第5章

数据验证


视频讲解

数据验证可分为以下两种方式。

(1) 客户端验证(也称为前端验证)：在客户的浏览器端执行 JavaScript 等脚本代码，对用户输入的表单数据等进行数据验证。

(2) 服务器端验证(也称为后端验证)：在服务器端执行程序代码，对客户提供的请求数据或者其他业务数据进行验证。

客户端验证的优点是迅速、便捷，由于直接在浏览器端执行，因此具有更快的响应速度，并且能减轻服务器端的工作负荷；缺点是数据验证的能力有限，而且不能确保通过验证的数据会完整地传送到服务器端。

服务器端验证适用于以下三种场合。

(1) 对某些数据的验证涉及访问服务器端的各种资源(如访问数据库)，在浏览器端无法完成验证。

(2) 客户端进行验证通过的数据在经过网络传输时，有可能被非法篡改，为了安全，需要在服务器端再次对数据进行验证。

(3) 一些非法用户没有通过常规的浏览器程序访问服务器端，而是直接编写黑客程序，向服务器端发送非法的请求数据。

本书介绍的数据验证指的是服务器端验证。第 2 章的 helloapp 应用范例已经介绍了数据验证的基本执行流程。本章将进一步介绍 Spring MVC 框架所支持的数据验证方式，主要包括以下两种。

(1) 按照 JSR-303 规范进行数据验证。

(2) 使用 Spring 框架自身提供的数据验证机制。

本章范例也位于 helloapp 应用中，主要包括以下 4 个组件。

(1) 视图组件：hello.jsp。它负责生成 HTML 表单，并且会显示数据验证产生的错误消息。

(2) 控制器组件：PersonController 类。它负责对表单进行数据验证。

(3) 模型组件：Person 类。它利用数据验证注解声明对特定的属性进行验证。

(4) 数据验证组件：Minimal 类，它是自定义的数据验证注解类型，MinimalValidator 类是实现@Minimal 注解的验证功能的数据验证类；PersonValidator 类，它是实现了 Spring 的 Validator 接口的数据验证类。

5.1 按照 JSR-303 规范进行数据验证

JSR-303(Java Specification Request 303，Java 规范提案 303)是 Java 领域的标准数据验证规范。JSR-303 API 位于 Java EE 类库的 javax. validation 包以及子包中。JSR-303 API 主要定义了一系列用于数据验证的注解，但是它并没有真正实现数据验证功能。

Hibernate Validator 验证器是 JSR-303 API 的具体实现，并且扩展了数据验证功能，提供了如@Email 等实用的数据验证注解。

在 Web 应用中，联合使用 JSR-303 API 和 Hibernate Validator 验证器，就能进行数据验证。

5.1.1 数据验证注解

所有的数据验证注解都有一个 message 属性，用来指定验证失败的错误消息。message 属性有以下两种赋值方式。

(1) 把错误消息的编号赋值给 message 属性。

(2) 直接把错误消息文本赋值给 message 属性。

以下代码通过这两种方式为 message 属性赋值。

```
//第一种方式:指定错误消息的编号
@NotBlank(message ="{person.no.username.error}")
private String userName;

//第二种方式:指定错误消息文本
@NotBlank(message = "UserName can't be empty.")
private String userName;
```

表 5-1 列出了 JSR-303 提供的数据验证注解，它们可以对字符串、布尔、集合、数字、日期时间等类型的数据进行验证。表 5-1 注解的括号内列出的是注解的主要属性，例如@Min(value)注解的 value 指的是@Min 注解的 value 属性。

表 5-1 JSR-303 提供的数据验证注解

数据验证注解	描 述
@Null	待验证数据必须为 null
@NotNull	待验证数据必须不为 null
@AssertTrue	待验证数据为布尔类型，并且必须为 true
@AssertFalse	待验证数据为布尔类型，并且必须为 false
@Min(value)	待验证数据必须大于或等于 value

