使用spring validation完成数据后端校验

2017年08月16日 15:52:47 下一秒升华 阅读数 71345

前言

数据的校验是交互式网站一个不可或缺的功能，前端的js校验可以涵盖大部分的校验职责，如用户名唯一性，生日格式，邮箱格式校验等等常用的校验。但是为了避免用户绕过浏览器，使用http工具直接向后端请求一些违法数据，服务端的数据校验也是必要的，可以防止脏数据落到数据库中，如果数据库中出现一个非法的邮箱格式，也会让运维人员头疼不已。我在之前保险产品研发过程中，系统对数据校验要求比较严格且追求可变性及效率，曾使用drools作为规则引擎，兼任了校验的功能。而在一般的应用，可以使用本文将要介绍的validation来对数据进行校验。

简述JSR303/JSR-349，hibernate validation，spring validation之间的关系。JSR303是一项标准,JSR-349是其的升级版本，添加了一些新特性，他们规定一些校验规范即校验注解，如@Null，@NotNull，@Pattern，他们位于javax.validation.constraints包下，只提供规范不提供实现。而hibernate validation是对这个规范的实践（不要将hibernate和数据库orm框架联系在一起），他提供了相应的实现，并增加了一些其他校验注解，如@Email，@Length，@Range等等，他们位于org.hibernate.validator.constraints包下。而万能的spring为了给开发者提供便捷，对hibernate validation进行了二次封装，显示校验validated bean时，你可以使用spring validation或者hibernate validation，而spring validation另一个特性，便是其在springmvc模块中添加了自动校验，并将校验信息封装进了特定的类中。这无疑便捷了我们的web开发。本文主要介绍在springmvc中自动校验的机制。

引入依赖

我们使用maven构建springboot应用来进行demo演示。

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

我们只需要引入spring-boot-starter-web依赖即可，如果查看其子依赖，可以发现如下的依赖：

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

验证了我之前的描述，web模块使用了hibernate-validation，并且databind模块也提供了相应的数据绑定功能。

构建启动类

无需添加其他注解，一个典型的启动类

```
@SpringBootApplication
public class ValidateApp {

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

创建需要被校验的实体类

```
public class Foo {

    @NotBlank
    private String name;

    @Min(18)
    private Integer age;

    @Pattern(regexp = "^1(3|4|5|7|8)\\d{9}$",message = "手机号码格式错误")
    @NotBlank(message = "手机号码不能为空")
    private String phone;

    @Email(message = "邮箱格式错误")
    private String email;

    //... getter setter

}
```

使用一些比较常用的校验注解，还是比较浅显易懂的，字段上的注解名称即可推断出校验内容，每一个注解都包含了message字段，用于校验失败时作为提示信息，特殊的校验注解，如Pattern（正则校验），还可以自己添加正则表达式。

在@Controller中校验数据

springmvc为我们提供了自动封装表单参数的功能，一个添加了参数校验的典型controller如下所示。

```
@Controller
public class FooController {

    @RequestMapping("/foo")
    public String foo(@Validated Foo foo <1>, BindingResult bindingResult <2>) {
        if(bindingResult.hasErrors()){
            for (FieldError fieldError : bindingResult.getFieldErrors()) {
                //...
            }
            return "fail";
        }
        return "success";
    }

