作者：华为云开发者社区  
链接：https://zhuanlan.zhihu.com/p/245405288  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

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）。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，但一个优秀的后端接口和一个糟糕的后端接口对比起来差异还是蛮大的，其中最重要的关键点就是看是否规范!

本文就一步一步演示如何构建起一个优秀的后端接口体系，体系构建好了自然就有了规范，同时再构建新的后端接口也会十分轻松。

**所需依赖包**

这里用的是SpringBoot配置项目，本文讲解的重点是后端接口，所以只需要导入一个spring-boot-starter-web包就可以了：

<!--web依赖包，web应用必备-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

本文还用了swagger来生成API文档，lombok来简化类，不过这两者不是必须的，可用可不用。

**参数校验**

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。

**业务层校验**

首先我们来看一下最常见的做法，就是在业务层进行参数校验：

public String addUser(User user) {

if (user == null || user.getId() == null || user.getAccount() == null || user.getPassword() == null || user.getEmail() == null) {

return "对象或者对象字段不能为空";

}

if (StringUtils.isEmpty(user.getAccount()) || StringUtils.isEmpty(user.getPassword()) || StringUtils.isEmpty(user.getEmail())) {

return "不能输入空字符串";

}

if (user.getAccount().length() < 6 || user.getAccount().length() > 11) {

return "账号长度必须是6-11个字符";

}

if (user.getPassword().length() < 6 || user.getPassword().length() > 16) {

return "密码长度必须是6-16个字符";

}

if (!Pattern.matches("^[a-zA-Z0-9\_-]+@[a-zA-Z0-9\_-]+(\\.[a-zA-Z0-9\_-]+)+$", user.getEmail())) {

return "邮箱格式不正确";

}

// 参数校验完毕后这里就写上业务逻辑

return "success";

}

这样做当然是没有什么错的，而且格式排版整齐也一目了然，不过这样太繁琐了，这还没有进行业务操作呢光是一个参数校验就已经这么多行代码，实在不够优雅。

我们来改进一下，使用Spring Validator和Hibernate Validator这两套Validator来进行方便的参数校验！这两套Validator依赖包已经包含在前面所说的web依赖包里了，所以可以直接使用。

**Validator + BindResult进行校验**

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需要校验的参数上加上@Valid注解，并添加BindResult参数即可方便完成验证：

@RestController

@RequestMapping("user")

public class UserController {

@Autowired

private UserService userService;

@PostMapping("/addUser")

public String addUser(@RequestBody @Valid 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

for (ObjectError error : bindingResult.getAllErrors()) {

return error.getDefaultMessage();

}

return userService.addUser(user);

}

}

这样当请求数据传递到接口的时候Validator就自动完成校验了，校验的结果就会封装到BindingResult中去，如果有错误信息我们就直接返回给前端，业务逻辑代码也根本没有执行下去。

此时，业务层里的校验代码就已经不需要了：

public String addUser(User user) {

// 直接编写业务逻辑

return "success";

}

现在可以看一下参数校验效果。我们故意给这个接口传递一个不符合校验规则的参数，先传递一个错误数据给接口，故意将password这个字段不满足校验条件：

{

"account": "12345678",

"email": "123@qq.com",

"id": 0,

"password": "123"

}

再来看一下接口的响应数据：

这样是不是方便很多？不难看出使用Validator校验有如下几个好处：

（1）简化代码，之前业务层那么一大段校验代码都被省略掉了。

（2）使用方便，那么多校验规则可以轻而易举的实现，比如邮箱格式验证，之前自己手写正则表达式要写那么一长串，还容易出错，用Validator直接一个注解搞定。（还有更多校验规则注解，可以自行去了解哦）

（3）减少耦合度，使用Validator能够让业务层只关注业务逻辑，从基本的参数校验逻辑中脱离出来。

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。

这样有点麻烦，并且重复代码很多（尽管可以将这个重复代码封装成方法）。我们能否去掉BindingResult这一步呢？当然是可以的！

**Validator + 自动抛出异常**

我们完全可以将BindingResult这一步给去掉：

@PostMapping("/addUser")

public String addUser(@RequestBody @Valid User user) {

return userService.addUser(user);

}

去掉之后会发生什么事情呢？直接来试验一下，还是按照之前一样故意传递一个不符合校验规则的参数给接口。此时我们观察控制台可以发现接口已经引发MethodArgumentNotValidException异常了：

其实这样就已经达到我们想要的效果了，参数校验不通过自然就不执行接下来的业务逻辑，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。也就是说，我们完全没必要添加相关BindingResult相关操作嘛。

不过事情还没有完，异常是引发了，可我们并没有编写返回错误信息的代码呀，那参数校验失败了会响应什么数据给前端呢？

我们来看一下刚才异常发生后接口响应的数据：

没错，是直接将整个错误对象相关信息都响应给前端了！这样就很难受，不过解决这个问题也很简单，就是我们接下来要讲的全局异常处理！

**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，不然还不如用之前BindingResult方式呢。**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！

**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：

@RestControllerAdvice

public class ExceptionControllerAdvice {

@ExceptionHandler(MethodArgumentNotValidException.class)

