

[🏠 / 开发指南 / 后端手册](#)[👤 芋道源码](#) [📅 2022-03-25](#)

异常处理（错误码）

本章节，将讲解异常相关的统一响应、异常处理、业务异常、错误码这 4 块的内容。

1. 统一响应

后端提供 RESTful API 给前端时，需要响应前端 API 调用是否成功：

- 如果成功，成功的数据是什么。后续，前端会将数据渲染到页面上
- 如果失败，失败的原因是什么。一般，前端会将原因弹出提示给用户

因此，需要有**统一响应**，而不能是每个接口定义自己的风格。一般来说，统一响应返回信息如下：

- 成功时，返回成功的状态码 + 数据
- 失败时，返回失败的状态码 + 错误提示

在标准的 RESTful API 的定义，是推荐使用 [HTTP 响应状态码](#) 作为状态码。一般来说，我们实践很少这么去做，主要原因如下：

- 业务返回的错误状态码很多，HTTP 响应状态码无法很好的映射。例如说，活动还未开始、订单已取消等等
- 学习成本高，开发者对 HTTP 响应状态码不是很了解。例如说，可能只知道 200、403、404、500 几种常见的

1.1 CommonResult

[ruoyi-vue-pro](#) 项目在实践中，将状态码放在 Response Body 响应内容中返回。一共有 3 个字段，通过 [CommonResult](#) 定义如下：

```
13  /**
14  * 通用返回
15  *
16  * @param <T> 数据泛型
17  */
18  @Data
19  public class CommonResult<T> implements Serializable {
20
21      /**
22       * 状态码
23       * 1. 成功时，状态码为 0
24       * 2. 失败时，对应业务的错误码 {@link ErrorCode#getCode()}
25       */
26      private Integer code;
27
28      /**
29       * 数据。成功时，返回该字段。
30       */
31      private T data; 泛型：每个业务自己定义
32
33      /**
34       * 错误提示。失败时，返回该字段，用户可阅读
35       *
36       * @see ErrorCode#getMsg()
37       */
38      private String msg;
```

```
// 成功响应
{
    code: 0,
    data: {
        id: 1,
        username: "yudaoyuanma"
    }
}
```

```
// 失败响应
{
    code: 233666,
    message: "徐妈太丑了"
}
```

可以增加 success 字段吗？

有些团队在实践时，会增加 `success` 字段，通过 `true` 和 `false` 表示成功还是失败。

这个看每个团队的习惯吧。芳芳的话，还是偏好基于约定，返回 `0` 时表示成功。

失败时的 `code` 字段，使用全局的错误码，稍后在「4. 错误码」小节来讲解。

① 在 RESTful API 成功时，定义 Controller 对应方法的返回类型为 `CommonResult`，并调用 `#success(T data)` 方法来返回。代码如下图：

```
AuthController.java
55  @PostMapping("/login")
56  @ApiOperation("使用账号密码登录")
57  @OperateLog(enable = false) // 避免 Post 请求被记录操作日志
58  public CommonResult<AuthLoginRespVO> login(@RequestBody @Valid AuthLoginReqVO reqVO) {
59      String token = authService.login(reqVO, getClientIP(), getUserAgent());
60      // 返回结果
61      return success(AuthLoginRespVO.builder().token(token).build());
62  }
63
64  @GetMapping("/get-permission-info")
65  @ApiOperation("获取登录用户的权限信息")
66  public CommonResult<AuthPermissionInfoRespVO> getPermissionInfo() {
67      // 获得用户信息
68      AdminUserDO user = userService.getUser(getLoginUserId());
69      if (user == null) {...}
70      // 获得角色列表
71      List<RoleDO> roleList = roleService.getRolesFromCache(getLoginUserRoleIds());
72      // 获得菜单列表
73      List<MenuDO> menuList = permissionService.getRoleMenuListFromCache(
74          getLoginUserRoleIds(), // 注意，基于登录的角色，因为后续的权限判断也是基于它
75          SetUtils.asSet(MenuTypeEnum.DIR.getType(), MenuTypeEnum.MENU.getType(), MenuTypeEnum.BUTTON.getType()),
76          SetUtils.asSet(CommonStatusEnum.ENABLE.getStatus()));
77      // 拼接结果返回
78      return success(AuthConvert.INSTANCE.convert(user, roleList, menuList));
79  }
```

CommonResult 的 `data` 字段是泛型，建议定义对应的 VO 类，而不是使用 Map 类。

② 在 RESTful API 失败时，通过抛出 Exception 异常，具体在「2. 异常处理」小节。

1.2 使用 @ControllerAdvice ?