续表

数据验证注解	描　　述
@Max(value)	待验证数据必须小于或等于 value
@DecimalMin(value)	待验证数据必须大于或等于 value
@DecimalMax(value)	待验证数据必须小于或等于 value
@Size(min,max)	待验证数据如果是 String 类型,那么其长度必须大于或等于 min,并且小于或等于 max;待验证数据如果是集合、Map 或数组类型,那么其包含的元素数目必须大于或等于 min,并且小于或等于 max
@Digits(integer,fraction)	待验证数据必须是数字,integer 指定数字的整数部分的最大位数,fraction 指定数字的小数部分的最大位数。例如@Digits(integer=6,fraction=2)表示数字的整数部分最多 6 位,小数部分最多 2 位
@Past	待验证数据为日期时间类型,并且必须小于当前日期时间
@Future	待验证数据为日期时间类型,并且必须大于当前日期时间
@Pattern(regex)	待验证数据必须符合特定的正则表达式,regex 指定正则表达式
@Valid	对待验证数据以及所关联的数据进行递归验证,参见 5.1.4 节

@Min(value)、@Max(value)、@DecimalMin(value)以及@DecimalMax(value)注解都能判断待验证数据是否小于或等于、大于或等于 value。它们的区别如下。

(1) @Min(value)和@Max(value)的 value 属性为 long 类型,例如@Min(value=6)表示待验证数据必须大于或等于 6。

(2) @DecimalMin(value)和@DecimalMax(value)的 value 属性为 String 类型,例如@ DecimalMin(value="6.0")表示待验证数据必须大于或等于 6.0。

如果要了解 JSR-303 提供的数据验证注解的更详细用法,可以参考 Oracle 官网提供的 JavaDoc 文档,参见图 5-1。

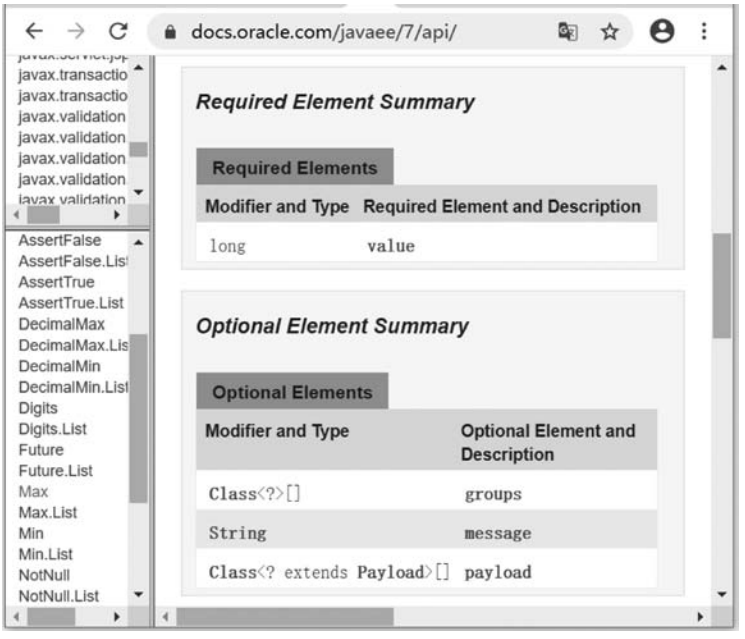

图 5-1　JSR-303 提供的数据验证注解的 JavaDoc 文档

Hibernate Validator 验证器也提供了一些实用的数据验证注解，参见表 5-2。

表 5-2 Hibernate Validator 提供的数据验证注解

数据验证注解	描 述
@NotBlank	待验证数据为 String 类型，去除两端空格后必须不为空。假定变量 data 为待验证数据，判断条件为(data ! =null) &&(data.trim().length()>0)
@NotEmpty	待验证数据如果是 String 类型，必须不为空。假定变量 data 为待验证数据，判断条件为(data !=null) &&(data.length()>0)；待验证数据如果是集合、Map 或数组类型，那么其包含的元素的数目必须大于 0
@Length(min,max)	待验证数据为 String 类型，其长度必须大于或等于 min，小于或等于 max
@Range(min,max)	待验证数据为数字类型或可以转换为数字的 String 类型，其数值必须大于或等于 min，小于或等于 max
@URL	待验证数据必须是有效的 URL
@Email	待验证数据必须是有效的 Email
@CreditCardNumber	待验证数据必须是有效的信用卡卡号

如果要了解 Hibernate Validator 验证器提供的数据验证注解的更详细用法，可以参考 Hibernate 提供的官方 JavaDoc 文档，网址为 https://docs.jboss.org/hibernate/validator/7.0/api/。

