告别混乱代码，这份 SpringBoot 后端接口规范太及时了！

java1234 2023-05-03 00:00 Posted on 江苏

大家好，我是锋哥！

文章目录

一、前言

一个后端接口大致分为四个部分组成：**接口地址（url）、接口请求方式（get、post等）、请求数据（request）、响应数据（response）**。虽然说后端接口的编写并没有统一规范要求，而且如何构建这几个部分每个公司要求都不同，没有什么“一定是最好的”标准，但其中最重要的关键点就是看是否规范。

二、环境说明

因为讲解的重点是后端接口，所以需要导入一个`spring-boot-starter-web`包，而lombok作用是简化类，前端显示则使用了knife4j，具体使用在Spring Boot整合knife4j实现Api文档已写明。另外从springboot-2.3开始，校验包被独立成了一个starter组件，所以需要引入如下依赖：

```
<dependency>
<!--新版框架没有自动引入需要手动引入-->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>

<dependency>
    <groupId>com.github.xiaoymin</groupId>
    <artifactId>knife4j-spring-boot-starter</artifactId>
    <!--在引用时请在maven中央仓库搜索最新版本号-->
    <version>2.0.2</version>
</dependency>

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

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

三、参数校验

1、介绍

一个接口一般对参数（请求数据）都会进行安全校验，参数校验的重要性自然不必多说，那么如何对参数进行校验就有讲究了。一般来说有三种常见的校验方式，我们使用了最简洁的第三种方法

- 业务层校验
- Validator + BindResult校验
- Validator + 自动抛出异常

业务层校验无需多说，即手动在java的Service层进行数据校验判断。不过这样太繁琐了，光校验代码就会有很多

而使用`Validator+ BindingResult`已经是非常方便实用的参数校验方式了，在实际开发中也有很多项目就是这么做的，不过这样还是不太方便，因为你每写一个接口都要添加一个BindingResult参数，然后再提取错误信息返回给前端（简单看一下）。

```
@PostMapping("/addUser")
public String addUser(@RequestBody @Validated User user, BindingResult bindingResult) {
    // 如果有参数校验失败，会将错误信息封装成对象组装在BindingResult里
    List<ObjectError> allErrors = bindingResult.getAllErrors();
    if(!allErrors.isEmpty()){
        return allErrors.stream()
            .map(o->o.getDefaultMessage())
            .collect(Collectors.toList()).toString();
    }
    // 返回默认的错误信息
    // return allErrors.get(0).getDefaultMessage();
    return validationService.addUser(user);
}
```

2、Validator + 自动抛出异常（使用）

内置参数校验如下：

注解	校验功能
@AssertFalse	必须是false
@AssertTrue	必须是true
@DecimalMax	小于等于给定的值
@DecimalMin	大于等于给定的值
@Digits	可设定最大整数位数和最大小数位数
@Email	校验是否符合Email格式
@Future	必须是将来的时间
@FutureOrPresent	当前或将来时间
@Max	最大值
@Min	最小值
@Negative	负数（不包括0）
@NegativeOrZero	负数或0
@NotBlank	不为null并且包含至少一个非空白字符
@NotEmpty	不为null并且不为空
@NotNull	不为null
@Null	为null
@Past	必须是过去的时间
@PastOrPresent	必须是过去的时间，包含现在
@PositiveOrZero	正数或0
@Size	校验容器的元素个数

首先Validator可以非常方便的制定校验规则，并自动帮你完成校验。首先在入参里需要校验的字段加上注解,每个注解对应不同的校验规则，并可制定校验失败后的信息：

```
@Data
public class User {
    @NotNull(message = "用户id不能为空")
    private Long id;

    @NotNull(message = "用户账号不能为空")
    @Size(min = 6, max = 11, message = "账号长度必须是6-11个字符")
    private String account;

    @NotNull(message = "用户密码不能为空")
    @Size(min = 6, max = 11, message = "密码长度必须是6-16个字符")
    private String password;

    @NotNull(message = "用户邮箱不能为空")
    @Email(message = "邮箱格式不正确")
    private String email;
}
```

校验规则和错误提示信息配置完毕后，接下来只需要在接口仅需要在校验的参数上加上`@Valid`注解（去掉BindingResult后会自动引发异常，异常发生了自然而然就不会执行业务逻辑）：

