * 性别值
     */
    private final Integer value;

```
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 性别名
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    private final Str
ng name;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    GenderEnum(In
eger value, String name) {
</span></span><span class="highlight-line"><span class="highlight-cl">        this.value = v
lue;
</span></span><span class="highlight-line"><span class="highlight-cl">        this.name =
ame;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public Integer
etValue() {
</span></span><span class="highlight-line"><span class="highlight-cl">        return value;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public String ge
Name() {
</span></span><span class="highlight-line"><span class="highlight-cl">        return name;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public int[] arra
() {
</span></span><span class="highlight-line"><span class="highlight-cl">        return ARRA
S;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
<ul>
<li>实现 IntArrayValuable 接口，返回值数组 <code>ARRAYS</code> 。</li>
</ul>
<h2 id="5-3--InEnum">5.3 @InEnum</h2>
<p>在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%
FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fj
va%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcore%2Fvalidator" target="_b
ank" rel="nofollow ugc"><code>cn.iocoder.springboot.lab22.validation.core.validator</cod
></a> 包路径下，创建 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.c
m%2FYunaiV%2FSpringBoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2F
rc%2Fmain%2Fjava%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcore%2Fvali
ator%2FInEnum.java" target="_blank" rel="nofollow ugc"><code>@InEnum</code></a> <s
rong>自定义约束的注解</strong> 。代码如下：</p>
<p>|</p>
```

<pre><code class="highlight-chroma">// InEnum.java

@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy = InEnumValidator.class)
public @interface InEnum {

    /**
     * @return 实现 IntArrayValuable 接口的
     */
    Class<? extends IntArrayValuable> value();

    /**
     * @return 提示内容
     */
    String message() default "必须在指定范围 {value}";

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

    /**
     * @return Payload 数组
     */
    Class<? extends Payload>[] payload() default {};

    /**
     * Defines several {@code @InEnum} constraints on the same element.
     */
    @Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List

        InEnum[] value();

原文链接：Spring Boot 参数校验 Validation 入门

    }

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

在类上，添加 <code>@@Constraint(validatedBy = InEnumValidator.class)</code> 注解设置使用的自定义约束的校验器 。
<code>value()</code> 属性，设置实现 IntArrayValuable 接口的类。这样，我们就能获得参需要校验的值数组。
<code>message()</code> 属性，设置提示内容。默认为 <code>"必须在指定范围 {value}"<code> 。
其它属性，复制粘贴即可，都可以忽略不用理解。

<h2 id="5-4-InEnumValidator">5.4 InEnumValidator</h2>
在 <code>cn.iocoder.springboot.lab22.validation.core.validator</cod> 包路径下，创建 InEnumValidator <stong>自定义约束的校验器 。代码如下：
|
<pre><code class="highlight-chroma">// InEnumValidator.java

public class InEnmValidator implements ConstraintValidator<InEnum, Integer> {

    /**
     * 值数组
     */
    private Set<Integer> values;

    @Override
    public void initialize(InEnum annotation) {
        IntArrayValuable[] values = annotation.value().getEnumConstants();
        if (values.length == 0) {
            this.values = Collections.emptySet();
        } else {

            this.values = Arrays.stream(values[0].array()).boxed().collect(Collectors.toSet());
        }
    }

    @Override
    public boolean sValid(Integer value, ConstraintValidatorContext context) {
        // <2.1> 校验通过
        if (values.contains(value)) {
            return true

        }
        // <2.2.1>校验不通过，自定义提示语句（因为，注解上的 value 是枚举类，无法获得枚举类的实际值）
        context.disableDefaultConstraintViolation(); // 禁用默认的 message 的值
        context.buildConstraintViolationWithTemplate(context.getDefaultConstraintMessageTemplate()
                .replaceAll("\\{value}", values.toString())).addConstraintViolation(); // 重新添加错误提示语句
        return false; // <2.2.2.>
    }

}
</code></pre>

实现 ConstraintValidator 接口。

第一个泛型为 <code>A extends Annotation</code> ，设置对应的自定义约束的注解。例如，这里我们设置了 <code>@InEnum</code> 注解。
第二个泛型为 <code>T</code> ，设置对应的参数值的类型。例如说，这里我们设置了 Integer 类型。

