10 Ajax 网络请求

准备知识

HTTP 协议

XMLHttpRequest

```
var xhr = new XMLHttpRequest();
xhr.open('get', '/api/test', true);
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var result = JSON.parse(xhr.responseText);
    }
}
xhr.send();
```

fetch

```
fetch('/api/test')
    .then(res => res.json())
    .then(data => console.log(data))
```

axios

搭建 mock 服务

> mock 服务用于模拟后端数据

技术选型

mock.js ×

1. 使用步骤：**定义**要模拟的路由和返回的结果 → mock.js **劫持** ajax 请求，返回模拟的结果
2. 安装相关依赖

```
npm install mockjs --save
npm install @types/mockjs --save-dev
```

> 安装 mockjs 会收到警告：mockjs vulnerable to Prototype Pollution via the Util.extend function。解决方式见，Snyk Vulnerability Database | Snyk

3. **使用示例**

   a. **定义模拟路由和返回结果**

```
import Mock from 'mockjs';

Mock.mock('/api/test', 'get', () => {
  return {
    errno: 0,
    data: {
      name: `yiTuChuan ${Date.now()}`,
    },
  };
});
```

   b. **使用 axios 发送 XMLHttpRequest 请求，测试拦截效果**

```
import axios from 'axios';
import '@/_mocks';

// ...
axios.get('/api/test').then(res => console.log('axios data', res));
// ...
```

4. 注意事项

   a. mock.js **只能劫持 XMLHttpRequest 请求**，不能劫持 fetch 类型的请求，有局限性

   b. 在生产环境（即上线时）要注释掉相关代码，否则会导致线上请求被劫持，可能会导致出错

mock.js + nodejs 服务（koa）√

> 这里没有使用 mockjs 的请求劫持功能，而是使用了其随机生成数据的功能

1. 安装相关依赖

```
npm install mockjs --save
npm install koa koa-router --save
```

```
npm install nodemon --save-dev
```

2. **使用示例**

   a. **创建一个新项目 wenjuan-mock，并使用 `npm init` 初始化。**

   b. **定义 mock 路由信息**

      i. /mock/test.js

```
const Mock = require("mockjs");

const Random = Mock.Random;

module.exports = [
  {
    url: "/api/test",
    method: "get",
    response() {
      return {
        errno: 0,
        data: {
          name: Random.cname(),
        },
      };
    },
  },
];
```

      ii. /mock/question.js

```
const Mock = require("mockjs");

const Random = Mock.Random;

module.exports = [
  {
    url: "/api/question/:id",
    method: "get",
    response() {
      return {
        errno: 0,
        data: {
          id: Random.id,
```

```
          title: Random.ctitle,
        },
      };
    },
  },
];
```

iii. /mock/index.js

```
const test = require("./test");
const question = require("./question");

const mockList = [...test, ...question];

module.exports = mockList;
```

iv. ...

c. **使用 nodejs + koa 启动服务**

i. **注册路由并监听端口** /index.js

```
const Koa = require("koa");
const Router = require("koa-router");
const mockList = require("./mock/index");

const app = new Koa();
const router = new Router();

/*
 * 注册 mock 路由
 * 遍历 mockList 中的每个路由配置，依次为其注册路由处理程序
 */
mockList.forEach((route) => {
  const { url, method, response } = route;
  router[method](url, async (ctx) => {
    const res = await getRes(response);
    ctx.body = res;
  });
});

/*
 * 使用 router 中定义的路由
 * 将路由中间件挂载到 Koa 应用实例上
```

```
 */
app.use(router.routes());

/*
 * 启动服务器，监听 3000 端口
 * 成功启动后输出详细的启动信息
 */
app.listen(3000, () => {
  console.log("Server is running on http://localhost:3000");
});

/**
 * 模拟异步获取响应的函数
 * 使用 setTimeout 模拟异步操作，1 秒后返回 response 函数的结果
 * @param {Function} fn - 用于生成响应数据的函数
 * @returns {Promise<any>} - 包含响应数据的 Promise
 */
async function getRes(fn) {
  return new Promise((resolve) => {
    setTimeout(() => {
      const res = fn();
      resolve(res);
    }, 1000);
  });
}
```

ii. **在 package.json 中配置 dev 命令**，用于快速启动服务

```
{
  "scripts": {
    "dev": "nodemon index.js"
  },
}
```

d. 此时可以使用 `npm run dev` 启动服务

3. 注意事项

a. 这里使用了 mock.js 的 Random 能力

b. 这里考虑到了多模块的扩展性，将所有路由规则定义在 `_mock` 文件夹下，并将所有路由规则合并到 `_mock/index.js` 中

在线 mock 平台 ×

fast-mock、y-api、swagger 等

> **不建议使用**，存在数据泄露风险；同时可能存在网络不稳定的问题

API 设计 —— Restful API

> API 的统一返回格式为 `{ errno, data, msg }`

用户 API

- 注册
  - method `post`
  - path `/api/user/register`
  - request body `{ username, password, nickname }`
  - response `{ errno: 0 }`
- 登录
  - method `post`
  - path `/api/user/login`
  - request body `{ username, password }`
  - response `{ errno: 0, data: { token } }` —— **JWT** 使用 token
- 获取用户信息
  - method `get`
  - path `/api/user/info`
  - response `{ errno: 0, data: {...} }` 或 `{ errno: 10001, msg: 'xxx' }`

问卷 API

- 创建问卷
  - method `post`
  - path `/api/question`
  - request body - 无 （点击一个按钮即可创建，title 自动生成）
  - response `{ errno: 0, data: { id } }`
- 获取单个问卷信息
  - method `get`
  - path `/api/question/:id`