在 Spring MVC 中，可以使用 `@ControllerAdvice` 注解，通过 Spring AOP 拦截修改 Controller 方法的返回结果，从而实现全局的统一返回。

使用 @ControllerAdvice 注解的实战案例？

如果你感兴趣的话，可以阅读《芋道 Spring Boot SpringMVC 入门》[文章](#)的「4. 全局统一返回」小节。

为什么项目不采用这种方式呢？主要原因是，这样的方式“破坏”了方法的定义，导致一些隐性的问题。例如说，Swagger 接口定义错误，展示的响应结果不是 CommonResult。

还有个原因，部分 RESTful API 不需要自动包装 CommonResult 结果。例如说，第三方支付回调只需要返回 "success" 字符串。

2. 异常处理

RESTful API 发生异常时，需要拦截 Exception 异常，转换成统一响应的格式，否则前端无法处理。

2.1 Spring MVC 的异常

在 Spring MVC 中，通过 `@ControllerAdvice` + `@ExceptionHandler` 注解，声明将指定类型的异常，转换成对应的 CommonResult 响应。实现的代码，可见 [GlobalExceptionHandler](#) 类，代码如下：

```
GlobalExceptionHandler.java
159
160 /** 处理 SpringMVC 请求方法不正确 ...*/
161 @ExceptionHandler({RequestMethodNotSupportedException.class})
162 public CommonResult<> httpRequestMethodNotSupportedExceptionHandler(RequestMethodNotSupportedException ex) {...}
171
172 /** 处理 Resilience4j 限流抛出的异常 */
173 @ExceptionHandler(value = RequestNotPermitted.class)
174 public CommonResult<> requestNotPermittedExceptionHandler(HttpServletRequest req, RequestNotPermitted ex) {...}
179
180 /** 处理 Spring Security 权限不足的异常 ...*/
181 @ExceptionHandler(value = AccessDeniedException.class)
182 public CommonResult<> accessDeniedExceptionHandler(HttpServletRequest req, AccessDeniedException ex) {
183     log.warn("[accessDeniedExceptionHandler][userId({})] 无法访问 url({})", WebFrameworkUtils.getLoginUserId(req),
184             req.getRequestURL(), ex);
185     return CommonResult.error(FORBIDDEN);
186 }
191
192 /** 处理 业务异常 ServiceException ...*/
193 @ExceptionHandler(value = ServiceException.class)
194 public CommonResult<> serviceExceptionHandler(ServiceException ex) {
195     log.info("[serviceExceptionHandler]", ex);
196     return CommonResult.error(ex.getCode(), ex.getMessage());
197 }
202
203 /** 处理系统异常，兜底 处理所有的一切 */
204 @ExceptionHandler(value = Exception.class)
205 public CommonResult<> defaultExceptionHandler(HttpServletRequest req, Throwable ex) {
206     log.error("[defaultExceptionHandler]", ex);
207     // 插入异常日志
208     this.createExceptionLog(req, ex);
209     // 返回 ERROR CommonResult
210     return CommonResult.error(INTERNAL_SERVER_ERROR.getCode(), INTERNAL_SERVER_ERROR.getMsg());
211 }
212
213 }
```

