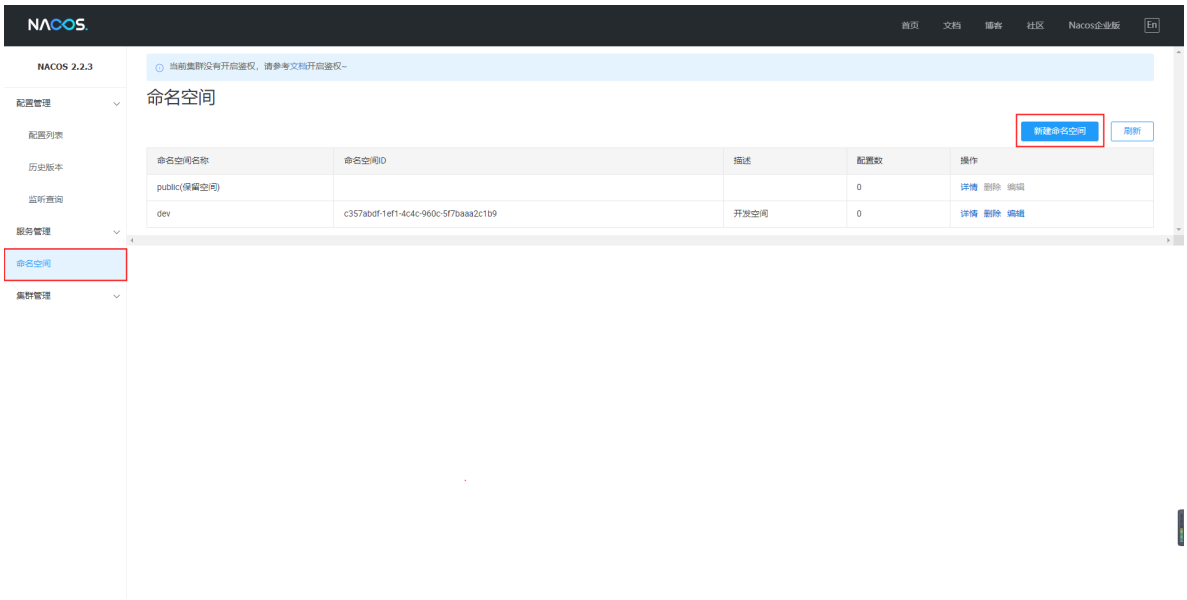
快速启动

初始化Nacos

在官网下载稳定版本后快速启动，具体参考https://nacos.io/zh-cn/docs/v2/quickstart/quick-start.html

启动完成后打开本地nacos地址http://localhost:8848/nacos/



如上图所示新建命名空间，将命名空间ID复制到微服务工程bootstrap-local.yaml文件中

```yaml
--- ##################### 注册中心相关配置 #####################

spring:
  cloud:
    nacos:
      server-addr: 127.0.0.1:8848
      discovery:
        namespace: c357abdf-1ef1-4c4c-960c-5f7baaa2c1b9 # 命名空间，这里使用 dev 开发环境

--- ##################### 配置中心相关配置 #####################

spring:
  cloud:
    nacos:
      # Nacos Config 配置项，对应 NacosConfigProperties 配置属性类
      config:
        server-addr: 127.0.0.1:8848 # Nacos 服务器地址
        namespace: c357abdf-1ef1-4c4c-960c-5f7baaa2c1b9 # 命名空间，这里使用 dev 开发环境
        group: DEFAULT_GROUP # 使用的 Nacos 配置分组，默认为 DEFAULT_GROUP
        name: # 使用的 Nacos 配置集的 dataId，默认为 spring.application.name
        file-extension: yaml # 使用的 Nacos 配置集的 dataId 的文件拓展名，同时也是 Nacos 配置集的配置格式，默认为 properties
```

启动后端服务

启动gateway服务

启动完成后，使用浏览器访问 http://127.0.0.1:48080 (opens new window)地址，返回如下 JSON 字符串，说明成功。

友情提示：注意，默认配置下，网关启动在 48080 端口。

```
{"code":404,"data":null,"msg":null}
```

启动system服务

启动完成后，使用浏览器访问 http://127.0.0.1:48081/admin-api/system/ (opens new window) 和 http://127.0.0.1:48080/admin-api/system/ (opens new window)地址，都返回如下 JSON 字符串，说明成功。

友情提示：注意，默认配置下，`zjugis-module-system` 服务启动在 48081 端口。

```
{"code":401,"data":null,"msg":"账号未登录"}
```

启动infra服务

启动完成后，使用浏览器访问 http://127.0.0.1:48082/admin-api/infra/ (opens new window)和 http://127.0.0.1:48080/admin-api/infra/ (opens new window)地址，都返回如下 JSON 字符串，说明成功。

友情提示：注意，默认配置下，`zjugis-module-infra` 服务启动在 48081 端口。

```
{"code":401,"data":null,"msg":"账号未登录"}
```

启动前端服务

```
# 安装 pnpm，提升依赖的安装速度
npm config set registry https://registry.npmjs.org
npm install -g pnpm
# 安装依赖
pnpm install

# 启动服务
npm run dev
```

项目结构

后端结构

后端采用模块化的架构，按照功能拆分成多个 Maven Module，提升开发与研发的效率，带来更好的可维护性。

一共有**四**类 Maven Module：

Maven Module	作用
zjugis-dependencies	Maven 依赖版本管理
zjugis-framework	Java 框架拓展
zjugis-module-xxx	XXX 功能的 Module 模块

zjugis-dependencies

该模块是一个 Maven Bom，只有一个 pom.xml 文件，定义项目中所有 Maven 依赖的**版本号**，解决依赖冲突问题。

从定位上来说，它和 Spring Boot 的 `spring-boot-starter-parent` (opens new window)和 Spring Cloud 的 `spring-cloud-dependencies` (opens new window)是一致的。

实际上, 本质上还是个**单体**项目，直接在根目录 pom.xml 管理依赖版本会更加方便，也符合绝大多数程序员的认知。但是要额外考虑一个场景，如果每个 `zjugis-module-xxx` 模块都维护在一个独立的 Git 仓库，那么 `zjugis-dependencies` 就可以在多个 `zjugis-module-xxx` 模块下复用。

zjugis-framework

该模块下每个每个 Maven Module 都是一个组件，分成两种类型：

① 技术组件：技术相关的组件封装，例如说 MyBatis、Redis 等等。

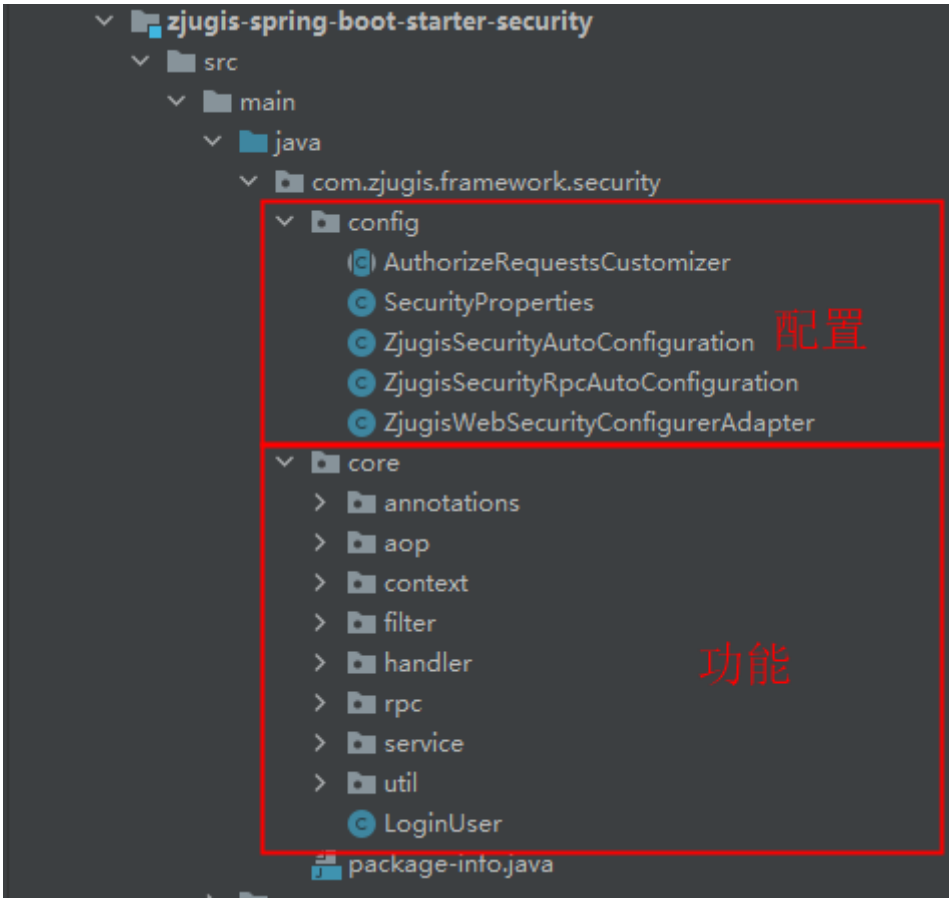
Maven Module	作用
zjugis-common	定义基础 pojo 类、枚举、工具类等
zjugis-spring-boot-starter-web	Web 封装，提供全局异常、访问日志等
zjugis-spring-boot-starter-security	认证授权，基于 Spring Security 实现
zjugis-spring-boot-starter-mybatis	数据库操作，基于 MyBatis Plus 实现
zjugis-spring-boot-starter-redis	缓存操作，基于 Spring Data Redis + Redisson 实现
zjugis-spring-boot-starter-rpc	服务调用，基于 Feign 实现
zjugis-spring-boot-starter-mq	消息队列，基于 RocketMQ 实现，支持集群消费和广播消费
zjugis-spring-boot-starter-job	定时任务，基于 XXL Job 实现，支持集群模式
zjugis-spring-boot-starter-env	多环境，实现类似阿里的特性环境的能力
zjugis-spring-boot-starter-flowable	工作流，基于 Flowable 实现
zjugis-spring-boot-starter-protection	服务保障，基于 Sentinel 实现，提供幂等、分布式锁、限流、熔断等功能
zjugis-spring-boot-starter-file	文件客户端，支持将文件存储到 S3（MinIO、阿里云、腾讯云、七牛云）、本地、FTP、SFTP、数据库等
zjugis-spring-boot-starter-excel	Excel 导入导出，基于 EasyExcel 实现
zjugis-spring-boot-starter-monitor	服务监控，提供链路追踪、日志服务、指标收集等功能
zjugis-spring-boot-starter-captcha	验证码 Captcha，提供滑块验证码
zjugis-spring-boot-starter-test	单元测试，基于 Junit + Mockito 实现
zjugis-spring-boot-starter-banner	控制台 Banner，启动打印各种提示
zjugis-spring-boot-starter-desensitize	脱敏组件：支持 JSON 返回数据时，将邮箱、手机等字段进行脱敏

② 业务组件：业务相关的组件封装，例如说数据字典、操作日志等等。如果是业务组件，名字会包含 `biz` 关键字。

Maven Module	作用
zjugis-spring-boot-starter-biz-tenant	SaaS 多租户
zjugis-spring-boot-starter-biz-data-permissionn	数据权限
zjugis-spring-boot-starter-biz-dict	数据字典
zjugis-spring-boot-starter-biz-operatelog	操作日志
zjugis-spring-boot-starter-biz-pay	支付客户端，对接微信支付、支付宝等支付平台
zjugis-spring-boot-starter-biz-sms	短信客户端，对接阿里云、腾讯云等短信服务
zjugis-spring-boot-starter-biz-social	社交客户端，对接微信公众号、小程序、企业微信、钉钉等三方授权平台
zjugis-spring-boot-starter-biz-weixin	微信客户端，对接微信的公众号、开放平台等
zjugis-spring-boot-starter-biz-error-code	全局错误码
zjugis-spring-boot-starter-biz-ip	地区 & IP 库

每个组件，包含两部分：

1. `core` 包：组件的核心封装，拓展相关的功能。
2. `config` 包：组件的 Spring Boot 自动配置。



zjugis-module-xxx

该模块是 XXX 功能的 Module 模块，目前内置了 2 个模块。 后续增加功能后新增对应模块。

项目	说明	是否必须
zjugis-module-system	系统功能	√
zjugis-module-infra	基础设施	√

每个模块包含两个 Maven Module，分别是：

Maven Module	作用
zjugis-module-xxx-api	提供给其它模块的 API 定义
zjugis-module-xxx-biz	模块的功能的具体实现

例如说，`zjugis-module-infra` 想要访问 `zjugis-module-system` 的用户、部门等数据，需要引入 `zjugis-module-system-api` 子模块。示例如下：

```
<!-- 依赖服务 -->
<dependency>
    <groupId>com.zjugis.cloud</groupId>
    <artifactId>zjugis-module-system-api</artifactId>
    <version>${revision}</version>
</dependency>
<dependency>
    <groupId>com.zjugis.cloud</groupId>
    <artifactId>zjugis-module-infra-api</artifactId>
    <version>${revision}</version>
</dependency>
```

前端结构

基于 Vue3 + element-plus 实现的管理后台

```
.
├── .github # github workflows 相关
├── .husky # husky 配置
├── .vscode # vscode 配置
├── mock # 自定义 mock 数据及配置
├── public # 静态资源
├── src # 项目代码
|   ├── api # api接口管理
|   ├── assets # 静态资源
|   ├── components # 公用组件
|   ├── hooks # 常用hooks
|   ├── layout # 布局组件
|   ├── locales # 语言文件
|   ├── plugins # 外部插件
|   ├── router # 路由配置
|   ├── store # 状态管理
```

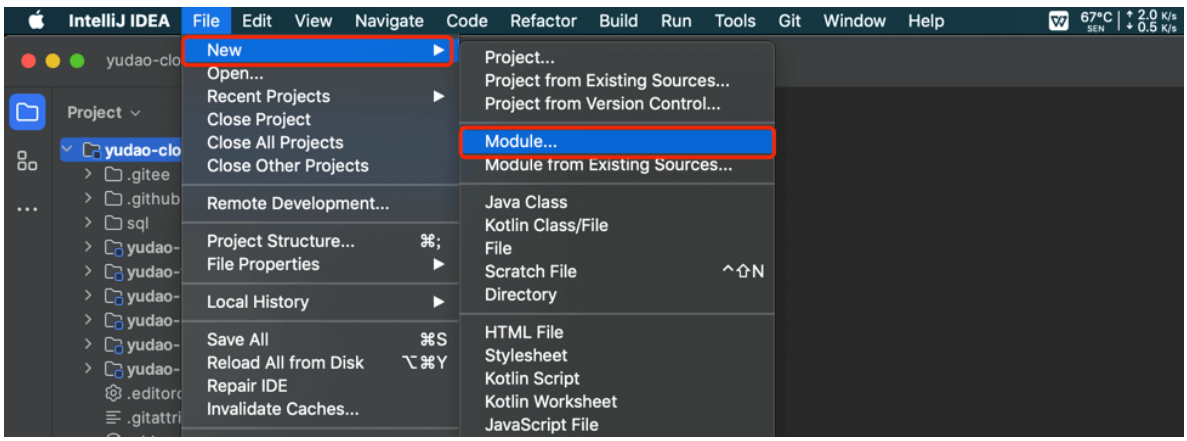
```
|    ├── styles # 全局样式
|    ├── utils # 全局工具类
|    ├── views # 路由页面
|    ├── App.vue # 入口vue文件
|    ├── main.ts # 主入口文件
|    └── permission.ts # 路由拦截
├── types # 全局类型
├── .env.base # 本地开发环境 环境变量配置
├── .env.dev # 打包到开发环境 环境变量配置
├── .env.gitee # 针对 gitee 的环境变量 可忽略
├── .env.pro # 打包到生产环境 环境变量配置
├── .env.test # 打包到测试环境 环境变量配置
├── .eslintignore # eslint 跳过检测配置
├── .eslintrc.js # eslint 配置
├── .gitignore # git 跳过配置
├── .prettierignore # prettier 跳过检测配置
├── .stylelintignore # stylelint 跳过检测配置
├── .versionrc 自动生成版本号及更新记录配置
├── CHANGELOG.md # 更新记录
├── commitlint.config.js # git commit 提交规范配置
├── index.html # 入口页面
├── package.json
├── .postcssrc.js # postcss 配置
├── prettier.config.js # prettier 配置
├── README.md # 英文 README
├── README.zh-CN.md # 中文 README
├── stylelint.config.js # stylelint 配置
├── tsconfig.json # typescript 配置
├── vite.config.ts # vite 配置
└── windi.config.ts # windicss 配置
```

后端手册

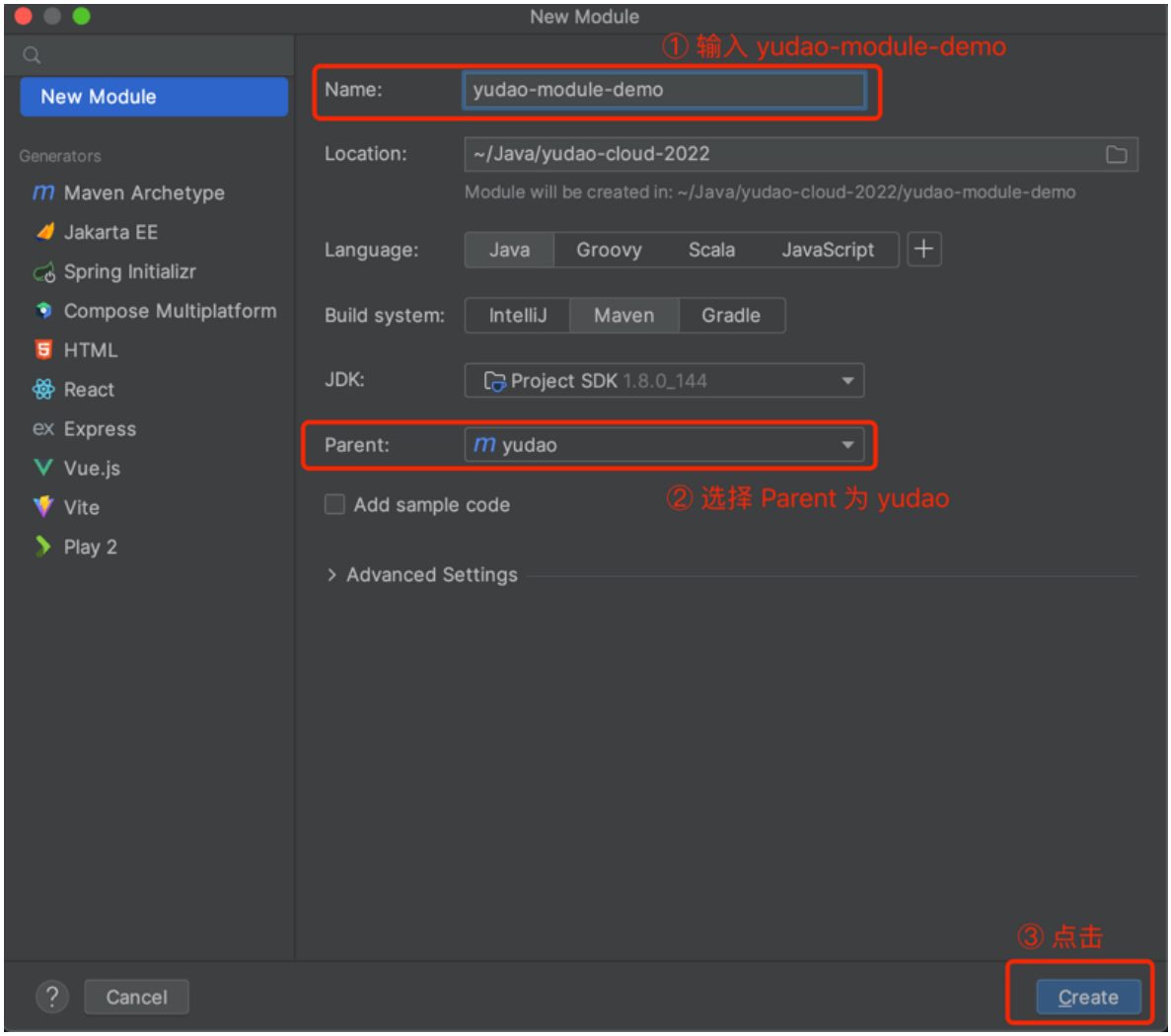
新建服务

新建demo模块

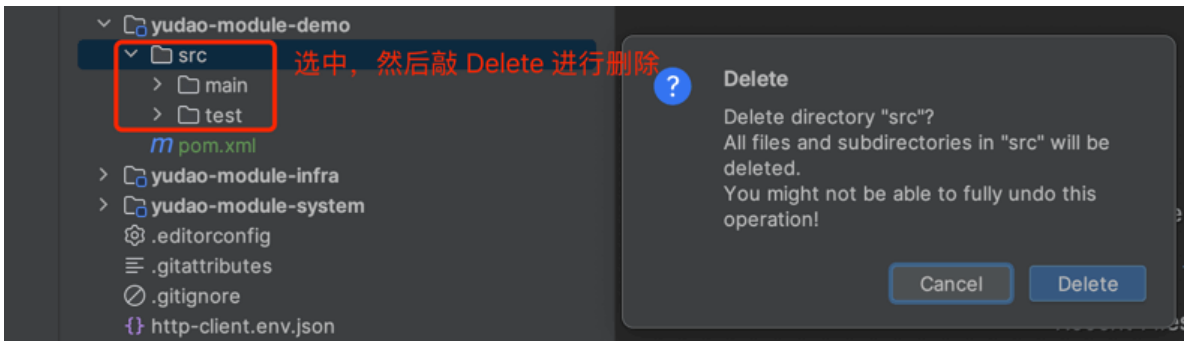
① 选择 File -> New -> Module 菜单，如下图所示：



② 选择 Maven 类型，选择父模块为 `yudao`，输入名字为 `yudao-module-demo`，并点击 Create 按钮，如下图所示：



③ 打开 `yudao-module-demo` 模块，删除 src 文件，如下图所示：



④ 打开 `yudao-module-demo` 模块的 `pom.xml` 文件，修改内容如下：

提示

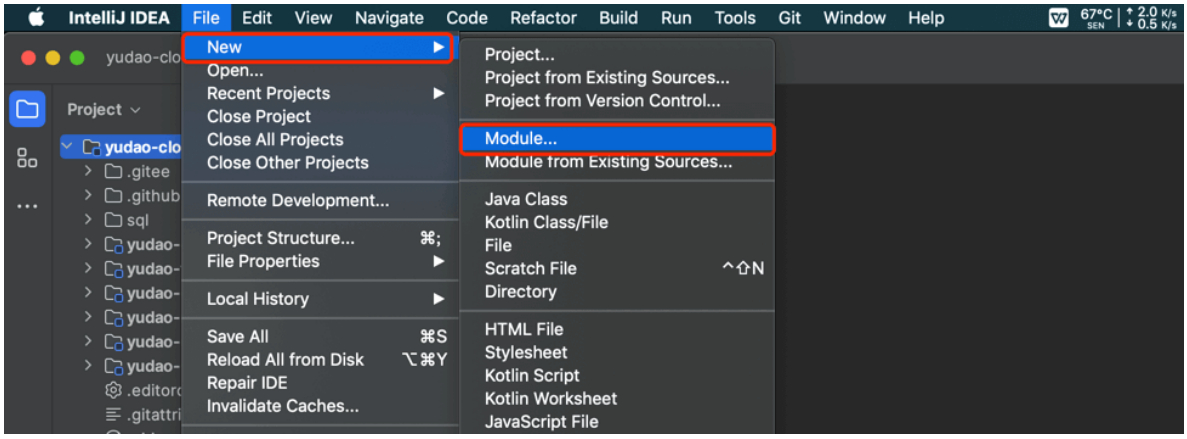
部分，只是注释，不需要写到 XML 中。

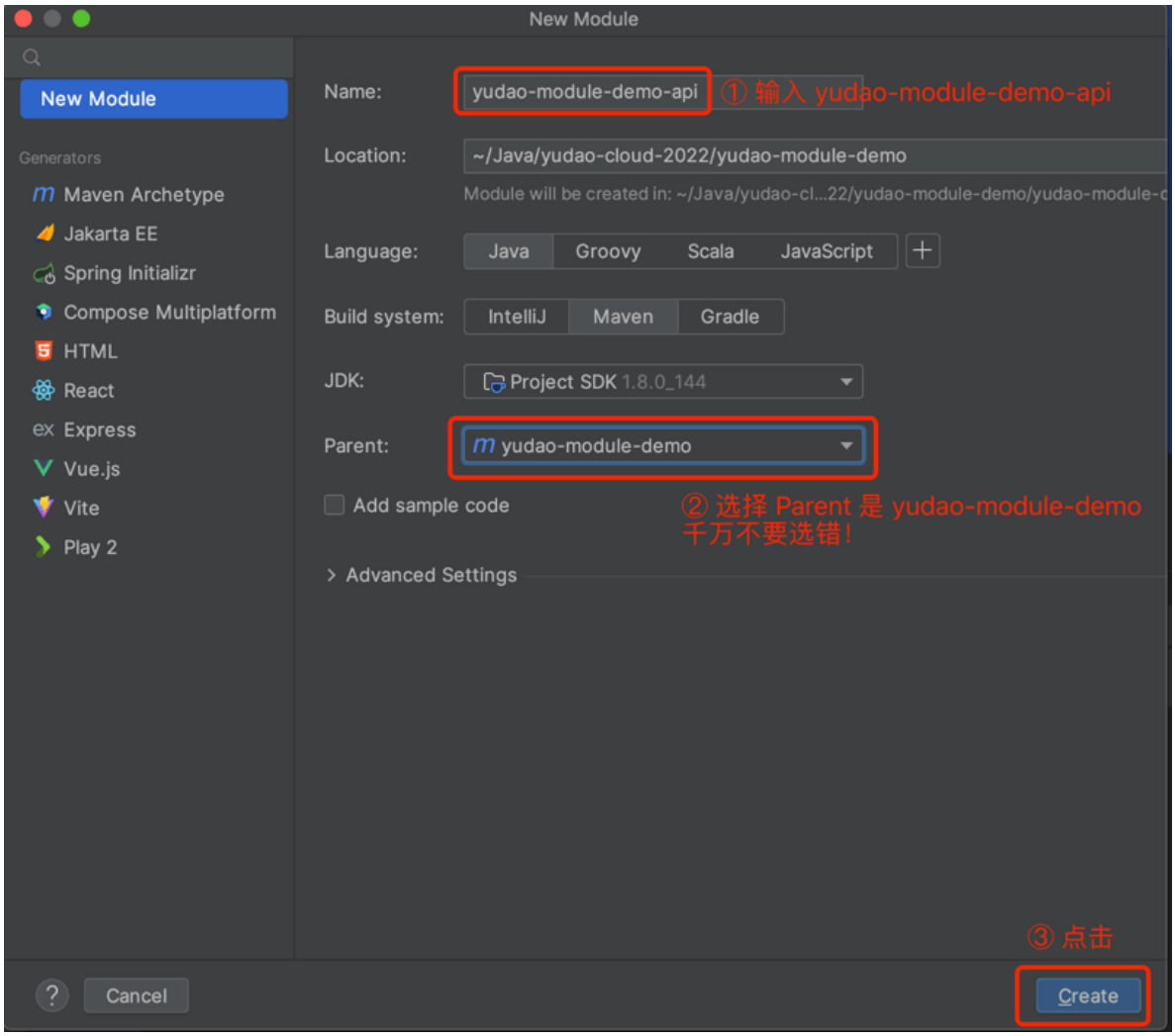
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>yudao</artifactId>
        <groupId>cn.iocoder.cloud</groupId>
        <version>${revision}</version> <!-- 1. 修改 version 为 ${revision} -->
    </parent>
    <modelVersion>4.0.0</modelVersion>
```

```
    <artifactId>yudao-module-demo</artifactId>
    <packaging>pom</packaging> <!-- 2. 新增 packaging 为 pom -->

    <name>${project.artifactId}</name> <!-- 3. 新增 name 为 ${project.artifactId}
-->
    <description> <!-- 4. 新增 description 为该模块的描述 -->
        demo 模块，主要实现 XXX、YYY、ZZZ 等功能。
    </description>

</project>
```

新建demo-api模块

① 新建 `yudao-module-demo-api` 子模块，整个过程和“新建 demo 模块”是一致的，如下图所示：

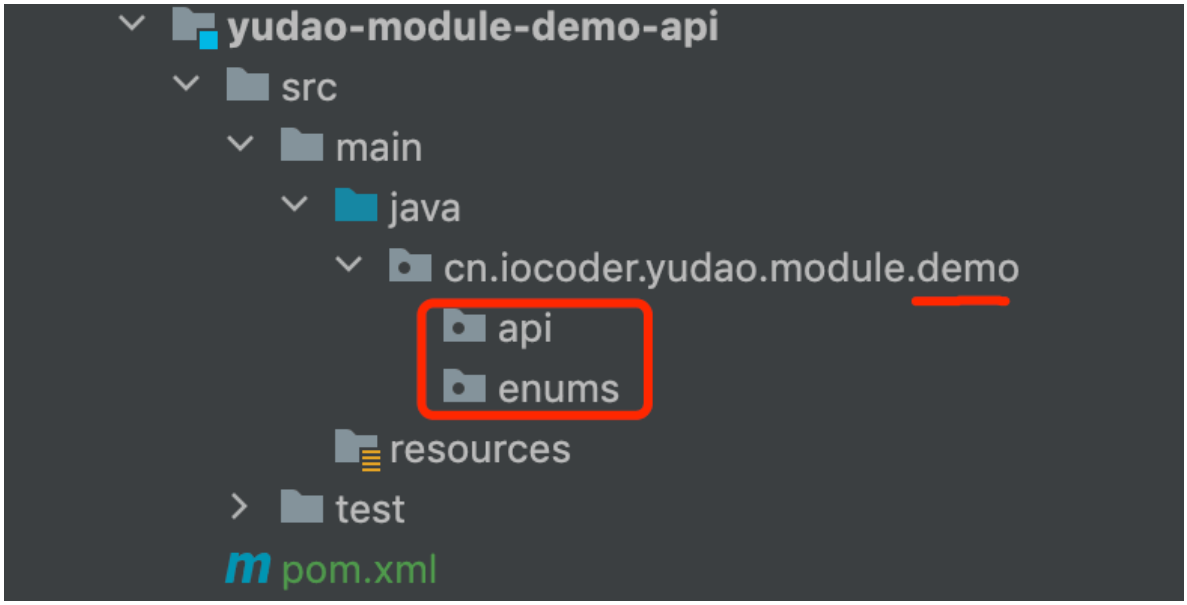




② 打开 `yudao-module-demo-api` 模块的 `pom.xml` 文件，修改内容如下：

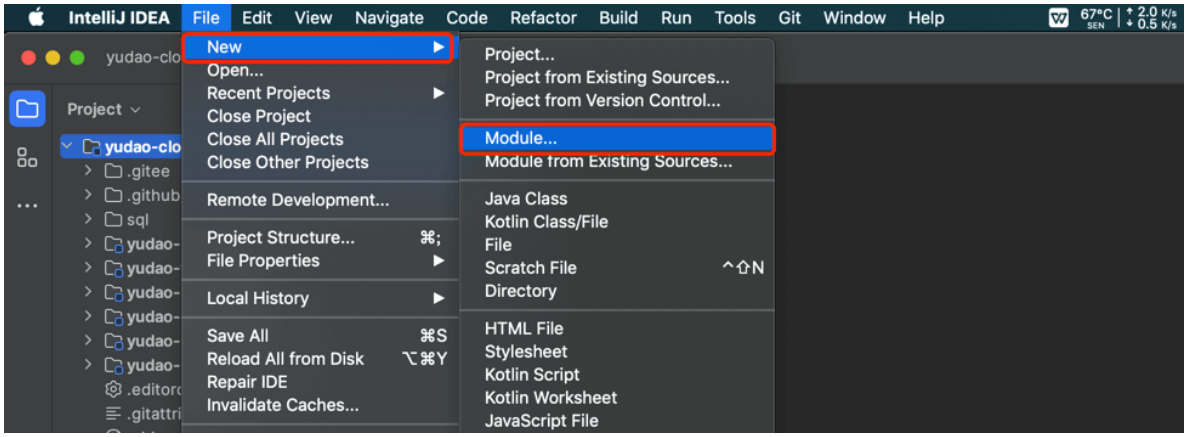
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>yudao-module-demo</artifactId>
        <groupId>cn.iocoder.cloud</groupId>
        <version>${revision}</version> <!-- 1. 修改 version 为 ${revision} -->
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>yudao-module-demo-api</artifactId>
    <packaging>jar</packaging> <!-- 2. 新增 packaging 为 jar -->

    <name>${project.artifactId}</name> <!-- 3. 新增 name 为 ${project.artifactId}
-->
    <description> <!-- 4. 新增 description 为该模块的描述 -->
        demo 模块 API，暴露给其它模块调用
    </description>

    <dependencies>  <!-- 5. 新增 yudao-common 依赖 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-common</artifactId>
        </dependency>
    </dependencies>

</project>
```

