数据格式检验

SpringBoot2+ **集成** Spring Boot 2 中的参数校验 spring-boot-starter-validation/Hibernate Validator

引依赖

注意: 只支持 spring-boot-starter-web , 不支持 spring-boot-starter-webflux

统一返回值

```
import lombok.Data;
import org.springframework.http.HttpStatus;
import java.util.HashMap;
import java.util.Map;
/**
* 统一结果返回类。方法采用链式调用的写法(即返回类本身 return this)。
* 构造器私有,不允许进行实例化,但提供静态方法 ok、error 返回一个实例。
* 静态方法说明:
      ok 返回一个 成功操作 的结果(实例对象)。
      error 返回一个 失败操作 的结果(实例对象)。
* 普通方法说明:
*
     success 用于自定义响应是否成功
      code用于自定义响应状态码message用于自定义响应消息data用于自定义响应数据
*
* 依赖信息说明:
      此处使用 @Data 注解, 需导入 lombok 相关依赖文件。
      使用 HttpStatus 的常量表示 响应状态码,需导入 httpcore 相关依赖文件。
*/
@Data
public class Result {
   /**
    * 响应是否成功, true 为成功, false 为失败
   */
   private Boolean success;
```

```
/**
    *响应状态码, 200 成功, 500 系统异常
   private Integer code;
   /**
    * 响应消息
   private String message;
   /**
    * 响应数据
   private Map<String, Object> data = new HashMap<>();
   /**
    * 默认私有构造器
   private Result(){}
   /**
    * 私有自定义构造器
    * @param success 响应是否成功
    * @param code 响应状态码
    * @param message 响应消息
    */
   private Result(Boolean success, Integer code, String message){
       this.success = success;
      this.code = code;
      this.message = message;
   }
   /**
    * 返回一个默认的 成功操作 的结果, 默认响应状态码 200
    * @return 成功操作的实例对象
   public static Result ok() {
       return new Result(true, HttpStatus.OK.value(), "success");
   }
   /**
   * 返回一个自定义 成功操作 的结果
    * @param success 响应是否成功
    * @param code 响应状态码
    * @param message 响应消息
    * @return 成功操作的实例对象
    */
   public static Result ok(Boolean success, Integer code, String message) {
      return new Result(success, code, message);
   }
   /**
    * 返回一个默认的 失败操作 的结果, 默认响应状态码为 500
   * @return 失败操作的实例对象
    */
   public static Result error() {
       return new Result(false, HttpStatus.INTERNAL_SERVER_ERROR.value(),
"error");
```

```
/**
* 返回一个自定义 失败操作 的结果
* @param success 响应是否成功
* @param code 响应状态码
* @param message 相应消息
* @return 失败操作的实例对象
*/
public static Result error(Boolean success, Integer code, String message) {
  return new Result(success, code, message);
}
/**
* 自定义响应是否成功
* @param success 响应是否成功
* @return 当前实例对象
public Result success(Boolean success) {
   this.setSuccess(success);
   return this;
}
/**
* 自定义响应状态码
* @param code 响应状态码
* @return 当前实例对象
public Result code(Integer code) {
   this.setCode(code);
   return this;
}
/**
* 自定义响应消息
* @param message 响应消息
* @return 当前实例对象
*/
public Result message(String message) {
   this.setMessage(message);
   return this;
}
/**
* 自定义响应数据,一次设置一个 map 集合
* @param map 响应数据
* @return 当前实例对象
*/
public Result data(Map<String, Object> map) {
   this.data.putAll(map);
   return this;
}
* 通用设置响应数据,一次设置一个 key - value 键值对
* @param key 键
 * @param value 数据
* @return 当前实例对象
```

```
*/
public Result data(String key, Object value) {
    this.data.put(key, value);
    return this;
}
```

统一异常处理

```
import com.ivan.common.Result;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationException;
import javax.validation.ValidationException;
import java.util.*;
import java.util.stream.Collectors;
@RestControllerAdvice
public class GlobalExceptionHandler {
    * 处理所有校验失败的异常(MethodArgumentNotValidException异常)
    * @param ex
    * @return
   @ExceptionHandler(value = MethodArgumentNotValidException.class)
   // 设置响应状态码为400
   @ResponseStatus(HttpStatus.BAD_REQUEST)
   public Result handleBindGetException(MethodArgumentNotValidException ex) {
       Map<String, Object> body = new LinkedHashMap<String, Object>();
       body.put("timestamp", new Date());
       // 获取所有异常
       List<String> errors = ex.getBindingResult()
                .getFieldErrors()
                .stream()
                .map(x -> x.getDefaultMessage())
                .collect(Collectors.toList());
       body.put("errors", errors);
       return Result.error().data(body).message("数据格式有误");
   }
   /**
    * 处理所有参数校验时抛出的异常
     * @param ex
    * @return
```

```
*/
   @ExceptionHandler(value = ValidationException.class)
   @ResponseStatus(HttpStatus.BAD_REQUEST)
   public Result handleBindException(ValidationException ex) {
       Map<String, Object> body = new LinkedHashMap<String, Object>();
       body.put("timestamp", new Date());
       // 获取所有异常
       List<String> errors = new LinkedList<String>();
       if(ex instanceof ConstraintViolationException){
            ConstraintViolationException exs = (ConstraintViolationException)
ex;
            Set<ConstraintViolation<?>> violations =
exs.getConstraintViolations();
           for (ConstraintViolation<?> item : violations) {
               errors.add(item.getMessage());
       }
       body.put("errors", errors);
       return Result.error().message("数据格式有误").data(body);
   }
}
```

