第2-1课：Spring Boot 对基础 Web 开发的支持（下）

接着上一课我们继续讲解。

数据校验

在很多时候，当我们要处理一个应用程序的业务逻辑时，数据校验是必须要考虑和面对的事情。应用程序必须通过某种手段来确保输入进来的数据从语义上来讲是正确的。在 Java 应用程序中，必须要对输入进来的数据从语义上分析是有效的，也就是数据校验。

输入验证是最重要的 Web 开发任务之一，在 Spring MVC 中有两种方式可以验证输入：一种是 Spring 自带的验证框架，另外一种是利用 JSR 实现。

JSR 是一个规范文档，指定了一整套 API，通过标注给对象属性添加约束。Hibernate Validator 就是 JSR 规范的具体实现，Hibernate Validator 提供了 JSR 规范中所有内置约束注解的实现，以及一些附加的约束注解，除此之外用户还可以自定义约束注解。

Spring Boot 的参数校验依赖于 hibernate-validator 来进行。使用 Hibernate Validator 校验数据，需要定义一个接收的数据模型，使用注解的形式描述字段校验的规则，我们以 User 对象为例为大家演示如何使用。

首先在 WebController 添加一个保存的方法 saveUser，参数为 User。

```
@RequestMapping("/saveUser")
public void saveUser(@Valid User user,BindingResult result) {
    System.out.println("user:"+user);
    if(result.hasErrors()) {
        List<ObjectError> list = result.getAllErrors();
        for (ObjectError error : list) {
            System.out.println(error.getCode()+ "-" + error.getDefaultMessage());
        }
    }
}
```

- @Valid 参数前面添加 @Valid 注解，代表此对象使用了参数校验；
- BindingResult 参数校验的结果会存储在此对象中，可以根据属性判断是否校验通过，校验不通过可以将错误信息打印出来。

接下来在 User 中给需要校验的参数添加对应的注解，对不同的属性，按照规则添加不同的校验内容。

```
public class User {
    @NotEmpty(message="姓名不能为空")
    private String name;
    @Max(value = 100, message = "年龄不能大于100岁")
    @Min(value= 18 ,message= "必须年满18岁！" )
    private int age;
    @NotEmpty(message="密码不能为空")
    @Length(min=6,message="密码长度不能小于6位")
    private String pass;
    //...
}
```

其中，`message="密码不能为空"`，为自定义返回的错误信息。

Hibernate Validator 基本上包含了常用的数据校验，包括校验属性是否为空、长度、大小、特定格式等，完整的注解可以看下表：

注解	应用目标	运行时检查	Hibernate 元数据影响
@Length(min=, max=)	属性（String）	检查字符串长度是否符合范围	列长度会被设到最大值
@Max(value=)	属性（以 numeric 或者 string 类型来表示一个数字）	检查值是否小于或等于最大值	对列增加一个检查约束
@Min(value=)	属性（以 numeric 或者 string 类型来表示一个数字）	检查值是否大于或等于最小值	对列增加一个检查约束
@NotNull	属性	检查值是否非空（not null）	列不为空
@Past	属性（date 或 calendar）	检查日期是否是过去时	对列增加一个检查约束
@Future	属性（date 或 calendar）	检查日期是否是将来时	无

@Pattern(regex="regexp", flag=)	属性（string）	检查属性是否与给定匹配标志的正则表达式相匹配（见 `java.util.regex.Pattern` ）	无
@Range(min=, max=)	属性（以 numeric 或者 string 类型来表示一个数字）	检查值是否在最小和最大值之间（包括临界值）	对列增加一个检查约束
@Size(min=, max=)	属性（array, collection, map）	检查元素大小是否在最小和最大值之间（包括临界值）	无
@AssertFalse	属性	检查方法的演算结果是否为 false（对以代码方式而不是注解表示的约束很有用）	无
@AssertTrue	属性	检查方法的演算结果是否为 true（对以代码方式而不是注解表示的约束很有用）	无
@Valid	属性（object）	对关联对象递归进行验证。如果对象是集合或数组，就递归地验证其元素；如果对象是 Map，则递归验证其值元素	无
@Email	属性（String）	检查字符串是否符合有效的 email 地址规范	无

