VUE+Element 前端应用

开发说明书

V1.0

序号	修改人	修改日期	修改后版本	修改说明
1	伍华聪	2021-5-30	V1.0	初稿

目 录

1. 引言

1.1. 背景

之前一直采用 VS 进行各种前端后端的开发，随着项目的需要，正逐步融合纯前端的开发模式，开始主要选型为 Vue + Element 进行 BS 前端的开发，后续会进一步整合 Vue + AntDesign 的界面套件，作为两种不同界面框架的展现方式。采用 Vue + Element 的前端开发和以期的开发方式有较大的转变，以及需要接触更多的相关知识。

Vue + Element 的前端开发方式，和以前的开发方式有很大的不同，不再依赖于 VS（Visual Studio）开发 IDE，而是采用微软另外一套轻型的开发 IDE--VS Code，VS Code 是跨平台的应用，在 Mac、Windows 平台均可使用，这样开发前端不用依赖于 Windows 环境了。VS Code 虽然是轻型开发工具，不过功能也是非常强大的，而且开发环境可以在 Windows 系统，也可以在 Mac 系统等，实现了多平台的开发环境。

Vue + Element 的前端开发方式，主要就是利用 node.js 的开发环境，使用各种前端框架或者组件，实现对前端视图页面和 JS 代码的统一整合。这里完全没有 C#代码，也没有 Java 代码，而是通过 JS 前端框架和后端 API 平台实现数据交互/或者于本地 Mock 服务交互。

1.2. 编写目的

本文档主要介绍《Vue + Element 前端框架》的特性以及如何整合后端 Web API 框架进行使用，通过前后端的整合，实现一个完整、强大的后台管理系统业务。

1.3. 参考资料

序号	名称	版本/日期	来源
1	《Bootstrap 开发框架-系统功能介绍白皮书.pdf》		内部
2	《伍华聪的博客》(http://wuhuacong.cnblogs.com)		博客园

3	《循序渐进开发 Web 项目操作说明书-Bootstrap.pdf》		内部
4	《Vue-Element 前端应用环境安装部署说明书.doc》		内部

1.4. 术语与缩写

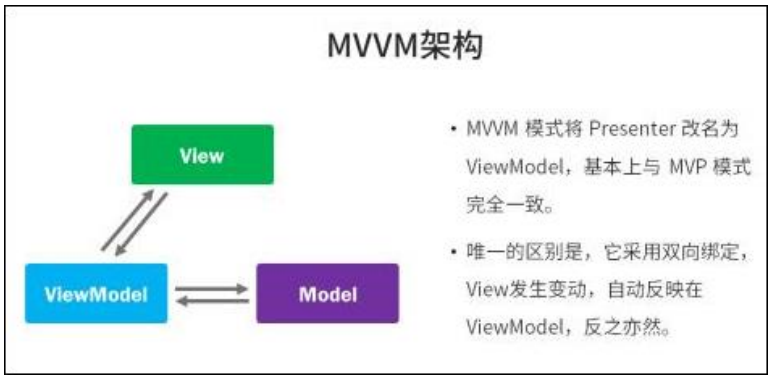
1 在本文件中出现的“系统”、“框架”一词，除非特别说明,均适用于《Bootstrap 开发框架》。

2 当前基于.netframework的版本为4.5.2,如无特殊说明，均指该版本。

2. 基础知识介绍

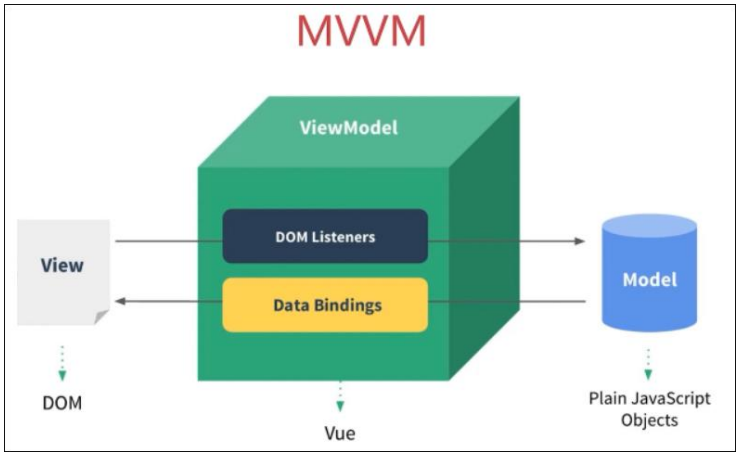
2.1. Vue 框架简介

Vue 是一套构建用户界面的框架， 开发只需要关注视图层， 它不仅易于上手，还便于与第三方库或既有项目的整合；另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。是基于 MVVM（Model-View-ViewModel）设计思想。提供 MVVM 数据双向绑定的库，专注于 UI 层面。



View 就是 DOM 层，ViewModel 就是通过 new Vue()的实例对象，Model 是原生 js。开发者修改了 DOM，ViewModel 对修改的行为进行监听，监听到了后去更改 Model 层的数据，然后再通过 ViewModel 去改变 View，从而达到自动同步。

Vue 核心思想，包括数据驱动、组件化等方面。



1、数据驱动

数据驱动（数据双向绑定），　在 Vue 中，Directives 对 view 进行了封装，当 model 中的数据发生变化时，Vue 就会通过 Directives 指令去修改 DOM，同时也通过 DOM Listener 实现对视图 view 的监听，当 DOM 改变时，就会被监听到，实现 model 的改变，从而实现数据的双向绑定。

2、组件化

组件化就是实现了扩展 HTML 元素，封装可用的代码。

1）页面上每个独立的可视/可交互区域视为一个组件。

2）页面不过是组件的容器，组件可以嵌套自由组合形成完整的页面

2.2. Element 介绍

Element，是饿了么公司开发的一套 UI 组件，一套为开发者、设计师和产品经理准备的基于 Vue 2.0 的桌面端组件库。提供了配套设计资源，帮助你的开发快速成型。由饿了么公司前端团队开源。

Element 前端界面套件，提供了几乎所有常见的 Web 组件封装，并扩展了很多功能丰富的 UI 组件，使用这些界面组件可以极大提高 Web 界面的开发效率，同时也能够把这些基础组件封装层更高级、功能更丰富的自定义组件。

2.3. 开发所需的软件环境

有别于之前的 Asp.net 的开发，纯前端的开发，一般不会再采用笨重的 VS 进行前端的开发，而改用 VS Code 或者 WebStorm 等轻型的开发工具来进行前端代码的开发和维护，虽

然是轻型开发工具，不过功能也是非常强大的，而且开发环境可以在 Windows 系统，也可以在 Mac 系统等，实现多平台的开发环境。

2.3.1. VS code 的安装

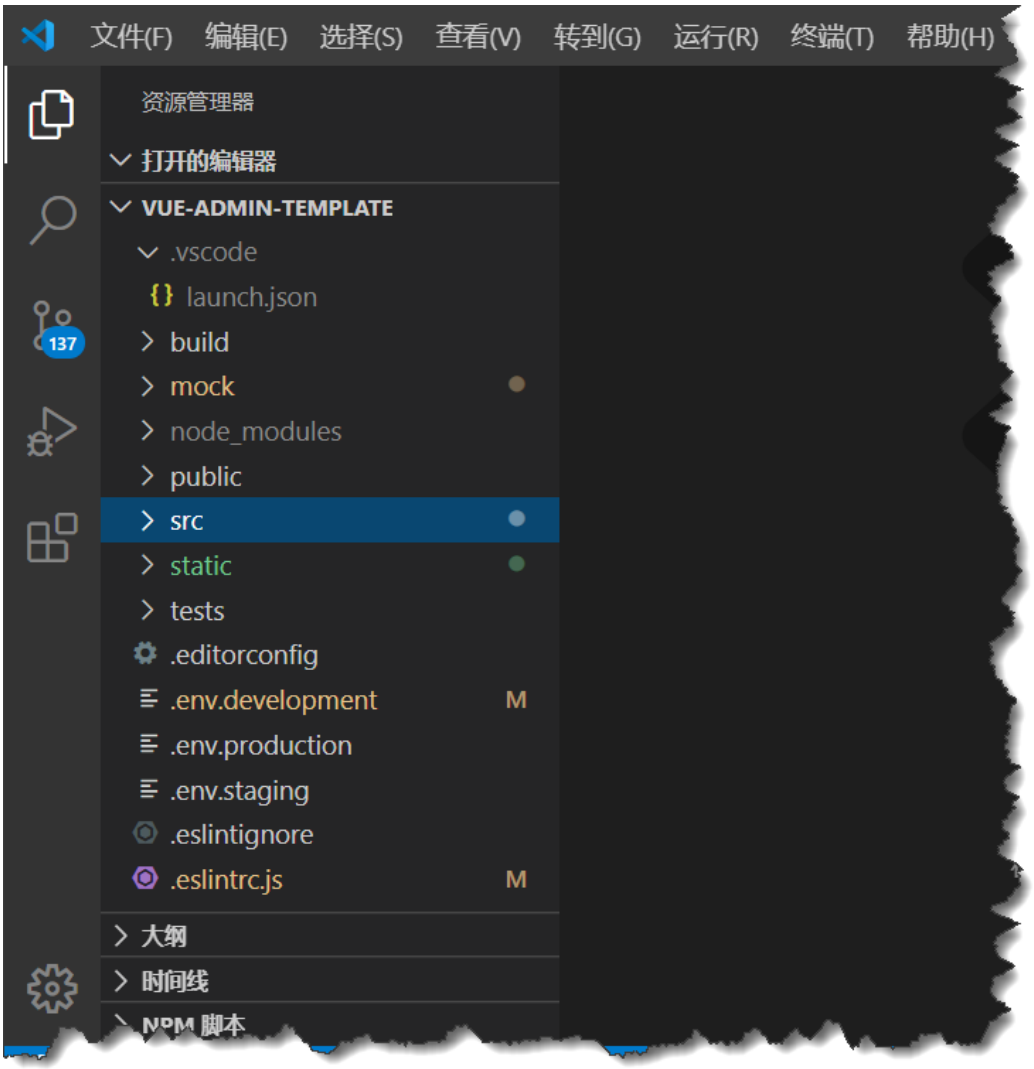
VS Code（Visual Studio Code）是由微软研发的一款免费、开源的跨平台文本（代码）编辑器。几乎完美的编辑器。

官网：https://code.visualstudio.com

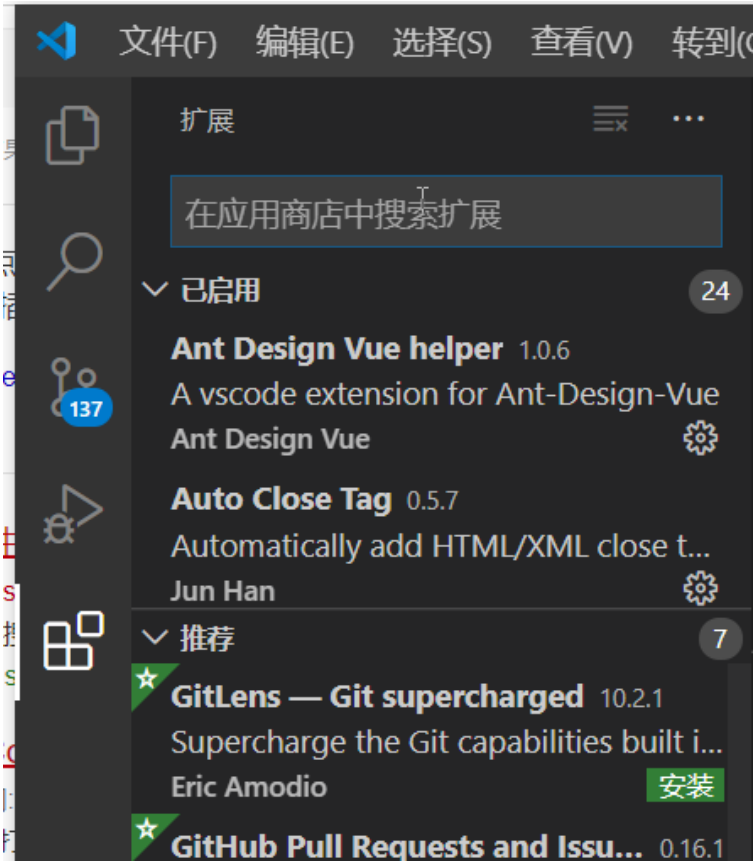
文档：https://code.visualstudio.com/docs

源码：https://github.com/Microsoft/vscode

VS Code 的界面大概如下所示，一般安装后，如果为英文界面，则安装它的中文包即可。



VS Code 安装后，我们一般还需要搜索安装一些所需要的插件辅助开发。安装插件很简单，在搜索面板中查找到后，直接安装即可。



一般我们需要安装这些 vs code 插件:

Vetur

Vue 多功能集成插件，包括：语法高亮，智能提示，emmet，错误提示，格式化，自动补全，debugger。vscode 官方钦定 Vue 插件，Vue 开发者必备。

ESLint

ESLint 是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。

而 VSCode 中的 ESLint 插件就直接将 ESLint 的功能集成好，安装后即可使用，对于代码格式与规范的细节还可以自定义，并且一个团队可以共享同一个配置文件，这样一个团队所有人写出的代码就可以使用同一个代码规范，在代码签入前每个人可以完成自己的代码规范检查。

VS Code - Debugger for Chrome 结合 Chrome 进行调试的插件

此工具简直就是前端开发必备，将大大地改变你的开发与调试模式。

以往的前端调试，主要是 JavaScript 调试，你需要在 Chrome 的控制台中找到对应代码的部分，添加上断点，然后在 Chrome 的控制台中单步调试并在其中查看值的变化。

而在使用了 Debugger for Chrome 后，当代码在 Chrome 中运行后，你可以直接在 VSCode 中加上断点，点击运行后，Chrome 中的页面继续运行，执行到你在 VSCode 中添加的断点后，你可以直接在 VSCode 中进行单步调试。

Beautify

Beautify 插件可以快速格式化你的代码格式，让你在编写代码时杂乱的代码结构瞬间变得非常规整，代码强迫症必备，较好的代码格式在后期维护以及他人阅读时都会有很多的便利。

2.3.2. 安装 node 开发环境

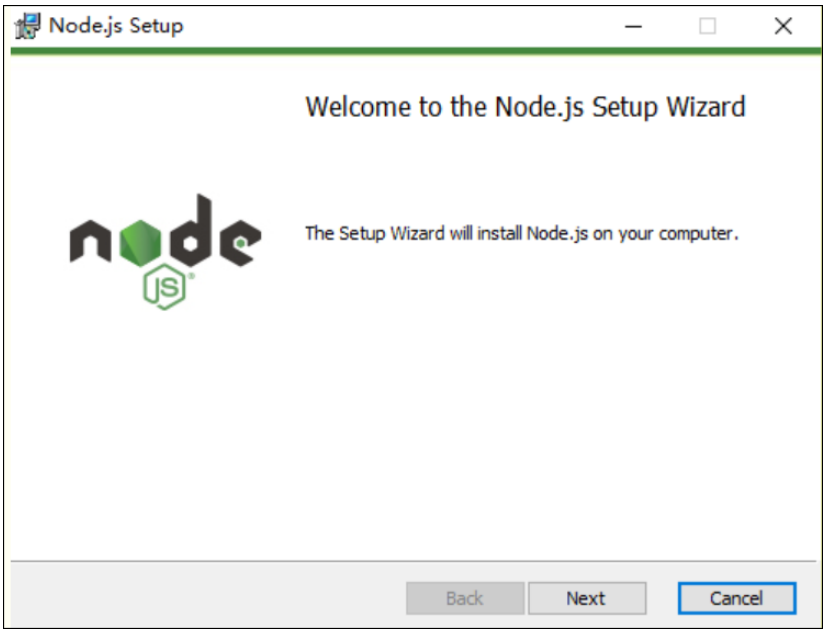
利用 VS Code 开发，我们很多时候，需要使用命令行 npm 进行相关模块的安装，这些需要 node 环境的支持，安装好 node 后，npm 也就一起安装好了。

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。

Node.js 的包管理器 npm，是全球最大的开源库生态系统。

node 下载：https://nodejs.org/en/



安装后，我们可以通过命令行或者 VS Code 里面的 Shell 进行查看 node 和 npm 的版本号了

node -v

npm -v

2.3.3. vue 脚手架的安装

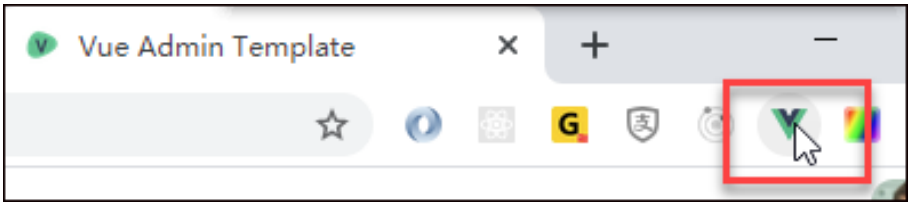
Vue (读音 /vju:/，类似于 view) 是一套用于构建用户界面的渐进式框架。

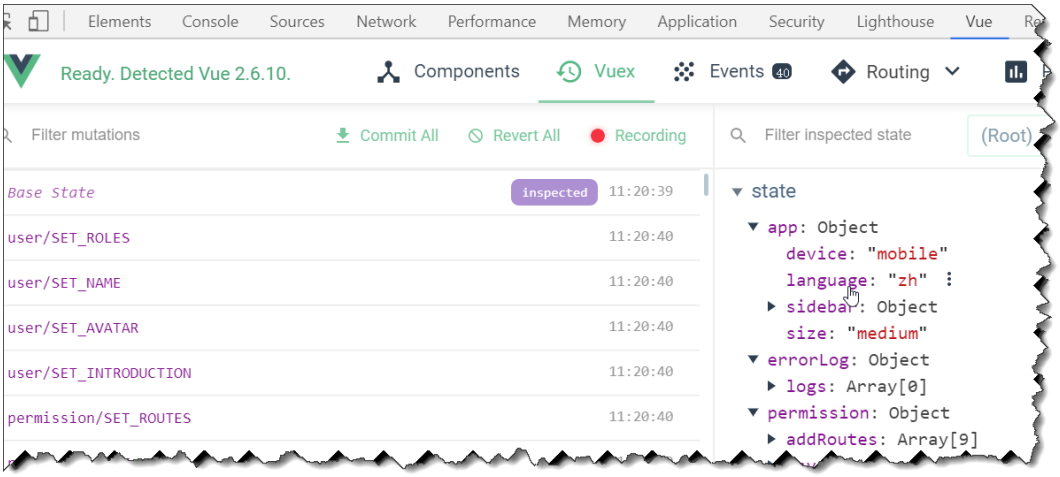
全局安装：npm install vue-cli -g （全局卸载：npm uninstall vue-cli -g）

2.3.4. Vue DevTool Chrome 插件的安装

这个插件也是开发 Vue 必备的 Chrome 插件，一般没有外网，不能直接在 Chrome 的插件官网上进行安装，而通过 GitHub 下载进行编译在安装又显得太过麻烦，后来在一个网站上下载安装成功。

https://chrome.zzzmh.cn/info?token=nhdogjmejiglipccpnnnanhbledajbpd





2.3.5. 开发环境的配置使用

对于 **Vetur** 等代码自动修正处理，我们需要在 VS Code 里面进行设置好，在【文件】【首选项】【设置】中，然后单击 Settings.json 进行编辑即可。

设置 { } settings.json ×

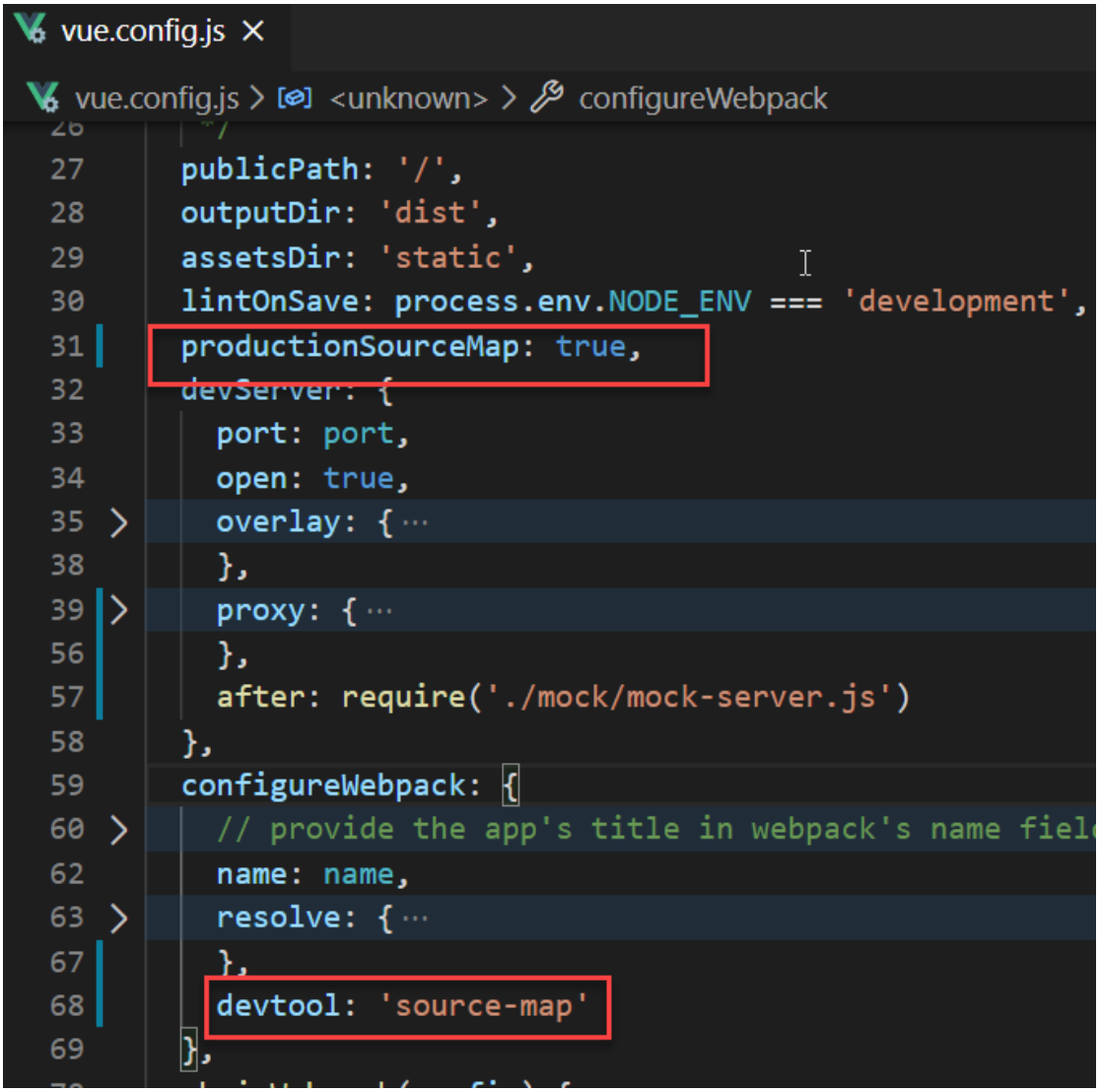
C: > Users > Administrator > AppData > Roaming > Code > User > { } settings.json > { } vetu

```
    // 强制单引号
    "prettier.singleQuote": true,
    // 尽可能控制尾随逗号的打印
    "prettier.trailingComma": "all",
    // 开启 eslint 支持
    "prettier.eslintIntegration": true,
    // 使用插件格式化 html
    "vetur.format.defaultFormatter.html": "js-beautify-html",
    // 格式化插件的配置
    "vetur.format.defaultFormatterOptions": {
        "js-beautify-html": {
            // 属性强制折行对齐
            "wrap_attributes": "force-aligned",
        }
    },
    "editor.codeActionsOnSave": {
        "source.fixAll.eslint": true,
    },
    "editor.quickSuggestions": {
        "strings": true
    },
```

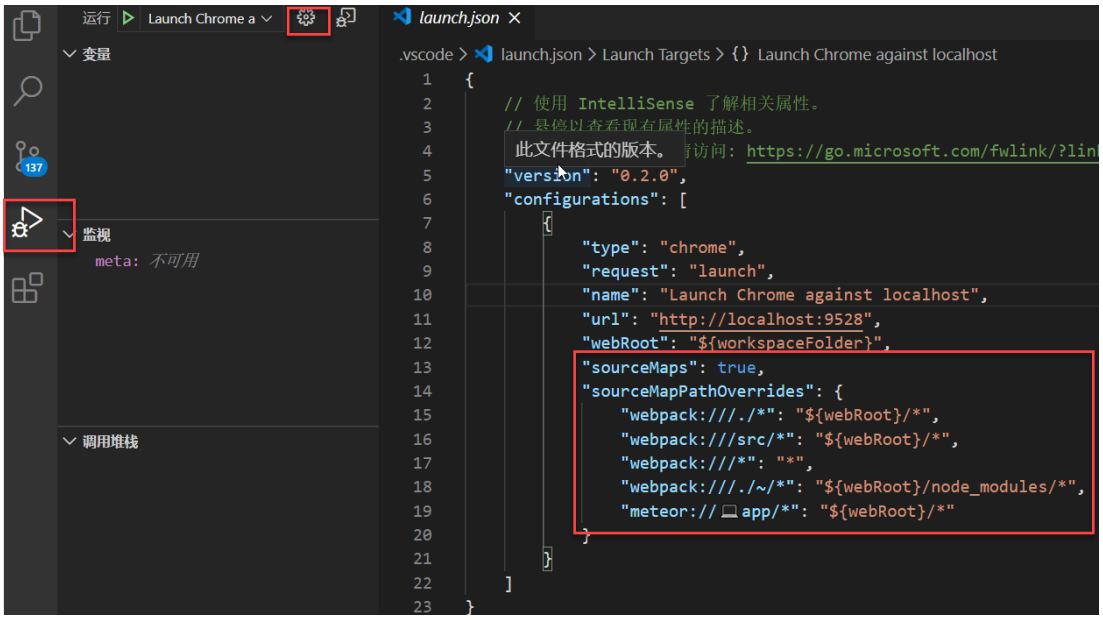
我这里主要设置保存代码后能够对代码进行缩进排版的常规的处理

调试环境的处理，为了结合 Chrome 调试 VScode，我们需要安装插件 Debugger for Chrome ，然后进行 Vue 项目代码的设置处理即可。

打开项目根目录的 Vue.Config.js 文件，在合适的位置，加入 productionSourceMap: true 以及 devtool: 'source-map' 如下所示

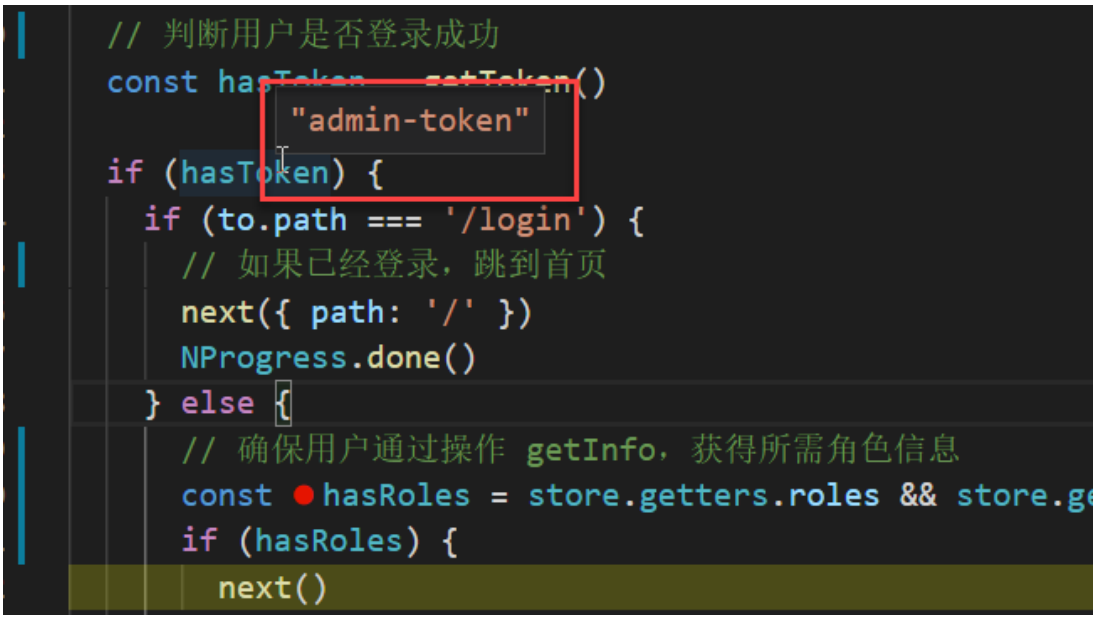


然后再在运行面板里面，进行调试参数设置的处理，如下所示



指定这些设置后，我们就可以以调试模式进行调试 VS Code 里面的代码了，代码只需要设置对应的断点即可跟踪对象的数据。

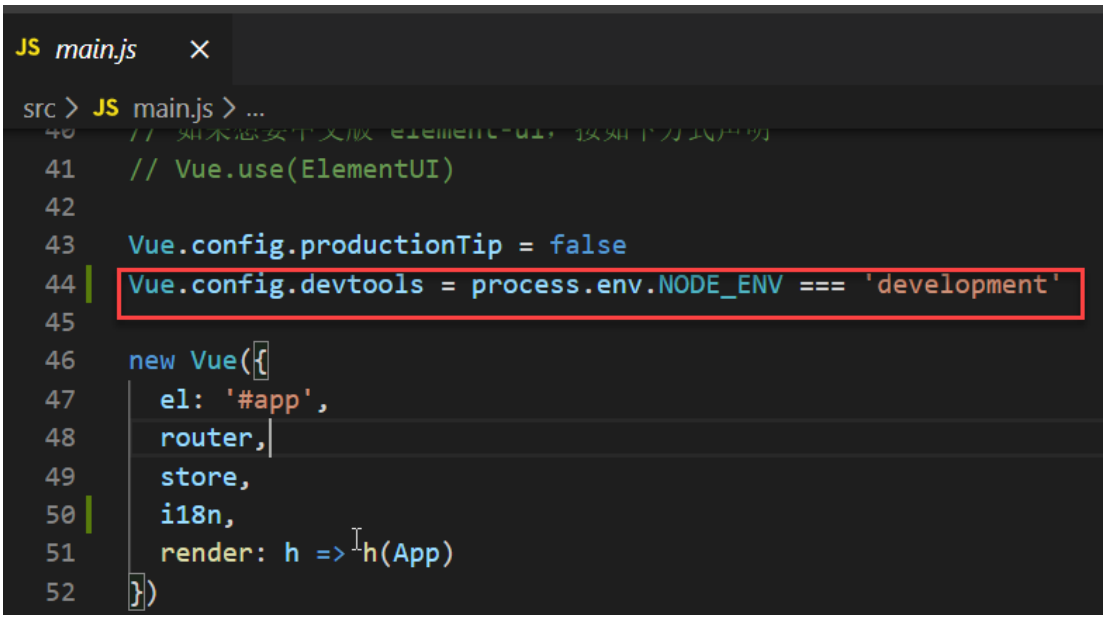
调试前，记得先使用 npm run dev 启动项目，项目完全启动后会在 Chrome 浏览器打开项目地址，再使用 F5 进行项目代码的调试。