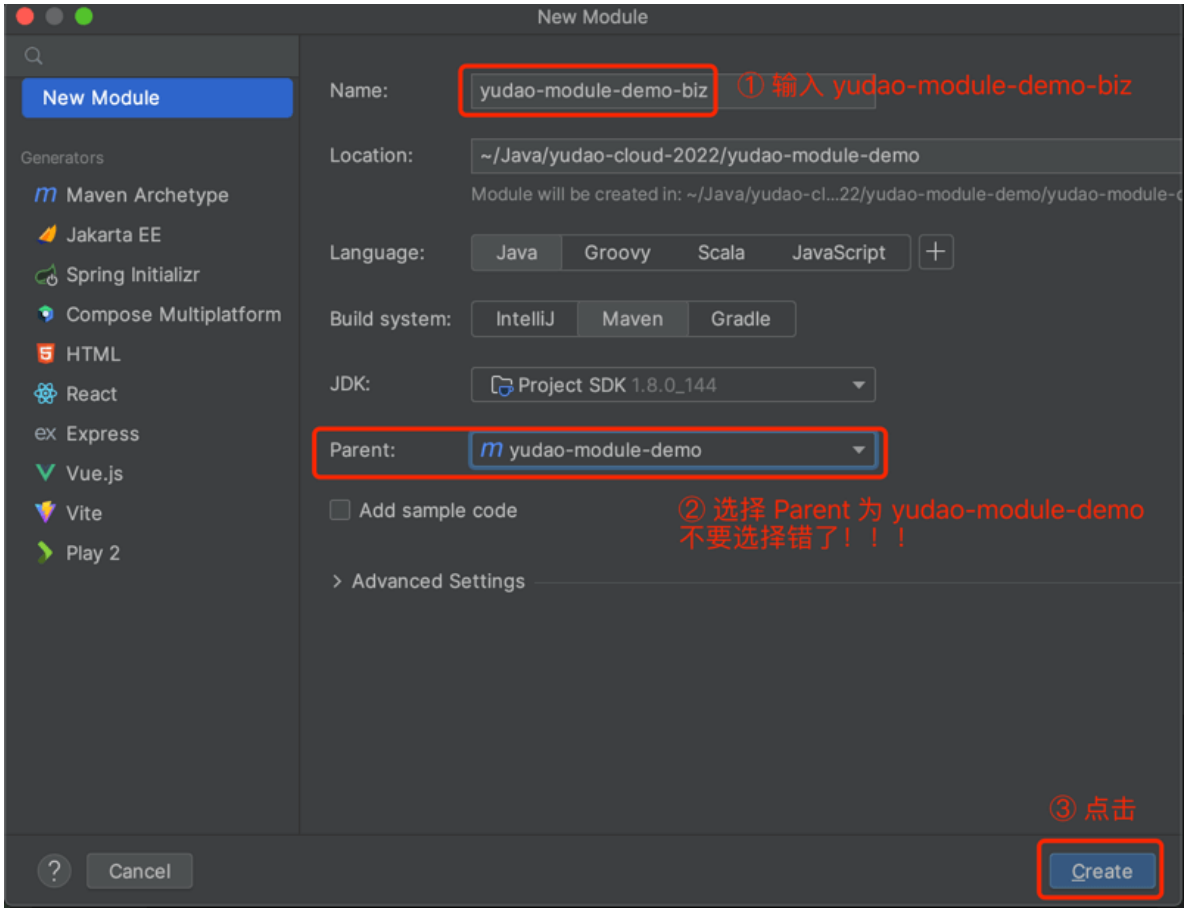
③ 【可选】新建 `cn.iocoder.yudao.module.demo` **基础**包，其中 `demo` 为模块名。之后，新建 `api` 和 `enums` 包。如下图所示：



新建demo-biz模块

① 新建 `yudao-module-demo-biz` 子模块，整个过程和“新建 demo 模块”也是一致的，如下图所示：





② 打开 `yudao-module-demo-biz` 模块的 `pom.xml` 文件，修改成内容如下：

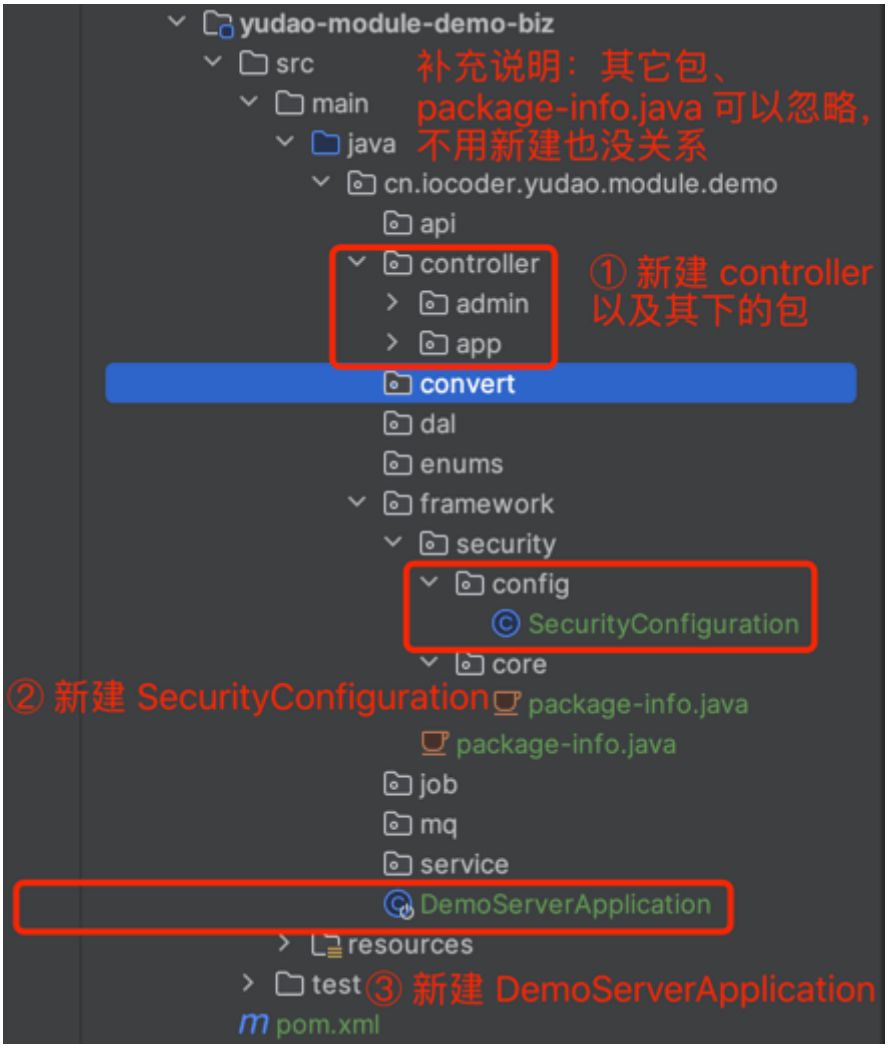
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>yudao-module-demo</artifactId>
        <groupId>cn.iocoder.cloud</groupId>
        <version>${revision}</version> <!-- 1. 修改 version 为 ${revision} -->
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <packaging>jar</packaging> <!-- 2. 新增 packaging 为 jar -->

    <artifactId>yudao-module-demo-biz</artifactId>

    <name>${project.artifactId}</name> <!-- 3. 新增 name 为 ${project.artifactId}
-->
```

```
    <description> <!-- 4. 新增 description 为该模块的描述 -->
        demo 模块，主要实现 XXX、YYY、ZZZ 等功能。
    </description>

    <dependencies>  <!-- 5. 新增依赖，这里引入的都是比较常用的业务组件、技术组件 -->
        <!-- Spring Cloud 基础 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-bootstrap</artifactId>
        </dependency>

        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-env</artifactId>
        </dependency>

        <!-- 依赖服务 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-module-system-api</artifactId>
            <version>${revision}</version>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-module-infra-api</artifactId>
            <version>${revision}</version>
        </dependency>

        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-module-demo-api</artifactId>
            <version>${revision}</version>
        </dependency>

        <!-- 业务组件 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-banner</artifactId>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-biz-operatelog</artifactId>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-biz-dict</artifactId>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-biz-data-
permission</artifactId>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-biz-tenant</artifactId>
        </dependency>
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
```

```
            <artifactId>yudao-spring-boot-starter-biz-error-code</artifactId>
        </dependency>

        <!-- Web 相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-security</artifactId>
        </dependency>

        <!-- DB 相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-mybatis</artifactId>
        </dependency>

        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-redis</artifactId>
        </dependency>

        <!-- RPC 远程调用相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-rpc</artifactId>
        </dependency>

        <!-- Registry 注册中心相关 -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
        </dependency>

        <!-- Config 配置中心相关 -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        </dependency>

        <!-- Job 定时任务相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-job</artifactId>
        </dependency>

        <!-- 消息队列相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-mq</artifactId>
        </dependency>

        <!-- Test 测试相关 -->
        <dependency>
```

```
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-test</artifactId>
        </dependency>

        <!-- 工具类相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-excel</artifactId>
        </dependency>

        <!-- 监控相关 -->
        <dependency>
            <groupId>cn.iocoder.cloud</groupId>
            <artifactId>yudao-spring-boot-starter-monitor</artifactId>
        </dependency>
    </dependencies>

    <build>
        <!-- 设置构建的 jar 包名 -->
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <!-- 打包 -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring.boot.version}</version>
                <configuration>
                    <fork>true</fork>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal> <!-- 将引入的 jar 打入其中 -->
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

③【必选】新建 `cn.iocoder.yudao.module.demo` **基础**包，其中 `demo` 为模块名。之后，新建 `controller.admin` 和 `controller.user` 等包。如下图所示：



其中 SecurityConfiguration 的 Java 代码如下：

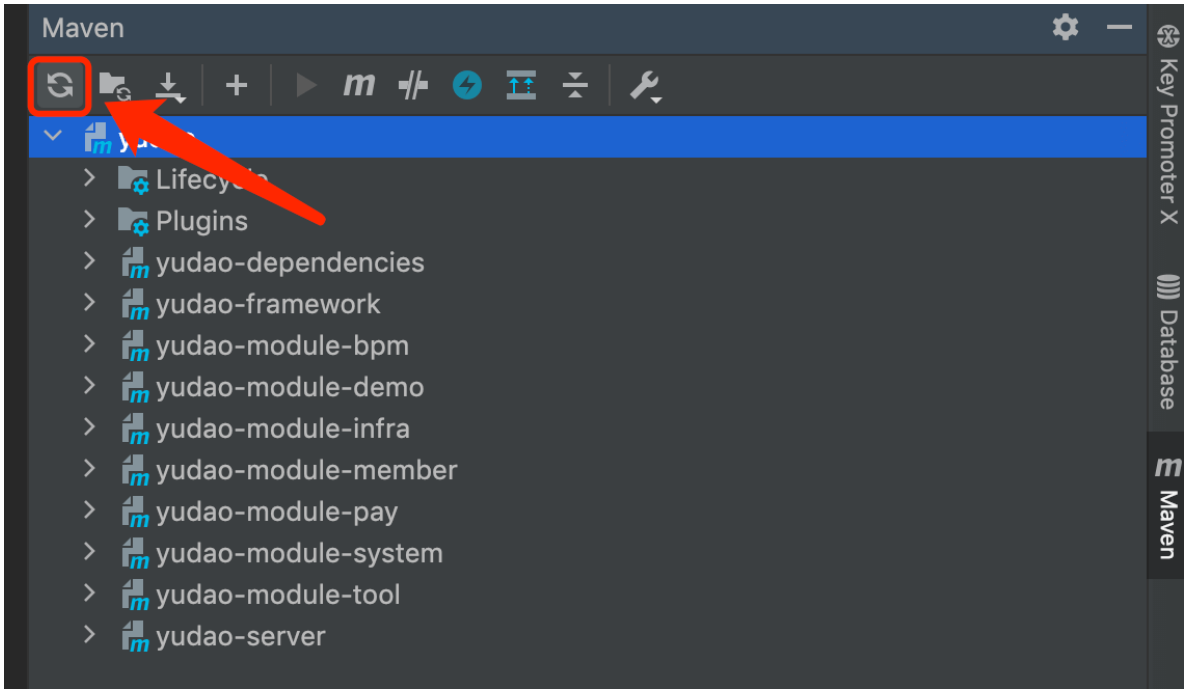
```
package cn.iocoder.yudao.module.demo.framework.security.config;

import cn.iocoder.yudao.framework.security.config.AuthorizeRequestsCustomizer;
import cn.iocoder.yudao.module.system.enums.ApiConstants;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configurers.ExpressionUrlAuth
orizationConfigurer;

/**
 * Demo 模块的 Security 配置
 */
@Configuration(proxyBeanMethods = false)
public class SecurityConfiguration {

    @Bean
    public AuthorizeRequestsCustomizer authorizeRequestsCustomizer() {
        return new AuthorizeRequestsCustomizer() {

            @Override
            public void
customize(ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionIntercept
UrlRegistry registry) {
                // Swagger 接口文档
```

```
                registry.antMatchers("/v3/api-docs/**").permitAll() // 元数据
                        .antMatchers("/swagger-ui.html").permitAll(); // Swagger
UI
                // Druid 监控
                registry.antMatchers("/druid/**").anonymous();
                // Spring Boot Actuator 的安全配置
                registry.antMatchers("/actuator").anonymous()
                        .antMatchers("/actuator/**").anonymous();
                // RPC 服务的安全配置
                registry.antMatchers(ApiConstants.PREFIX + "/**").permitAll();
            }

        };
    }

}
```

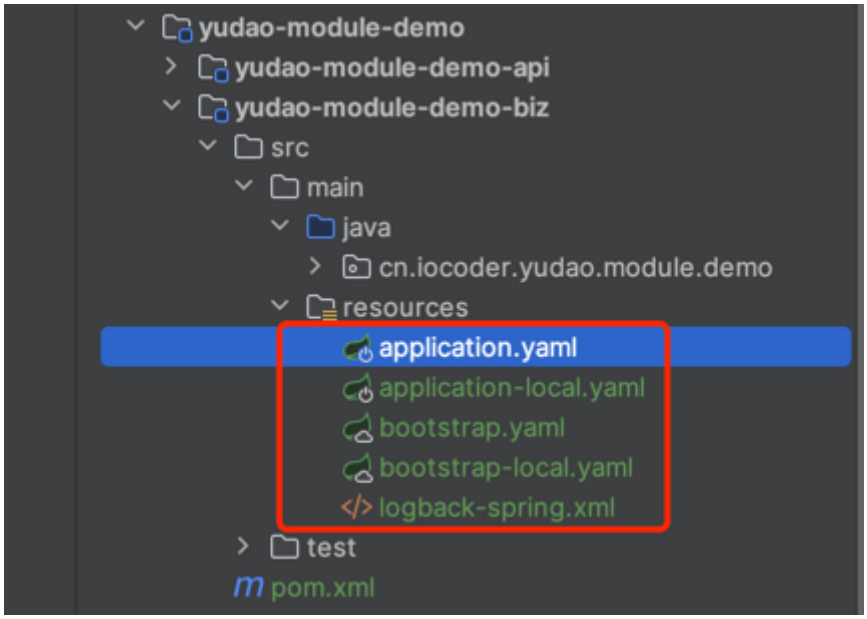
其中 DemoServerApplication 的 Java 代码如下：

```
package cn.iocoder.yudao.module.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * 项目的启动类
 *
 * @author 芋道源码
 */
@SpringBootApplication
public class DemoServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoServerApplication.class, args);
    }

}
```

④ 打开 Maven 菜单，点击刷新按钮，让引入的 Maven 依赖生效。如下图所示：



⑤ 在 `resources` 目录下，新建配置文件。如下图所示：



其中 `application.yml` 的配置如下：

```yaml
spring:
  main:
    allow-circular-references: true # 允许循环依赖，因为项目是三层架构，无法避免这个情况。
    allow-bean-definition-overriding: true # 允许 Bean 覆盖，例如说 Feign 等会存在重复定义的服务

  # Servlet 配置
  servlet:
    # 文件上传相关配置项
    multipart:
      max-file-size: 16MB # 单个文件大小
      max-request-size: 32MB # 设置总上传的文件大小
  mvc:
    pathmatch:
```

```
      matching-strategy: ANT_PATH_MATCHER # 解决 SpringFox 与 SpringBoot 2.6.x 不兼容的问题，参见 SpringFoxHandlerProviderBeanPostProcessor 类

  # Jackson 配置项
  jackson:
    serialization:
      write-dates-as-timestamps: true # 设置 LocalDateTime 的格式，使用时间戳
      write-date-timestamps-as-nanoseconds: false # 设置不使用 nanoseconds 的格式。例如说 1611460870.401，而是直接 1611460870401
      write-durations-as-timestamps: true # 设置 Duration 的格式，使用时间戳
      fail-on-empty-beans: false # 允许序列化无属性的 Bean

  # Cache 配置项
  cache:
    type: REDIS
    redis:
      time-to-live: 1h # 设置过期时间为 1 小时

--- ################### 接口文档配置 ###################

springdoc:
  api-docs:
    enabled: true # 1. 是否开启 Swagger 接文档的元数据
    path: /v3/api-docs
  swagger-ui:
    enabled: true # 2.1 是否开启 Swagger 文档的官方 UI 界面
    path: /swagger-ui.html
  default-flat-param-object: true # 参见 https://doc.xiaominfo.com/docs/faq/v4/knife4j-parameterobject-flat-param 文档

knife4j:
  enable: true # 2.2 是否开启 Swagger 文档的 Knife4j UI 界面
  setting:
    language: zh_cn

# MyBatis Plus 的配置项
mybatis-plus:
  configuration:
    map-underscore-to-camel-case: true # 虽然默认为 true ，但是还是显示去指定下。
  global-config:
    db-config:
      id-type: NONE # “智能”模式，基于 IdTypeEnvironmentPostProcessor + 数据源的类型，自动适配成 AUTO、INPUT 模式。
      #      id-type: AUTO # 自增 ID，适合 MySQL 等直接自增的数据库
      #      id-type: INPUT # 用户输入 ID，适合 Oracle、PostgreSQL、Kingbase、DB2、H2 数据库
      #      id-type: ASSIGN_ID # 分配 ID，默认使用雪花算法。注意，Oracle、PostgreSQL、Kingbase、DB2、H2 数据库时，需要去除实体类上的 @KeySequence 注解
      logic-delete-value: 1 # 逻辑已删除值(默认为 1)
      logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
    banner: false # 关闭控制台的 Banner 打印
  type-aliases-package: ${yudao.info.base-package}.module.*.dal.dataobject
  encryptor:
    password: XDV71a+xqStEA3WH # 加解密的秘钥，可使用 https://www.imaegoo.com/2020/aes-key-generator/ 网站生成

mybatis-plus-join:
  banner: false # 关闭控制台的 Banner 打印
```

```
--- ###################### RPC 远程调用相关配置 ######################

--- ###################### MQ 消息队列相关配置 ######################

--- ###################### 定时任务相关配置 ######################

xxl:
  job:
    executor:
      appname: ${spring.application.name} # 执行器 AppName
      logpath: ${user.home}/logs/xxl-job/${spring.application.name} # 执行器运行日志文件存储磁盘路径
    accessToken: default_token # 执行器通讯TOKEN

--- ###################### 芋道相关配置 ######################

yudao:
  info:
    version: 1.0.0
    base-package: cn.iocoder.yudao.module.demo
  web:
    admin-ui:
      url: http://dashboard.yudao.iocoder.cn # Admin 管理后台 UI 的地址
  swagger:
    title: 管理后台
    description: 提供管理员管理的所有功能
    version: ${yudao.info.version}
    base-package: ${yudao.info.base-package}
  tenant: # 多租户相关配置项
    enable: true

debug: false
```

- `yudao.info.version.base-package` 配置项：可以改成你的项目的基准包名。

其中 `application-local.yml` 的配置如下：

```
--- ###################### 数据库相关配置 ######################
spring:
  # 数据源配置项
  autoconfigure:
    exclude:
      - com.alibaba.druid.spring.boot.autoconfigure.DruidDataSourceAutoConfigure # 排除 Druid 的自动配置，使用 dynamic-datasource-spring-boot-starter 配置多数据源
      - de.codecentric.boot.admin.client.config.SpringBootAdminClientAutoConfiguration # 禁用 Spring Boot Admin 的 Client 的自动配置
  datasource:
    druid: # Druid 【监控】相关的全局配置
      web-stat-filter:
        enabled: true
      stat-view-servlet:
        enabled: true
        allow: # 设置白名单，不填则允许所有访问
        url-pattern: /druid/*
        login-username: # 控制台管理用户名和密码
```

```
        login-password:
      filter:
        stat:
          enabled: true
          log-slow-sql: true # 慢 SQL 记录
          slow-sql-millis: 100
          merge-sql: true
        wall:
          config:
            multi-statement-allow: true
    dynamic: # 多数据源配置
      druid: # Druid 【连接池】相关的全局配置
        initial-size: 1 # 初始连接数
        min-idle: 1 # 最小连接池数量
        max-active: 20 # 最大连接池数量
        max-wait: 600000 # 配置获取连接等待超时的时间，单位：毫秒
        time-between-eviction-runs-millis: 60000 # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位：毫秒
        min-evictable-idle-time-millis: 300000 # 配置一个连接在池中最小生存的时间，单位：毫秒
        max-evictable-idle-time-millis: 900000 # 配置一个连接在池中最大生存的时间，单位：毫秒
        validation-query: SELECT 1 FROM DUAL # 配置检测连接是否有效
        test-while-idle: true
        test-on-borrow: false
        test-on-return: false
      primary: master
      datasource:
        master:
          name: ruoyi-vue-pro
          url: jdbc:mysql://127.0.0.1:3306/${spring.datasource.dynamic.datasource.master.name}?allowMultiQueries=true&useUnicode=true&useSSL=false&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai&autoReconnect=true&nullCatalogMeansCurrent=true # MySQL Connector/J 8.X 连接的示例
#          url: jdbc:mysql://127.0.0.1:3306/${spring.datasource.dynamic.datasource.master.name}?useSSL=false&allowPublicKeyRetrieval=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=CTT # MySQL Connector/J 5.X 连接的示例
#          url: jdbc:postgresql://127.0.0.1:5432/${spring.datasource.dynamic.datasource.slave.name} # PostgreSQL 连接的示例
#          url: jdbc:oracle:thin:@127.0.0.1:1521:xe # Oracle 连接的示例
#          url: jdbc:sqlserver://127.0.0.1:1433;DatabaseName=${spring.datasource.dynamic.datasource.master.name} # SQLServer 连接的示例
          username: root
          password: 123456
#          username: sa
#          password: JSm:g(*%lU4ZAkzO6cd52KqT3)i1?H7W
        slave: # 模拟从库，可根据自己需要修改
          name: ruoyi-vue-pro
          url: jdbc:mysql://127.0.0.1:3306/${spring.datasource.dynamic.datasource.slave.name}?allowMultiQueries=true&useUnicode=true&useSSL=false&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai&autoReconnect=true&nullCatalogMeansCurrent=true # MySQL Connector/J 8.X 连接的示例
```

```yaml
#          url:
jdbc:mysql://127.0.0.1:3306/${spring.datasource.dynamic.datasource.slave.name}?
useSSL=false&allowPublicKeyRetrieval=true&useUnicode=true&characterEncoding=UTF-
8&serverTimezone=CTT # MySQL Connector/J 5.X 连接的示例
#          url:
jdbc:postgresql://127.0.0.1:5432/${spring.datasource.dynamic.datasource.slave.na
me} # PostgreSQL 连接的示例
#          url: jdbc:oracle:thin:@127.0.0.1:1521:xe # Oracle 连接的示例
#          url:
jdbc:sqlserver://127.0.0.1:1433;DatabaseName=${spring.datasource.dynamic.datasou
rce.slave.name} # SQLServer 连接的示例
          username: root
          password: 123456
#          username: sa
#          password: JSm:g(*%lU4ZAkzO6cd52KqT3)i1?H7W

  # Redis 配置。Redisson 默认的配置足够使用，一般不需要进行调优
  redis:
    host: 127.0.0.1 # 地址
    port: 6379 # 端口
    database: 0 # 数据库索引
#    password: 123456 # 密码，建议生产环境开启

--- ################### MQ 消息队列相关配置 ####################
spring:
  cloud:
    stream:
      rocketmq:
        # RocketMQ Binder 配置项，对应 RocketMQBinderConfigurationProperties 类
        binder:
          name-server: 127.0.0.1:9876 # RocketMQ Namesrv 地址

--- ################### 定时任务相关配置 ####################

xxl:
  job:
    admin:
      addresses: http://127.0.0.1:9090/xxl-job-admin # 调度中心部署跟地址

--- ################### 服务保障相关配置 ####################

# Lock4j 配置项
lock4j:
  acquire-timeout: 3000 # 获取分布式锁超时时间，默认为 3000 毫秒
  expire: 30000 # 分布式锁的超时时间，默认为 30 毫秒

--- ################### 监控相关配置 ####################

# Actuator 监控端点的配置项
management:
  endpoints:
    web:
      base-path: /actuator # Actuator 提供的 API 接口的根目录。默认为 /actuator
      exposure:
        include: '*' # 需要开放的端点。默认值只打开 health 和 info 两个端点。通过设置 *
，可以开放所有端点。

# Spring Boot Admin 配置项
```

```yaml
spring:
  boot:
    admin:
      # Spring Boot Admin Client 客户端的相关配置
      client:
        instance:
          service-host-type: IP # 注册实例时，优先使用 IP [IP, HOST_NAME,
CANONICAL_HOST_NAME]

# 日志文件配置
logging:
  level:
    # 配置自己写的 MyBatis Mapper 打印日志
    cn.iocoder.yudao.module.demo.dal.mysql: debug

--- ################### 芋道相关配置 ###################

# 芋道配置项，设置当前项目所有自定义的配置
yudao:
  env: # 多环境的配置项
    tag: ${HOSTNAME}
  security:
    mock-enable: true
  xss:
    enable: false
    exclude-urls: # 如下两个 url，仅仅是为了演示，去掉配置也没关系
      - ${spring.boot.admin.context-path}/** # 不处理 Spring Boot Admin 的请求
      - ${management.endpoints.web.base-path}/** # 不处理 Actuator 的请求
  access-log: # 访问日志的配置项
    enable: false
  error-code: # 错误码相关配置项
    enable: false
  demo: false # 关闭演示模式
```

- `logging.level.cn.iocoder.yudao.module.demo.dal.mysql` 配置项：可以改成你的项目的基准包名。

其中 `bootstrap.yml` 的配置如下：

```yaml
spring:
  application:
    name: demo-server

  profiles:
    active: local

server:
  port: 48099

# 日志文件配置。注意，如果 logging.file.name 不放在 bootstrap.yaml 配置文件，而是放在
application.yaml 中，会导致出现 LOG_FILE_IS_UNDEFINED 文件
logging:
  file:
    name: ${user.home}/logs/${spring.application.name}.log # 日志文件名，全路径
```

- `spring.application.name` 配置项：可以改成你想要的服务名。

- `server.port` 配置项：可以改成你想要的端口号。

其中 `bootstrap-local.yml` 的配置如下：

```
--- ################### 注册中心相关配置 ###################

spring:
  cloud:
    nacos:
      server-addr: 127.0.0.1:8848
      discovery:
        namespace: dev # 命名空间。这里使用 dev 开发环境
        metadata:
          version: 1.0.0 # 服务实例的版本号，可用于灰度发布

--- ################### 配置中心相关配置 ###################

spring:
  cloud:
    nacos:
      # Nacos Config 配置项，对应 NacosConfigProperties 配置属性类
      config:
        server-addr: 127.0.0.1:8848 # Nacos 服务器地址
        namespace: dev # 命名空间。这里使用 dev 开发环境
        group: DEFAULT_GROUP # 使用的 Nacos 配置分组，默认为 DEFAULT_GROUP
        name: # 使用的 Nacos 配置集的 dataId，默认为 spring.application.name
        file-extension: yaml # 使用的 Nacos 配置集的 dataId 的文件拓展名，同时也是
Nacos 配置集的配置格式，默认为 properties
```

其中 `logback-spring.xml` 的配置如下：

```
<configuration>
    <!-- 引用 Spring Boot 的 logback 基础配置 -->
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />
    <!-- 变量 yudao.info.base-package，基础业务包 -->
    <springProperty scope="context" name="yudao.info.base-package"
source="yudao.info.base-package"/>
    <!-- 格式化输出：%d 表示日期，%X{tid} SkWalking 链路追踪编号，%thread 表示线程名，
%-5level：级别从左显示 5 个字符宽度，%msg：日志消息，%n是换行符 -->
    <property name="PATTERN_DEFAULT" value="%d{${LOG_DATEFORMAT_PATTERN:-yyyy-
MM-dd HH:mm:ss.SSS}} ${LOG_LEVEL_PATTERN:-%5p} ${PID:- } --- [%thread] [%tid]
%-40.40logger{39} : %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>

    <!-- 控制台 Appender -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackL
ayout">
                <pattern>${PATTERN_DEFAULT}</pattern>
            </layout>
        </encoder>
    </appender>

    <!-- 文件 Appender -->
    <!-- 参考 Spring Boot 的 file-appender.xml 编写 -->
```

```
    <appender name="FILE"
 class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackL
ayout">
                <pattern>${PATTERN_DEFAULT}</pattern>
            </layout>
        </encoder>
        <!-- 日志文件名 -->
        <file>${LOG_FILE}</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <!-- 滚动后的日志文件名 -->

 <fileNamePattern>${LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN:-${LOG_FILE}.%d{yyyy
-MM-dd}.%i.gz}</fileNamePattern>
            <!-- 启动服务时，是否清理历史日志，一般不建议清理 -->

 <cleanHistoryOnStart>${LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START:-false}
</cleanHistoryOnStart>
            <!-- 日志文件，到达多少容量，进行滚动 -->
            <maxFileSize>${LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE:-10MB}
</maxFileSize>
            <!-- 日志文件的总大小，0 表示不限制 -->
            <totalSizeCap>${LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP:-0}
</totalSizeCap>
            <!-- 日志文件的保留天数 -->
            <maxHistory>${LOGBACK_ROLLINGPOLICY_MAX_HISTORY:-30}</maxHistory>
        </rollingPolicy>
    </appender>
    <!-- 异步写入日志，提升性能 -->
    <appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
        <!-- 不丢失日志。默认的，如果队列的 80% 已满,则会丢弃 TRACT、DEBUG、INFO 级别的日
志 -->
        <discardingThreshold>0</discardingThreshold>
        <!-- 更改默认的队列的深度，该值会影响性能。默认值为 256 -->
        <queueSize>256</queueSize>
        <appender-ref ref="FILE"/>
    </appender>

    <!-- SkyWalking GRPC 日志收集，实现日志中心。注意：SkyWalking 8.4.0 版本开始支持 --
>
    <appender name="GRPC"
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.log.GRPCLogClientAppen
der">
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout
class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackL
ayout">
                <pattern>${PATTERN_DEFAULT}</pattern>
            </layout>
        </encoder>
    </appender>

    <!-- 本地环境 -->
    <springProfile name="local">
        <root level="INFO">
```

```
            <appender-ref ref="STDOUT"/>
            <appender-ref ref="GRPC"/> <!-- 本地环境下，如果不想接入 SkyWalking 日志服务，可以注释掉本行 -->
            <appender-ref ref="ASYNC"/>  <!-- 本地环境下，如果不想打印日志，可以注释掉本行 -->
        </root>
    </springProfile>
    <!-- 其它环境 -->
    <springProfile name="dev,test,stage,prod,default">
        <root level="INFO">
            <appender-ref ref="STDOUT"/>
            <appender-ref ref="ASYNC"/>
            <appender-ref ref="GRPC"/>
        </root>
    </springProfile>

</configuration>
```

新建Restful API 接口

① 在 `controller.admin` 包，新建一个 DemoTestController 类，并新建一个 `/demo/test/get` 接口。代码如下：

```
package cn.iocoder.yudao.module.demo.controller.admin;

import cn.iocoder.yudao.framework.common.pojo.CommonResult;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import static cn.iocoder.yudao.framework.common.pojo.CommonResult.success;

@Tag(name = "管理后台 - Test")
@RestController
@RequestMapping("/demo/test")
@Validated
public class DemoTestController {

    @GetMapping("/get")
    @Operation(summary = "获取 test 信息")
    public CommonResult<String> get() {
        return success("true");
    }

}
```

注意，`/demo` 是该模块所有 RESTful API 的基础路径，`/test` 是 Test 功能的基础路径。

① 在 `controller.app` 包，新建一个 AppDemoTestController 类，并新建一个 `/demo/test/get` 接口。代码如下：

```
package cn.iocoder.yudao.module.demo.controller.app;
```

```
import cn.iocoder.yudao.framework.common.pojo.CommonResult;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import static cn.iocoder.yudao.framework.common.pojo.CommonResult.success;

@Tag(name = "用户 App - Test")
@RestController
@RequestMapping("/demo/test")
@Validated
public class AppDemoTestController {

    @GetMapping("/get")
    @Operation(summary = "获取 test 信息")
    public CommonResult<String> get() {
        return success("true");
    }

}
```

在 Controller 的命名上，额外增加 **App** 作为前缀，一方面区分是管理后台还是用户 App 的 Controller，另一方面避免 Spring Bean 的名字冲突。

可能你会奇怪，这里我们定义了两个 `/demo/test/get` 接口，会不会存在重复导致冲突呢？答案，当然是并不会。原因是：

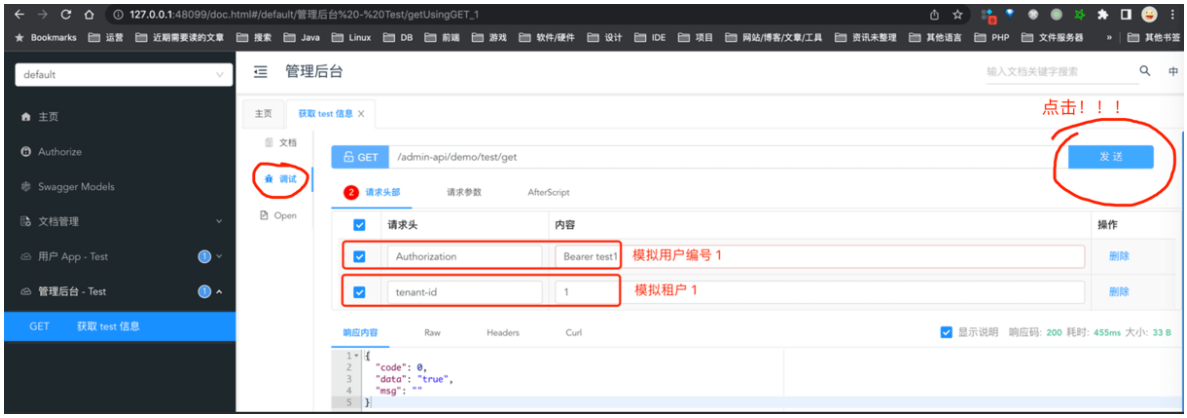
- `controller.admin` 包下的接口，默认会增加 `/admin-api`，即最终的访问地址是 `/admin-api/demo/test/get`
- `controller.app` 包下的接口，默认会增加 `/app-api`，即最终的访问地址是 `/app-api/demo/test/get`