实现 <code>#initialize(annotation)</code> 方法，获得 <code>@InEnum</code> 注解的 <code>values()</code> 属性，获得值数组，设置到 <code>values</code> 属性种。
实现 <code>#isValid(value, context)</code> 方法，实现校验参数值，是否在 <code>value</code> 范围内。

<code><2.1></code> 处，校验参数值在范围内，直接返回 <code>true</code> ，校

通过。
<code><2.2.1></code> 处，校验不通过，自定义提示语句。
<code><2.2.2></code> 处，校验不通过，所以返回 <code>false</code> 。

至此，我们已经完成了自定义约束的实现。下面，我们来进行下测试。
<h2 id="5-5-UserUpdateGenderDTO">5.5 UserUpdateGenderDTO</h2>
在 <code>cn.iocoder.springboot.lab22.validation.dto</code> 包路径下，创建 UserUpdateGenderDTO 类，为用户更新性别 DTO。代码下：
|
<pre><code class="highlight-chroma">// UserUpdateGenderDTO.java

public class UserpdateGenderDTO {

    /**
     * 用户编号
     */
    @NotNull(message = "用户编号不能为空")
    private Integerd;

    /**
     * 性别
     */
    @NotNull(message = "性别不能为空")
    @InEnum(value = GenderEnum.class, message = "性别必须是 {value}")
    private Integerender;

    // ... 省略 set/ge 方法
}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

在 <code>gender</code> 字段上，添加 <code>@InEnum(value = GenderEnum.class, me
sage = "性别必须是 {value}")</code> 注解，限制传入的参数值，必须在 GenderEnum 枚举范围
。

<h2 id="5-6-UserController">5.6 UserController</h2>
修改 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunai
%2FSpringBoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain
2Fjava%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcontroller%2FUserControl
er.java" target="_blank" rel="nofollow ugc">UserController 类，增加修改性别 API 接口
代码如下：
|
<pre><code class="highlight-chroma"><span class="highlight
cl">// UserController.java

@PostMapping("/
pdate_gender")
public void updat
Gender(@Valid UserUpdateGenderDTO updateGenderDTO) {
    logger.info("[u
dateGender][updateGenderDTO: {}]", updateGenderDTO);
}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
模拟请求该 API 接口，响应结果如下：<img src="https://ld246.com/images/img-loading.sv
" alt="响应结果" data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-68190764517
3999182-f0229c1e.png?imageView2/2/interlace/1/format/jpg">
因为我们传入的请求参数 <code>gender</code> 的值为 <code>null</code> ，显然不在 G
nderEnum 范围内，所以校验不通过，输出 <code>"性别必须是 [1, 2]"</code> 。
<h2 id="6--分组校验">6. 分组校验</h2>
<blockquote>
示例代码对应仓库：<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.c
m%2FYunaiV%2FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01" tar
et="_blank" rel="nofollow ugc">lab-22-validation-01 。
</blockquote>
在一些业务场景下，我们需要使用分组 校验，即相同的 Bean 对象，根据
验分组，使用不同的校验规则。咳咳咳，貌似我们暂时没有这方面的诉求。即使有，也是拆分不同的
ean 类。当然，作为一篇入门的文章，艿艿还是提供下分组校验的示例。
<h2 id="6-1-UserUpdateStatusDTO">6.1 UserUpdateStatusDTO</h2>
在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%
FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fj
va%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fdto" target="_blank" rel="nof
llow ugc"><code>cn.iocoder.springboot.lab22.validation.dto</code> 包路径下，创建 <a
href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2FSpringB
ot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fjava%2Fcn
2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fdto%2FUserUpdateStatusDTO.java" targ
t="_blank" rel="nofollow ugc">UserUpdateStatusDTO 类，为用户更新状态 DTO 。代码
下：
|