2.2 Filter 的异常

在请求被 Spring MVC 处理之前，是先经过 Filter 处理的，此时发生异常时，是无法通过 `@ExceptionHandler` 注解来处理的。只能通过 `try catch` 的方式来实现，代码如下：

```
JWTAAuthenticationTokenFilter.java
34
35 @Override
36 /NullableProblems/
37 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
38                               FilterChain chain)
39     throws ServletException, IOException {
40     String token = SecurityFrameworkUtils.obtainAuthorization(request, securityProperties
41         .getTokenHeader());
42     if (StringUtil.isEmpty(token)) {
43         try {
44             // 验证 token 有效性
45             LoginUser loginUser = authenticationProvider.verifyTokenAndRefresh(request, token);
46             // 模拟 Login 功能，方便日常开发测试
47             if (loginUser == null) {
48                 loginUser = this.mockLoginUser(request, token);
49             }
50             // 设置当前用户
51             if (loginUser != null) {
52                 SecurityFrameworkUtils.setLoginUser(loginUser, request);
53             }
54         } catch (Throwable ex) {
55             CommonResult<> result = globalExceptionHandler.allExceptionHandler(request, ex);
56             ServletUtils.writeJSON(response, result);
57             return;
58         }
59     }
60 }
```

```
GlobalExceptionHandler.java
51
52 /**
53  * 处理所有异常，主要是提供给 Filter 使用
54  * 因为 Filter 不走 SpringMVC 的流程，但是我们又需要兜底处理异常，所以这里提供一个主要的异常处理
55  * 过程，保持逻辑统一。
56  * @param request 请求
57  * @param ex 异常
58  * @return 通用返回
59  */
60 public CommonResult<> allExceptionHandler(HttpServletRequest request, Throwable
61     ex) {
62     if (ex instanceof MissingServletRequestParameterException) {
63         return missingServletRequestParameterExceptionHandler(
64             (MissingServletRequestParameterException) ex);
65     }
66     if (ex instanceof MethodArgumentTypeMismatchException) {
67         return methodArgumentTypeMismatchExceptionHandler(
68             (MethodArgumentTypeMismatchException) ex);
69     }
70     if (ex instanceof BindException) {
71         return bindExceptionHandler((BindException) ex);
72     }
73     if (ex instanceof ConstraintViolationException) {
74         return constraintViolationExceptionHandler((ConstraintViolationException)
75             ex);
76     }
77 }
```

3. 业务异常

在 Service 发生业务异常时，如果进行返回呢？例如说，用户名已经存在，商品库存不足等。常用的方案选择，主要有两种：

- 方案一，使用 `CommonResult` 统一响应结果，里面有错误码和错误提示，然后进行 `return` 返回
- 方案二，使用 `ServiceException` 统一业务异常，里面有错误码和错误提示，然后进行 `throw` 抛出

选择方案一 `CommonResult` 会存在两个问题：

- 因为 Spring `@Transactional` 声明式事务，是基于异常进行回滚的，如果使用 `CommonResult` 返回，则事务回滚会非常麻烦
- 当调用别的方法时，如果别人返回的是 `CommonResult` 对象，还需要不断的进行判断，写起来挺麻烦的

因此，项目采用方案二 `ServiceException` 异常。

3.1 ServiceException

定义 `ServiceException` 异常类，继承 `RuntimeException` 异常类（非受检），用于定义业务异常。代码如下：

```
public final class ServiceException extends RuntimeException {  
  
    /**  
     * 业务错误码 对应 CommonResult 的 code 字段  
     *  
     * @see ServiceErrorCodeRange  
     */  
    private Integer code;  
  
    /**  
     * 错误提示 对应 CommonResult 的 msg 字段  
     */  
    private String message;  
  
    /** 空构造方法，避免反序列化问题 */  
    public ServiceException() {}  
  
    public ServiceException(ErrorCode errorCode) {  
        this.code = errorCode.getCode();  
        this.message = errorCode.getMsg();  
    }  
  
    public ServiceException(Integer code, String message) {  
        this.code = code;  
        this.message = message;  
    }  
}
```

为什么继承 `RuntimeException` 异常？

大多数业务场景下，我们无需处理 `ServiceException` 业务异常，而是通过 `GlobalExceptionHandler` 统一处理，转换成对应的 `CommonResult` 对象，进而提示给前端即可。