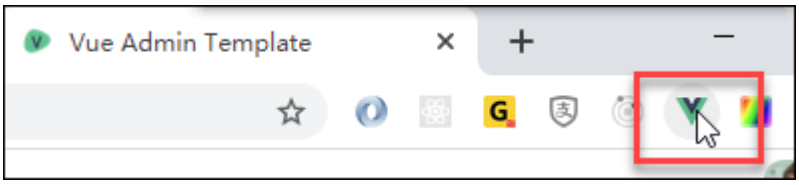


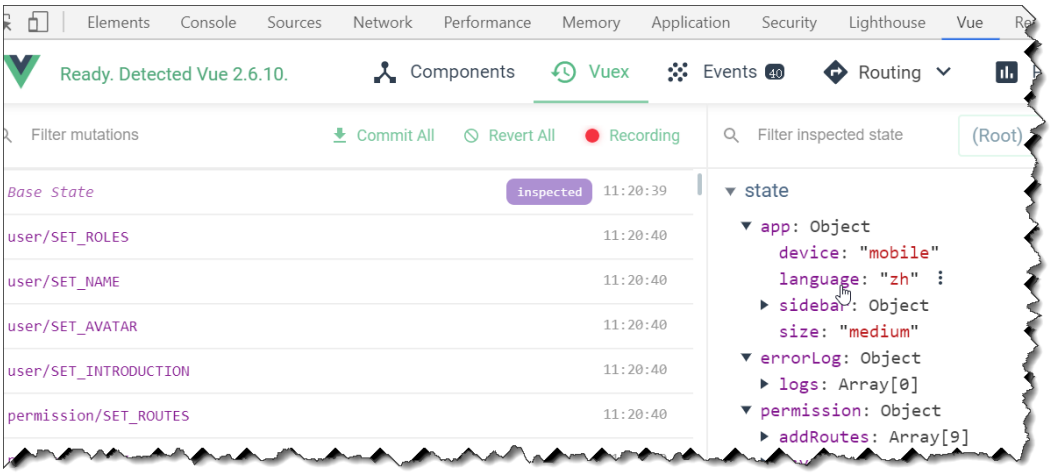
Vue DevTools 也是用来跟踪 Vue 项目路由、状态等信息的，可以信息很好的跟踪处理。为了点亮 Chrome 浏览器上面 Vue DevTools 图标，我们可以在 Vue 项目的 main.js 里面加入一行代码。

```
Vue.config.devtools = process.env.NODE_ENV === 'development'
```

如下界面所示







3. 前端开发知识介绍

3.1. Vuex 中的 API、Store 和 View 的使用

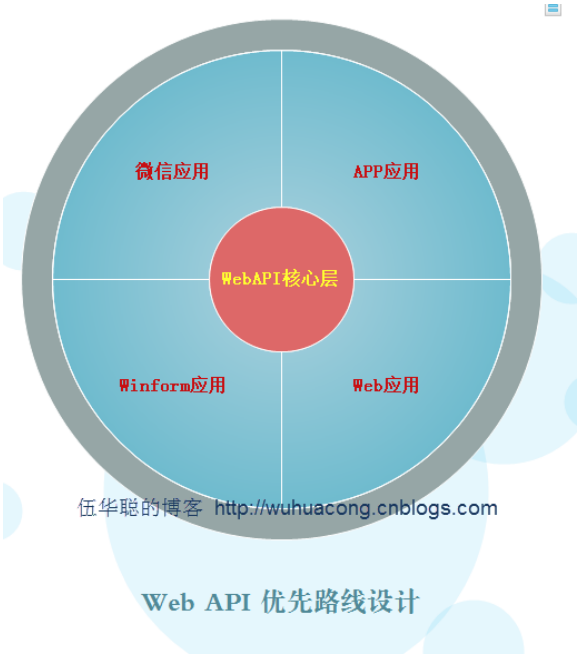
在我们开发 Vue 应用的时候，很多时候需要记录一些变量的内容，这些可以用来做界面状态的承载，也可以作为页面间交换数据的处理，处理这些内容可以归为 Vuex 的状态控制。例如我们往往前端需要访问后端数据，一般是通过封装的 Web API 调用获取数据，而使用 Store 模式来处理相关的数据或者状态的变化，而视图 View 主要就是界面的展示处理。这里主要介绍如何整合这三者之间的关系，实现数据的获取、处理、展示等逻辑操作。

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。关于 Vuex 的相关 State、Getter、Mutation、Action、Module 之间的差异和联系，详细可以参考下：https://vuex.vuejs.org/zh/ 。

3.1.1. 前后端的分离和 Web API 优先路线设计

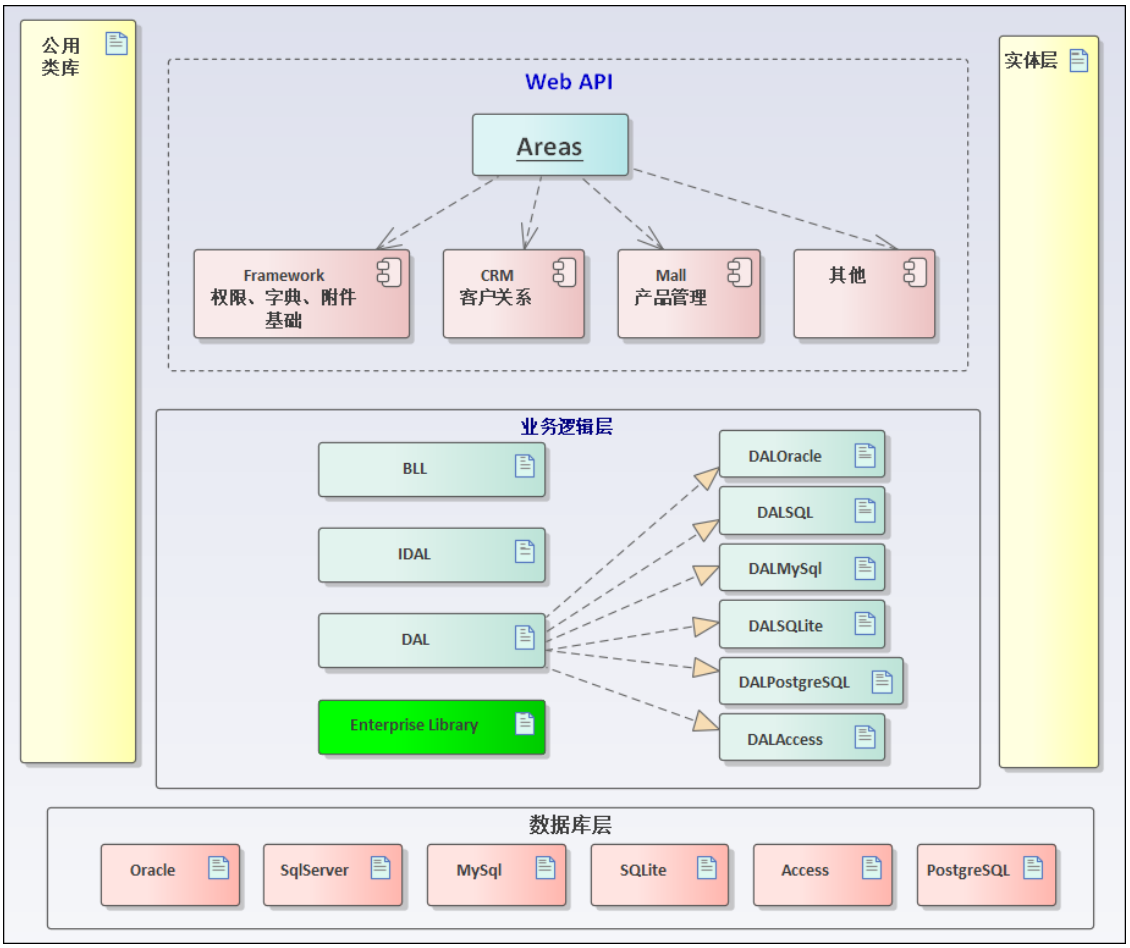
Web API 是一种应用接口框架，它能够构建 HTTP 服务以支撑更广泛的客户端（包括浏览器，手机和平板电脑等移动设备）的框架， ASP.NET Web API 是一种用于在 .NET Framework/ .net Core 上构建 RESTful 应用程序的理想平台。

在目前发达的应用场景下，我们往往需要接入 Winform 客户端、APP 程序、网站程序、以及微信应用等程序，这些数据应该可以由同一个服务提供，这个就是我们所需要构建的 Web API 平台。由于 Web API 层作为一个公共的接口层，我们就很好保证了各个界面应用层的数据一致性。

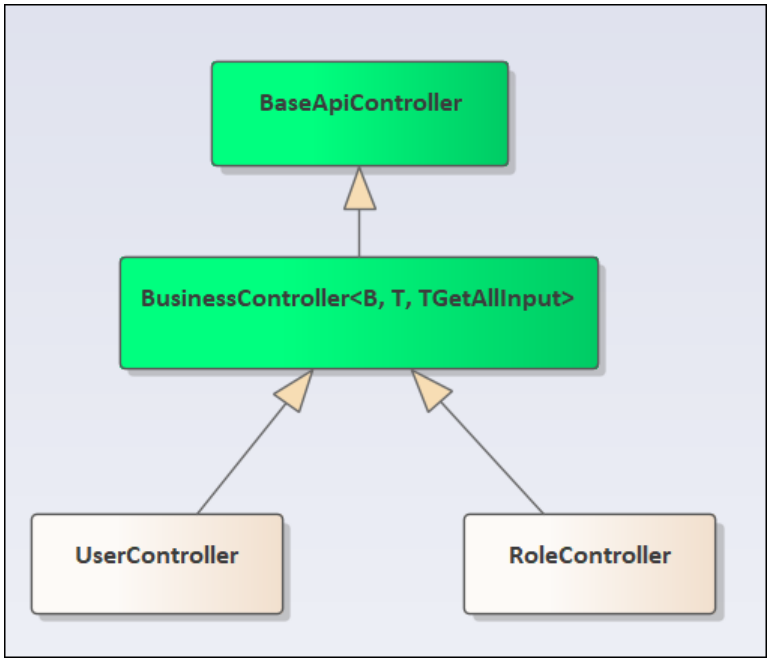


由于倾向于前后端的完全分离，我们后端就可以完全由 Web API 统一构建支持，可以采用.net framework 或者.net core 构建的统一接口平台。

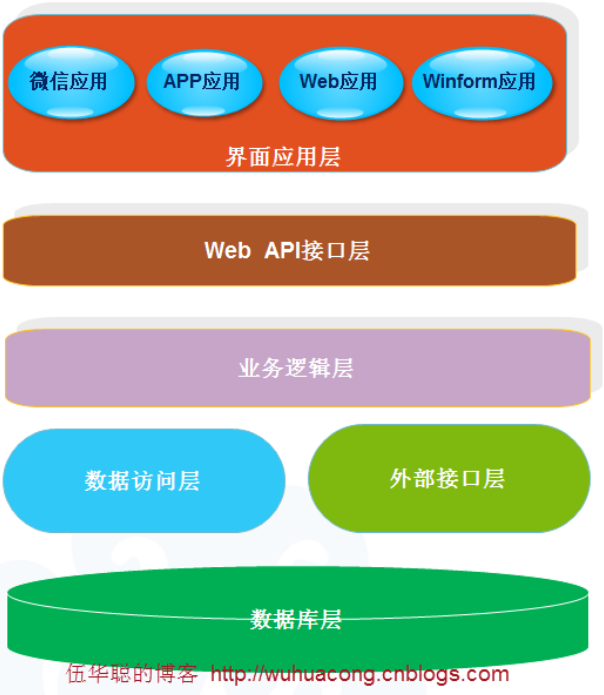
后端采用基于 Asp.net 的 Web API 技术，为前端提供相应的 API 接口服务，并提供按域来管理 API 的分类，Web API 如下架构所示。



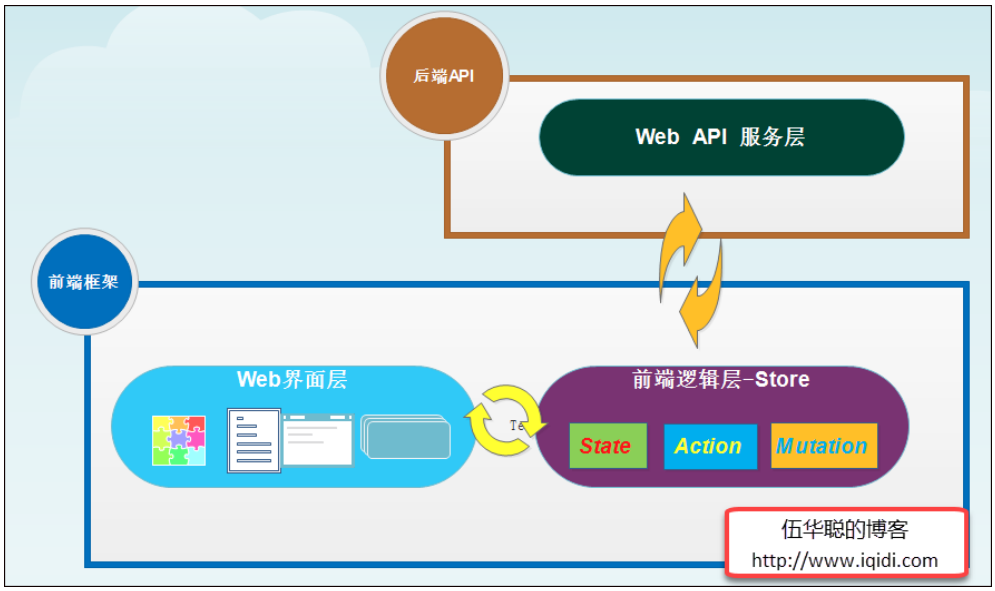
为了更好的进行相关方法的封装处理，我们把一些常规的接口处理放在 BaseApiController 里面，而把基于业务表的操作接口放在 BusinessController 里面定义，如下所示。



这样我们就可以基于这些 API 接口构建前端多项应用，如包括 Web 前端、Winform 前端、以及对接各种 APP 等应用。



引入了前后端分离的 VUE + Element 的开发方式，那么前后端的边界则非常清晰，前端可以在通过网络获取对应的 JSON 就可以构建前端的应用了。



在前端处理中，主要就是利用 Vuex 模式中的 Store 对象里实现对 Action 和 Mutation 的请求处理，获取数据后，实现对 State 状态中的数据进行更新。如果仅仅是当前页面的数据处理，甚至可以不需要存储 State 信息，直接获取到返回的数据，直接更新到界面视图上即可。

在开发前期，我们甚至可以不需要和后端发生任何关系，通过 Mock 数据代替从 Web API 上请求数据，只要 Mock 的数据结构和 Web API 接口返回的 JSON 一致，我们就可以在后期实现快速的对接，而不影响现有的代码处理方式。

3.1.2. Axios 网络请求处理

在我们进一步处理前，我们需要知道 Vuex 里面的一些对象概念和他们之间的关系。Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。关于 Vuex 的相关 State、Getter、Mutation、Action、Module 之间的差异和联系，详细可以参考下：https://vuex.vuejs.org/zh/

在开始发起网络请求之前，我们需要了解 axios 这个东西，axios 是一个基于 Promise 用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生 XHR 的封装，只不过它是 Promise 的实现版本，符合最新的 ES 规范。在这里我们只需要知道它是非常强大的网络请求处理库，且得到广泛应用即可，列举几个代码案例进行了解。

POST 请求：

```
axios({
    method: 'post',
    url: '/user/12345',
    data: {
        firstName: 'Fred',
        lastName: 'Flintstone'
    }
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

GET 请求：

```
axios
  .get('http://rap2api.taobao.org/app/mock/23080/resources/search',{
     params: {
        id: 5
     }
   })
  .then(res => {
    console.log('数据是:', res);
  })
  .catch((e) => {
    console.log('获取数据失败');
  });
```

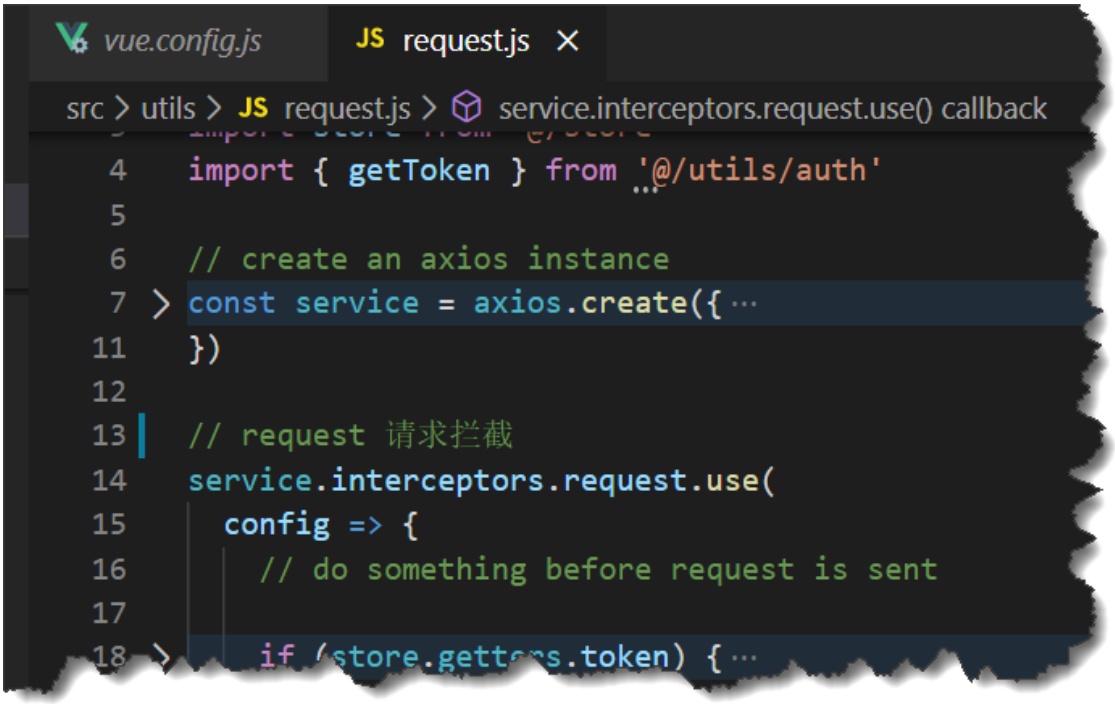
如果我们要跨域请求数据，在配置文件里设置代理，vue-cli3 项目，需要在 vue.config.js 里面写配置。

request.js vue.config.js

vue.config.js > <unknown> > devServer

```
32    devServer: {
33      port: port,
34      open: true,
35      overlay: {
36        warnings: false,
37        errors: true
38      },
39      proxy: {
40        // 修改地址 api/login => mock/login
41        // detail: https://cli.vuejs.org/config/#devserver-proxy
42        '/api': {
43          target: `http://127.0.0.1:${port}/mock`,
44          changeOrigin: true,
45          pathRewrite: {
46            '^/api': ''
47          }
48        },
49        '/iqidi': {
50          target: `http://www.iqidi.com`,
51          changeOrigin: true,
52          pathRewrite: {
53            '^/iqidi': ''
54          }
55        }
56      },
57      after: require('./mock/mock-server.js')
58    },
```

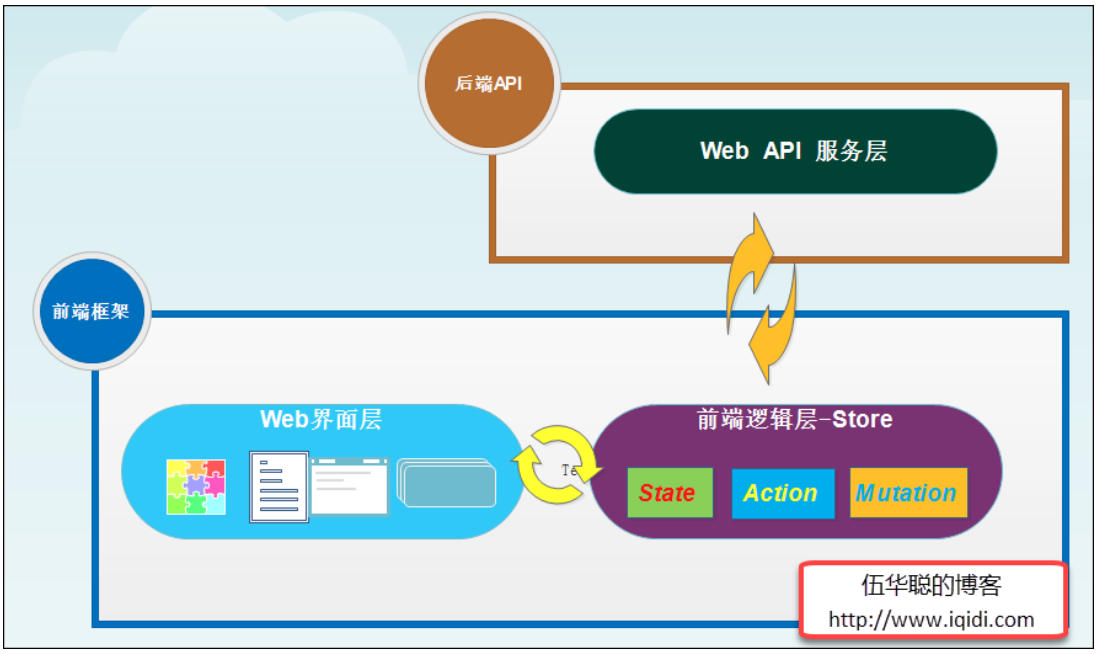
可以分别设置请求拦截和响应拦截，在发出请求和响应到达 then 之前进行判断处理，一般的处理方式就是封装一个类如 request 类，然后进行对拦截器的统一处理，如在请求前增加一些用户身份信息等。

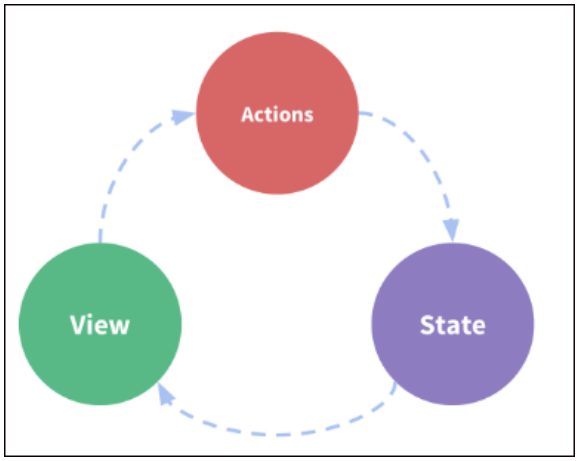

vue.config.js
request.js
src > utils > request.js > service.interceptors.request.use() callback
import { getToken } from '@/utils/auth'
// create an axios instance
const service = axios.create({ ···
})
// request 请求拦截
service.interceptors.request.use(
config => {
// do something before request is sent
if (store.getters.token) {···

```
// create an axios instance
const service = axios.create({
  timeout: 5000 // request timeout
})

// request 请求拦截
service.interceptors.request.use(
  config => {

    if (store.getters.token) {
      config.headers['X-Token'] = getToken()
    }
    return config
  },
  error => {
    // do something with request error
    console.log(error) // for debug
    return Promise.reject(error)
  }
)
```

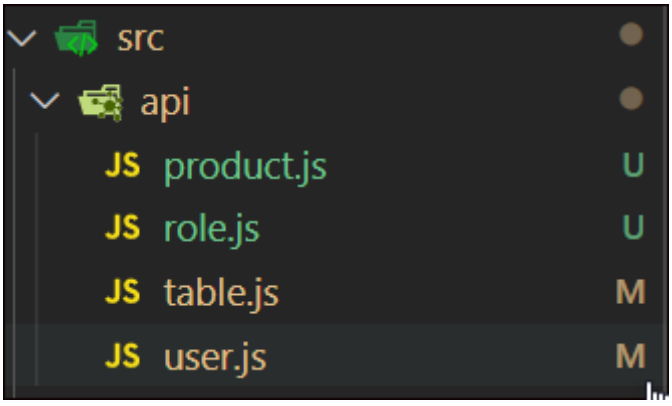
3.1.3. Vuex 中的 API、Store 和 View 的使用

我们再次回到 Vuex 中的 API、Store 和 View 的使用介绍上。

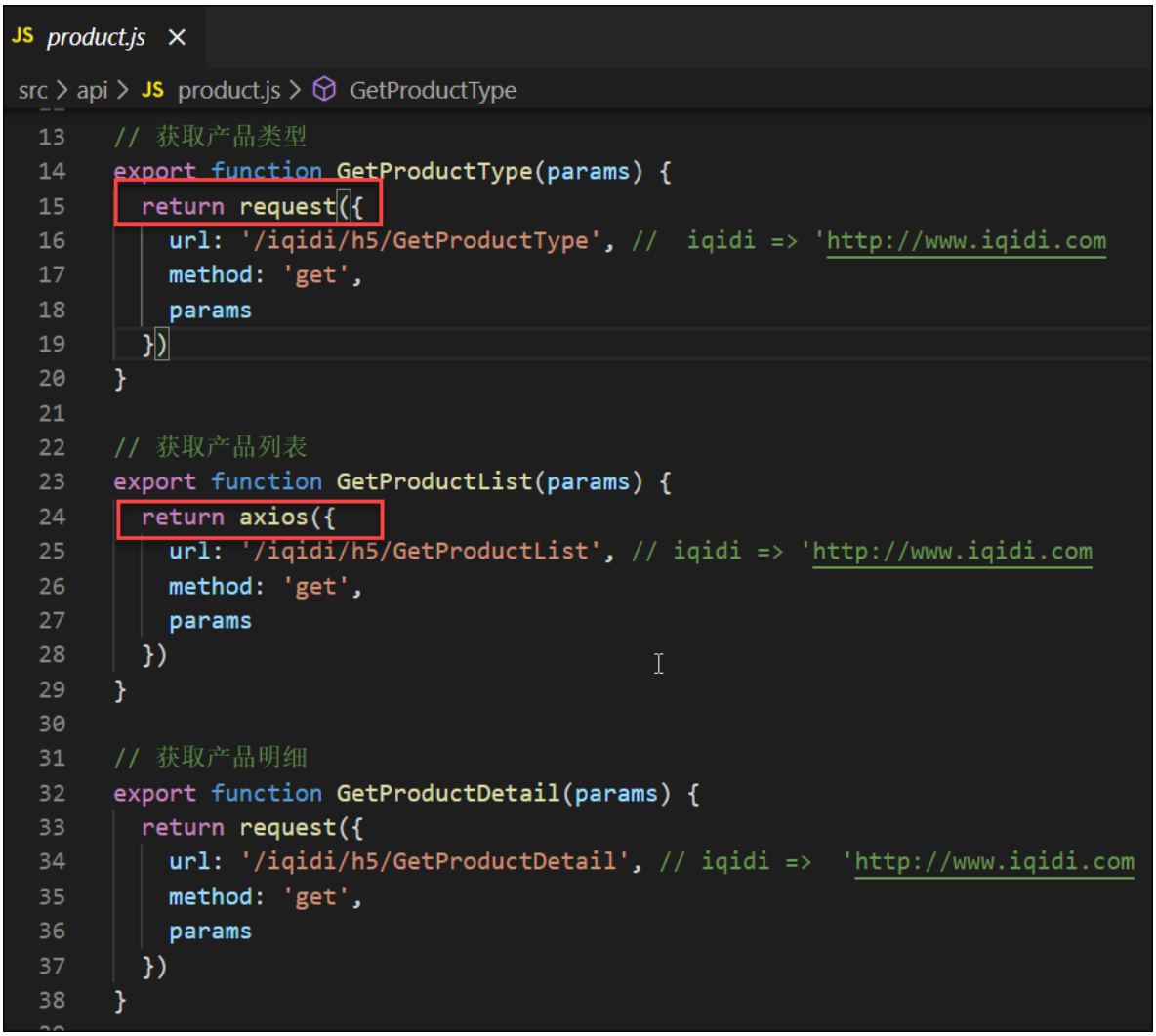




我们来看看 API 的封装请求调用类的封装，如下所示，我们创建了一些操作数据的 API 类文件，每个 API 名称对应一个业务的集中处理，包括特定业务的列表请求、单个请求、增加、删除、修改等等都可以封装在一个 API 类里面。



我们来看看 Product.js 的类文件定义如下所示。



这里我用了 Request 和 Axios 的操作对比，两者很接近，因为 request 是对 Axios 的简单封装，主要就是拦截注入一些登录信息和一些响应的错误处理而已。

```
import request from '@/utils/request'
import axios from 'axios'
```

这里的 Url 里面，通过代理配置的处理，会把对应的 iqidi 替换为对应外部域名的处理，从而实现对跨域处理请求数据的获取了，我们这里只需要知道，url 最终会转换为类似 http://www.iqidi.com/h5/GetProductList 这样实际的地址进行请求的即可，返回是一个 JSON 数据集合。

由于 Vue 视图里面的 JS 处理部分，可以直接引入 API 进行请求数据，如下所示。

```
import { GetProductList } from '@/api/product'
```

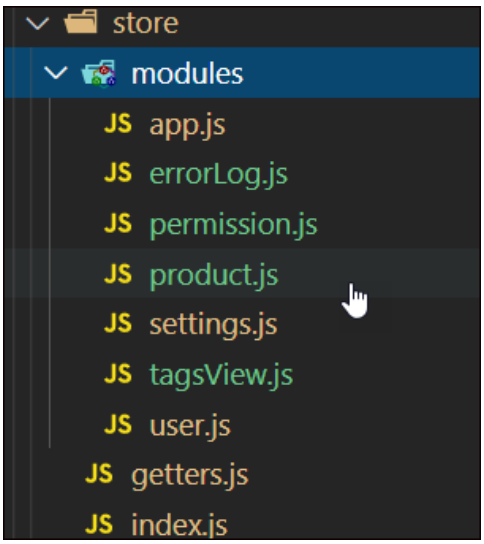
然后我们就可以在 method 方法里面定义一个获取 API 数据的方法了。

```
methods: {
  getlist(type) {
    GetProductList({ type: type }).then(response => {
      const { data } = response
      this.productlist = data.list
      this.listLoading = false
    })
  }
```

这种调用是最直接的 API 调用，没有引入 Store 模块中封装的 Action 或者 Mutation 进行异步或者同步的处理。一般情况下直接使用这种方式比较简洁，因为大多数页面处理或者组件处理，不需要对数据进行全局状态的存储处理，也就是不需要进行全局 Store 对象的处理了。

如果我们需要在全局存储对应的信息，那么就需要引入 Store 模块中对 API 调用的封装了，包括 Action 或者 Mutation 的处理。

我们先来定义 Store 存储类，如下界面所示。



如果我们需要对产品列表等数据进行全局状态的存储，那么我们可以考虑创建一个对应 Store 目录下的模块，如 product.js，来管理 Action、Mutation 和 State 等信息。

src > store > modules > JS product.js > ...