在本章范例的 Person 类中，使用了来自 JSR-303 和 Hibernate Validator 验证器的各种数据验证注解。例程 5-1 是 Person 类的部分代码。

例程 5-1 Person 类的部分代码

```
public class Person{
  @NotBlank(message = "{person.no.username.error}")
  private String userName;

  @Size(min = 6,max = 6,message = "{person.tooshort.password.error}")
  private String password;

  @Email(message = "{person.invalid.email.error}")
  private String email;

  @DecimalMin(value = "0.0",message = "{person.invalid.salary.error}")
  private double salary;

  @Minimal(value = 1,message = "{person.invalid.age.error}")
  private int age;
  …
}
```

Person 类除了使用@Size 和@Email 等来自 JSR-303 和 Hibernate Validator 验证器的注解，还使用了一个自定义的数据验证注解@Minimal，5.1.2 节将介绍如何创建自定义的

数据验证注解。

5.1.2 自定义数据验证注解

JSR-303 和 Hibernate Validator 提供的数据验证注解是有限的。为了能实现特定的数据验证功能,JSR-303 还支持开发人员创建自定义的数据验证注解,包括以下两个步骤。

(1) 创建自定义注解类型,在本范例中为 Minimal 类。

(2) 创建数据验证实现类,在本范例中为 MinimalValidator 类。

1. 创建 Minimal 自定义注解类型

例程 5-2 是自定义的注解类型,它用@Constraint 注解标识。来自 javax.validation 包的@Constraint 注解表明 Minimal 类是用于数据验证的注解类型。@Constraint 注解的 validatedBy 属性指定实现数据验证功能的数据验证类。

例程 5-2 Minimal.java

```
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MinimalValidator.class)
public @interface Minimal {

  int value() default 0;

  String message();

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

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

2. 创建 MinimalValidator 数据验证实现类

例程 5-3 实现了 javax.validation.ConstraintValidator 接口,为@Minimal 注解实现具体的数据验证功能。

例程 5-3 MinimalValidator.java

```
package mypack;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class MinimalValidator
              implements ConstraintValidator<Minimal, Integer> {
  private int minValue;

  public void initialize(Minimal min) {
    //把 Minimal 注解的 value 属性赋值给成员变量 minValue
    minValue = min.value();
  }
```

```
    public boolean isValid(Integer value,
                           ConstraintValidatorContext context) {
      //value 参数表示被检验的数据
      return value >= minValue;
    }
}
```

@Minimal 注解的数据验证逻辑和 JSR-303 的@Min 注解相同，因此，在 Person 类中可以用@Minimal 注解或者@Min 注解来标识 age 属性，例如：

```
@Minimal(value = 1,message = "{person.invalid.age.error}")
private int age;
```

或者：

```
@Min(value = 1,message = "{person.invalid.age.error}")
private int age;
```

5.1.3　在 Spring MVC 的配置文件中配置 Hibernate Validator 验证器

为了把 Hibernate Validator 整合到 Spring MVC 框架中，需要在 Spring MVC 的配置文件中进行如下配置：

```
<bean id = "hibernateValidator"
        class = "org.springframework.validation.beanvalidation
                    .LocalValidatorFactoryBean">

  <property name = "providerClass"
          value = "org.hibernate.validator.HibernateValidator" />
  <property name = "validationMessageSource" ref = "messageSource" />
</bean>

<mvc:annotation-driven validator = "hibernateValidator" />
```

<bean>元素向 Spring MVC 框架注册了 Hibernate Validator 验证器。<mvc:annotation-driven>的 validator 属性指向这个数据验证器。这样，当程序需要进行数据验证时，Spring MVC 框架就会利用 Hibernate Validator 验证器完成实际的数据验证功能。

提示：无论是使用了来自 Hibernate Validator 的数据验证注解，还是使用了来自 JSR-303 或者自定义的数据验证注解，都必须在 Spring MVC 的配置文件中配置 Hibernate Validator 验证器，这样才能保证这些注解的正常工作。

5.1.4　在控制器类中进行数据验证

在 PersonController 类中，用@Valid 注解来标识 greet()方法的 person 参数，例如：