添加测试方法，对属性校验进行测试：

```
@Test
public void saveUsers() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post("/saveUser")
            .param("name","")
            .param("age","666")
            .param("pass","test")
    );
}
```

结果返回：

```
user:name=,age=666,pass=test
Max-年龄不能大于100岁
Length-密码长度不能小于6位
NotEmpty-姓名不能为空
```

结果显示均已经触发了校验规则，返回了错误信息，在实际使用过程中可以对错误信息进行包装，最后返回到前端进行展示。

自定义 Filter

Filter 也称之为过滤器，可以在前端拦截所有用户的请求，可以认为是 Servlet 的一种加强版，Web 开发人员通过 Filter 技术，对 Web 服务器管理的所有 Web 资源，例如 JSP、Servlet、静态图片文件或静态 HTML 文件等进行拦截，从而实现一些特殊的功能。例如，实现 URL 级别的权限访问控制、过滤敏感词汇、排除有 XSS 威胁的字符、记录请求日志等一些高级功能。

Spring Boot 内置了一些 Filter，比如，处理编码的 OrderedCharacterEncodingFilter 和请求转化的 HiddenHttpMethodFilter，也支持根据我们的需求来可以自定义 Filter。

自定义 Filter 有两种实现方式，第一种是使用 @WebFilter，第二种是使用 FilterRegistrationBean，经过实践之后发现使用 @WebFilter 自定义的过滤器优先级顺序不能生效，因此推荐使用第二个方案，接下来我们详细介绍第二种方案。

自定义 Filter 两个步骤：

- 实现 Filter 接口，实现其中的 doFilter() 方法；
- 添加 @Configuration 注解，将自定义 Filter 加入过滤链。

新建 MyFilter 类，重写 doFilter() 方法：

```
public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig arg0) throws ServletException {
        // TODO Auto-generated method stub
    }

    @Override
    public void doFilter(ServletRequest srequest, ServletResponse sresponse, Filte
rChain filterChain)
            throws IOException, ServletException {
        // TODO Auto-generated method stub
        HttpServletRequest request = (HttpServletRequest) srequest;
        System.out.println("this is MyFilter,url :"+request.getRequestURI());
        filterChain.doFilter(srequest, sresponse);
    }

    @Override
    public void destroy() {
        // TODO Auto-generated method stub
    }

}
```

将自定义 Filter 加入过滤链：

```
@Configuration
public class WebConfiguration {
    @Bean
    public FilterRegistrationBean testFilterRegistration() {

        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(new MyFilter());
        registration.addUrlPatterns("/*");
        registration.setName("MyFilter");
        registration.setOrder(6);
        return registration;
    }
}
```

添加完后启动项目，在浏览器中输入地址：http://localhost:8080/getUsers，就会看到控制台打印如下信息：

```
this is MyFilter, url :/xxx
```

说明 MyFilter 已经对所有的 URL 进行了监控。当有多个过滤器时可以通过设置 Order 属性来决定它们的执行顺序，Order 值越小优先级越高。我们复制上面的 MyFilter 重命名为 MyFilter2，在 WebConfiguration 中

添加 MyFilter2 的配置：

```
public FilterRegistrationBean test2FilterRegistration() {
    FilterRegistrationBean registration = new FilterRegistrationBean();
    registration.setFilter(new MyFilter2());
    registration.addUrlPatterns("/*");
    registration.setName("MyFilter2");
    registration.setOrder(1);
    return registration;
}
```

将 MyFilter 的 Order 属性设置为 6，将 MyFilter 2 的 Order 属性设置为 1，重新启动项目，在浏览器中输入地址：http://localhost:8080/getUsers，就会看到控制台打印如下信息：

```
this is MyFilter2,url :/getUsers
this is MyFilter,url :/getUsers
```

可以发现过滤器 MyFilter 2 因为 Order 值设置得低，会优先被执行。

配置文件

在 Web 开发的过程中，经常需要自定义一些配置文件，如何使用呢?