```
1  import { GetProductList, GetProductDetail } from '@/api/product'
2
3  const state = {
4    productlist: [],
5    productdetail: null
6  }
7  const mutations = {
8    SET_PRODUCT_LIST: (state, list) => {
9      state.productlist = list
10   },
11   SET_PRODUCT_DETAIL: (state, detail) => {
12     state.productdetail = detail
13   }
14 }
15
16 const actions = {
17   // 产品列表
18   getProductList({ commit }, { type }) {
19     console.log(type);
20     return new Promise((resolve, reject) => {
21       GetProductList({ type: type }).then(response => {
22         const { data } = response
23         commit('SET_PRODUCT_LIST', data)
24         resolve(data)
25       }).catch(error => {
26         reject(error)
27       })
28     })
29   },
```

```
30
31    // 获取产品明细
32    getProductDetail({ commit }, { id }) {
33      return new Promise((resolve, reject) => {
34        GetProductDetail({ id: id }).then(response => {
35          const { data } = response
36          commit('SET_PRODUCT_DETAIL', data)
37          resolve(data)
38        }).catch(error => {
39          reject(error)
40        })
41      })
42    }
43  }
44
45  export default {
46    namespaced: true,
47    state,
48    mutations,
49    actions
50  }
```

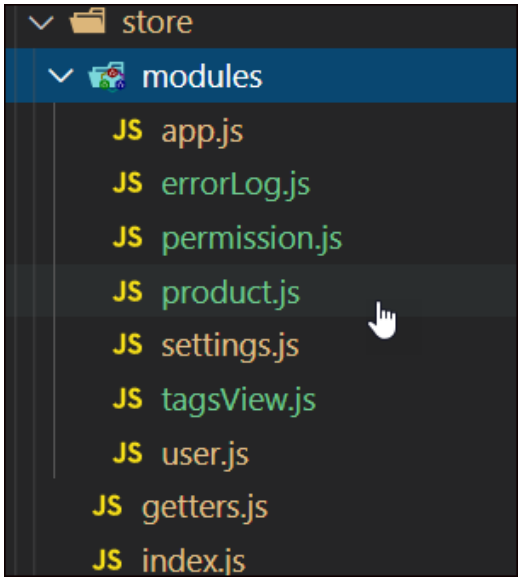
我们下来看看，如果引入了 Store 模块的业务类，那么在界面视图中调用代码则修改为调用对应的 Action 或者 Mutation 了。

```
methods: {
  getlist(type) {
    // GetProductList({ type: type }).then(response => {
    //   const { data } = response
    //   this.productlist = data.list
    //   this.listLoading = false
    // })

    this.$store.dispatch('product/getProductList', { type: type }).then(data => {
      this.productlist = data.list
      // this.loading = false
    }).catch((e) => {
      // this.loading = false
    })
  }
```

我们这里强调一下，一般情况下在视图模块中使用 API 的类调用即可，不需要累赘的每个业务模块，都创建一个 Store 的模块类进行相应的管理，只有在这些状态数据需要在多

个页面或者组件中需要共享的时候，才考虑引入 Store 模块类进行细化管理。

我们刚才说到，如果需要创建对应业务模块的 Store 状态管理模块，那么需要创建对应的模块类，如前面说到的 product.js 类文件。



其中 Modules 目录里面的按照业务区分边界的 Vuex 的 Store 管理类了，每个对应业务创建一个单独的文件进行管理（如果需要用到的话）。

在 index.js 里面我们通过模块动态加载的方式，把这些类按照不同的命名空间进行加载进来，统一使用。

src > store > index.js > store

```
import Vue from 'vue'
import Vuex from 'vuex'
import getters from './getters'

Vue.use(Vuex)

// https://webpack.js.org/guides/dependency-management/#requirecontext
const modulesFiles = require.context('./modules', true, /\.js$/)

// you do not need `import app from './modules/app'`
// it will auto require all vuex module from modules file
const modules = modulesFiles.keys().reduce((modules, modulePath) => {
  // set './app.js' => 'app'
  const moduleName = modulePath.replace(/^\.\/(.*)\.\w+$/, '$1')
  const value = modulesFiles(modulePath)
  modules[moduleName] = value.default
  return modules
}, {})

const store = new Vuex.Store({
  modules,
  getters
})

export default store

```

3.2. 动态菜单和路由的关联处理

在我开发的很多系统里面，包括 Winform 混合框架、Bootstrap 开发框架等系列产品中，我都倾向于动态配置菜单，并管理对应角色的菜单权限和页面权限，实现系统对用户权限的控制，菜单一般包括有名称、图标、顺序、URL 连接等相关信息。

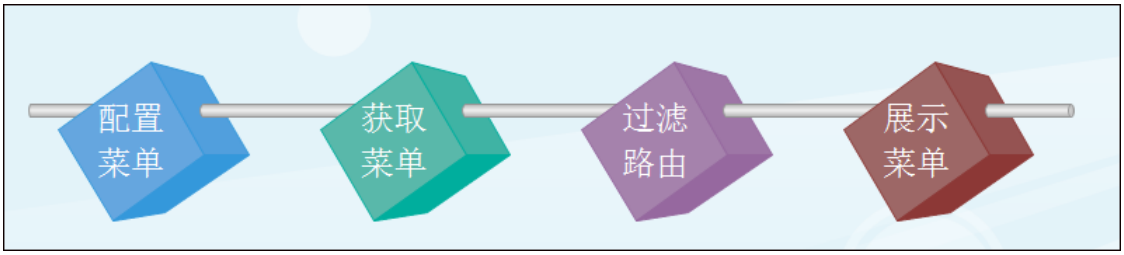
对于 VUE+Element 前端应用来说，应该原理上差不多，这里介绍结合服务端的动态菜单配置和本地路由的关联处理，实现动态菜单的维护和展示的处理。

3.2.1. 菜单和路由的处理过程

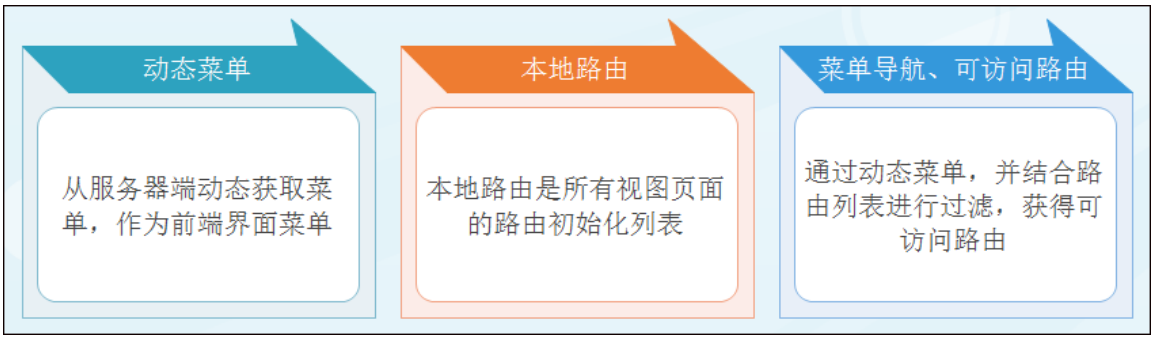
由于 Vue 前端还需要引入路由这个概念，路由是我们前端可以访问到的对应路径集合，路由定义了常规菜单说没有的很多复杂信息，但是往往这些是我们不能随意修改的，因此我们做法是以本地配置好的路由列表为基准，而菜单我们采用在后盾配置方式，前端通过接口动态获取菜单列表，通过菜单的名称和路由名称的对应关系，我们以菜单集合为对照，然后

过滤本地所有静态路由的列表，然后获得用户可以访问的路由列表，设置动态路由给前端，从而实现了界面根据用户角色/权限的不同，而变化用户的菜单界面和可访问路由集合。

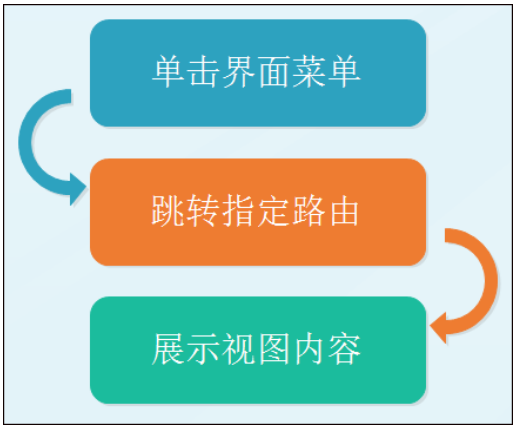
菜单路由处理的大概的操作过程如下所示



前端界面的动态菜单、本地路由、菜单导航和可访问路由的几个概念如下所示。



在前端界面处理中，我们通过 Element 界面组件的方式展示动态菜单信息，并结合菜单和路由的关系，实现菜单跳转到对应视图的处理过程。

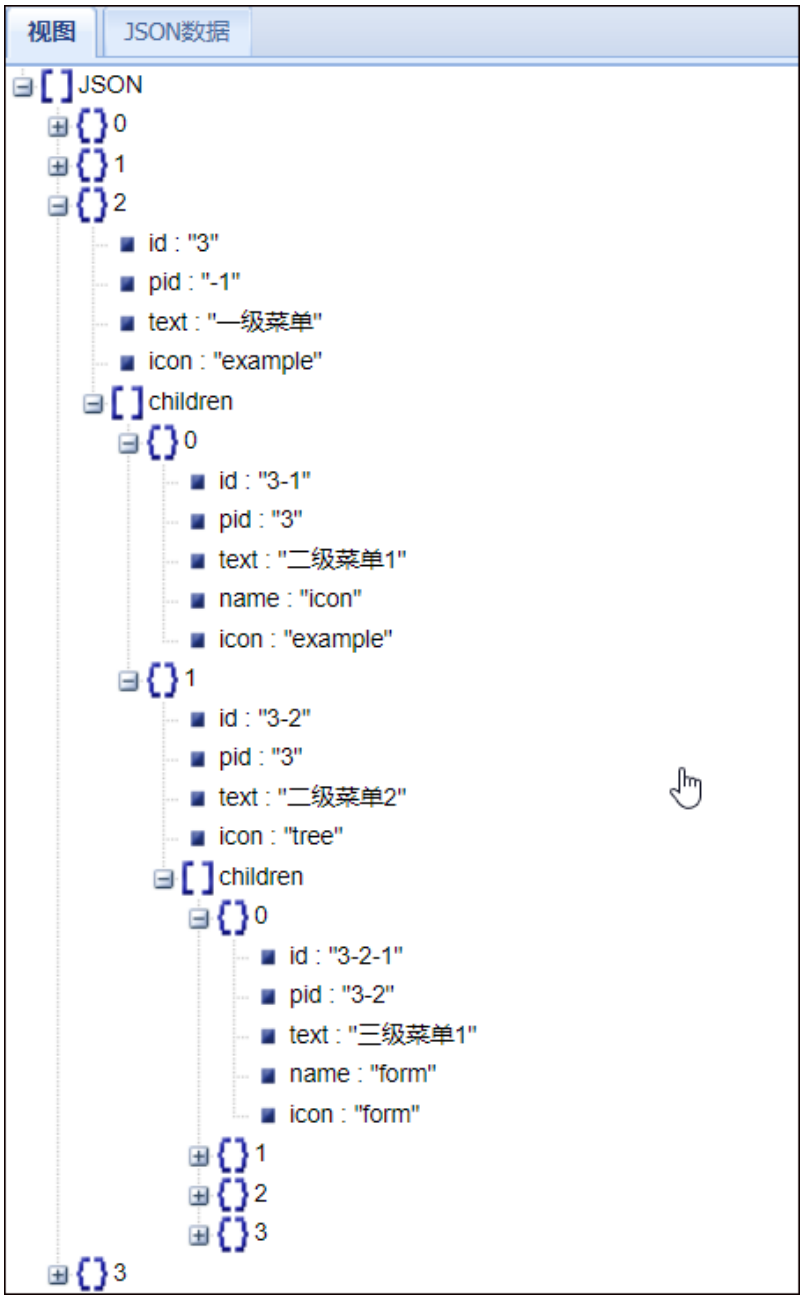


3.2.2. 菜单和路由列表

根据前面的介绍，我们定义了一些从服务端返回的动态菜单信息，这些菜单信息是一个 JSON 对象集合，如下界面所示。

```
[
  {
    id: '1',
    pid: '-1',
    text: '首页',
    icon: 'dashboard',
    name: 'dashboard'
  },
  {
    id: '2',
    pid: '-1',
    text: '产品列表',
    icon: 'table',
    name: 'product'
  },
  {
    id: '3',
    pid: '-1',
    text: '一级菜单',
    icon: 'example',
    children: [
      {
        id: '3-1',
        pid: '3',
        text: '二级菜单1',
        name: 'icon',
        icon: 'example'
```

菜单的 JSON 是根据角色进行动态获取的，不同的角色对应不同的菜单集合，并且菜单是一个多层次的树列表，可以定义无穷多级的展示，JSON 格式化视图如下所示。



而 Vue 前端需要初始化定义前端页面的所有路由，这些包括路由页面的 Layout 等信息。我们可以在一个 JS 文件里面定义好对应前端所有的路由信息，如下所示

```
// 定义本系统的所有路由，具体路由呈现经过菜单数据过滤
export const asyncRoutes = {
  'dashboard': {
    path: '/dashboard',
    component: Layout,
    children: [{
      path: 'dashboard',
      name: 'dashboard',
      component: () => import('@/views/dashboard/index')
    }]
  },
  'product': {
    path: '/product',
    component: Layout,
    children: [{
      path: '/product',
      name: 'product',
      component: () => import('@/views/Product/index')
    }]
  },

  ............................. //省略部分

  'icon': {
    path: '/icon',
    component: Layout,
    children: [{
      path: '/icon',
      name: 'icon',
      component: () => import('@/views/icons/index')
    }]
  },

  'external-link': {
```

这里的路由不需要嵌套，因为菜单展示才需要定义嵌套关系。

另外，由于系统在未登录请求后端动态菜单前，我们系统也需要正常运行起来，那么就需要预设一些基础的路由信息，如登录界面、重定向页面、首页链接等这些路由信息，因此我们可以分开两个路由对象，用来分开管理这些信息。

```
/**
 * 定义系统默认的一些路由，否则无法处理常规页面链接，必须包括首页
 */
const constantRoutes = [
  {…
  },
  {
    path: '/login',
    component: () => import('@/views/login/index'),
    hidden: true
  },
  {…
  }

  // 404 page 必须放在路由列表的最后
  // { path: '*', redirect: '/404', hidden: true }
]
// 定义本系统的所有路由，具体路由呈现经过菜单数据过滤
export const asyncRoutes = {…
}
```

对路由的管理，一个需要默认创建路由的处理、重置路由的处理，以及动态设置新的路由处理，我们封装几个函数来处理这些操作。

```
const createRouter = () => new Router({
  // mode: 'history', // require service support
  scrollBehavior: () => ({ y: 0 }),
  routes: constantRoutes
})

const router = createRouter()

// 重置路由
export function resetRouter() {
  const newRouter = createRouter()
  router.matcher = newRouter.matcher // reset router
}
```

用户在经过登录界面处理后，就会通过对应的 Action 获取动态路由信息（注意，这里是先获取动态菜单，然后过滤本地路由，即为动态路由信息），获得动态路由后，就设置前

端所能访问的路由集合即可，如下代码所示。

```
// 根据角色获取能够访问的路由集合
const accessRoutes = await store.dispatch('permission/generateRoutes', roles)
console.log(accessRoutes);

// 动态添加可访问路由
router.addRoutes(accessRoutes)
router.options.routes = accessRoutes
```

有了这些新的路由允许，前端系统的菜单才能够正常运转起来，否则即使界面展示了菜单，也不能访问特定的视图页面而跳到了 404 页面，因为路由没有。

3.2.3. 登录的过程处理

前面大概介绍了路由的处理过程，其实我们路由信息，应该需要从登录界面开始讲起。

以登录界面为例，在用户登录处理后，需要先验证用户的账号密码，成功后继续请求该用户对应的动态菜单集合，并通过路由切换到对应的页面或者首页。

```
handleLogin() {
  this.$refs.loginForm.validate(valid => {
    if (valid) {
      this.loading = true
      this.$store.dispatch('user/login', this.loginForm).then(() => {
        this.$router.push({ path: this.redirect || '/' })
        this.loading = false
      }).catch(() => {
        this.loading = false
      })
    } else {
      console.log('error submit!!')
      return false
    }
  })
}
```

在 Store/Modules/user.js 模块里面，定义了对应的登陆处理 Action，如下所示

```
// user login
login({ commit }, userInfo) {
  const { username, password } = userInfo
  return new Promise((resolve, reject) => {
    login({ username: username.trim(), password: password }).then(response => {
      const { data } = response
      commit('SET_TOKEN', data.token)
      setToken(data.token)
      resolve()
    }).catch(error => {
      reject(error)
    })
  })
},
```

我们这里忽略用户登录的检验和处理 token 的过程，主要关注动态菜单请求并设置路由的过程。在我们需要拦截路由到达前的处理中，我们定义对应的路由信息请求逻辑，如下所示。

```
router.beforeEach(async(to, from, next) => {
```

```
router.beforeEach(async(to, from, next) => {
  // 进度条显示开始
  NProgress.start()

  // 设置页面标题
  document.title = getPageTitle(to.meta.title)

  // 判断用户是否登录成功
  const hasToken = getToken()

  if (hasToken) {
    if (to.path === '/login') {
      // 如果已经登录，跳到首页
      next({ path: '/' })
      NProgress.done()
    } else {
      // 确保用户通过操作 getInfo，获得所需角色信息
      const hasRoles = store.getters.roles && store.getters.roles.length > 0
      if (hasRoles) {
        next()
      } else {
        try {
          // 获取用户信息
          // 注意：角色roles 为对象集合， 例如： ['admin'] or ,['developer','editor']
          const { roles } = await store.dispatch('user/getInfo')
          // console.log(roles);

          // 根据角色获取能够访问的路由集合
          const accessRoutes = await store.dispatch('permission/generateRoutes', roles)
          console.log(accessRoutes);
```

动态获取路由的处理逻辑

在处理菜单路由的对应模块里面，我们定义了一个状态用来承载这些重要信息，如下定义 State 所示。

```
const state = {
  menuItems: [],
  routes: [],
  addRoutes: [],
  asyncRoutes: asyncRoutes
}
```

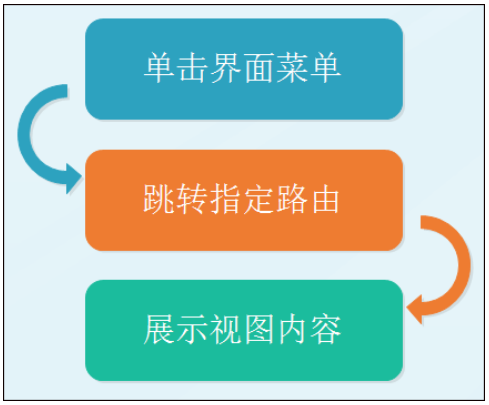
```
// 定义了路由和菜单的Mutation
const mutations = {
  SET_ROUTES: (state, routes) => {
    // var list = convertRoute(routes)
    routes.push({ path: '*', redirect: '/404', hidden: true }) // 此为默认错误路由

    state.addRoutes = routes
    state.routes = [].concat(routes)// constantRoutes.concat(routes)
  },
  SET_MENUS: (state, menus) => {
    state.menuItems = menus
  }
}
```

```
// 定义了生成动态路由的Action处理
const actions = {
  generateRoutes({ commit }, roles) {
    return new Promise(resolve => {
      getMenus().then(res => {
        const menus = res.data || [] // 统一通过接口获取菜单信息
        const routes = []

        menus.forEach(item => {
          filterRoutes(routes, item)
        })
        console.log(routes)// 打印路由

        commit('SET_ROUTES', routes)
        commit('SET_MENUS', menus)
        resolve(routes)
      });
    })
  }
}
```

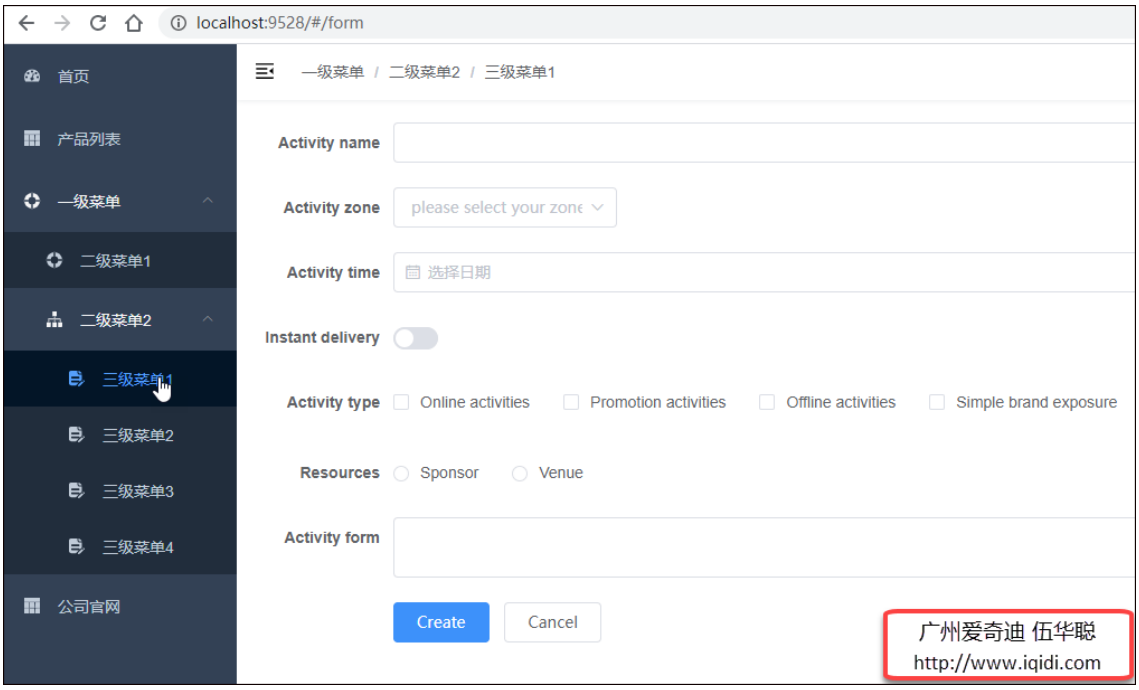
最后返回对应的 JS 定义模块类信息即可。

```
export default {
  namespaced: true,
  state,
  mutations,
  actions
}
```

在前端界面处理中，我们通过 Element 界面组件的方式展示动态菜单信息，并结合菜单和路由的关系，实现菜单跳转到对应视图的处理过程。

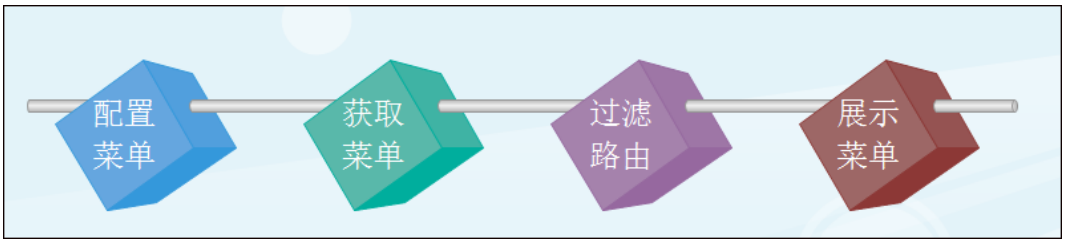


我们来看看界面生成的动态菜单效果。



由于菜单动态展示和动态路由配合，因此既能在前端实现动态菜单的展示，又会根据菜单的集合刷新可访问路由，两者结合就可以顺利打开对应的视图页面了。

再来回顾一下，菜单路由处理的大概的操作过程如下所示



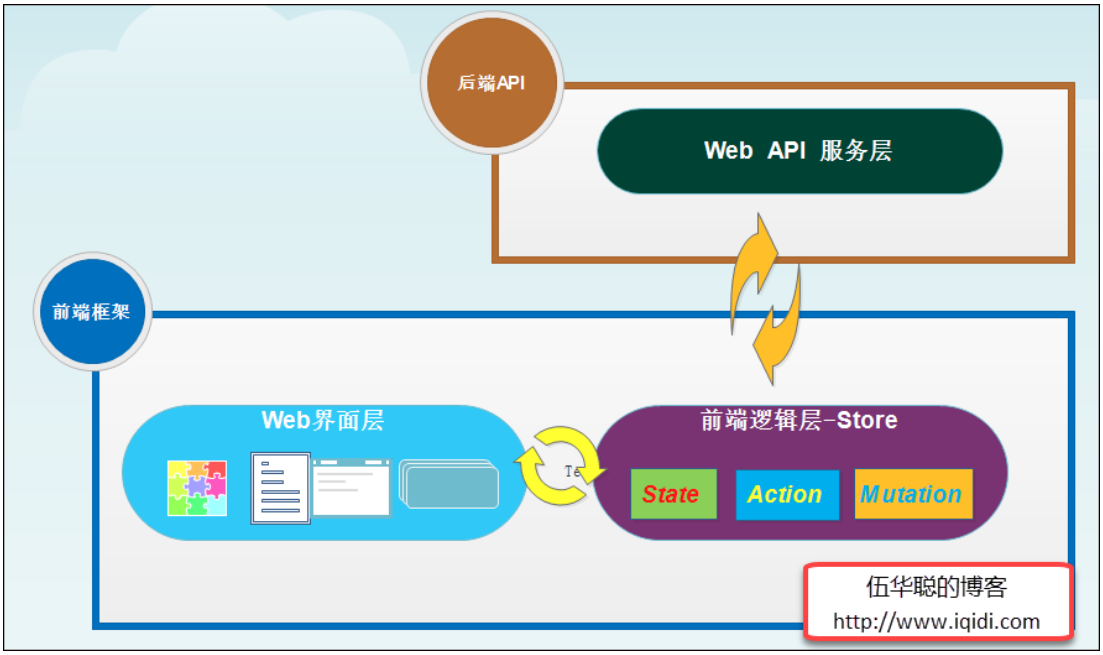
3.3. 获取后端数据及产品信息页面的处理

前面小节实现了动态菜单和动态路由的处理，从而可以根据用户角色对应的菜单实现本地路由的过滤和绑定，菜单顺利弄好了，就需要进一步开发页面功能了，这里介绍一个案例，通过获取后端数据后，进行产品信息页面的处理。

3.3.1. 后端数据的获取处理

前面随笔，我们介绍过了 Vue + Element 的前端框架中，主要通过后端获取数据，并呈现在界面视图或者组件上的。

前后端的边界则非常清晰，前端可以在通过网络获取对应的 JSON 就可以构建前端的应用了。



我们通过 Vue.config.js 的配置信息，可以指定 Proxy 来处理是本地 Mock 数据还是实际的后端数据，如下所示。

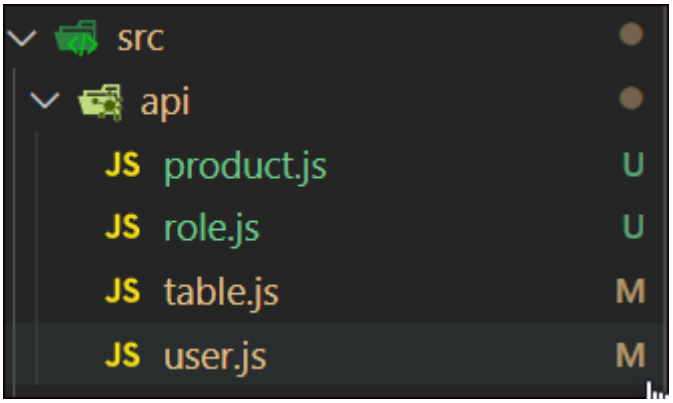
我们要跨域请求数据，在配置文件里设置代理，vue-cli3 项目，需要在 vue.config.js 里面写配置。

request.js　vue.config.js

vue.config.js > <unknown> > devServer