```java
@RestController
@RequestMapping("user")
public class ValidationController {

    @Autowired
    private ValidationService validationService;

    @PostMapping("/addUser")
    public String addUser(@RequestBody @Validated User user) {

        return validationService.addUser(user);
    }
}
```

现在我们进行测试，打开**knife4j**文档地址，当输入的请求数据为空时，**Validator**会将所有的报错信息全部进行返回，所以需要与全局异常处理一起使用。

```java
// 使用form data方式调用接口，校验异常抛出 BindException
// 使用 json 请求体调用接口，校验异常抛出 MethodArgumentNotValidException
// 单个参数校验异常抛出ConstraintViolationException
// 处理 json 请求体调用接口校验失败抛出的异常
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResultVO<String> MethodArgumentNotValidException(MethodArgumentNotValidException e) {
    List<FieldError> fieldErrors = e.getBindingResult().getFieldErrors();
    List<String> collect = fieldErrors.stream()
        .map(DefaultMessageSourceResolvable::getDefaultMessage)
        .collect(Collectors.toList());
    return new ResultVO(ResultCode.VALIDATE_FAILED, collect);
}
// 使用form data方式调用接口，校验异常抛出 BindException
@ExceptionHandler(BindException.class)
public ResultVO<String> BindException(BindException e) {
    List<FieldError> fieldErrors = e.getBindingResult().getFieldErrors();
    List<String> collect = fieldErrors.stream()
        .map(DefaultMessageSourceResolvable::getDefaultMessage)
        .collect(Collectors.toList());
    return new ResultVO(ResultCode.VALIDATE_FAILED, collect);
}
```

```
{
  "account": "",
  "email": "1221",
  "id": 0,
  "password": ""
}
```

响应内容 Raw Headers Curl

```
{
  "code": 1002,
  "msg": "参数校验失败",
  "data": [
    "密码长度必须是6-16个字符",
    "账号长度必须是6-11个字符",
    "邮箱格式不正确"
  ]
}
```


3、分组校验和递归校验

分组校验有三个步骤：

- 定义一个分组类（或接口）
- 在校验注解上添加groups属性指定分组
- Controller方法的 `@Validated` 注解添加分组类

```
public interface Update extends Default{
}
```

```
@Data
public class User {
    @NotNull(message = "用户id不能为空",groups = Update.class)
    private Long id;
  ......
}
```

```
@PostMapping("update")
public String update(@Validated({Update.class}) User user) {
    return "success";
}
```

如果Update不继承Default， `@Validated({Update.class})` 就只会校验属于 `Update.class` 分组的参数字段；如果继承了，会校验了其他默认属于 `Default.class` 分组的字段。

对于递归校验（比如类中类），只要在相应属性类上增加 `@Valid` 注解即可实现（对于集合同样适用）

4、自定义校验

Spring Validation允许用户自定义校验，实现很简单，分两步：

- 自定义校验注解
- 编写校验者类

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {HaveNoBlankValidator.class})// 标明由哪个类执行校验逻辑
public @interface HaveNoBlank {

    // 校验出错时默认返回的消息
    String message() default "字符串中不能含有空格";
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { };
    /**
     * 同一个元素上指定多个该注解时使用
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        NotBlank[] value();
    }
}
```

```
public class HaveNoBlankValidator implements ConstraintValidator<HaveNoBlank, String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        // null 不做检验
        if (value == null) {
            return true;
        }
        // 校验失败
        return !value.contains(" ");
        // 校验成功
    }
}
```

四、全局异常处理

参数校验失败会自动引发异常，我们当然不可能再去手动捕捉异常进行处理。但我们又不想手动捕捉这个异常，又要对这个异常进行处理，那正好使用SpringBoot全局异常处理来达到一劳永逸的效果！

1、基本使用

首先，我们需要新建一个类，在这个类上加上 `@ControllerAdvice` 或 `@RestControllerAdvice` 注解，这个类就配置成全局处理类了。