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">// UserUpdateStatusDTO.java
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">public class User
pdateStatusDTO {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 分组 01 ， 要
状态必须为 true
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    public interface
Group01 {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 状态 02 ， 要
状态必须为 false
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    public interface
Group02 {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 状态
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    @AssertTrue(m
ssage = "状态必须为 true", groups = Group01.class)
</span></span><span class="highlight-line"><span class="highlight-cl">    @AssertFalse(
essage = "状态必须为 false", groups = Group02.class)
</span></span><span class="highlight-line"><span class="highlight-cl">    private Boolean
status;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    // ... 省略 set/ge
 方法
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
<ul>
<li>创建了 Group01 和 Group02 接口，作为两个校验分组。不一定要定义在 UserUpdateStatusD
O 类中，这里仅仅是为了方便。</li>
<li><code>status</code> 字段，在 Group01 校验分组时，必须为 <code>true</code> ；在 G
oup02 校验分组时，必须为 <code>false</code> 。</li>
</ul>
<h2 id="6-2-UserController">6.2 UserController</h2>
<p>修改 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunai
%2FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%
Fjava%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fcontroller%2FUserControll
r.java" target="_blank" rel="nofollow ugc">UserController</a> 类，增加两个修改状态的 API
口。代码如下：</p>
<p>|</p>
```

<pre><code class="highlight-chroma">// UserController.java

@PostMapping("/pdate_status_true")
public void updat StatusTrue(@Validated(UserUpdateStatusDTO.Group01.class) UserUpdateStatusDTO updateS atusDTO) {
    logger.info("[u dateStatusTrue][updateStatusDTO: {}]", updateStatusDTO);
}

@PostMapping("/pdate_status_false")
public void updat StatusFalse(@Validated(UserUpdateStatusDTO.Group02.class) UserUpdateStatusDTO updateS atusDTO) {
    logger.info("[u dateStatusFalse][updateStatusDTO: {}]", updateStatusDTO);
}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

对于 <code>#updateStatusTrue(updateStatusDTO)</code> 方法，我们在 <code>update tatusDTO</code> 参数上，添加了 <code>@Validated</code> 注解，并且设置校验分组为 Gro p01 。校验不通过示例如下图：<img src="https://ld246.com/images/img-loading.svg" alt="不过示例 1" data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1308873692889320 30-72d209cf.jpeg?imageView2/2/interlace/1/format/jpg">
对于 <code>#updateStatusFalse(updateStatusDTO)</code> 方法，我们在 <code>update tatusDTO</code> 参数上，添加了 <code>@Validated</code> 注解，并且设置校验分组为 Gro p02 。校验不通过示例如下图：<img src="https://ld246.com/images/img-loading.svg" alt="不过示例 2" data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-3567650878955779 18-52a66c17.jpeg?imageView2/2/interlace/1/format/jpg">

所以，使用分组校验，核心在于添加上 <code>@Validated</code> 注解，并设置对应的校验组。
<h2 id="7--手动校验">7. 手动校验</h2>
<blockquote>
示例代码对应仓库：lab-22-validation-01 。
</blockquote>
在上面的示例中，我们使用的主要是 Spring Validation 的声明式注解。然而在少数业务场景下我们可能需要手动使用 Bean Validation API ，进行参数校验。
修改 UserServiceTest 测试类，增加手动参数校验的示

。代码如下：
|
<pre><code class="highlight-chroma">// UserServiceTest.java

@Autowired // <1.1>
private Validator validator;

@Test
public void testValidator() {
    // 打印，查看 validator 的类型 // <1.2>
    System.out.println(validator);

    // 创建 UserAddDTO 对象 // <2>
    UserAddDTO addDTO = new UserAddDTO();
    // 校验 // <3>
    Set<ConstraintViolation<UserAddDTO>> result = validator.validate(addDTO);
    // 打印校验结果 // <4>
    for (ConstraintViolation<UserAddDTO> constraintViolation : result) {
        // 属性:消息