```
32    devServer: {
33      port: port,
34      open: true,
35      overlay: {
36        warnings: false,
37        errors: true
38      },
39      proxy: {
40        // 修改地址 api/login => mock/login
41        // detail: https://cli.vuejs.org/config/#devserver-proxy
42        '/api': {
43          target: `http://127.0.0.1:${port}/mock`,
44          changeOrigin: true,
45          pathRewrite: {
46            '^/api': ''
47          }
48        },
49        '/iqidi': {
50          target: `http://www.iqidi.com`,
51          changeOrigin: true,
52          pathRewrite: {
53            '^/iqidi': ''
54          }
55        }
56      },
57      after: require('./mock/mock-server.js')
58    },
```

我们创建了一些操作数据的 API 类文件，每个 API 名称对应一个业务的集中处理，包括特定业务的列表请求、单个请求、增加、删除、修改等等都可以封装在一个 API 类里面。



然后在对应的业务 API 访问类，如 product.js 里面定义我们的请求逻辑，主要就是利用简单封装 axios 的 request 辅助类来实现数据的请求。

```
JS product.js ×  list.vue  index.vue
src > api > JS product.js > GetProductList
1   import request from '@/utils/request'
2
3   // 获取产品类型
4   export function GetTopBanners(params) {···
10  }
11  // 获取产品类型
12  export function GetProductType(params) {
13    return request({
14      url: '/iqidi/h5/GetProductType', //  iqidi => 'http://www.iqidi.com
15      method: 'get',
16      params
17    })
18  }
19  // 获取产品列表
20  export function GetProductList(params) {
21    return request({
22      url: '/iqidi/h5/GetProductList', // iqidi => 'http://www.iqidi.com
23      method: 'get',
24      params
25    })
26  }
```

下一步就是在 src/views/product 目录里面定义我们的视图文件，这个也就是页面文件，其中包含了常规 VUE 的几个部分，包括

```
<template>

</template>

<script>
export default {

}
</script>

<style>

</style>
```

其中在<template>里面的是界面元素部分，可以添加我们相关的界面组件等内容，如 element 的界面组件在里面。

其中<script>是 vue 脚本交互的部分了，这里面封装我们很多处理逻辑和对应的 modal 对象信息，在调用 API 类进行访问数据前，我们往往需要引入对应的 API 类文件，如下所示。

```
import { GetTopBanners, GetProductList } from '@/api/product'
```

其中<style>则定义相关的样式。

在开始介绍 Vue 的 Script 部分，我们认为你已经对 VUE 的 script 相关内容，以及它的生命周期有所了解了，script 部分的内容包括有。

```
<script>
export default {
  data() {
    return {};
  },
  methods: {
    // 组件的方法
  },
  watch: {
    // watch擅长处理的场景：一个数据影响多个数据
  },
  computed: {
    // computed擅长处理的场景：一个数据受多个数据影响
  },
  beforeCreate: function() {
    // 在实例初始化之后，数据观测(data observer) 和 event/watcher 事件配置之前被调用。
  },
  created: function() {
    // 实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始
  },
  beforeMount: function() {
    // 在挂载开始之前被调用：相关的 render 函数首次被调用。
  },
  mounted: function() {
    // 编译好的HTML挂载到页面完成后执行的事件钩子
    // el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。
    // 此钩子函数中一般会做一些ajax请求获取数据进行数据初始化
    console.log("Home done");
  },
  beforeUpdate: function() {
    // 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。 你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。
  },
  updated: function() {
```

其中我们主要涉及到内容包括：

data，用于定义整个页面的 modal 对象或属性，

method，用于定义各种处理方法

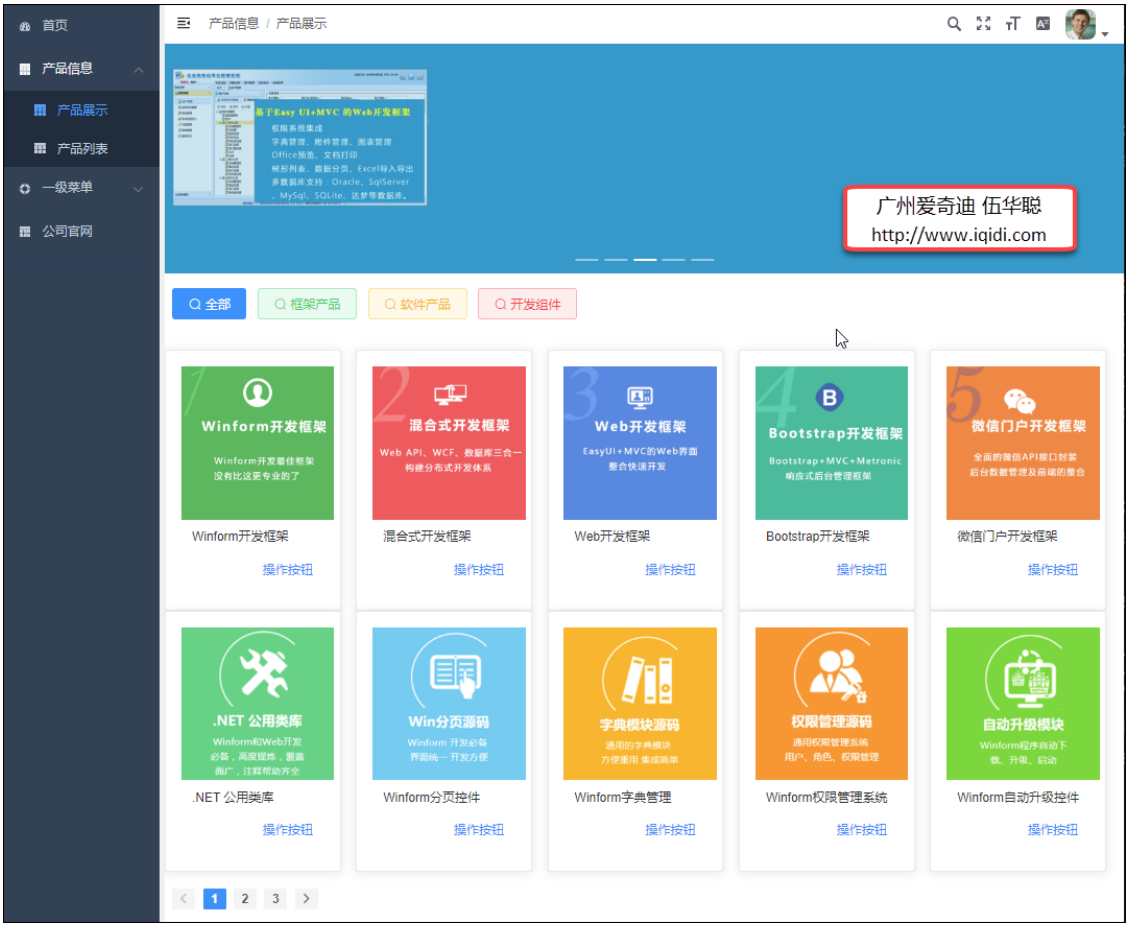
computed，用于定义一些计算的树形

created，用于我们在元素创建，但是没有挂载的时候

mounted，完成页面挂载的时候

3.3.2. 界面展示处理

例如我们要展示一个界面内容，需要展示产品的图片，以及产品信息介绍



那么需要定义相关的数据模板，以及对应的处理方法，在页面加载前实现数据的绑定处理。

```
export default {
  data() {
    return {
      activeName: 'all',
      currentDate: new Date(),
      banners: [],
      productlist: [],
      pageinfo: {
        pageindex: 1,
        pagesize: 10,
        total: 0
      },
      producttype: 'all'
    };
  },
  created() {
    this.loadbanners()
    this.getlist()
  },
```

界面部分，我们利用 Element 的界面组件定义一个走马灯的展示效果，如下所示。

在 Template 模块里面定义好的界面元素如下所示。

```
<template>
  <div>
    <div class="block" style="background:#0099cc;">
      <!-- <span class="demonstration">推荐产品</span> -->
      <el-carousel height="350px;">
        <el-carousel-item v-for="item in banners" :key="item">
          <el-image :src="item" fit="scale-down" style="padding:5px;width:350px;height:auto" />
        </el-carousel-item>
      </el-carousel>
    </div>
```

这里面的 el-carousel 是 Element 组件的走马灯，而 v-for="item in banners" 就是 vue 的处理语法，对 data 模型里面的数据进行循环处理，然后逐一展示多个图片，从而实现了走马灯的效果展示。

对于列表展示，我们采用了一个卡片的展示内容，以及分页处理的方式实现数据的展示。

分类按钮部分，代码如下所示。

```
<el-row :gutter="20" style="padding:20px">
  <el-button type="primary" icon="el-icon-search" :plain="producttype !='all'" @click="handleClick($event, 'all')">全部</el-button>
  <el-button type="success" icon="el-icon-search" :plain="producttype !='1'" @click="handleClick($event, '1')">框架产品</el-button>
  <el-button type="warning" icon="el-icon-search" :plain="producttype !='2'" @click="handleClick($event, '2')">软件产品</el-button>
  <el-button type="danger" icon="el-icon-search" :plain="producttype !='3'" @click="handleClick($event, '3')">开发组件</el-button>
</el-row>
```

主要就是根据 data 属性进行一些样式的控制，以及响应对应的 click 事件。

而每个卡片内容介绍，Demo 代码如下所示。

```
<el-col :span="4"><div class="grid-content bg-purple" />
<el-card class="box-card">
  <img src="https://shadow.elemecdn.com/app/element/hamburger.9cf7b091-55e9-11e9-a976-7f4d0b07eef6.png" class="image">
  <div style="padding: 14px;">
    <span>好吃的汉堡</span>
    <div class="bottom clearfix">
      <time class="time">价格：¥200</time>
      <el-button type="text" class="button" @click="loadbanners()">操作按钮</el-button>
    </div>
  </div>
</el-card>
</el-col>
```

但是我们要根据实际获得的动态数据进行绑定，因此需要一个循环来进行处理，类似上面的 v-for 循环，对产品列表进行展示即可。

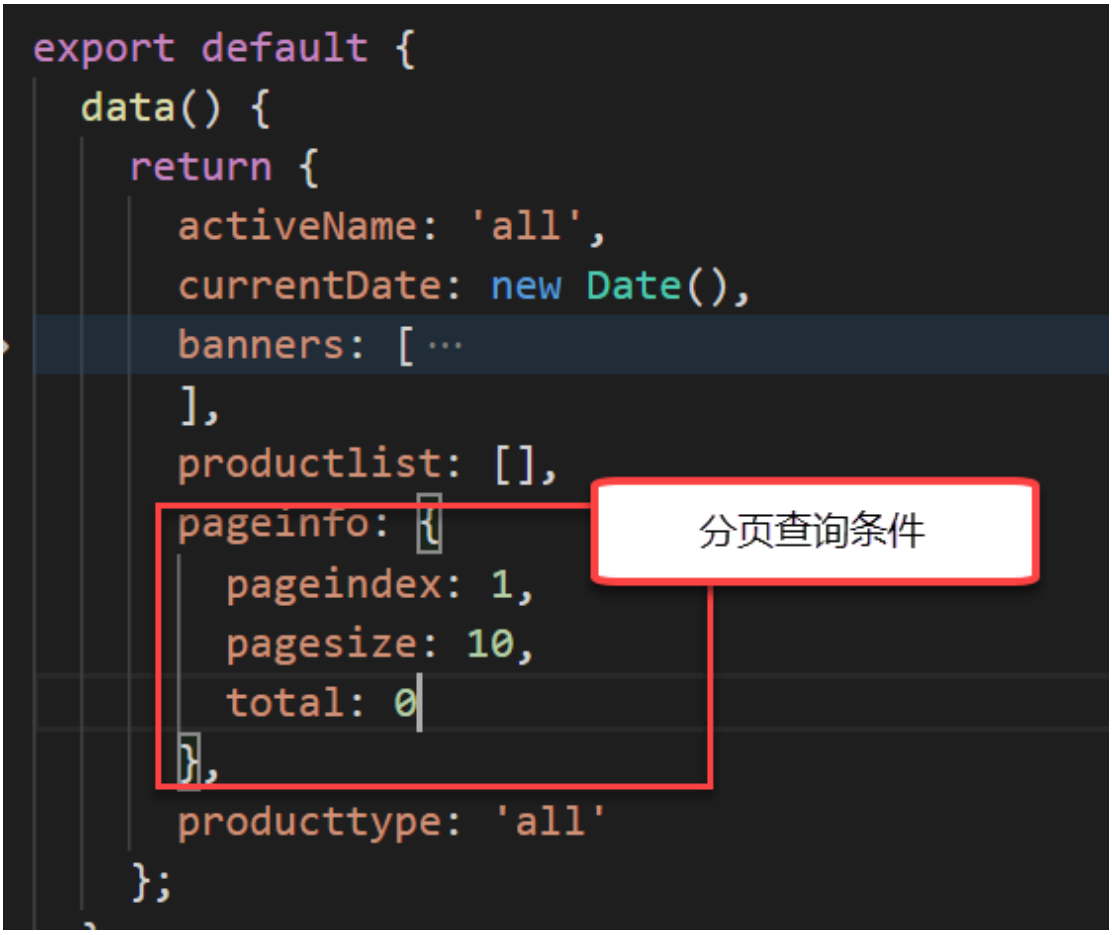
```
<el-col v-for="(item, index) in productlist" :key="index" :span="4" style="min-width:250px">
<div class="grid-content bg-purple">
  <el-card class="box-card">
    <img :src="item.Picture" class="image" style="width:200px;height:200px">
    <div style="padding: 14px;">
      <span>{{ item.ProductName }}</span>
      <div class="bottom clearfix">
        <!-- <time class="time">价格：¥{{ item.Price }}</time> -->
        <el-button type="text" class="button">操作按钮</el-button>
      </div>
    </div>
  </el-card>
</div>
</el-col>
```

为了有效的请求和展示数据，我们还需要利用分页组件来进行数据的分页查询处理，分页组件界面的定义代码如下所示。

```
<el-pagination
  background
  layout="prev, pager, next"
  :page-sizes="[10,20,50]"
  :total="pageinfo.total"
  :current-page="pageinfo.pageindex"
  :page-size="pageinfo.pagesize"
  @size-change="handleSizeChange"
  @current-change="handleCurrentChange"
/>
```

为了实现对数据的分页，我们需要定义当前页码、每页面数据大小、总共数据记录数等

几个变量，用来进行分页查询的需要。



我们定义的 getList 的方法如下所示。

```
getlist() {
  // 构造分页查询条件
  var param = {
    type: this.producttype === 'all' ? '' : this.producttype,
    pageindex: this.pageinfo.pageindex,
    pagesize: this.pageinfo.pagesize
  };

  this.listLoading = true
  // 发起数据查询请求，并设置本地Data的值
  GetProductList(param).then(data => {
    this.productlist = data.list
    this.pageinfo.total = data.total_count
    this.listLoading = false
  })
},
```

另外定义几个方法，对数据进行查询的操作。

```
// 单击某类别的时候，进行查询
handleClick(e, type) {
  // console.log(e, type);

  this.producttype = type
  this.pageinfo.pageindex = 1;

  this.getlist()
},
// 页面数量改变后查询处理
handleSizeChange(val) {
  console.log(`每页 ${val} 条`);
  this.pageinfo.pagesize = val;
  this.getlist()
},
// 页码改变后查询处理
handleCurrentChange(val) {
  console.log(`当前页: ${val}`);
  this.pageinfo.pageindex = val;
  this.getlist()
}
```

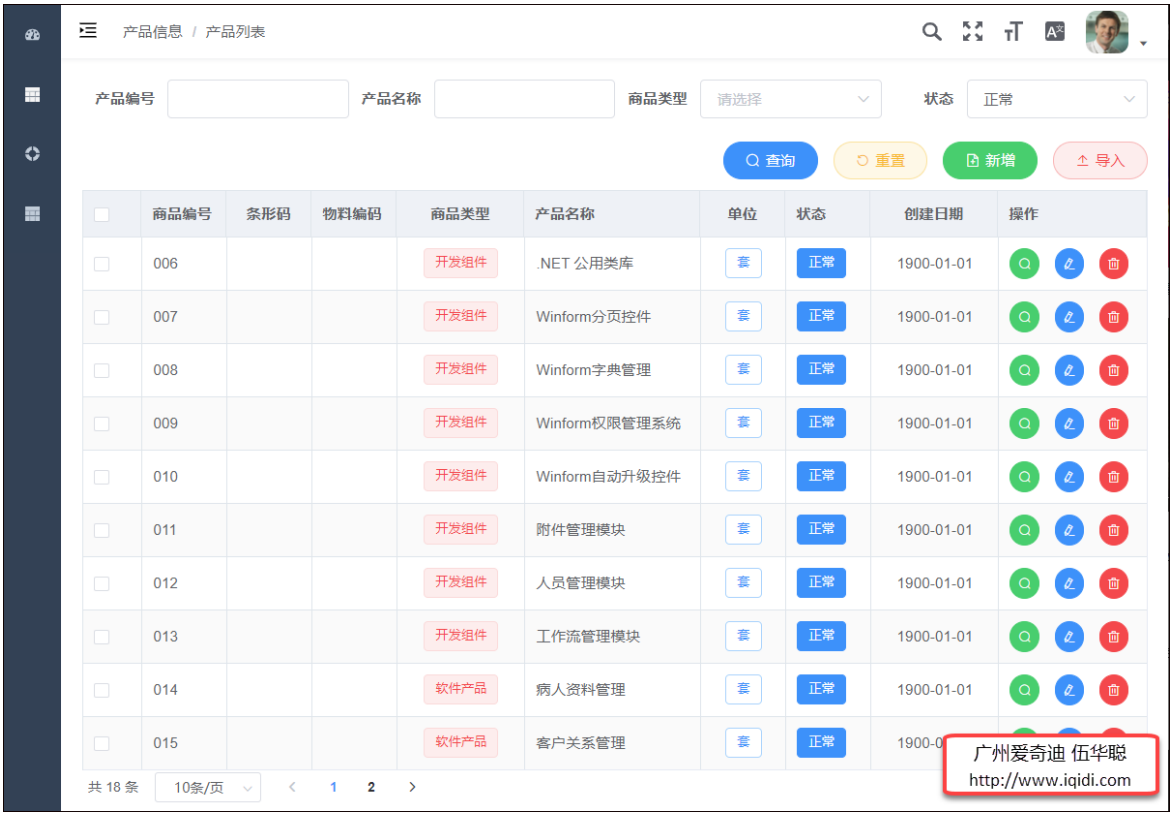
以上就是我们利用 Element 的界面组件，以及 Vue 的方法进行数据的分页查询请求的基础操作，通过这个简单的案例，我们可以了解一些基础的 Element 界面组件的使用，以及对 Data / Method 等内容的了解，并指导如何封装调用跨域的 API 请求，实现后端数据在界面上的展示处理了。

3.4. 表格列表页面的查询，列表展示和字段转义处理

在我们一般开发的系统界面里面，列表页面是一个非常重要的综合展示界面，包括有条件查询、列表展示和分页处理，以及对每项列表内容可能进行的转义处理，这里介绍基于 Vue +Element 基础上实现表格列表页面的查询，列表展示和字段转义处理。

3.4.1. 列表查询界面效果

在介绍任何代码处理逻辑之前，我们先来做一个感官的认识，贴上一个效果图，在逐一介绍其中处理的步骤和注意事项。



常规的列表展示界面，一般分为几个区域，一个是查询区域，一个是列表展示区域，一个是底部的分页组件区域。查询区域主要针对常规条件进行布局，以及增加一些全局或者批量的操作，如导入、导出、添加、批量添加、批量删除等按钮；而其中主体的列表展示区域，是相对比较复杂一点的地方，需要对各项数据进行比较友好的展示，可以结合 Tag,图标，按钮等界面元素来展示，其中列表一般后面会包括一些对单行记录处理的操作，如查看、编辑、删除的操作，如果是批量删除，可以放到顶部的按钮区域。

3.4.2. 查询区域的处理

查询区域一般的界面效果如下所示，除了包含一些常用的查询条件，并增加一些常规的处理按钮，如查询、重置、新增、批量删除、导入、导出等按钮。

对于查询区域来说，它也是一个表单的处理，因此也需要添加一一个对应的对象来承载表单的数据，在 data 里面增加一个 searchForm 的模型对象，以及一个用于分页查询的 pageinfo 对象，如下代码所示。

```
export default {
  data() {
    return {
      listLoading: true,
      pageinfo: {
        pageindex: 1,
        pagesize: 10,
        total: 0
      },
      searchForm: {
        ProductNo: '',
        BarCode: '',
        ProductType: '',
        ProductName: '',
        Status: 0
      },
```

视图模板代码如下所示：

```
<el-form ref="searchForm" :model="searchForm" label-width="80px">
  <el-row>
    <el-col :span="6">
      <el-form-item label="产品编号" prop="ProductNo">
        <el-input v-model="searchForm.ProductNo" />
      </el-form-item>
    </el-col>
    <el-col :span="6">
      <el-form-item label="产品名称" prop="ProductName">
        <el-input v-model="searchForm.ProductName" />
      </el-form-item>
    </el-col>
    <el-col :span="6">
      <el-form-item label="商品类型" prop="ProductType">
        <el-select v-model="searchForm.ProductType" filterable clearable placeholder="请选择">
          <el-option
            v-for="(item, key) in typeList"
            :key="key"
            :label="item.value"
            :value="item.key"
          />
        </el-select>
      </el-form-item>
    </el-col>
```

```
    <el-col :span="6">
      <el-form-item label="状态" prop="Status">
        <el-select v-model="searchForm.Status" filterable clearable placeholder="请选择">
          <el-option
            v-for="item in Status"
            :key="item.Value"
            :label="item.Text"
            :value="item.Value"
          />
        </el-select>
      </el-form-item>
    </el-col>
  </el-row>
</el-form>
<el-row style="float:right;padding-bottom:10px">
  <el-button icon="el-icon-search" type="primary" round @click="search()">查询</el-button>
  <el-button icon="el-icon-refresh-left" type="warning" round plain @click="resetForm('searchForm')">重置</el-button>
  <el-button icon="el-icon-document-add" type="success" round @click="showAdd()">新增</el-button>
  <el-button icon="el-icon-document-remove" type="danger" round @click="BatchDelete()">批量删除</el-button>
  <el-button icon="el-icon-upload2" type="danger" plain="" round @click="showImport()">导入</el-button>
</el-row>
```

其中产品类型的是下拉列表，我们通过在 data 区域获取一个对象，并在此遍历可以展示字典内容，如果我们花点时间，可以把这些下拉列表统一按照一个常规的处理模式，定义一个字典组件的方式实现，简单赋予一个字典类型的 Prop 值，就可以绑定下拉列表了，这个稍后在细讲。

在 Vue 的脚本处理逻辑里面，我们可以在 Created 声明周期里面，通过 API 获取数据，绑定在模型上，界面就会自动进行更新了，处理过程代码如下所示。

```
  created() {
    // 获取产品类型，用于绑定字典等用途
    GetProductType().then(data => {
      if (data) {
        data.forEach(item => {
          this.productTypes.set(item.id, item.name)
          this.typeList.push({ key: item.id, value: item.name })
        })
      }
    });

    // 获取列表信息
    this.getlist()
  },
  methods: {
    getlist() {
      // 构造常规的分页查询条件
      var param = {
        type: this.producttype === 'all' ? '' : this.producttype,
        pageindex: this.pageinfo.pageindex,
        pagesize: this.pageinfo.pagesize
      };

      // 把SearchForm的条件加入到param里面，进行提交查询
      param.type = this.searchForm.ProductType // 转换为对应属性
      Object.assign(param, this.searchForm);

      // 获取产品列表，绑定到模型上，并修改分页数量
      this.listLoading = true
      GetProductList(param).then(data => {
        this.productlist = data.list
        this.pageinfo.total = data.total_count
        this.listLoading = false
      })
    },
```

其中 Object.assign(param, this.searchForm); 语句处理，是把获得的查询条件，覆盖原来对象里面的属性，从而实现查询条件的变量赋值。

获得列表数据，就是介绍如何展示表格列表数据的过程了，表格界面效果如下所示。

商品编号	条形码	物料编码	商品类型	产品名称	单位	状态	创建日期	操作
006			开发组件	.NET 公用类库	套	正常	1900-01-01	
007			开发组件	Winform分页控件	套	正常	1900-01-01	
008			开发组件	Winform字典管理	套	正常	1900-01-01	
009			开发组件	Winform权限管理系统	套	正常	1900-01-01	
010			开发组件	Winform自动升级控件	套	正常	1900-01-01	
011			开发组件	附件管理模块	套	正常	1900-01-01	
012			开发组件	人员管理模块	套	正常	1900-01-01	
013			开发组件	工作流管理模块	套	正常	1900-01-01	
014			软件产品	病人资料管理	套	正常	1900-01-01	
015			软件产品	客户关系管理	套	正常	1900-01-01	

共 18 条 10条/页 1 2

先定义一个表格头部，类似 HTML 里面的<table>的标签，指定样式和一些常规的操作函数，如下代码所示。

```
<el-table
  v-loading="listLoading"
  :data="productlist"
  border
  fit
  stripe
  highlight-current-row
  :header-cell-style="{background:'#eef1f6',color:'#606266'}"
  @selection-change="selectionChange"
  @row-dblclick="rowDbclick"
>
```

具体的属性可以参考下 Element 组件关于表格控件的属性了，在表格列里面，我们主要关注它的 data 绑定即可。

接着定义一列复选框选择的列，用于批量处理的勾选，如批量删除操作。

```
<el-table-column type="selection" width="55"/>
```

接着就是根据返回 JSON 属性，逐一进行内容转换为表格列的展示过程了，如下所示。

```
<el-table-column label="商品编号" width="80">
  <template slot-scope="scope">
    {{ scope.row.ProductNo }}
  </template>
</el-table-column>
```

我们如果需要在显示里面增加处理效果，一般在 template 里面修改展示效果即可，如下是单位的处理，增加一个 tag 标志强调下。

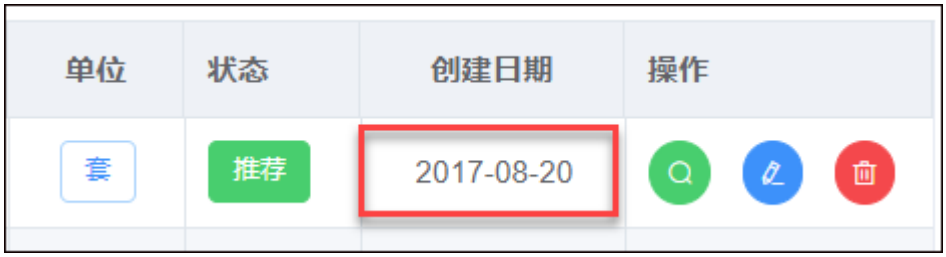
```
<el-table-column align="center" label="单位" width="80">
  <template slot-scope="scope">
    <el-tag type="" effect="plain"> {{ scope.row.Unit }}</el-tag>
  </template>
</el-table-column>
```

而对于一些需要判断处理的效果，我们可以对内容进行判断输出，如下状态所示。

```
<el-table-column label="状态" width="80">
  <template slot-scope="scope">
    <el-tag v-if="scope.row.Status==0" type="" effect="dark">正常</el-tag>
    <el-tag v-else-if="scope.row.Status==1" type="success" effect="dark">推荐</el-tag>
    <el-tag v-else-if="scope.row.Status==2" type="danger" effect="dark">停用</el-tag>
  </template>
</el-table-column>
```

另外，对于一些常见的日期处理，我们可以使用 Formatter，Filter 等手段进行内容的转义处理，可以去掉后面的时间部分。

```
<el-table-column align="center" label="创建日期" width="120" prop="CreateTime" :formatter="dateFormat" />
```



dataFormat 就是一个转义函数，函数代码如下所示。

```
dateFormat(row, column, cellValue) {
  return cellValue ? fecha.format(new Date(cellValue), 'yyyy-MM-dd') : ''
},
```

使用的时候，需要在顶部引入一个类库即可

```
import * as fecha from 'element-ui/lib/utils/date'
```

对于类似需要进行字典转义的操作，我们可以使用 Formatter 的方式转义，如增加一个函数来解析对应的值为中文信息

效果可以使用 Formatter 来转义

```
productTypeFormat(row, column, cellValue) {
  var display = this.productTypes.get(cellValue)
  return display || ''
},
```

也可以使用 Filter 模式来进行处理。这里介绍使用 Filter 的操作处理，首先在界面 HTML 代码里面增加对应的操作，如下代码所示。

```
<el-table-column align="center" label="商品类型" width="120" prop="ProductType">
  <template slot-scope="scope">
    <el-tag type="danger">  {{ scope.row.ProductType | productTypeFilter }}</el-tag>
  </template>
</el-table-column>
```

Filter 其实就是一个 | 过滤符号，以及接着一个过滤函数处理即可。

```
export default {
  filters: {
    productTypeFilter: function(value) {
      if (!value) return ''

      var display = that.productTypes.get(value)
      return display || ''
    }
  },
```

值得注意的是，Filter 本身不能引用 data 里面的属性列表进行转义的需要，如果需要，那么需要在 beforeCreate 的钩子函数里面记录 this 的引用，如下代码所示。

```
let that;
export default {
  filters: {
    productTypeFilter: function(value) {
      if (!value) return ''

      var display = that.productTypes.get(value)
      return display || ''
    }
  },
  data() {
    return {···
    }
  },
  beforeCreate: function() {
    that = this;//用于在使用Filter的时候，引用到了Data数据
  },
```

对于操作按钮，我们需要增加一行来显示几个按钮即可，如果需要权限控制，可以再根据权限集合判断一下可操作权限即可。