```
public String greet(
            @Valid @ModelAttribute("personbean")Person person,
            BindingResult bindingResult,Model model){ … }
```

@Valid 注解来自 JSR-303 API 的 javax. validation 包。@Valid 注解的作用是对当前数据验证时,递归验证与当前数据关联的数据。例如在对 greet()方法的 person 参数进行数据验证时,会递归验证它所引用的 Person 对象的所有属性,假如 person 参数引用的 Person 对象包含集合属性,那么还会对集合中的元素进行递归验证。

此外,在控制器类的方法中,还可以直接编写数据验证代码。例如,在以下 greet()方法中,还会对 Person 对象的 userName 属性做进一步的数据验证,要求 userName 属性中不能包含 Monster 字符串。

```
@RequestMapping(value = "/sayHello",
                              method = RequestMethod.POST)
public String greet(
      @Valid @ModelAttribute("personbean")Person person,
      BindingResult bindingResult,Model model) {

  //直接在控制器类中提供的数据验证
  if (person.getUserName()!= null &&
      person.getUserName().indexOf("Monster")!= -1) {

      bindingResult.rejectValue("userName",
                        "person.forbidden.username.error");
  }

  if(bindingResult.hasErrors()){
      return "hello";
  }

  //调用 Person 对象的 save()方法把 Person 对象保存到数据库中
  person.save();

  return "hello";
}
```

org. springframework. validation. BindingResult 接口的父接口是 org. springframework. validation. Errors。Errors 接口的 rejectValue()方法生成错误消息,它有以下两种重载方法。

(1) rejectValue(String field, String errorCode):参数 errorCode 指定错误消息编号。

(2) rejectValue(String field, String errorCode, String defaultMessage):参数 defaultMessage 指定默认的错误消息文本。

5.1.5 在 JSP 文件中指定显示错误消息的 CSS 样式

在 JSP 文件中通过< form:errors >标签输出错误消息,它的 cssStyle 属性和 cssClass

属性指定显示错误消息的 CSS 样式。

1. 用 cssStyle 属性指定显示错误消息的 CSS 样式

使用 cssStyle 属性设定 CSS 样式比较简单，例如以下代码指定用红色字体显示错误消息。

```
<form:errors path="userName" cssStyle="color:red" />
```

2. 用 cssClass 属性指定显示错误消息的 CSS 样式

以下 JSP 代码指定用 error_class 样式显示错误消息。

```
<link rel="STYLESHEET" type="text/css"
  href="${pageContext.request.contextPath}/resource/css/error.css" >
...
<form:errors path="userName" cssClass="error_class" />
```

在 error.css 文件中定义了 error_class 样式。error.css 文件的内容如下：

```
.error_class {
  FONT-SIZE: 11px;
  COLOR: #FF0000;
}
```

error.css 是静态资源文件，error.css 的真实文件路径为 helloapp/css/error.css。在 Spring MVC 的配置文件中进行了如下的相应配置：

```
<mvc:resources location="/" mapping="/resource/**" />
```

对于本范例的 hello.jsp，当用户在表单中输入了不合法的数据，hello.jsp 会在网页上显示如图 5-2 所示的数据验证错误消息。

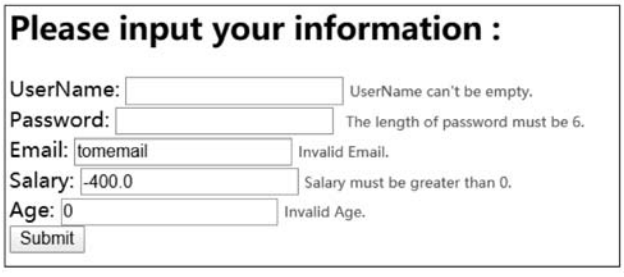

图 5-2 hello.jsp 显示的数据验证错误消息

5.2 Spring 框架的数据验证机制

Spring 框架本身也提供了一套数据验证机制。运用这套验证机制需要以下两个步骤。

(1) 创建实现 org.springframework.validation.Validator 接口的数据验证类。在本范

例中为 PersonValidator 类。

(2) 在 Spring MVC 框架中创建数据验证类的对象并通过它进行数据验证。

5.2.1 实现 Spring 的 Validator 接口

Spring API 提供了一个数据验证接口：org.springframework.validation.Validator 接口。例程 5-4 实现了 Validator 接口。PersonValidator 类会验证 Person 类的 userName 属性和 password 属性。

例程 5-4 PersonValidator.java