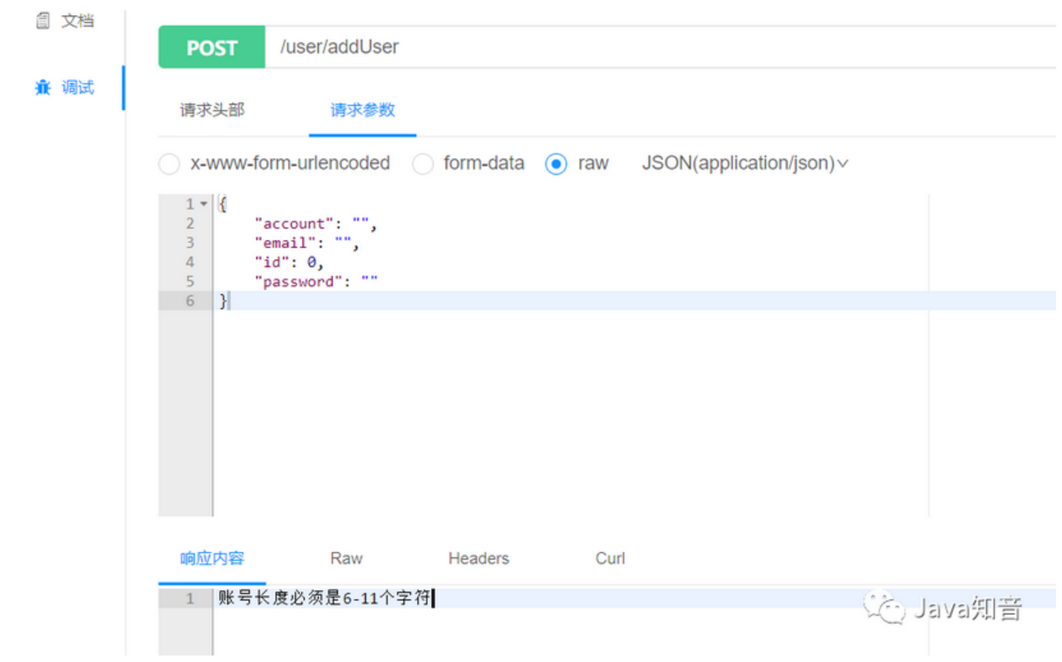
> 这个根据你的Controller层用的是 `@Controller` 还是 `@RestController` 来决定。

然后在类中新建方法，在方法上加上 `@ExceptionHandler` 注解并指定你想处理的异常类型，接着在方法内编写对该异常的操作逻辑，就完成了对该异常的全局处理！我们现在就来演示一下对参数校验失败抛出的 `MethodArgumentNotValidException` 全局处理：

```java
package com.csdn.demo1.global;

import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
@ResponseBody
public class ExceptionControllerAdvice {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public String MethodArgumentNotValidExceptionHandler(MethodArgumentNotValidException e) {
        // 从异常对象中拿到ObjectError对象
        ObjectError objectError = e.getBindingResult().getAllErrors().get(0);
        // 然后提取错误提示信息进行返回
        return objectError.getDefaultMessage();
    }

     /**
     * 系统异常 预期以外异常
     */
    @ExceptionHandler(Exception.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public ResultVO<?> handleUnexpectedServer(Exception ex) {
        log.error("系统异常：", ex);
        // GlobalMsgEnum.ERROR是我自己定义的枚举类
        return new ResultVO<>(GlobalMsgEnum.ERROR);
    }

    /**
     * 所以异常的拦截
     */
    @ExceptionHandler(Throwable.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public ResultVO<?> exception(Throwable ex) {
        log.error("系统异常：", ex);
        return new ResultVO<>(GlobalMsgEnum.ERROR);
    }
}
```

我们再次进行测试，这次返回的就是我们制定的错误提示信息！我们通过全局异常处理优雅的实现了我们想要的功能！

以后我们再想写接口参数校验，就只需要在入参的成员变量上加上Validator校验规则注解，然后在参数上加上 `@Valid` 注解即可完成校验，校验失败会自动返回错误提示信息，无需任何其他代码！