```
<el-table-column label="操作" width="140">
  <template scope="scope">
    <el-row>
      <el-tooltip effect="light" content="查看" placement="top-start">
        <el-button icon="el-icon-search" type="success" circle size="mini" @click="showView(scope.row.ID)" />
      </el-tooltip>
      <el-tooltip effect="light" content="编辑" placement="top-start">
        <el-button icon="el-icon-edit" type="primary" circle size="mini" @click="showEdit(scope.row.ID)" />
      </el-tooltip>
      <el-tooltip effect="light" content="删除" placement="top-start">
        <el-button icon="el-icon-delete" type="danger" circle size="mini" @click="showDelete(scope.row.ID)" />
      </el-tooltip>
    </el-row>
  </template>
</el-table-column>
```

这里 showView/showEdit/ShowDelete 主要就是做一些弹出层前的处理操作，我们在 data 项里面定义了几个变量，用来确定是那个操作显示的需要。

```
isAdd: false,
isEdit: false,
isView: false,
isImport: false,
```

例如对应编辑操作，我们是需要通过 API 处理类，获取后端数据，并赋值给编辑框的表单对象上，进行展示即可。

```
showEdit(id) {
  // 通过ID参数，使用API类获取数据后，赋值给对象展示
  var param = { id: id }
  GetProductDetail(param).then(data => {
    Object.assign(this.editForm, data);
  })
  this.isEdit = true
},
```

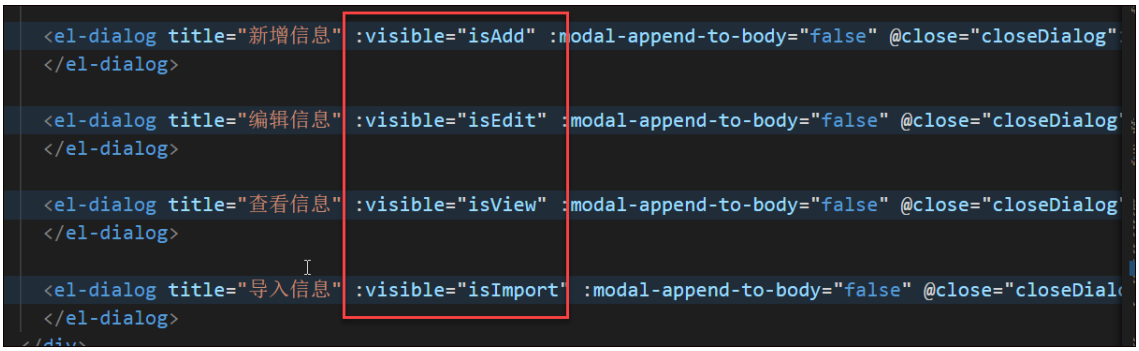
对于查看处理，我们除了在每行按钮上可以单击进行查看指定行记录外，我们双击指定的行，也应该弹出对应的查看记录界面。

```
rowDbclick(row, column) {
  var id = row.ID
  this.showView(id);
},
```

这个就是表格定义里面的一些处理事件。

```
<el-table
  v-loading="listLoading"
  :data="productlist"
  border
  fit
  stripe
  highlight-current-row
  :header-cell-style="{background:'#eef1f6',color:'#606266'}"
  @selection-change="selectionChange"
 @row-dblclick="rowDbclick"
>
```

每个对话框的:visible 的属性值，则是确定哪个模态对话框的显示和隐藏。



而对于删除操作，我们只需要确认一下，然后提交远端处理，返回正常结果，就提示用户删除成功即可。如下逻辑代码所示。

```
showDelete(id) {
  this.$confirm('您确认删除选定的记录吗?', '操作提示',
    {
      type: 'warning' // success, error, info, warning
      // confirmButtonText: '确定',
      // cancelButtonText: '取消'
    }
  ).then(() => {
    // 删除操作处理代码

    this.$message({
      type: 'success',
      message: '删除成功!'
    });
  }).catch(() => {
    this.$message({
      type: 'info',
      message: '已取消删除'
    });
  });
},
```

以上就是常规表格列表页面查询、列表展示、字段转义的一些常规操作，以及对新增、编辑、查看、删除操作的一些常规处理，通过对这些模型的操作，减少了我们以往重新获取对应 DOM 的繁琐操作，是的数据的操作处理，变得方便了很多。

3.5. 常规 Element 界面组件的使用

在我们开发 BS 页面的时候，往往需要了解常规界面组件的使用，小到最普通的单文本输入框、多文本框、下拉列表，以及按钮、图片展示、弹出对话框、表单处理、条码二维码等等，这里介绍基于普通表格业务的展示录入的场景介绍这些常规 Element 组件的使用，使得我们对如何利用 Element 组件有一个大概的认识。

3.5.1. 列表界面和其他模块展示处理

列表界面整合了增删改查等常规的业务操作处理。

常规的列表展示界面，一般分为几个区域，一个是查询区域，一个是列表展示区域，一个是底部的分页组件区域。查询区域主要针对常规条件进行布局，以及增加一些全局或者批量的操作，如导入、导出、添加、批量添加、批量删除等按钮；而其中主体的列表展示区域，是相对比较复杂一点的地方，需要对各项数据进行比较友好的展示，可以结合 Tag,图标，按钮等界面元素来展示，其中列表一般后面会包括一些对单行记录处理的操作，如查看、编辑、删除的操作，如果是批量删除，可以放到顶部的按钮区域。

对于常规按钮、表格、分页组件，前面已经做了相关的介绍，这里就不再赘述。

在介绍具体界面组件的时候，我们先来了解下，整体的界面布局，我们把常规的列表界面，新增、编辑、查看、导入等界面放在一起，除了列表页面，其他内容以弹出层对话框的方式进行处理，如下界面示意所示。

每个对话框的:visible 的属性值，则是确定哪个模态对话框的显示和隐藏。

```
<el-dialog title="新增信息" :visible="isAdd" :modal-append-to-body="false" @close="closeDialog"
</el-dialog>

<el-dialog title="编辑信息" :visible="isEdit" :modal-append-to-body="false" @close="closeDialog
</el-dialog>

<el-dialog title="查看信息" :visible="isView" :modal-append-to-body="false" @close="closeDialog
</el-dialog>

<el-dialog title="导入信息" :visible="isImport" :modal-append-to-body="false" @close="closeDial
</el-dialog>
```

在 Vue 的 JS 模块里面，我们除了定义对应的对话框显示的变量外，对每个对话框，我们定义一个表单信息用来进行数据的双向绑定处理。

```
isAdd: false,
isEdit: false,
isView: false,
isImport: false,
searchForm: {
  ProductNo: '',
  BarCode: '',
  ProductType: '',
  ProductName: '',
  Status: 0
},
addForm: {…
},
editForm: {…
},
viewForm: {…
},
```

常规的新增、编辑、查看、导入等内容的定义，作为一个对话框组件定义，常规的对话框组件的使用代码如下所示。

```
<el-dialog
  title="提示"
  :visible.sync="dialogVisible"
  width="30%"
  :before-close="handleClose">
  <span>这是一段信息</span>
  <span slot="footer" class="dialog-footer">
    <el-button @click="dialogVisible = false">取 消</el-button>
    <el-button type="primary" @click="dialogVisible = false">确 定</el-button>
  </span>
</el-dialog>
```

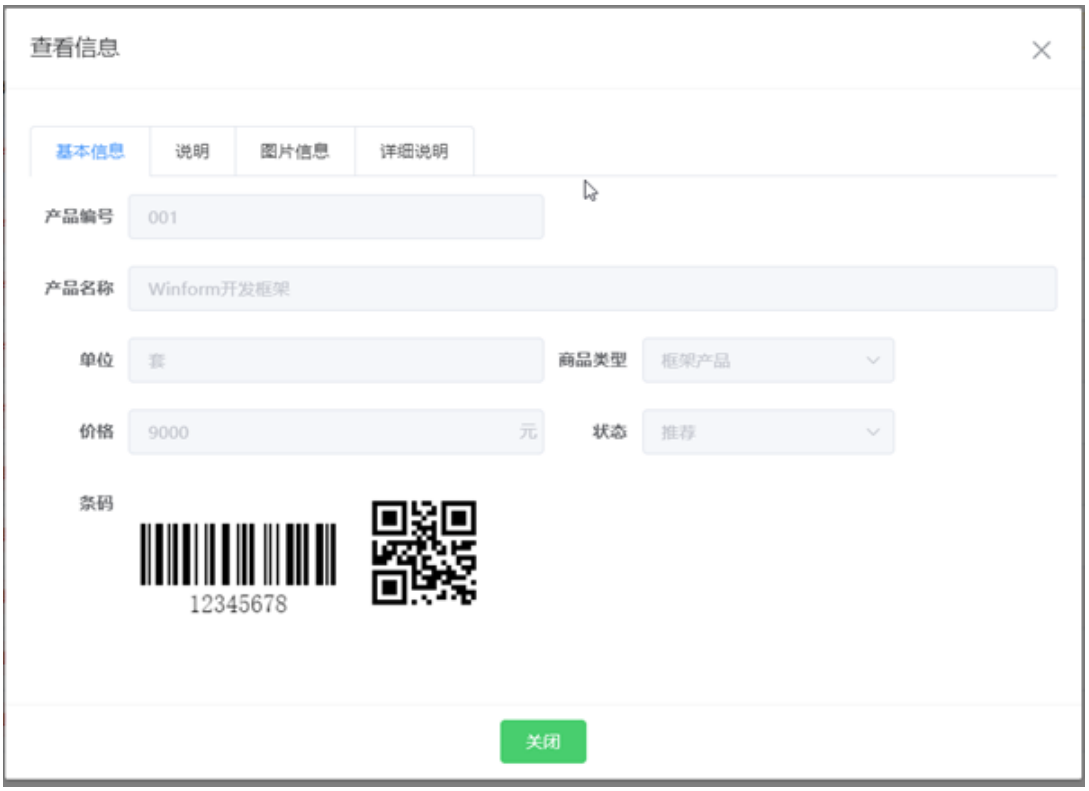
为了控制对话框的样式，我们这里注意下 footer 的 slot，这个我们一般是把处理按钮放在这里，如对于查看界面对话框，我们定义如下所示。

```
<el-dialog title="查看信息" :visible="isView" :modal-append-to-body="false" @close="closeDialog">
  <el-form ref="viewForm" :model="viewForm" label-width="80px">
    <el-tabs value="viewPage1" type="card">
      <el-tab-pane name="viewPage1" label="基本信息">…
      </el-tab-pane>
      <el-tab-pane label="说明">…
      </el-tab-pane>
      <el-tab-pane label="图片信息">…
      </el-tab-pane>

      <el-tab-pane label="详细说明">…
      </el-tab-pane>
    </el-tabs>

  </el-form>
  <div slot="footer" class="dialog-footer">
    <el-button type="success" @click="closeDialog">关闭</el-button>
  </div>
</el-dialog>
```

使用Tab分开内容

定义按钮放置区域

一般来说，对于表单内容比较多的场景，我们一般分开多个选项卡进行展示或者录入，这样方便管理，查看界面整体效果如下所示。



对于对话框的数据绑定，我们在打开对话框前，先通过 API 模块请求获得 JSON 数据，然后绑定在对应的对话框模型属性上即可，如对于查看界面，我们的处理逻辑如下所示。

```
showView(id) {
  var param = { id: id }
  GetProductDetail(param).then(data => {
    Object.assign(this.viewForm, data);
  })
  this.isView = true
},
```

对于表格的双击，我们同样绑定它的查看明细处理操作，如下模板代码和 JS 代码所示。

模板 HTML 代码如下：

```
<el-table
  v-loading="listLoading"
  :data="productlist"
  border
  fit
  stripe
  highlight-current-row
  :header-cell-style="{background:'#eef1f6',color:'#606266'}"
  @selection-change="selectionChange"
  @row-dblclick="rowDbclick"
>
```

JS 逻辑代码如下

```
rowDbclick(row, column) {
  var id = row.ID
  this.showView(id);
},
```

3.5.2. 常规界面组件的使用

一般情况下，我们使用界面组件的时候，参考下官网《Element 组件使用》，寻找对应组件的代码进行参考，就差不多了，这里还是就各种常规的 Element 组件进行大概的介绍吧。

1）表单和表单项、单文本框

对于表单，我们一般定义一个对应的名称，并设置它的 data 对应的模型名称即可，如下所示。

```
<el-form ref="viewForm" :model="viewForm" label-width="80px">
```

而表单项，一般是定义好表单项的 Label 即可，然后在其中插入对应的录入控件或者展示控件。如对于单文本组件使用，如下所示。

```
<el-form-item label="产品编号">
  <el-input v-model="editForm.ProductNo" />
</el-form-item>
```

其中 v-model="editForm.ProductNo" 就是对应绑定的数据。

而表单项，可以添加对字段的验证处理，在数据提交前，可以校验客户的录入是否有效等。

```
<el-form-item
  prop="email"
  label="邮箱"
  :rules="[
    { required: true, message: '请输入邮箱地址', trigger: 'blur' },
    { type: 'email', message: '请输入正确的邮箱地址', trigger: ['blur', 'change'] }
  ]"
>
  <el-input v-model="dynamicValidateForm.email"></el-input>
</el-form-item>
```

注意这里表单项，必须添加一个 prop 的属性设置，如 prop="email" 所示。

一般为了控制布局，我们还结合 el-row 进行一个布局的处理，如下代码所示（一行等于 span 为 24，span=12 也就是一行放两个控件组）。

```
<el-row>
  <el-col :span="12">
    <el-form-item label="产品编号">
      <el-input v-model="editForm.ProductNo" />
    </el-form-item>
  </el-col>
  <el-col :span="12">
    <el-form-item label="条码">
      <el-input v-model="editForm.BarCode" />
    </el-form-item>
  </el-col>
</el-row>
```

2）、下拉列表控件的绑定

下拉列表的绑定处理，也是通过 v-model 进行值的绑定，而选项则可以通过数据列表进行绑定。

```
<el-form-item label="商品类型">
  <el-select v-model="editForm.ProductType" filterable="" placeholder="请选择">
    <el-option
      v-for="(item, key) in typeList"
      :key="key"
      :label="item.value"
      :value="item.key"
    />
  </el-select>
</el-form-item>
```

而选项中的 typeList，我们可以在页面初始化的时候获取出来即可。

```
created() {
  // 获取产品类型，用于绑定字典等用途
  GetProductType().then(data => {
    if (data) {
      data.forEach(item => {
        this.productTypes.set(item.id, item.name)
        this.typeList.push({ key: item.id, value: item.name })
      })

      // 获取列表信息
      this.getlist()
    }
  });
},
```

对于 textarea 常规的多行文本框，其实和普通单行文本框处理差不多，指定它的 type="textarea" 和 rows 的数值即可。

```
<el-tab-pane label="说明" name="second">
  <el-form-item label="说明">
    <el-input v-model="editForm.Description" type="textarea" :rows="10" />
  </el-form-item>
</el-tab-pane>
```

而对于一些可能需要展示 HTML 内容的，我们可以使用 DIV 控件来展示，通过 v-html 标识来处理包含 HTML 代码的内容。

```
<el-tab-pane label="详细说明">
  <el-form-item label="详细说明">
    <div class="border-radius" v-html="viewForm.Note" />
  </el-form-item>
</el-tab-pane>
```

3）、图片展示

对于一些需要展示服务器图片，我们请求后，根据 Element 图片组件的设置处理即可，如下包括单个图片和多个图片的展示和预览操作。

图片展示的代码如下所示。

```
<el-tab-pane label="图片信息">
  <el-form-item label="封面图片">
    <el-image
      style="width: 100px; height: 100px"
      :src="viewForm.Picture"
      :preview-src-list="[viewForm.Picture]"
    />
  </el-form-item>
  <el-form-item label="Banner图片">
    <el-image
      style="width: 100px; height: 100px"
      :src="viewForm.Banner"
      :preview-src-list="[viewForm.Banner]"
    />
  </el-form-item>
  <el-form-item label="商品展示图片">
    <el-image
      v-for="item in viewForm.pics"
      :key="item.key"
      class="border-radius"
      :src="item.pic"
      style="width: 100px; height: 100px;padding:10px"
      :preview-src-list="getPreviewPics()"
    />
  </el-form-item>
</el-tab-pane>
```

上图中，如果是单个图片，那么预览我们设置一个集合为一个 url 即可，如 [viewForm.Banner]，如果是多个图片，需要通过一个函数来获取图片列表，如 getPreviewPics() 函数所示。

```
getPreviewPics() {
  // 转换viewForm.pics里面的pic集合
  var list = []
  if (this.viewForm.pics) {
    this.viewForm.pics.forEach(item => {
      if (item.pic) {
        list.push(item.pic)
      }
    })
  }
  return list
}
```

4）、第三方扩展控件

对于一些需要使用扩展组件的，我们一般搜索下解决方案，通过 npm 安装对应的组件即可解决，如对于条码和二维码，我使用 @chenfengyuan/vue-barcode 和 @chenfengyuan/vue-qrcode，一般在 Github 上搜索下关键字，总能找到一些很受欢迎的第三方组件。

```
"dependencies": {
  "@chenfengyuan/vue-barcode": "^1.0.1",
  "@chenfengyuan/vue-qrcode": "^1.0.2",
```

安装这些组件都有具体的说明，如下所示(如果卸载，直接修改 install 为 uninstall 即可）。

```
npm install @chenfengyuan/vue-barcode vue
```

以及

```
npm install @chenfengyuan/vue-qrcode vue
```

条码和二维码的展示效果如下所示

如果全局引入 barcode 和 qrcode 组件，我们在 main.js 里面引入即可，如下代码所示

```
// 引入barcode,qrcode
import VueBarcode from '@chenfengyuan/vue-barcode';
import VueQrcode from '@chenfengyuan/vue-qrcode';
Vue.component(VueBarcode.name, VueBarcode);
```

富文本编辑，我这里采用了 Tinymce 第三方组件来实现编辑处理，展示效果如下所示。

代码如下所示

```
<el-tab-pane label="详细说明" name="third">
  <el-form-item label="详细说明">
    <tinymce v-model="editForm.Note" :height="300" />
  </el-form-item>
</el-tab-pane>
```

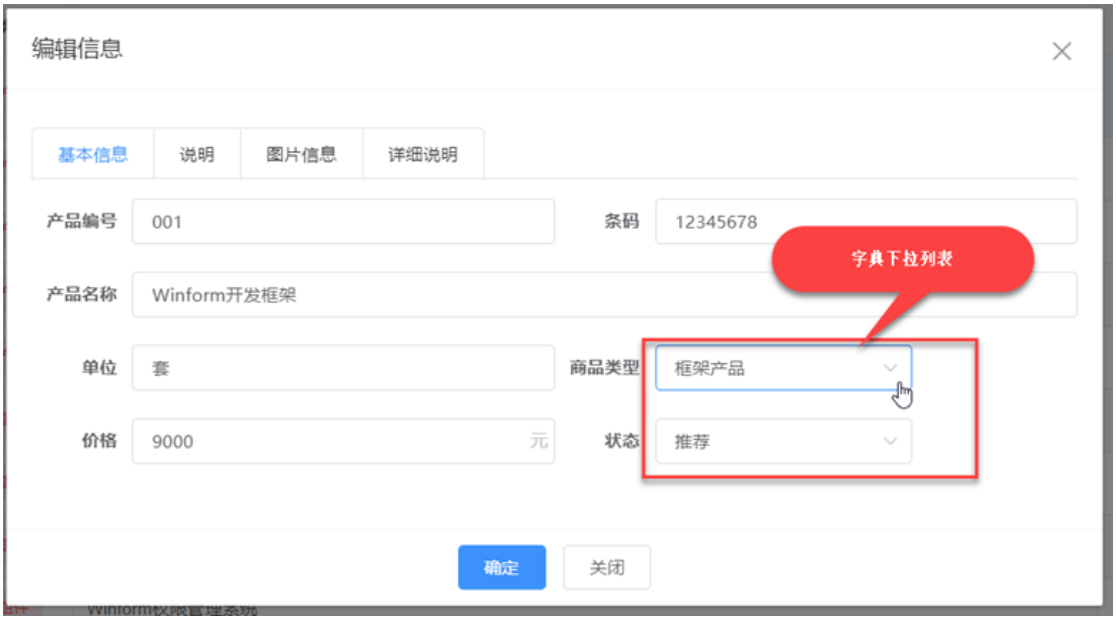
以上就是一些常规的界面组件的使用，后面在继续介绍文件上传和图片结合的操作。

3.5.3. 自定义组件的创建使用

使用 Vue 的比以往 BS 开发的好处，就是可以很容易实现组件化，这点很好，一旦我们定义好一个控件，就可以在多个界面里面进行使用，非常方便，而且封装性可以根据自己的需要进行处理。

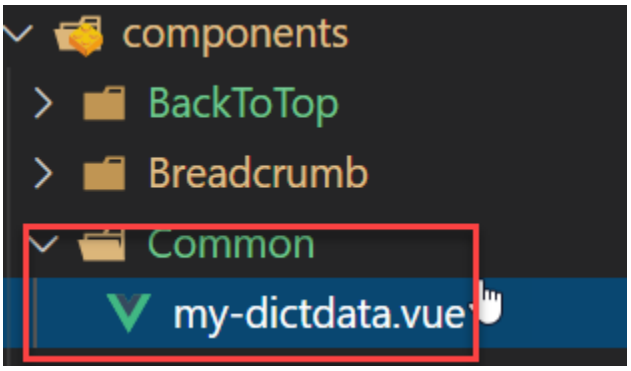
查询区域一般的界面效果如下所示，除了包含一些常用的查询条件，一般会有一些下拉列表，这些可能是后台字典里面绑定的内容，可以考虑作为一个通用的字典下拉列表组件来做。

其实界面录入的时候，也往往需要这些条件下拉列表的。



那么我们来定义一个自定义组件，并在界面上使用看看。

在 Components 目录创建一个目录，并创建一个组件的 vue 文件，命名为 my-dictdata.vue，如下所示。



界面模板代码我们就一个 select 组件为主即可。

```
<template>
  <el-select v-model="svalue" filterable clearable placeholder="请选择">
    <el-option
      v-for="(item, index) in dictItems"
      :key="index"
      :label="item.Text"
      :value="item.Value"
    />
  </el-select>
</template>
```

script 脚本逻辑代码如下所示。

```
<script>
// 引入API模块类方法
import { GetDictData } from '@/api/dictdata'

export default {
  name: 'MyDictdata', // 组件的名称
  props: {
    typeName: { // 字典类型方式，从后端字典接口获取数据
      type: String,
      default: ''
    },
    options: {// 固定列表方式，直接绑定
      type: Array,
      default: () => { return [] }
    }
  },
  data() {
    return {
      dictItems: [], // 设置的字典列表
      svalue: '' // 选中的值
    }
  },
  watch: {
    // 判断下拉框的值是否有改变
    svalue(val, oldVal) {
      if (val !== oldVal) {
        this.$emit('input', this.svalue);
      }
    }
  },
```

```
  mounted() {
    var that = this;

    if (this.typeName && this.typeName !== '') {
      // 使用字典类型，从服务器请求数据
      GetDictData(this.typeName).then(data => {
        if (data) {
          data.forEach(item => {
            if (item && typeof (item.Value) !== 'undefined' && item.Value !== '') {
              that.dictItems.push(item)
            }
          });
        }
      })
    } else if (this.options && this.options.length > 0) {
      // 使用固定字典列表
      this.options.forEach(item => {
        if (item && typeof (item.Value) !== 'undefined' && item.Value !== '') {
          that.dictItems.push(item)
        }
      });
    }
    // 设置默认值
    this.svalue = this.value;
  },
  methods: {

  }
}
</script>
```

主要就是处理字典数据的获取，并绑定到模型对象上即可。

在页面上使用前，需要引入我们定义的组件

```
import myDictdata from '@/components/Common/my-dictdata'
```

然后包含进去 components 里面即可

```
export default {
  components: { myDictdata },
```

那么原来需要直接使用 select 组件的代码

```
<el-select v-model="searchForm.ProductType" filterable clearable placeholder="请选择">
  <el-option
    v-for="(item, key) in typeList"
    :key="key"
    :label="item.value"
    :value="item.key"
  />
</el-select>
```

则可以精简为一行代码

```
<my-dictdata v-model="searchForm.ProductType" type-name="商品类型" />
```

而对于固定列表的，我们也可以通用的处理代码

```
<my-dictdata v-model="searchForm.Status" :options="Status" />
```

其中 Status 是定义的一个对象集合

```
Status: [
  { Text: '正常', Value: 0 },
  { Text: '推荐', Value: 1 },
  { Text: '停用', Value: 2 }
]
```

是不是非常方便，而得到的效果则不变。

以上就是多个页面内容，通过对话框层模式整合在一起，并介绍如何使用，以及对界面中常见的 Element 组件进行介绍如何使用，以及定义一个字典列表的主定义组件，用于简化界面代码使用，

3.6. 介绍一些常规的 JS 处理函数

在我们使用 VUE+Element 处理界面的时候，往往碰到需要利用 JS 集合处理的各种方法，如 Filter、Map、reduce 等方法，也可以涉及到一些对象属性赋值等常规的处理或者递归的处理方法，以前对于这些不是很在意，但往往真正使用的时候，需要了解清楚，否则很

容易脑袋出现短路的情况。这里列出一些在 VUE+Element 前端开发中经常碰到的 JS 处理场景，供参考学习。

3.6.1. 常规集合的 filter、map、reduce 处理方法

filter 函数的主要用途是对数组元素进行过滤，并返回一个符合条件的元素的数组

```
const nums = [10,20,30,111,222,333]
let newNums=nums.filter(function(n){
    return n<100
})
```

输出：[10,20,30]

map 函数是对数组每个元素的映射操作，并返回一个新数组，原数组不会改变将 newNums 中每个数字乘 2

```
const nums = [10,20,30,111,222,333]
let newNums=nums.map(function(n){
    return n*2
})
```

输出：[20,40,60,222,666]

reduce 函数主要用于对数组所有元素的汇总操作，如全部相加、相乘等

```
const nums = [10,20,30,111,222,333]
let newNums=nums.reduce(function(preValue,n){
    return PreValue+n
},0)
```

输出：726

有时候可以结合几种处理方式一起，如下综合案例所示。

```
const nums = [10,20,30,111,222,333]
let newNums=nums.filter(function(n){
    return n<100
}).map(function(n){
    return n*2
}).reduce(function(preValue,n){
    return preValue+n
},0)
```

结果：120

另外还有一个数组集合的 find 方法，和 filter 方法类似。

find()方法主要用来返回数组中符合条件的第一个元素（没有的话，返回 undefined）

```
var Array = [1,2,3,4,5,6,7];
 var result = Array.find(function(value){
     return value > 5;   //条件
 });
 console.log(result);//6
 console.log(Array);//[1,2,3,4,5,6,7]
```

同样我们也可以在 vue 里面，利用 require.context 的处理机制，遍历文件进行处理，也需要用到了 filter，如下代码所示。

下面代码是我对某个文件夹里面的文件进行一个过滤处理操作

```
const req = require.context('vue-awesome/icons', true, /\.js$/)
const requireAll = requireContext => requireContext.keys()

const re = /\.\/(.*)\.js/

const vueAwesomeIcons = requireAll(req).filter((key) => { return
key.indexOf('index.js') < 0 }).map(i => {
  return i.match(re)[1]
})

export default vueAwesomeIcons
```

3.6.2. 递归处理

有时候，我们需要从一个 JSON 集合里面，由于集合是嵌套的，如 children 里面还有 chilren 集合，根据某个关键属性进行查询，这种处理方式就要用到递归了。

例如我定义的一个菜单集合里面，就是这样一个嵌套的结构，需要根据名称来获得对应

的对象的时候，就涉及到了一个递归处理函数。

首先我们来看看菜单的 JSON 集合。

```
// 此菜单数据一般由服务器端返回
export const asyncMenus = [
  {
    id: '1',
    pid: '-1',
    text: '首页',
    icon: 'dashboard',
    name: 'dashboard'
  },
  {
    id: '2',
    pid: '-1',
    text: '产品信息',
    icon: 'table',
    children: [
      {
        id: '2-1',
        pid: '2',
        text: '产品展示',
        name: 'product-show',
        icon: 'table'
      }]
  },
  {
    id: '3',
    pid: '-1',
    text: '杂项管理',
```

```
    icon: 'example',
    children: [
      {
        id: '3-1',
        pid: '3',
        text: '图标管理',
        name: 'icon',
        icon: 'example'
      },
      {
        id: '3-3',
        pid: '3',
        text: '树功能展示',
        name: 'tree',
        icon: 'tree'
      },
      {
        id: '3-2',
        pid: '3',
        text: '二级菜单 2',
        icon: 'tree',
        children: [
          {
            id: '3-2-2',
            pid: '3-2',
            text: '三级菜单 2',
            name: 'menu1-1',
            icon: 'form'
          }
```

```
      ]
    }
  ]
}]
```

如果我们需要根据 ID 来遍历查询，就是一个典型的递归查询处理。

```
// 根据菜单 id 来获取对应菜单对象
FindMenuById(menuList, menuid) {
  for (var i = 0; i < menuList.length; i++) {
    var item = menuList[i];
    if (item.id && item.id === menuid) {
      return item
    } else if (item.children) {
      var foundItem = this.FindMenuById(item.children, menuid)
      if (foundItem) { // 只有找到才返回
        return foundItem
      }
    }
  }
}
```

这里值得注意的是，不能在递归的时候，使用下面直接返回

```
return this.FindMenuById(item.children, menuid)
```

而需要判断是否有结果在进行返回，否则嵌套递归就可能返回 undefined 类型

```
var foundItem = this.FindMenuById(item.children, menuid)
if (foundItem) { // 只有找到才返回
  return foundItem
}
```

3.6.3. forEach 遍历集合处理

在很多场合，我们也需要对集合进行一个 forEach 的遍历处理，如下根据它的键值进行处理，注册全局过滤器的处理操作