}
```

值得注意的地方：

<1> 参数Foo前需要加上@Validated注解，表明需要spring对其进行校验，而校验的信息会存放到其后的BindingResult中。注意，必须相邻，如果有多个参数需要校验，形式可以如下。foo(@Validated Foo foo, BindingResult fooBindingResult ，@Validated Bar bar, BindingResult barBindingResult);即一个校验类对应一个校验结果。

<2> 校验结果会被自动填充，在controller中可以根据业务逻辑来决定具体的操作，如跳转到错误页面。

一个最基本的校验就完成了，总结下框架已经提供了哪些校验：

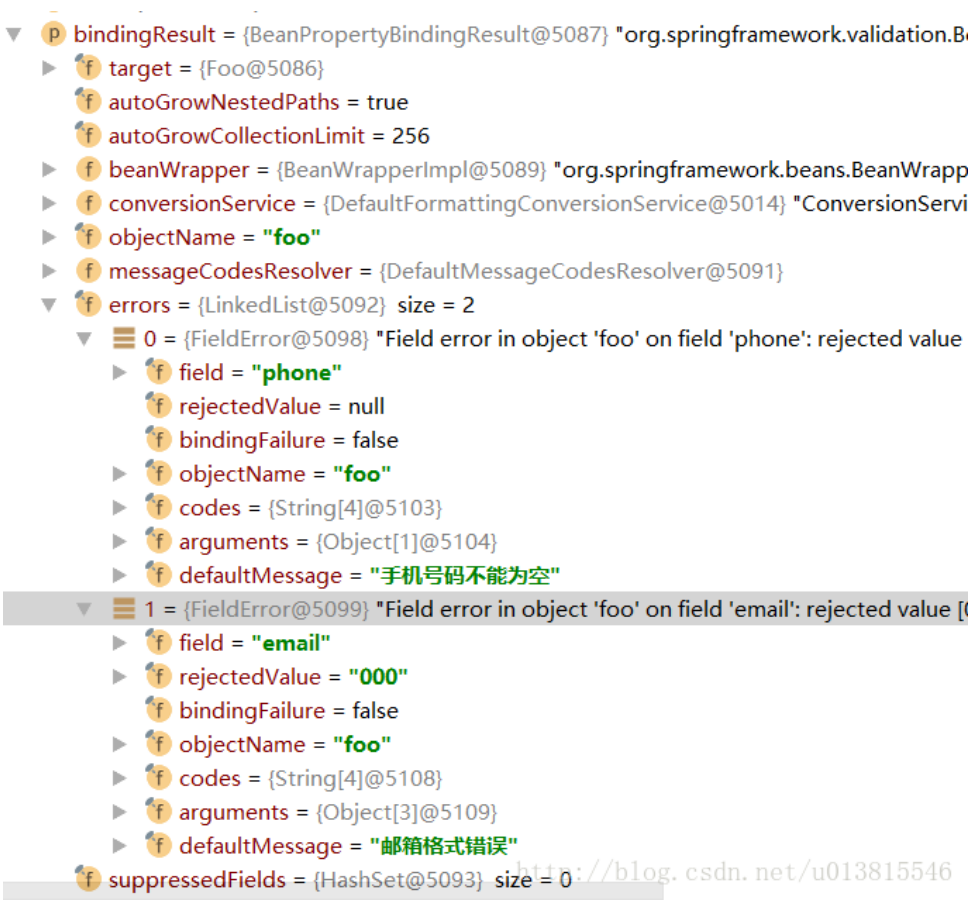
```
JSR提供的校验注解：
@Null   被注释的元素必须为 null
@NotNull    被注释的元素必须不为 null
@AssertTrue     被注释的元素必须为 true
@AssertFalse    被注释的元素必须为 false
@Min(value)     被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)     被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)  被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)  被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max=, min=)   被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)     被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past   被注释的元素必须是一个过去的日期
```

```
@Past    被注释的元素必须是一个过去的日期
@Future     被注释的元素必须是一个将来的日期
@Pattern(regex=,flag=)  被注释的元素必须符合指定的正则表达式


Hibernate Validator提供的校验注解：
@NotBlank(message =)   验证字符串非null，且长度必须大于0
@Email  被注释的元素必须是电子邮箱地址
@Length(min=,max=)  被注释的字符串的大小必须在指定的范围内
@NotEmpty   被注释的字符串的必须非空
@Range(min=,max=,message=)  被注释的元素必须在合适的范围内
```

校验实验

我们对上面实现的校验入口进行一次测试请求：
访问 http://localhost:8080/foo?name=xujingfeng&email=000&age=19 可以得到如下的debug信息：



实验告诉我们，校验结果起了作用。并且，可以发现当发生多个错误，spring validation不会在第一个错误发生后立即停止，而是继续试错，告诉我们所有的错误。debug可以查看到更多丰富的错误信息，这些都是spring validation为我们提供的便捷特性，基本适用于大多数场景。

你可能不满足于简单的校验特性，下面进行一些补充。

分组校验

如果同一个类，在不同的使用场景下有不同的校验规则，那么可以使用分组校验。未成年人是不能喝酒的，而在其他场景下我们不做特殊的限制，这个需求如何体现同一个实体，不同的校验规则呢？

改写注解，添加分组：

