从零开发企业级中后台项目（实战篇）

这是 Vue 实战内容的开篇，本文将带着大家完成一个企业级中后台项目的搭建。

源码地址：https://git.yummy.tech/common/admin-demo

首先，笔者倡导的是拥抱变化，所以本文所有内容，都会采用最新且生产可用的技术栈来进行企业级中后台的搭建。在此之前，推荐大家先看 从零开发企业级中后台项目（前置知识） 这篇文章，并对文章提及的技术栈有一定的了解。

本项目技术栈包括：

- Vue 3
- Vuex 4.x
- Vue-router 4.x
- Vite

一、项目初始化

Vue 项目初始化最快、最稳妥的方式无非两种：

- vue-cli
- vite

当然也可以直接使用社区比较优秀的模板如：

- vue-vben-admin
- antd-vue-pro

本章主要介绍基于 Vite 快速初始化项目的方式，先来简单了解下。Vite 是尤雨溪又一力作，号称下一代开发构建工具，由于其支持原生 ES 模块导入方法，所以它允许快速提供代码，使得开发效率大大提高。 如果读者好奇为什么要选 Vite ，可以看看官方文档给出的答案。

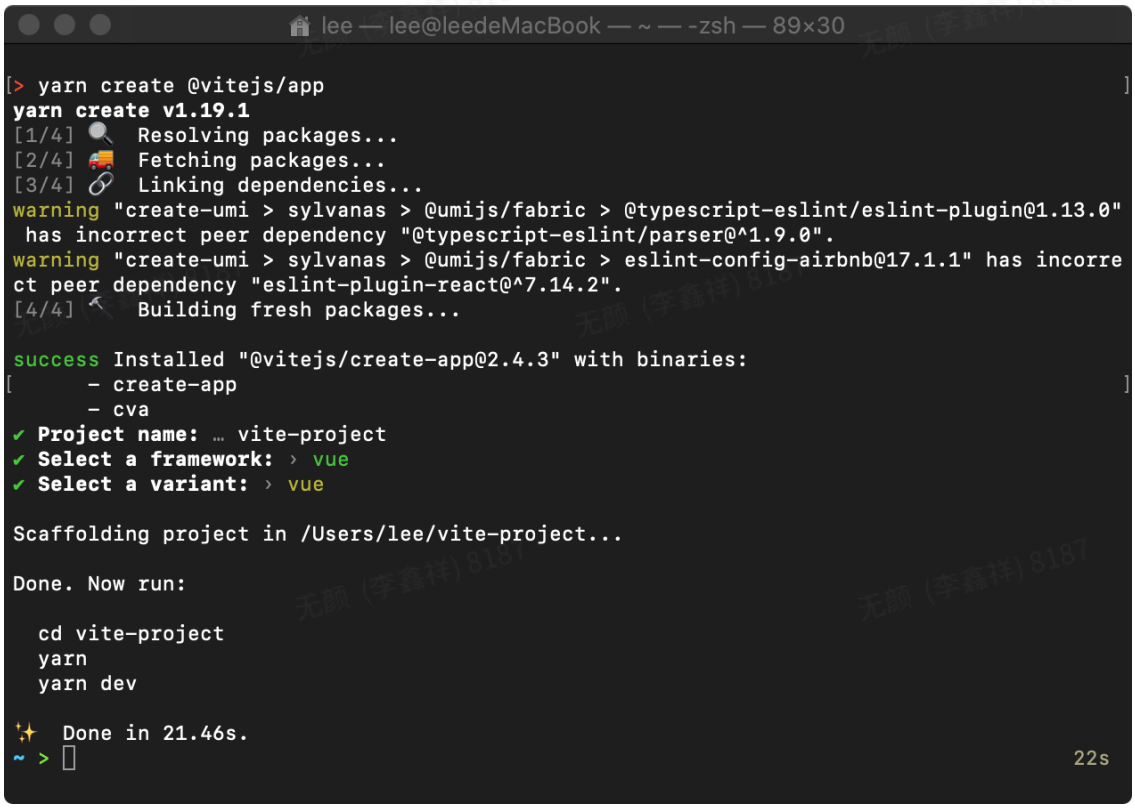
1.1 搭建项目

第一步，通过命令行搭建项目：

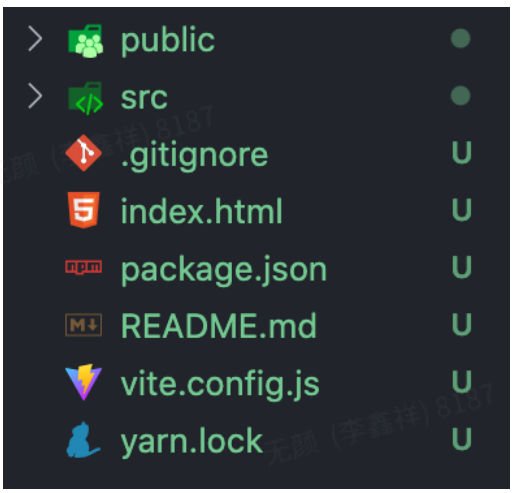
Bash

```
# 在终端执行下面命令
yarn create @vitejs/app
```

执行效果如图：



项目整体目录如图：



通过 Vite 命令行创建的项目 `package.json` 如下：

JSON

```json
{
  "version": "0.0.0",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "serve": "vite preview"
  },
  "dependencies": {
    "vue": "^3.0.5"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^1.2.3",
    "@vue/compiler-sfc": "^3.0.5",
    "vite": "^2.3.7"
  }
}
```

1.2 启动项目

安装依赖并启动项目：

Bash

```bash
# 安装依赖
yarn
# 启动开发服务器
yarn dev
```

启动成功之后可以看到如下页面：

Hello Vue 3 + Vite

Vite Documentation | Vue 3 Documentation

count is: 0

Edit components/HelloWorld.vue to test hot module replacement.