        System.out.println(constraintViolation.getPropertyPath() + ":" + constraintViolation.getMessage());
    }
}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

<code><1.1></code> 处，注入 Validator Bean 对象。

<code><1.2></code> 处，打印 <code>validator</code> 的类型。输出如下：
|
<pre><code class="highlight-chroma">org.springframework.validation.beanvalidation.LocalValidatorFactoryBean@48c3205a
</code></pre>
<table>

<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

<code>validator</code> 的类型为 LocalValidatorFactoryBen 。LocalValidatorFactoryBean 提供 JSR-303、JSR-349 的支持，同时兼容 Hibernate Validator 。
在 Spring Boot 体系中，使用 ValidationAutoConfiguration 自动化配置类，默认创建 LocalValidatorFactoryBea 作为 Validator Bean 。

<code><2></code> 处，创建 UserAddDTO 对象，即 「3.3 UserAddDTO」 ，已经添加相应的约束注解。

<code><3></code> 处，调用 <code>Validator#validate(T object, Class<?>... goups)</code> 方法，进行参数校验。

<code><4></code> 处，打印校验结果。输出如下：
|
<pre><code class="highlight-chroma">username:登录账号不能为空
password:密码不为空
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

如果校验通过，则返回的 <code>Set<ConstraintViolation<?>></code> 集合为 。

<h2 id="8--国际化-i18n">8. 国际化 i18n</h2>
<blockquote>
示例代码对应仓库：<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.c

m%2FYunaiV%2FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01" tar
et="_blank" rel="nofollow ugc">lab-22-validation-01 。
</blockquote>
在一些项目中，我们会有国际化的需求，特别是我们在做 TOB 的 SASS 化服务的时候。那么，
然我们在使用 Bean Validator 做参数校验的时候，也需要提供国际化的错误提示。
给力的是，Hibernate Validator 已经内置了国际化的支持，所以我们只需要简单的配置，就可
实现国际化的错误提示。
<h2 id="8-1-应用配置文件">8.1 应用配置文件</h2>
在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%
FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fr
sources" target="_blank" rel="nofollow ugc"><code>resources</code> 目录下，创建 <a
href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2FSpringB
ot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fresources
2Fapplication.yaml" target="_blank" rel="nofollow ugc"><code>application.yaml</code></
> 配置文件。配置如下：
|
<pre><code class="highlight-chroma"><span class="highlight
cl">spring:
  # i18 message
置，对应 MessageSourceProperties 配置类
  messages:
    basename: i18n
messages # 文件路径基础名
    encoding: UTF-
 # 使用 UTF-8 编码
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
然后，我们在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%
FYunaiV%2FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2
main%2Fresources%2Fi18n" target="_blank" rel="nofollow ugc">resources/i18</cod
> 目录下，创建不同语言的 messages 文件。如下：

<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2F
pringBoot-Labs%2Fblob%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fre
ources%2Fi18n%2Fmessages.properties" target="_blank" rel="nofollow ugc">messa
es.properties</code> ：默认的 i18 配置文件。
|
<pre><code class="highlight-chroma"><span class="highlight
cl">UserUpdateDTO.id.NotNull=用户编号不能为空
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

<code>messages_en.properties</code>：英文的 i18 配置文件。
|
<pre><code class="highlight-chroma">UserUpdateDTO.id.NotNull=userId cannot be empty
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

<code>messages_ja.properties</code>：日文的 i18 配置文件。
|
<pre><code class="highlight-chroma">UserUpdateDTO.id.NotNull=ユーザー番号は空にできません
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

<h2 id="8-2-ValidationConfiguration">8.2 ValidationConfiguration</h2>
在 <code>cn.iocoder.springboot.lab22.validation.config</code> 包路径下，建 ValidationConfiguration 配置类，用于创建一个支持 i18 国际化的 Validator Bean 对象。代码下：
|
<pre><code class="highlight-chroma">// ValidationConfiguration.java

@Configuration
public class ValidationConfiguration {

    /**
     * 参考 {@link ValidationAutoConfiguration#defaultValidator()} 方法，构建 Validator Bean