2、自定义异常

在很多情况下，我们需要手动抛出异常，比如在业务层当有些条件并不符合业务逻辑，而使用自定义异常有诸多优点：

- 自定义异常可以携带更多的信息，不像这样只能携带一个字符串。
- 项目开发中经常是很多人负责不同的模块，使用自定义异常可以统一了对外异常展示的方式。
- 自定义异常语义更加清晰明了，一看就知道是项目中手动抛出的异常。

我们现在就来开始写一个自定义异常：

```
package com.csdn.demo1.global;

import lombok.Getter;

@Getter //只要getter方法，无需setter
public class APIException extends RuntimeException {
    private int code;
    private String msg;

    public APIException() {
        this(1001, "接口错误");
    }

    public APIException(String msg) {
        this(1001, msg);
    }

    public APIException(int code, String msg) {
        super(msg);
        this.code = code;
        this.msg = msg;
    }
}
```

然后在刚才的全局异常类中加入如下:

```
//自定义的全局异常
  @ExceptionHandler(APIException.class)
  public String APIExceptionHandler(APIException e) {
      return e.getMsg();
  }
```

这样就对异常的处理就比较规范了，当然还可以添加对Exception的处理，这样无论发生什么异常我们都能屏蔽掉然后响应数据给前端，不过建议最后项目上线时这样做，能够屏蔽掉错误信息暴露给前端，在开发中为了方便调试还是不要这样做。

另外，当我们抛出自定义异常的时候全局异常处理只响应了异常中的错误信息msg给前端，并没有将错误代码code返回。这还需要配合数据统一响应。

如果在多模块使用，全局异常等公共功能抽象成子模块，则在需要的子模块中需要将该模块包扫描加入，`@SpringBootApplication(scanBasePackages = {"com.xxx"})`

五、数据统一响应

统一数据响应是我们自己自定义一个响应体类，无论后台是运行正常还是发生异常，响应给前端的数据格式是不变的！这里我包括了响应信息代码code和响应信息说明msg，首先可以设置一个枚举规范响应体中的响应码和响应信息。

```
@Getter
public enum ResultCode {
    SUCCESS(1000, "操作成功"),
    FAILED(1001, "响应失败"),
    VALIDATE_FAILED(1002, "参数校验失败"),
    ERROR(5000, "未知错误");
    private int code;
    private String msg;
    ResultCode(int code, String msg) {
        this.code = code;
        this.msg = msg;
    }
}
```

自定义响应体

```java
package com.csdn.demo1.global;

import lombok.Getter;

@Getter
public class ResultVO<T> {
    /**
     * 状态码，比如1000代表响应成功
     */
    private int code;
    /**
     * 响应信息，用来说明响应情况
     */
    private String msg;
    /**
     * 响应的具体数据
     */
    private T data;

    public ResultVO(T data) {
        this(ResultCode.SUCCESS, data);
    }

    public ResultVO(ResultCode resultCode, T data) {
        this.code = resultCode.getCode();
        this.msg = resultCode.getMsg();
        this.data = data;
    }
}
```

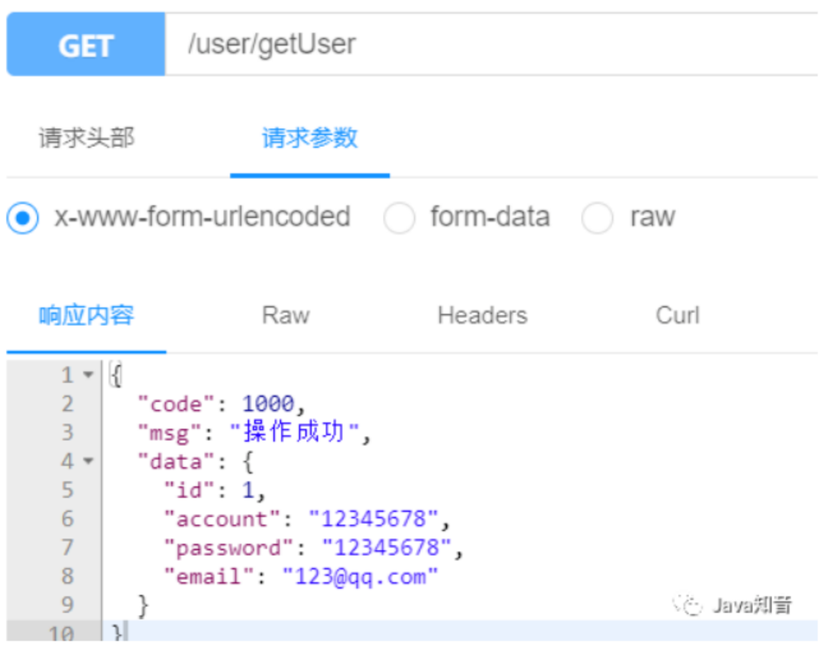
最后需要修改全局异常处理类的返回类型

```java
@RestControllerAdvice
public class ExceptionControllerAdvice {

    @ExceptionHandler(APIException.class)
    public ResultVO<String> APIExceptionHandler(APIException e) {
        // 注意哦，这里传递的响应码枚举
        return new ResultVO<>(ResultCode.FAILED, e.getMsg());
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResultVO<String> MethodArgumentNotValidExceptionHandler(MethodArgumentNotValidException e
        ObjectError objectError = e.getBindingResult().getAllErrors().get(0);
        // 注意哦，这里传递的响应码枚举
        return new ResultVO<>(ResultCode.VALIDATE_FAILED, objectError.getDefaultMessage());
    }
}
```