二、项目配置

在开始开发前，可以做一些基础的配置来提升开发效率。

2.1 配置别名

避免在项目中使用路径时，写一长串路径字符，可设置常用别名如：

- @ 代表 src
- ~ 代表根目录下的别名

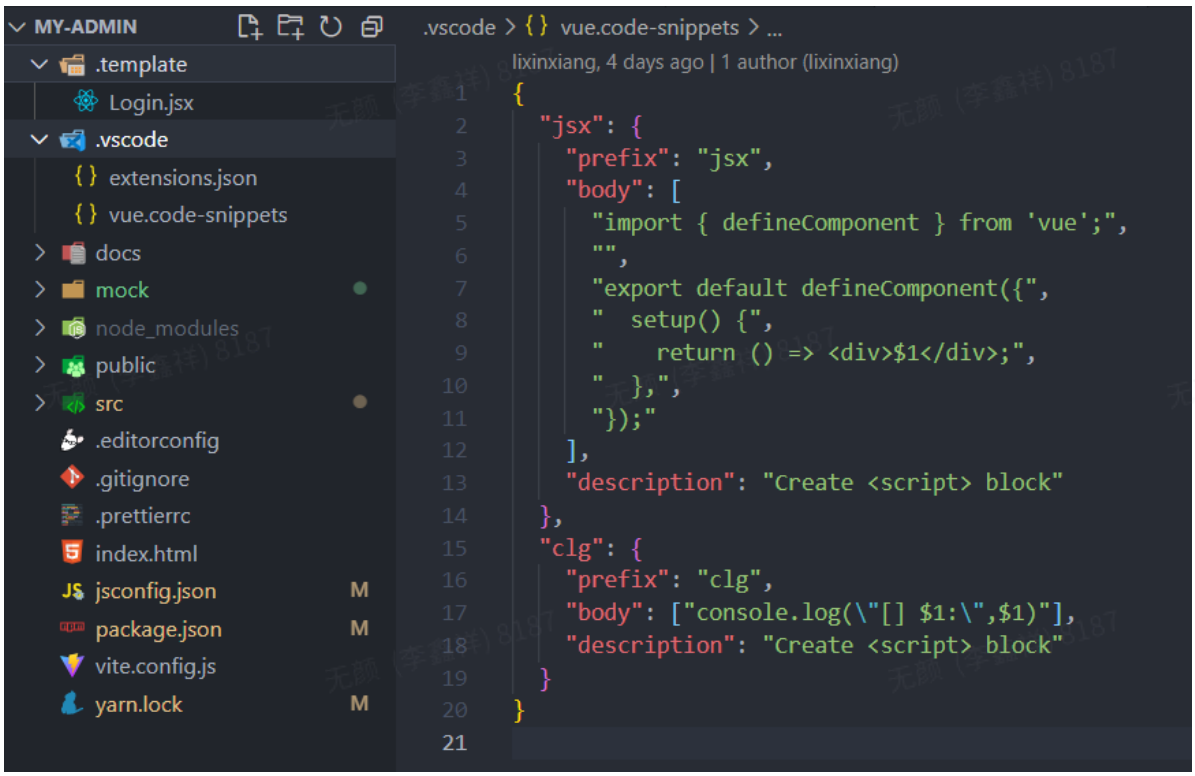
在 `vite.config.js` 下添加 `resolve.alias`，代码如下：

JavaScript

```javascript
import { defineConfig } from "vite";
import vue from "@vitejs/plugin-vue";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      "~": path.resolve(__dirname, "./"),
      "@": path.resolve(__dirname, "src"),
    },
  },
});
```

2.2 代码片段

我们在开发过程中，通常会沉淀非常多的业务代码片段，提高开发效率的同时，这部分代码还可以作为后续低代码平台的场景提供者。在如下地方添加：



Vscode 还有很多小技巧，但本章的重点是结合项目的配置，一些项目标准的配置，如 Eslint、Prettier 等不属于本章的内容，感兴趣的同学可以查阅相关文章进行了解。

2.3 Vite JSX 设置

由于笔者比较推荐使用 React，所以推荐使用 JSX 这种更为灵活的方式进行组件和业务开发。读者可以结合自身情况自行选择。

安装 plugin-vue-jsx：

Bash

```
yarn add -D @vitejs/plugin-vue-jsx@next
```

在 `vite.config.js` 中设置：

JavaScript

```
import vueJsx from "@vitejs/plugin-vue-jsx";
import { defineConfig } from "vite";

export default defineConfig({
  plugins: [vueJsx()],
});
```

添加 JSX 之后可以直接 import 使用，无需注册：

JavaScript

```
import { defineComponent } from "vue";
import ProLayout from "@ant-design-vue/pro-layout";

export default defineComponent({
  setup() {
    return () => (
      <ProLayout>
        <router-view />
      </ProLayout>
    );
  },
});
```

单文件的编码方式想必很多读者都非常熟悉了，而本文秉承拥抱变化的初衷，所以后续教程均使用 JSX 语法。

三、安装路由

中后台项目作为单页应用，路由是必不可少的。下面我们基于 Vue-Router 来进行简单的配置讲解。

第一步，新增 Vue-Router 依赖：

Bash

```
yarn add vue-router@next
```

第二步，新建 `src/router/index.js`，并添加如下代码：

JavaScript

```
import { createRouter, createWebHashHistory } from "vue-router";

import Home from "@/views/Home.vue";
import Dashboard from "@/views/Dashboard.vue";

// 参考: https://next.router.vuejs.org/zh/
const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    {
      path: "/",
      name: "/",
      component: Home,
    },
    {
      path: "/dashboard",
      name: "dashboard",
      component: Dashboard,
    },
  ],
});

export default router;
```

第三步，在 `src/main.js` 中添加如下代码：

JavaScript

```
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";

const app = createApp(App);
app.use(router);
app.mount("#app");
```

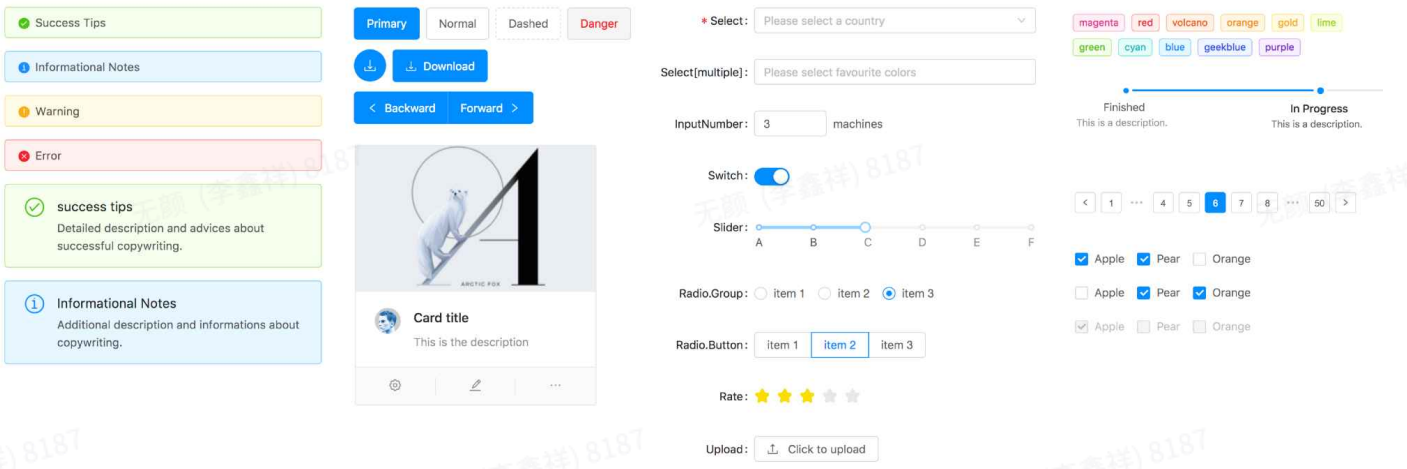
在 `src/views` 下创建 Home、Dashboard 组件文件，就可以开始联调看相应的效果了。