```
// 导入全局过滤器
import * as filters from './filters'
// 注册全局过滤器
Object.keys(filters).forEach(key => {
  Vue.filter(key, filters[key])
})
```

或者我们在通过 API 方式获取数据后，对集合进行处理的操作

```
    // 获取产品类型，用于绑定字典等用途
    GetProductType().then(data => {
      if (data) {
        this.treedata = [];// 树列表清空
        data.forEach(item => {
          this.productTypes.set(item.id, item.name)
          this.typeList.push({ key: item.id, value: item.name })

          var node = { id: item.id, label: item.name }
          this.treedata.push(node)
        })

        // 获取列表信息
        this.getlist()
      }
    });
```

又或者请求字典数据的时候，进行一个非空值的判断处理。

```
      // 使用字典类型，从服务器请求数据
      GetDictData(this.typeName).then(data => {
        if (data) {
          data.forEach(item => {
            if (item && typeof (item.Value) !== 'undefined' &&
item.Value !== '') {
              that.dictItems.push(item)
            }
          });
        }
      })
```

forEach()方法也是用于对数组中的每一个元素执行一次回调函数，**但它没有返回值**（或者说它的返回值为 undefined，即便我们在回调函数中写了 return 语句，返回值依然为 undefined）

注意： 如果 forEach 里有两个参数，则第一个参数为该集合里的元素，第二个参数为集合的索引。

3.6.4. Object.assign 赋值方法

在有些场合，我们需要把全新的集合，复制到另一个对象上，替换原来对象的属性值，那么我们可以利用 Object 对象的 assign 方法。

如在编辑界面展示的时候，把请求到的对象属性复制到表单对象上。

```
      var param = { id: id }
      GetProductDetail(param).then(data => {
        Object.assign(this.editForm, data);
      })
```

或者查询的时候，获得查询条件，进行部分替换

```
      // 构造常规的分页查询条件
      var param = {
        type: this.producttype === 'all' ? '' : this.producttype,
```

```
        pageindex: this.pageinfo.pageindex,
        pagesize: this.pageinfo.pagesize
      };

      // 把 SearchForm 的条件加入到 param 里面，进行提交查询
      param.type = this.searchForm.ProductType // 转换为对应属性
      Object.assign(param, this.searchForm);
```

3.6.5. slice() 方法

slice() 方法可从已有的数组中返回选定的元素。

语法如下所示。

```
arrayObject.slice(start,end)
```

如下案例所示。

```
let red = parseInt(color.slice(0, 2), 16)
let green = parseInt(color.slice(2, 4), 16)
let blue = parseInt(color.slice(4, 6), 16)
```

或者我们结合 filter 函数对图标集合进行获取部分处理

```
vueAwesomeIconsFiltered: function() {
  const that = this
  var list = that.vueAwesomeIcons.filter(item => { return
item.indexOf(that.searchForm.label) >= 0 })
  if (that.searchForm.pagesize > 0) {
    return list.slice(0, that.searchForm.pagesize)
  } else {
    return list;
  }
```

```
}
```

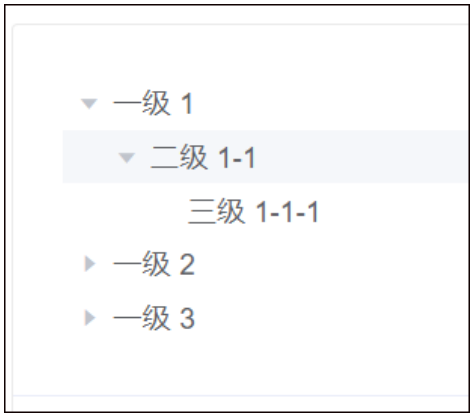
3.7. 树列表组件的使用

前面介绍过一些常规的界面组件的处理，主要介绍到单文本输入框、多文本框、下拉列表，以及按钮、图片展示、弹出对话框、表单处理，这里补充这一个主题，介绍树列表组件和下拉列表树组件在项目中的使用，以及一个 SplitPanel 的组件。

3.7.1. 常规树列表控件的使用

众所周知，一般界面很多情况涉及到树列表的处理，如类型展示，如果是一层的，可以用下拉列表代替，如果是多个层级的，采用树控件展示会更加直观。

在 Element 里面也有一个 el-tree 的控件，如下所示，这里主要对它的各种属性和方法进行介绍。



简单的代码如下所示

```
<el-tree :data="data"  @node-click="handleNodeClick"></el-tree>
```

主要在 script 部分里面指定它的 data 数据，以及单击节点的事件处理，结合卡片控件的展示，我们可以把树放在其中进行展示

界面代码如下所示，通过 default-expand-all 可以设置全部展开，icon-class 指定节点图标（也可以默认不指定）

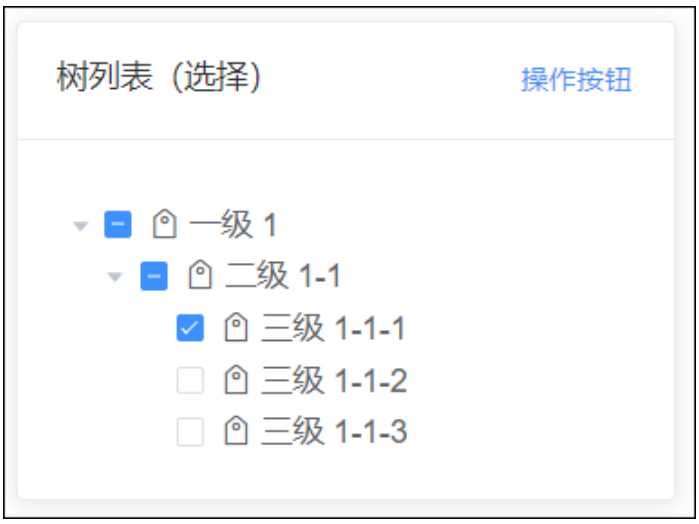
```
<el-card class="box-card">
  <div slot="header" class="clearfix">
    <span>树列表</span>
    <el-button style="float: right; padding: 3px 0" type="text">操作按钮</el-button>
  </div>
  <div>
    <el-tree
      style="padding-top:10px"
      :data="treedata"
      node-key="id"
      default-expand-all
      icon-class="el-icon-price-tag"
      highlight-current
      @node-click="handleNodeClick"
    >
      <span slot-scope="{ node, data }" class="custom-tree-node">
        <span>
          <i :class="node.icon ? node.icon : 'el-icon-price-tag'" />
          {{ node.label }}
         
        </span>
      </span>
    </el-tree>
  </div>
</el-card>
```

其中界面里面，我们通过 class="custom-tree-node"，来指定树列表的展现内容，可以加入图标等信息

而在 script 里面，定义了一个 treedata 的属性

```
// 初始化树列表
treedata: [
  {
    label: '一级 1',
    id: '1',
    children: [{
      id: '1-1',
      label: '二级 1-1',
      children: [{
        label: '三级 1-1-1',
        id: '1-1-1'
      }, {
        label: '三级 1-1-2',
        id: '1-1-2'
      }, {
        label: '三级 1-1-3',
        id: '1-1-3'
      }]
    }]
  }
]
```

如果设置有选择框，得到界面如下所示。

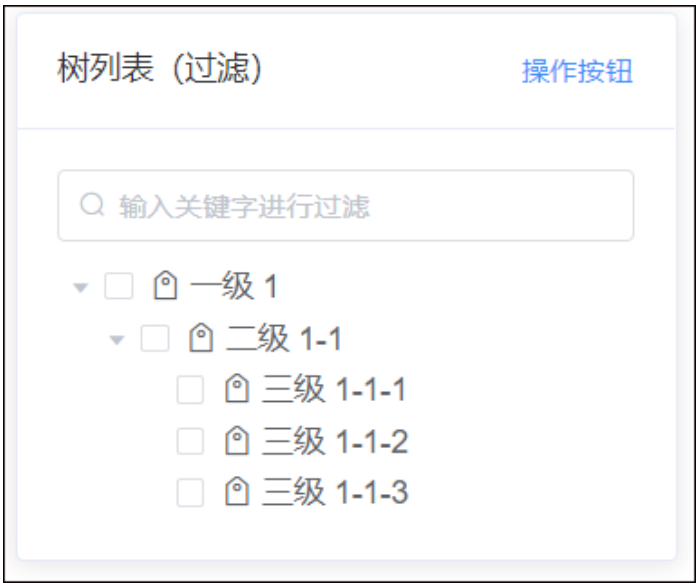


主要设置 show-checkbox 和 @check-change="handleCheckChange" 即可。

界面代码如下所示

```
<el-tree
  style="padding-top:10px"
  :data="treedata"
  node-key="id"
  default-expand-all
  highlight-current
  show-checkbox
  :default-checked-keys="['1-1-1']"
  @node-click="handleNodeClick"
  @check-change="handleCheckChange"
>
  <span slot-scope="{ node, data }" class="custom-tree-node">
    <span>
      <i :class="node.icon ? node.icon : 'el-icon-price-tag'" />
      {{ node.label }}
     
    </span>
  </span>
```

而对于树列表，可以进行一个过滤处理操作，如下界面所示。



在内容区增加一个 input 的文本框进行过滤处理，并绑定对应的属性变量

```
<el-input
  v-model="filterText"
  placeholder="输入关键字进行过滤"
  clearable
  prefix-icon="el-icon-search"
/>
```

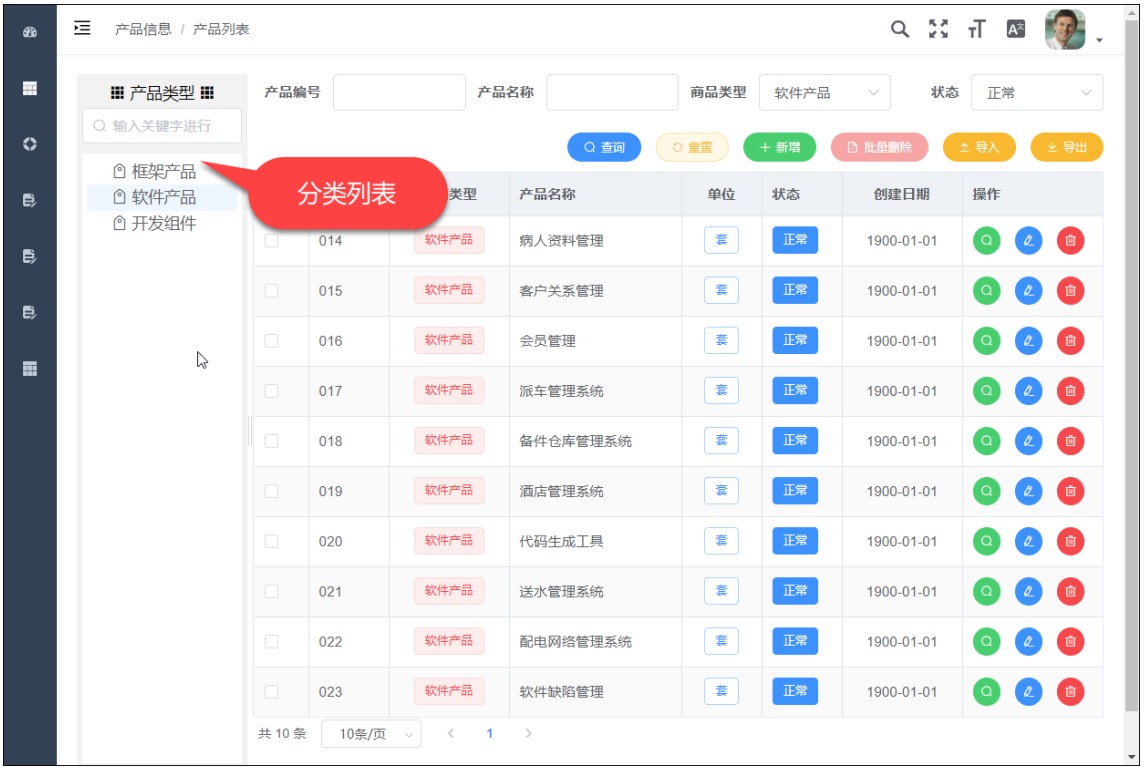
树列表控件需要增加过滤函数绑定 :filter-node-method="filterNode"，如下代码所示。

```
<el-tree
  ref="tree"
  class="filter-tree"
  style="padding-top:10px"
  :data="treedata"
  node-key="id"
  default-expand-all
  highlight-current
  show-checkbox
  :filter-node-method="filterNode"
  @check-change="handleCheckChange"
  @node-click="handleNodeClick"
>
  <span slot-scope="{ node, data }" class="custom-tree-node">
    <span>
      <i :class="node.icon ? node.icon : 'el-icon-price-tag'" />
      {{ node.label }}
     
    </span>
  </span>
</el-tree>
```

script 的处理代码如下所示，需要 watch 过滤的绑定值，变化就进行过滤处理。

```
watch: {
  filterText(val) {
    this.$refs.tree.filter(val);
  }
},

methods: {
  filterNode(value, data) {
    if (!value) return true;
    return data.label.indexOf(value) !== -1;
  },
  handleNodeClick(data) {
    console.log(data);
    if (data && typeof (data.id) !== 'undefined') {
    // 操作代码
    }
  },
```

为了在列表结合中进行快速的过滤，我们可以在上次介绍的列表界面里面增加一个树列表的快速查询处理。如下界面所示。



这里列表里面增加了一个第三方组件 splitpanes，用来划分区块展示，而且可以拖动，非常不错，地址是：https://github.com/antoniandre/splitpanes

这个组件的 Demo 展示地址如下所示：https://antoniandre.github.io/splitpanes

效果大概如下所示

npm 安装如下所示

```
npm i --S splitpanes
```

安装成功后，然后在 vue 文件的 script 部分里面引入即可

```
import { Splitpanes, Pane } from 'splitpanes'
```

```
import 'splitpanes/dist/splitpanes.css'
```

它的使用代码也很简单

```
<splitpanes style="height: 400px">
  <pane min-size="20">1</pane>
  <pane>
    <splitpanes horizontal>
      <pane>2</pane>
      <pane>3</pane>
      <pane>4<pane>
    </splitpanes>
  </pane>
  <pane>5</pane>
</splitpanes>
```

我的列表界面使用了两个 Panel 即可实现左侧树的展示，和右侧常规列表查询的处理。

```
<template>
  <div class="app-container">
    <splitpanes class="default-theme">
      <pane min-size="10" max-size="20">…
      </pane>
      <pane style="background-color:#fff">
        <el-form ref="searchForm" :model="searchForm" label-width="80px">…
        </el-form>
        <el-row style="float:right;padding-bottom:10px">…
        </el-row>

        <el-table…
        >…
        </el-table>
        <div class="block" style="height:70px;">…
        </div>
      </pane>
    </splitpanes>
```

3.7.2. 下拉框树列表的处理

除了常规的树列表展示内容外，我们也需要一个在下拉列表中展示树内容的界面组件。

这里又得引入一个第三方的界面组件，因此 Element 的 Select 组件不支持树列表。

GitHub 地址：https://github.com/riophae/vue-treeselect

官网地址：https://vue-treeselect.js.org/

NPM 安装：

```
npm install --save @riophae/vue-treeselect
```

界面代码如下所示。

```
<template>
  <div id="app">
    <treeselect v-model="value" :multiple="true" :options="options" />
  </div>
</template>
```

这里的 value 就是选中的集合，options 则是树列表的节点数据。

```
<script>
  // import the component
  import Treeselect from '@riophae/vue-treeselect'
  // import the styles
  import '@riophae/vue-treeselect/dist/vue-treeselect.css'

  export default {
    // register the component
    components: { Treeselect },
    data() {
      return {
        // define the default value
        value: null,
        // define options
        options: [ {
          id: 'a',
          label: 'a',
          children: [ {
            id: 'aa',
            label: 'aa',
          }, {
            id: 'ab',
            label: 'ab',
          } ],
        }, {
          id: 'b',
          label: 'b',
        }, {
          id: 'c',
          label: 'c',
        } ],
      }
    },
  }
</script>
```

我的测试界面代码如下所示

```
<div style="height:180px">
  <!--
      v-model 绑定选中的集合
      options 树节点数据
       defaultExpandLevel 展开层次，Infinity为所有
       flat 为子节点不影响父节点，不关联
   -->
  <treeselect
    v-model="value"
    :options="treedata"
    :multiple="true"
    :flat="true"
    :default-expand-level="Infinity"
    :open-on-click="true"
    :open-on-focus="true"
    clearable
    :max-height="200"
  />
</div>
```

```
<script>
// import vue-treeselect component
import Treeselect from '@riophae/vue-treeselect'
// import the styles
import '@riophae/vue-treeselect/dist/vue-treeselect.css'

export default {
  name: 'Tree',
  components: { Treeselect },
  data() {
    return {
      // 过滤条件
      filterText: '',
      // 初始化树列表
      treedata: [
        {
          label: '一级 1',
          id: '1',
          children: [{
            id: '1-1',
            label: '二级 1-1',
            children: [{
              label: '三级 1-1-1',
              id: '1-1-1'
            }, {
              label: '三级 1-1-2',
              id: '1-1-2'
            }, {
              label: '三级 1-1-3',
              id: '1-1-3'
            }]
          }]
        }
      ],
      1-1
```

来一张几个树列表一起的对比展示界面。

以上就是普通树列表和下拉列表树展示的界面效果，往往我们一些特殊的界面处理，就需要多利用一些封装良好的第三方界面组件实现，可以丰富我们的界面展示效果。

3.8. 界面语言国际化的处理

我们开发的系统，一般可以不用考虑语言国际化的问题，大多数系统一般是给本国人使用的，而且直接使用中文开发界面会更加迅速 一些，不过框架最好能够支持国际化的处理，以便在需要的时候，可以花点时间来实现多语言切换的处理，使系统具有更广泛的受众用户。

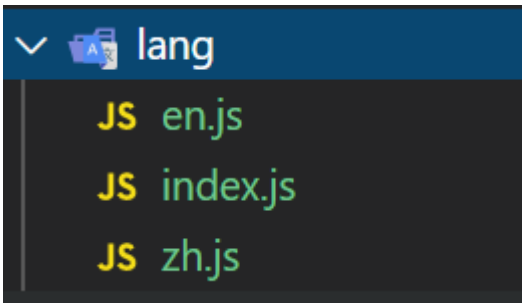
VUE+Element 前端应用实现国际化的处理还是非常方便的，一般在 Main.js 函数里面引入语言文件，然后在界面上进行一定的处理，把对应的键值转换为对应语言的语义即可。这里介绍在VUE+Element 前端应用中如何实现在界面快速的支持多语言国际化的处理逻辑代码。

3.8.1. main 入口函数支持

Element 组件内部默认使用中文，若希望使用其他语言，则需要进行多语言设置。以英文为例，在 main.js 中：

```
// 完整引入 Element
import Vue from 'vue'
import ElementUI from 'element-ui'
import locale from 'element-ui/lib/locale/lang/en'

Vue.use(ElementUI, { locale })
```

由于我们现在是需要处理多语言的切换，那么，我们在 src 下面的一个目录里面创建一个 lang 目录，在其中里面编辑 zh.js 和 en.js 分别代表中英文语言对照信息，index.js 文件则为引入这两个文件的处理关系。



在 index.js 里面，需要设置一个函数，用来获取 Cookie 里面存储的语言，如果没有找到，以浏览器国际化语言为准，如下代码所示。

```
export function getLanguage() {
  const chooseLanguage = Cookies.get('language')
  if (chooseLanguage) return chooseLanguage

  // 如果没有选择语言，那么使用浏览器语言
  const language = (navigator.language || navigator.browserLanguage).toLowerCase()
  const locales = Object.keys(messages)
  for (const locale of locales) {
    if (language.indexOf(locale) > -1) {
      return locale
    }
  }
  return 'en'
}
```

其中代码行

```
const locales = Object.keys(messages)
```

是获取 message 对象里面的键，如下所示。

```
import Vue from 'vue'
import VueI18n from 'vue-i18n'
import Cookies from 'js-cookie'
import elementEnLocale from 'element-ui/lib/locale/lang/en' // element-ui lang
import elementZhLocale from 'element-ui/lib/locale/lang/zh-CN'// element-ui lang
import enLocale from './en'
import zhLocale from './zh'

Vue.use(VueI18n)

// 定义对应语言键，展开对应的键值对应表
const messages = {
  en: {
    ...enLocale,
    ...elementEnLocale
  },
  zh: {
    ...zhLocale,
    ...elementZhLocale
  }
}
```

其中 message 就是一个两个语言（en/zh）字典下的对照表，包含各自对应键值下的内容。

然后整个 index.js 文件就是公布对应的多语言处理接口和属性。

```
const i18n = new VueI18n({
  // set locale
  // options: en | zh | es
  locale: getLanguage(),
  // set locale messages
  messages
})

export default i18n
```

然后在 main.js 函数里面处理国际化的处理即可

```
// set ElementUI lang to EN`
Vue.use(ElementUI, {
  size: Cookies.get('size') || 'medium', // set element-ui def
  i18n: (key, value) => i18n.t(key, value) })

// 如果想要中文版 element-ui，按如下方式声明
// Vue.use(ElementUI)

Vue.config.productionTip = false
Vue.config.devtools = process.env.NODE_ENV === 'development'

new Vue({
  el: '#app',
  router,
  store,
  i18n,
  render: h => h(App)
})
```

有了这些准备，那么我们在界面上就可以调用对应的键来获取对应语言的语义了，

3.8.2. 界面处理实现

首先，我们编辑一下对应国际化的键值内容，例如中文参照如下所示。

```
JS zh.js
src > lang > JS zh.js > default
export default
{
  navbar: {···
  },
  login: {
    title: '系统登录',
    logIn: '登录',
    username: '账号',
    password: '密码',
    any: '随便填写',
    thirdparty: '第三方登录',
    thirdpartyTips: '本地不能模拟，请结合自己业务进行模拟！！！'
  },
  theme: {···
```

例如对应登录界面上，界面效果如下所示。

或者

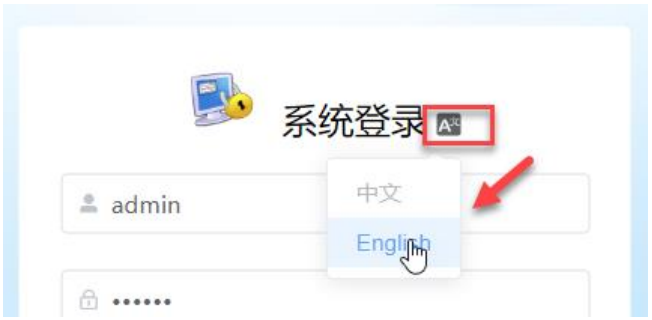
其中里面的文本内容，我们都是以国际化处理内容。

如登陆表单里面的代码如下所示。

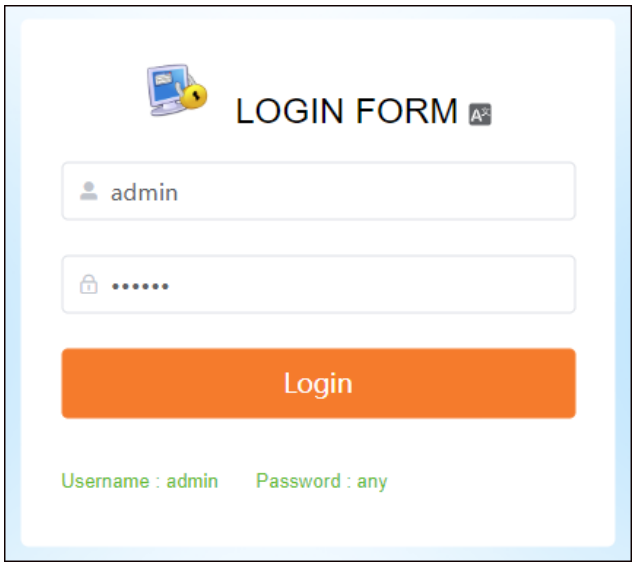
```
<el-form ref="loginForm" :model="loginForm" :rules="rules" class="loginForm">
  <el-form-item prop="username" class="login-item">
    <el-input
      v-model="loginForm.username"
      class="area"
      type="text"
      :placeholder="$t('login.username')"
      prefix-icon="el-icon-user-solid"
      @keyup.enter.native="submitForm('loginForm')"
    />
  </el-form-item>
  <el-form-item prop="password" class="login-item">
    <el-input
      v-model="loginForm.password"
      class="area"
      type="password"
      :placeholder="$t('login.password')"
      prefix-icon="el-icon-lock"
      @keyup.enter.native="submitForm('loginForm')"
    />
  </el-form-item>

  <el-form-item>
    <el-button :loading="loading" type="primary" class="submit_btn" @click="submitForm('loginForm')">{{ $t('login.logIn') }}</el-button>
  </el-form-item>
  <div class="tiparea">
    <span style="margin-right:20px;">{{ $t('login.username') }} : admin</span>
    <span> {{ $t('login.password') }} : {{ $t('login.any') }}</span>
  </div>
</el-form>
```

我们多处采用了类似 $t('login.username') 的函数处理方式来动态获取对应语言的内容即可，其中$t(）函数里面就是对应的语义解析的键参数，对应我们 lang/zh.js 里面或者 lang/en.js 里面的内容即可。

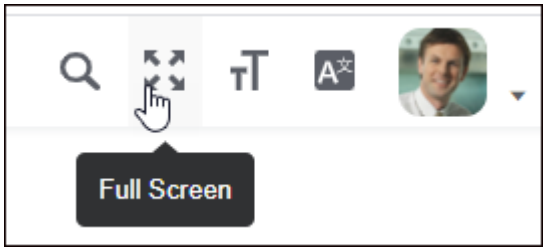
其中多语言切换的时候，单击图标就可以切换为其他语言内容了。



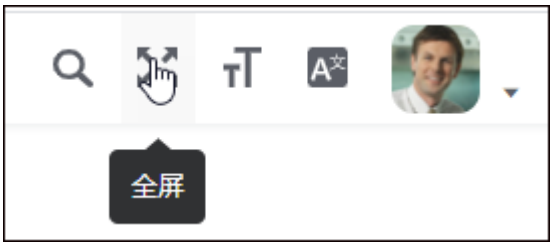
切换英文后界面如下所示



同样，其他地方，如果需要切换多语言的国际化处理，也可以使用$t 的转义方式，在顶部导航栏里面，我们可以设置得到多语言支持的界面。



中文界面提示如下所示。



这部分的实现代码是在组件模块里面，一样可以实现国际化的处理的。

```
<template v-if="device!=='mobile'">
  <search id="header-search" class="right-menu-item" />
  <error-log class="errLog-container right-menu-item hover-effect" />
  <el-tooltip :content="$t('navbar.fullscreen')" effect="dark" placement="bottom">
    <screenfull id="screenfull" class="right-menu-item hover-effect" />
  </el-tooltip>
  <el-tooltip :content="$t('navbar.size')" effect="dark" placement="bottom">
    <size-select id="size-select" class="right-menu-item hover-effect" />
  </el-tooltip>
  <el-tooltip :content="$t('navbar.language')" effect="dark" placement="bottom">
    <lang-select class="right-menu-item hover-effect" />
  </el-tooltip>
</template>
```

3.9. 基于 vue-echarts 处理各种图表展示

在我们做应用系统的时候，往往都会涉及图表的展示，综合的图表展示能够给客户带来视觉的享受和数据直观体验，同时也是增强客户认同感的举措之一。基于图表的处理，我们一般往往都是利用对应第三方的图表组件，然后在这个基础上为它的数据模型提供符合要求的图表数据即可。

VUE+Element 前端应用也不例外，我们这里使用了基于 vue-echarts 组件模块来处理各种图表 vue-echarts 是对 echarts 图表组件的封装。

3.9.1. 图表组件的安装使用

首先使用 npm 安装 vue-echarts 组件。

git 地址：https://github.com/ecomfe/vue-echarts

NPM 安装命令