如果真的需要处理 `ServiceException` 时，通过 `try catch` 的方式进行主动捕获。

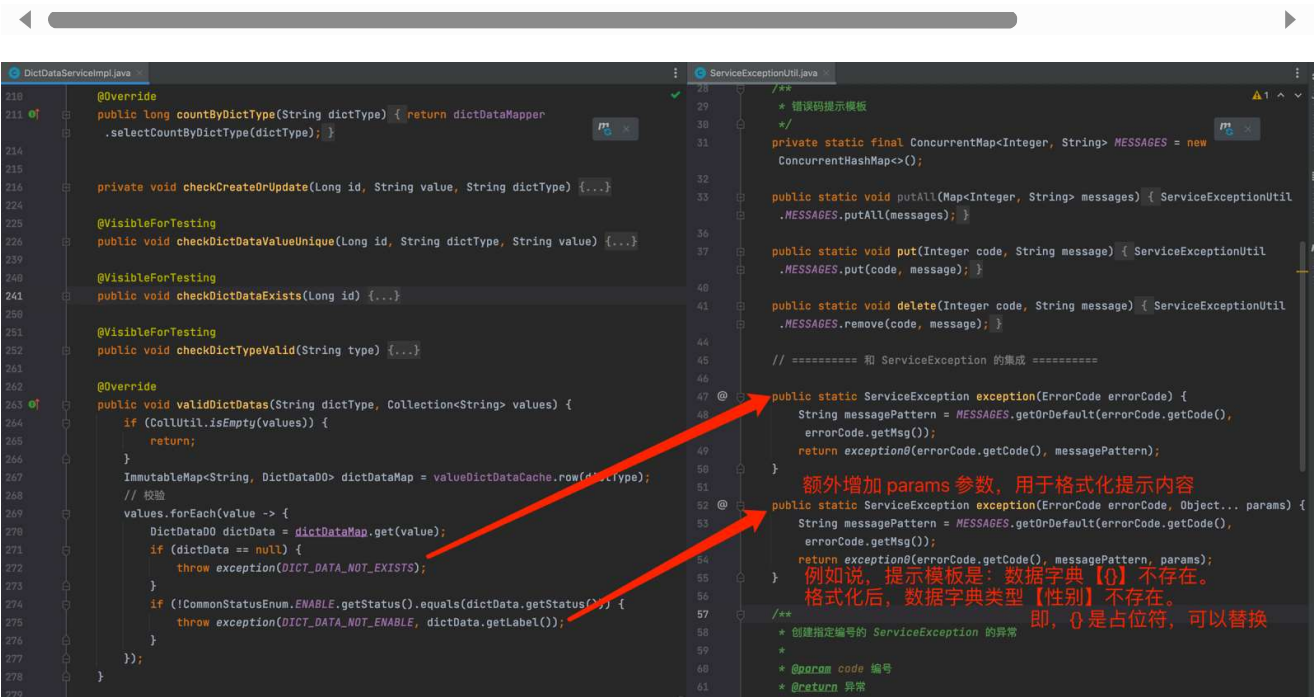
3.2 ServiceExceptionUtil

在 Service 需抛出业务异常时，通过调用 `ServiceExceptionUtil` 的 `#exception(ErrorCode errorCode, Object... params)` 方法来构建 `ServiceException` 异常，然后使用 `throw` 进行

抛出。代码如下：

```
// ServiceExceptionUtil.java
```

```
public static ServiceException exception(ErrorCode errorCode) { /** 省略参数 */ }
public static ServiceException exception(ErrorCode errorCode, Object... params)
```



为什么使用 ServiceExceptionUtil 来构建 ServiceException 异常？

错误提示的内容，支持使用管理后台进行动态配置，所以通过 ServiceExceptionUtil 获取内容的配置与格式化。

4. 错误码

错误码，对应 `ErrorCode` 类，枚举项目中的错误，**全局唯一**，方便定位是谁的错、错在哪。

```
@Data
public class ErrorCode {

    /**
     * 错误码
     */
    private final Integer code;

    /**
     * 错误提示
     */
    private final String msg;

    public ErrorCode(Integer code, String message) {
        this.code = code;
        this.msg = message;
    }
}
```

4.1 错误码分类

错误码分成两类：全局的系统错误码、模块的业务错误码。

4.1.1 系统错误码

全局的系统错误码，使用 0-999 错误码段，和 [HTTP 响应状态码](#) 对应。虽然说，HTTP 响应状态码作为业务使用表达能力偏弱，但是使用在系统层面还是非常不错的。