四、安装 UI 库

对于企业级中后台产品来说，目前社区比较推荐使用 Ant Design Vue 这一套。

4.1 Ant Design Vue 简介

Ant Design Vue 是使用 Vue 实现的遵循 Ant Design 设计规范的高质量 UI 组件库，用于开发和服务于企业级中后台产品。除了优秀的开发支持，还具备完整的 设计资源包。团队可以基于这些资源构建属于自己的产品设计体系。



4.2 如何使用

第一步，安装 ant-design-vue：

Bash

```
yarn add -D ant-design-vue@next
```

第二步，Antd 本身使用的是 less，故推荐安装 less 作为样式的：

Bash

```
yarn add -D less
```

第三步，`src/main.js` 中引入：

JavaScript

```
// Vue 3.0 中 全局引入，也可以按需 https://2x.antdv.com/docs/vue/introduce-cn
import { createApp } from "vue";
import Antd from "ant-design-vue";
import "ant-design-vue/dist/antd.less";

const app = createApp(App);
app.use(Antd);
```

第四步，在 `vite.config.js` 开启 less：

JavaScript

```
import { defineConfig } from "vite";

export default defineConfig({
  css: {
    preprocessorOptions: {
      less: {
        javascriptEnabled: true,
      },
    },
  },
});
```

第五步 (可选)，在 `vite.config.js` 设置每个组件自动引入 antd 变量：

首先新建 `src/themes/var.less` 文件：

CSS

```
@import "ant-design-vue/dist/antd.less";// 这里可以覆盖 antd 变量
```

这里的内容主要涉及主题定制，更多信息参考 antd 主题定制。另外，所有样式变量可以在 这里 找到。

注意：理论上，所有设计 Design Token 的值都应该从变量中获取，以便最大强度地进行主题定制。避免后续，手动替换值得尴尬局面。

接着修改 `vite.config.js` 配置文件：

JavaScript

```
import { defineConfig } from "vite";
function resolve(dir) {
  return path.resolve(__dirname, ".", dir);
}

export default defineConfig({
  css: {
    preprocessorOptions: {
      modifyVars: {
        hack: `true; @import "${resolve("./src/themes/var.less")}";`,
      },
    },
  },
});
```

添加最终代码后的最终效果如图：

注意：引入主题修改配置会使得 vite 启动速度大幅度降低（教程为提升开发体验，暂时移除该配置）。如果对主题定制没有特别的要求，可以移除 less 变量修改相关配置，并将视觉呈现放置统一的自定义 UI 组件库中处理。

五、基础布局

对于中后台管理系统来说，通常会有基础布局，可以使用自研或者参考诸如 pro-layout 这类的开源库(涵盖了大量业务场景)。下面简单介绍与基础布局相关的知识点：

第一步，新建 `src/layouts/BasicLayout.jsx` (这里的后缀在 vite 中不建议忽略)：

JavaScript

```
import { defineComponent } from "vue";
import { useRouter } from "vue-router";
import { getMenuData } from "@ant-design-vue/pro-layout";
import ProLayout from "@ant-design-vue/pro-layout";

export default defineComponent({
  setup() {
    const { getRoutes } = useRouter();
    const routes = getRoutes();
    const { menuData } = getMenuData(routes); // 通过路由配置获取菜单数据

    return () => (
      <ProLayout>
        <router-view />
      </ProLayout>
    );
  },
});
```