启动demo服务

① 运行 SystemServerApplication 类，将 `system` 服务启动。运行 InfraServerApplication 类，将 `infra` 服务启动。

② 运行 DemoServerApplication 类，将新建的 `demo` 服务进行启动。启动完成后，使用浏览器打开 http://127.0.0.1:48099/doc.html (opens new window)地址，进入该服务的 Swagger 接口文档。

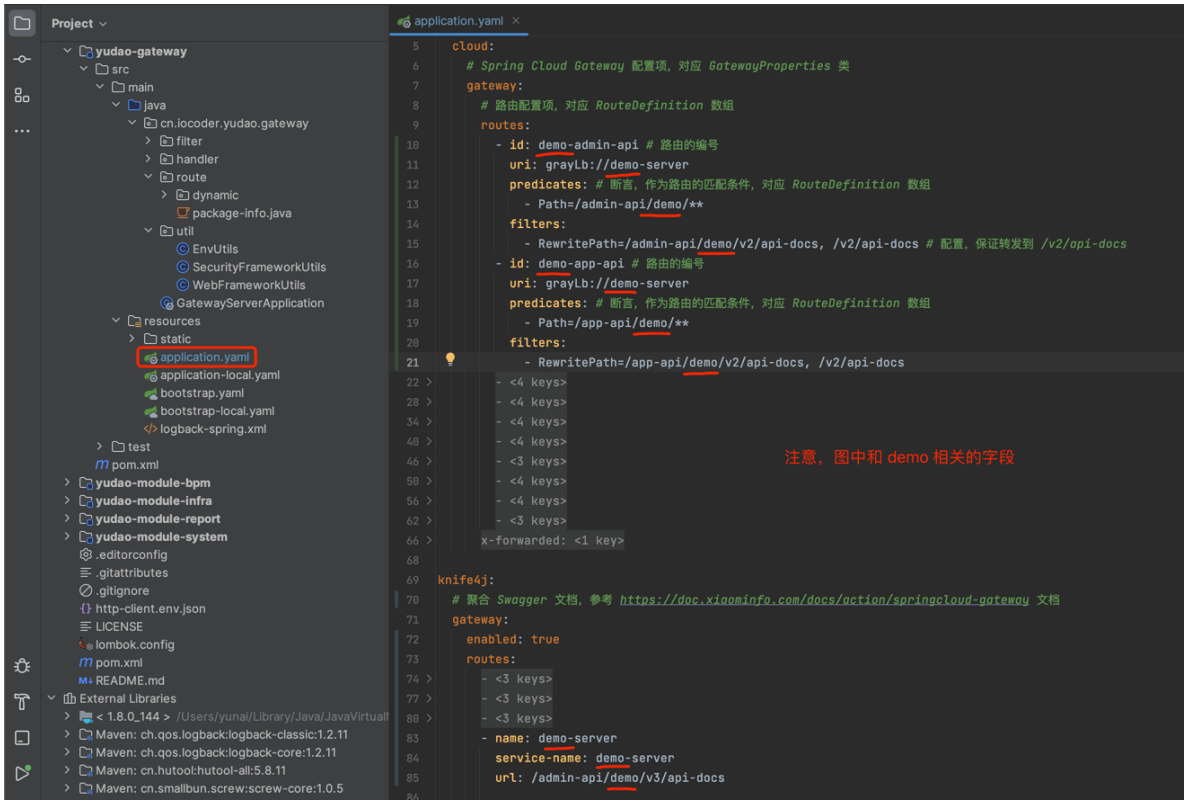
③ 打开“管理后台 - Test”接口，进行 `/admin-api/demo/test/get` 接口的调试，如下图所示：



④ 打开“用户 App - Test”接口，进行 `/app-api/demo/test/get` 接口的调试，如下图所示：

网关配置

① 打开 `yudao-gateway` 网关项目的 `application.yml` 配置文件，增加 `demo` 服务的路由配置。代码如下：

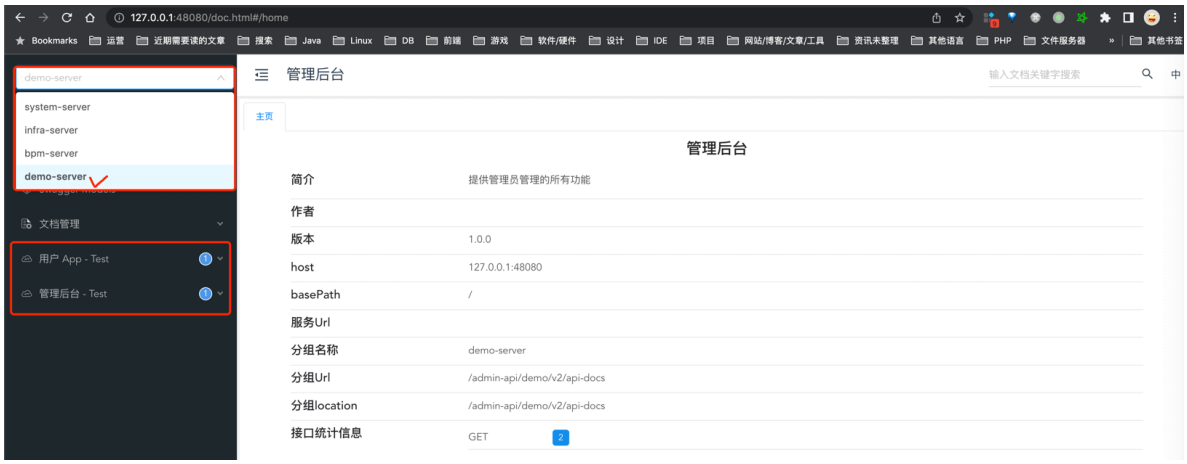


友情提示：图中的 /v2/ 都改成 /v3/，或者以下面的文字为准！！！

```
        - id: demo-admin-api # 路由的编号
          uri: grayLb://demo-server
          predicates: # 断言，作为路由的匹配条件，对应 RouteDefinition 数组
            - Path=/admin-api/demo/**
          filters:
            - RewritePath=/admin-api/demo/v3/api-docs, /v3/api-docs # 配置，保证转发到 /v2/api-docs
        - id: demo-app-api # 路由的编号
          uri: grayLb://demo-server
          predicates: # 断言，作为路由的匹配条件，对应 RouteDefinition 数组
            - Path=/app-api/demo/**
          filters:
            - RewritePath=/app-api/demo/v3/api-docs, /v3/api-docs
      - name: demo-server
        service-name: demo-server
        url: /admin-api/demo/v3/api-docs
```

② 运行 GatewayServerApplication 类，将 `gateway` 网关服务启动。

③ 使用浏览器打开 http://127.0.0.1:48080/doc.html (opens new window)地址，进入网关的 Swagger 接口文档。然后，选择 `demo-server` 服务，即可进行 `/admin-api/demo/test/get` 和 `/app-api/demo/test/get` 接口的调试，如下图所示：



代码生成

大部分项目里，其实有很多代码是重复的，几乎每个模块都有 CRUD 增删改查的功能，而这些功能的实现代码往往是大同小异的。如果这些功能都要自己去手写，非常无聊枯燥，浪费时间且效率很低，还可能会写错。

所以这种重复性的代码，项目提供了 codegen (opens new window)代码生成器，只需要在数据库中设计好表结构，就可以一键生成前后端代码 + 单元测试 + Swagger 接口文档 + Validator 参数校验。

下面，我们使用代码生成器，在 `zjugis-module-system` 模块中，开发一个**【用户组】**的功能。

1. 数据库表结构设计

设计用户组的数据库表名为 `system_group`，其建表语句如下：

```
CREATE TABLE `system_group` (
    `id` bigint NOT NULL AUTO_INCREMENT COMMENT '编号',
    `name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '名字',
    `description` varchar(512) COLLATE utf8mb4_unicode_ci DEFAULT NULL COMMENT
'描述',
    `status` tinyint NOT NULL COMMENT '状态',
    `creator` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci
DEFAULT '' COMMENT '创建者',
    `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    `updater` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci
DEFAULT '' COMMENT '更新者',
    `update_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT '更新时间',
    `deleted` bit(1) NOT NULL DEFAULT b'0' COMMENT '是否删除',
    `tenant_id` bigint NOT NULL DEFAULT '0' COMMENT '租户编号',
    PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci COMMENT='用户
组';
```

① 表名的前缀，要和 Maven Module 的模块名保持一致。例如说，用户组在 `yudao-module-system` 模块，所以表名的前缀是 `system_`。

疑问：为什么要保持一致？

代码生成器会自动解析表名的前缀，获得其所属的 Maven Module 模块，简化配置过程。

② 设置 ID 主键，一般推荐使用 `bigint` 长整形，并设置自增长。

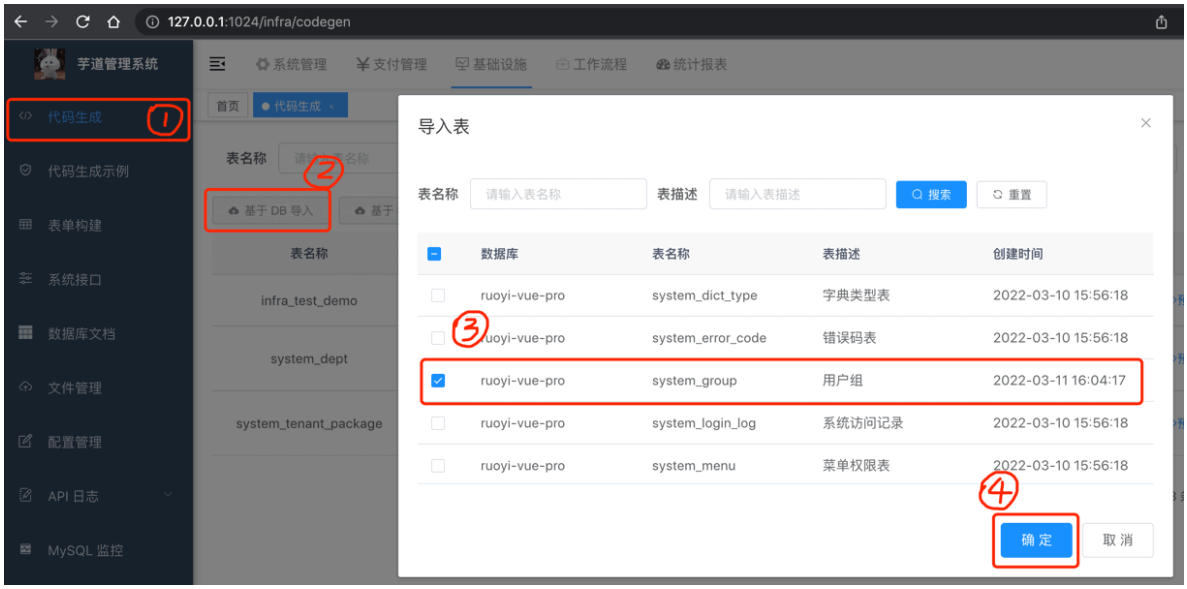
③ 正确设置每个字段是否允许空，代码生成器会根据它生成参数是否允许空的校验规则。

④ 正确设置注释，代码生成器会根据它生成字段名与提示等信息。

⑤ 添加 `creator`、`create_time`、`updater`、`update_time`、`deleted` 是必须设置的系统字段；如果开启多租户的功能，并且该表需要多租户的隔离，则需要添加 `tenant_id` 字段。

2. 代码生成

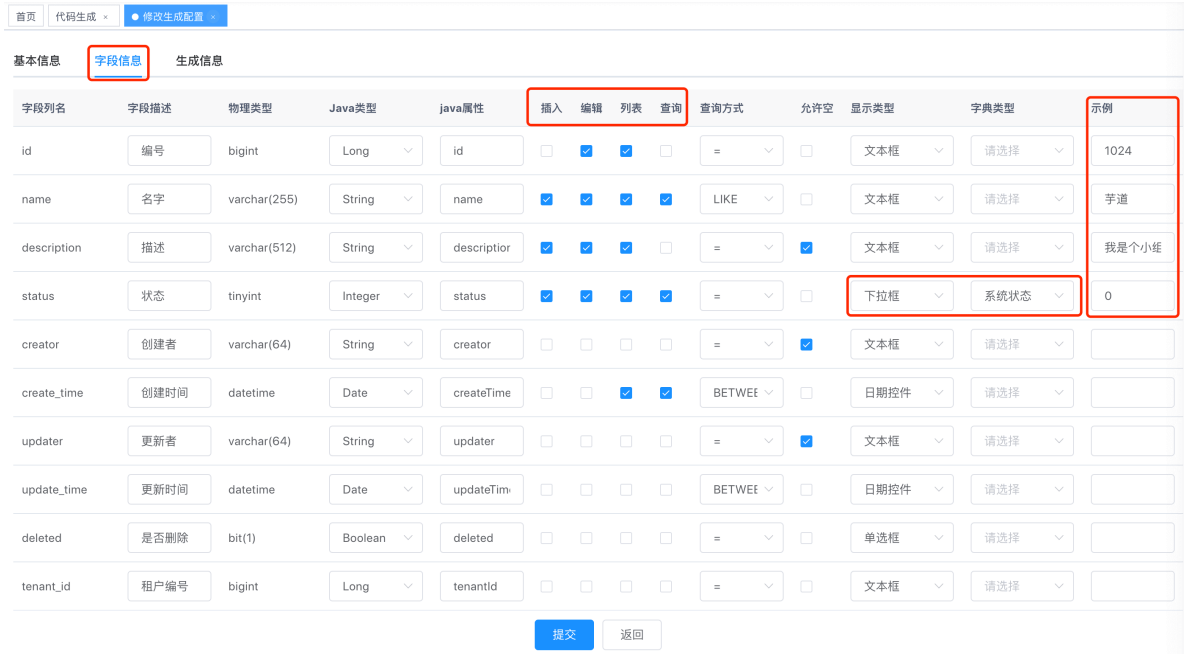
① 点击 [基础设施 -> 代码生成] 菜单，点击 [基于 DB 导入] 按钮，选择 `system_group` 表，后点击 [确认] 按钮。



代码实现？

可见 CodegenBuilder (opens new window)类，自动解析数据库的表结构，生成默认的配置。

② 点击 `system_group` 所在行的 [编辑] 按钮，修改生成配置。后操作如下：



- 将 `status` 字段的显示类型为【下拉框】，字典类型为【系统状态】。
- 将 `description` 字段的【查询】取消。
- 将 `id`、`name`、`description`、`status` 字段的【示例】填写上。

字段信息

- 插入：新增时，是否传递该字段。
- 编辑：修改时，是否传递该字段。
- 列表：Table 表格，是否展示该字段。
- 查询：搜索框，是否支持该字段查询，查询的条件是什么。
- 允许空：新增或修改时，是否必须传递该字段，用于 Validator 参数校验。
- 字典类型：在显示类型是下拉框、单选框、复选框时，选择使用的字典。
- 示例：参数示例，用于 Swagger 接口文档的 `example` 示例。

- 将【前端类型】设置为【Vue2 Element UI 标准模版】或【Vue3 Element Plus 标准模版】，具体根据你使用哪种管理后台。

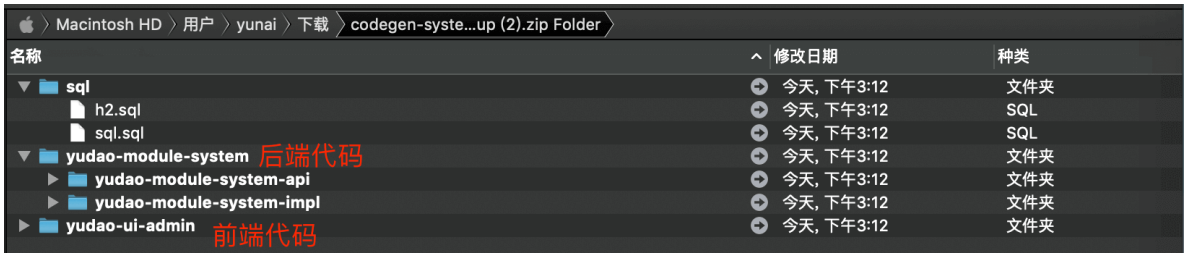
生成信息

- 生成场景：分成管理后台、用户 App 两种，用于生成 Controller 放在 `admin` 还是 `app` 包。
- 上级菜单：生成场景是管理后台时，需要设置其所属的上级菜单。
- 前端类型： 提供多种 UI 模版。
    - 【Vue3 Element Plus Schema 模版】，对应 《前端手册 Vue 3.X —— CRUD 组件》 说明。
    - 后端的 `application.yaml` 配置文件中的 `yudao.codegen.front-type` 配置项，设置默认的 UI 模版，避免每次都需要设置。

完成后，点击 [提交] 按钮，保存生成配置。

③ 点击 `system_group` 所在行的 [预览] 按钮，在线预览生成的代码，检查是否符合预期。

④ 点击 `system_group` 所在行的 [生成代码] 按钮，下载生成代码的压缩包，双击进行解压。



代码实现？

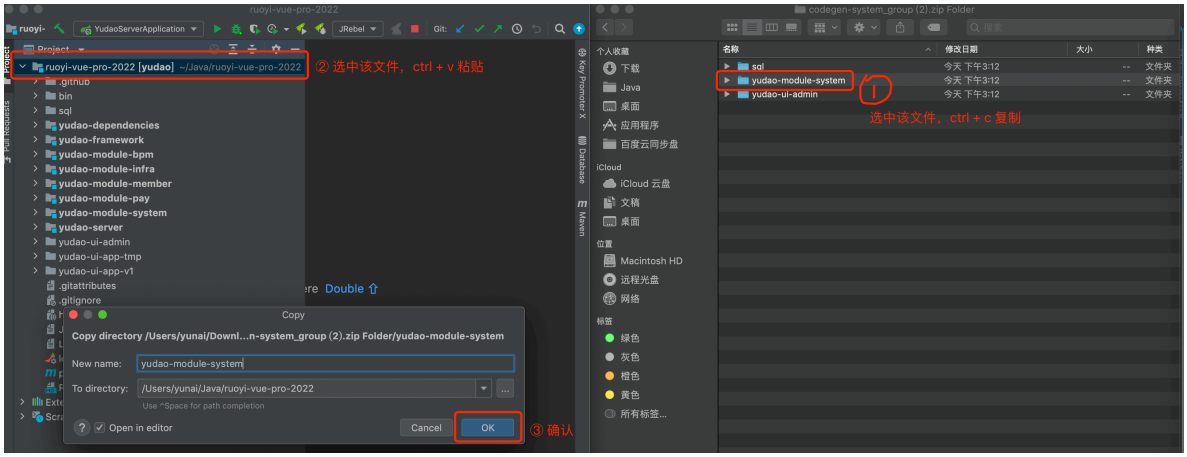
可见 CodegenEngine (opens new window)类，基于 Velocity 模板引擎，生成具体代码。模板文件，可见 `resources/codegen` (opens new window)目录。

3. 代码运行

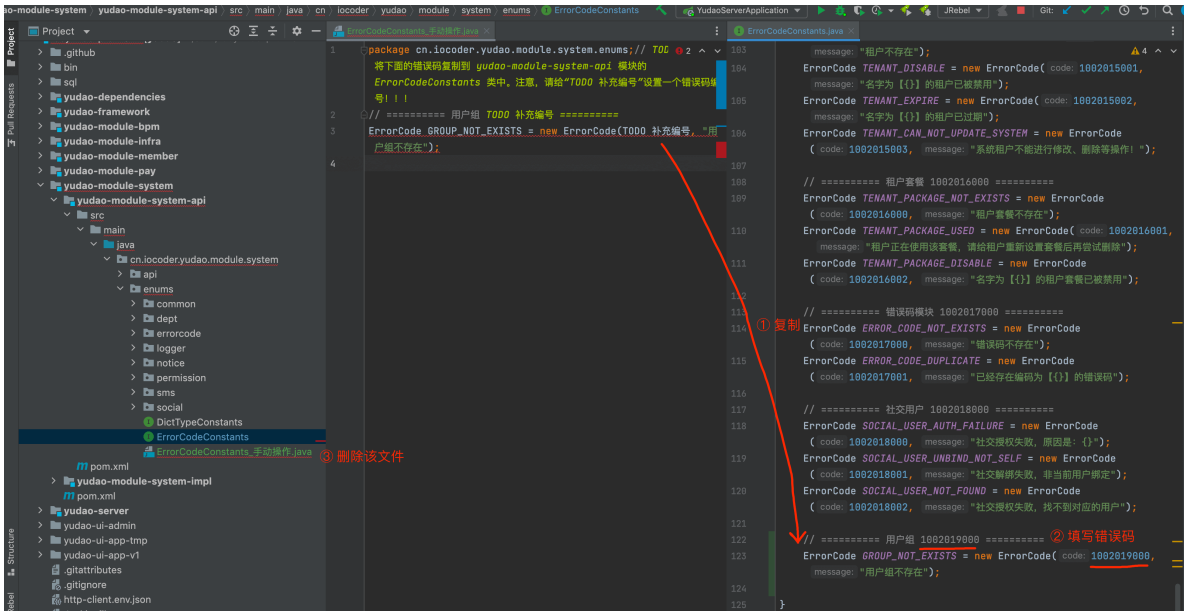
本小节，我们将生成的代码，复制到项目中，并进行运行。

3.1 后端运行

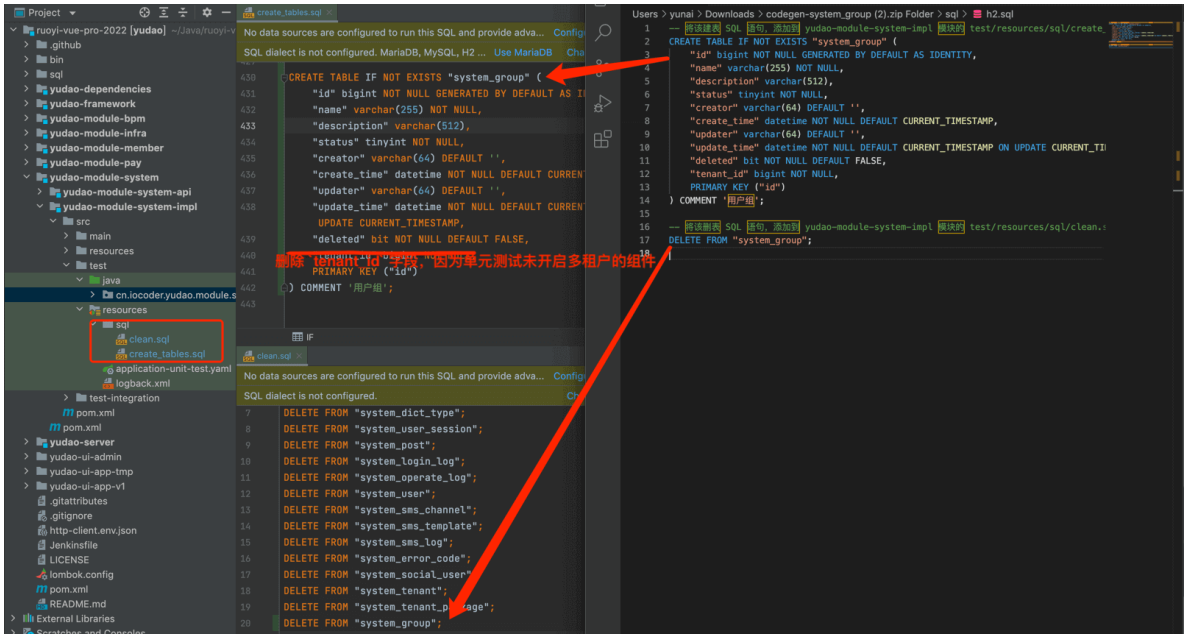
① 将生成的后端代码，复制到项目中。操作如下图所示：



② 将 `ErrorCodeConstants.java_手动操作` 文件的错误码，复制到该模块 ErrorCodeConstants 类中，并设置对应的错误码编号，之后进行删除。操作如下图所示：



③ 将 `h2.sql` 的 CREATE 语句复制到该模块的 `create_tables.sql` 文件，DELETE 语句复制到该模块的 `clean.sql`。操作如下图：

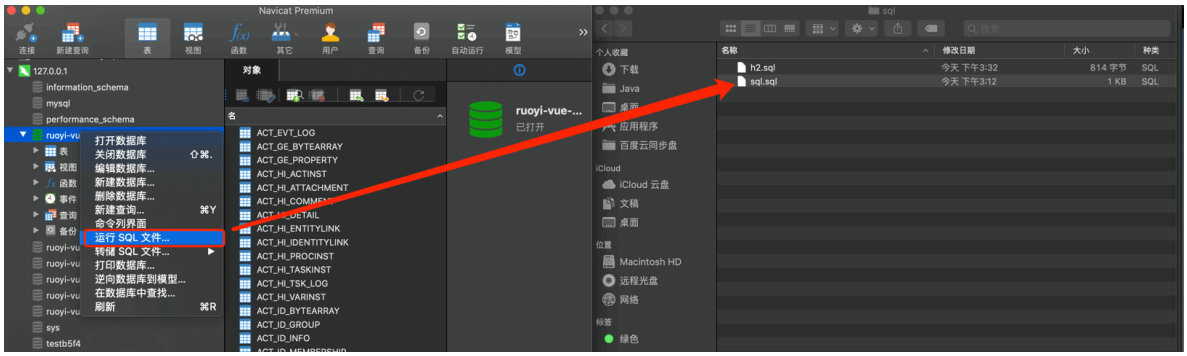


疑问：`create_tables.sql` 和 `clean.sql` 文件的作用是什么？

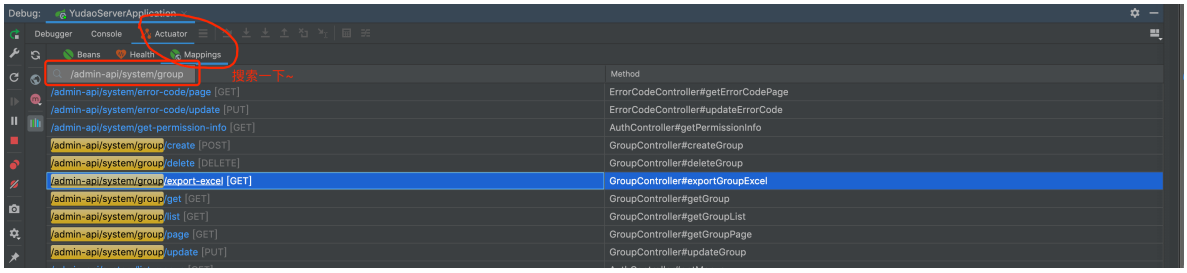
项目的单元测试，需要使用到 H2 内存数据库，`create_tables.sql` 用于创建所有的表结构，`clean.sql` 用于每个单元测试的方法跑完后清理数据。

然后，运行 GroupServiceImplTest 单元测试，执行通过。

④ 打开数据库工具，运行代码生成的 `sql/sql.sql` 文件，用于菜单的初始化。

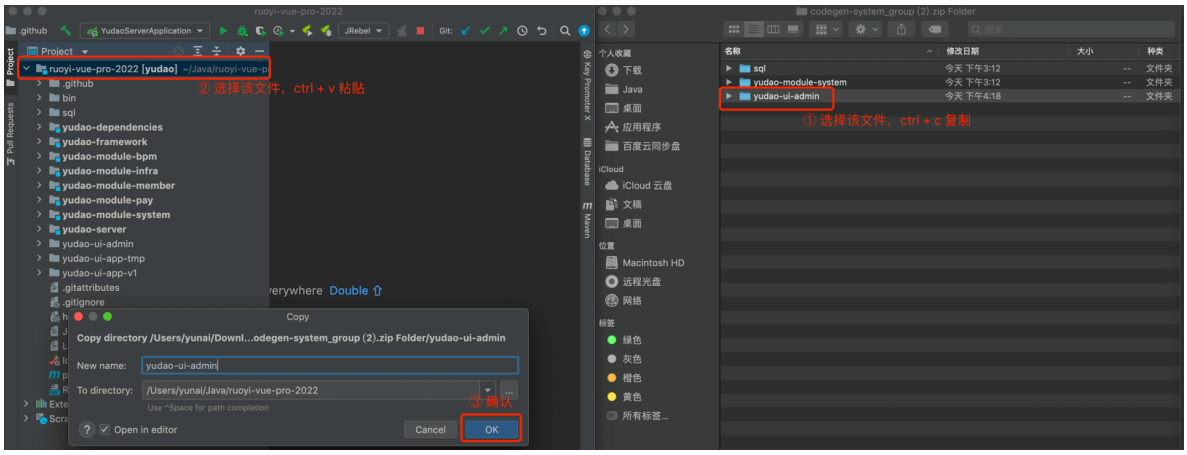


⑤ Debug 运行 YudaoServerApplication 类，启动后端项目。通过 IDEA 的 [Actuator -> Mappings] 菜单，可以看到代码生成的 GroupController 的 RESTful API 接口已经生效。



3.2 前端运行

① 将生成的前端代码，复制到项目中。操作如下图所示：



② **重新**执行 `npm run dev` 命令，启动前端项目。点击 [系统管理 -> 用户组管理] 菜单，就可以看到用户组的 UI 界面。

至此，我们已经完成了【**用户组**】功能的代码生成，基本节省了你 80% 左右的开发任务，后续可以根据自己的需求，进行剩余的 20% 的开发！

微服务手册

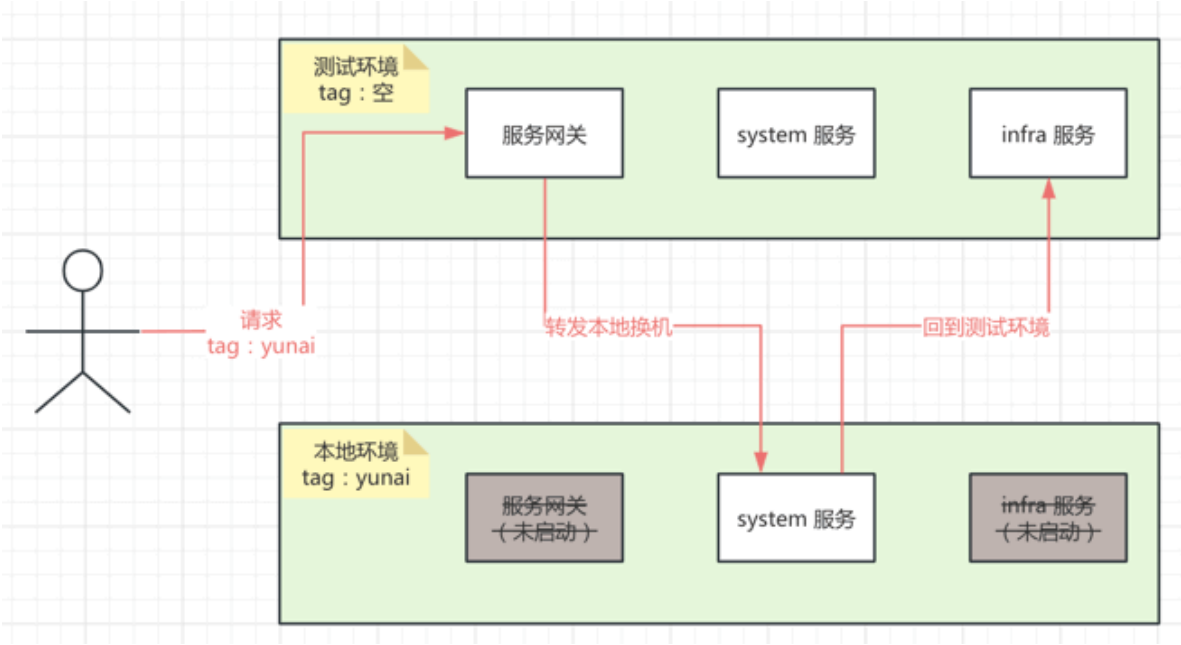
微服务调试

1. 多环境 env 组件

在微服务架构下，多服务的调试是个非常大的痛点：在大家使用 **同一个** 注册中心时，如果多个人在本地启动 **相同** 服务，可能需要调试的一个请求会打到其他开发的本地服务，实际是期望达到自己本地的服务。

一般的解决方案是，使用 **不同** 注册中心，避免出现这个情况。但是，服务一多之后，就会产生新的痛点，同时本地启动所有服务很占用电脑内存。

因此，我们实现了 `yudao-spring-boot-starter-env` (opens new window)组件，通过 Tag 给服务打标，实现在使用 **同一个** 注册中心的情况下，本地只需要启动需要调试的服务，并且保证自己的请求，必须达到自己本地的服务。如下图所示：



- 测试环境：启动了 gateway、system、infra 服务；本地环境：只启动了 system 服务
- 请求时，带上

```
tag = yunai
```

，优先请求本地环境 +

```
tag = yunai
```

的服务：

  - ① 由于本地未启动 gateway 服务，所以请求打到测试环境的 gateway 服务

- ② 由于请求 `tag = yunai`，所以转发到本地环境的 system 服务
- ③ 由于本地未启动 infra 服务，所以转发回测试环境的 infra 服务

2. 功能演示

在本地模拟，启动三个进程，如下图所示：

- `tag` 为空的 gateway、system 服务
- `tag` 为本机 `hostname`（例如说我本地是 `Yunai.local`）的 system 服务