系统错误码定义在 [GlobalErrorCodeConstants](#) 类，代码如下：

```
public interface GlobalErrorCodeConstants {

    ErrorCode SUCCESS = new ErrorCode( code: 0, message: "成功");

    // ===== 客户端错误段 =====

    ErrorCode BAD_REQUEST = new ErrorCode( code: 400, message: "请求参数不正确");
    ErrorCode UNAUTHORIZED = new ErrorCode( code: 401, message: "账号未登录");
    ErrorCode FORBIDDEN = new ErrorCode( code: 403, message: "没有该操作权限");
    ErrorCode NOT_FOUND = new ErrorCode( code: 404, message: "请求未找到");
    ErrorCode METHOD_NOT_ALLOWED = new ErrorCode( code: 405, message: "请求方法不正确");
    ErrorCode LOCKED = new ErrorCode( code: 423, message: "请求失败，请稍后重试"); // 并发请求，不允许
    ErrorCode TOO_MANY_REQUESTS = new ErrorCode( code: 429, message: "请求过于频繁，请稍后重试");

    // ===== 服务端错误段 =====

    ErrorCode INTERNAL_SERVER_ERROR = new ErrorCode( code: 500, message: "系统异常");

    // ===== 自定义错误段 =====

    ErrorCode REPEATED_REQUESTS = new ErrorCode( code: 900, message: "重复请求，请稍后重试"); // 重复请求
    ErrorCode DEMO_DENY = new ErrorCode( code: 901, message: "演示模式，禁止写操作");

    ErrorCode UNKNOWN = new ErrorCode( code: 999, message: "未知错误");
}
```