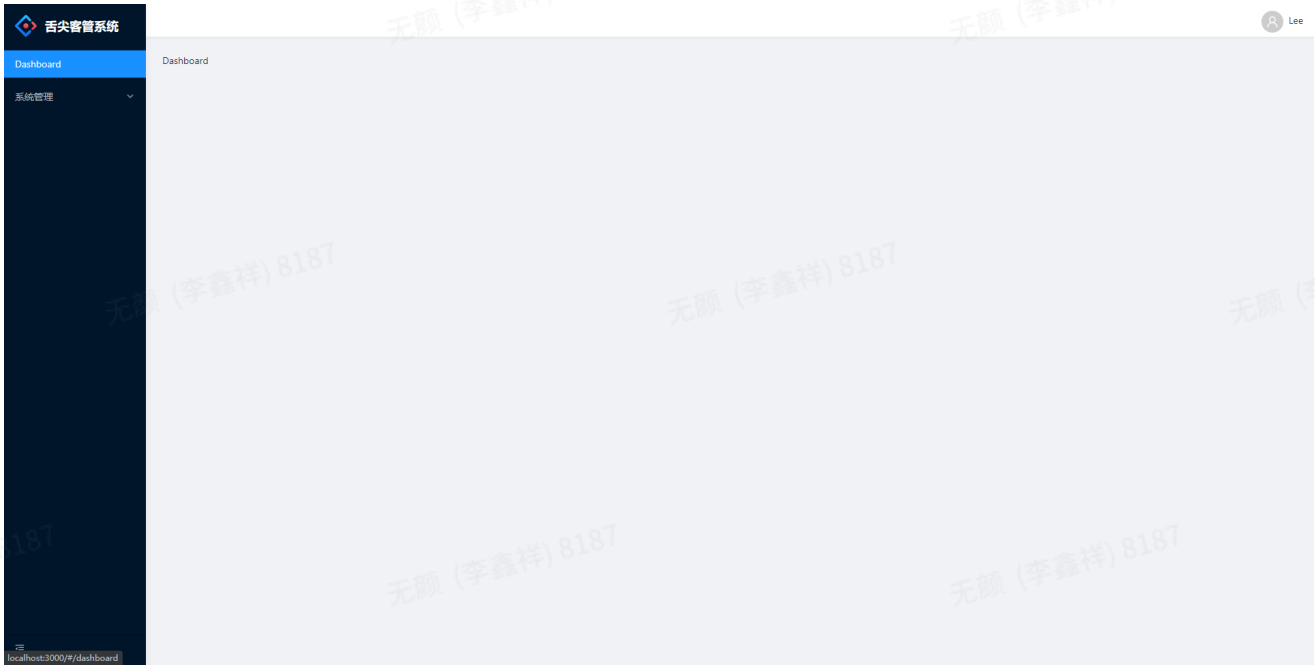
注意：为避免纯黏贴代码导致教程冗余，示例代码非完整可运行代码。

第二步，配置路由参数如下：

JavaScript

```javascript
import { createRouter, createWebHashHistory } from "vue-router";
import { BasicLayout } from "@/layouts";

const router = createRouter({
  routes: [
    {
      path: "/",
      name: "index",
      meta: { title: "home" },
      redirect: "/dashboard",
      component: BasicLayout,
      children: [
        {
          path: "/dashboard",
          name: "dashboard",
          meta: { title: "Dashboard", icon: "SmileOutlined" },
          // 动态引入路由文件，起到代码分割的作用（懒加载）
          component: () =>
            import(/* webpackChunkName: "dashboard" */
"@/views/Dashboard.vue"),
        },
      ],
    },
  ],
});

export default router;
```

大概效果如图：



这样，一个基础的左右布局的模板就完成了，后续只需要在如有里面简单关联即可，并且可以在 layouts 目录下新增更多的布局模板供路由使用。

六、路由权限

权限对于中后台系统来说也是必不可少的，路由的权限通常可以通过两种方式来实现：

- 前端在本地配置好路由表，根据服务端返回的权限标识来动态将路由表添加到 Vue-router 配置中。
- 通过服务端来返回对应的路由表，动态添加带配置中。

考虑到前端分离，服务解耦，首选前端本地维护路由表的方式，并且这种方式维护成本最低、且最灵活。（服务端无需配置 Component 和 Icon 等信息）。

6.1 前端配置

根据是否需要权限控制，可以把路由分为两种：

- staticRoutes：静态路由，不受权限控制
- dynamicRoutes：动态路由，受权限控制

首先，我们要做的就是配置本地动态路由（dynamicRoutes）用于后续根据权限生成路由：

JavaScript

```javascript
import { BasicLayout } from "@/layouts";

/**
 * 根据权限生成的路由
 * @type { *[] }
 */
const dynamicRoutes = [
  {
    path: "/",
    name: "index",
    meta: { title: "home" },
    redirect: "/dashboard",
    component: BasicLayout,
    children: [
      {
        path: "/dashboard",
        name: "dashboard",
        meta: { title: "Dashboard", icon: "SmileOutlined", permission:'' },
        component: () =>
          import(/* webpackChunkName: "dashboard" */ "@/views/Dashboard.vue"),
      },
    ],
  },
],

export default dynamicRoutes
```

有了动态路由表，就可结合 Vue-router 4.0 的 动态路由 在应用程序运行的时候添加或删除路由。提供了下列 API 来实现路由动态配置（查看更多 API）：

- `router.addRoute()`
- `router.removeRoute()`
- `router.hasRoute()`
- `router.getRoutes()`

另外，对于权限的控制需要实现在无论进入哪个路由，都可以根据权限生成路由表，这里需要用到导航守卫，伪代码如下：

JavaScript

```javascript
import router from "./router";

router.beforeEach(() => {
  // 用全局数据中的角色来标识是否已经根据角色生成路由表 ( 如 vuex 中的 store )
  if (!store.roles) {
    // 发送请求获取 roles
    const { roles } = request("/role");

    // 根据角色生成路由表
    const addRouters = generateRoutes(roles);

    // 动态添加可访问路由表
    router.addRoutes(addRouters);
  }
});
```

最后来说一下 generateRoutes 生成路由的方式：

- 根据 dynamicRoutes 中的每个路由 meta 信息中的权限标识与服务端请求的返回的标识进行过滤
- 通常服务端返回标识的方式有两种，一种是方式：返回角色 roles，这时 meta 中的 permission 通常为数组。另一种方式是：直接返回唯一约定好的权限标识。这种比较暴力，但相对灵活。所以具体如何写 generateRoutes 的逻辑，视具体服务端逻辑

此外，为提升研发以及交互体验通常会用到这些库：

- vuex 用于管理全局数据
- axios 用于发送请求
- NProgress 用于提升页面加载体验的进度条

注意：后续结合这些库的方式请参考源码，这里就不罗列代码了。

七、mock 服务

在前后端分离的项目，通常会接入 mock 服务。前端可以和服务端约定好数据格式，在本地 mock 对应的数据。另外 mock 数据有利于边界情况测试。接下来，我们结合 vite 接入 mock 服务（结合登录场景）。

7.1 基础配置

接入 mock 服务需要用到下列依赖，读者可提前了解一下：

- mockjs 用于 mock 数据
- vite-plugin-mock 用于启动 vite mock 服务
- axios 用于发送请求
- ahooks-vue 用于结合 Vue 维护请求数据

第一步，安装依赖：

Bash

```bash
# 安装 mock 数据生成库
yarn add -D mockjs

# 安装 vite mock 服务插件
yarn add -D vite-plugin-mock

# 安装 axios 请求
yarn add axios

# 安装 ahooks-vue 常用 vue hook
yarn add ahooks-vue
```

第二步，配置 `vite.config.js` ：

JavaScript

```javascript
import { defineConfig } from "vite";
import { viteMockServe } from "vite-plugin-mock";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    viteMockServe({
      // default
      mockPath: "mock",
      localEnabled: true,
    }),
  ],
});
```

第三步，在 `vite.config.js` 同级目录新建 mock 目录，并添加 `user.js` 文件：

CoffeeScript

```coffeescript
export default [
  {
    url: "/api/user/login",
    method: "post",
    response: ({ body }) => {
      console.log("[mock response]", body);
      return {
        code: 0,
        message: "ok",
        data: null,
      };
    },
  },
];
```