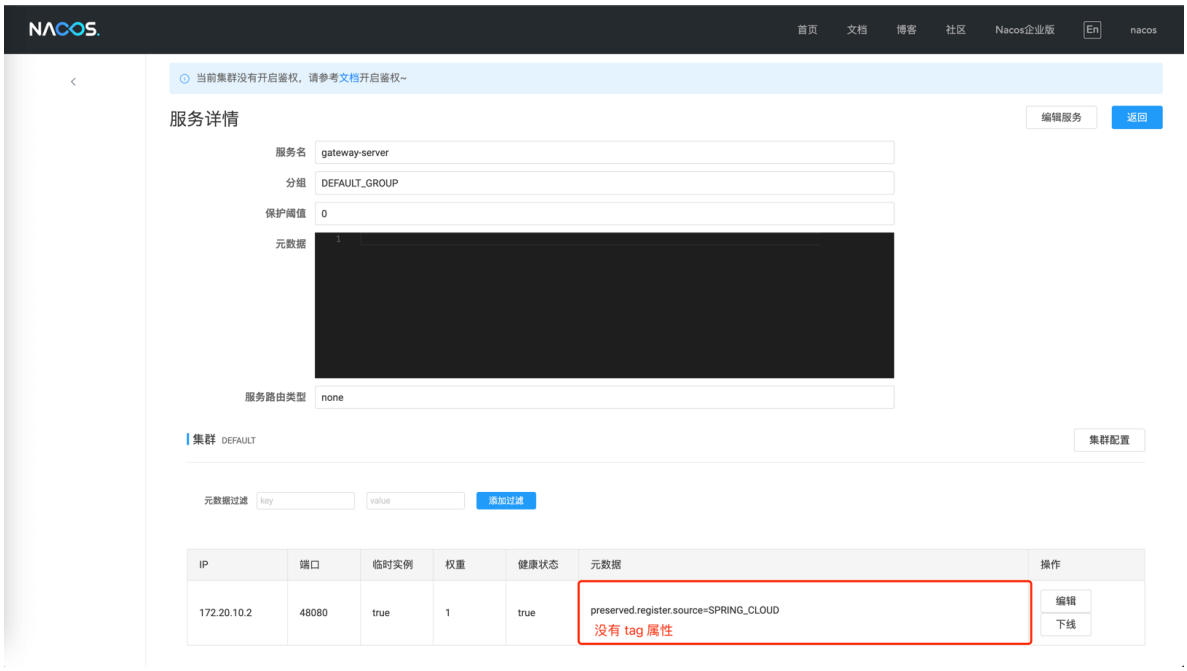
注意！！！

`hostname` 是你的主机名：

- Windows 在 cmd 里输入 hostname
- MacOS 在 terminal 里输入 hostname

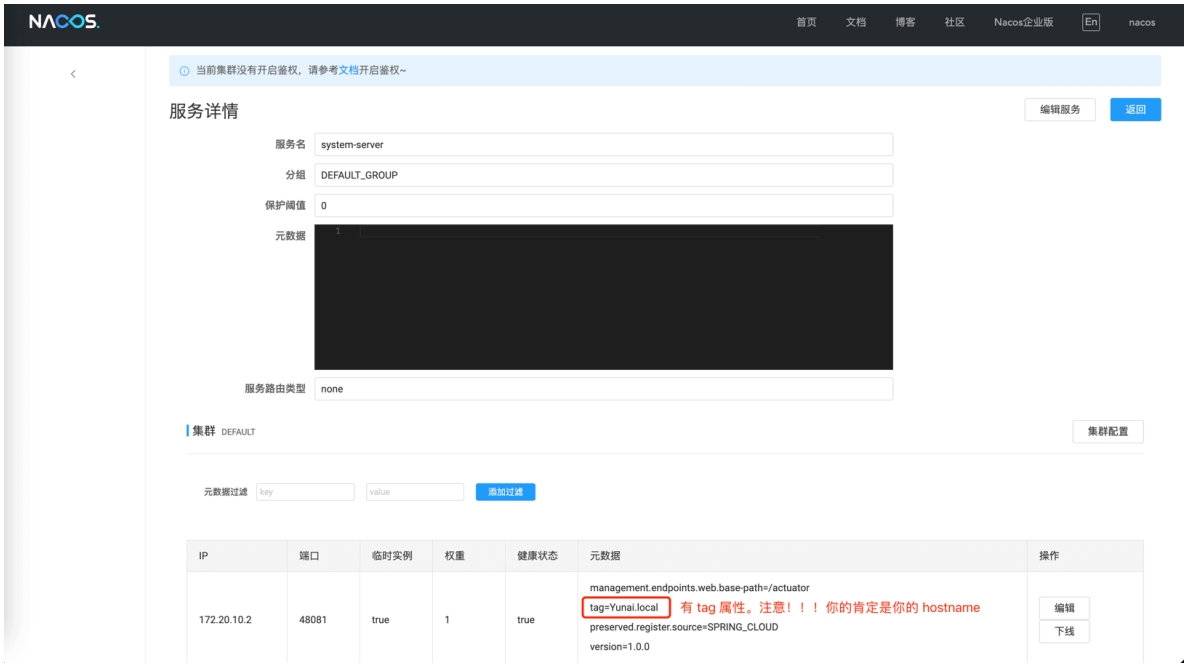
第一步，启动 gateway 服务

直接运行 GatewayServerApplication 类，启动 gateway 服务。此时，我们可以在 Nacos 看到该实例，它是没 `tag` 属性。如下图所示：

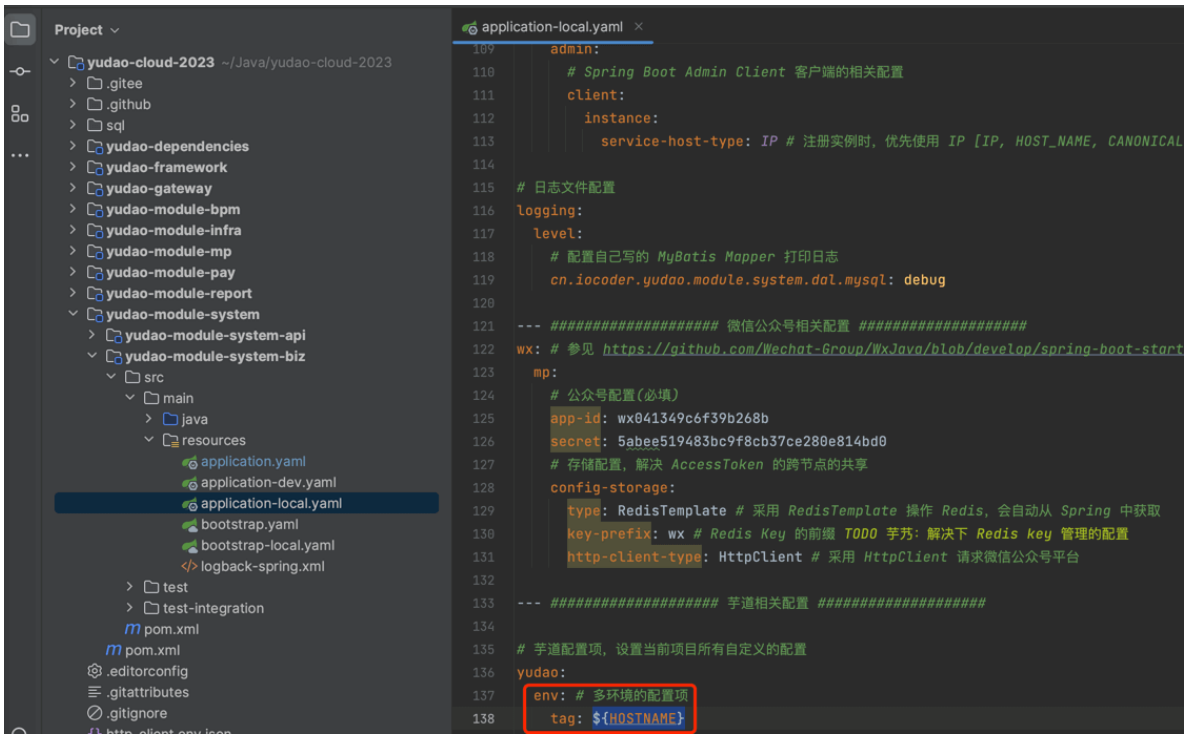


第二步，启动 system 服务【有 tag】

运行 SystemServerApplication 类，启动 system 服务。此时，我们可以在 Nacos 看到该实例，它是有 `tag` 属性。如下图所示：

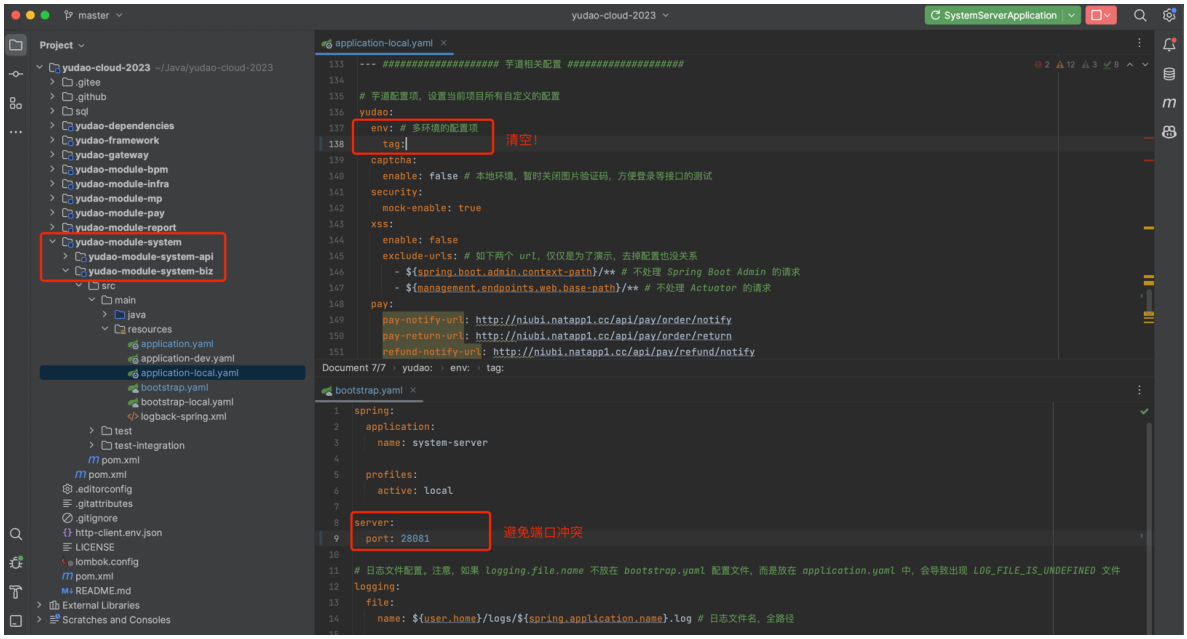


因为我们默认在 `application-local.yaml` 配置文件里，添加了 `yudao.env.tag` 配置项为 `${HOSTNAME}`。如下图所示：

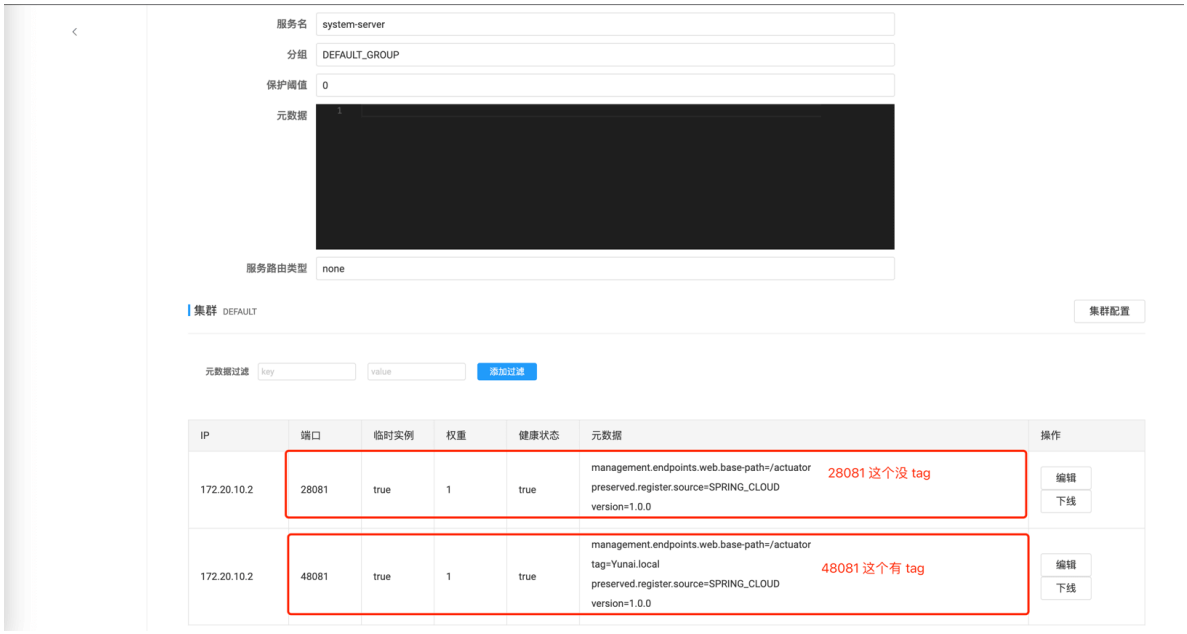


第三步，启动 system 服务 【无 tag】

① 修改 system 服务的端口为 28081，`yudao.env.tag` 配置项为空。如下图所示：

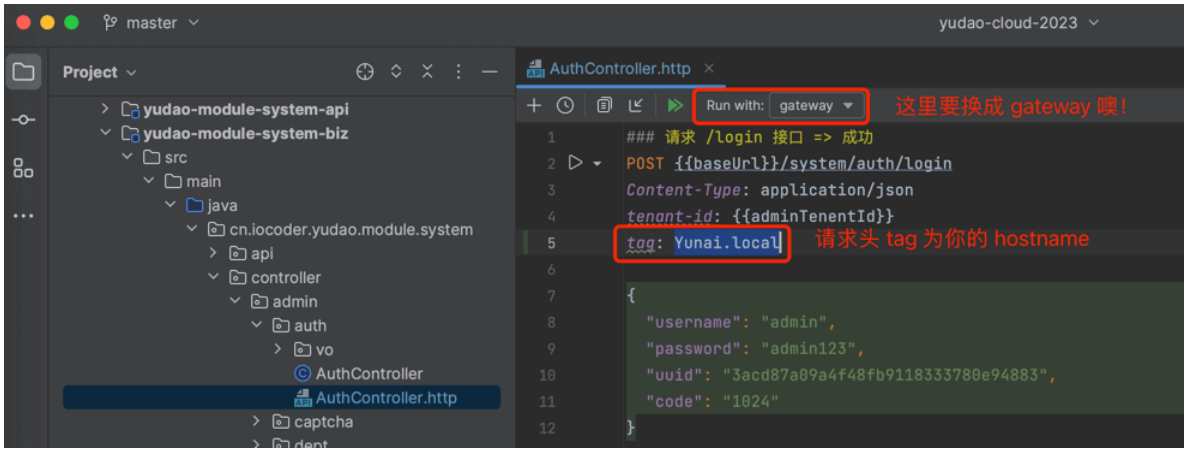


② 再一个 SystemServerApplication，**额外**启动 system 服务。此时，我们可以在 Nacos 看到该实例，它是没 `tag` 属性。如下图所示：



第四步，请求测试

① 打开 `AuthController.http` 文件，设置第一个请求的 `tag` 为 `Yunai.local`（要替换成你的 hostname），如下图所示：



② 点击前面的绿色小箭头，发起请求。从 IDEA 控制台的日志可以看到，只有有 tag 的 system 服务才会被调用。

你可以多点几次，依然是这样的情况噢！

3. 实现原理

① 在服务注册时，会将 `yudao.env.tag` 配置项，写到 Nacos 服务实例的元数据，通过 EnvEnvironmentPostProcessor (opens new window)类实现。

② 在服务调用时，通过 EnvLoadBalancerClient (opens new window)类，筛选服务实例，通过服务实例的 `tag` 元数据，匹配请求的 `tag` 请求头。

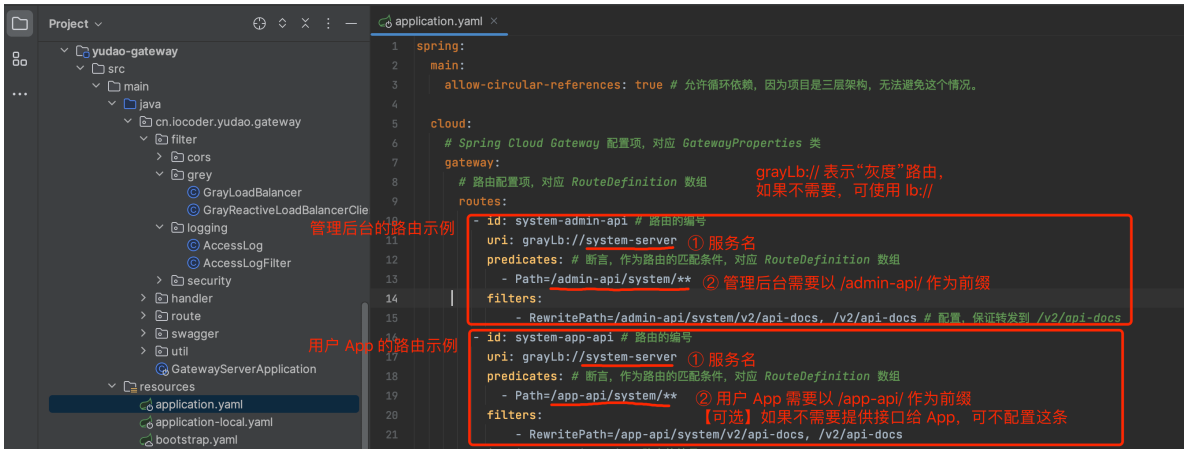
③ 在网关转发时，通过 GrayLoadBalancer (opens new window)类，效果和 EnvLoadBalancerClient 一致。

服务网关 Spring Cloud Gateway

zjugis-gateway 模块，基于 Spring Cloud Gateway 构建 API 服务网关，提供用户认证、服务路由、灰度发布、访问日志、异常处理等功能。

1. 服务路由

新建服务后，在 `application.yaml` (opens new window)配置文件中，需要添加该服务的路由配置。示例如下图：

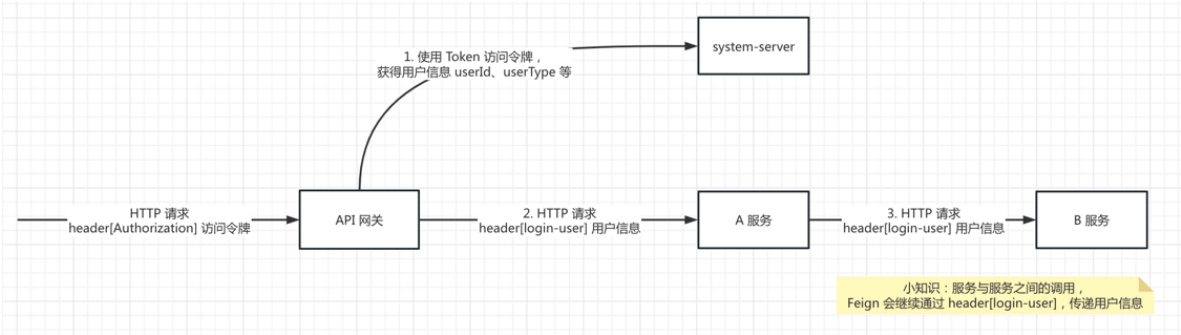


2. 用户认证

由 `filter/security` (opens new window)包实现，无需配置。

TokenAuthenticationFilter 会获得请求头中的 `Authorization` 字段，调用 `system-server` 服务，进行用户认证。

- 如果认证成功，会将用户信息放到 `login-user` 请求头，转发到后续服务。后续服务可以从 `login-user` 请求头，解析 (opens new window)到用户信息。
- 如果认证失败，依然会转发到后续服务，由该服务决定是否需要登录，是否需要校验权限。



考虑到性能，API 网关会本地缓存 (opens new window)Token 与用户信息，每次收到 HTTP 请求时，异步从 `system-server` 刷新本地缓存。

3. 灰度发布

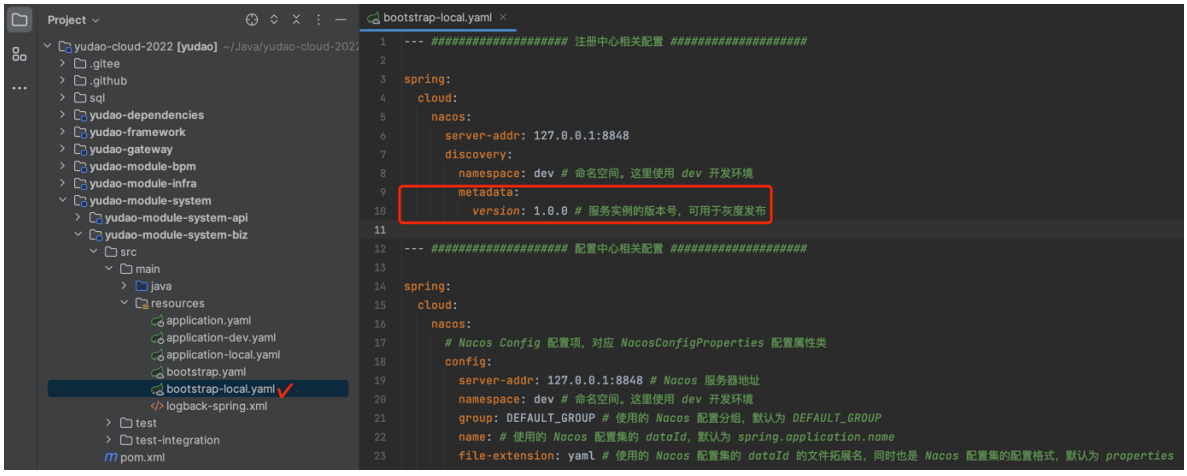
由 `filter/grey` (opens new window)包实现，实现原理如下：

```java
/**
 * 灰度 {@link GrayLoadBalancer} 实现类
 *
 * 根据请求的 header[version] 匹配，筛选满足 metadata[version] 相等的服务实例列表，然后随机 + 权重进行选择一个
 * 1. 假如请求的 header[version] 为空，则不进行筛选，所有服务实例都进行选择
 * 2. 如果 metadata[version] 都不相等，则不进行筛选，所有服务实例都进行选择
 *
 * 注意，考虑到实现的简易，它的权重是使用 Nacos 的 nacos.weight，所以随机 + 权重也是基于 {@link NacosBalancer} 筛选。
 * 也就是说，如果你不使用 Nacos 作为注册中心，需要微调一下筛选的实现逻辑
 *
 * @author 芋道源码
 */
@RequiredArgsConstructor
@Slf4j
public class GrayLoadBalancer implements ReactorServiceInstanceLoadBalancer {
```

所以在使用灰度时，如要如下配置：

① 第一步，【网关】配置服务的路由配置使用 `grebLb://` 协议，指向灰度服务。例如说：

② 第二步，【服务】配置服务的版本 `version` 配置。例如说：



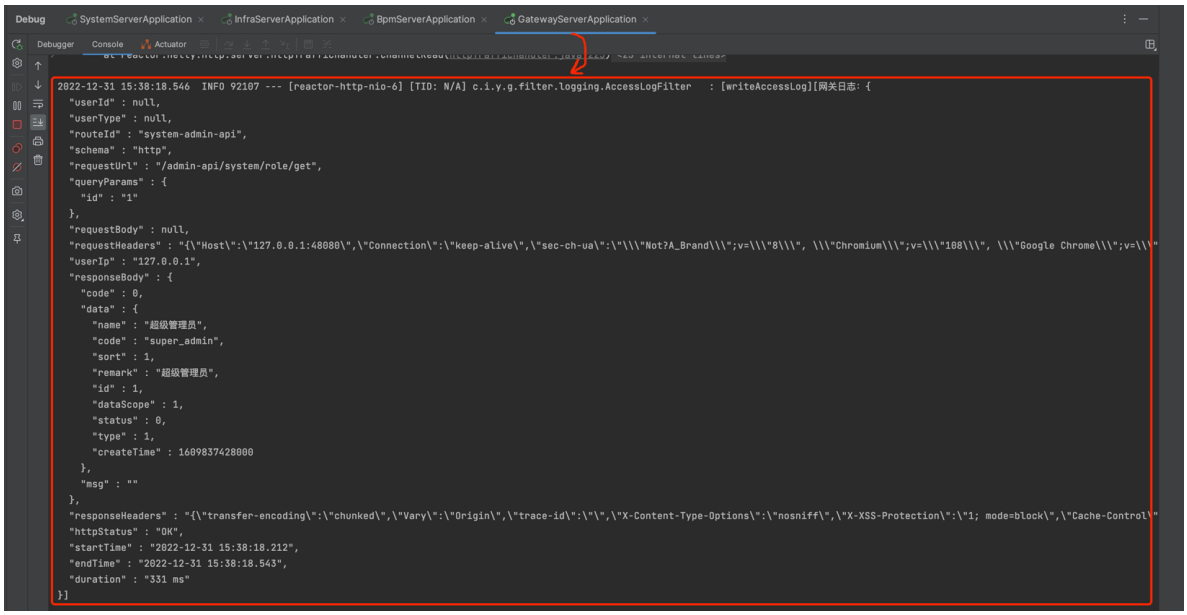
③ 第三步，请求 API 网关时，请求头带上想要 `version` 版本。

可能想让用户的请求带上 `version` 请求头比较难，可以通过 Spring Cloud Gateway 修改请求头，通过 User Agent、Cookie、登录用户等信息，来判断用户想要的版本。详细的解析，可见《Spring Cloud Gateway 实现灰度发布功能 》(opens new window)文章。

4. 访问日志

由 `filter/logging` (opens new window)包实现，无需配置。

每次收到 HTTP 请求时，会打印访问日志，包括 Request、Response、用户等信息。如下图所示：



5. 异常处理

由 GlobalExceptionHandler (opens new window)累实现，无需配置。

请求发生异常时，会翻译异常信息，返回给用户。例如说：

```
{
  "code": 500,
  "data": null,
  "msg": "系统异常"
}
```

6. 动态路由

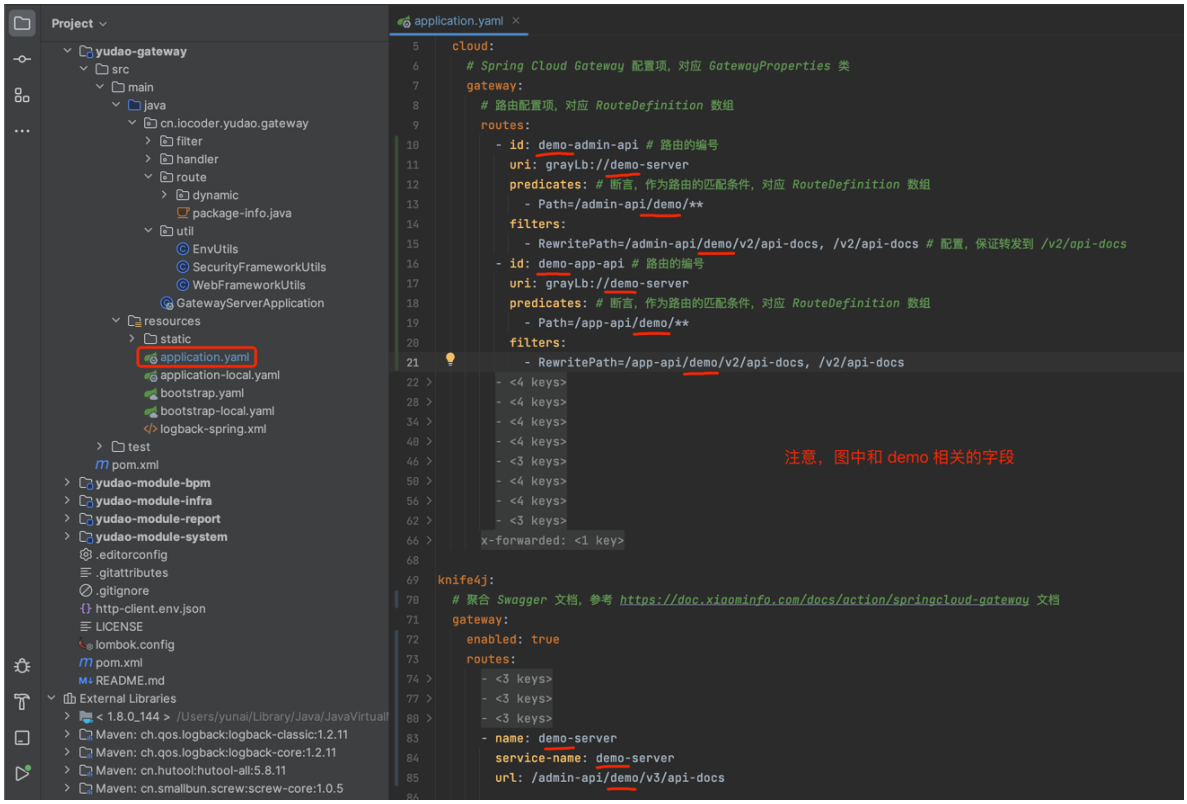
在 Nacos 配置发生变化时，Spring Cloud Alibaba Nacos Config 内置的监听器，会监听到配置刷新，最终触发 Gateway 的路由信息刷新。

参见《芋道 Spring Cloud 网关 Spring Cloud Gateway 入门 》(opens new window)博客的「6. 基于配置中心 Nacos 实现动态路由」小节。

使用方式：在 Nacos 新增 DataId 为 `gateway-server.yaml` 的配置，修改 `spring.cloud.gateway.routes` 配置项。

7. Swagger 接口文档

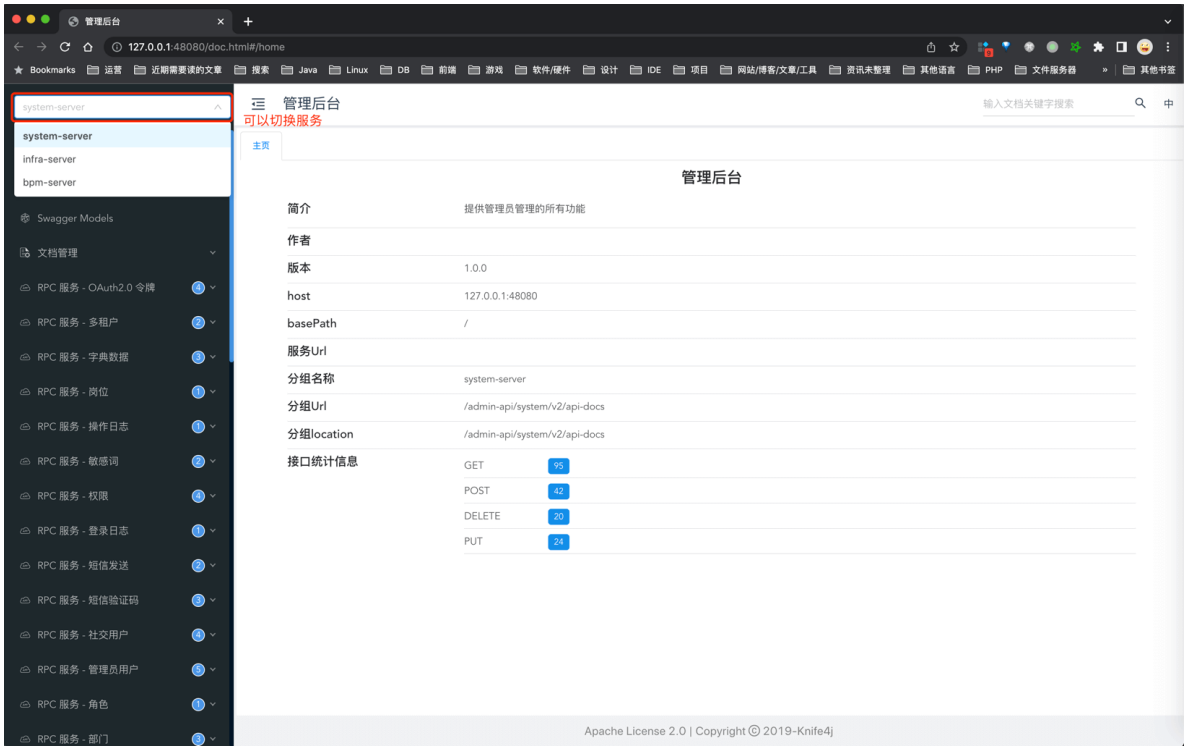
基于 Knife4j 实现 Swagger 接口文档的 网关聚合 (opens new window)。需要路由配置如下：



友情提示：图中的 /v2/ 都改成 /v3/，或者以下面的文字为准！！！

- 管理后台的接口：`- RewritePath=/admin-api/{服务的基础路由}/v3/api-docs, /v3/api-docs`
- 用户 App 的接口：`- RewritePath=/app-api/{服务的基础路由}/v3/api-docs, /v3/api-docs`
- Knife4j 配置：`knife4j.gateway.routes` 添加

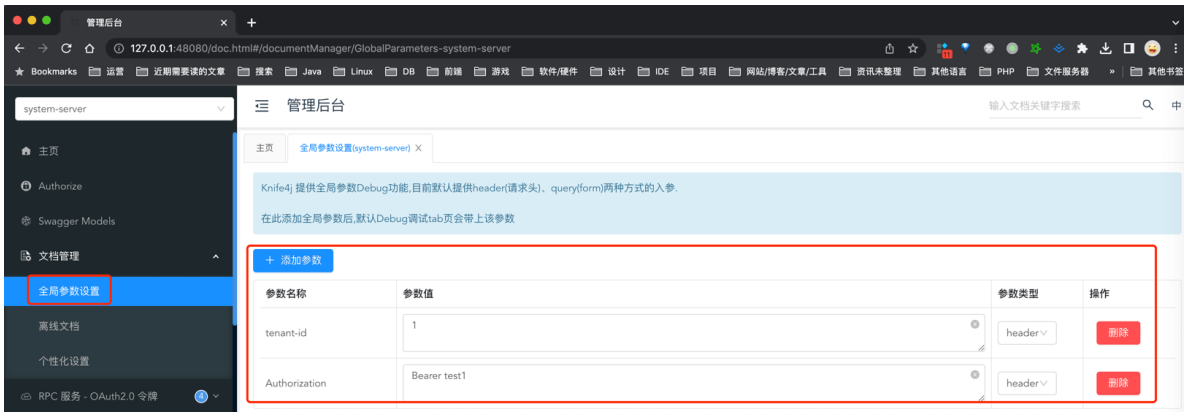
浏览器访问 http://127.0.0.1:48080/doc.html (opens new window)地址，可以看到所有接口的信息。如下图所示：



7.1 如何调用

○ 点击左边「文档管理 - 全局参数设置」菜单，设置 `header-id` 和 `Authorization` 请求头。如下图所示：

```
tenant-id：1
Authorization: Bearer test1
```