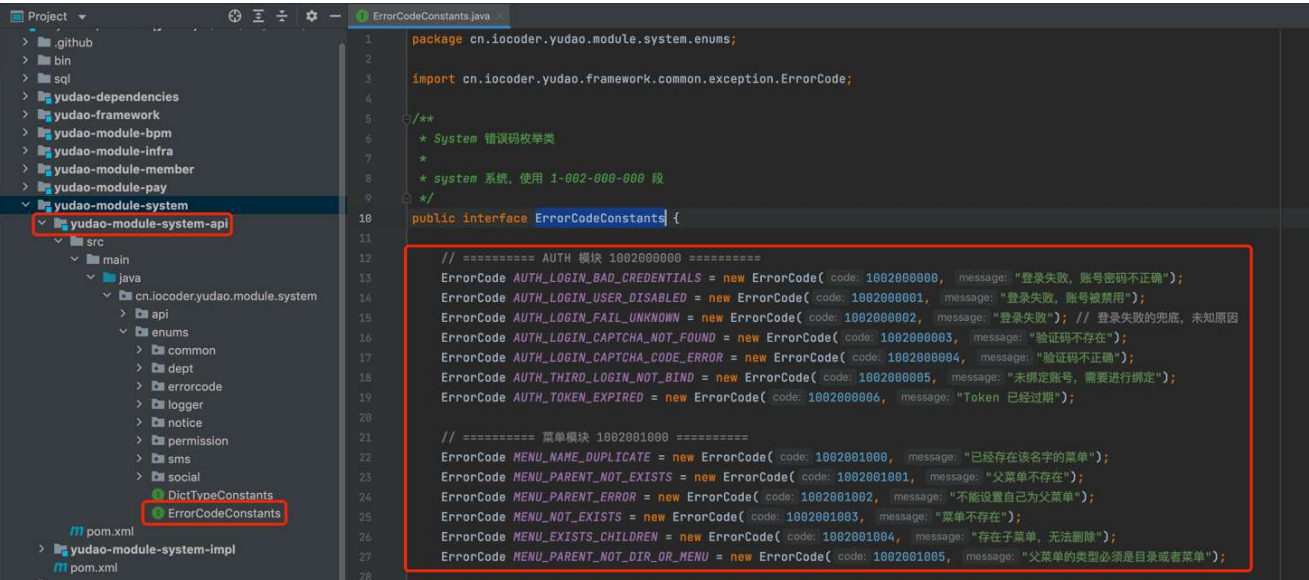
4.1.2 业务错误码

模块的业务错误码，按照模块分配错误码的**区间**，避免模块之间的错误码冲突。

① 业务错误码一共 10 位，分成 4 段，在 [ServiceErrorCodeRange](#) 分配，规则与代码如下图：

```
3  /**
4   * 业务异常的错误码区间，解决：解决各模块错误码定义，避免重复，在此只声明不做实际使用
5   *
6   * 一共 10 位，分成四段
7   *
8   * 第一段，1 位，类型
9   *     1 - 业务 module 模块的业务级别异常
10    *     2 - 框架 framework 模块的业务级别异常
11    * 第二段，3 位，系统类型
12    *     001 - 用户系统；002 - 商品系统；003 - 订单系统；004 - 支付系统；005 - 优惠券系统
13    * 第三段，3 位，模块
14    *     不限制规则。
15    *     一般建议，每个系统里面，可能有多个模块，可以再去分段。以用户系统为例子：
16    *         001 - OAuth2 模块；002 - User 模块；003 - MobileCode 模块
17    * 第四段，3 位，错误码
18    *     不限制规则。
19    *     一般建议，每个模块自增。
20    *
21    * @author 芋道源码
22    */
23    public class ServiceErrorCodeRange {
24
25        // 模块 infra 错误码区间 1-001-000-000
26        // 模块 system 错误码区间 1-002-000-000
27        // 模块 member 错误码区间 1-004-000-000
28        // 模块 pay 错误码区间 1-007-000-000
29        // 模块 bpm 错误码区间 1-009-000-000
30
31        // 框架 sms 错误码区间 2-001-000-000
32        // 框架 pay 错误码区间 2-002-000-000
33
34    }
```

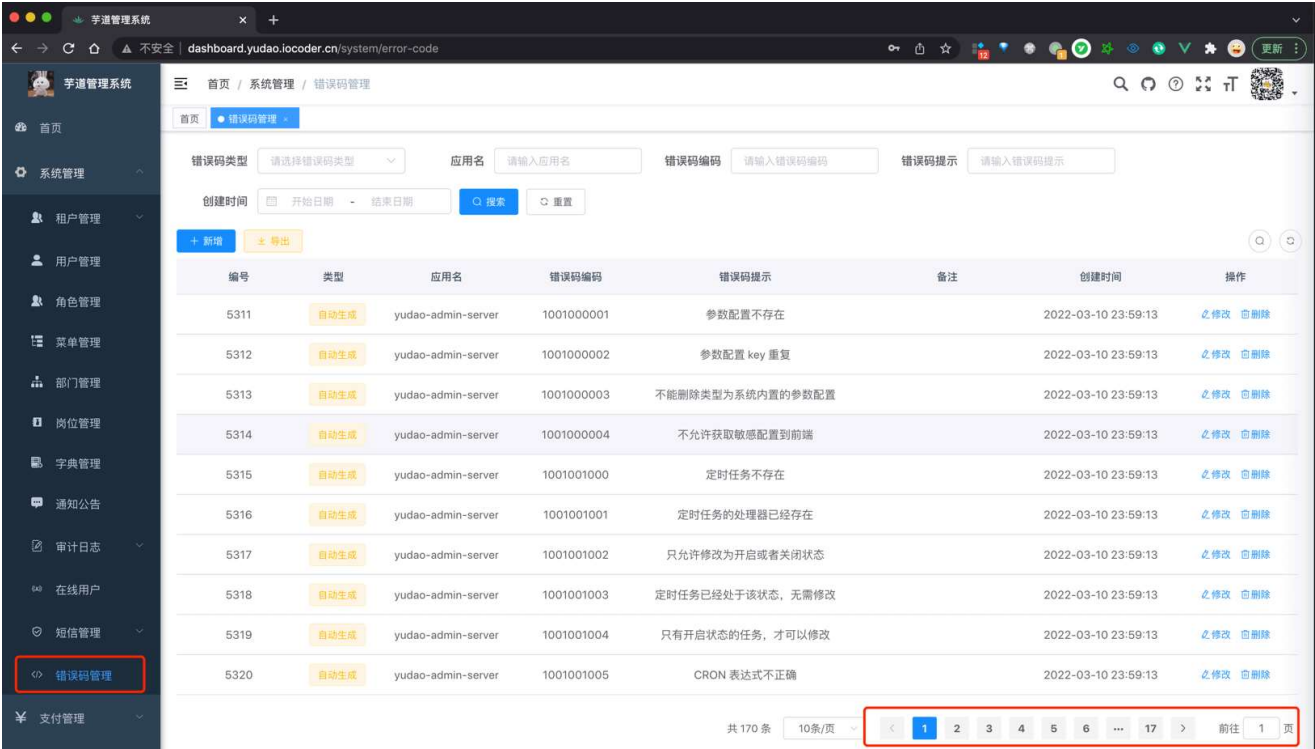
② 每个业务模块，定义自己的 ErrorCodeConstants 错误码枚举类。以 yudao-module-system 模块举例子，代码如下：



```
1 package cn.iocoder.yudao.module.system.enums;
2
3 import cn.iocoder.yudao.framework.common.exception.ErrorCode;
4
5 /**
6  * System 错误码枚举类
7  *
8  * system 系统，使用 1-002-000-000 段
9  */
10 public interface ErrorCodeConstants {
11