如果需要添加更灵活的数据 可以参考mock 规则。下面结合登录场景讲解，如何使用 mock 服务。

7.2 接口服务

首先新建 `helpers/request.js` 用于项目输出统一的 axios 实例用于发送请求：

JavaScript

```javascript
import axios from "axios";

// 创建 axios 实例
const request = axios.create({
  // API 请求的默认前缀
  baseURL: "/",
  timeout: 6000, // 请求超时时间
});

// 异常拦截处理器
const errorHandler = (error) => {
  if (error.response) {
    console.log("[axios] error.response:", error.response);
  }
  return Promise.reject(error);
};

// 请求拦截器
request.interceptors.request.use((config) => {
  console.log("[axios] config:", config);
  return config;
}, errorHandler);

// 响应拦截器
request.interceptors.response.use((response) => {
  console.log("[axios] response:", response);
  return response.data;
}, errorHandler);

export default request;
```

创建 `services/user.js` 文件：

JavaScript

```javascript
import { request } from "@/helpers";

const userApi = {
  Login: "/api/user/login",
};

export function login(data) {
  return request({
    url: userApi.Login,
    method: "post",
    data,
  });
}
```

添加登录页面（重点关注发送请求部分的逻辑，更多细节参看源码）：

JavaScript

```javascript
import { defineComponent, reactive } from "vue";
import { Button, Form, Input, Typography } from "ant-design-vue";
import { useAxios } from "ahooks-vue";

import { login } from "@/services/user";

export default defineComponent({
  setup() {
    // 通过 useAxios 来建立请求机制
    const { run, loading } = useAxios(login, { manual: true });

    const state = reactive({ username: "", password: "" });

    const handleFinish = async (values) => {
      console.log(values);
      const { code } = await run(values);
      if (code === 200) {
        router.replace("/");
      }
    };

    return () => (
      <Form
        model={state}
        onFinish={handleFinish}
```

```
        onFinishFailed={handleFinishFailed}
      >
        <Form.Item name="username">
          <Input vModel={[state.username, "value"]} placeholder="Username" />
        </Form.Item>
        <Form.Item name="password">
          <Input
            vModel={[state.password, "value"]}
            type="password"
            placeholder="Password"
          />
        </Form.Item>
        <Button
          htmlType="submit"
          block
          type="primary"
          disabled={state.username === "" || state.password === ""}
          loading={loading.value}
        >
          Login
        </Button>
      </Form>
    );
  },
});
```

八、登录功能

我们先来看一下常规的登录场景：

场景一：

- 访问登录页面，登录之后跳转首页，同时记录 token 且在后续的每个请求头上都带上 token 信息
- 退出登录，清空 token 记录

场景二：

- 未登录的情况下，访问非登录页，重定向到登录页

根据这些场景可以知道需要做什么，接着来完成如下功能：

- 登录页、其他页面的路由配置
- mock 登录、用户信息等接口
- 接口统一鉴权，无权限则重定向到登录页

- 登录页、首页 UI 实现 (看源码)
- 用户信息 (如 token、name) 、路由表数据全局维护 (vuex)
- 根据权限动态生成路由表

8.1 路由配置

我们在路由权限讲了如何配置动态路由，这里来简单配置下。首先登录页面和 404 这类属于不需要权限的路由，我们约定为 staticRoutes。代码如下：

JavaScript

```
import { UserLayout } from "@/layouts";

/**
 * 不需要权限的路由
 * @type { *[] }
 */
const staticRoutes = [
  {
    path: "/user",
    component: UserLayout,
    redirect: "/user/login",
    hidden: true,
    children: [
      {
        path: "login",
        name: "login",
        component: () =>
          import(/* webpackChunkName: "user" */ "@/views/user/Login.jsx"),
      },
    ],
  },

  {
    path: "/404",
    component: () =>
      import(/* webpackChunkName: "fail" */ "@/views/exception/404.jsx"),
  },
];

export default staticRoutes;
```

需要动态加载的路由表基础配置如下：

JavaScript

```javascript
import { BasicLayout } from "@/layouts";

/**
 * 根据权限生成的路由
 * @type { *[] }
 */
const dynamicRoutes = [
  {
    path: "/",
    name: "index",
    meta: { title: "home" },
    redirect: "/dashboard",
    component: BasicLayout,
    children: [
      {
        path: "/dashboard",
        name: "dashboard",
        meta: {
          title: "Dashboard",
          icon: "SmileOutlined",
        },
        component: () =>
          import(/* webpackChunkName: "dashboard" */ "@/views/Dashboard.vue"),
      },
    ],
  },
  {
    path: "/:catchAll(.*)",
    redirect: "/404",
    hidden: true,
  },
];

export default dynamicRoutes;
```

8.2 模拟数据

登录的时候，需要匹配账号和密码，并且密码需要加密。为了避免大量地贴代码，伪代码实现如下：

JavaScript

```javascript
import md5 from "md5";

const fakeUserList = [
  {
    id: "4291d7da9005377ec9aec4a71ea837f",
    username: "admin",
    password: md5("admin"),
    token: "fakeToken1",
    name: "admin",
    avatar: "https://q1.qlogo.cn/g?b=qq&nk=339449197&s=640",
    permissions: ["admin"],
  },
];

export default [
  {
    url: "/api/user/login",
    method: "post",
    response: ({ body }) => {
      // body 请求体
      const { username, password } = body;

      return "你需要 mock 的任何数据，这里 mock 通常返回的是从 fakeUserList 匹配的用户数据";
    },
  },
];
```

8.3 请求接口权限校验

得益于 axios 的请求拦截器功，我们可以很轻松的实现接口的拦截并做响应的处理。

JavaScript

```javascript
import axios from "axios";
import storage from "./storage";
import { ACCESS_TOKEN } from "@/store/mutation-types";

// 创建 axios 实例
const request = axios.create({
  // API 请求的默认前缀
  baseURL: "/",
  timeout: 6000, // 请求超时时间
});

// 异常拦截处理器
const errorHandler = (error) => {
  if (error.response) {
    console.log("[axios] error.response:", error.response);
  }
  return Promise.reject(error);
};

// 请求拦截器
request.interceptors.request.use((config) => {
  const token = storage.getItem(ACCESS_TOKEN);
  if (token) {
    // 如果 localstorage 中存有 token，则在请求中带上
    config.headers["authorization"] = token;
  }
  return config;
}, errorHandler);

// 响应拦截器
request.interceptors.response.use((response) => {
  return response.data;
}, errorHandler);

export default request;
```