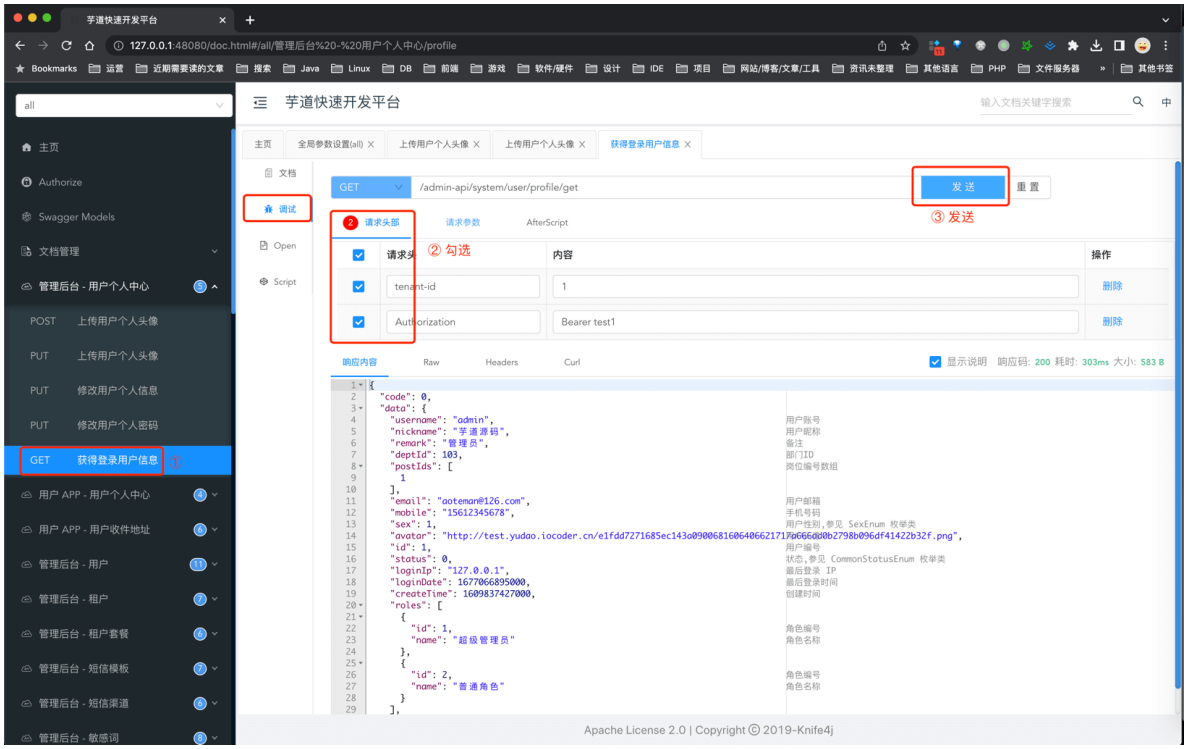
添加完后，需要 F5 刷新下网页，否则全局参数不生效。

① 点击任意一个接口，进行接口的调用测试。这里，使用「管理后台 - 用户个中心」的“获得登录用户信息”举例子。

② 点击左侧「调试」按钮，并将请求头部的 `header-id` 和 `Authorization` 勾选上。

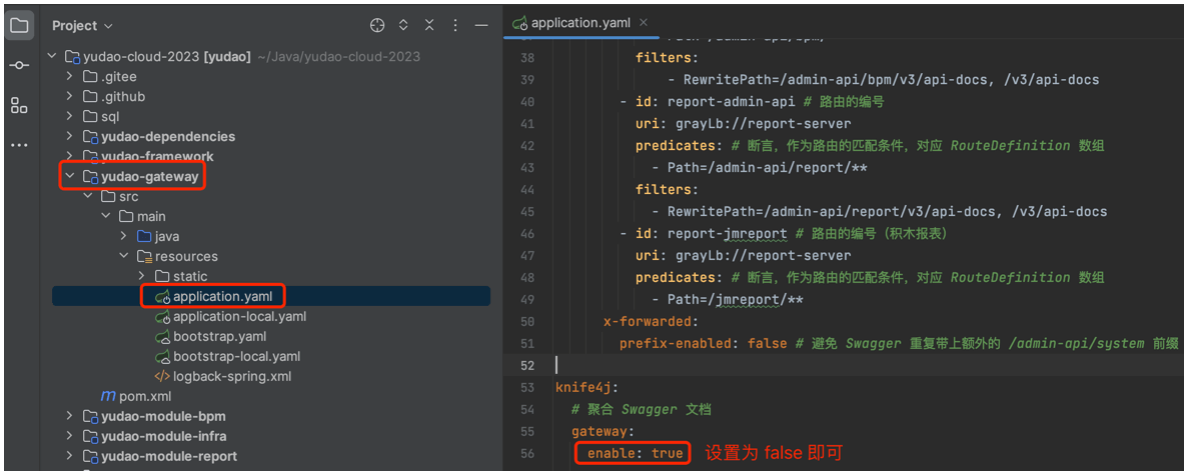
其中，`header-id` 为租户编号，`Authorization` 的 `"Bearer test"` 后面为用户编号（模拟哪个用户操作）。

③ 点击「发送」按钮，即可发起一次 API 的调用。



7.2 如何关闭

如果想要禁用 Swagger 功能，可通过 `knife4j.gateway.enabled` 配置项为 `false`。一般情况下，建议 prod 生产环境进行禁用，避免发生安全问题。



8. Cors 跨域处理

由 `filter/cors` (opens new window)包实现，无需配置。

服务调用 Feign

1. RPC 使用规约

本小节，我们来讲解下项目中 RPC 使用的规约。

1.1 API 前缀

API 使用 HTTP 协议，所有的 API 前缀，都以 `/rpc-api` (opens new window)开头，方便做统一的全局处理。

1.2 API 权限

服务之间的调用，不需要进行权限校验，所以需要在每个服务的 SecurityConfiguration 权限配置类中，添加如下配置：

```
// RPC 服务的安全配置
registry.antMatchers(ApiConstants.PREFIX + "/**").permitAll();
```

1.3 API 全局返回

所有 API 接口返回使用 CommonResult (opens new window)返回，和前端 RESTful API 保持统一。例如说：

```
public interface DeptApi {

    @GetMapping(PREFIX + "/get")
    @Operation(summary = "获得部门信息")
    @Parameter(name = "id", description = "部门编号", required = true, example = "1024")
    CommonResult<DeptRespDTO> getDept(@RequestParam("id") Long id);

}
```

1.4 用户传递

服务调用时，已经封装 Feign 将用户信息通过 HTTP 请求头 `login-user` 传递，通过 LoginUserRequestInterceptor.java 类实现。

这样，被调用服务，可以通过 SecurityFrameworkUtils 获取到用户信息，例如说：

- `#getLoginUser()` 方法，获取当前用户。
- `#getLoginUserId()` 方法，获取当前用户编号。

2. 如何定义一个 API 接口

本小节，我们来讲解下如何定义一个 API 接口。以 AdminUserApi 提供的 getUser 接口来举例子。

2.1 服务提供者

AdminUserApi 由 `system-server` 服务所提供。

2.1.1 ApiConstants

在 `yudao-module-system-api` 模块，创建 ApiConstants (opens new window)类，定义 API 相关的枚举。代码如下：

```
public class ApiConstants {

    /**
     * 服务名
     *
     * 注意，需要保证和 spring.application.name 保持一致
     */
    public static final String NAME = "system-server";

    public static final String PREFIX = RpcConstants.RPC_API_PREFIX +
 "/system";

    public static final String VERSION = "1.0.0";

}
```

2.1.2 AdminUserApi

在 `yudao-module-system-api` 模块，创建 AdminUserApi (opens new window)类，定义 API 接口。代码如下：

```
@FeignClient(name = ApiConstants.NAME) // ① @FeignClient 注解
@Tag(name = "RPC 服务 - 管理员用户") // ② Swagger 接口文档
public interface AdminUserApi {

    String PREFIX = ApiConstants.PREFIX + "/user";

    @GetMapping(PREFIX + "/get") // ③ Spring MVC 接口注解
    @Operation(summary = "通过用户 ID 查询用户")  // ② Swagger 接口文档
    @Parameter(name = "id", description = "部门编号", required = true, example =
"1024") // ② Swagger 接口文档
    CommonResult<AdminUserRespDTO> getUser(@RequestParam("id") Long id);

}
```

另外，需要创建 AdminUserRespDTO (opens new window)类，定义用户 Response DTO。代码如下：

```
@Data
public class AdminUserRespDTO {

    /**
     * 用户ID
     */
    private Long id;
    /**
     * 用户昵称
     */
    private String nickname;
    /**
     * 帐号状态
     *
     * 枚举 {@link CommonStatusEnum}
     */
    private Integer status;

    /**
     * 部门ID
     */
    private Long deptId;
    /**
     * 岗位编号数组
     */
    private Set<Long> postIds;
    /**
     * 手机号码
     */
    private String mobile;

}
```

2.1.3 AdminUserRpcImpl

在 `yudao-module-system-biz` 模块，创建 AdminUserRpcImpl (opens new window)类，实现 API 接口。代码如下：

```
@RestController // 提供 RESTful API 接口，给 Feign 调用
@Validated
public class AdminUserApiImpl implements AdminUserApi {

    @Resource
    private AdminUserService userService;

    @Override
    public CommonResult<AdminUserRespDTO> getUser(Long id) {
        AdminUserDO user = userService.getUser(id);
        return success(UserConvert.INSTANCE.convert4(user));
    }

}
```

2.2 服务消费者

`bpm-server` 服务，调用了 AdminUserApi 接口。

2.2.1 引入依赖

在 `yudao-module-bpm-biz` 模块的 `pom.xml` (opens new window)，引入 `yudao-module-system-api` 模块的依赖。代码如下：

```
<dependency>
    <groupId>cn.iocoder.cloud</groupId>
    <artifactId>yudao-module-system-api</artifactId>
    <version>${revision}</version>
</dependency>
```

2.2.2 引用 API

在 `yudao-module-bpm-biz` 模块，创建 RpcConfiguration (opens new window)配置类，注入 AdminUserApi 接口。代码如下：

```
@Configuration(proxyBeanMethods = false)
@EnableFeignClients(clients = {AdminUserApi.class.class})
public class RpcConfiguration {
}
```

2.2.3 调用 API

例如说，BpmTaskServiceImpl (opens new window)调用了 AdminUserApi 接口，代码如下：

```
@Service
public class BpmTaskServiceImpl implements BpmTaskService {

    @Resource
    private AdminUserApi adminUserApi;

    @Override
    public void updateTaskExtAssign(Task task) {
        // ... 省略非关键代码
        AdminUserRespDTO startUser = adminUserApi.getUser(id).getCheckedData();
    }
}
```

前端手册 VUE 3

开发规范

0. 实战案例

本小节，提供大家开发管理后台的功能时，最常用的普通列表、树形列表、新增与修改的表单弹窗、详情表单弹窗的实战案例。

0.1 普通列表

可参考 [系统管理 -> 岗位管理] 菜单：

- API 接口：`/src/api/system/post/index.ts` (opens new window)
- 列表界面：`/src/views/system/post/index.vue` (opens new window)
- 表单界面：`/src/views/system/post/PostForm.vue` (opens new window)

为什么界面拆成列表和表单两个 Vue 文件？

每个 Vue 文件，只实现一个功能，更简洁，维护性更好，Git 代码冲突概率低。

0.2 树形列表

可参考 [系统管理 -> 部门管理] 菜单：

- API 接口：`/src/api/system/dept/index.ts` (opens new window)
- 列表界面：`/src/views/system/dept/index.vue` (opens new window)
- 表单界面：`/src/views/system/dept/DeptForm.vue` (opens new window)

0.3 高性能列表

可参考 [系统管理 -> 地区管理] 菜单，对应 `/src/views/system/area/index.vue` (opens new window)列表界面

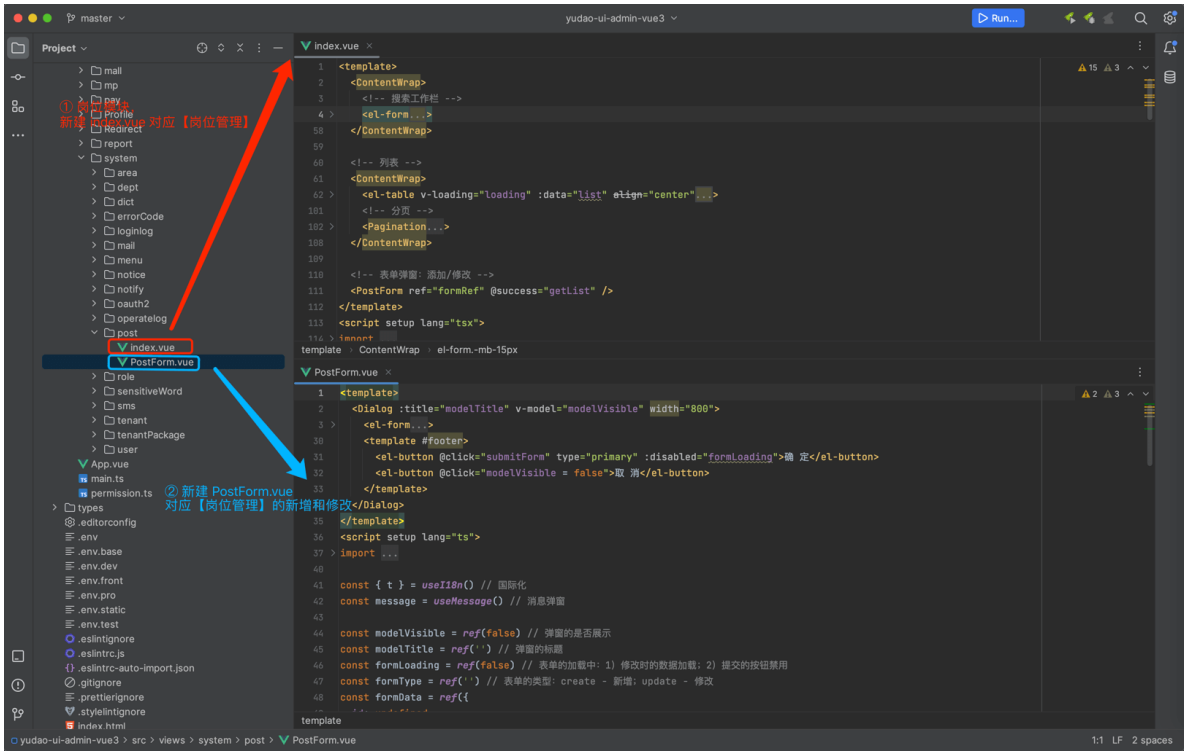
基于 Virtualized Table 虚拟化表格 (opens new window)实现，解决一屏里超过 1000 条数据记录时，就会出现卡顿等性能问题。

0.4 详情弹窗

可参考 [基础设施 -> API 日志 -> 访问日志] 菜单，对应 `/src/views/infra/apiAccessLog/ApiAccessLogDetail.vue` (opens new window)详情弹窗

1. view 页面

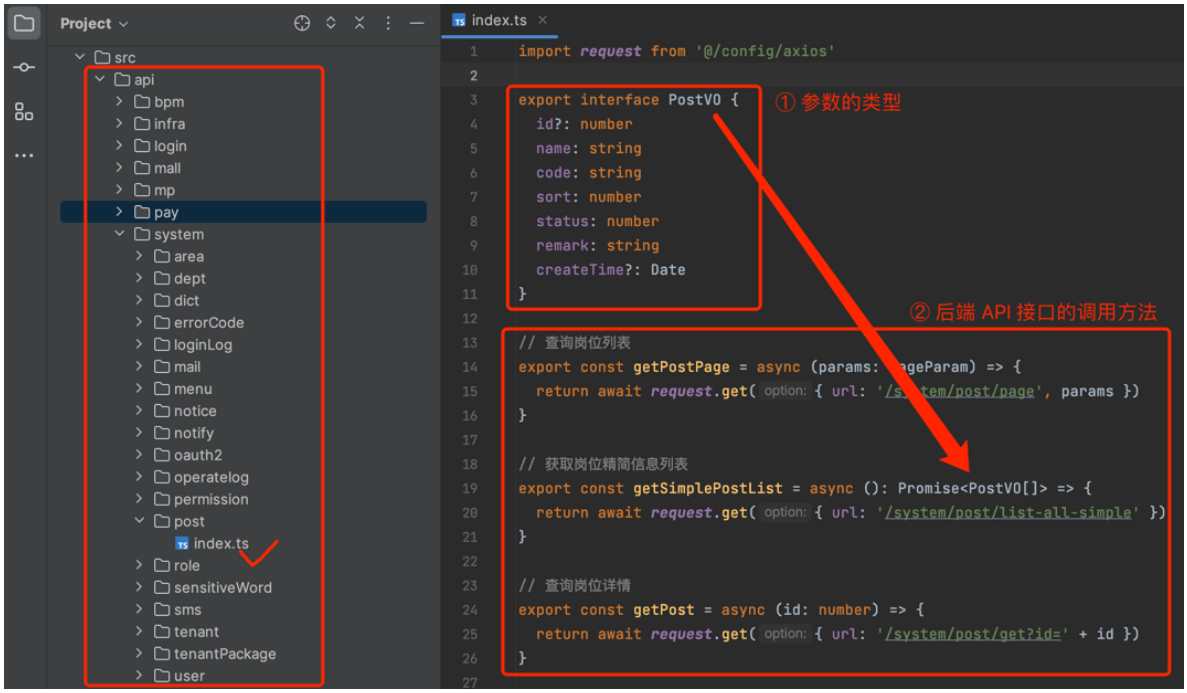
在 `@views` (opens new window)目录下，每个模块对应一个目录，它的所有功能的 `.vue` 都放在该目录里。



一般来说，一个路由对应一个 `index.vue` 文件。

2. api 请求

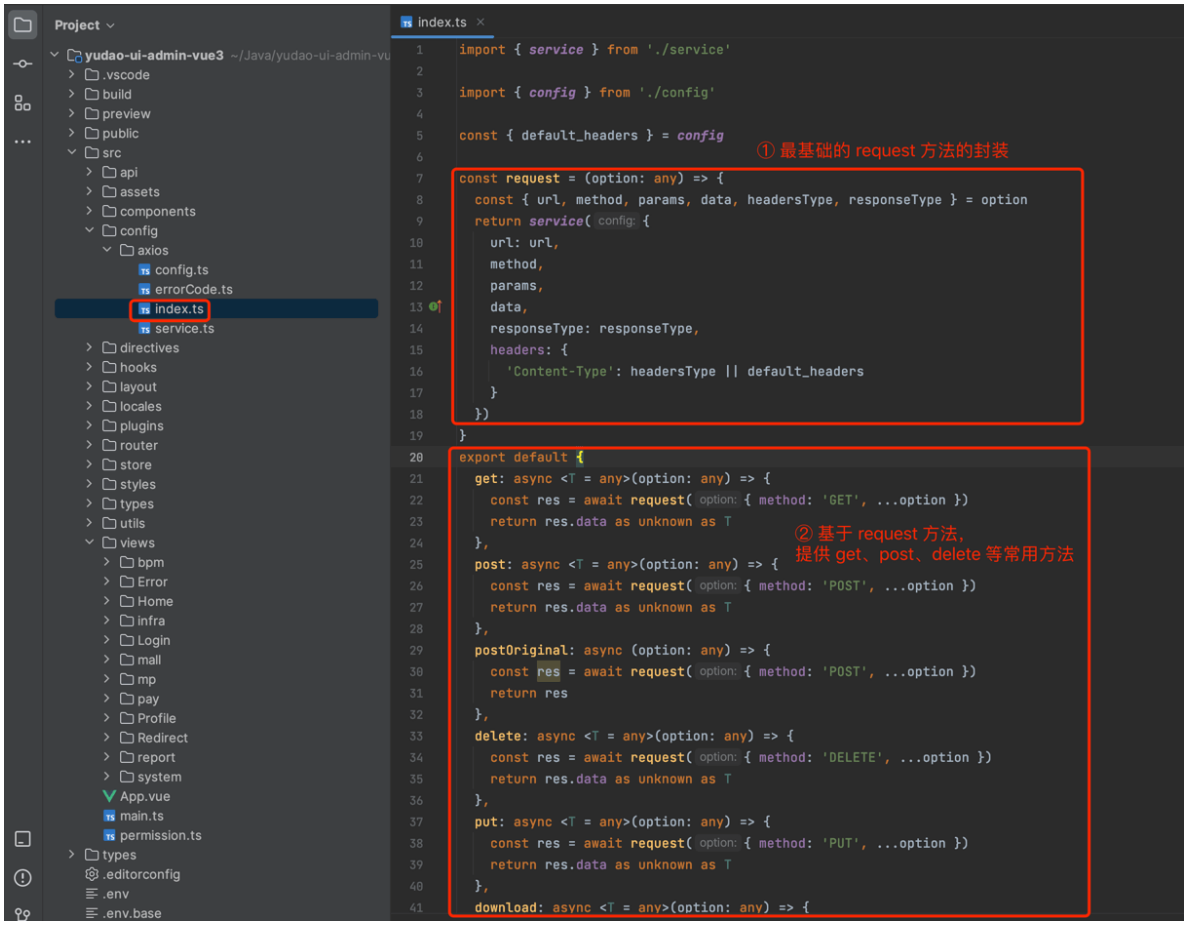
在 `@/api` (opens new window)目录下，每个模块对应一个 `index.ts` API 文件。



- API 方法：会调用 `request` 方法，发起对后端 RESTful API 的调用。
- `interface` 类型：定义了 API 的请求参数和返回结果的类型，对应后端的 VO 类型。

2.1 请求封装

`/src/config/axios/index.ts` (opens new window)基于 axios (opens new window)封装，统一处理 GET、POST 方法的请求参数、请求头，以及错误提示信息等。



2.1.1 创建 axios 实例

- `baseURL` 基础路径
- `timeout` 超时时间，默认为 30000 毫秒

▶ 实现代码 /src/config/axios/service.ts

2.1.2 Request 拦截器

- 【重点】`Authorization`、`tenant-id` 请求头
- GET 请求参数的拼接

▶ 实现代码 /src/config/axios/service.ts

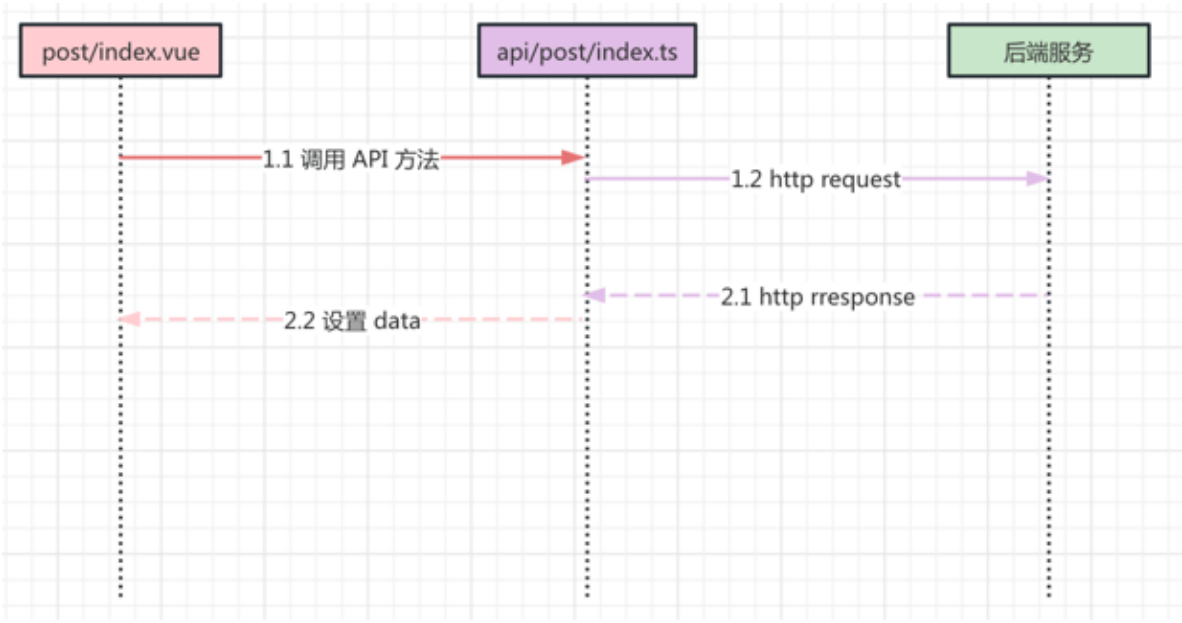
2.1.3 Response 拦截器

- 访问令牌 AccessToken 过期时，使用刷新令牌 RefreshToken 刷新，获得新的访问令牌
- 刷新令牌失败（过期）时，跳回首页进行登录
- 请求失败，Message 错误提示

▶ 实现代码 /src/config/axios/service.ts

2.2 交互流程

一个完整的前端 UI 交互到服务端处理流程，如下图所示：



继续以 [系统管理 -> 岗位管理] 菜单为例，查看它是如何读取岗位列表的。代码如下：

```
// ① api/system/post/index.ts
import request from '@/config/axios'

// 查询岗位列表
export const getPostPage = async (params: PageParam) => {
  return await request.get({ url: '/system/post/page', params })
}
```

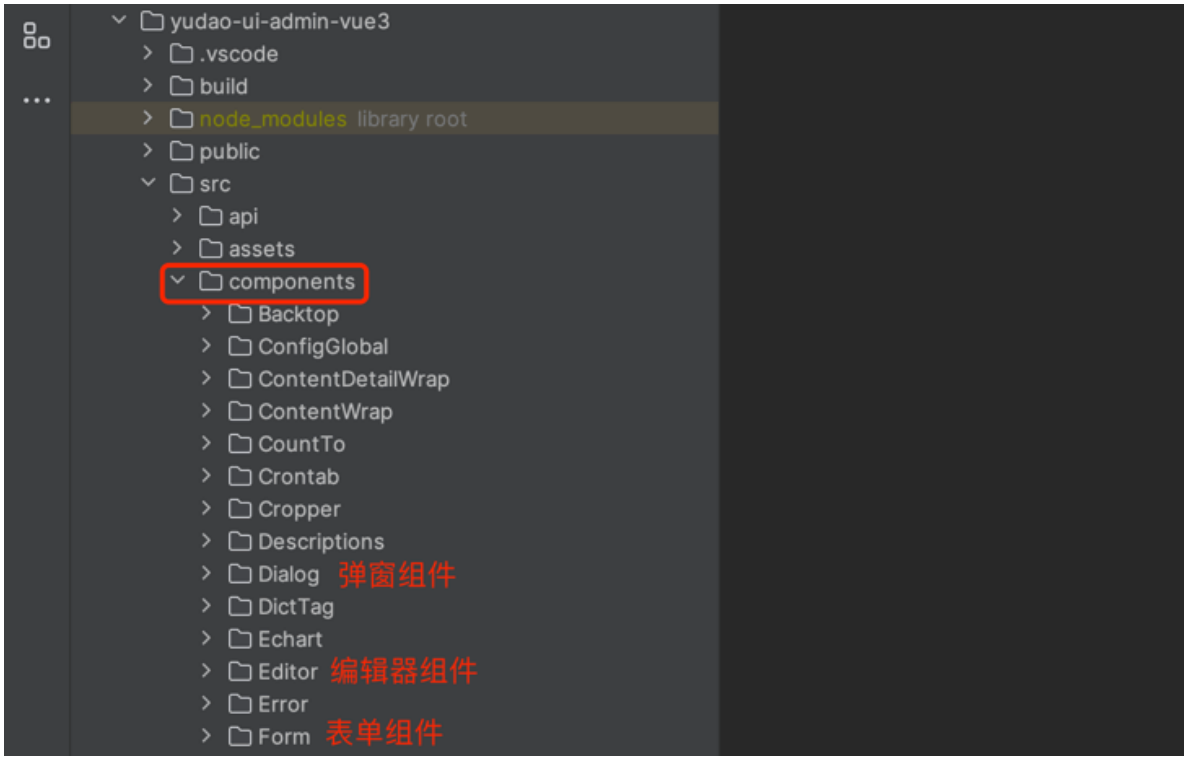
```vue
// ② views/system/post/index.vue
<script setup lang="tsx">
const loading = ref(true) // 列表的加载中
const total = ref(0) // 列表的总页数
const list = ref([]) // 列表的数据
const queryParams = reactive({
    pageNo: 1,
    pageSize: 10,
    code: '',
    name: '',
    status: undefined
})

/** 查询岗位列表 */
const getList = async () => {
    loading.value = true
    try {
        const data = await PostApi.getPostPage(queryParams)
        list.value = data.list
        total.value = data.total
    } finally {
        loading.value = false
    }
}
</script>
```

3. component 组件

3.1 全局组件

在 `@/components` (opens new window)目录下，实现**全局**组件，被所有模块所公用。

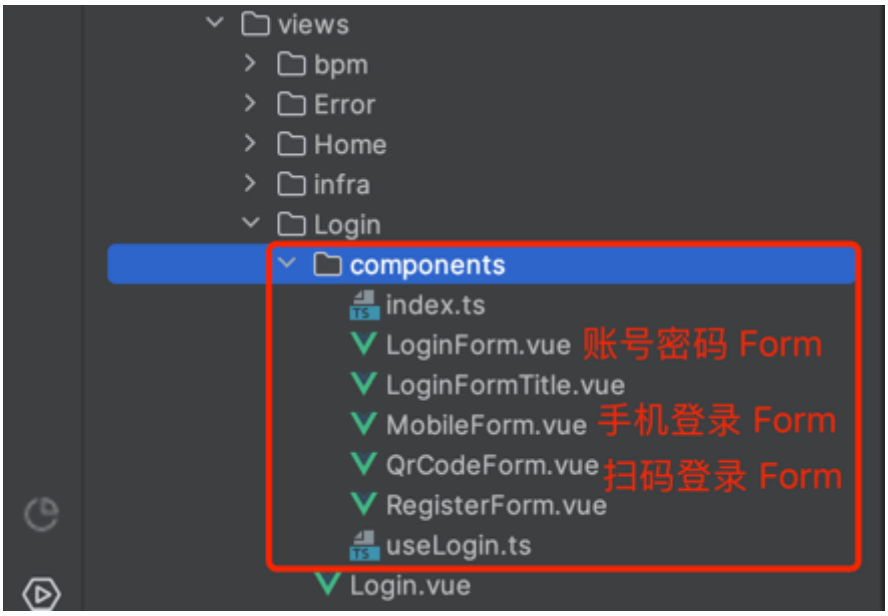
例如说，富文本编辑器、各种各搜索组件、封装的分页组件等等。



3.2 模块内组件

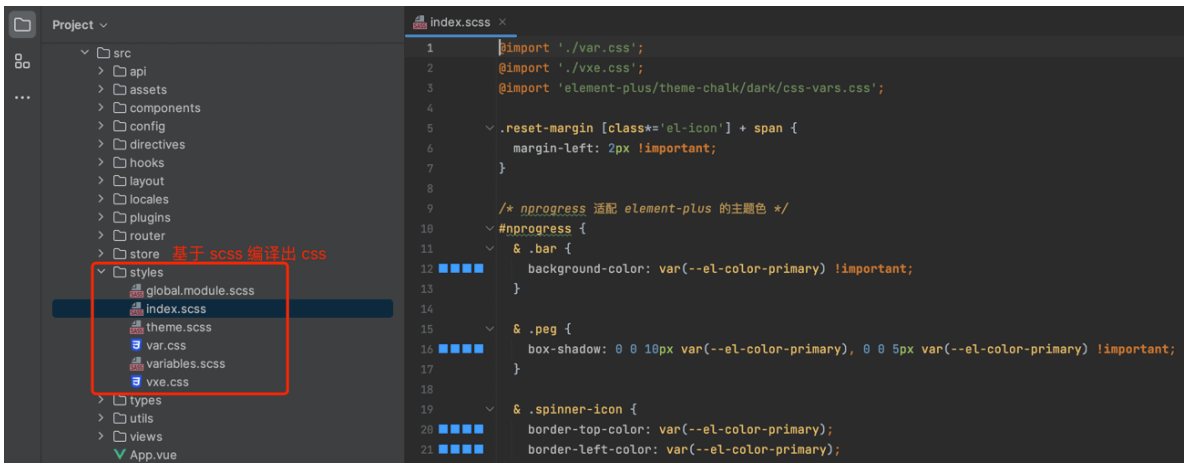
每个模块的业务组件，可实现在 `views` 目录下，自己模块的目录的 `components` 目录下，避免单个 `.vue` 文件过大，降低维护成功。