最后在**controller**层进行接口信息数据的返回

```java
@GetMapping("/getUser")
public ResultVO<User> getUser() {
    User user = new User();
    user.setId(1L);
    user.setAccount("12345678");
    user.setPassword("12345678");
    user.setEmail("123@qq.com");

    return new ResultVO<>(user);
}
```

经过测试，这样响应码和响应信息只能是枚举规定的那几个，就真正做到了响应数据格式、响应码和响应信息规范化、统一化！



还有一种全局返回类如下

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Msg {
    //状态码
    private int code;
    //提示信息
    private String msg;
    //用户返回给浏览器的数据
    private Map<String,Object> data = new HashMap<>();

    public static Msg success() {
        Msg result = new Msg();
        result.setCode(200);
        result.setMsg("请求成功！");
        return result;
    }

    public static Msg fail() {
        Msg result = new Msg();
        result.setCode(400);
        result.setMsg("请求失败！");
        return result;
    }

    public static Msg fail(String msg) {
        Msg result = new Msg();
        result.setCode(400);
        result.setMsg(msg);
        return result;
    }

    public Msg(ReturnResult returnResult){
        code = returnResult.getCode();
        msg = returnResult.getMsg();
    }

    public Msg add(String key,Object value) {
        this.getData().put(key, value);
        return this;
    }
}
```

六、全局处理响应数据(可选择)

接口返回统一响应体 + 异常也返回统一响应体，其实这样已经很好了，但还是有可以优化的地方。要知道一个项目下来定义的接口搞个几百个太正常不过了，要是每一个接口返回数据时都要用响应体来包装一下好像有点麻烦，有没有办法省去这个包装过程呢？

当然是有的，还是要用到全局处理。但是为了扩展性，就是允许绕过数据统一响应（这样就可以提供多方使用），我们可以自定义注解，利用注解来选择是否进行全局响应包装

首先创建自定义注解，作用相当于全局处理类开关：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD}) // 表明该注解只能放在方法上
public @interface NotResponseBody {
}
```

其次创建一个类并加上注解使其成为全局处理类。然后继承 `ResponseBodyAdvice` 接口重写其中的方法，即可对我们的controller进行增强操作，具体看代码和注释：

```java
package com.csdn.demo1.global;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.core.MethodParameter;
import org.springframework.http.MediaType;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;

@RestControllerAdvice(basePackages = {"com.scdn.demo1.controller"}) // 注意哦，这里要加上需要扫描的包
public class ResponseControllerAdvice implements ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter returnType, Class<? extends HttpMessageConverter<?>> aCl
        // 如果接口返回的类型本身就是ResultVO那就没有必要进行额外的操作，返回false
        // 如果方法上加了我们的自定义注解也没有必要进行额外的操作
        return !(returnType.getParameterType().equals(ResultVO.class) || returnType.hasMethodAnnotat
    }

    @Override
    public Object beforeBodyWrite(Object data, MethodParameter returnType, MediaType mediaType, Clas
        // String类型不能直接包装，所以要进行些特别的处理
        if (returnType.getGenericParameterType().equals(String.class)) {
            ObjectMapper objectMapper = new ObjectMapper();
            try {
                // 将数据包装在ResultVO里后，再转换为json字符串响应给前端
                return objectMapper.writeValueAsString(new ResultVO<>(data));
            } catch (JsonProcessingException e) {
                throw new APIException("返回String类型错误");
            }
        }
        // 将原本的数据包装在ResultVO里
        return new ResultVO<>(data);
    }
}
```