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();

}

}

我们再来看下这次校验失败后的响应数据：

没错，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！更多的校验思路：SpringBoot实现通用的接口参数校验

**自定义异常**

全局处理当然不会只能处理一种异常，用途也不仅仅是对一个参数校验方式进行优化。在实际开发中，如何对异常处理其实是一个很麻烦的事情。传统处理异常一般有以下烦恼：

* 是捕获异常(try…catch)还是抛出异常(throws)
* 是在controller层做处理还是在service层处理又或是在dao层做处理
* 处理异常的方式是啥也不做，还是返回特定数据，如果返回又返回什么数据
* 不是所有异常我们都能预先进行捕捉，如果发生了没有捕捉到的异常该怎么办？

以上这些问题都可以用全局异常处理来解决，全局异常处理也叫统一异常处理，全局和统一处理代表什么？代表规范！规范有了，很多问题就会迎刃而解！

全局异常处理的基本使用方式大家都已经知道了，我们接下来更进一步的规范项目中的异常处理方式：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，我这时候就可以手动抛出异常从而触发事务回滚。那手动抛出异常最简单的方式就是throw new RuntimeException("异常信息")了，不过使用自定义会更好一些：

* 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：

@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;

}

}

在刚才的全局异常处理类中记得添加对我们自定义异常的处理：

@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。

现在全局异常处理和自定义异常已经弄好了，不知道大家有没有发现一个问题，就是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。这就要引申出我们接下来要讲的东西了：数据统一响应

**数据统一响应**

现在我们规范好了参数校验方式和异常处理方式，然而还没有规范响应数据！比如我要获取一个分页信息数据，获取成功了呢自然就返回的数据列表，获取失败了后台就会响应异常信息，即一个字符串，就是说前端开发者压根就不知道后端响应过来的数据会是啥样的！所以，统一响应数据是前后端规范中必须要做的！

**自定义统一响应体**

统一数据响应第一步肯定要做的就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！那么如何定义响应体呢？关于异常的设计：如何更优雅的设计异常

可以参考我们自定义异常类，也来一个响应信息代码code和响应信息说明msg：

@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(1000, "success", data);

}

public ResultVO(int code, String msg, T data) {

this.code = code;

this.msg = msg;

this.data = data;

}

}

然后我们修改一下全局异常处理那的返回值：

@ExceptionHandler(APIException.class)

public ResultVO<String> APIExceptionHandler(APIException e) {

// 注意哦，这里返回类型是自定义响应体

return new ResultVO<>(e.getCode(), "响应失败", e.getMsg());

}

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResultVO<String> MethodArgumentNotValidExceptionHandler(MethodArgumentNotValidException e) {

ObjectError objectError = e.getBindingResult().getAllErrors().get(0);

// 注意哦，这里返回类型是自定义响应体

return new ResultVO<>(1001, "参数校验失败", objectError.getDefaultMessage());

}

我们再来看一下此时如果发生异常了会响应什么数据给前端：

OK，这个异常信息响应就非常好了，状态码和响应说明还有错误提示数据都返给了前端，并且是所有异常都会返回相同的格式！异常这里搞定了，别忘了我们到接口那也要修改返回类型，我们新增一个接口好来看看效果：

@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);

}

看一下如果响应正确返回的是什么效果：

这样无论是正确响应还是发生异常，响应数据的格式都是统一的，十分规范！

数据格式是规范了，不过响应码code和响应信息msg还没有规范呀！大家发现没有，无论是正确响应，还是异常响应，响应码和响应信息是想怎么设置就怎么设置，要是10个开发人员对同一个类型的响应写10个不同的响应码，那这个统一响应体的格式规范就毫无意义！所以，必须要将响应码和响应信息给规范起来。

**响应码枚举**

要规范响应体中的响应码和响应信息用枚举简直再恰当不过了，我们现在就来创建一个响应码枚举类：

@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;

}

}

然后修改响应体的构造方法，让其只准接受响应码枚举来设置响应码和响应信息：

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;

}

然后同时修改全局异常处理的响应码设置方式：

@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());

}

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！这些可以参考：Java项目构建基础：统一结果，统一异常，统一日志

**全局处理响应数据**

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？当然是有滴，还是要用到全局处理。

首先，先创建一个类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：

@RestControllerAdvice(basePackages = {"com.rudecrab.demo.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<?>> aClass) {

// 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

return !returnType.getGenericParameterType().equals(ResultVO.class);

}

@Override

public Object beforeBodyWrite(Object data, MethodParameter returnType, MediaType mediaType, Class<? extends HttpMessageConverter<?>> aClass, ServerHttpRequest request, ServerHttpResponse response) {

// 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。

我们可以现在去掉接口的数据包装来看下效果：

@GetMapping("/getUser")

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;

}

然后我们来看下响应数据：

成功对数据进行了包装！

**注意：beforeBodyWrite方法里包装数据无法对String类型的数据直接进行强转，所以要进行特殊处理，这里不讲过多的细节，有兴趣可以自行深入了解。**

**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