8.4 用户信息全局维护

在具体实现登录功能之前，我们可以先来设计一下登录相关的全局数据维护，下面使用 vuex 来实现用户信息和路由权限表的 store：

入口文件：

JavaScript

```
import Vuex from "vuex";

import permission from "./modules/permission"; // 权限表
import user from "./modules/user"; // 用户信息
import getters from "./getters";

const store = new Vuex.Store({
  modules: {
    permission,
    user,
  },
  getters, // 相当于 vue 的 computed，方便取值
});

export default store;
```

用户信息：

JavaScript

```
import storage from "@/helpers/storage";
import { login, logout, getInfo } from "@/services";
import { ACCESS_TOKEN } from "@/store/mutation-types";

const user = {
  state: {
    token: "",
    name: "",
    avatar: "",
    permissions: [],
    info: {},
  },

  mutations: {
    SET_TOKEN: (state, token) => {
      state.token = token;
    },
    SET_NAME: (state, { name }) => {
      state.name = name;
    },
    SET_AVATAR: (state, avatar) => {
      state.avatar = avatar;
```

```
    },
    SET_PERMISSIONS: (state, permissions) => {
      state.permissions = permissions;
    },
    SET_INFO: (state, info) => {
      state.info = info;
    },
  },

  actions: {
    // 登录
    Login({ commit }, userInfo) {
      return new Promise((resolve, reject) => {
        // 1、请求登录接口 login
        // 2、commit("SET_TOKEN", data.token);
      });
    },

    // 获取用户信息
    GetInfo({ commit }) {
      return new Promise((resolve, reject) => {
        // 1、请求获取用户信息 getInfo
        // 2、commit("SET_PERMISSIONS", data.permissions);
        // commit("SET_INFO", data);
      });
    },

    // 登出
    Logout({ commit, state }) {
      return new Promise((resolve) => {
        // 1、请求退出接口 logout
        // 2、commit("SET_TOKEN", "");
        // commit("SET_PERMISSIONS", []);
      });
    },
  },
};

export default user;
```

路由权限表：

JavaScript

```javascript
import { dynamicRoutes, staticRoutes } from "@/router/routes";

function hasPermission(permission, route) {
  // 判断是否有权限
}

function filterAsyncRouter(routes, permissions) {
  // 过滤有权限的路由
}

const permission = {
  state: {
    routers: staticRoutes,
    addRouters: [],
  },
  mutations: {
    SET_ROUTERS: (state, routers) => {
      state.addRouters = routers;
      state.routers = staticRoutes.concat(routers);
    },
  },
  actions: {
    GenerateRoutes({ commit }, data) {
      return new Promise((resolve) => {
        const { permissions } = data;
        const accessedRouters = filterAsyncRouter(dynamicRoutes, permissions);
        commit("SET_ROUTERS", accessedRouters);
        resolve(accessedRouters);
      });
    },
  },
};

export default permission;
```

8.5 路由动态生成

无论用户是否登录，访问任何一个路由做的其中一件事就是请求用户信息，并且在应用的整个生命周期都可以方便的获取。

也就是需要有一个地方统一校验当前用户的权限和可访问的路由表，这里需要用到路由拦截器，大概的实现逻辑如下：

JavaScript

```
import router from "@/router";
import store from "@/store";
import storage from "./helpers/storage";
import { ACCESS_TOKEN } from "./store/mutation-types";

const loginRoutePath = "/user/login";
const whiteList = ["login", "register", "registerResult"];

// 路由守卫：https://router.vuejs.org/zh/guide/advanced/navigation-guards.html
router.beforeEach((to, from, next) => {
  const token = storage.getItem(ACCESS_TOKEN);

  if (token) {
    if (to.path === loginRoutePath) {
      next({ path: "/" });
      return;
    }

    if (store.getters.permissions.length === 0) {
      // 1、store.dispatch("GetInfo")
      // 2、store.dispatch("GenerateRoute")
      // 3、addRouters.forEach((route) => router.addRoute(route));
    } else {
      next();
    }
  } else {
    if (whiteList.includes(to.name)) {
      // 在免登录名单，直接进入
      next();
    } else {
      next({ path: loginRoutePath, query: { redirect: to.fullPath } });
    }
  }
});
```

