1.参数校验

spring validation

spring-boot-starter-validation

1.Controller方法参数校验

Spring 提供了相应的 Bean Validation 实现：Java Bean Validation[2]，并在 Spring MVC 中添加了自动校验，默认就会对 @Valid/@Validated 修饰的方法参数使用 Validator 来做校验逻辑。

举个例子： 第一步，在方法在入参对应元素上配置校验注解。

```
@Data

public class UserRequest {

    @NotBlank(message = "用户ID不能为空")
    private String userId;

    @NotBlank(message = "电话号码不能为空")
    @Pattern(regexp = "^(13[0-
```

```
9]|14[01456879]|15[0-35-9]|16[2567]|17[0-
8]|18[0-9]|19[0-35-9])\\d{8}$", message =
"电话号码格式错误")
        private String mobilePhone;

    @Min(message = "年龄必须大于0", value =
0)
    @Max(message = "年龄不能超过150", value
= 150)
    private Integer age;

    @NotNull(message = "用户详情不能为空")
    @Valid
    private UserDetail userDetail;

    //省略其他参数

}
```

第二步，在 Controller 相应方法中，使用 @Valid/@Validated 注解开启数据校验功能。

```
@RestController

public class TestController {

    @RequestMapping(value =
```

```
"/api/saveUser", method =
RequestMethod.POST)
    public ResponseEntity<BaseResult>
saveUser(@Validated @RequestBody
UserRequest user) {

        // 省略其他业务代码
        return new ResponseEntity<>
(HttpStatus.OK);
    }
}
```

如果数据校验通过，就会继续执行方法里的业务逻辑；否则，就会抛出一个 MethodArgumentNotValidException 异常。默认情况下，Spring 会将该异常及其信息以错误码 400 进行下发，返回结果示例如下：

```
{

  "timestamp": 1666777674977,

  "status": 400,

  "error": "Bad Request",
```

```
  "exception":
"org.springframework.web.bind.MethodArgume
ntNotValidException",

  "errors": [

    {

      "codes": [


"NotBlank.UserRequest.mobilePhone",

        "NotBlank.mobilePhone",

        "NotBlank.java.lang.String",

        "NotBlank"

      ],

      "arguments": [

        {

          "codes": [

            "UserRequest.mobilePhone",
```

```
            "mobilePhone"
          ],
          "arguments": null,
          "defaultMessage":
"mobilePhone",
          "code": "mobilePhone"
        }
      ],
      "defaultMessage": "电话号码不能为空",
      "objectName": "UserRequest",
      "field": "mobilePhone",
      "rejectedValue": null,
      "bindingFailure": false,
      "code": "NotBlank"
```

```
    }

  ],

  "message": "Validation failed for
object='UserRequest'. Error count: 1",

  "path": "/api/saveUser"

}
```

但是返回的异常结果不是需要的格式，所以再来个全局异常捕获器拦截该异常，就可以得到一个完美的异常结果：

```
@RestControllerAdvice

public class GlobalExceptionHandler {


@ExceptionHandler(MethodArgumentNotValidEx
ception.class)

@ResponseStatus(HttpStatus.BAD_REQUEST)
    public BaseResult
handlerMethodArgumentNotValidException(Me
thodArgumentNotValidException e) {
```

```
        FieldError fieldError =
e.getBindingResult().getFieldErrors().get
(0);

        return new
BaseResult(CommonResultCode.ILLEGAL_PARAM
ETERS.getErrorCode(),
                "入参中的" +
fieldError.getField() +
fieldError.getDefaultMessage(),
EagleEye.getTraceId());
    }
}
```

设置了如上捕获器后，如果数据校验不通过，返回的结果为：

```
{

  "success": false,

  "errorCode": "ILLEGAL_PARAMETERS",

  "errorMessage": "入参中的mobilePhone电话号
码不能为空",
```

```
  "traceId":
"1ef974931667466369611l017d73c9",

  "extInfo": {}

}
```