重写的这两个方法是用来在controller将数据进行返回前进行增强操作，supports方法要返回为true才会执行 `beforeBodyWrite` 方法，所以如果有些情况不需要进行增强操作可以在supports方法里进行判断。

对返回数据进行真正的操作还是在 `beforeBodyWrite` 方法中，我们可以直接在该方法里包装数据，这样就不需要每个接口都进行数据包装了，省去了很多麻烦。此时controller只需这样写就行了：

```java
@GetMapping("/getUser")
//@NotResponseBody  //是否绕过数据统一响应开关
public User getUser() {
    User user = new User();
    user.setId(1L);
    user.setAccount("12345678");
    user.setPassword("12345678");
    user.setEmail("123@qq.com");
    // 注意哦，这里是直接返回的User类型，并没有用ResultVO进行包装
    return user;
}
```

七、接口版本控制

1、简介

在spring boot项目中，如果要进行restful接口的版本控制一般有以下几个方向：

- 基于path的版本控制
- 基于header的版本控制

在spring MVC下，url映射到哪个method是由 `RequestMappingHandlerMapping` 来控制的，那么我们也是通过 `RequestMappingHandlerMapping` 来做版本控制的。

2、Path控制实现

首先定义一个注解

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ApiVersion {
    // 默认接口版本号1.0开始，这里我只做了两级，多级可在正则进行控制
    String value() default "1.0";
}
```

`ApiVersionCondition`用来控制当前request 指向哪个method

```
public class ApiVersionCondition implements RequestCondition<ApiVersionCondition> {
    private static final Pattern VERSION_PREFIX_PATTERN = Pattern.compile("v(\\d+\\.\\d+)");

    private final String version;

    public ApiVersionCondition(String version) {
        this.version = version;
    }

    @Override
    public ApiVersionCondition combine(ApiVersionCondition other) {
        // 采用最后定义优先原则，则方法上的定义覆盖类上面的定义
        return new ApiVersionCondition(other.getApiVersion());
    }

    @Override
    public ApiVersionCondition getMatchingCondition(HttpServletRequest httpServletRequest) {
        Matcher m = VERSION_PREFIX_PATTERN.matcher(httpServletRequest.getRequestURI());
        if (m.find()) {
            String pathVersion = m.group(1);
            // 这个方法是精确匹配
            if (Objects.equals(pathVersion, version)) {
                return this;
            }
            // 该方法是只要大于等于最低接口version即匹配成功，需要和compareTo()配合
            // 举例：定义有1.0/1.1接口，访问1.2，则实际访问的是1.1，如果从小开始那么排序反转即可
//            if(Float.parseFloat(pathVersion)>=Float.parseFloat(version)){
//                return this;
//            }

        }
        return null;
    }

    @Override
    public int compareTo(ApiVersionCondition other, HttpServletRequest request) {
        return 0;
        // 优先匹配最新的版本号，和getMatchingCondition注释掉的代码同步使用
//        return other.getApiVersion().compareTo(this.version);
    }

    public String getApiVersion() {
        return version;
    }

}
```

`PathVersionHandlerMapping`用于注入spring用来管理

```
public class PathVersionHandlerMapping extends RequestMappingHandlerMapping {

    @Override
    protected boolean isHandler(Class<?> beanType) {
        return AnnotatedElementUtils.hasAnnotation(beanType, Controller.class);
    }

    @Override
    protected RequestCondition<?> getCustomTypeCondition(Class<?> handlerType) {
        ApiVersion apiVersion = AnnotationUtils.findAnnotation(handlerType,ApiVersion.class);
        return createCondition(apiVersion);
    }