在 application.properties 中配置：

```
neo.title=纯洁的微笑
neo.description=分享技术，品味生活
```

Spring Boot 也支持 Yaml 语法书写，比如上面的配置内容可以写到 application.yml 文件中，格式如下：

```
neo:
 title:  纯洁的微笑
 description:  分享技术，品味生活
```

在选择两种语法时，Yaml 语法更加简洁，properties 使用更加广泛，团队中保持一致即可。内容配置完成后，需要自定义一个对象来接收配置内容。

> 注：同时存在 application.yml 和 application.properties，并且里面配置相同，application.properties 的配置会覆盖 application.yml。

读取单个配置项

当需要从配置文件加载单个配置内容时，只需要使用 @Value 属性即可，新建 PropertiesTest 测试类进行测试。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class PropertiesTest {
    @Value("${neo.title}")
    private String title;

    @Test
    public void testSingle() {
        Assert.assertEquals(title,"纯洁的微笑");
    }
}
```

- @Value("${neo.title}") 会默认读取 application.properties 或者 application.yml 文件中的 neo.title 配置属性值，并赋值给 title。
- Assert.assertEquals 是判断属性值是否和目标值一致。

执行测试用例运行正常，说明属性值加载成功。

读取多个配置

通常在项目中使用配置文件时，往往需要加载多个配置项，比如数据库连接参数等，通常会定义一个对象来接收多个配置项，方便在项目中使用。比如定义一个 NeoProperties 对象，来接收所有以 neo 开头的配置内容。

```
@Component
@ConfigurationProperties(prefix="neo")
public class NeoProperties {
    private String title;
    private String description;

    //省略getter settet方法
}
```

- @Component 的定义为实例，方便在 Spring Boot 项目中引用；
- @ConfigurationProperties(prefix="neo")，表示以 neo 开头的属性会自动赋值到对象的属性中，比如，neo.title 会自动赋值到对象属性 title 中。

写单元测试进行验证，使用属性时直接将 NeoProperties 对象注入即可。

```
@Resource
private NeoProperties properties;

@Test
public void testMore() throws Exception {
    System.out.println("title:"+properties.getTitle());
    System.out.println("description:"+properties.getDescription());
}
```

运行 test 后输出结果：

```
title:纯洁的微笑
description:分享技术，品味生活
```

自定义配置文件

有时候需要自定义配置文件，以便和系统使用的 application.properties 文件区分开，避免混淆。Spring Boot 对这种情况也有很好的支持。

在 resources 目录下创建一个 other.properties 文件，内容如下：

```
other.title=keep smile
other.blog=www.ityouknow.com
```

和上面一样定义一个对象来接收配置文件的内容：

```
@Component
@ConfigurationProperties(prefix="other")
@PropertySource("classpath:other.properties")
public class OtherProperties {
    private String title;
    private String blog;

    //省略getter settet方法
}
```

对比上面读取多个配置示例，多了一个注解来指明配置文件地址：@PropertySource("classpath:other.properties")，同样创建一个测试方法，检测是否正确加载了外部配置文件。

```
@Test
public void testOther() throws Exception {
    System.out.println("title:"+otherProperties.getTitle());
    System.out.println("blog:"+otherProperties.getBlog());
}
```

运行 test 后输出结果：

```
title:my title
blog:www.ityouknow.com
```

说明自定义配置文件加载成功。

如果测试中出现中文乱码，可按照以下方法进行设置：

> 依次单击 File | Settings | Editor | File Encodings 命令，将 Properties Files (*.properties) 下的 Default encoding for properties files 设置为 UTF-8，勾选 Transparent native-to-ascii conversion 复选框。

总结

Spring Boot 集成了参数校验、内嵌容器、Restful、JSON 等内容，这些技术在我们进行 Web 开发中必不可少。Spring Boot 对这些技术进行了包装，让我们在 Web 项目开发过程中非常容易地使用这些技术，降低了开发 Web 项目的技术难度。

> 点击这里下载源码。