借助Spring和约束注解，就非常简单明了、优雅地完成了方法参数校验。 而且，假如以后入参对象里新增了参数，只需要顺便添加一个注解，而不用去改业务代码

@Valid 和 @Validated

- @Valid [3]注解，是 Bean Validation 所定义，可以添加在普通方法、构造方法、方法参数、方法返回、成员变量上，表示它们需要进行约束校验。
- @Validated [4]注解，是 Spring Validation 所定义，可以添加在类、方法参数、普通方法上，表示它们需要进行约束校验。

两者的区别在于 @Validated 有 value 属性，支持分组校验，即根据不同的分组采用不同的校验机

制，@Valid 可以添加在成员变量上，支持嵌套校验。所以建议的使用方式就是：启动校验（即 Controller 层）时使用 @Validated 注解，嵌套校验时使用 @Valid 注解，这样就能同时使用分组校验和嵌套校验功能。

分组校验

但是，对于同个参数，不同的场景可能需要不同的校验，这时候就可以用分组校验能力。

比如创建 User 时，userId为空；但是更新 User 时，userId值则不能为空。示例如下：

```
@Data

public class UserRequest {

    @NotBlank(message = "用户ID不能为空",
groups = {UpdateUser.class})
    private String userId;

    @NotBlank(message = "电话号码不能为空",
groups = {UpdateUser.class,
InsertUser.class})
    @Pattern(regexp = "^(13[0-
```

```
9]|14[01456879]|15[0-35-9]|16[2567]|17[0-
8]|18[0-9]|19[0-35-9])\\d{8}$", message =
"电话号码格式错误")
    private String mobilePhone;

    @Min(message = "年龄必须大于0", value =
0, groups = {UpdateUser.class,
InsertUser.class})
    @Max(message = "年龄不能超过150", value
= 150, groups = {UpdateUser.class,
InsertUser.class})
    private Integer age;

    @NotNull(message = "用户详情不能为空",
groups = {UpdateUser.class,
InsertUser.class})
    @Valid
    private UserDetail userDetail;

    //省略其他参数

}
```

```
@RestController
public class TestController {

    @RequestMapping(value =
"/api/saveUser", method =
```

```
RequestMethod.POST)
    public ResponseEntity<BaseResult>
saveUser(@Validated(value =
InsertUser.class) @RequestBody
UserRequest user) {
        // 省略其他业务代码
        return new ResponseEntity<>
(HttpStatus.OK);
    }
}
```

自定义校验注解

还有，如果现有的基础校验注解没法满足校验需求，那就可以使用自定义注解[5]。由两部分组成：

- 由 @Constraint 注解的注解。
- 实现了 javax.validation.ConstraintValidator 的 validator。

两者通过 @Constraint 关联到一起。

假设有个性别枚举，需要校验用户的性别是否属于此范围内，示例如下：

```
public enum GenderEnum implements
BasicEnum {
```

```
    male("male", "男"),
    female("female", "女");

    private String code;

    private String desc;
    // 省略其他
}
```

第一步，自定义约束注解 InEnum，可以参考 NotNull 的定义：

```
@Target({ElementType.METHOD,
ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy =
InEnumValidator.class)
public @interface InEnum {
    /**
     * 枚举类型
     *
     * @return
     */
    Class<? extends BasicEnum>
enumType();
```

```
    String message() default "枚举类型不匹
配";

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

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

第二步，自定义约束校验器 InEnumValidator。如果校验通过，返回 true；反之返回 false：

```
public class InEnumValidator implements
ConstraintValidator<InEnum, Object> {
    private Class<? extends BasicEnum>
enumType;
    @Override
    public void initialize(InEnum inEnum)
{
        enumType = inEnum.enumType();
    }
    @Override
    public boolean isValid(Object object,
ConstraintValidatorContext
constraintValidatorContext) {
        if (object == null) {
            return true;
```

```
        }

        if (enumType == null ||
!enumType.isEnum()) {
            return false;
        }

        for (BasicEnum basicEnum :
enumType.getEnumConstants()) {
            if
(basicEnum.getCode().equals(object)) {
                return true;
            }
        }
        return false;
    }
}
```

第三步，参数上增加 @InEnum 注解校验：

```
@Data
public class UserRequest {
    @InEnum(enumType = GenderEnum.class,
message = "用户性别不在枚举范围内")
    private String gender;
    //省略其他参数
}
```

设置了如上校验后，如果数据校验不通过，返回的结果为：