例如说，`@/views/pay/app/components/xxx.vue`：

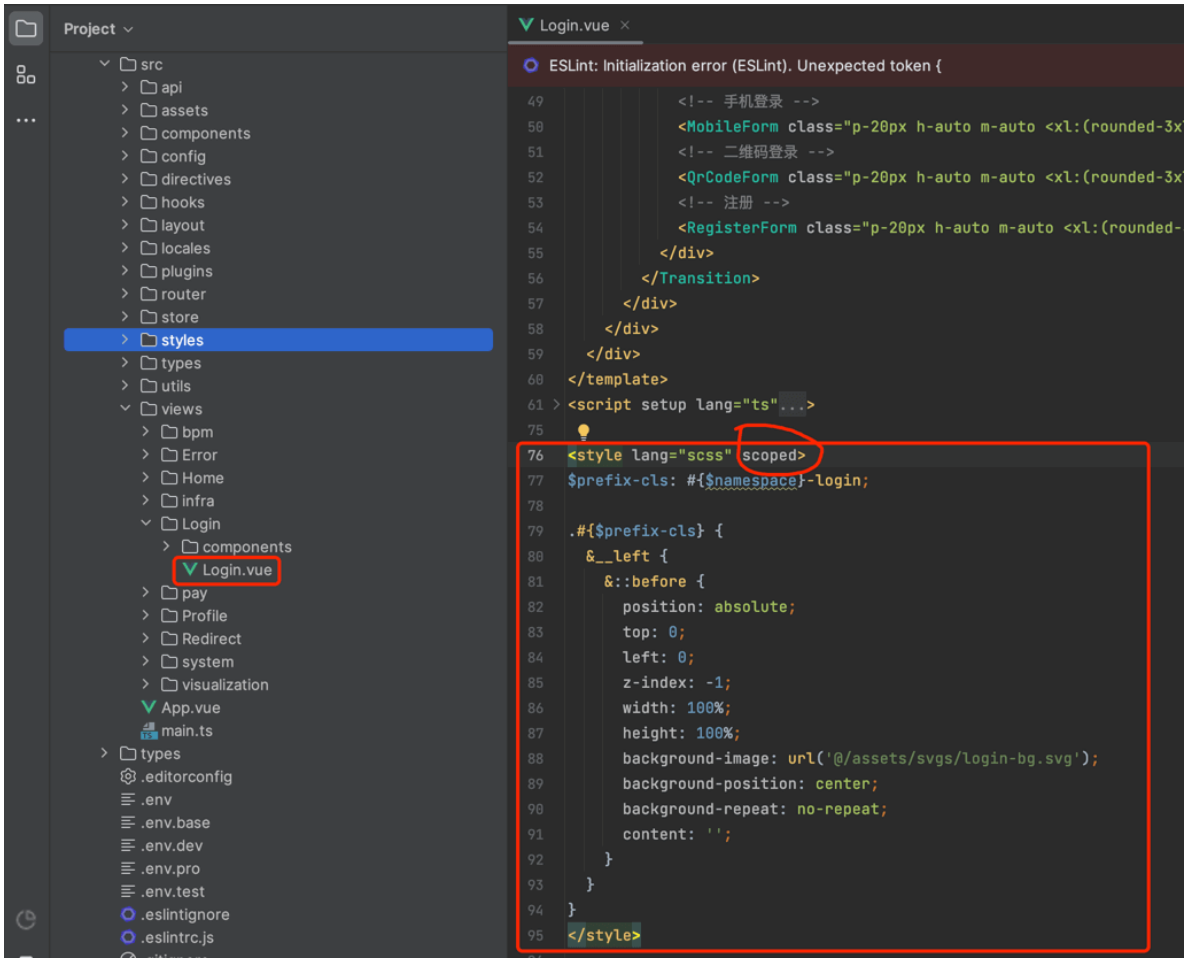


4. style 样式

① 在 `@/styles` (opens new window)目录下，实现**全局**样式，被所有页面所公用。



② 每个 `.vue` 页面，可在 ` 标签中添加样式，注意需要添加scoped ` 表示只作用在当前页面里，避免造成全局的样式污染。



更多也可以看看如下两篇文档：

- 《vue-element-plus-admin —— 项目配置「样式配置」》(opens new window)
- 《vue-element-plus-admin —— 样式》(opens new window)

5. 项目规范

可参考 《vue-element-plus-admin —— 项目规范》 (opens new window)文档。

菜单路由

前端项目基于 vue-element-plus-admin 实现，它的 路由和侧边栏 (opens new window)是组织起一个后台应用的关键骨架。

侧边栏和路由是绑定在一起的，所以你只有在 @/router/index.js (opens new window)下面配置对应的路由，侧边栏就能动态的生成了，大大减轻了手动重复编辑侧边栏的工作量。

当然，这样就需要在配置路由的时候，遵循一些约定的规则。

1. 路由配置

首先，我们了解一下本项目配置路由时，提供了哪些配置项：

```
/**
* redirect: noredirect           当设置 noredirect 的时候该路由在面包屑导航中不可被点击
* name:'router-name'             设定路由的名字，一定要填写不然使用<keep-alive>时会出现各种
问题
* meta : {
    hidden: true                 当设置 true 的时候该路由不会再侧边栏出现 如404，login等页
面(默认 false)
```

```
    alwaysShow: true               当你一个路由下面的 children 声明的路由大于1个时，自动会变成嵌套的模式，
                                   只有一个时，会将那个子路由当做根路由显示在侧边栏，
                                   若你想不管路由下面的 children 声明的个数都显示你的根路由，
                                   你可以设置 alwaysShow: true，这样它就会忽略之前定义的规则，
                                   一直显示根路由(默认 false)

    title: 'title'                 设置该路由在侧边栏和面包屑中展示的名字

    icon: 'svg-name'               设置该路由的图标

    noCache: true                  如果设置为true，则不会被 <keep-alive> 缓存(默认 false)

    breadcrumb: false              如果设置为false，则不会在breadcrumb面包屑中显示(默认 true)

    affix: true                    如果设置为true，则会一直固定在tag项中(默认 false)

    noTagsView: true               如果设置为true，则不会出现在tag中(默认 false)

    activeMenu: '/dashboard'       显示高亮的路由路径

    followAuth: '/dashboard'       跟随哪个路由进行权限过滤

    canTo: true                    设置为true即使hidden为true，也依然可以进行路由跳转(默认 false)
  }
**/
```

1.1 普通示例

注意事项：

- 整个项目所有路由 `name` 不能重复
- 所有的多级路由最终都会转成二级路由，所以不能内嵌子路由
- 除了 `layout` 对应的 `path` 前面需要加 `/`，其余子路由都不要以 `/` 开头

```
{
  path: '/level',
  component: Layout,
  redirect: '/level/menu1/menu1-1/menu1-1-1',
  name: 'Level',
  meta: {
    title: t('router.level'),
    icon: 'carbon:skill-level-advanced'
  },
  children: [
    {
      path: 'menu1',
      name: 'Menu1',
      component: getParentLayout(),
      redirect: '/level/menu1/menu1-1/menu1-1-1',
      meta: {
        title: t('router.menu1')
      },
```

```
      children: [
        {
          path: 'menu1-1',
          name: 'Menu11',
          component: getParentLayout(),
          redirect: '/level/menu1/menu1-1/menu1-1-1',
          meta: {
            title: t('router.menu11'),
            alwaysShow: true
          },
          children: [
            {
              path: 'menu1-1-1',
              name: 'Menu111',
              component: () => import('@/views/Level/Menu111.vue'),
              meta: {
                title: t('router.menu111')
              }
            }
          ]
        },
        {
          path: 'menu1-2',
          name: 'Menu12',
          component: () => import('@/views/Level/Menu12.vue'),
          meta: {
            title: t('router.menu12')
          }
        }
      ]
    },
    {
      path: 'menu2',
      name: 'Menu2Demo',
      component: () => import('@/views/Level/Menu2.vue'),
      meta: {
        title: t('router.menu2')
      }
    }
  ]
}
```

1.2 外链示例

只需要将 `path` 设置为需要跳转的 HTTP 地址即可。

```
{
  path: '/external-link',
  component: Layout,
  meta: {
    name: 'ExternalLink'
  },
  children: [
    {
      path: 'https://www.iocoder.cn',
      meta: { name: 'Link', title: '芋道源码' }
    }
  ]
}
```

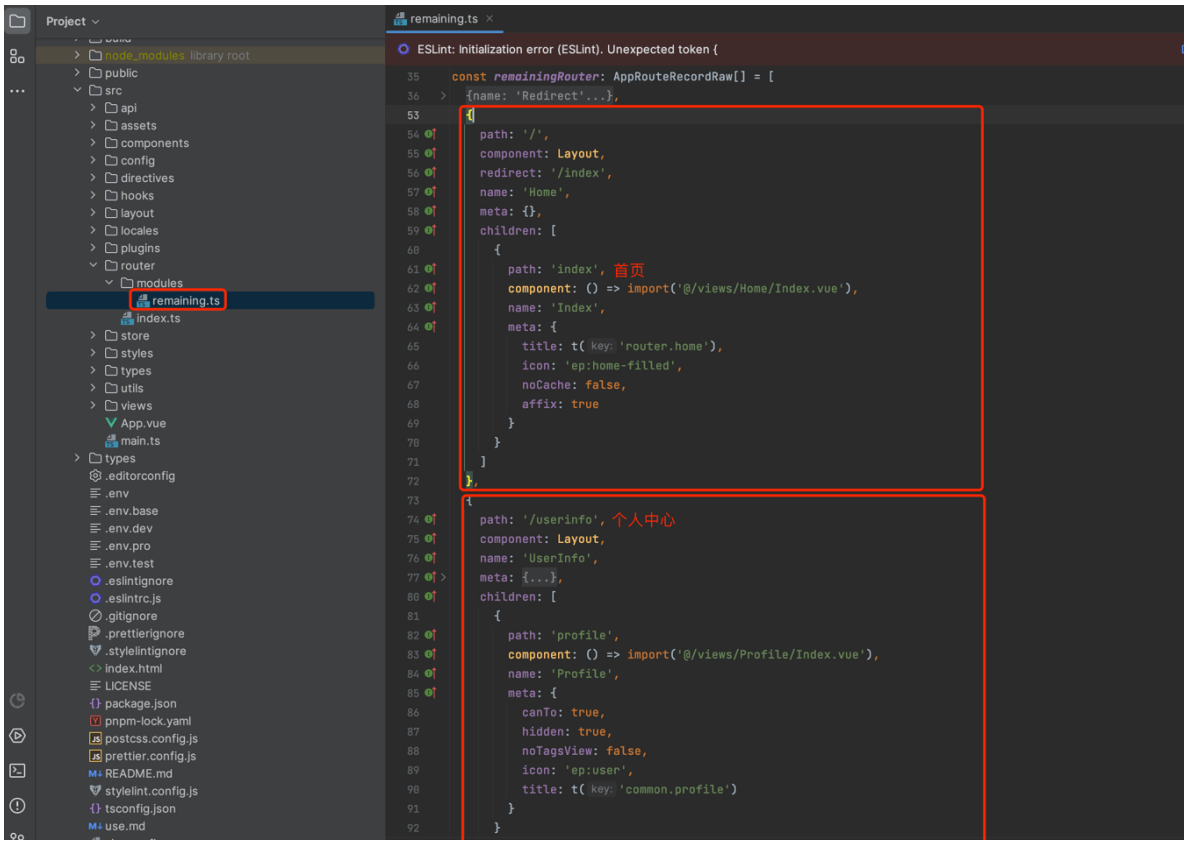
2. 路由

项目的路由分为两种：静态路由、动态路由。

2.1 静态路由

静态路由，代表那些不需要动态判断权限的路由，如登录页、404、个人中心等通用页面。

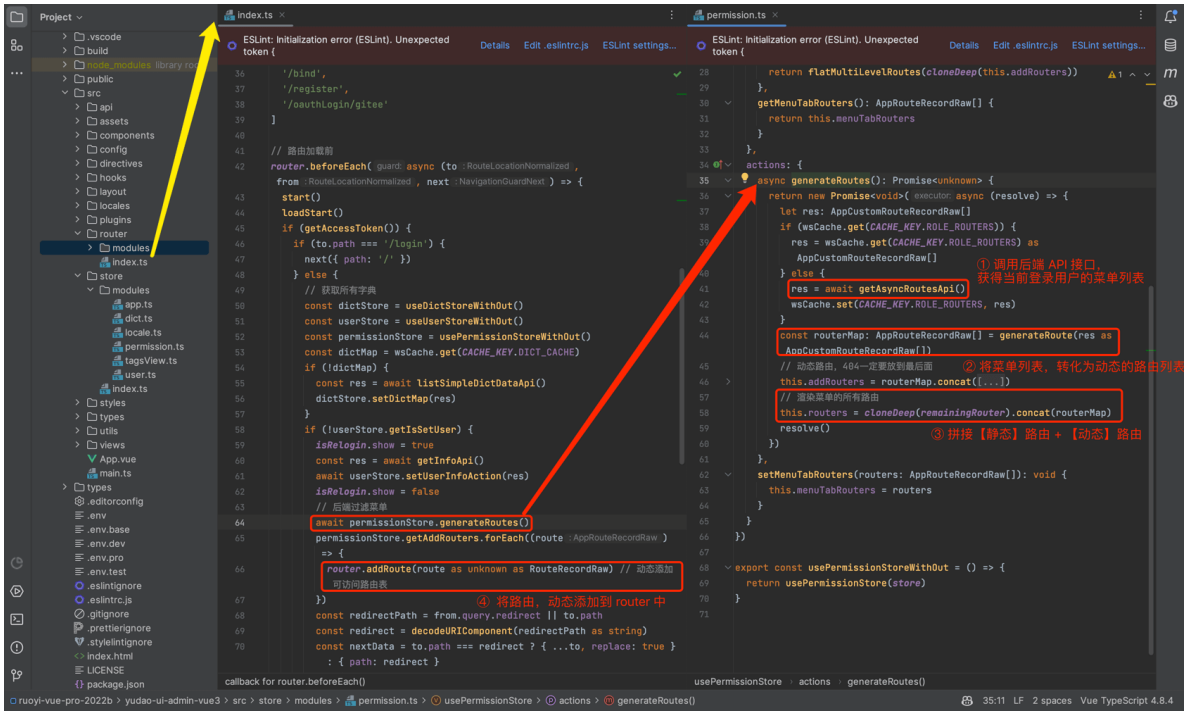
在 @/router/modules/remaining.ts (opens new window)的 `remainingRouter`，就是配置对应的公共路由。如下图所示：



2.2 动态路由

动态路由，代表那些需要根据用户动态判断权限，并通过 addRoutes (opens new window)动态添加的页面，如用户管理、角色管理等功能页面。

在用户登录成功后，会触发 `@/store/modules/permission.ts` (opens new window)请求后端的菜单 RESTful API 接口，获取用户**有权限**的菜单列表，并转化添加到路由中。如下图所示：



友情提示：

1. 动态路由可以在 [系统管理 -> 菜单管理] 进行新增和修改操作，请求的后端 RESTful API 接口是 `/admin-api/system/auth/get-permission-info` (opens new window)
2. 动态路由在生产环境下会默认使用路由懒加载，实现方式参考 import.meta.glob('../views/*/.{vue,tsx}') (opens new window)方法的判断

补充说明：

最新的代码，部分逻辑重构到 `@/permission.ts` (opens new window)

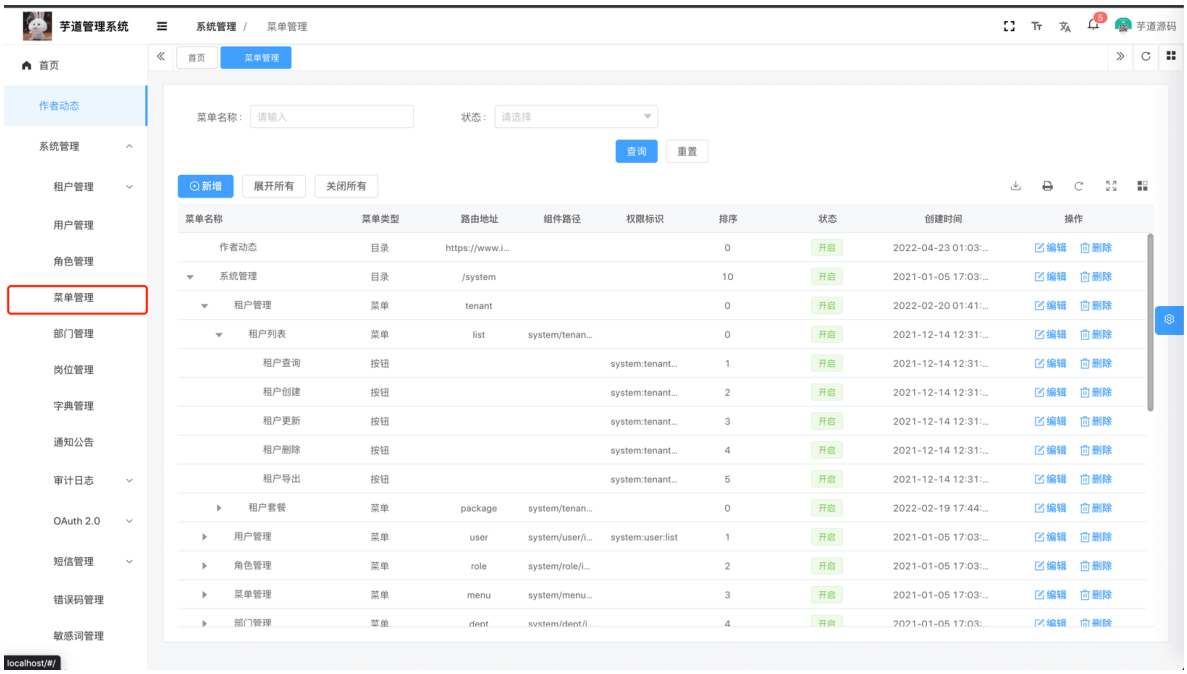
2.3 路由跳转

使用 `router.push` 方法，可以实现跳转到不同的页面。

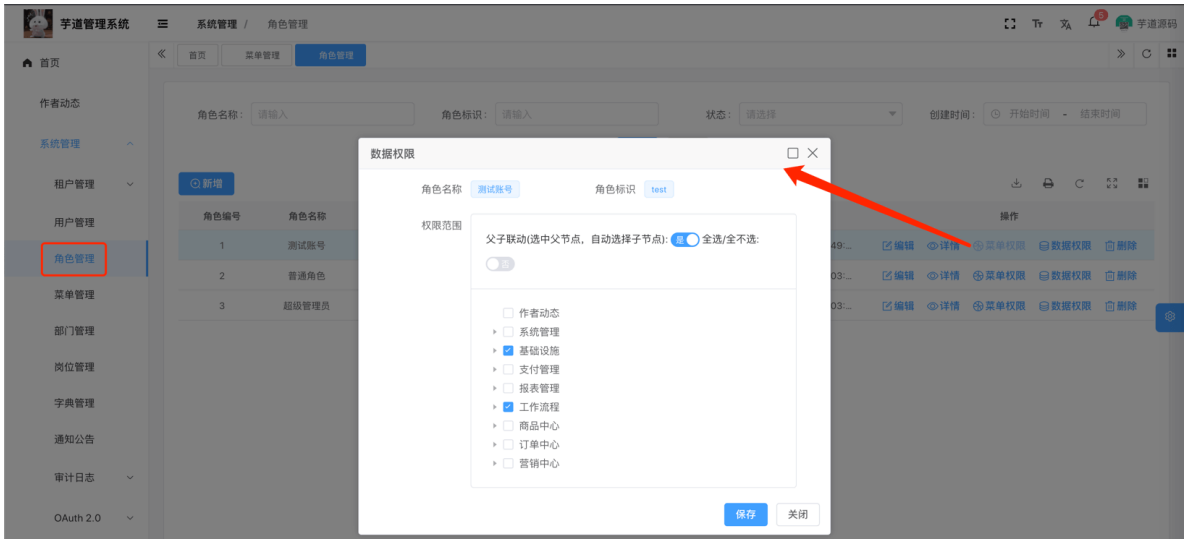
```
const { push } = useRouter()

// 简单跳转
push('/job/job-log');

// 跳转页面并设置请求参数，使用 `query` 属性
push('/bpm/process-instance/detail?id=' + row.processInstance.id)
```

3. 菜单管理

项目的菜单在 [系统管理 -> 菜单管理] 进行管理，支持**无限**层级，提供目录、菜单、按钮三种类型。如下图所示：

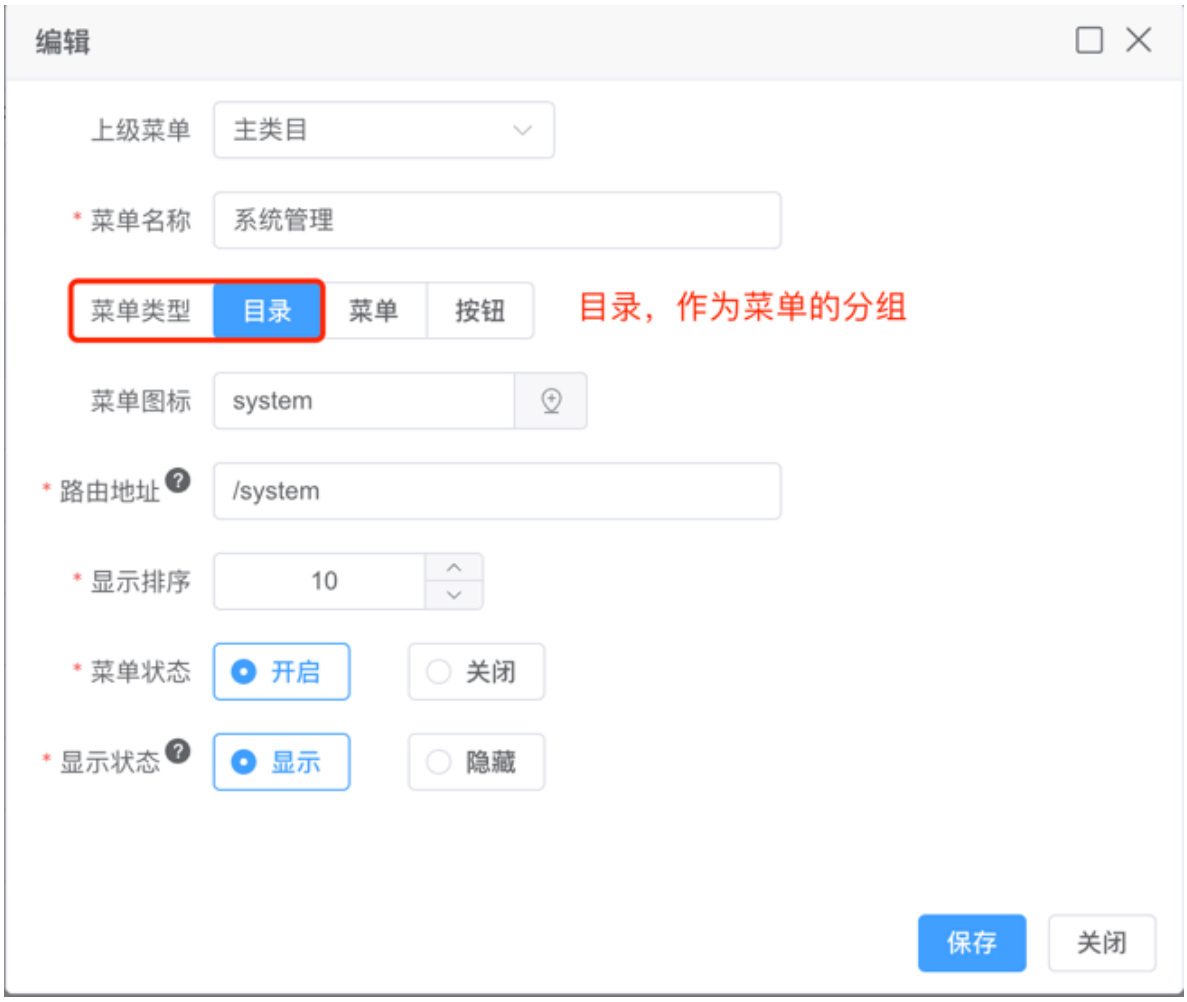


菜单可在 [系统管理 -> 角色管理] 被分配给角色。如下图所示：

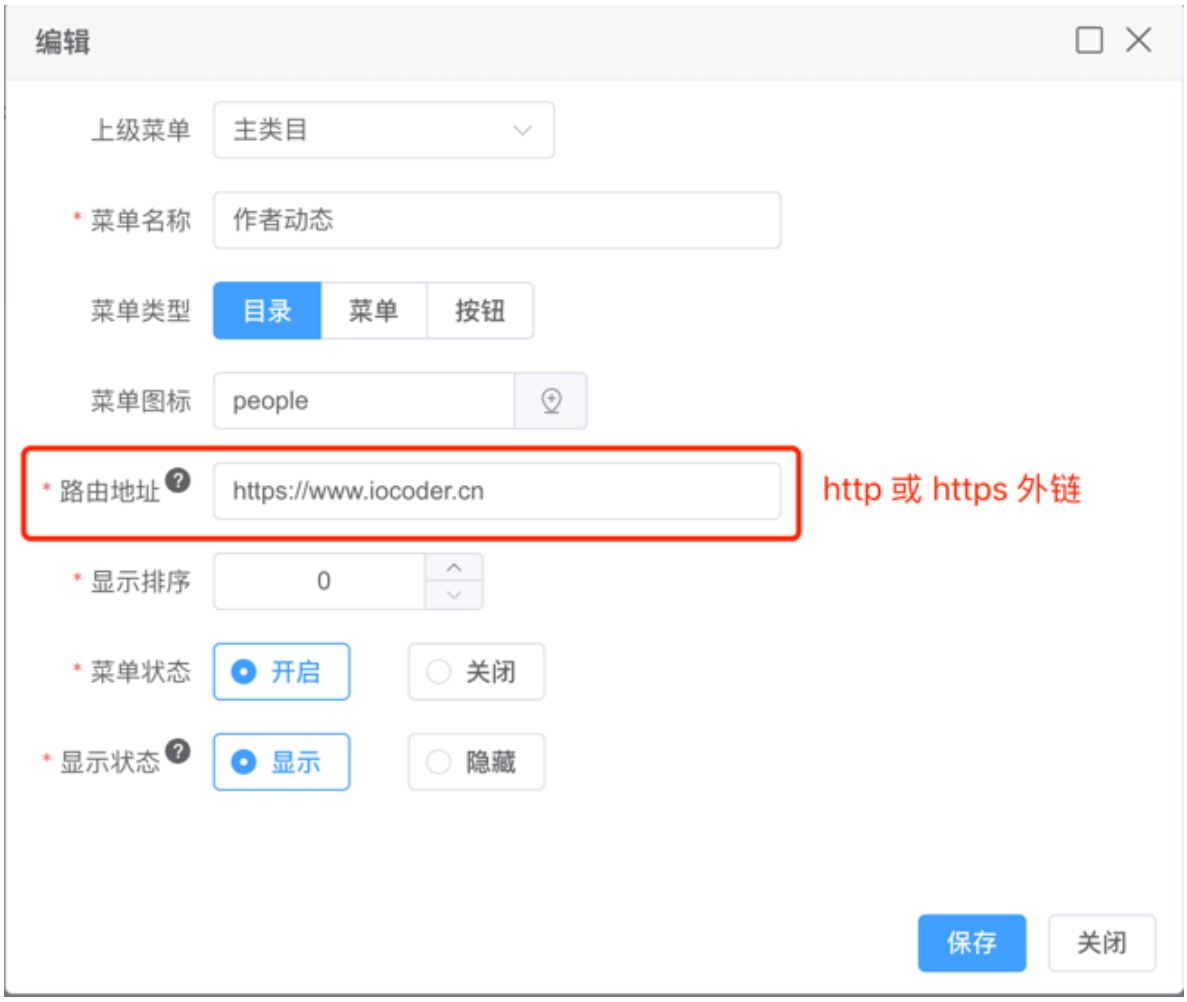


3.1 新增目录

① 大多数情况下，目录是作为菜单的【分类】：

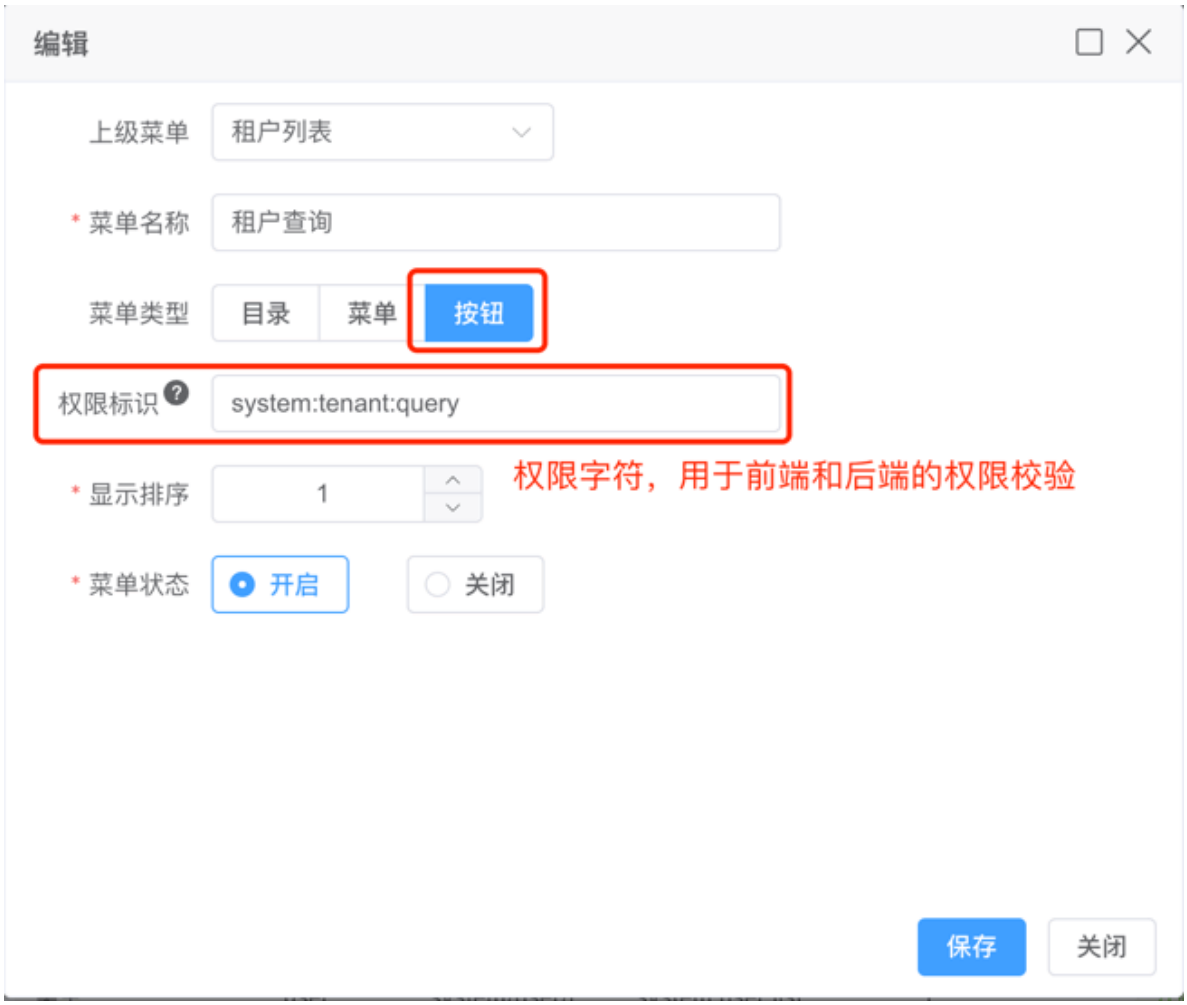


② 目录也提供实现【外链】的能力：



3.2 新增菜单

3.3 新增按钮



4. 权限控制

前端通过权限控制，隐藏用户没有权限的按钮等，实现功能级别的权限。

友情提示：前端的权限控制，主要是提升用户体验，避免操作后发现没有权限。

最终在请求到后端时，还是会进行一次权限的校验。

4.1 v-hasPermi 指令

`v-hasPermi` (opens new window)指令，基于权限字符，进行权限的控制。

```
<!-- 单个 -->
<el-button v-hasPermi="['system:user:create']">存在权限字符串才能看到</el-button>

<!-- 多个，满足任一一个即可 -->
<el-button v-hasPermi="['system:user:create', 'system:user:update']">包含权限字符串才能看到</el-button>
```

4.2 v-hasRole 指令

`v-hasRole` (opens new window)指令，基于角色标识，机进行的控制。

```
<!-- 单个 -->
<el-button v-hasRole="['admin']">管理员才能看到</el-button>

<!-- 多个，满足任一一个即可 -->
<el-button v-hasRole="['role1', 'role2']">包含角色才能看到</el-button>
```

4.3 结合 v-if 指令

在某些情况下，它是不适合使用 `v-hasPermi` 或 `v-hasRole` 指令，如元素标签组件。此时，只能通过手动设置 `v-if`，通过使用全局权限判断函数，用法是基本一致的。

```
<template>
  <el-tabs>
    <el-tab-pane v-if="checkPermi(['system:user:create'])" label="用户管理"
name="user">用户管理</el-tab-pane>
    <el-tab-pane v-if="checkPermi(['system:user:create', 'system:user:update'])"
label="参数管理" name="menu">参数管理</el-tab-pane>
    <el-tab-pane v-if="checkRole(['admin'])" label="角色管理" name="role">角色管理
</el-tab-pane>
    <el-tab-pane v-if="checkRole(['admin','common'])" label="定时任务"
name="job">定时任务</el-tab-pane>
   </el-tabs>
</template>

<script>
import { checkPermi, checkRole } from "@/utils/permission"; // 权限判断函数

export default{
   methods: {
    checkPermi,
    checkRole
  }
}
</script>
```

5. 页面缓存

开启缓存有 2 个条件

- 路由设置 `name`，且不能重复
- 路由对应的组件加上 `name`，与路由设置的 `name` 保持一致

友情提示：页面缓存是什么?