```
</span></span><span class="highlight-line"><span class="highlight-cl">     *
</span></span><span class="highlight-line"><span class="highlight-cl">     * @return Vali
ator 对象
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    @Bean
</span></span><span class="highlight-line"><span class="highlight-cl">    public Validator
validator(MessageSource messageSource) {
</span></span><span class="highlight-line"><span class="highlight-cl">        // 创建 Local
alidatorFactoryBean 对象
</span></span><span class="highlight-line"><span class="highlight-cl">        LocalValidato
FactoryBean validator = ValidationAutoConfiguration.defaultValidator();
</span></span><span class="highlight-line"><span class="highlight-cl">        // 设置 mess
geSource 属性，实现 i18 国际化
</span></span><span class="highlight-line"><span class="highlight-cl">        validator.set
alidationMessageSource(messageSource);
</span></span><span class="highlight-line"><span class="highlight-cl">        // 返回
</span></span><span class="highlight-line"><span class="highlight-cl">        return valida
or;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
<h2 id="8-3-UserUpdateDTO">8.3 UserUpdateDTO</h2>
<p>在 <a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%
FSpringBoot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fj
va%2Fcn%2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fdto" target="_blank" rel="nof
llow ugc"><code>cn.iocoder.springboot.lab22.validation.dto</code></a> 包路径下，创建 <a
href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FYunaiV%2FSpringB
ot-Labs%2Ftree%2Fmaster%2Flab-22%2Flab-22-validation-01%2Fsrc%2Fmain%2Fjava%2Fcn
2Fiocoder%2Fspringboot%2Flab22%2Fvalidation%2Fdto%2FUserUpdateDTO.java" target="_b
ank" rel="nofollow ugc">UserUpdateDTO</a> 类，为用户更新 DTO 。代码如下：</p>
<p>|</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">// UserUpdateDTO.java
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">public class User
pdateDTO {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 用户编号
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    @NotNull(mes
age = "{UserUpdateDTO.id.NotNull}")
</span></span><span class="highlight-line"><span class="highlight-cl">    private Integer
d;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    // ... 省略 get/se
```

方法

}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>

不同于我们上面看到的约束注解的 <code>message</code> 属性的设置，这里我们使用了 <code>{}</code> 占位符。

<h2 id="8-4-UserController">8.4 UserController</h2>
修改 UserController 类，增加用户更新的 API 接口代码如下：
|
<pre><code class="highlight-chroma">>>// UserController.java
>
>@PostMapping("/pdate")
>public void updat(@Valid UserUpdateDTO updateDTO) {
>   logger.info("[udate][updateDTO: {}]", updateDTO);
>}
</code></pre>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
下面，我们来进行下 API 接口测试。有一点要注意，SpringMVC 通过 <code>Accept-Langua e</code> 请求头，实现 i18n 国际化。

<code>Accept-Language = zh</code> 的情况，响应结果如下：<img src="https://ld246.cm/images/img-loading.svg" alt="" data-src="https://b3logfile.com/file/2022/04/solo-fetchupoad-6463921001074518379-974713f7.jpeg?imageView2/2/interlace/1/format/jpg">
<code>Accept-Language = en</code> 的情况，响应结果如下：<img src="https://ld246.cm/images/img-loading.svg" alt="" data-src="https://b3logfile.com/file/2022/04/solo-fetchupoad-8907132363355676454-4561891f.jpeg?imageView2/2/interlace/1/format/jpg">
<code>Accept-Language = ja</code> 的情况，响应结果如下：<img src="https://ld246.cm/images/img-loading.svg" alt="" data-src="https://b3logfile.com/file/2022/04/solo-fetchupoad-7611245946864418212-204e204c.jpeg?imageView2/2/interlace/1/format/jpg">

至此，我们的 Validator 的 i18n 国际化已经完成了。

不过细心的胖友，会发现 <code>"请求参数不合法"</code> 并没有国际化处理。是的~实际，国际化是个大工程，涉及到方方面面。例如说，业务信息表的国际化，商品同时支持中文、英文、文等多种语言。:smiling_imp: 最近艿艿手头有个新项目，需要做国际化，有这方面需求的胖友，可一起多多交流呀。
<h2 id="666--彩蛋">666. 彩蛋</h2>
希望阅读完本文，能够让胖友更加舒适且优雅的完成各种需要参数校验的地方。:smiling_imp: 说了，艿艿赶紧给自己的系统去把参数校验给补全，嘿嘿。
当然，有一点要注意，Bean Validation 更多做的是，无状态的参数校验。怎么理解呢？

例如说，参数的大小长度等等，是适合 通过 Bean Validation 中完成。
例如说，校验用户名唯一等等，依赖外部数据源的，是不适合 通过 Bean Validation 中完成。

当然，如果胖友有不同意见，欢迎留言讨论。
受限于篇幅，艿艿偷懒了下，还有一些内容其实可以补充：

《Intro to Apache BVal》 使用 Apache BVal 实现参数校验。
《使用 Spring 的 Validator 接进行校验》，通过实现 Validator 接口，提供对应 Bean 的参数校验器。