    - response `{ errno: 0, data: { id, title ... } }`
- 更新问卷（删除是假删除，通过 isDeleted 属性进行控制，本质是更新 isDeleted 属性）
    - method `patch`
    - path `/api/question/:id`
    - request body `{ title, isStar ... }`
    - response: `{ errno: 0 }`
- 删除问卷（彻底删除）
    - method `delete`
    - path `/api/question`
    - request body `{ ids: [ ... ] }`
    - response: `{ errno: 0 }`
- 查询问卷列表
    - method `get`
    - path `/api/question`
    - response: `{ errno: 0, data: { list: [ ... ], total } }`
- 复制问卷
    - method `post`
    - path `/api/question/duplicate/:id`
    - response: `{ errno: 0, data: { id } }`

axios 的配置

二次封装 axios

```
// src/service/ajax.ts
/* eslint-disable @typescript-eslint/no-explicit-any */
import axios from 'axios';
import { message } from 'antd';

const instance = axios.create({
  timeout: 1000 * 10, // 指定请求超时的毫秒数（10000 毫秒，即 10 秒）
});

// 添加响应拦截器
instance.interceptors.response.use(
```

```
  function (response) {
    // 该函数会在 HTTP 响应状态码在 2xx 范围内时被触发
    // 对响应数据进行处理
    const resData: ResType = response.data || {};
    const { errno, data, msg = '来自于 axios 响应拦截器的未知错误' } = resData;

    /* 表示相应的操作失败 */
    if (errno !== 0) {
      message.error(msg); // 显示错误消息
      throw new Error(msg); // 抛出错误，阻止后续操作
    }
    return data as any; // 返回响应数据的 data 部分
  },
  function (error) {
    // 该函数会在 HTTP 响应状态码超出 2xx 范围时被触发
    // 对响应错误进行处理
    return Promise.reject(error); // 返回一个被拒绝的 Promise，并传递错误信息
  }
);

// 定义响应数据的类型
export type ResType = {
  errno: number; // 错误码，0 表示成功，非 0 表示失败
  data?: ResDataType; // 可选的数据字段，包含实际响应的数据
  msg?: string; // 可选的消息字段，包含错误或提示信息
};

// 定义实际响应数据的类型
export type ResDataType = {
  [key: string]: any; // 任意数量的字符串属性
};

export default instance;
```

封装对应 API 请求

以问卷 API 的获取问卷信息的 API 封装为例，

```
// src/service/question.ts
import instance from '../ajax';
import { ResDataType } from '../ajax';

export async function getQuestionService(id: string): Promise<ResDataType> {
  const url = `/api/question/${id}`;
  const data = await instance.get(url);
```

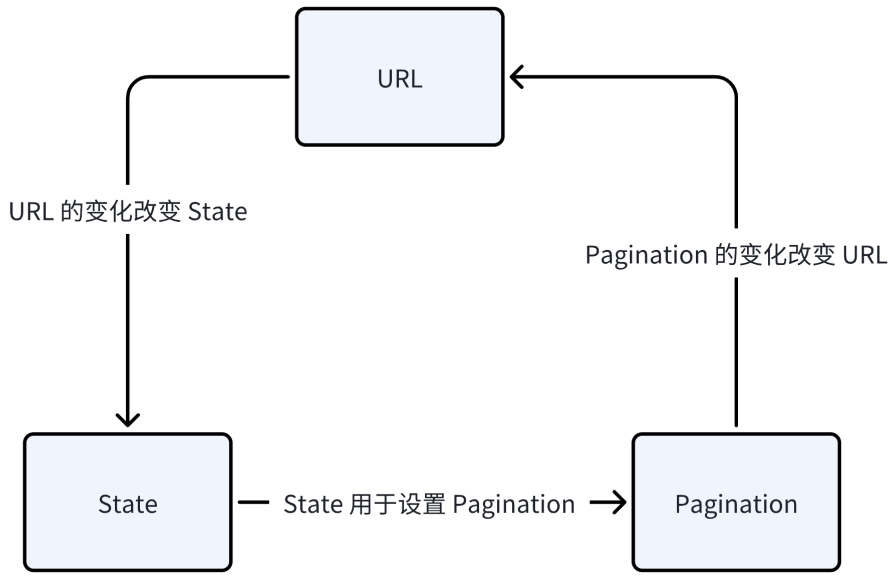
```
8   return data;
9 }
```

useRequest —— 异步请求数据

分页 pagination

分页组件变化 → 修改 url 参数 → 修改 State → 修改分页组件

> 在修改 url 参数后，列表页会监测 url 参数的变化请求新的数据



下拉更多 loadMore

最底下的 div 出现在视口 → 加载更多（看似简单，实际麻烦）

- 考虑防抖：即多次下拉只能响应一次
- 考虑加载更多时机：最底下 div 出现在视口时 Element.getBoundingClientRect() + 用户下拉 scroll + 还有更多数据

JWT 用户令牌

JWT 全称为 JSON Web Token，**用户登录成功后，服务端返回一个 token**（称之为令牌，是一段字符串），之后**每次请求都要携带这个 token 以表明自己的身份（通过 axios 的请求拦截器实现）**。

附

Postman 请求测试

Vite 跨域请求配置

> create-react-app 跨域配置见 craco

在 vite.config.ts 中添加以下配置，

```
export default defineConfig({
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3000', // 目标地址
        changeOrigin: true,
      },
    },
  },
});
```

Vscode 插件的同步

1. 导出插件列表 `code --list-extensions > extensions.txt`
2. 根据插件列表下载插件 `cat extensions.txt |% { code --install-extension $_ }`

- 搭建 mock 服务
- API 设计（使用 Restful API）
- 实战：为列表页、登录页、注册页，增加 Ajax 请求