    @Override
    protected RequestCondition<?> getCustomMethodCondition(Method method) {
        ApiVersion apiVersion = AnnotationUtils.findAnnotation(method,ApiVersion.class);
        return createCondition(apiVersion);
    }

    private RequestCondition<ApiVersionCondition>createCondition(ApiVersion apiVersion) {
        return apiVersion == null ? null : new ApiVersionCondition(apiVersion.value());
    }
}
```

`WebMvcConfiguration`配置类让spring来接管

```
@Configuration
public class WebMvcConfiguration implements WebMvcRegistrations {

    @Override
    public RequestMappingHandlerMapping getRequestMappingHandlerMapping() {
        return new PathVersionHandlerMapping();
    }
}
```

最后controller进行测试，默认是v1.0，如果方法上有注解，以方法上的为准（该方法vx.x在路径任意位置出现都可解析）

```
@RestController
@ApiVersion
@RequestMapping(value = "/{version}/test")
public class TestController {

    @GetMapping(value = "one")
    public String query(){
        return "test api default";
    }

    @GetMapping(value = "one")
    @ApiVersion("1.1")
    public String query2(){
        return "test api v1.1";
    }


    @GetMapping(value = "one")
    @ApiVersion("3.1")
    public String query3(){
        return "test api v3.1";
    }
}
```

3、header控制实现

总体原理与Path类似，修改 `ApiVersionCondition` 即可，之后访问时在header带上 `X-VERSION` 参数即可

```
public class ApiVersionCondition implements RequestCondition<ApiVersionCondition> {
    private static final String X_VERSION = "X-VERSION";
    private final String version ;

    public ApiVersionCondition(String version) {
        this.version = version;
    }

    @Override
    public ApiVersionCondition combine(ApiVersionCondition other) {
        // 采用最后定义优先原则，则方法上的定义覆盖类上面的定义
        return new ApiVersionCondition(other.getApiVersion());
    }

    @Override
    public ApiVersionCondition getMatchingCondition(HttpServletRequest httpServletRequest) {
        String headerVersion = httpServletRequest.getHeader(X_VERSION);
        if(Objects.equals(version,headerVersion)){
            return this;
        }
        return null;
    }

    @Override
    public int compareTo(ApiVersionCondition apiVersionCondition, HttpServletRequest httpServletRequ
        return 0;
    }
    public String getApiVersion() {
        return version;
    }