```
{
  "success": false,
  "errorCode": "ILLEGAL_PARAMETERS",
  "errorMessage": "入参中的gender用户性别不在枚举范围内",
  "traceId": "1ef9749316674663696111017d73c9",
  "extInfo": {}
}
```

2.Service方法参数校验

效果示例

更多情况下是需要对 Service 层的接口进行参数校验的，那么该怎么配置呢?

在校验方法入参的约束时，若是 @Override 父类/接口的方法，那么这个入参约束只能写在父类/接口上面。

（至于为什么只能写在接口处，其实是和 Bean Validation 的实现产品有关，可参考此类：

OverridingMethodMustNotAlterParameterConstraints)

如果入参是平铺的参数

首先需要在父类/接口的方法入参里增加注解约束，然后用 @Validated 修饰我们的实现类。

```
public interface SchedulerServiceClient {
    /**
     * 获取应用不同环境的所有定时任务
     * @param appName 应用名称
     * @param env 环境
     * @param status 任务状态
     * @param userId 用户工号
     * @return
     */
    List<JobConfigInfo>
queryJobList(@NotBlank(message = "应用名称
不能为空")String appName,

                 @NotBlank(message = "环境
不能为空")String env,
                     Integer status,
                  @NotBlank(message = "用
户id不能为空")String userId);
}
```

```
@Component
@Slf4j(topic = "BIZ-SERVICE")
@HSFProvider(serviceInterface =
SchedulerServiceClient.class,
clientTimeout = 3000)
@Validated
public class SchedulerServiceClientImpl
implements SchedulerServiceClient {
    @Override
    @Log(type = LogSourceEnum.SERVICE)
    public List<JobConfigInfo>
queryJobList(String appName, String env,
Integer status, String userId) {
        // 省略业务代码
    }
```

如果数据校验通过，就会继续执行方法里的业务逻辑；否则，就会抛出一个 ConstraintViolationException 异常。

如果入参是对象

在实际开发中，其实大多数情况下我们方法入参是个对象，而不是单单平铺的参数。

首先需要在方法入参类里增加 @NotNull 等注解约束，然后在父类/接口的方法入参里增加 @Valid

（便于嵌套校验），最后用 @Validated 修饰我们的实现类。

```
@Data

public class
CreateDingNotificationRequest extends
ToString {
    /**
     * 通知类型
     */
    @NotNull(message = "通知类型不能为空")
    @InEnum(enumType =
ProcessControlDingTypeEnum.class, message
= "通知类型不在枚举值范围内")
    private String dingType;
    // 省略其他
}
```

```
public interface
ProcessControlDingService {
    /**
    * 发送钉钉通知
    * @param request
    * @return
    **/
    void createDingNotification(@Valid
```

```
CreateDingNotificationRequest request);
}
```

```
@Component
@HSFProvider(serviceInterface =
ProcessControlDingService.class,
clientTimeout = 5000)
@Validated
public class
ProcessControlDingServiceImpl implements
ProcessControlDingService {
    private static final Logger LOGGER =
LoggerFactory.getLogger(LoggerNames.BIZ_S
ERVICE);

    @Autowired
    private ProcessControlTaskService
processControlTaskService;

    @Override
    @Log(type = LogSourceEnum.SERVICE)
    public void
createDingNotification(CreateDingNotifica
tionRequest request) {
        // 省略业务代码
    }
}
```

如果需要格式化错误结果，可以再来个异常处理切面，就可以得到一个完美的异常结果。

较简洁的方式-FastValidatorUtils

```
// 返回 bean 中所有约束违反约束校验结果
Set<ConstraintViolation<T>> violationSet
= FastValidatorUtils.validate(bean);
```

具体示例如下： 自定义注解@RequestValid和对应切面RequestValidAspect。注解在具体的方法上，对于被注解的方法，在 AOP 中会扫描所有入参，对参数进行校验。

```
@Aspect
@Component
@Slf4j(topic = "BIZ-SERVICE")
public class RequestValidAspect {

    @Around("@annotation(requestValid)")
    public Object
around(ProceedingJoinPoint joinPoint,
RequestValid requestValid) throws
Throwable {
        // 获取方法入参、入参类型、出参类型
        Object[] args =
```