```
npm install echarts vue-echarts
```

然后在对应模块页面里面引入对应的组件对象，如下代码所示。

```
<script>
import ECharts from 'vue-echarts' // 主图表对象
import 'echarts/lib/chart/line' // 曲线图表
import 'echarts/lib/chart/bar' // 柱状图
import 'echarts/lib/chart/pie' // 饼状图
import 'echarts/lib/component/tooltip' // 提示信息
```

接着在 Vue 组件里面对象中加入对象即可。

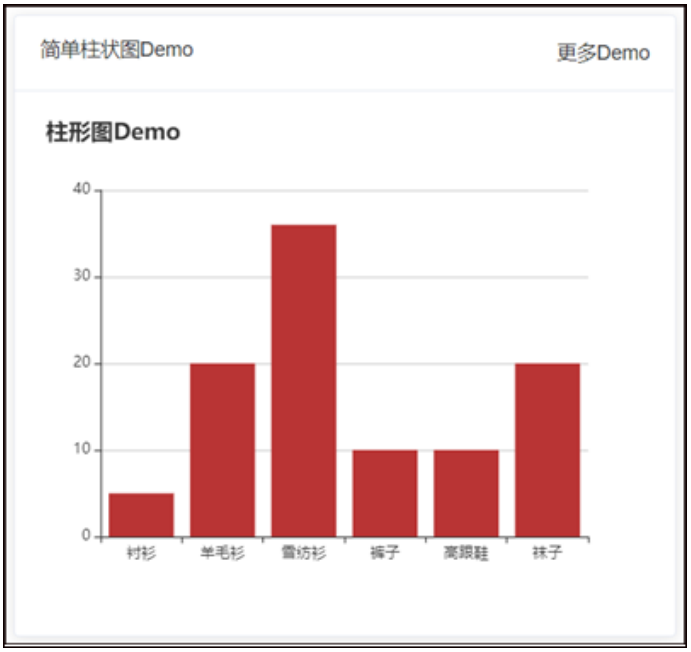
```
export default {
  components: {
    'v-chart': ECharts
  },
```

如果是全局注册使用，那么可以在 main.js 里面进行加载

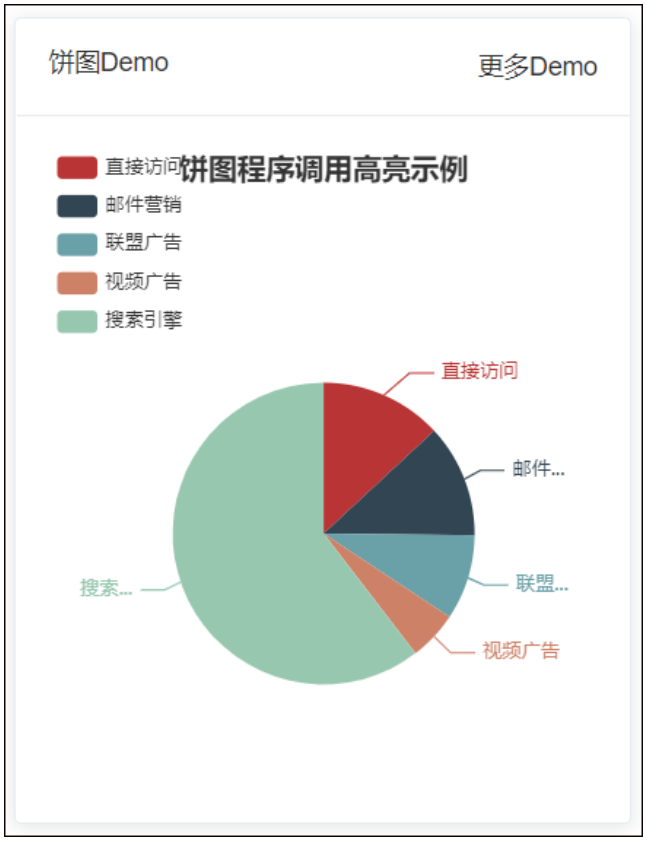
```
// 注册组件后即可使用
Vue.component('v-chart', VueECharts)
```

我们来看看图表展示的效果图

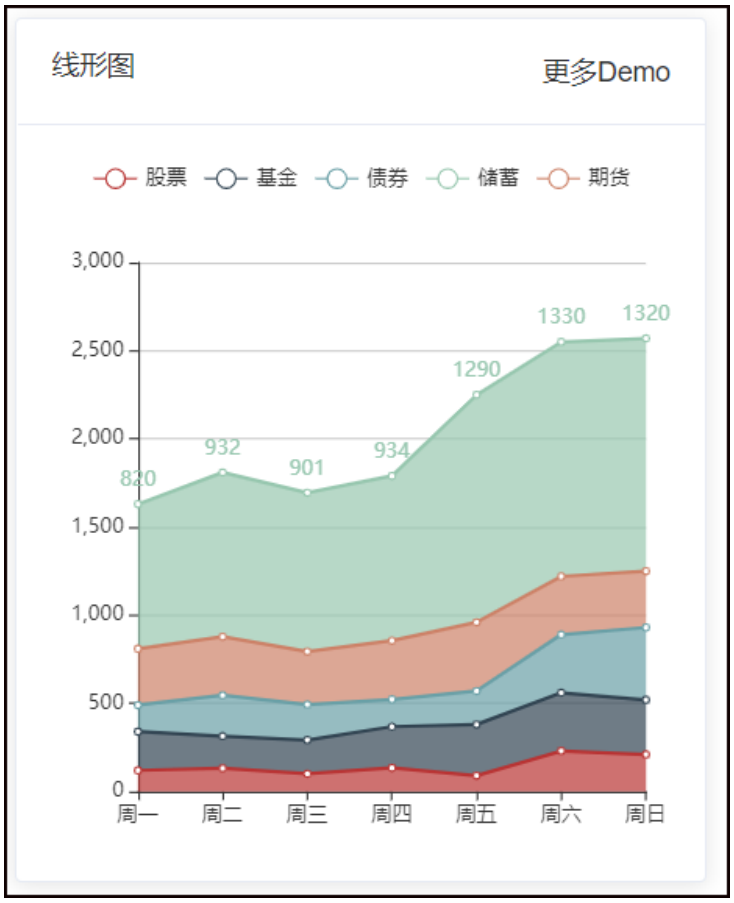
柱状图效果



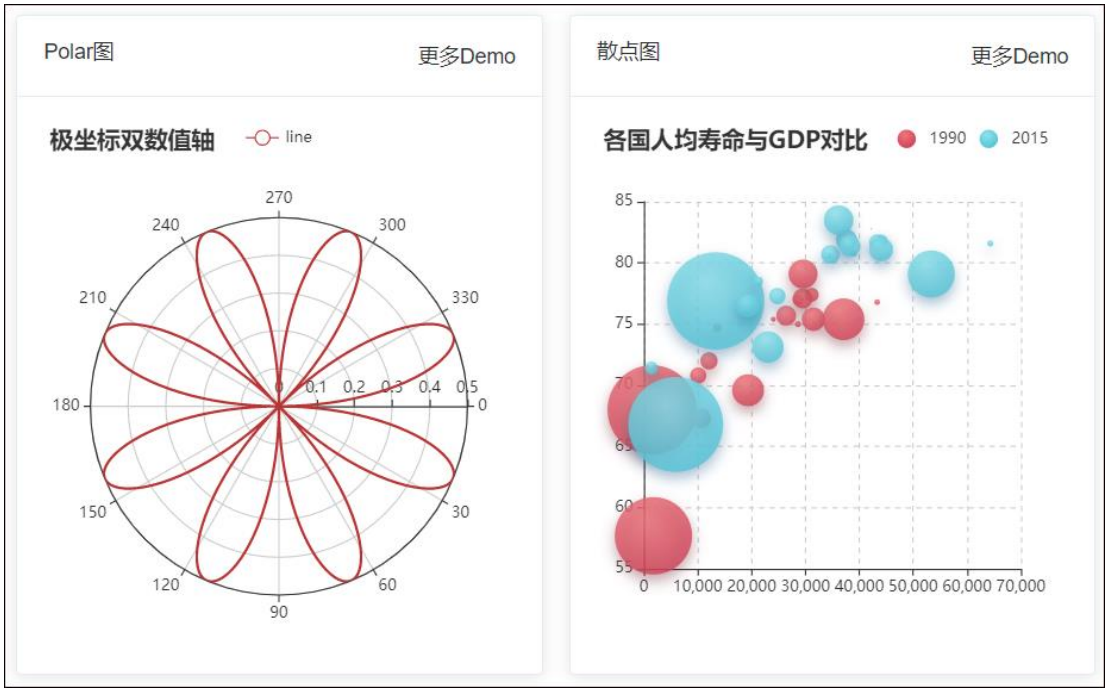
饼状图


饼图Demo
更多Demo
饼图程序调用高亮示例
直接访问
邮件营销
联盟广告
视频广告
搜索引擎
直接访问
邮件...
联盟...
视频广告
搜索...

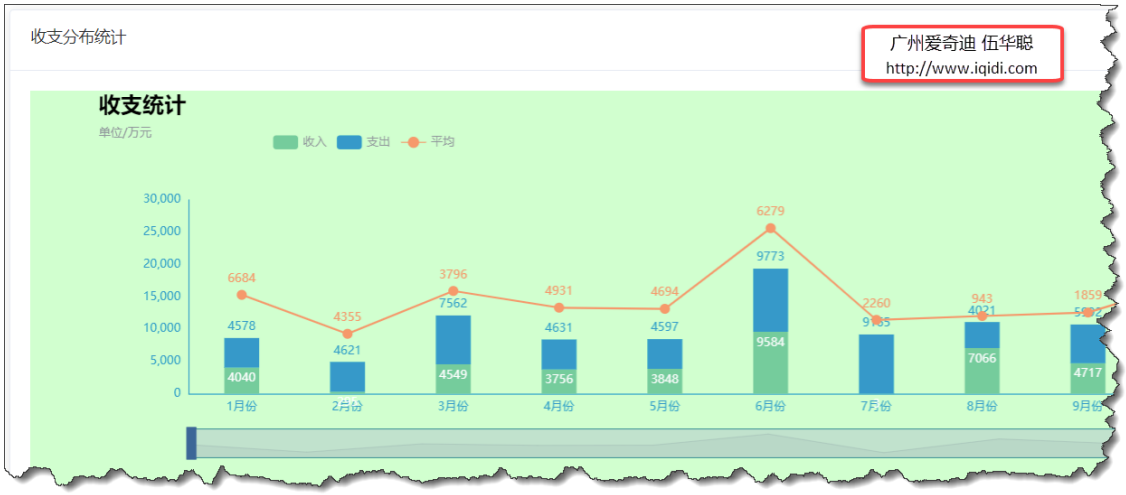
曲线图



其他类型，极坐标和散点图形



或者曲线和柱状图组合的图形



更多的案例可以参考官网的展示介绍：

https://echarts.apache.org/examples/zh/index.html

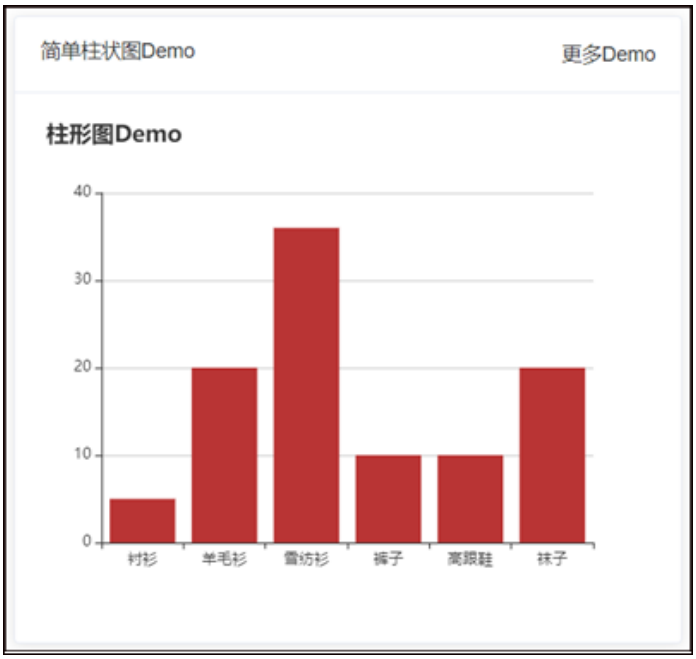
3.9.2. 各种图表的展示处理

对于我们需要的各种常规的柱状图、饼状图、折线图（曲线图）等，我下来介绍几个案例代码，其他的一般我们根据官方案例提供的 data 数据模型，构造对应的数据即可生成，就不再一一赘述。

另外，我们也可以参考 Vue-echarts 封装的处理：

demo:https://github.com/ecomfe/vue-echarts/tree/master/src/demo

对于柱状图，效果如下



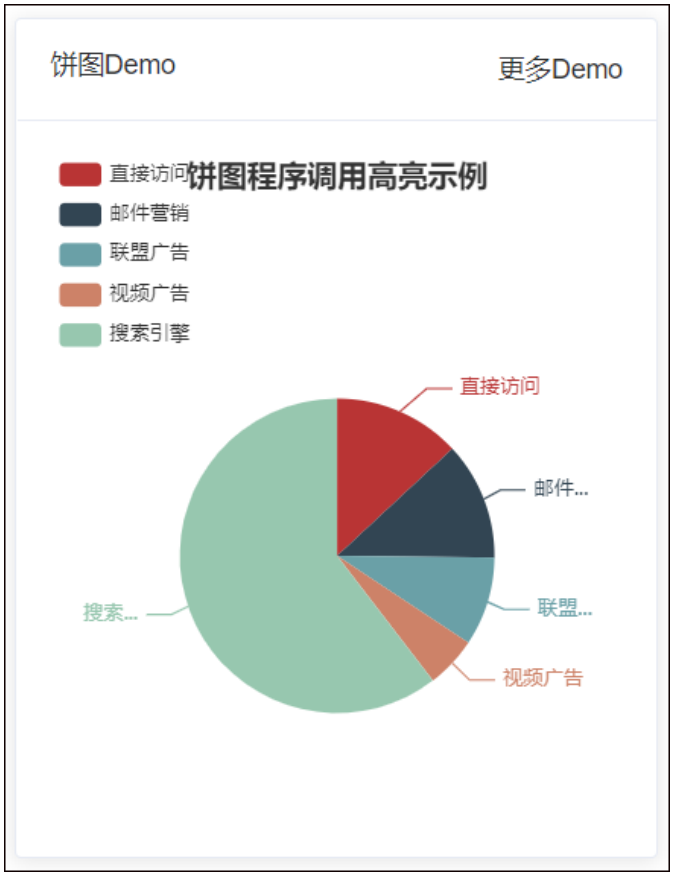
在 Vue 模块页面的 Template 里面，我们定义界面代码如下即可。

```
<v-chart
  ref="simplebar"
  :options="simplebar"
  autoresize />
```

然后在 data 里面为它准备好数据即可，如下代码所示。

```
data() {
    return {
      simplebar: {
        title: { text: '柱形图Demo' },
        tooltip: {},
        xAxis: {
          data: ['衬衫', '羊毛衫', '雪纺衫', '裤子', '高跟鞋', '袜子']
        },
        yAxis: {},
        series: [{
          name: '销量',
          type: 'bar',
          data: [5, 20, 36, 10, 10, 20]
        }]
      }
    }
  }
```

当然我们也可以把这些构造对应数据的逻辑放在单独的 JS 文件里面，然后导入即可。例如对于饼图，它的界面效果如下所示。



它的 vue 视图下，Template 里面的代码如下所示。

```
<v-chart
  ref="pie"
  :options="pie"
  autoresize />
```

一般对于图表的数据，由于处理代码可能不少，建议是独立放在一个 JS 文件里面，然后我们通过 import 导入即可使用。

```
// 导入报表数据
import getBar from './chartdata/bar'
import pie from './chartdata/pie'
import scatter from './chartdata/scatter'
import lineChart from './chartdata/lineChart'
import incomePay from './chartdata/incomePay'
```

然后在 data 里面引入对应的对象即可，如下所示。

```
<script>
import ECharts from 'vue-echarts' // 主图表对象
import 'echarts/lib/chart/line' // 曲线图表
import 'echarts/lib/chart/bar' // 柱状图
import 'echarts/lib/chart/pie' // 饼状图
import 'echarts/lib/component/tooltip' // 提示信息

// 导入报表数据
import getBar from './chartdata/bar'
import pie from './chartdata/pie'
import scatter from './chartdata/scatter'
import lineChart from './chartdata/lineChart'
import incomePay from './chartdata/incomePay'

export default {
  components: {
    'v-chart': ECharts
  },
   return {
      pie,
      scatter,,
      lineChart,
      incomePay,
      simplebar: {
        title: { text: '柱形图Demo' },
        tooltip: {},
        xAxis: {
          data: ['衬衫', '羊毛衫', '雪纺衫', '裤子', '高跟鞋', '袜子']
        },
        yAxis: {},
        series: [{
          name: '销量',
          type: 'bar',
          data: [5, 20, 36, 10, 10, 20]
        }]
```

其中 pie.js 里面放置的是处理饼图数据的逻辑，如下代码所示。

```
export default {
  title: {
    text: '饼图程序调用高亮示例',
    x: 'center'
  },
  tooltip: {
    trigger: 'item',
    formatter: '{a} <br/>{b} : {c} ({d}%)'
  },
  legend: {
    orient: 'vertical',
    left: 'left',
    data: ['直接访问', '邮件营销', '联盟广告', '视频广告', '搜索引擎']
  },
  series: [
    {
      name: '访问来源',
      type: 'pie',
      radius: '55%',
      center: ['50%', '60%'],
      data: [
        { value: 335, name: '直接访问' },
        { value: 310, name: '邮件营销' },
        { value: 234, name: '联盟广告' },
        { value: 135, name: '视频广告' },
        { value: 1548, name: '搜索引擎' }
      ],
      itemStyle: {
        emphasis: {
          shadowBlur: 10,
          shadowOffsetX: 0,
          shadowColor: 'rgba(0, 0, 0, 0.5)'
        }
      }
    }
```

在界面处理的时候，值得注意的时候，有时候 Vue 页面处理正常，但是图表就是没有出来，可能是因为高度或者宽度为 0 的原因，需要对对应的样式进行处理设置，以便能够正常显示出来。

如下是我 对图表的设置的样式处理，使得图表在一个卡片的位置能够显示正常。

```
<style scoped>
  .echarts { width: 100%; height: 400px;}

  .el-row {
    margin-bottom: 20px;
  }
  .el-col {
    border-radius: 4px;
    margin-bottom: 20px;
  }
</style>
```

最后几个界面组合一起的效果如下所示。

以上就是基于 vue-echarts 处理各种图表展示，其中常规的引入组件很容易的，主要就

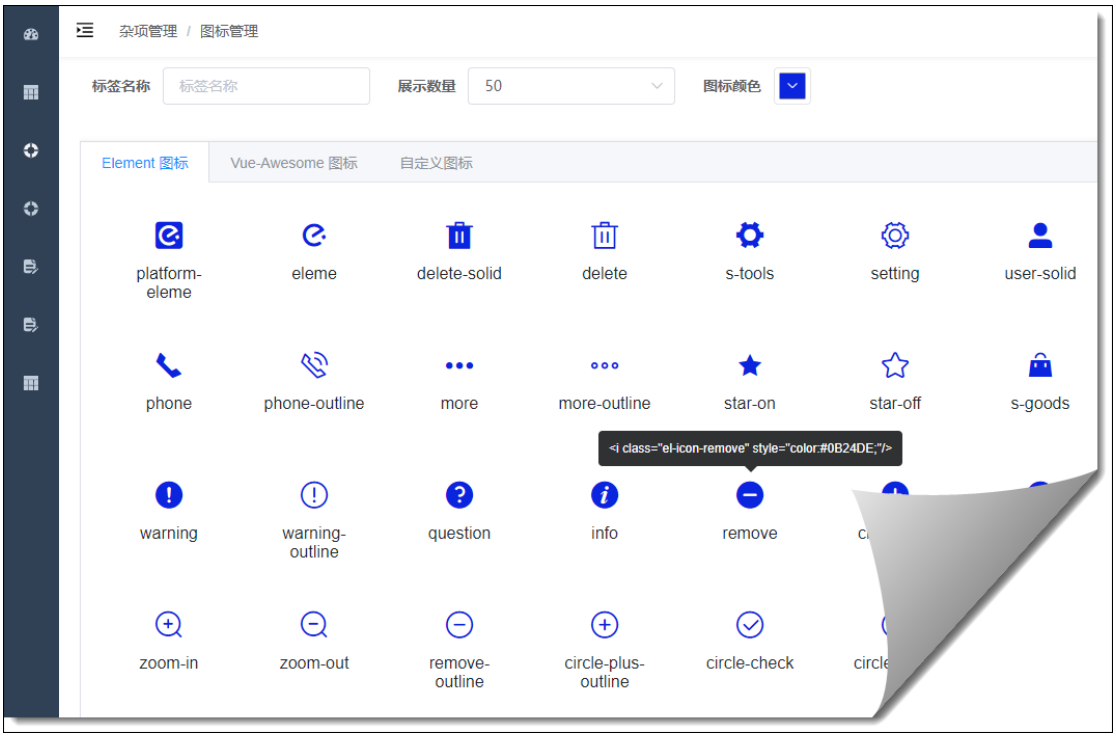
是需要根据对应的图表案例，参考数据组成的规则，从而根据我们实际情况构建对应的数据，赋值给对应的模型变量即可。

3.10. 图标的维护和使用

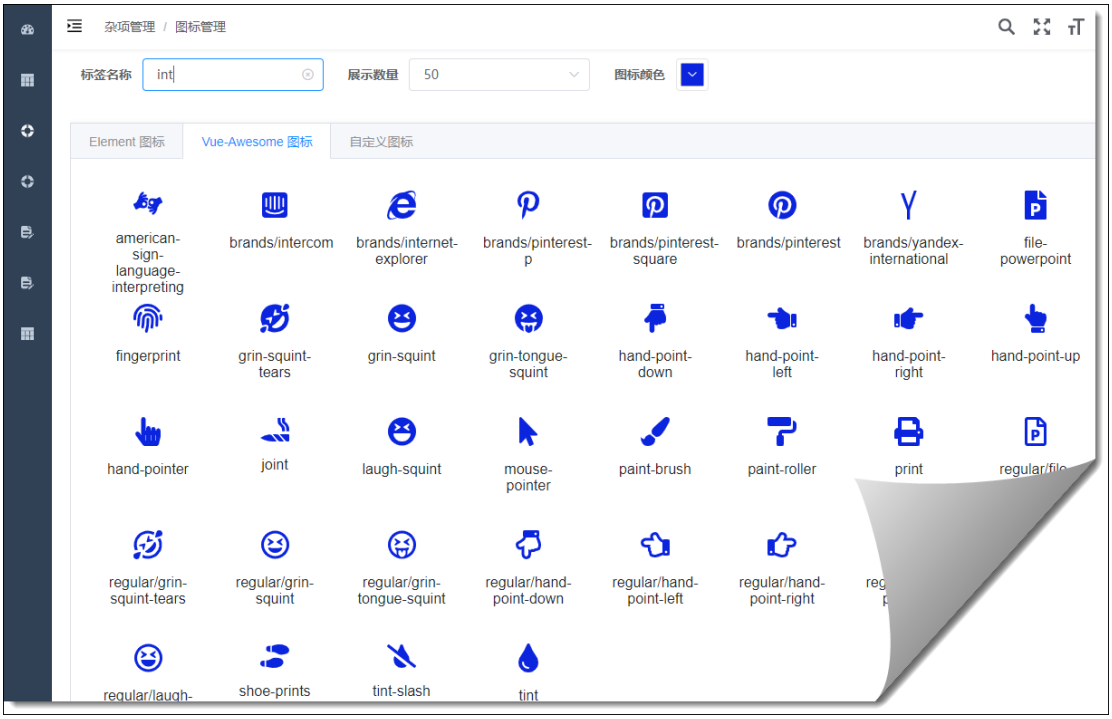
在 VUE+Element 前端应用中，图标是必不可少点缀界面的元素，因此整合一些常用的图标是非常必要的，还好 Element 界面组件里面提供了很多常见的图标，不过数量不是很多，应该是 300 个左右吧，因此考虑扩展更多图标，我引入了 vue-awesome 组件，它利用了 Font Awesome 的内置图标，实现了更多图标的整合，可以在项目中使用更多的图标元素了，另外在本随笔的图标管理中，提供了对图标名称进行搜索，并根据图标颜色样式生成对应图标的代码，非常方便使用。

Vue-Awesome 是基于 Vue.js 的 SVG 图标组件，内置图标来自 Font Awesome。本篇随笔先来上一个图标管理的界面效果，然后在逐一进行介绍 Element 内置图标和 Vue-Awesome 的图标吧。

Element 图标管理界面如下：



基于 Vue-Awesome 的图标管理如下所示。



其中查询提供了名称进行图标查询，以及限制一次性展示多少个，另外提供一个自定义颜色选项，从而可以改变图标的展示颜色。

3.10.1. Vue-Awesome 的使用介绍

Vue-Awesome 的 npm 的安装命令如下所示：

```
npm install vue-awesome
```

安装方式如下所示

```
import Vue from 'vue'

/* 在下面两种方式中任选一种 */

// 仅引入用到的图标以减小打包体积
import 'vue-awesome/icons/flag'

// 或者在不关心打包体积时一次引入全部图标
import 'vue-awesome/icons'

/* 任选一种注册组件的方式 */

import Icon from 'vue-awesome/components/Icon'

// 全局注册（在 `main .js` 文件中）
Vue.component('v-icon', Icon)

// 或局部注册（在组件文件中）
export default {
  components: {
    'v-icon': Icon
  }
}
```

界面使用代码如下所示，详细 Demo 可以参考：

https://justineo.github.io/vue-awesome/demo/ 了解。

```
<!-- 基础用法 -->
<v-icon name="beer"/>

<!-- 添加选项 -->
<v-icon name="sync" scale="2" spin/>
<v-icon name="comment" flip="horizontal"/>
<v-icon name="code-branch" label="Forked Repository"/>

<!-- 堆叠图标 -->
<v-icon label="No Photos">
  <v-icon name="camera"/>
  <v-icon name="ban" scale="2" class="alert"/>
</v-icon>
```

Vue-Awesome 图标提供了一些 prop 的属性设置，参考下面列表所示。

- name: string

图标名称。如果本组件没有用作图标堆栈的容器，那么这个字段是必须的。所有合法的值都对应于图标文件相对于 icons 目录的路径。请注意当你在 FontAwesome 官网查找到图标名词后，需要确认一下图标集的名称。比如，在 500px 这个图标的详情页会有一个 URL 参数 style=brands，故图标名称就是 brands/500px。

默认情况下，只能使用 FontAwesome 的免费图标，另外由于 solid 样式中的图标最多，我们将其设为了默认图标集，所以路径前缀可以省略。

当传入 null 时，整个组件将不会渲染。

- scale: number|string

用来调整图标尺寸，默认值为 1。

- spin: boolean

用来指定图标是否需要旋转。默认值为 false。（不能与 pulse 一同使用。）

- pulse: boolean

用来指定图标是否有脉冲旋转的效果。默认值为 false。（不能与 spin 一同使用。）

- inverse: boolean

为 true 时图标颜色将被设置为 #fff。默认值为 false。

- flip: 'vertical'|'horizontal'|'both'

用来指定图标是否需要翻转。

- label: string

当指定时会设置图标的 aria-label。

- title: string

为图标设置标题。

另外，我们也可以注册自定义图标，如下所示。

```
import Icon from 'vue-awesome/components/Icon'

Icon.register({
  baidu: {
    width: 23.868,
    height: 26,
    d: 'M3.613 13.701c2.827-.608 2.442-3.986 2.357-4.725-.138-1.139-1.477-3.
  }
```

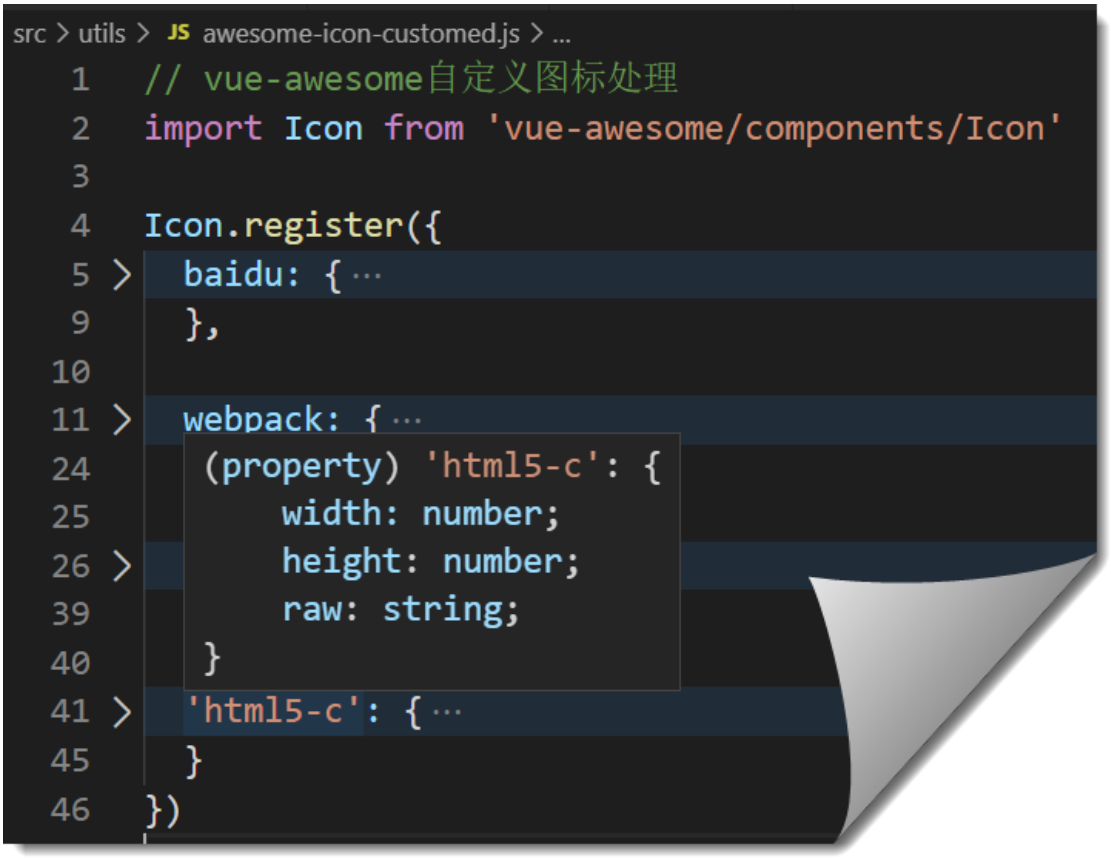
如果你的 SVG 文件有多个路径或多边形，以及/或者你想预定义一些样式，可以用如下方式进行注册，路径方式：

```
import Icon from 'vue-awesome/components/Icon'

Icon.register({
  webpack: {
    width: 1200,
    height: 1200,
    paths: [
      {
        style: 'fill:#8ED6FB',
        d: 'M1035.6 879.3l-418.1 236.5V931.6L878 788.3l157.6 91zm28.6-25.9V358.8l-153 88.3V765l153 8
      },
      {
        style: 'fill:#1C78C0',
        d: 'M580.8 889.7l-257-141.3v-280l257 148.4v272.9zm36.7 0l257-141.3v-280l-257 148.4v272.9zm-1
      }
    ]
```

多边形方式：

```
import Icon from 'vue-awesome/components/Icon'
Icon.register({
  vue: {
    width: 256,
    height: 221,
    polygons: [
      {
        style: 'fill:#41B883',
        points: '0,0 128,220.8 256,0 204.8,0 128,132.48 50.56,0 0,0'
      },
      {
        style: 'fill:#35495E',
        points: '50.56,0 128,133.12 204.8,0 157.44,0 128,51.2 97.92,0 50.56,0'
      }
    ]
  }
})
```

对于自定义的这些图标，我们可以把它们一起放在一个独立的 JS 文件里面进行定义，然后全局统一加入即可。



使用上和其他的图标没有差异，只是名称不同而已。

