Knife4j使用说明

前后端分离

在一个采用前后端分离开发模式的团队中，想要前后端人员高效合作，提高团队整体开发效率，通常都需要做好以下几点：

- 后端编写一份完善的接口文档，接口中的各种细节都要明确
- 开接口评审会，前后端达成一致后，再各自开始编码
- 严格约定开发和联调的边界，严禁一边开发一边联调
- 前后端各自测试好自己编写的功能后再联调，避免联调时问题频出

接口文档要素

想要编写一个完善的接口文档需要注意以下几点：

- 接口名称：
- 接口地址
- 请求方式
- 请求参数
    - 参数名称
    - 参数类型
    - 参数说明
    - 是否必须：重点！！！
    - 参数示例
- 响应参数
    - 参数名称
    - 参数说明
    - 参数示例

后端编写接口文档的痛点

接口文档的要点并不多，但是想要拿出一份非常完善的文档也很困难，通常都需要花费很多时间。后端开发一个功能需要以下流程：

- 理解产品需求，根据需求设计数据库表接口
- 分析功能，确定有哪些接口
- 从数据库设计文档中挨个拷贝字段到 word 文档
- 从 word 文档中拷贝字段到代码中，编写代码
- 从 word 文档中拷贝字段到 postman 中，进行测试

以上步骤需要不断的重复，接口数量一多，就难免会出错。而且这个工作及其耗时，枯燥！

swagger 的作用

通过在 Controller 中接口和请求/响应参数中，添加注解信息来描述接口。最后扫描对应注解，并统一生成接口文档，提供在线调试工具。这种方式能够极大的提升开发效率，某种程度上还起到了规范代码编写的作用。

swagger、swagger-bootstrap-ui、knife4j 三者的关系及优缺点

swagger 是 SmartBear Software 组织开发的接口文档工具，后两者则是由国内的 萧明 在 swagger 的基础上改造的，提供了更多的功能，优化了使用方式。其中 knife4j 是 swagger-bootstrap-ui 的升级版。三者之中，尽管原理相同，但是推荐使用 knife4j。

swagger 的缺点：

- 无接口分组排序、分组中的接口排序功能
- 无接口模块区分，无法形成 模块-接口分组-接口 的三级排序
- 文档 UI 界面查看和调试不方便
- 无法导出文档

萧明在 swagger 的基础上提供了更多的功能，如下：

- 提供了 模块-分组-接口 三级接口组织接口文档
- 提供了分组排序、接口排序功能
- 更美观易用的 UI 界面、接口要素都能清晰展示
- 拥有侧边栏，调试也更为方便
- 导出文档功能已完善，格式良好，稍作修改即可作为对外发布文档

但是更为推荐使用 knife4j ，而不是 swagger-bootstrap-ui 的理由如下：

- UI 界面略丑
- 作者已不再更新和维护
- 导出文档功能格式没有达到稍作修改即可对外发布的标准

如何在项目中集成 knife4j

1. 导入依赖

```
<dependency>
    <groupId>com.github.xiaoymin</groupId>
    <artifactId>knife4j-spring-boot-starter</artifactId>
    <version>2.0.7</version>
</dependency>
```

2. 在 application.yml 文件中做出以下配置

```
knife4j:
    enable: true # 开启 knife4j 增强功能
    basic:
        enable: true # 开启文档访问权限（首次访问，会在浏览器开启弹框要求输入用户名密码）
        username: admin # 账号
        password: 123456 # 密码
```

3. 在拦截器或者过滤器中放行以下路径

```
String[] knife4jPaths = new String[]{
    "/doc.html",
    "/webjars/**",
    "/swagger-resources/**",
    "/swagger-ui.html/**",
    "/v2/**",
    "/favicon.ico",
    "/error"
};
```

4. 创建 knife4j 配置类

```
@Configuration
@EnableSwagger2WebMvc
public class Knife4jConfiguration {

    /**
     * 如果对项目中的接口进行分模块，拷贝多份即可
     *
     * @return r
     */
    @Bean
    public Docket sys() {
        return new Docket(DocumentationType.SWAGGER_2)
                .useDefaultResponseMessages(false) // 关闭 swagger 默认响应状态码
                .groupName("一、系统管理模块") // 指定模块名称
                .apiInfo(systemMangerInfo())
                .select()

.apis(RequestHandlerSelectors.basePackage("com.keqi.knife4j.sys")) // 扫描指定包路径下的接口
                .paths(PathSelectors.any())
                .build();
    }

    private ApiInfo systemMangerInfo() {
        return new ApiInfoBuilder()
                .title("knife4j")
                .description("knife4j 项目接口文档")
                .termsOfServiceUrl("http://localhost:9090/knife4j")
                .version("1.0")
                .build();
    }
}
```

knife4j 相关注解

真正常用的注解只有 6 个，学习起来也没有任何成本。

用在 Controller 类上的注解

- @Api
  - 作用：填写 Controller 类中所有接口总的信息，也就是接口分组的信息
  - 属性：
    - tags：填写分组的名称
- @ApiSupport
  - 作用：用于给同一个模块下的多个 Controller 类进行排序
  - 属性：
    - order：指定 Controlelr 的顺序，默认是从小到大的
    - author：用于指定这个 Controller 中所有接口的作者，作用范围是整个 Controller 类