实体类上添加注解

```
@Data
public class User implements Serializable {
    private String id;
    @NotNull(message = "姓名不能为空")
    @Size(min = 1, max = 20, message = "姓名长度必须在1-20之间")
    private String name;

@Min(value = 10, message = "年龄必须大于10")
    @Max(value = 150, message = "年龄必须小于150")
    private Integer age;

@Email(message = "邮箱格式不正确")
    private String email;
}
```

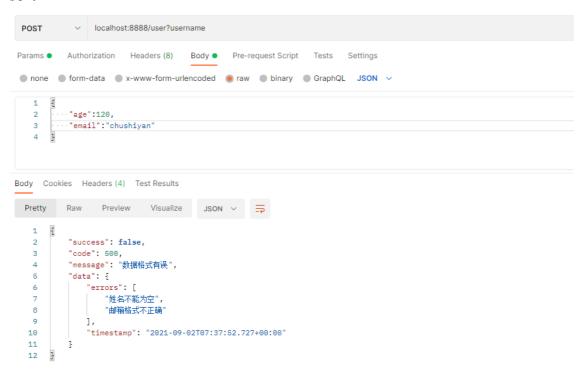
编写Controller进行测试

1. 先测试 post 请求 (实体类上的注解)

```
package com.ivan.validator.controller;
import com.ivan.common.Result;
import com.ivan.validator.entity.User;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import javax.validation.valid;
import javax.validation.constraints.NotNull;
/**
 * @ClassName UserController
 * @Description TODO
 * @author cuiyingfan
 * @date 2021/9/2 14:55
 * @version 1.0
 */
@RestController
public class UserController {
    @PostMapping("user")
    public Result test(@Valid @RequestBody User user){
        System.out.println(user);
       return Result.ok();
   }
}
```

测试:



2. 再测试 get 请求 (方法体内的注解)

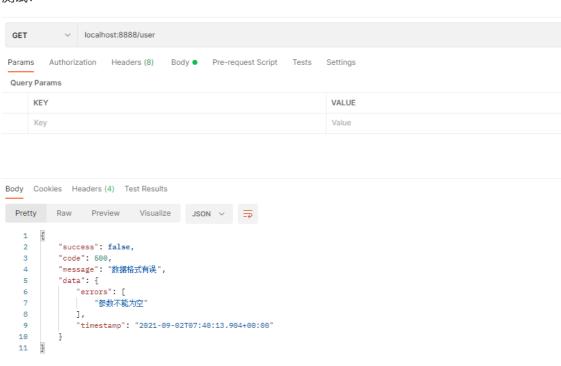
```
@RestController
@Validated
public class UserController {

    @PostMapping("user")
    public Result test(@Valid @RequestBody User user){
        System.out.println(user);
        return Result.ok();
    }
}
```

```
}
@GetMapping("user")
public String f(@NotNull(message = "参数不能为空")String str) {
    return str;
}
}
```

注意:在形参前添加@NotNull等校验器的注解,校验单个形参数据时,需要在Controller上面添加@Validated注解

测试:



分组校验

1. 写一个接口用于分组

```
public interface GroupA {
}
```

2. 修改 entity 上的注解 (添加 groups 关键字修饰)

```
@Data
public class User implements Serializable {
    @NotNull(groups = GroupA.class, message = "id不能为空")
    private String id;

@NotNull(message = "姓名不能为空")
    @Size(min = 1, max = 20, message = "姓名长度必须在1-20之间")
    private String name;

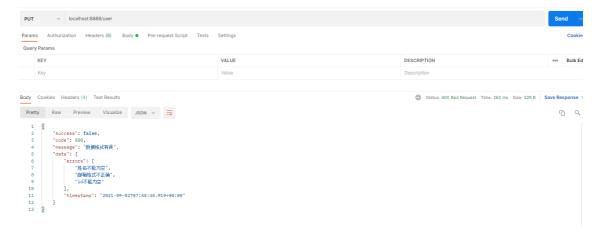
@Min(value = 10, message = "年龄必须大于10")
    @Max(value = 150, message = "年龄必须小于150")
    private Integer age;
```

```
@Email(message = "邮箱格式不正确")
private String email;
}
```

3. 在 controller 中使用 @validated 指定使用哪个组

```
@PutMapping("user")
    // 指定GroupA, 这样就会校验id属性是否为空
    // 注意: 还得必须添加Default.class, 否则不会执行其他的校验(如我们案例中的@Email)
    public Result updateUser(@Validated({GroupA.class, Default.class})
@RequestBody User user) {
        return Result.ok().message("更新用户成功");
    }
```

4. 测试



标签

注解	功能
@AssertFalse	可以为null,如果不为null的话必须为false
@AssertTrue	可以为null,如果不为null的话必须为true
@DecimalMax	设置不能超过最大值
@DecimalMin	设置不能超过最小值
@Digits	设置必须是数字且数字整数的位数和小数的位数必须在指定范围内
@Future	日期必须在当前日期的未来
@Past	日期必须在当前日期的过去
@Max	最大不得超过此最大值
@Min	最大不得小于此最小值
@NotNull	不能为null,可以是空
@Null	必须为null
@Pattern	必须满足指定的正则表达式
@Size	集合、数组、map等的size()值必须在指定范围内
@Email	必须是email格式
@Length	长度必须在指定范围内
@NotBlank	字符串不能为null,字符串trim()后也不能等于""
@NotEmpty	不能为null,集合、数组、map等size()不能为0;字符串trim()后可以等于""
@Range	值必须在指定范围内
@URL	必须是一个URL