12     // ===== AUTH 模块 1002000000 =====
13     ErrorCode AUTH_LOGIN_BAD_CREDENTIALS = new ErrorCode( code: 1002000000, message: "登录失败，账号密码不正确");
14     ErrorCode AUTH_LOGIN_USER_DISABLED = new ErrorCode( code: 1002000001, message: "登录失败，账号被禁用");
15     ErrorCode AUTH_LOGIN_FAIL_UNKNOWN = new ErrorCode( code: 1002000002, message: "登录失败"); // 登录失败的兜底，未知原因
16     ErrorCode AUTH_LOGIN_CAPTCHA_NOT_FOUND = new ErrorCode( code: 1002000003, message: "验证码不存在");
17     ErrorCode AUTH_LOGIN_CAPTCHA_CODE_ERROR = new ErrorCode( code: 1002000004, message: "验证码不正确");
18     ErrorCode AUTH_THIRD_LOGIN_NOT_BIND = new ErrorCode( code: 1002000005, message: "未绑定账号，需要进行绑定");
19     ErrorCode AUTH_TOKEN_EXPIRED = new ErrorCode( code: 1002000006, message: "Token 已经过期");
20
21     // ===== 菜单模块 1002001000 =====
22     ErrorCode MENU_NAME_DUPLICATE = new ErrorCode( code: 1002001000, message: "已经存在该名字的菜单");
23     ErrorCode MENU_PARENT_NOT_EXISTS = new ErrorCode( code: 1002001001, message: "父菜单不存在");
24     ErrorCode MENU_PARENT_ERROR = new ErrorCode( code: 1002001002, message: "不能设置自己为父菜单");
25     ErrorCode MENU_NOT_EXISTS = new ErrorCode( code: 1002001003, message: "菜单不存在");
26     ErrorCode MENU_EXISTS_CHILDREN = new ErrorCode( code: 1002001004, message: "存在子菜单，无法删除");
27     ErrorCode MENU_PARENT_NOT_DIR_OR_MENU = new ErrorCode( code: 1002001005, message: "父菜单的类型必须是目录或者菜单");
28 }
```