```
Class Foo{

    @Min(value = 18,groups = {Adult.class})
    private Integer age;

```

```
    public interface Adult{}

    public interface Minor{}
}
```

这样表明，只有在Adult分组下，18岁的限制才会起作用。

Controller层改写：

```
@RequestMapping("/drink")
public String drink(@Validated({Foo.Adult.class}) Foo foo, BindingResult bindingResult) {
    if(bindingResult.hasErrors()){
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            //...
        }
        return "fail";
    }
    return "success";
}

@RequestMapping("/live")
public String live(@Validated Foo foo, BindingResult bindingResult) {
    if(bindingResult.hasErrors()){
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            //...
        }
        return "fail";
    }
    return "success";
}
```

drink方法限定需要进行Adult校验，而live方法则不做限制。

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，主要分为两步。

1 自定义校验注解
我们尝试添加一个“字符串不能包含空格”的限制。

```
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {CannotHaveBlankValidator.class})<1>
public @interface CannotHaveBlank {

    //默认错误消息
    String message() default "不能包含空格";

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

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

    //指定多个时使用
    @Target({FIELD, METHOD, PARAMETER, ANNOTATION_TYPE})
    @Retention(RUNTIME)
    @Documented
    @interface List {
        CannotHaveBlank[] value();
    }

```

```
    }
```

我们不需要关注太多东西，使用spring validation的原则便是便捷我们的开发，例如payload，List ， groups，都可以忽略。

<1> 自定义注解中指定了这个注解真正的验证者类。

2 编写真正的校验者类

```
public class CannotHaveBlankValidator implements <1> ConstraintValidator<CannotHaveBlank, String> {

    @Override
    public void initialize(CannotHaveBlank constraintAnnotation) {
    }


    @Override
    public boolean isValid(String value, ConstraintValidatorContext context <2>) {
        //null时不进行校验
        if (value != null && value.contains(" ")) {
            <3>
            //获取默认提示信息
            String defaultConstraintMessageTemplate = context.getDefaultConstraintMessageTemplate();
            System.out.println("default message :" + defaultConstraintMessageTemplate);
            //禁用默认提示信息
            context.disableDefaultConstraintViolation();
            //设置提示语
            context.buildConstraintViolationWithTemplate("can not contains blank").addConstraintViolation();
            return false;
        }
        return true;
    }
}
```

<1> 所有的验证者都需要实现ConstraintValidator接口，它的接口也很形象，包含一个初始化事件方法，和一个判断是否合法的方法。

```
public interface ConstraintValidator<A extends Annotation, T> {

    void initialize(A constraintAnnotation);

    boolean isValid(T value, ConstraintValidatorContext context);
}
```

<2> ConstraintValidatorContext 这个上下文包含了认证中所有的信息，我们可以利用这个上下文实现获取默认错误提示信息，禁用错误提示信息，改写错误提示信息等操作。

<3> 一些典型校验操作，或许可以对你产生启示作用。

值得注意的一点是，自定义注解可以用在`METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER`之上，ConstraintValidator的第二个泛型参数T，是需要被校验的类型。

手动校验

可能在某些场景下需要我们手动校验，即使用校验器对需要被校验的实体发起validate，同步获得校验结果。理论上我们既可以使用Hibernate Validation提供Validator，也可以使用Spring对其的封装。在spring构建的项目中，提倡使用经过spring封装过后的方法，这里两种方法都介绍下：

Hibernate Validation：