```
joinPoint.getArgs();
        MethodSignature signature =
(MethodSignature)
joinPoint.getSignature();
        Class<?>[] parameterTypes =
signature.getParameterTypes();
        Class<?> returnType =
signature.getMethod().getReturnType();
        // 调用前校验每个入参
        for (Object arg : args) {
            if (arg == null) {
                continue;
            }
            try {
                if (arg instanceof List
&& ((List<?>) arg).size() > 0) {
                    for (int j = 0; j <
((List<?>) arg).size(); j++) {
                        validate(((List<?
>) arg).get(j));
                    }
                } else {
                    validate(arg);
                }
            } catch
(AlscBoltBizValidateException e) {
                // 将异常处理为需要的格式返回
            }
```

```
            }
            // 方法运行后校验是否有入参约束
            Object result;
            try {
                result = joinPoint.proceed();
            } catch
(ConstraintViolationException e) {
                // 将异常处理为需要的格式返回
            }
            return result;
        }
        public static <T> void validate(T t)
{
            try {
                Set<ConstraintViolation<T>>
res = FastValidatorUtils.validate(t);
                if (!res.isEmpty()) {

                    ConstraintViolation<T>
constraintViolation =
res.iterator().next();


FastValidatorHelper.throwFastValidateExce
ption(constraintViolation);

                }
```

```
            LoggerUtil.info(log,
"validator,校验成功");

        } catch (FastValidatorException
e) {

            LoggerUtil.error(log,
"validator,校验报错,request=[{}],result=
[{}]", JSON.toJSONString(t),
e.getMessage());
            throw new
AlscBoltBizValidateException(CommonResultC
ode.ILLEGAL_PARAMETERS, e.getMessage());
        }
    }
}
```

最后在父类/接口的方法上加上自定义的注解@RequestValid即可

```
@Override
@RequestValid
public boolean
saveCheckResult(List<CheckResultInfoModel
> models) { //xxxx }
```

总结

1.Controller

1. *实体类里面给成员变量加注解*，有@NotBlank,@Pattern,@Min,@Max,@Valid，属性有message，min和max还有value，valid注解是用在对象上的递归校验对象的成员
2. Controller传入实体类作为*参数*的时候加上*@Validated注解*
3. 写一个*全局异常处理器*

@Valid 和 @Validated

- @Valid [3]注解，是 Bean Validation 所定义，可以添加在*普通方法、构造方法、方法参数、方法返回、成员变量*上，表示它们需要进行约束校验。
- @Validated [4]注解，是 Spring Validation 所定义，可以添加在*类、方法参数、普通方法上*，表示它们需要进行约束校验。 两者的区别在于 @Validated 有 *value 属性，支持分组校验*，即根据不同的分组采用不同的校验机制，

@Valid 可以添加在成员变量上，支持*嵌套校验*。所以建议的使用方式就是：*启动校验（即 Controller 层）时使用 @Validated 注解，嵌套校验时使用 @Valid 注解*，这样就能同时使用分组校验和嵌套校验功能。

1.分组校验

1. 在@NotBlank等注解的属性多加一个*groups*，写法如：groups= {UpdateUser.class,InsertUser.class}
2. 在*Controller*用@Validated时加上*属性如value=InsertUser.class*

2.自定义校验注解

例子：性别枚举，校验传入的性别是否在男女范围内

1. 写个*性别枚举类*
2. *自定义约束注解*，注解上面用注解修饰：*@Constraint(validateBy=InEnumValidator.class)*

3. *自定义约束校验器实现ConstraintValidator*，校验通过返回true否则返回false
    1. *2和3两点暂时没搞懂具体逻辑，等用上了再研究吧*
4. *实体类的成员加上注解*如：@InEnum(enumType=GenderEnum.class,message="用户性别不在枚举范围内")

2.Service

1. 入参是平铺的参数
    1. 在*父类/接口*的*方法入参*加注解约束(@NotBlank等)
    2. 用 *@Validated*修饰*实现类*
2. 入参是对象
    1. *入参类*里面在成员上加 *@NotNull*等注解约束
    2. *父类/接口*的入参加 *@Valid（便于嵌套校验）*
    3. @Validated修饰*实现类*

4.