}
```

八、API接口安全

1、简介

APP、前后端分离项目都采用API接口形式与服务器进行数据通信，传输的数据被偷窥、被抓包、被伪造时有发生，那么如何设计一套比较安全的API接口方案至关重要，一般的解决方案有以下几点：

- Token授权认证，防止未授权用户获取数据；
- 时间戳超时机制；
- URL签名，防止请求参数被篡改；
- 防重放，防止接口被第二次请求，防采集；
- 采用HTTPS通信协议，防止数据明文传输；

2、Token授权认证

> 因为HTTP协议是无状态的，Token的设计方案是用户在客户端使用用户名和密码登录后，服务器会给客户端返回一个Token，并将Token以键值对的形式存放在缓存（一般是Redis）中，后续客户端对需要授权模块的所有操作都要带上这个Token，服务器端接收到请求后进行Token验证，如果Token存在，说明是授权的请求。

Token生成的设计要求

- 应用内一定要唯一，否则会出现授权混乱，A用户看到了B用户的数据；
- 每次生成的Token一定要不一样，防止被记录，授权永久有效；
- 一般Token对应的是Redis的key，value存放的是这个用户相关缓存信息，比如：用户的id；
- 要设置Token的过期时间，过期后需要客户端重新登录，获取新的Token，如果Token有效期设置较短，会反复需要用户登录，体验比较差，我们一般采用Token过期后，客户端静默登录的方式，当客户端收到Token过期后，客户端用本地保存的用户名和密码在后台静默登录来获取新的Token，还有一种是单独出一个刷新Token的接口，但是一定要注意刷新机制和安全问题；

根据上面的设计方案要求，我们很容易得到Token=md5(用户ID+登录的时间戳+服务器端秘钥)这种方式来获得Token，因为用户ID是应用内唯一的，登录的时间戳保证每次登录的时候都不一样，服务器端秘钥是配置在服务器端参与加密的字符串（即：盐），目的是提高Token加密的破解难度，注意一定不要泄漏

3、时间戳超时机制

客户端每次请求接口都带上当前时间的时间戳timestamp，服务端接收到timestamp后跟当前时间进行比对，如果时间差大于一定时间（比如：1分钟），则认为该请求失效。**时间戳超时机制是防御DOS攻击的有效手段。** 例如 `http://url/getInfo?id=1&timetamp=1661061696`

4、URL签名

写过支付宝或微信支付对接的同学肯定对URL签名不陌生，我们只需要将原本发送给server端的明文参数做一下签名，然后在server端用相同的算法再做一次签名，对比两次签名就可以确保对应明文的参数有没有被中间人篡改过。例如 `http://url/getInfo?id=1&timetamp=1559396263&sign=e10adc3949ba59abbe56e057f20f883e`

签名算法过程

- 首先对通信的参数按key进行字母排序放入数组中（一般请求的接口地址也要参与排序和签名，那么需要额外添加 `url=http://url/getInfo` 这个参数）
- 对排序完的数组键值对用&进行连接，形成用于加密的参数字符串
- 在加密的参数字符串前面或者后面加上私钥，然后用md5进行加密，得到sign，然后随着请求接口一起传给服务器。服务器端接收到请求后，用同样的算法获得服务器的sign，对比客户端的sign是否一致，如果一致请求有效

5、防重放

客户端第一次访问时，将签名sign存放到服务器的Redis中，超时时间设定为跟时间戳的超时时间一致，二者时间一致可以保证无论在timestamp限定时间内还是外 URL都只能访问一次，如果被非法者截获，使用同一个URL再次访问，如果发现缓存服务器中已经存在了本次签名，则拒绝服务。

如果在缓存中的签名失效的情况下，有人使用同一个URL再次访问，则会被时间戳超时机制拦截，这就是为什么要求sign的超时时间要设定为跟时间戳的超时时间一致。拒绝重复调用机制确保URL被别人截获了也无法使用（如抓取数据）

方案流程

- 客户端通过用户名密码登录服务器并获取Token；
- 客户端生成时间戳timestamp，并将timestamp作为其中一个参数；
- 客户端将所有的参数，包括Token和timestamp按照自己的签名算法进行排序加密得到签名sign
- 将token、timestamp和sign作为请求时必须携带的参数加在每个请求的URL后边，例：`http://url/request?token=h40adc3949bafjhbbe56e027f20f583a&timetamp=1559396263&sign=e10adc3949ba59abbe56e057f20f883e`
- 服务端对token、timestamp和sign进行验证，只有在token有效、timestamp未超时、缓存服务器中不存在sign三种情况同时满足，本次请求才有效；

6、采用HTTPS通信协议

安全套接字层超文本传输协议HTTPS，为了数据传输的安全，HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为客户端和服务器之间的通信加密。

HTTPS也不是绝对安全的，比如中间人劫持攻击，中间人可以获取到客户端与服务器之间所有的通信内容

九、总结

自此整个后端接口基本体系就构建完毕了

- 通过Validator + 自动抛出异常来完成了方便的参数校验
- 通过全局异常处理 + 自定义异常完成了异常操作的规范
- 通过数据统一响应完成了响应数据的规范

- 多个方面组装非常优雅的完成了后端接口的协调，让开发人员有更多的经历注重业务逻辑代码，轻松构建后端接口

这里再说几点

- controller做好try-catch工作，及时捕获异常，可以再次抛出到全局，统一格式返回前端
- 做好日志系统，关键位置一定要有日志
- 做好全局统一返回类，整个项目规范好定义好
- controller入参字段可以抽象出一个公共基类，在此基础上进行继承扩充
- controller层做好入参参数校验
- 接口安全验证

来源：blog.csdn.net/lemon_TT/article/details/108309900

End

Read more

People who liked this content also liked

1.8w 字的 SQL 优化大全
数据分析与开发

公司新入职一位大佬，把SpringBoot项目启动时间从7分钟降到了40秒！
macrozheng

又一个 SQL 神器，开源了！
GitHubDaily