```
Foo foo = new Foo();
foo.setAge(22);
foo.setEmail("000");
ValidatorFactory vf = Validation.buildDefaultValidatorFactory();
```

```
Validator validator = vf.getValidator();
Set<ConstraintViolation<Foo>> set = validator.validate(foo);
for (ConstraintViolation<Foo> constraintViolation : set) {
    System.out.println(constraintViolation.getMessage());
}
```

由于依赖了Hibernate Validation框架，我们需要调用Hibernate相关的工厂方法来获取validator实例，从而校验。

在spring framework文档的Validation相关章节，可以看到如下的描述：

> Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303/JSR-349 Bean Validation provider as a Spring bean. This allows for a javax.validation.ValidatorFactory or javax.validation.Validator to be injected wherever validation is needed in your application. Use the LocalValidatorFactoryBean to configure a default Validator as a Spring bean:
> bean id=" validator" class=" org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"
> The basic configuration above will trigger Bean Validation to initialize using its default bootstrap mechanism. A JSR-303/JSR-349 provider, such as Hibernate Validator, is expected to be present in the classpath and will be detected automatically.

上面这段话主要描述了spring对validation全面支持JSR-303、JSR-349的标准，并且封装了LocalValidatorFactoryBean作为validator的实现。值得一提的是，这个类的责任其实是非常重大的，他兼容了spring的validation体系和hibernate的validation体系，也可以被开发者直接调用，代替上述的从工厂方法中获取的hibernate validator。由于我们使用了springboot，会触发web模块的自动配置，LocalValidatorFactoryBean已经成为了Validator的默认实现，使用时只需要自动注入即可。

```
@Autowired
Validator globalValidator; <1>

@RequestMapping("/validate")
public String validate() {
    Foo foo = new Foo();
    foo.setAge(22);
    foo.setEmail("000");

    Set<ConstraintViolation<Foo>> set = globalValidator.validate(foo);<2>
    for (ConstraintViolation<Foo> constraintViolation : set) {
        System.out.println(constraintViolation.getMessage());
    }

    return "success";
}
```

<1> 真正使用过Validator接口的读者会发现有两个接口，一个是位于javax.validation包下，另一个位于org.springframework.validation包下，注意我们这里使用的是前者javax.validation，后者是spring自己内置的校验接口，LocalValidatorFactoryBean同时实现了这两个接口。

<2> 此处校验接口最终的实现类便是LocalValidatorFactoryBean。

基于方法校验

```
@RestController
@Validated <1>
public class BarController {

    @RequestMapping("/bar")
    public @NotBlank <2> String bar(@Min(18) Integer age <3>) {
        System.out.println("age : " + age);
        return "";
    }

    @ExceptionHandler(ConstraintViolationException.class)
    public Map handleConstraintViolationException(ConstraintViolationException cve){
        Set<ConstraintViolation<?>> cves = cve.getConstraintViolations();<4>
        for (ConstraintViolation<?> constraintViolation : cves) {
            System.out.println(constraintViolation.getMessage());
        }
```

```
18            Map map = new HashMap();
19            map.put("errorCode",500);
20            return map;
21        }
22
    }
```

<1> 为类添加@Validated注解

<2> <3> 校验方法的返回值和入参

<4> 添加一个异常处理器，可以获得没有通过校验的属性相关信息

基于方法的校验，个人不推荐使用，感觉和项目结合的不是很好。

使用校验框架的一些想法

理论上spring validation可以实现很多复杂的校验，你甚至可以使你的Validator获取ApplicationContext，获取spring容器中所有的资源，进行诸如数据库校验，注入其他校验工具，完成组合校验（如前后密码一致）等等操作，但是寻求一个易用性和封装复杂性之间的平衡点是我们作为工具使用者应该考虑的，我推崇的方式，是仅仅使用自带的注解和自定义注解，完成一些简单的，可复用的校验。而对于复杂的校验，则包含在业务代码之中，毕竟如用户名是否存在这样的校验，仅仅依靠数据库查询还不够，为了避免并发问题，还是得加上唯一索引之类的额外工作，不是吗?