简单来说，Tab 切换时，开启页面缓存的 Tab 保持原本的状态，不进行刷新。

详细可见 Vue 文档 —— KeepAlive(opens new window)

5.1 静态路由的示例

① router 路由的 `name` 声明如下：

```
{
    path: 'menu2',
    name: 'Menu2',
    component: () => import('@/views/Level/Menu2.vue'),
    meta: {
        title: t('router.menu2')
    }
}
```

② view component 的 `name` 声明如下：

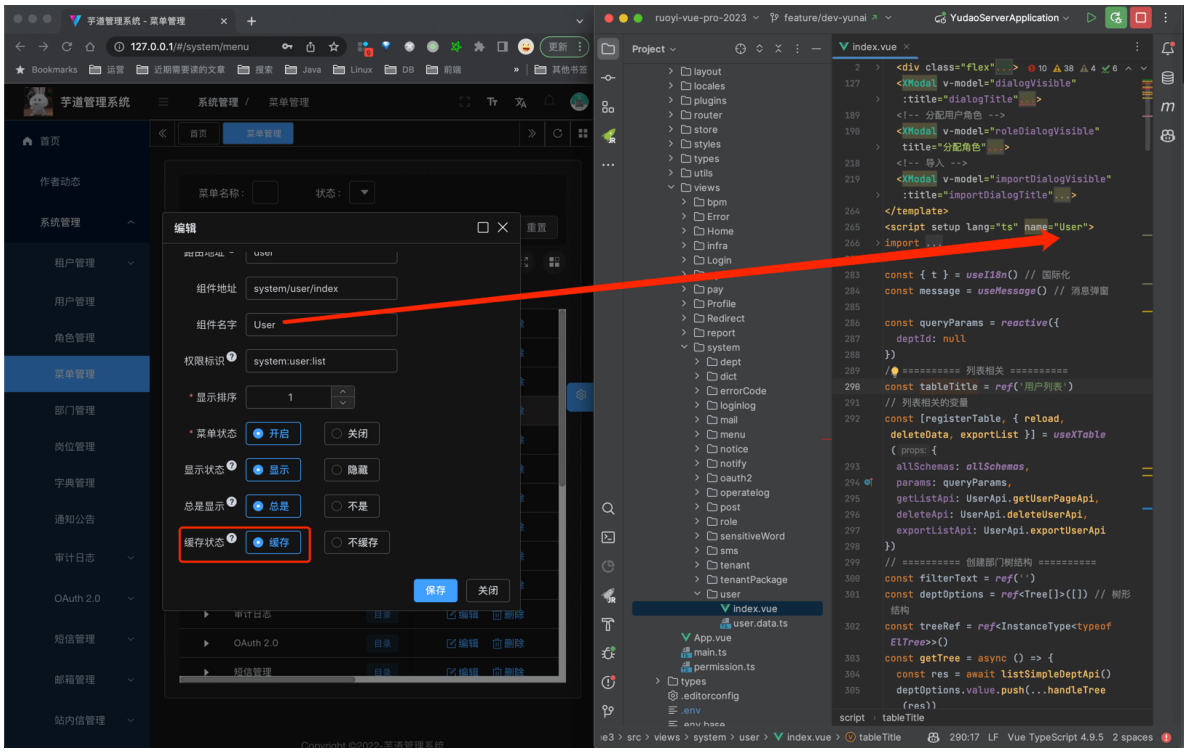
```
<script setup lang="ts">
    defineOptions({
    name: 'Menu2'
})
</script>
```

注意：

`keep-alive` 生效的前提是：需要将路由的 `name` 属性及对应的页面的 `name` 设置成一样。

因为：`include -` 字符串或正则表达式，只有名称匹配的组件会被缓存

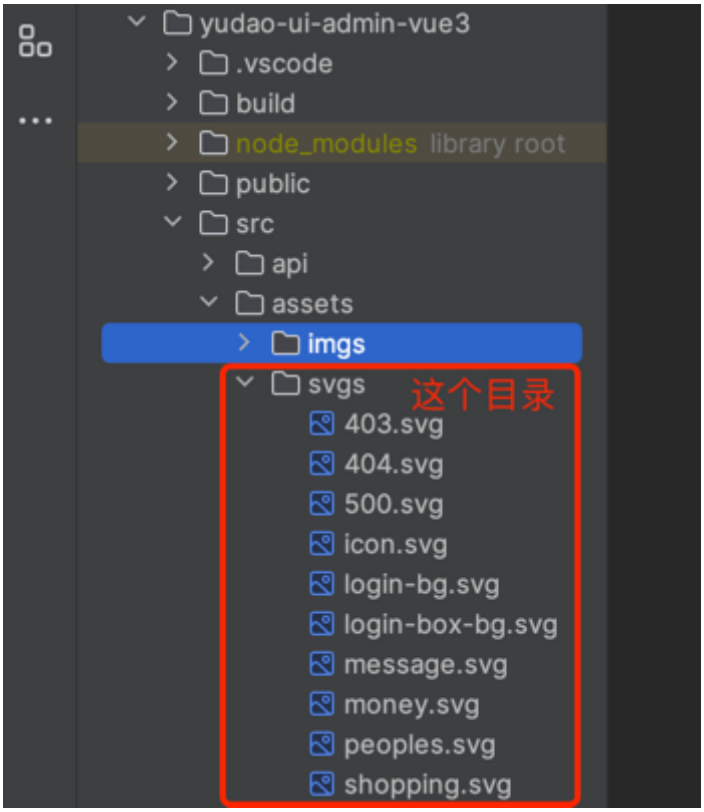
5.2 动态路由的示例



Icon图标

Element Plus 内置多种 Icon 图标，可参考 Element Plus —— Icon 图标 (opens new window)的文档。

在项目的 `/src/assets/svgs` (opens new window)目录下，自定义了 Icon 图标，默认注册到全局中，可以在项目中任意地方使用。如下图所示：



1. Icon 图标组件

友情提示：

该小节，基于 《vue element plus admin —— Icon 图标组件 》 (opens new window)的内容修改。

Icon 组件位于 src/components/Icon (opens new window)内，用于项目内组件的展示，基本支持所有图标库（支持按需加载，只打包所用到的图标），支持使用本地 svg 和 Iconify (opens new window)图标。

提示

在 Iconify (opens new window)上，你可以查询到你想要的所有图标并使用，不管是不是 `element-plus` 的图标库。

1.1 基本用法

如果以 `svg-icon:` 开头，则会在本地中找到该 `svg` 图标，否则，会加载 `Iconify` 图标。代码如下：

```
<template>
  <!-- 加载本地 svg -->
  <Icon icon="svg-icon:peoples" />

  <!-- 加载 Iconify -->
  <Icon icon="ep:aim" />
</template>
```

1.2 useIcon

如果需要在其他组件中如 ElButton 传入 `icon` 属性，可以使用 `useIcon`。代码如下：

```vue
<script setup lang="ts">
import { useIcon } from '@/hooks/web/useIcon'
import { ElButton } from 'element-plus'

const icon = useIcon({ icon: 'svg-icon:save' })
</script>

<template>
  <ElButton :icon="icon"> button </ElButton>
</template>
```

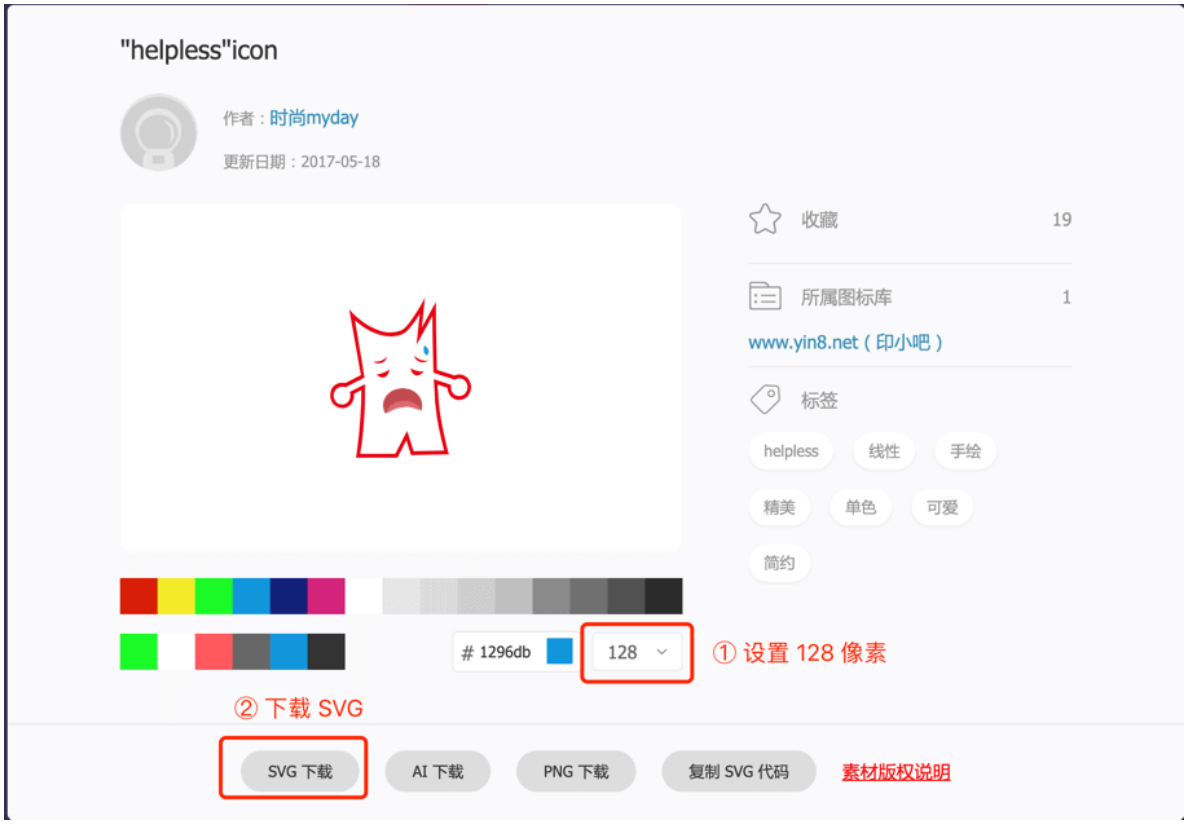
`useIcon` 的 **props** 属性如下：

属性	说明	类型	可选值	默认值
icon	图标名	`string`	-	-
color	图标颜色	`string`	-	-
size	图标大小	`number`	-	16

2. 自定义图标

① 访问 https://www.iconfont.cn/ (opens new window)地址，搜索你想要的图标，下载 SVG 格式。如下图所示：

友情提示：其它 SVG 图标网站也可以。

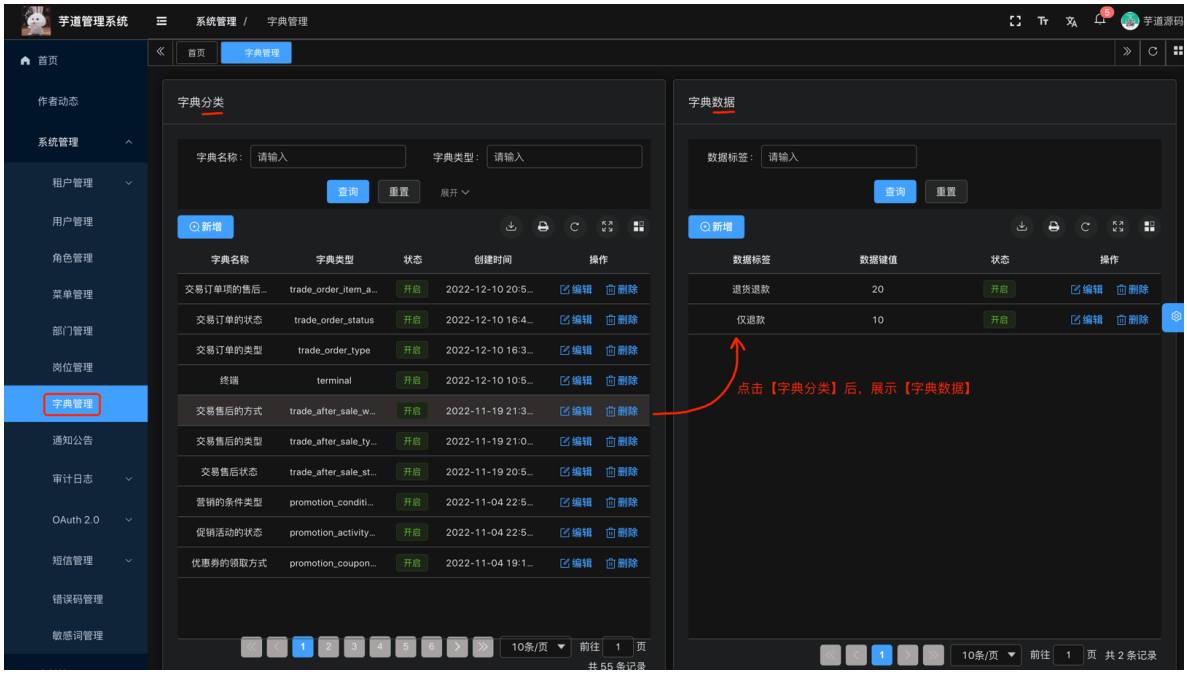


② 将 SVG 图标添加到 `/src/assets/svgs` (opens new window)目录下，然后进行使用。

```vue
<Icon icon="svg-icon:helpless" />
```

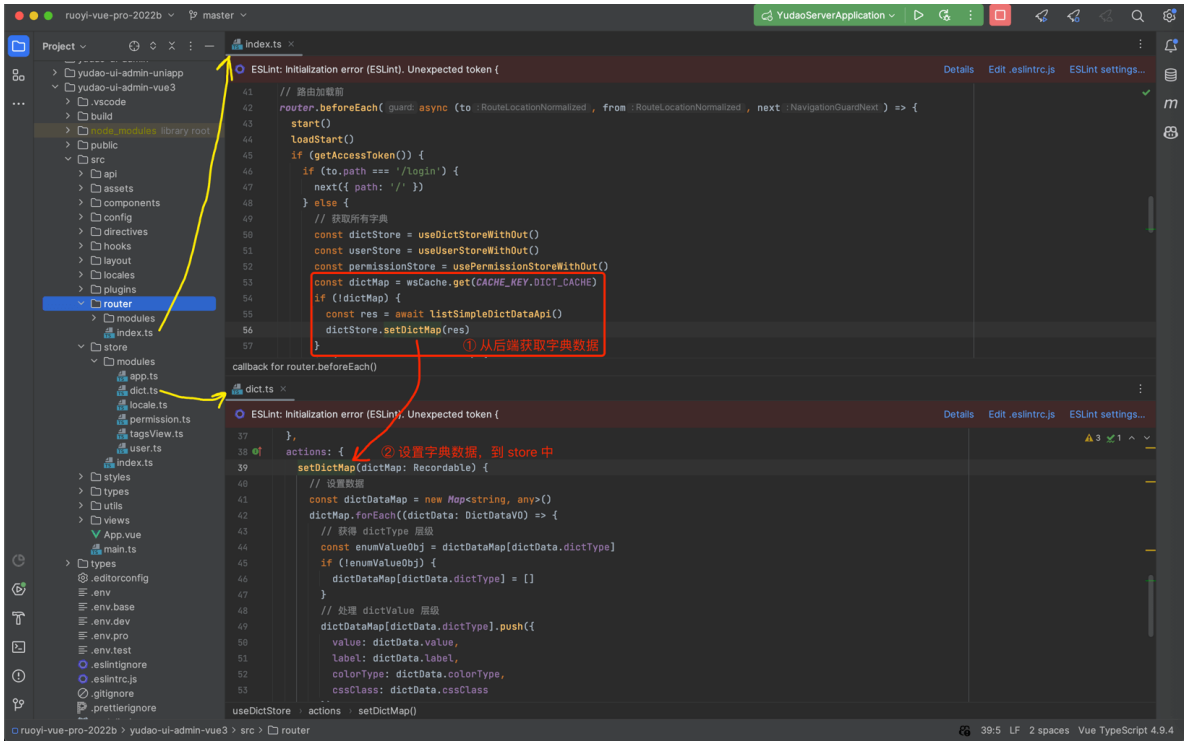
字典数据

本小节，讲解前端如何使用 [系统管理 -> 字典管理] 菜单的字典数据，例如说字典数据的下拉框、单选 / 多选按钮、高亮展示等等。



1. 全局缓存

用户登录成功后，前端会从后端获取到全量的字典数据，缓存在 store 中。如下图所示：

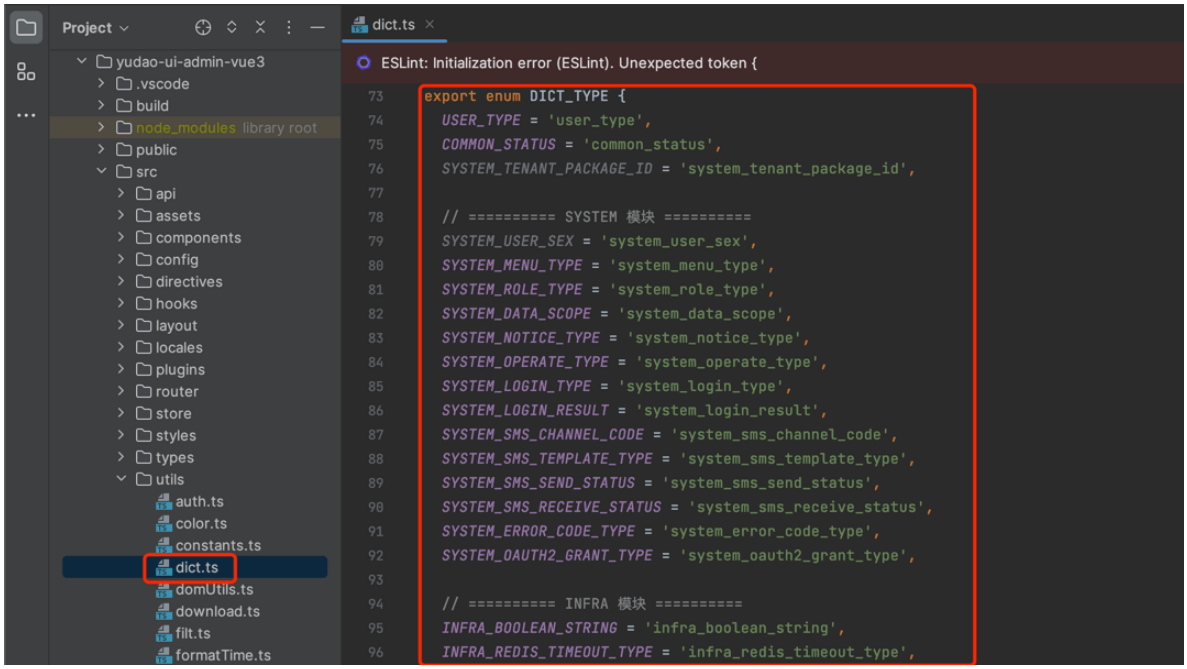


这样，前端在使用到字典数据时，无需重复请求后端，提升用户体验。

不过，缓存暂时未提供刷新，所以在字典数据发生变化时，需要用户刷新浏览器，进行重新加载。

2. DICT_TYPE

在 `dict.ts` (opens new window)文件中，使用 `DICT_TYPE` 枚举了字典的 KEY。如下图所示：

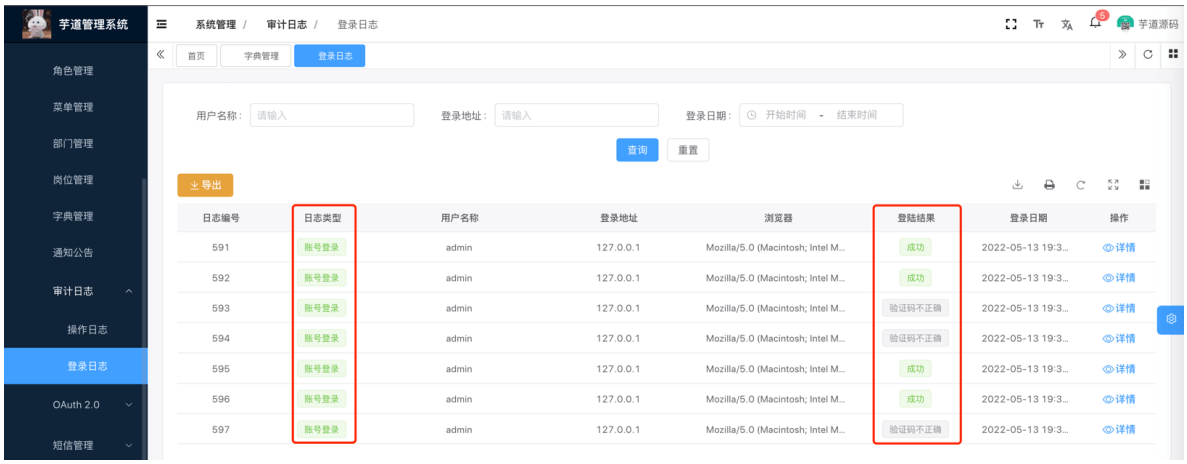


后续如果有新的字典 KEY，需要你自己进行添加。

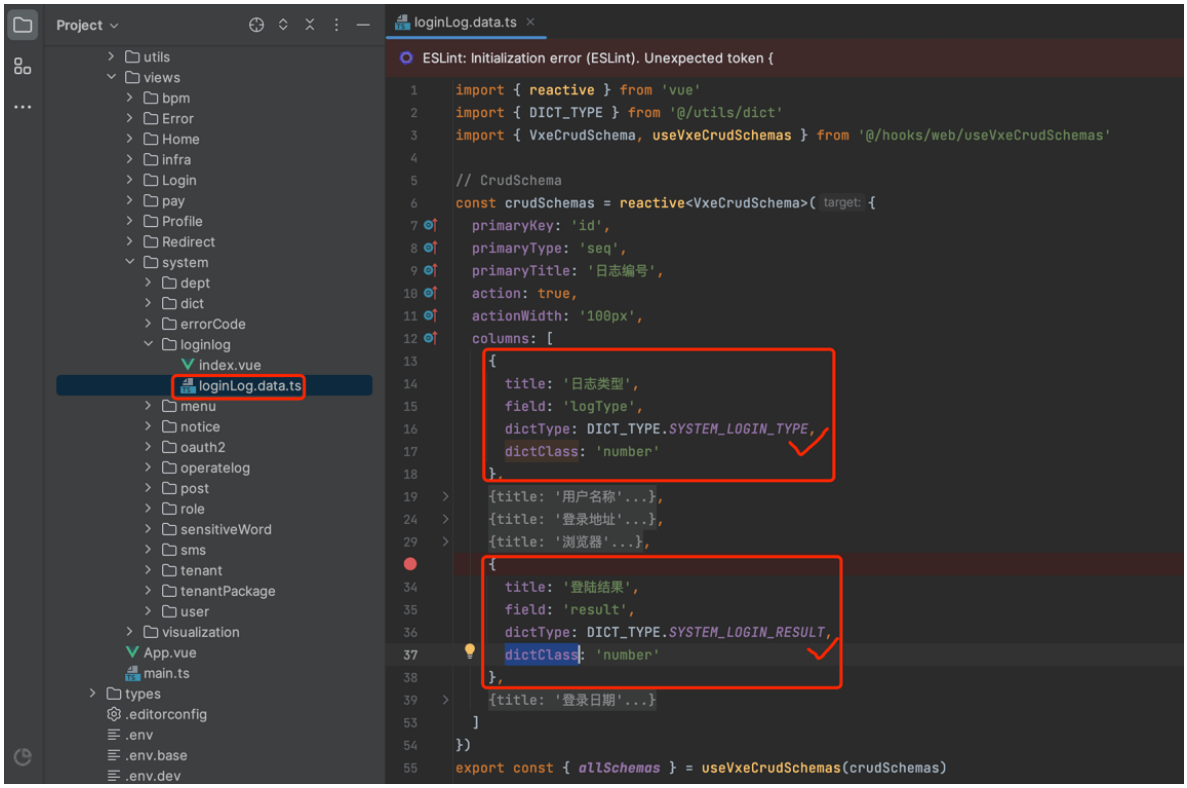
3. DictTag 字典标签

`` (opens new window)组件，翻译字段对应的字典展示文本，并根据 `colorType`、`cssClass` 进行高亮。使用示例如下：

```
<!--
    type: 字典 KEY
    value: 字典值
-->
<dict-tag :type="DICT_TYPE.SYSTEM_LOGIN_TYPE" :value="row.logType" />
```



【推荐】注意，一般情况下使用 CRUD schemas 方式，不需要直接使用 `，而是通过 columns 的 dictType 和 dictClass` 属性即可。如下图所示：



4. 字典工具类

在 `dict.ts` (opens new window)文件中，提供了字典工具类，方法如下：

```
// 获取 dictType 对应的数据字典数组【object】
export const getDictOptions = (dictType: string) => {{ /** 省略代码 */ }

// 获取 dictType 对应的数据字典数组【int】
export const getIntDictOptions = (dictType: string) => { /** 省略代码 */ }

// 获取 dictType 对应的数据字典数组【string】
export const getStrDictOptions = (dictType: string) => { /** 省略代码 */ }

// 获取 dictType 对应的数据字典数组【boolean】
export const getBoolDictOptions = (dictType: string) => { /** 省略代码 */ }

// 获取 dictType 对应的数据字典数组【object】
export const getDictObj = (dictType: string, value: any) => { /** 省略代码 */ }
```

结合 Element Plus 的表单组件，使用示例如下：

```
<template>
  <!-- radio 单选框 -->
  <el-radio
    v-for="dict in getIntDictOptions(DICT_TYPE.COMMON_STATUS)"
    :key="dict.value"
    :label="parseInt(dict.value)"
  >
    {{dict.label}}
  </el-radio>

  <!-- select 下拉框 -->
  <el-select v-model="form.code" placeholder="请选择渠道编码" clearable>
    <el-option
```

```
      v-for="dict in getStrDictOptions(DICT_TYPE.SYSTEM_SMS_CHANNEL_CODE)"
      :key="dict.value"
      :label="dict.label"
      :value="dict.value"
    />
  </el-select>
</template>
<script setup lang="tsx">
import { DICT_TYPE, getIntDictOptions, getStrDictOptions } from '@/utils/dict'
</script>
```

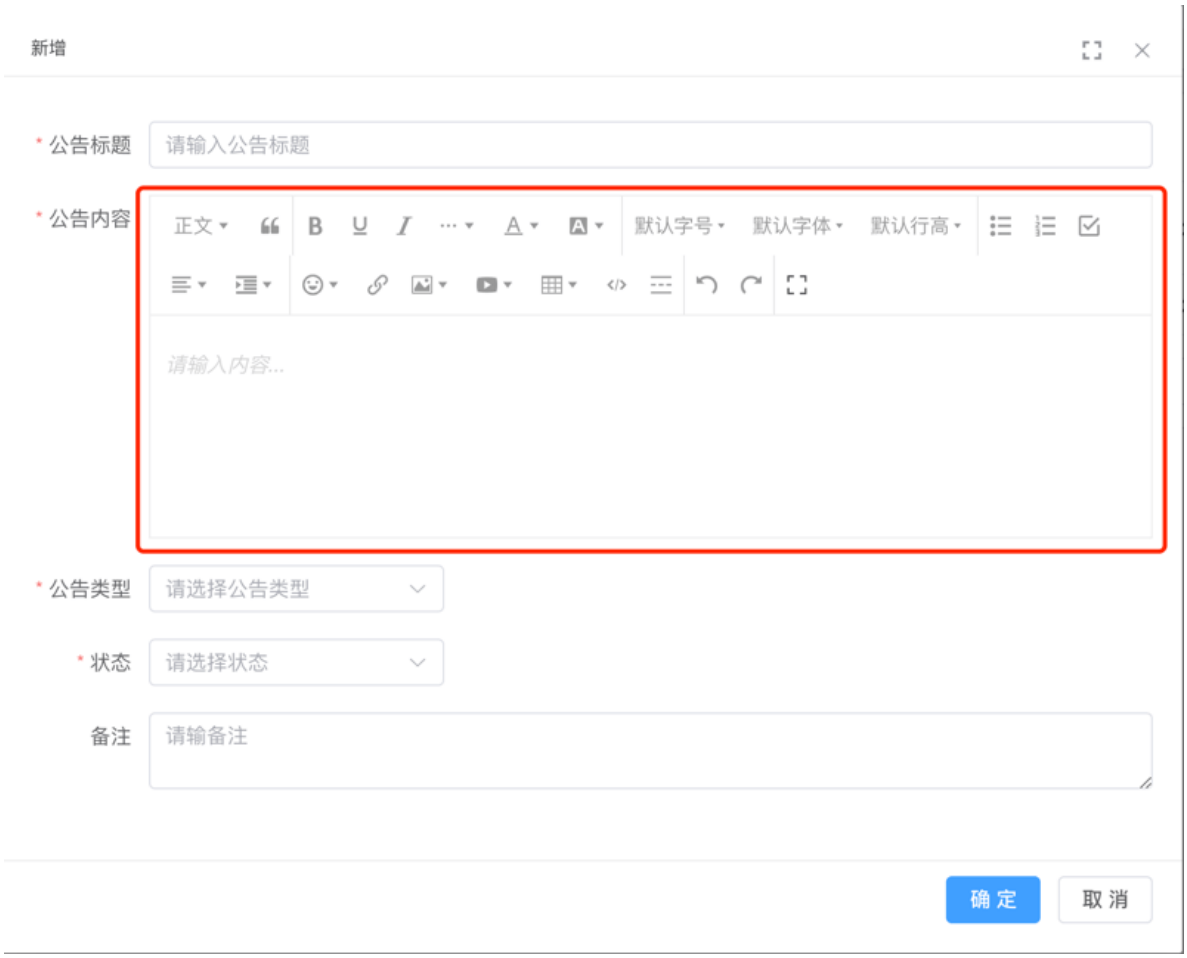
系统组件

1. 常用组件

1.1 Editor 富文本组件

基于 wangEditor (opens new window)封装

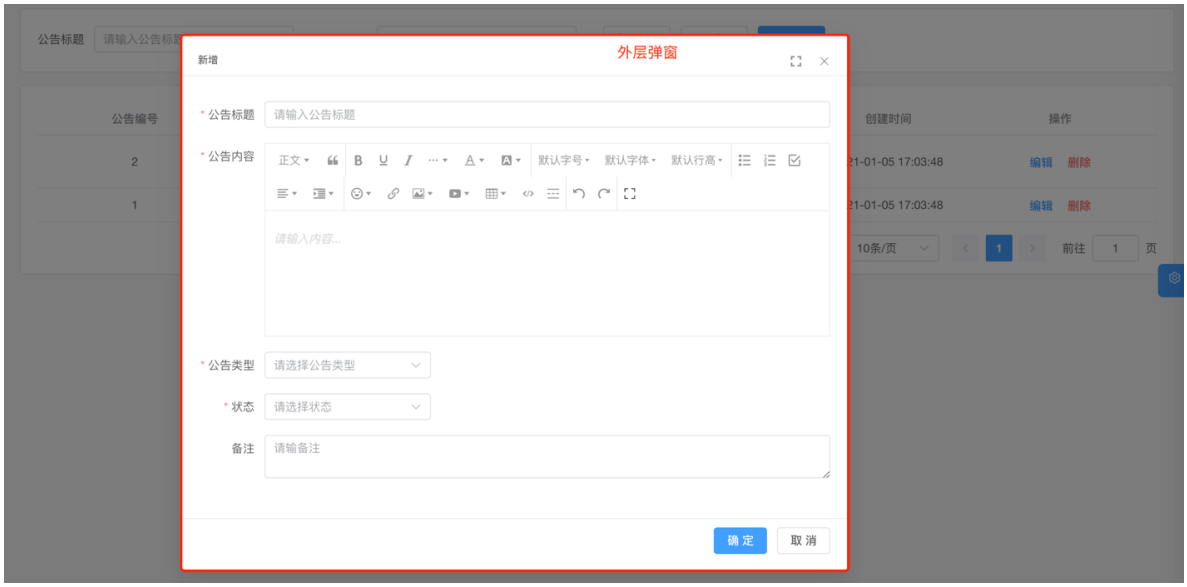
- Editor 组件：位于 src/components/Editor (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/editor.html(opens new window)
- 实战案例：src/views/system/notice/form.vue (opens new window)TODO



1.2 Dialog 弹窗组件

对 Element Plus 的 Dialog 组件进行封装，支持最大化、最大高度等特性

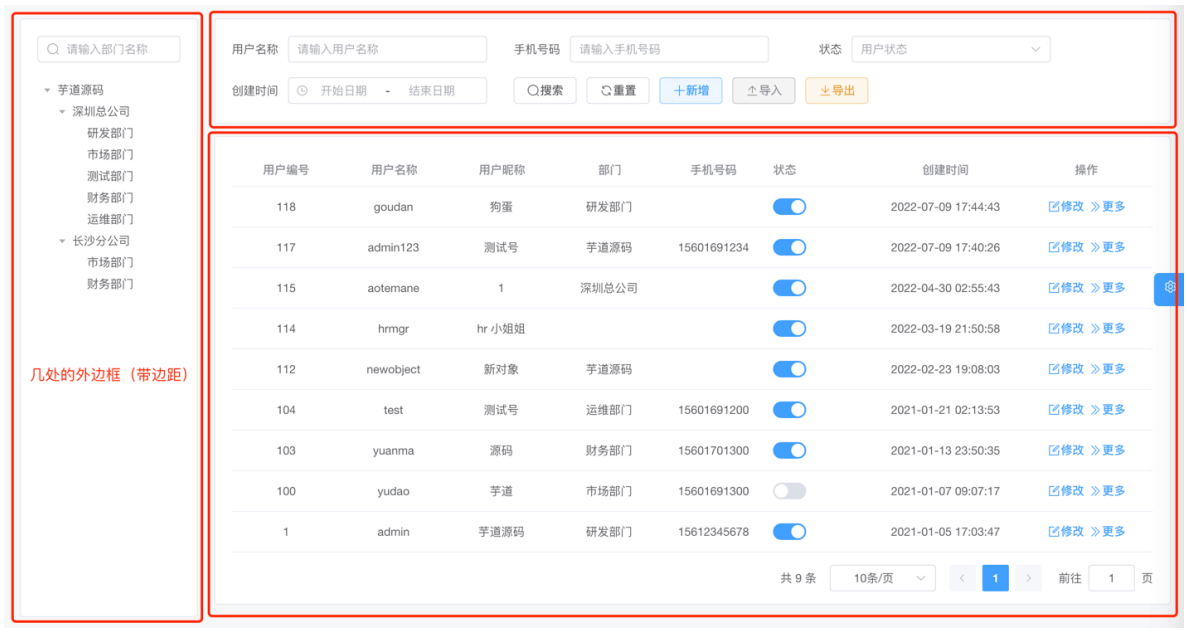
- Dialog 组件：位于 src/components/Dialog (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/dialog.html(opens new window)
- 实战案例：src/views/system/dept/DeptForm.vue(opens new window)



1.3 ContentWrap 包裹组件

对 Element Plus 的 ElCard 组件进行封装，自带标题、边距

- ContentWrap 组件：位于 src/components/ContentWrap (opens new window)内
- 实战案例：src/views/system/post/index.vue(opens new window)



1.4 Pagination 分页组件

对 Element Plus 的 Pagination (opens new window)组件进行封装

- Pagination 组件：位于 src/components/Pagination (opens new window)内
- 实战案例：src/views/system/post/index.vue(opens new window)