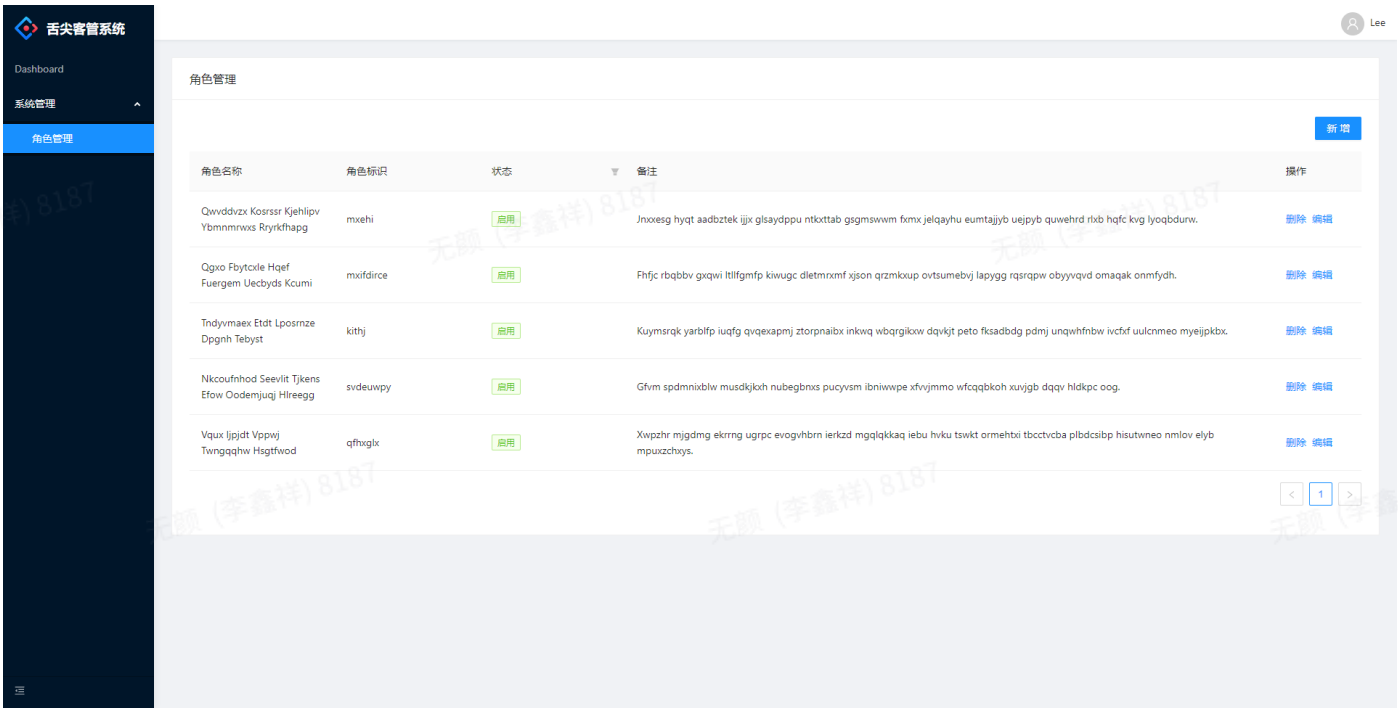
九、CURD

中后台常见的场景业务通常包括增删改查，即：

- 列表的展示

- 新增列表项
- 编辑列表项
- 删除列表项

下面带着大家简单实现一个基本角色管理，效果如图：



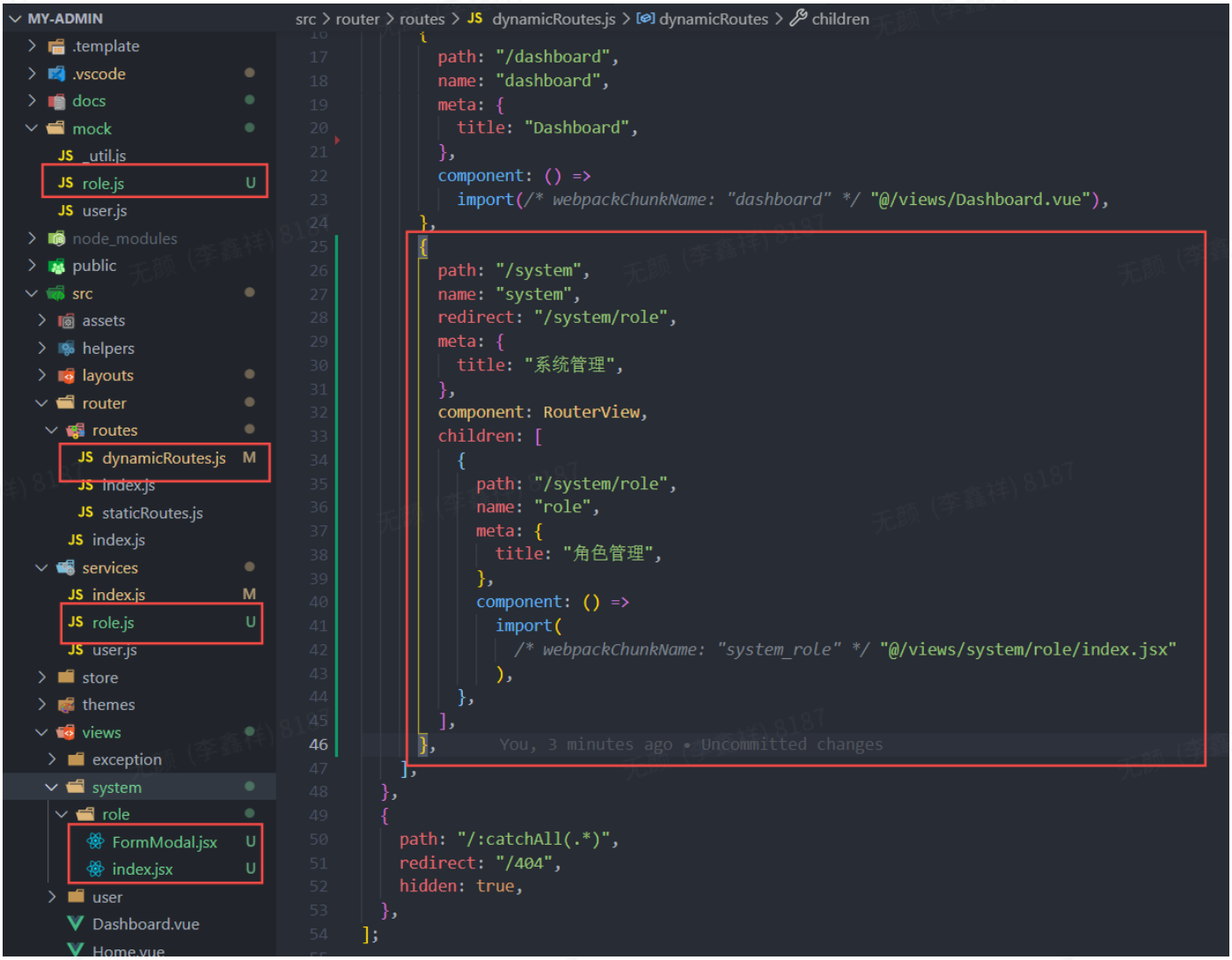
9.1 项目配置

常规的业务开发需要经过这几步：

- 路由配置
- 接口配置
- mock 数据

路由配置：

目录组织以及路由配置如下图：



接口配置：

接口满足 RESTful 规范，配置如下：

JavaScript

```javascript
import { request } from "@/helpers";

const userApi = {
  systemRole: "/api/system/role",
};

export function getRole(data) {
  return request({
    url: userApi.systemRole,
    method: "get",
    data,
  });
}

export function addRole(data) {
  return request({
    url: userApi.systemRole,
    method: "post",
    data,
  });
}

export function editRole(data) {
  return request({
    url: userApi.systemRole,
    method: "put",
    data,
  });
}

export function removeRole(data) {
  return request({
    url: userApi.systemRole,
    method: "delete",
    data,
  });
}
```