用在 Controller 接口方法上的注解

- @ApiOperation
  - 作用：用在 Controller 接口上方，填写接口的描述信息
  - 属性：
    - value：填写接口名称
    - notes：填写接口的备注信息，有时候需要额外的信息来描述一个接口。（不是必填项，如果不填写，UI 界面上就不会有 "接口描述" 这一项）
- @ApiOperationSupport
  - 作用：用在 Controller 接口上方，用于给一个 Controller 中的接口在 UI 界面上进行排序
  - 属性：
    - order：指定接口的顺序，默认是从小到大的
    - author：指定接口的作者（不是必填项，如果不填写，UI 界面上就不会出现 "开发者" 这一项，如果填写则会覆盖 @ApiOperation 注解的 author 属性）
    - ignoreParameters：增加和修改共用一个 xxxParam 类的时候，修改是必须要传id的，新增则不用。这个时候就可以在增加的接口上使用此属性，来指定忽略某个参数。
      - content-type 为 application/json 时，需要使用 `实体类参数名称.属性名` 的方式忽略指定属性
      - content-type 为 x-www-form-urlencoded 时，直接使用 `属性名` 的方式忽略指定属性即可

用在请求参数和响应参数上的注解

- @ApiModelProperty
  - 作用：用在请求参数或者响应参数实体类的属性中
  - 属性：
    - value：指定属性的名称，直接会显示在 UI 界面上
    - example：用于给定一个示例值，直接和 value 属性一同在 UI 界面上显示。在使用 debug 调试时，默认会取这个值作为参数。请求和响应示例中的 JSON 结构，也会使用这个值显示。
    - required：默认值是 flase，如果该参数是必须传递的，那么一定要设置此值为 true，同样会显示在 UI 界面上显示出来。
    - hidden：尽管编码规范要求 Param 类和 VO 类不允许出现额外的属性，但有时真的有点不可避免。那么就可以使用将 hidden 属性设置为 true，则不会再 UI 界面上显示出来
- @ApiImplicitParam / @ApimplicitParams

    - 作用：如果只有一两个参数时，懒得单独封装成一个实体类，那么就可以使用表单的方式提交，这时就可以使用此注解来标识请求参数
    - 属性：
        - name：和 @RequestParam 注解 value 属性的值一样，也就是要和方法中形参名称一样（必须搭配此注解，否则不会生效）
        - value：同上
        - example：同上
        - required：同上
    - 补充：如果接口方法中只有一个参数，那么直接使用 @ApiImplicitParam 注解即可，如果有多个，那么就需要使用 @ApimplicitParams 注解的 value 属性来包含多个 @ApiImplicitParam 注解来指定请求参数。

knife4j 最佳实践

想要写一份真正清楚明确的接口文档，有以下规则必须要遵守：

- 项目包结构按照业务功能划分，而不是三层结构划分（此种方式能够利用 knife4j 提供的分模块功能，控制接口数量，便于查找接口）。最佳包结构如下：

```
- sys
    - controller
    - domain
        - db
        - param
        - vo
    - mapper
    - service
        - impl
    - 其它包
```

- 改变传统的编码方式，不再从上到下使用一个实体类。
    - db：命名方式为 xxxDO，实体类中的字段和数据库表结构一一对应，不多加任何一个字段。
    - param：命名方式为 xxxParam，增加和修改接口公用一个请求参数类，其他接口每个接口都单独封装一个实体类，除非只有一个两个参数可以使用表单方式提交
        - 如果是必传的请求参数，必须要填写 @ApiModelProperty 注解的 value、example、required 属性
        - 如果不是必传的请求参数，仅填写 @ApiModelProperty 注解的 value 属性即可（不填写 example 属性，是为了在 debug 调试时，能够明确区分非必填参数）
    - vo：命名方式为 xxxVO，任何一个具有返回值的接口都需要单独封装一个响应参数类
        - VO 实体类中的每个属性都必须要填写 @ApiModelProperty 注解的 value 和 example 属性值。
- 必须要指定 Controller 之间的顺序以及 Controller 之间接口的顺序。且命名上也要加上对应的序号，如 1.1 增加用户， 1.2 修改用户 等
- 实现 ResponseBodyAdvice 接口，统一处理 Controller 中 @ResponseBody 注解响应的接口返回值。代码更为简洁，且在需要展示多层响应结果时，能减少一个层次。
    - 使用示例如下：

```
@ApiOperation(value = "2.3根据ID删除字典项")
@ApiOperationSupport(order = 3)
@ApiImplicitParam(name = "id", value = "字典项ID", example = "1",
required = true)
```

```
@PostMapping("/deleteById")
public void deleteById(@RequestParam Long id) {
    this.dictItemService.deleteById(id);
}

@ApiOperation(value = "2.4 分页查询字典项列表")
@ApiOperationSupport(order = 4)
@PostMapping("/page")
public PageVO<DictItemVO> page(@RequestBody DictItemPageParam pageParam)
{
    return this.dictItemService.page(pageParam);
}
// 没有返回参数的直接使用 void 即可，有返回参数的接口才需要使用响应对象VO 来作为方法返回值
```

- ResponseBodyAdvice 接口实现类如下：

```
@ControllerAdvice(basePackages = "com.keqi.bestpracticeone")
public class ResultResponseBodyAdvice implements
ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter methodParameter, Class
aClass) {
        return true;
    }

    @Override
    public Object beforeBodyWrite(Object o, MethodParameter
methodParameter, MediaType mediaType, Class aClass, ServerHttpRequest
serverHttpRequest, ServerHttpResponse serverHttpResponse) {
        // 获取 Controler 中方法的返回值类型对应的 Class 反射对象
        Type type = methodParameter.getGenericParameterType();

        if (String.class == type || ResultEntity.class == type) {
            return o;
        } else if (void.class == type) {
            return ResultEntityBuilder.success();
        } else {
            return ResultEntityBuilder.success(o);
        }
    }
}
```