4.2 错误码管理

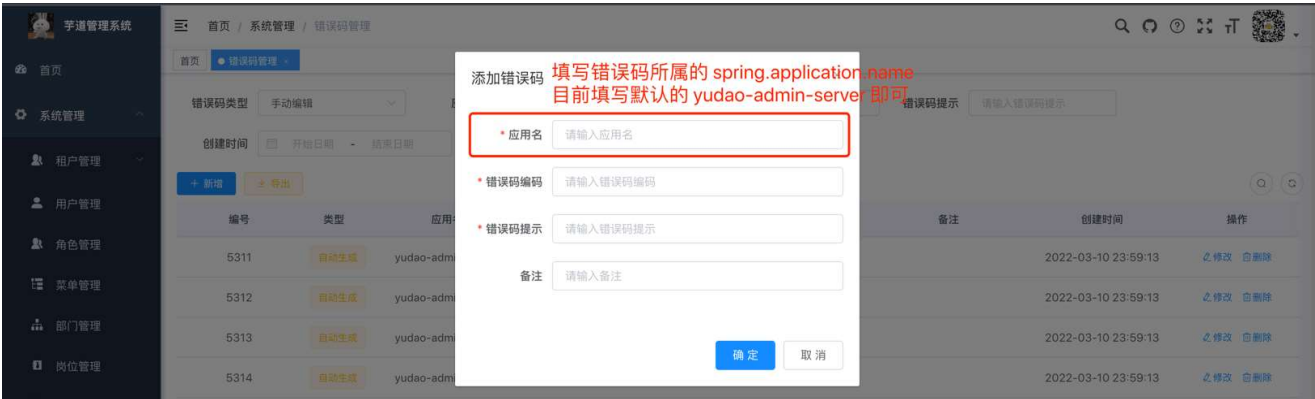
在管理后台的 [系统管理 -> 错误码管理] 菜单，可以进行错误码的管理。



启动中的项目会每 60 秒，加载最新的错误码配置。所以，我们在修改完错误码的提示后，无需重启项目。

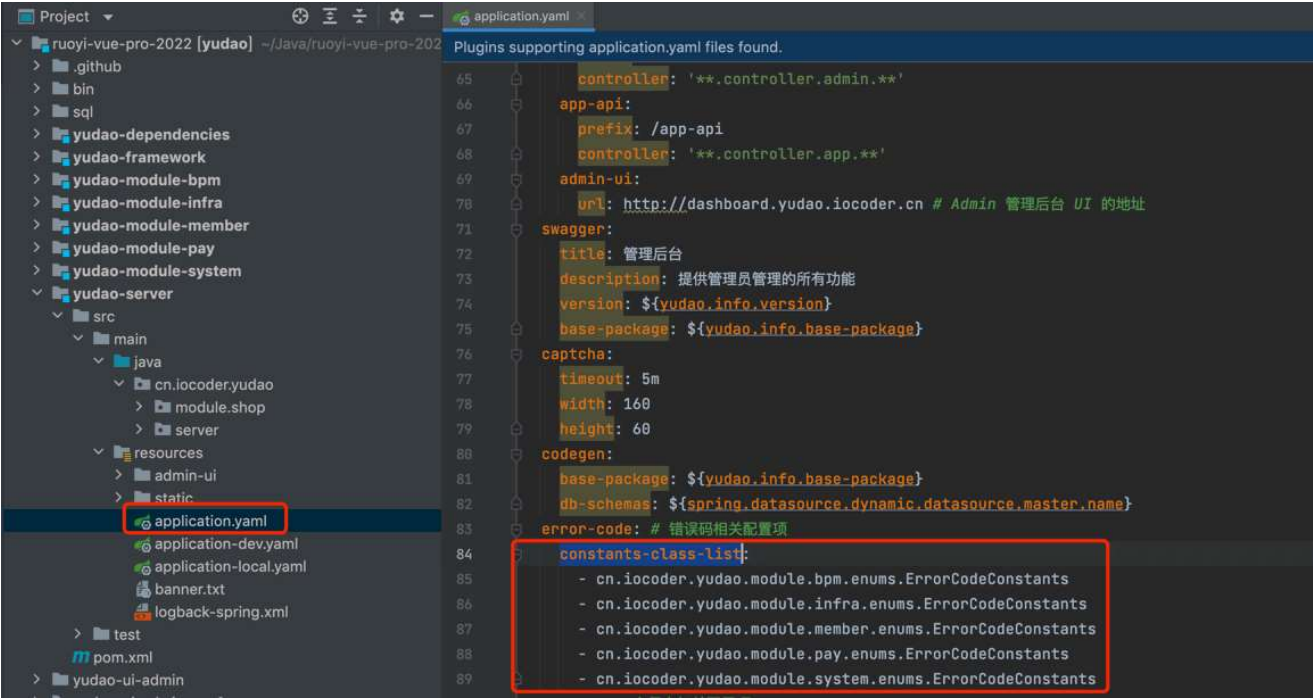
4.2.1 手动添加

点击 [新增] 按钮，进行错误码的手动添加。如下图所示：



4.2.2 自动添加

通过 `yudao.error-code.constants-class-list` 配置项，设置需要自动添加的 `ErrorCodeConstants` 错误码枚举类。如下图所示：



项目启动时，会自动扫描对应的 ErrorCodeConstants 中的错误码，自动添加或修改错误码的配置。

注意，自动添加的错误码的类型为【自动生成】，一旦在管理后台手动 [编辑] 后，该错误码就不再支持自动修改。

自动添加是如何实现的？

参见 [system/framework/errorcode](#) 包的代码。

← [WebSocket 实时通信](#)

[参数校验、时间传参](#) →



Theme by [Vdoing](#) | Copyright © 2019-2024 芋道源码 | MIT License