请求 URL 固定，通过方法来区分功能。如 GET、POST、PUT、DELETE 分别表示：查、新增、修改、删除。

Mock 数据：

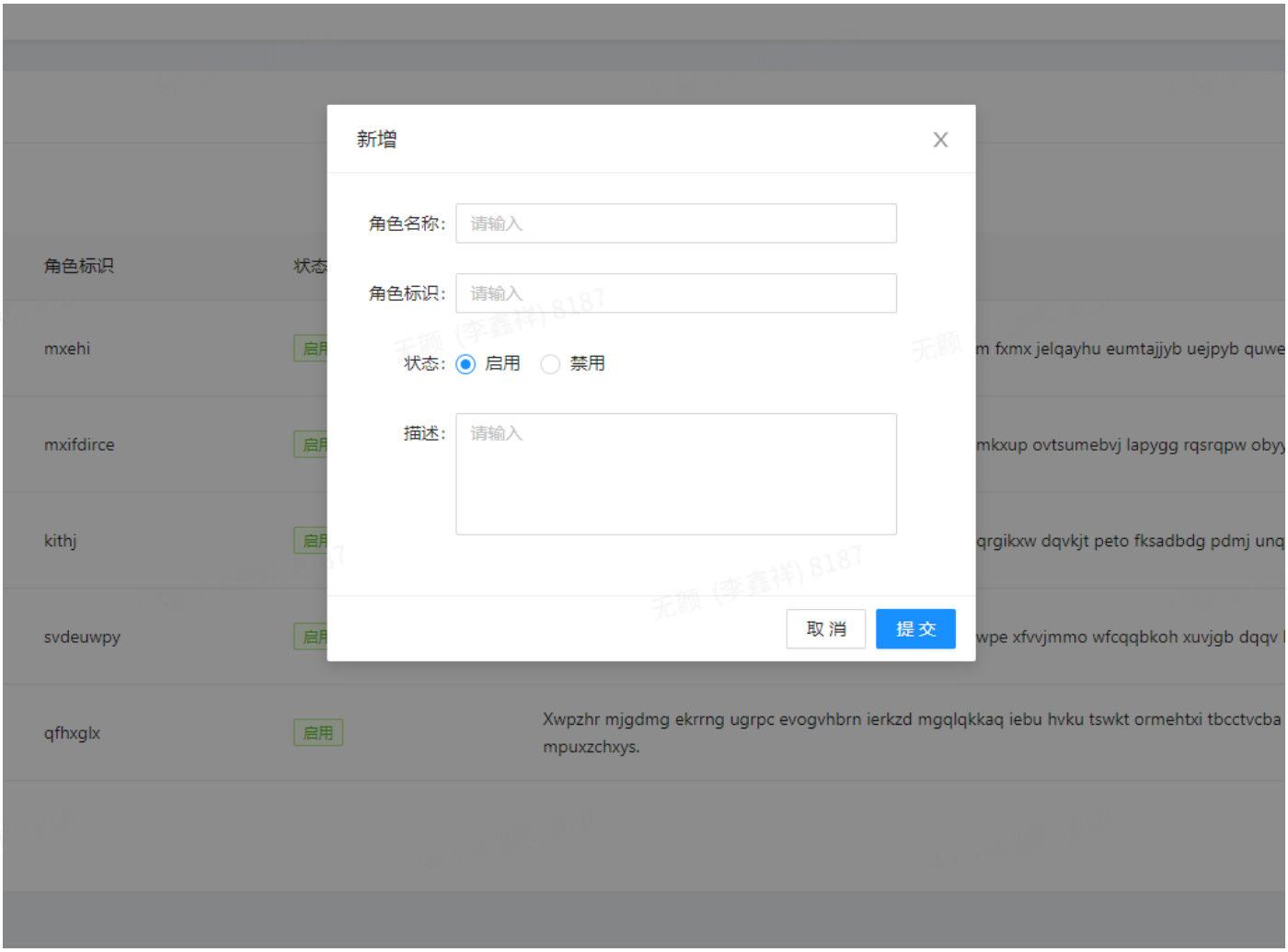
这里会用到 mockjs 用于模拟数据。

JavaScript

```
import Mock from "mockjs";
import { success, checkToken } from "./_util";

// http://mockjs.com/examples.html#Array
let { data } = Mock.mock({
  "data|5-10": [
    {
      id: "@guid",
      name: "@title",
      code: "@word",
      status: 1,
      desc: "@sentence",
    },
  ],
});

export default [
  {
    url: "/api/system/role",
    method: "post", // geh、put、delete
    response: (request) => {
      const { body } = request;

      // 新增
      data.push({ id: Mock.mock("@guid"), ...body });

      // 修改
      data = data.map((item) => {
        if (item.id === body.id) {
          return { ...item, ...body };
        }
        return item;
      });

      // 删除
      data = data.filter((item) => {
        return item.id !== body.id;
      });
    },
  },
];
```

9.2 业务开发

对于页面的布局，这里就不进行详细的说明。主要说一下，这里新增、编辑是采用弹窗的形式来实现的。如下：



表单主要用于数据收集，所有的业务逻辑在入口文件处理，核心逻辑如下 (为方便理解，仅保留关键逻辑):

TypeScript

```typescript
import { defineComponent, onMounted, reactive, ref } from "vue";
import { useAxios } from "ahooks-vue";
import { addRole, editRole, getRole, removeRole } from "@/services/role";
import FormModal from "./FormModal";

export default defineComponent({
  setup() {
    // 这里采用 hook 的方式来维护请求状态，参见: https://dewfall123.github.io/ahooks-vue/zh/use-axios/
    const { loading, data, run } = useAxios(getRole, { manual: true });
    const { run: add } = useAxios(addRole, { manual: true });
    const { run: edit } = useAxios(editRole, { manual: true });
```

```
  const { run: remove } = useAxios(removeRole, { manual: true });

  const visible = ref(false);

  // 通过 type 来区分是新增还是修改，这里可以用魔法字符串或枚举代替，具体看团队规范
  const type = ref("edit");

  const handleAdd = () => {
    visible.value = true;
    type.value = "add";
  };

  const handleEdit = (record) => {
    visible.value = true;
    type.value = "edit";
  };

  const handleRemove = async (record) => {
    await remove({ id: record.id });
    await run();
  };

  const handleCancel = () => {
    visible.value = false;
  };

  // 表单提交逻辑
  const handelOk = async (values) => {
    if (type.value === "add") {
      await add(values);
    }

    if (type.value === "edit") {
      await edit({ id: currentData.id, ...values });
    }

    run();

    visible.value = false;
  };

  onMounted(() => {
    run();
  });

  return () => (
    <Card title="角色管理">
```

```
        <Space direction="vertical" size="middle" style="width:100%">
          <Row type="flex" justify="end">
            <Col>
              <Button type="primary" onClick={handleAdd}>
                新增
              </Button>
            </Col>
          </Row>
          <Table
            rowKey="id"
            columns={columns}
            dataSource={data.value}
            loading={loading.value}
          />
          <FormModal
            title={type.value === "add" ? "新增" : "编辑"}
            visible={visible.value}
            onOk={handelOk}
            onCancel={handleCancel}
          />
        </Space>
      </Card>
    );
  },
});
```

这里仅仅简单的 CURD 示例，更多细节可以查看 源码。