```
package mypack;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class PersonValidator implements Validator {
  public boolean supports(Class <?> clazz) {
      return Person.class.equals(clazz);
  }

  public void validate(Object obj, Errors errors) {
    ValidationUtils.rejectIfEmpty(errors, "userName",
                                  "person.no.username.error");

    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
                                  "person.no.password.error");

    Person person = (Person) obj;
    if(person.getPassword() != null && person.getPassword()!= ""
                && person.getPassword().length()!= 6){
      errors.rejectValue("password",
                         "person.tooshort.password.error");
    }
  }
}
```

PersonValidator 类实现了 Validator 接口的以下两种方法。

(1) supports()方法：指定 PersonValidator 类所验证的数据类型。

(2) validate()方法：进行数据验证。

validate()方法首先利用 ValidationUtils 类来验证 Person 对象的 userName 属性和 password 属性。ValidationUtils 类是 Spring API 提供的用于数据验证的实用类，它有以下两种常用的静态方法。

(1) rejectIfEmpty()：如果待验证数据为空，就生成错误消息。验证逻辑等同于 Hibernate Validator 验证器的@NotEmpty 注解。

(2) rejectIfEmptyOrWhitespace()：如果待验证数据去除两端空格后为空，就生成错

误消息。验证逻辑等同于 Hibernate Validator 验证器的@NotBlank 注解。

ValidationUtils 类的 rejectIfEmpty()和 rejectIfEmptyOrWhitespace()都有一些重载方法。以下是 rejectIfEmpty()的两种常用的重载方法。

(1) rejectIfEmpty(Errors errors, String field, String errorCode)：参数 errorCode 指定错误消息编号。

(2) rejectIfEmpty(Errors errors, String field, String errorCode, String defaultMessage)：参数 defaultMessage)：参数 defaultMessage 指定默认的错误消息文本。

5.2.2 用数据验证类进行数据验证

创建以及使用 PersonValidator 对象有以下三种方式。

1. 把 PersonValidator 对象和 PersonController 的当前 DataBinder 对象绑定

在 PersonController 类中，以下 initBinder()方法会把 PersonBinder 对象加入到当前的 DataBinder 对象中。

```
@InitBinder
public void initBinder(DataBinder binder) {
  binder.setValidator(new PersonValidator());
}
```

org.springframework.validation.DataBinder 类是 Spring 框架提供的用来存放数据验证对象的容器类。initBinder()方法用@InitBinder 注解标识，Spring MVC 框架在每次调用 PersonController 的请求处理方法之前，先调用 initBinder()方法创建 PersonValidator 对象，并把它与当前的 DataBinder 对象绑定。这个 PersonValidator 对象专门为 PersonController 的请求处理方法进行数据验证。

PersonController 类的请求处理方法可以通过 JSR-303 的@Valid 注解来声明对 person 参数进行数据验证，例如：

```
public String greet(
        @Valid @ModelAttribute("personbean")Person person,
        BindingResult bindingResult,Model model){…}
```

@Valid 注解声明要对 person 参数进行验证，Spring MVC 框架就会利用与当前 DataBinder 对象绑定的 PersonValidator 进行相应的数据验证。

2. 注册 PersonValidator Bean 组件，并把它设为全局的数据验证类

在 Spring MVC 的配置文件中配置 PersonValidator 类的代码如下：

```
<bean id="personValidator" class="mypack.PersonValidator"/>
<mvc:annotation-driven validator="personValidator" />
```

<bean>元素向 Spring MVC 框架注册了 PersonValidator Bean 组件，<mvc:annotation-driven>的 validator 属性指向这个 PersonValidator Bean 组件。这样，当控制器类需要对 Person

类的数据进行验证时,Spring MVC 框架就会利用 PersonValidator 对象来完成实际的数据验证功能。按照这种方式配置的 PersonValidator 对象的数据验证范围是整个 Web 应用。

PersonController 类的请求处理方法通过 JSR-303 的@Valid 注解来声明对 person 参数进行数据验证,例如:

```
public String greet(
        @Valid @ModelAttribute("personbean")Person person,
        BindingResult bindingResult,Model model){…}
```

@Valid 注解声明要对 person 参数进行验证,Spring MVC 框架就会利用全局范围内的 PersonValidator 类进行相应的数据验证。

值得注意的是,PersonValidator 类并没有提供通用的数据验证功能,它只能对 Person 类的数据进行验证,实际上并不推荐把它作为全局的数据验证类。

3. 注册 PersonValidator Bean 组件,PersonController 通过@Resource 注解访问它

在 Spring MVC 的配置文件中配置 PersonValidator 类的代码如下:

```
<bean id="personValidator" class="mypack.PersonValidator"/>
```

<bean>元素向 Spring MVC 框架注册了 PersonValidator Bean 组件,Spring MVC 框架会负责管理 PersonValidator 对象的生命周期。

在 PersonController 类中,通过来自 javax.annotation 包的@Resource 注解访问由 Spring MVC 框架提供的 PersonValidator 对象,例如:

```
@Controller
public class PersonController {
  @Resource          //由 Spring MVC 框架提供 PersonValidator 对象
  private PersonValidator personValidator;

  @RequestMapping(value = {"/input","/"},method = RequestMethod.GET)
  public String init(Model model) {
    model.addAttribute("personbean",new Person());
    return "hello";
  }

  @RequestMapping(value = "/sayHello", method = RequestMethod.POST)
  public String greet(
            @ModelAttribute("personbean")Person person,
            BindingResult bindingResult,Model model) {

    personValidator.validate(person,bindingResult);

    if(bindingResult.hasErrors()){
      return "hello";
    }
    …
  }
}
```

PersonController 类的 personValidator 成员变量用@Resource 注解标识，意味着 PersonController 类无须创建 PersonValidator 对象，Spring MVC 框架会把 PersonValidator Bean 组件赋值给 personValidator 成员变量。

5.3 小结

本章介绍了数据验证的各种方式。表 5-3 比较了各种验证方式的特点以及优缺点。

表 5-3 各种数据验证方式的特点以及优缺点

数据验证方式	优 缺 点
直接在控制器类中编写数据验证代码，参见 5.1.4 节	比较灵活，可以在请求处理方法的任何地方进行数据验证；数据验证代码分散在各处，代码的可重用性和可维护性差
按照 JSR-303 规范进行数据验证	具有很好的通用性和可重用性；数据验证注解主要用来对 JavaBean 类的属性进行验证，不能在程序的任何地方使用该数据验证功能
使用 Spring 框架的数据验证机制	可以通过编程的方式实现复杂的数据验证逻辑，在数据验证类中可以方便地访问 Spring 框架提供的各种资源；必须亲自编写程序代码来实现各种数据验证逻辑

对于规模比较小的 Web 应用，直接在控制器类中编写数据验证代码更加灵活、方便，避免了配置数据验证类的麻烦。

对于大型 Web 应用，为了促进软件应用的模块化和层次化，可以联合使用 JSR-303 数据验证和 Spring 的自带验证机制：

(1) 用 JSR-303 的数据验证注解对常见的数据类型进行通用的数据验证。

(2) 对于部分需要定制的数据验证逻辑，则通过实现 Spring 的 Validator 接口来完成。

把这两种验证方式结合，就能满足各种数据验证的需求。

5.4 思考题

1. (　　)属于 JSR-303 的数据验证注解。(多选)

   A. @NotNull　　B. @Email

   C. @Past　　D. @Size

2. (　　)用@NotEmpty 注解进行数据验证会通过验证。(多选)

   A. ""　　B. "　　"

   C. null　　D. "　hello　"

3. 按照 JSR-303 规范创建自定义的数据验证注解时，完成实际验证功能的类应该实现或者继承(　　)。(单选)

   A. javax. validation. Valid

   B. javax. validation. ConstraintValidator

   C. org. springframework. validation. Validator

D. org. springframework. validation. ValidationUtils

4. (　　)属于 org. springframework. validation. Validator 接口的方法。(多选)

A. validate()　　B. supports()

C. message()　　D. value()

5. 对于 BindingResult 类型的 bindingResult 变量,(　　)合法调用了它的 rejectValue()方法。(多选)

A. bindingResult. rejectValue("age");

B. bindingResult. rejectValue("age", "age. error");

C. bindingResult. rejectValue("age", null,"Invalid age");

D. bindingResult. rejectValue("age", "age. error","Invalid age");