1.5 UploadFile 上传文件组件

对 Element Plus 的 Upload (opens new window)组件进行封装，上传文件到文件服务

- UploadFile 组件：位于 src/components/UploadFile/src/UploadFile.vue (opens new window)内
- 实战案例：暂无

1.6 UploadImg 上传图片组件

对 Element Plus 的 Upload (opens new window)组件进行封装，上传图片到文件服务

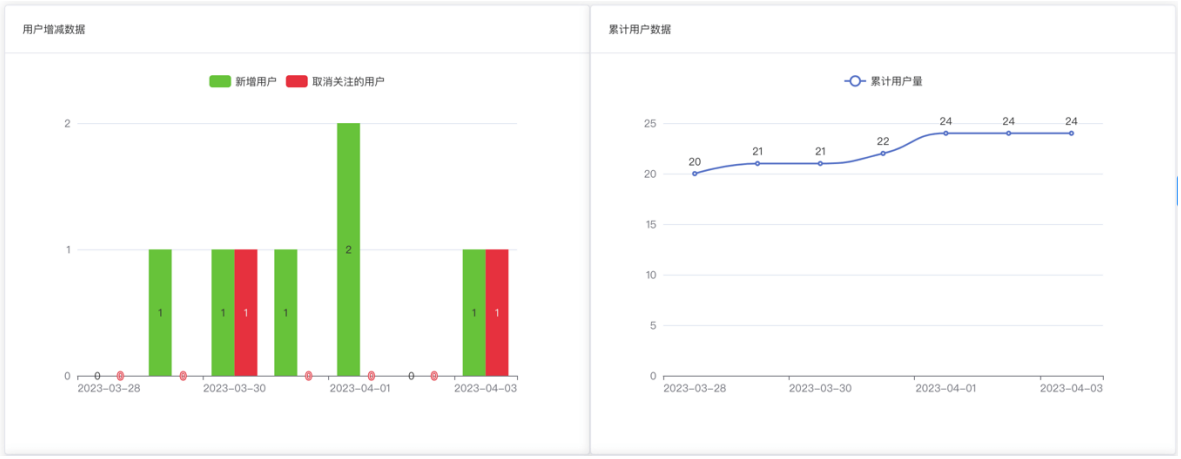
- UploadImg 组件：位于 src/components/UploadFile/src/UploadImg.vue (opens new window)内
- 实战案例：src/views/system/oauth2/client/ClientForm.vue(opens new window)

2. 不常用组件

2.1 EChart 图表组件

基于 Apache ECharts (opens new window)封装，自适应窗口大小

- EChart 组件：位于 src/components/EChart (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/echart.html(opens new window)
- 实战案例：src/views/mp/statistics/index.vue(opens new window)



2.2 InputPassword 密码输入框

对 Element Plus 的 Input 组件进行封装

- InputPassword 组件：位于 src/components/InputPassword (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/input-password.html(opens new window)
- 实战案例：src/views/Profile/components/ResetPwd.vue(opens new window)

2.3 ContentDetailWrap 详情包裹组件

用于展示详情，自带返回按钮。

- ContentDetailWrap 组件：位于 src/components/ContentDetailWrap (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/content-detail-wrap.html(opens new window)
- 实战案例：暂无

2.4 ImageViewer 图片预览组件

将 Element Plus 的 ImageViewer (opens new window)组件函数化，通过函数方便创建组件

- ImageViewer 组件：位于 src/components/ImageViewer (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/image-viewer.html(opens new window)
- 实战案例：暂无

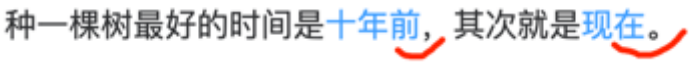
2.5 Qrcode 二维码组件

基于 qrcode (opens new window)封装

- Qrcode 组件：位于 src/components/Qrcode (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/qrcode.html(opens new window)
- 实战案例：暂无

2.6 Highlight 高亮组件

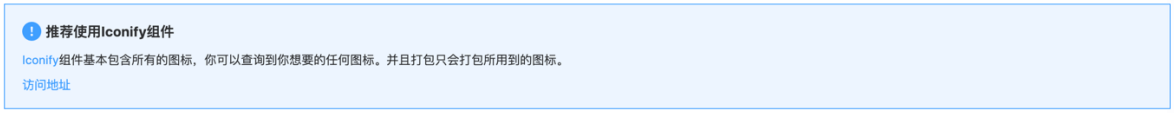
- Highlight 组件：位于 src/components/Highlight (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/highlight.html(opens new window)
- 实战案例：暂无



2.6.1 Infotip 信息提示组件

基于 Highlight 组件封装

- Infotip 组件：位于 src/components/Infotip (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/infotip.html(opens new window)
- 实战案例：暂无



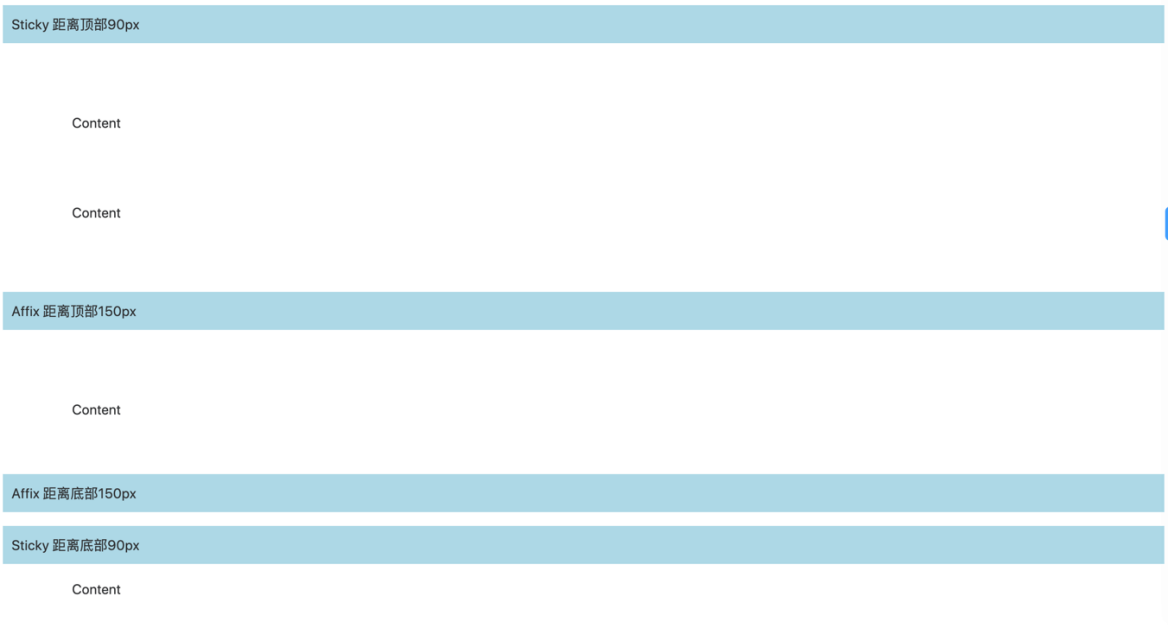
2.7 Error 缺省组件

用于各种占位图组件，如 404、403、500 等错误页面。

- Error 组件：位于 src/components/Error (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/error.html(opens new window)
- 实战案例：403.vue (opens new window)、404.vue (opens new window)、500.vue(opens new window)

2.8 Sticky 黏性组件

- Sticky 组件：位于 src/components/Sticky (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/sticky.html(opens new window)
- 实战案例：暂无



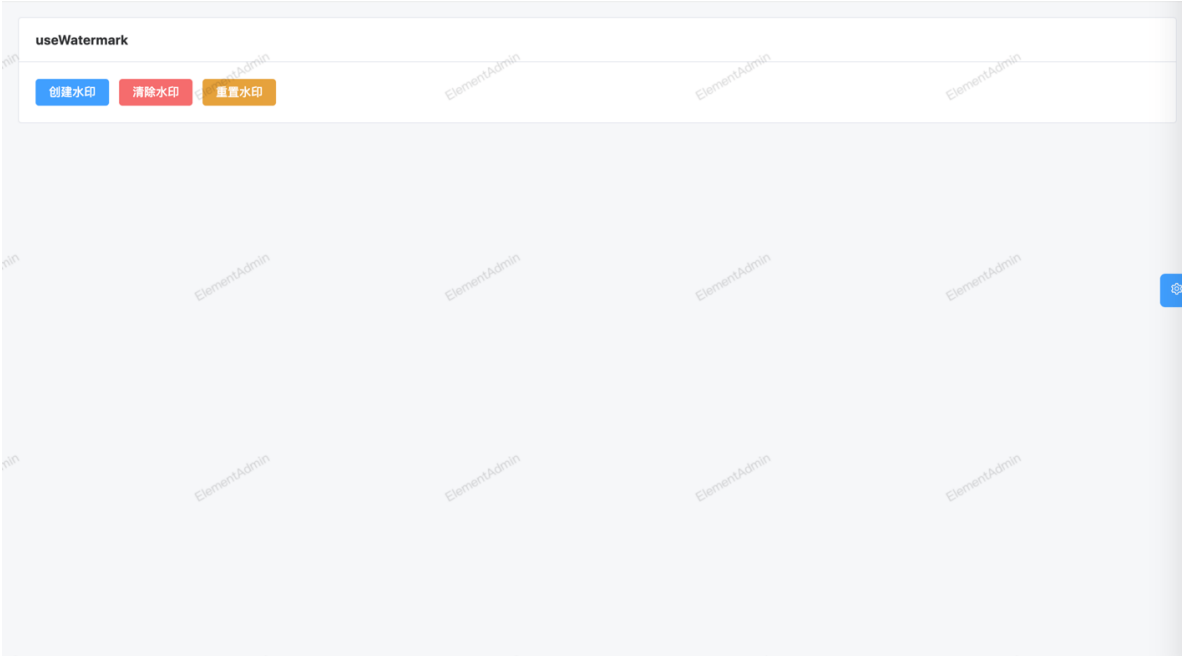
2.9 CountTo 数字动画组件

- CountTo 组件：位于 src/components/CountTo (opens new window)内
- 详细文档：vue-element-plus-admin-doc/components/count-to.html(opens new window)
- 实战案例：暂无

2.10 useWatermark 水印组件

为元素设置水印

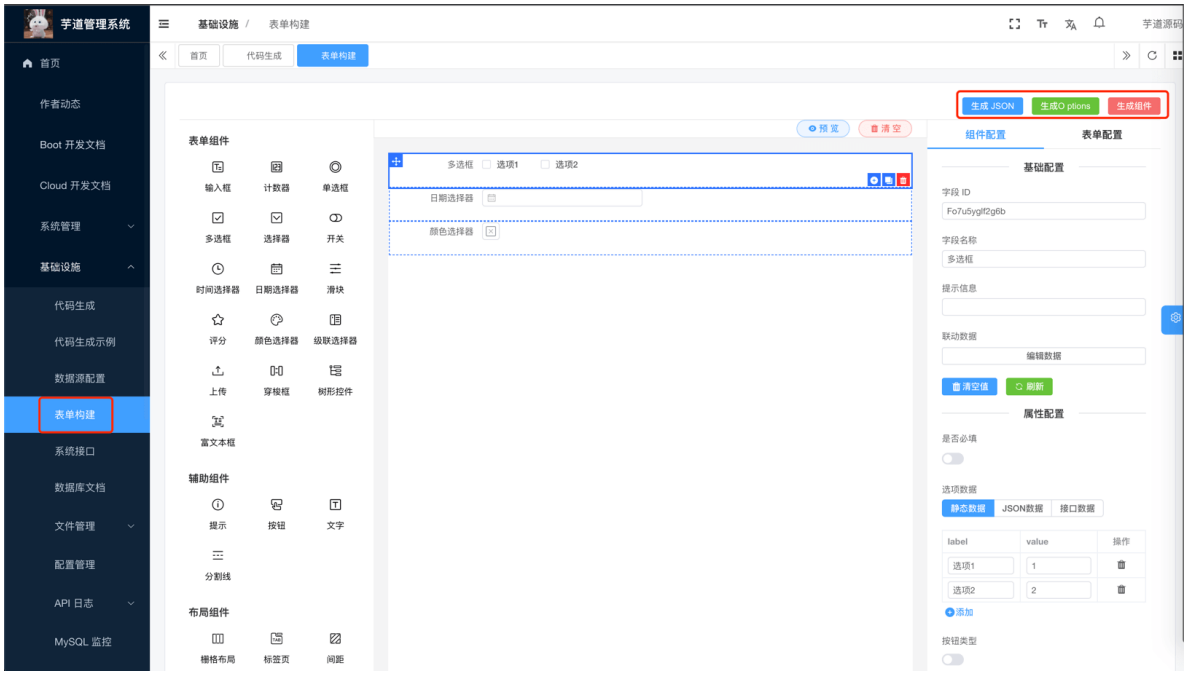
- useWatermark 组件：位于 src/hooks/web/useWatermark.ts (opens new window)内
- 详细文档：vue-element-plus-admin-doc/hooks/useWatermark.html(opens new window)
- 实战案例：暂无



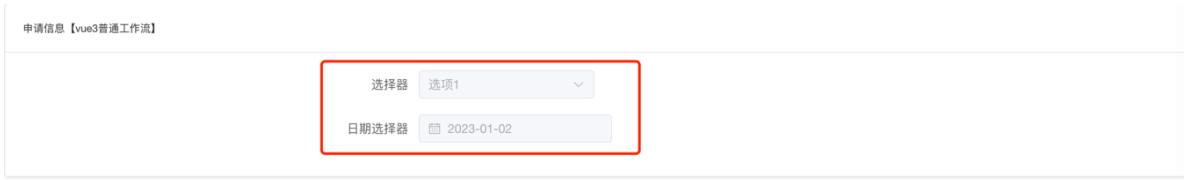
2.11 form-create 动态表单生成器

详细文档：http://www.form-create.com/(opens new window)

① 实战案例 - 表单设计：src/views/infra/build/index.vue(opens new window)



② 实战案例 - 表单展示：src/views/bpm/processInstance/detail/index.vue(opens new window)

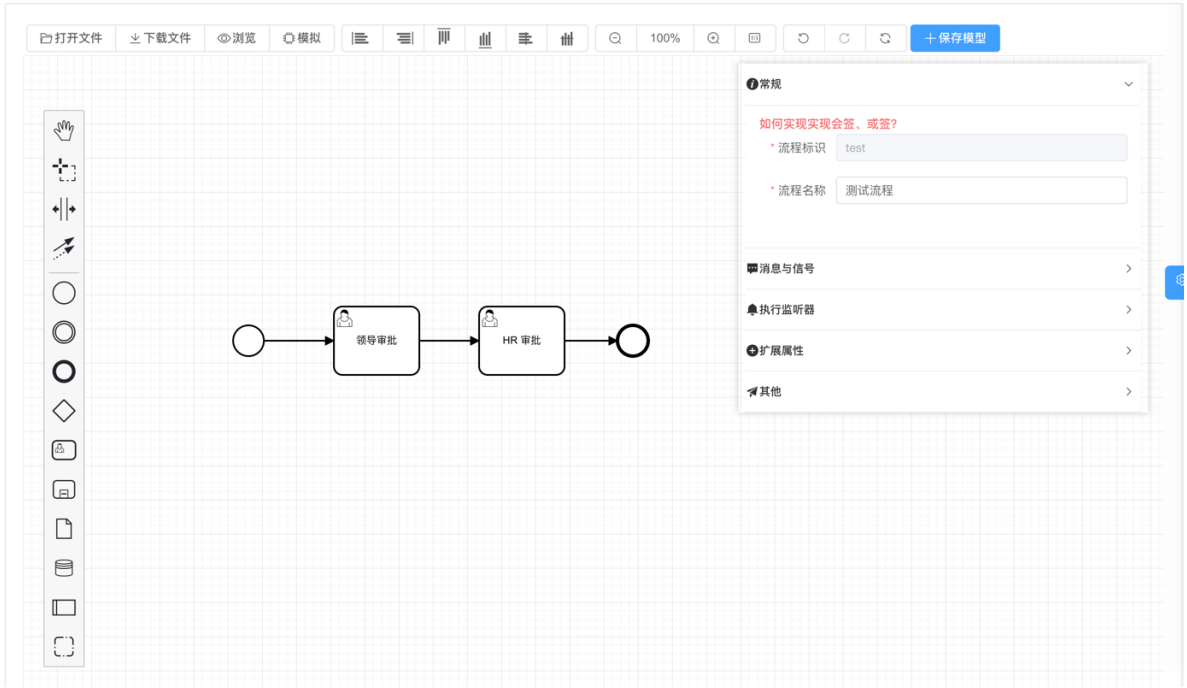


2.12 bpmn-js 工作流组件

核心是基于 bpmn-js (opens new window)封装

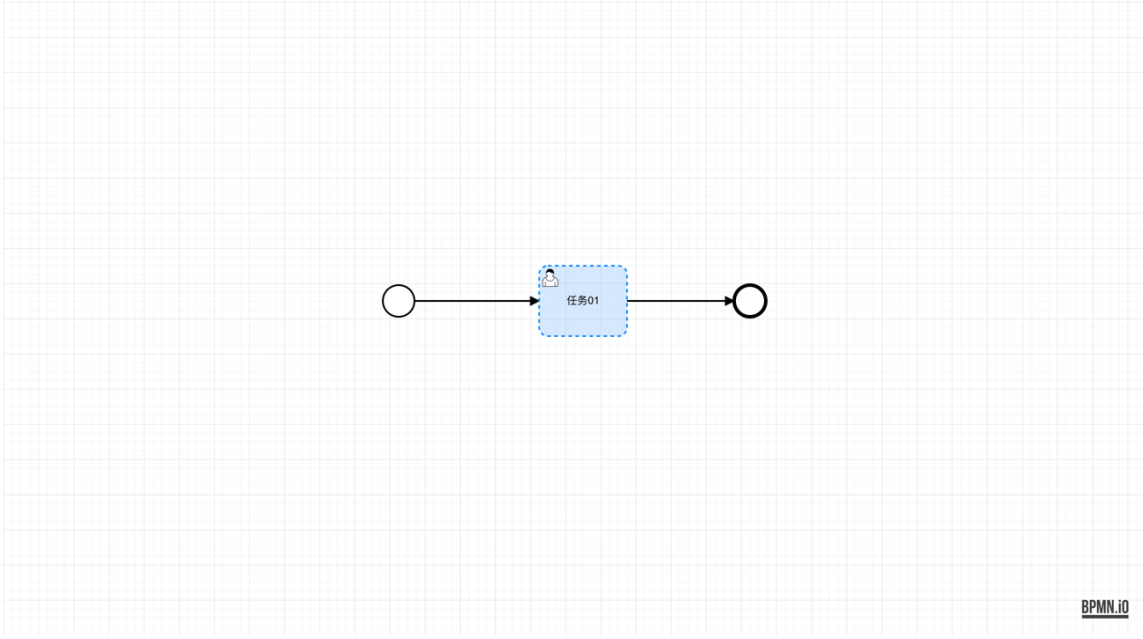
2.12.1 MyProcessDesigner 流程设计组件

- MyProcessDesigner 组件：位于 src/components/bpmnProcessDesigner/package/designer/index.ts (opens new window) 内，基于 https://gitee.com/MiyueSC/bpmn-process-designer (opens new window)项目适配
- 实战案例：src/views/bpm/model/editor/index.vue(opens new window)



2.12.2 MyProcessViewer 流程展示组件

- MyProcessViewer 组件：位于 src/components/bpmnProcessDesigner/package/designer/index2.ts (opens new window)内
- 实战案例：src/views/bpm/processInstance/detail/ProcessInstanceBpmnViewer.vue(opens new window)



3. 组件注册

友情提示：

该小节，基于 《vue element plus admin —— 组件注册 》 (opens new window)的内容修改。

组件注册可以分成两种类型：按需引入、全局注册。

3.1 按需引入

项目目前的组件注册机制是按需注册，是在需要用到的页面才引入。

```
<script setup lang="ts">
import { ElBacktop } from 'element-plus'
import { useDesign } from '@/hooks/web/useDesign'

const { getPrefixCls, variables } = useDesign()

const prefixCls = getPrefixCls('backtop')
</script>

<template>
  <ElBacktop
    :class="`${prefixCls}-backtop`"
    :target="`.${variables.namespace}-layout-content-scrollbar
.${variables.elNamespace}-scrollbar__wrap`"
  />
</template>
```

注意：**tsx 文件内不能使用全局注册组件**，需要手动引入组件使用。

3.2 全局注册

如果觉得按需引入太麻烦，可以进行全局注册，在 src/components/index.ts (opens new window)，添加需要注册的组件。

以 `Icon` 组件进行了全局注册，举个例子：

```
import type { App } from 'vue'
import { Icon } from './Icon'

export const setupGlobCom = (app: App<Element>): void => {
  app.component('Icon', Icon)
}
```

如果 Element Plus 的组件需要全局注册，在 src/plugins/elementPlus/index.ts (opens new window)添加需要注册的组件。

以 Element Plus 中只有 `ElLoading` 与 `ElScrollbar` 进行全局注册，举个例子：

```
import type { App } from 'vue'

// 需要全局引入一些组件，如 ElScrollbar，不然一些下拉项样式有问题
import { ElLoading, ElScrollbar } from 'element-plus'

const plugins = [ElLoading]

const components = [ElScrollbar]

export const setupElementPlus = (app: App) => {
  plugins.forEach((plugin) => {
    app.use(plugin)
  })

  components.forEach((component) => {
    app.component(component.name, component)
  })
}
```

通用方法

1. 缓存配置

友情提示：

该小节，基于 《vue element plus admin —— 项目配置「缓存配置」》 (opens new window)的内容修改。

1.1 说明

在项目中，你可以看到很多地方都使用了 `wsCache.set` 或者 `wsCache.get`，这是基于 web-storage-cache (opens new window)进行封装，采用 `hook` 的形式。

该插件对HTML5 `localStorage` 和 `sessionStorage` 进行了扩展，添加了超时时间，序列化方法。可以直接存储 `json` 对象，同时可以非常简单的进行超时时间的设置。

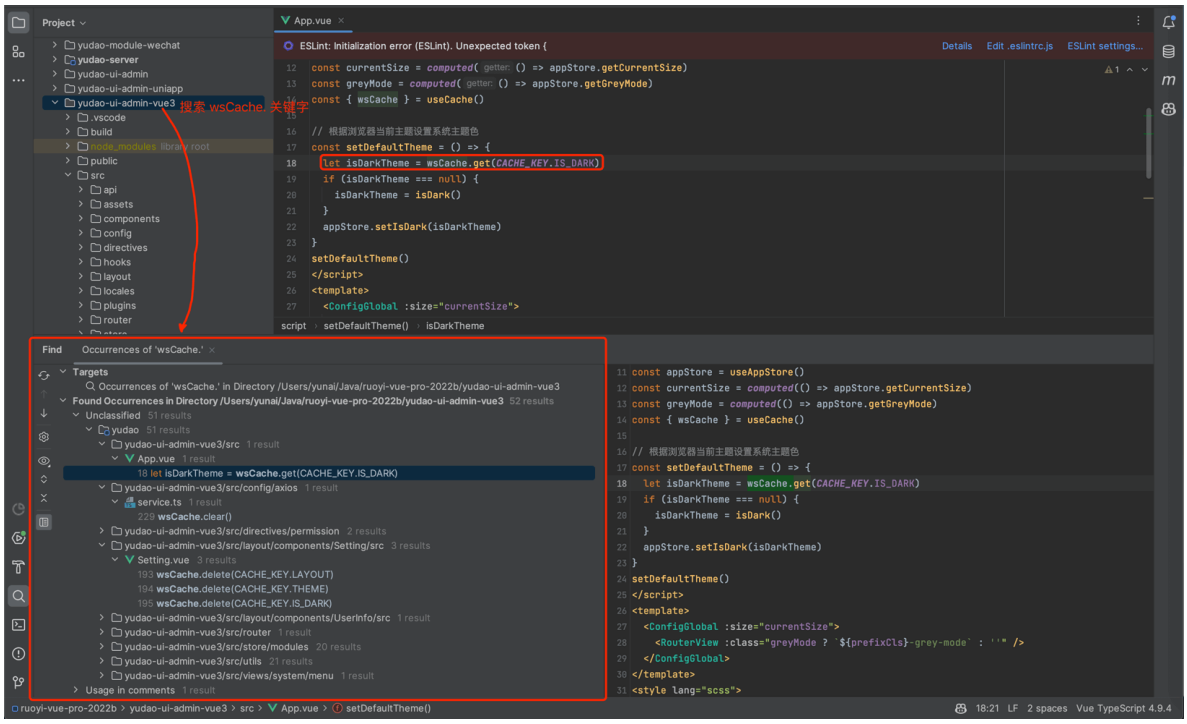
本项目默认是采用 `sessionStorage` 的存储方式，如果更改，可以直接在 useCache.ts (opens new window)中把 `type: CacheType = 'sessionStorage'` 改为 `type: CacheType = 'localStorage'`，这样项目中的所有用到的地方，都会变成该方式进行数据存储。

如果只想单个更改，可以传入存储类型 `const { wsCache } = useCache('localStorage')`，既可只适用当前存储对象。

注意：

更改完默认存储方式后，需要清除浏览器缓存并重新登录，以免造成不可描述的问题。

1.2 示例

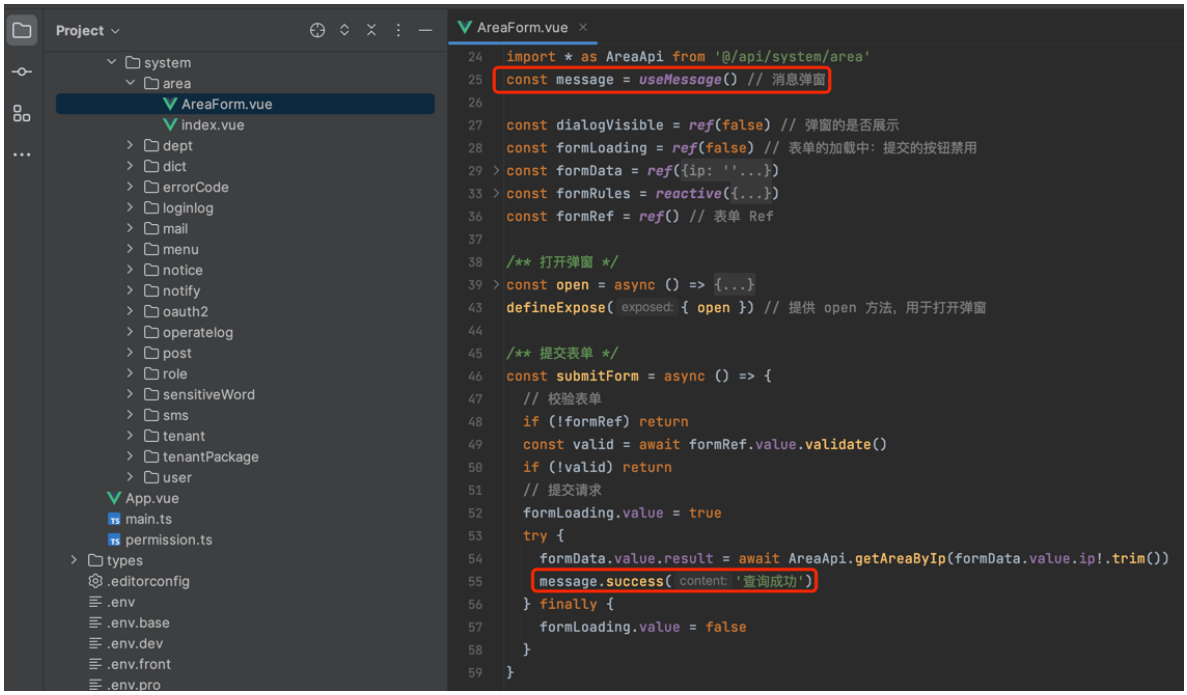


2. `message` 对象

2.1 说明

`message` 对象，由 src/hooks/web/useMessage.ts (opens new window)实现，基于 ElMessage、ElMessageBox、ElNotification 封装，用于做消息提示、通知提示、对话框提醒、二次确认等。

2.2 示例

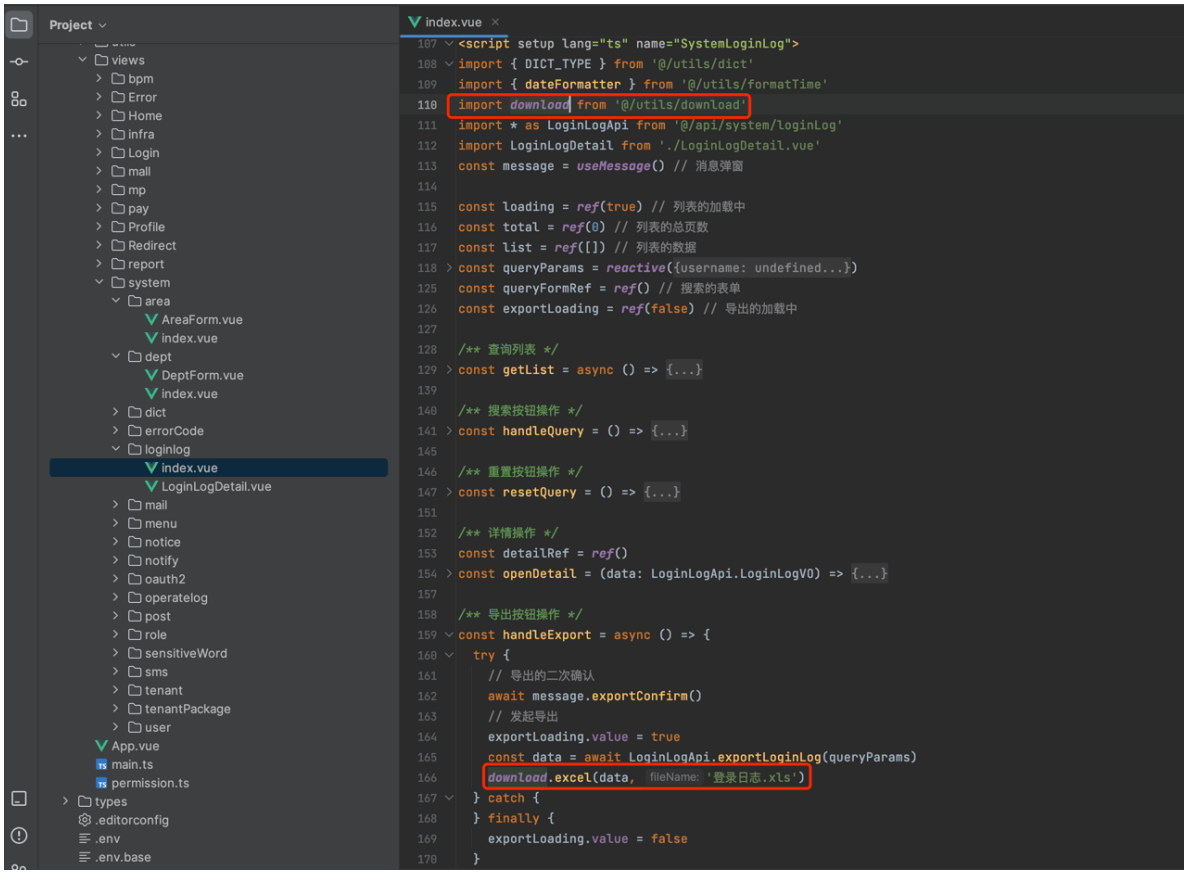


3. download 对象

3.1 说明

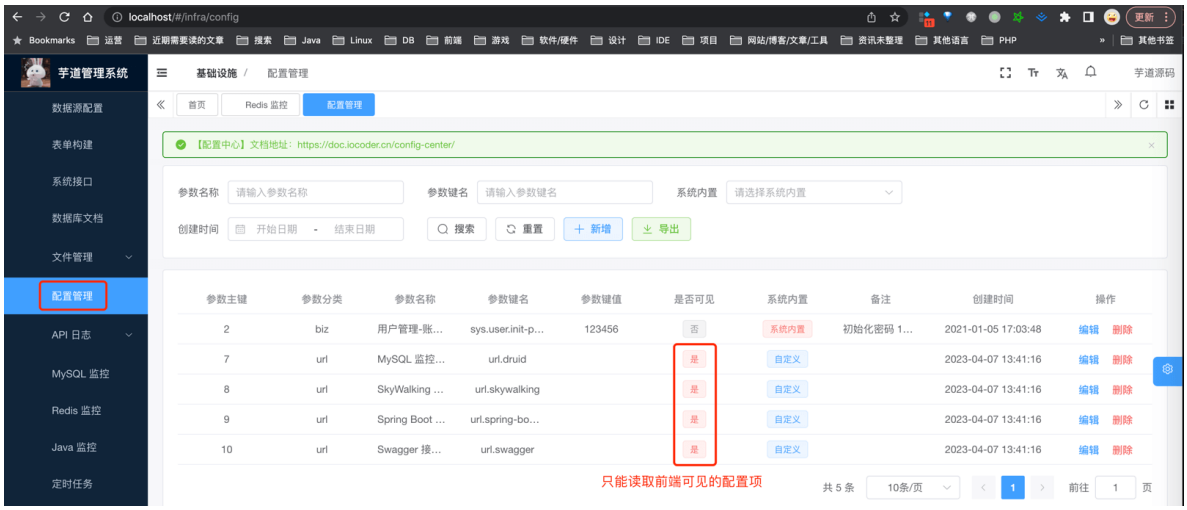
`$download` 对象，由 `util/download.ts` (opens new window)实现，用于 Excel、Word、Zip、HTML 等类型的文件下载。

3.2 示例



配置读取

在 [基础设施 -> 配置管理] 菜单，可以动态修改配置，无需重启服务器即可生效。



1. 读取配置

前端调用 `/@api/infra/config/index.ts` (opens new window)的 `#getConfigKey(configKey)` 方法，获取指定 key 对应的配置的值。代码如下：

```
// 根据参数键名查询参数值
export const getConfigKey = (configKey: string) => {
    return request.get({ url: '/infra/config/get-value-by-key?key=' + configKey
})
}
```

2. 实战案例

在 `src/views/infra/server/index.vue` (opens new window)页面中，获取 key 为 `"url.skywalking"` 的配置的值。代码如下：