```
<div style="height:100px;padding:10px">
  <span>vue-Awesome 自定义图标:(在utils/awesome-icon-customed.js中引入)</span>
  <v-icon name="baidu" scale="2" :style="iconStyle" />
  <v-icon name="vue" scale="2" />
  <v-icon name="webpack" scale="2" spin />
  <v-icon name="html5-c" scale="2" spin />
</div>
```

3.10.2. 导入 Element 图标和 Vue-Awesome 图标

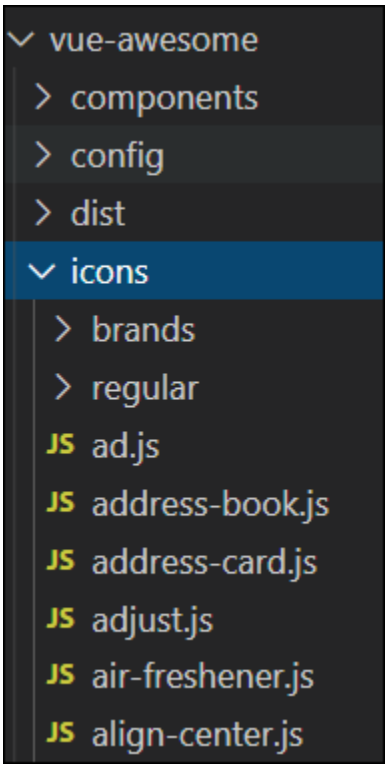
在我们进行页面管理的时候，我们需要提取 Element 图标和 Vue-Awesome 图标，以便能够进行界面的展示处理。

Element 图标，我们只需要在一个 JS 文件里面，包含它的名称进去一个列表里面即可，如下定义所示。

```
const elementIcons = [
  'platform-eleme', 'eleme', 'delete-solid', 'delete', ..........
]


export default elementIcons
```

在其中录入对应 Element 的图表名称，移除 el-icon-的前缀即可构成所需列表的每项内容。

而对于 Vue-Awesome 图标，我们安装对应的组件后，它的图标资源肯定也一定下载下来了，我们找到对应的 node_modules 目录下的文件就可以看到了，如下截图所示。



可以看到它的每个图标对应一个 js 文件，而且不同样式还有不同的目录的，我们只需要自动加入对应的文件名称即可。

定义一个 js 文件，如 vue-awesome-icon.js，用来获取对应 Awesome 图标名称，如下逻辑所示

```
// Vue-Awesome图标自动加入
const req = require.context('vue-awesome/icons', true, /\.js$/)
const requireAll = requireContext => requireContext.keys()

const re = /\.\/(.*)\.js/

const vueAwesomeIcons = requireAll(req).filter((key) => {
  return key.indexOf('index.js') < 0
}).map(i => {
  return i.match(re)[1]
})

export default vueAwesomeIcons
```

通过 require.context 的操作以及仅需 filter 的数组过滤，我们就可以获得对应的 Awesome

图标名称了。

在管理页面里面，我们需要引入 Element 和 Vue-Awesome 的图标管理文件，如下所示。

```
import elementIcons from './element-icons' // 引入 Element 图标
import vueAwesomeIcons from './vue-awesome-icons' // 引入 vue-awesome 图标
```

然后构造页面的 data 数据，如下所示，其中 searchForm 是用来承载查询条件的。

```
export default {
  name: 'Icons',
  data() {
    return {
      // 查询表单
      searchForm: {
        label: '',
        pagesize: 50,
        color: '#409EFF'
      },
      // 自定义svg图标集合
      svgIcons,
      // element图标集合
      elementIcons,
      // vueAwesome图标集合
      vueAwesomeIcons
    }
  },
```

然后增加几个 Computed 函数，用来处理数据的过滤查询等操作。

```
computed: {
    iconStyle: function() {
      return { color: this.searchForm.color }
    },
    elementIconsFiltered: function() {
      const that = this
      var list = that.elementIcons.filter(item => {
        return item.indexOf(that.searchForm.label) >= 0
      })
      if (that.searchForm.pagesize > 0) {
        return list.slice(0, that.searchForm.pagesize)
      } else {
        return list
      }
    },
    vueAwesomeIconsFiltered: function() {
      const that = this
      var list = that.vueAwesomeIcons.filter(item => {
        return item.indexOf(that.searchForm.label) >= 0
      })
      if (that.searchForm.pagesize > 0) {
        return list.slice(0, that.searchForm.pagesize)
      } else {
        return list
      }
    }
  },
```

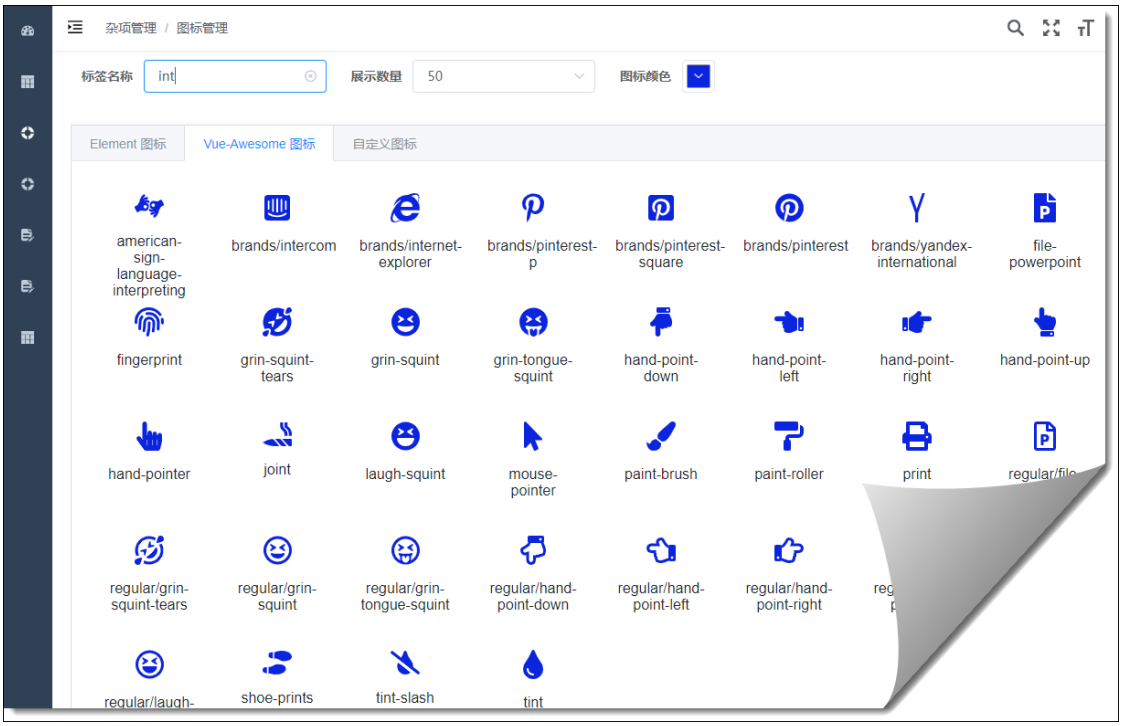
然后在 method 里面，对常规的图标进行生成处理即可。

```
methods: {
  generateElementIconCode(symbol) {
    return `<i class="el-icon-${symbol}" style="color:${this.searchForm.color};"/>`
  },
  generateAwesomeIconCode(symbol) {
    return `<v-icon name="${symbol}" style="color:${this.searchForm.color};"/>`
  },
  handleClipboard(text, event) {
    if (text) {
      clipboard(text, event)
    }
  }
}
```

对于界面的展示，我们以 Vue-awesome 图标展示为例介绍，如下所示。

```
<el-tab-pane label="Vue-Awesome 图标">
  <div
    v-for="item of vueAwesomeIconsFiltered"
    :key="item"
    @click="handleClipboard(generateAwesomeIconCode(item),$event)"
  >
    <el-tooltip placement="top">
      <div slot="content">{{ generateAwesomeIconCode(item) }}</div>
      <div class="icon-item">
        <v-icon :name="item" scale="2" :style="iconStyle" />
        <span>{{ item }}</span>
      </div>
    </el-tooltip>
  </div>
</el-tab-pane>
```

这样就可以实现对应图标的动态过滤和展示了。



3.11. 前端 API 接口的封装处理

在前面介绍了一个系统最初接触到的前端登录处理的实现，但往往对整个系统来说，一般会有很多业务对象，而每个业务对象的 API 接口又有很多，不过简单来说也就是常规的增删改查，以及一些自定义的接口，通用都比较有规律性。

而本身我们这个 VUE+Element 前端应用就是针对开发框架的业务对象，因此前端的业务对象接口也是比较统一的，那么可以考虑在前端中对后端 API 接口调用进行封装，引入 ES6 的方式进行前端 API 的抽象简化。这里主要针对这个方面，介绍前端 API 接口的封装处理，以便简化我们大量类似的业务接口的累赘代码实现。

一般前端调用，通过前端 API 类的封装，即可发起对后端 API 接口的调用，如系统登录 API 定义如下代码所示。

```
export function getInfo(id) {
  return request({
    url: '/abp/framework/User/Get',
    method: 'get',
    params: {
      id
```

```
    }
  })
}
```

按照常规的 API 类的处理，我们对应的业务类，就需要定义很多这样的函数，如之前介绍产品信息处理的 API 接口一样。

src > api > JS product.js > GetProductType

```
import request from '@/utils/request'

// 获取产品类型
export function GetTopBanners(params) {
  return request({
    url: '/iqidi/h5/GetTopBanners', //
    method: 'get',
    params
  })
}
// 获取产品类型
export function GetProductType(params) {
  return request({
    url: '/iqidi/h5/GetProductType', //
    method: 'get',
    params
  })
}
```

由于常规的增删改查，都是标准的 API 接口，那么如果我们按照每个 API 类都需要重复定义这些 API，显然不妥，太臃肿。

如果是常规的 JS，那么就以公布函数方式定义 API 接口，不过我们可以引入 ES6 的处理方式，在 JS 中引入类和继承的概念进行处理相同的 API 接口封装。

3.11.1. 基于 ES6 的 JS 业务类的封装

关于 ES6，大家可以有空了解一下《ES6 入门教程》，可以全面了解 ES6 很多语法和相关概念。

不过这里只需要了解一下 JS 里面关于类的定义和继承的处理关系即可。

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 class 关键字，可以定义类。

基本上，ES6 的 class 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 class 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的 class 改写，就是下面这样。

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

上面代码定义了一个“类”，可以看到里面有一个 constructor 方法，这就是构造方法，而 this 关键字则代表实例对象。也就是说，ES5 的构造函数 Point，对应 ES6 的 Point 类的构造方法。

Point 类除了构造方法，还定义了一个 toString 方法。注意，定义“类”的方法的时候，前面不需要加上 function 这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。

Class 可以通过 extends 关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

```
class Point {
```

```
}


class ColorPoint extends Point {
}
```

上面代码定义了一个 ColorPoint 类，该类通过 extends 关键字，继承了 Point 类的所有属性和方法。

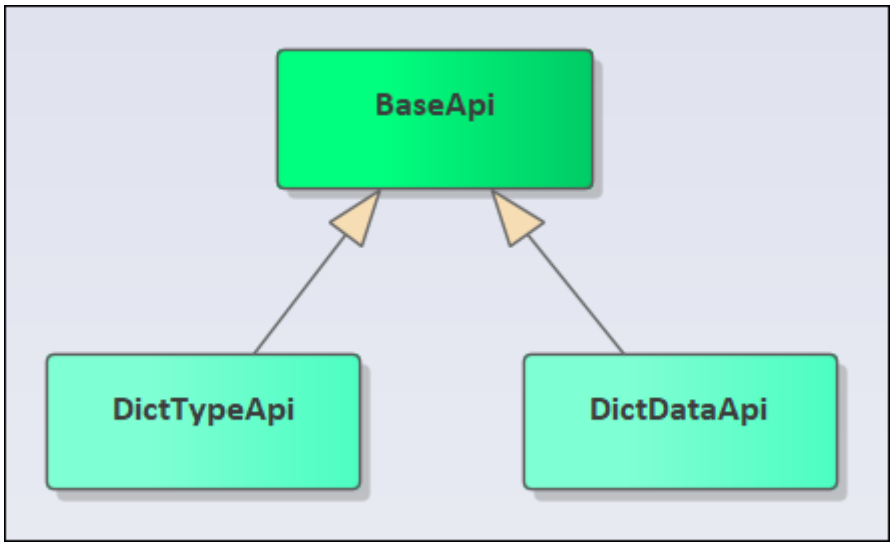
有了这些知识准备，那么我们来定义一个 API 接口的封装类，如下 base-api.js 代码所示。

```
// 定义框架里面常用的API接口：Get/GetAll/Create/Update/Delete/Count等
export default class BaseApi {
  constructor(baseurl) {
    this.baseurl = baseurl
  }

  // 获取指定的单个记录
  Get(data) {
    return request({
      url: this.baseurl + 'Get',
      method: 'get',
      params: data
    })
  }

  // 根据条件获取所有记录
  GetAll(data) {
    return request({
      url: this.baseurl + 'GetAll',
      method: 'get',
      params: data
    })
  }

  // 创建记录
  Create(data) {
    return request({
      url: this.baseurl + 'Create',
      method: 'post',
      data: data
    })
  }
```

```
  // 更新记录
  Update(data) {
    return request({
      url: this.baseurl + 'Update',
      method: 'put',
      data: data
    })
  }

  // 删除指定数据
  Delete(data) {
    return request({
      url: this.baseurl + 'Delete',
      method: 'delete',
      params: data
    })
  }

  // 获取条件记录数量
  Count(data) {
    return request({
      url: this.baseurl + 'Count',
      method: 'post',
      data: data
    })
  }
}
```

以上我们定义了很多常规的 WebAPI 后端接口的封装处理，其中我们调用的地址通过组合的方式处理，而具体的地址则交由子类（业务对象 API）进行赋值处理。

加入我们定义子类有 DIctType、DictData 等业务类，那么这些类继承 BaseApi，就会具有相关的接口了，如下所示继承关系。



例如，我们对于 DictDataApi 的 JS 类定义如下所示。

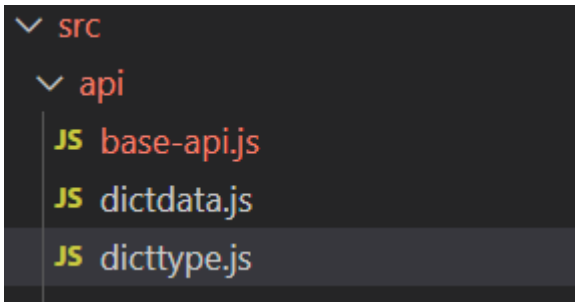
```
import request from '@/utils/request'

import BaseApi from '@/api/base-api'
// 业务类自定义接口实现，通用的接口已经在BaseApi中定义
class Api extends BaseApi {
  // 说明：…
  // 根据字典类型ID获取所有该类型的字典列表集合(使用缓存)
  GetDictByTypeIDCached(data) {…
  }
  // 根据字典类型ID获取所有该类型的字典列表集合(Key为名称，Value为值)
  GetDictByTypeID(data) {…
  }

  // 根据字典类型名称获取所有该类型的字典列表集合(Key为名称，Value为值)
  GetDictByDictType(data) {…
  }

  // 根据字典类型获取对应的CListItem集合(包括value，text属性)
  GetListItemByDictType(data) {…
  }
  //根据字典类型获取对应的TreeNodeItem集合(包括id，label属性)
  GetTreeItemByDictType(data) {…
  }
  // 根据字典类型ID删除对应下面的字典数据
  DeleteByTypeID(data) {…
  }

  // 批量插入字典数据
  BatchAdd(data) {…
  }
}

// 构造Api实例，并传递业务类接口地址
export default new Api('/abp/framework/dictdata/')
```

通过一行代码 export default new Api('/abp/framework/dictdata/') 就可以构造一个子类实例供使用了。

对于 DictTypeApi 来说，处理方式也是类似，继承自基类，并增加一些自己的接口实现即可。

```
import BaseApi from '@/api/base-api'
// 业务类自定义接口实现，通用的接口已经在BaseApi中定义
class Api extends BaseApi {
  // 说明：
  // params 是get请求 会把参数放到url 中;
  // data 是post请求会把参数添加到请求体(body)中
  // axios get/delete 用params, put/post等用data

  // 自定义接口
  // 获取所有类型
  GetAllType(data) {
    return request({
      url: this.baseurl + 'GetAllType',
      method: 'get',
      params: data
    })
  }

  // 获取指定类型的树结构
  GetTree(data) {
    return request({
      url: this.baseurl + 'GetTree',
      method: 'get',
      params: data
    })
  }
}

// 构造Api实例，并传递业务类接口地址
export default new Api('/abp/framework/dicttype/')
```

这些 API 类的文件视图如下所示。



有了这些准备，我们就可以在视图页面类中导入这些定义，并使用 JS 类了。

```
// 业务 API 对象
import dicttype from '@/api/dicttype'
import dictdata from '@/api/dictdata'
```

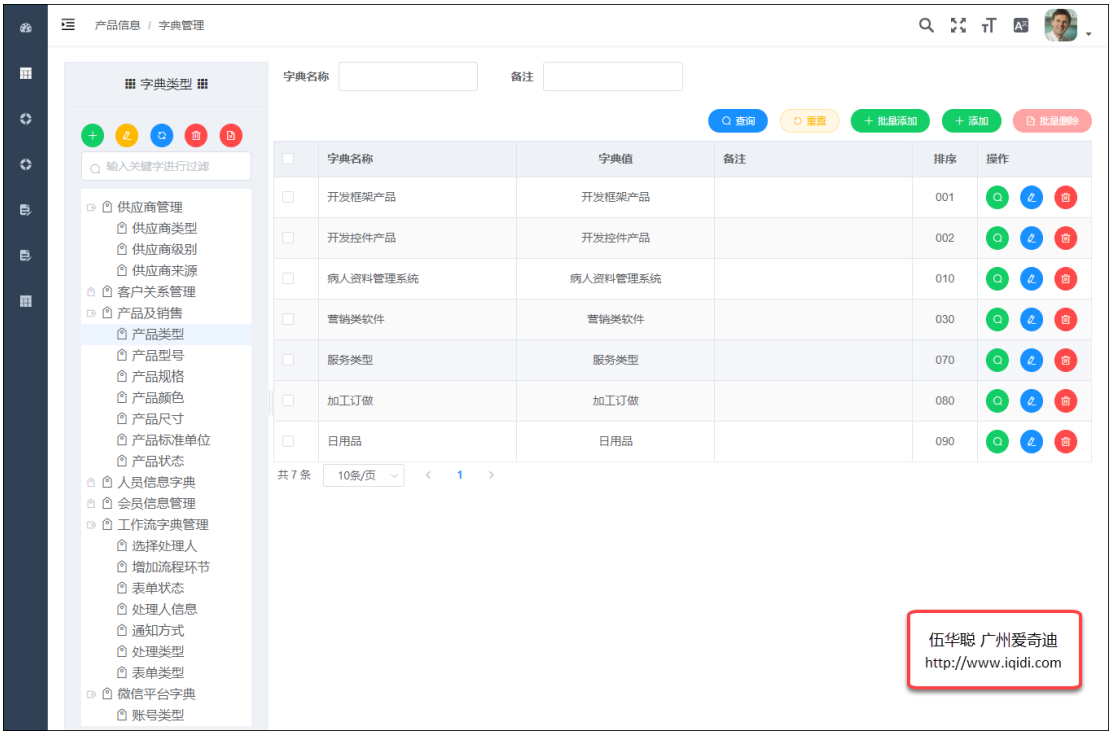
加入我们要在视图页面中查询结果，直接就可以通过使用 dictdata 或者 dicttype 对象来实现对应的 API 调用，如下代码所示。

```
getlist() {
  // 构造常规的分页查询条件
  var param = {
    SkipCount: (this.pageinfo.pageindex - 1) * this.pageinfo.pagesize,
    MaxResultCount: this.pageinfo.pagesize,
    // 过滤条件
    Name: this.searchForm.name,
    Remark: this.searchForm.remark,
    DictType_ID: this.searchForm.dictType_ID
  };

  // 获取产品列表，绑定到模型上，并修改分页数量
  this.listLoading = true
  dictdata.GetAll(param).then(data => {
    this.list = data.result.items
    this.pageinfo.total = data.result.totalCount
    this.listLoading = false
  })
}
```

或者如下代码所示。

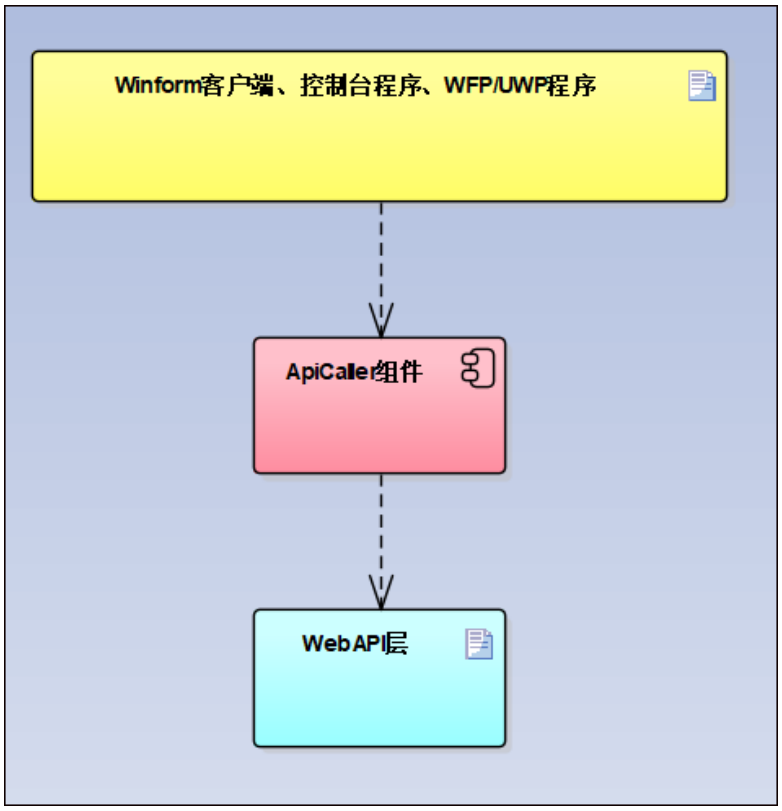
```
// 删除指定字典类型
deleteDictType() {
  if (!this.searchForm.dictType_ID || typeof (this.searchForm.dictType_ID) === 'undefined') {
    return;
  }

  this.$confirm('您确认删除选定类型吗?', '操作提示',
    {
      type: 'warning' // success, error, info, warning
      // confirmButtonText: '确定',
      // cancelButtonText: '取消'
    }
  ).then(() => {
    var param = { id: this.searchForm.dictType_ID }
    dicttype.Delete(param).then(data => {
      if (data.success) {
        // 提示信息
        this.$message({
          type: 'success',
          message: '删除成功!'
        })
        // 刷新数据
        this.getTree();
      }
    })
  })
}
```

最后我们来看看使用这些接口处理，对字典管理界面的实现。

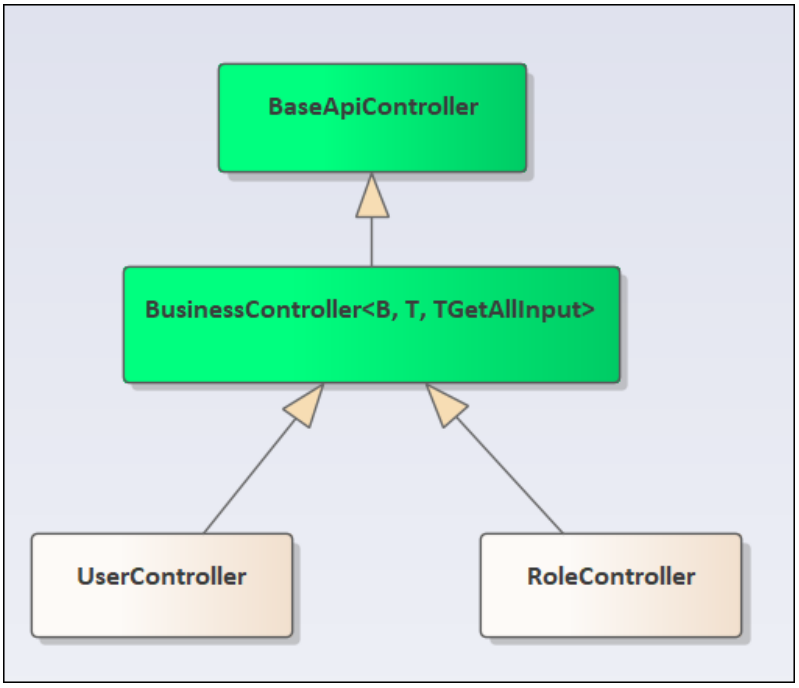


3.11.2. Web API 接口和前端对接处理

针对 Web API 接口调用的封装，为了适应客户端快速调用的目的，这个封装作为一个独立的封装层，以方便各个模块之间进行共同调用。



在后端 Web API 的框架中，为了更好的进行相关方法的封装处理，我们把一些常规的接口处理放在 BaseApiController 里面，而把基于业务表的操作接口放在 BusinessController 里面定义，如下所示。



在 BaseApiController 里面，我们使用了结果封装和异常处理的过滤器统一处理，以便简

化代码，如下控制器类定义。

其中 ExceptionHandling 和 WrapResult 的过滤器处理，可以参考我的随笔《利用过滤器 Filter 和特性 Attribute 实现对 Web API 返回结果的封装和统一异常处理》进行详细了解。

而业务类的接口通用封装，则放在了 BusinessController 控制器里面，其中使用了泛型定义，包括实体类，业务操作类，分页条件类等内容作为约束参数，如下所示。

```
/// <summary>
/// 本控制器基类专门为访问数据业务对象而设的基类
/// </summary>
/// <typeparam name="B">业务对象类型</typeparam>
/// <typeparam name="T">实体类类型</typeparam>
[ApiAuthorize]
public class BusinessController<B, T, TGetAllInput> : BaseApiController
    where B : class
    where TGetAllInput : IPagedAndSortedResultRequest
    where T : BaseEntity, new()
```

其中 IPagedAndSortedResultRequest 接口，是借鉴 ABP 框架中对于分页部分的处理，因此分页函数需要实现这个接口，这个接口包含了请求的数量，偏移量， 以及排序等属性定义的。而 BusinessController 的分页查询处理函数 GetAll 定义如下所示。

```
/// <summary>
/// 分页获取记录
/// </summary>
/// <param name="input"></param>
/// <returns></returns>
[HttpGet]
public virtual PagedResultDto<T> GetAll([FromUri] TGetAllInput input)
{
    var condition = GetCondition(input);
    var list = GetPagedData(condition, input);
    return list;
}
```

其中 GetCondition 函数是给子类进行重写，以便处理不同的条件查询的。我们以 UserController 控制器为例进行说明。

```
        /// <summary>
        /// 用户信息的业务控制器
        /// </summary>
        public class UserController : BusinessController<User, UserInfo, UserPagedDto>
```

其中传入的 User 是 BLL 业务层类，用来操作数据库；UserInfo 是实体类，用来传递记录信息；UserPagedDto 则是分页查询条件类。

```
    /// <summary>
    /// 用户信息的业务查询类
    /// </summary>
    public class UserPagedDto : PagedAndSortedInputDto, IPagedAndSortedResultRequest
    {
        /// <summary>
        /// 默认构造函数
        /// </summary>
        public UserPagedDto() : base() { }

        /// <summary>
        /// 参数化构造函数
        /// </summary>
        /// <param name="skipCount">跳过的数量</param>
        /// <param name="resultCount">最大结果集数量</param>
        public UserPagedDto(int skipCount, int resultCount) : base(skipCount, resultCount)
        {
        }

        /// <summary>
        /// 使用分页信息进行初始化SkipCount 和 MaxResultCount
        /// </summary>
        /// <param name="pagerInfo">分页信息</param>
        public UserPagedDto(PagerInfo pagerInfo) : base(pagerInfo)
        {
```

它的基类属性包括了 MaxResultCount，SkipCount，Sorting 等分页排序所需的信息。

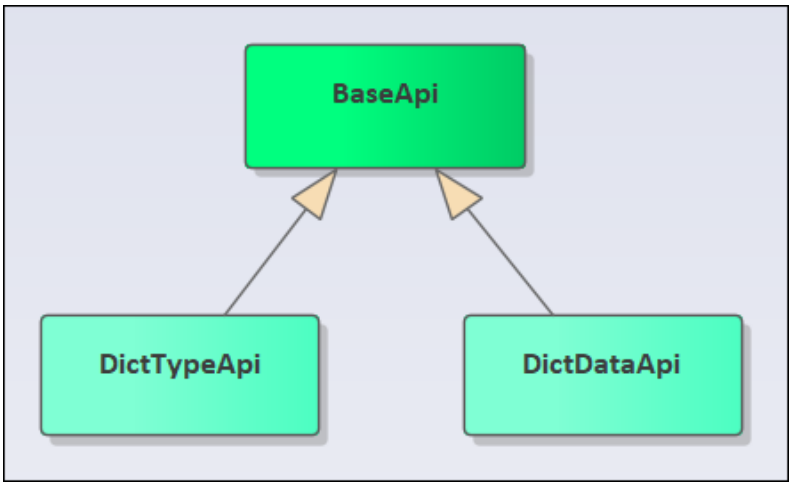
另外还包含了对条件查询的属性信息，如果是数值的，布尔类型的，则是可空类型，日期则有起始条件的范围属性等等，也可以根据自己需要定义更多属性用户过滤条件。

```
/// <summary>
/// 获取查询条件并转换为SQL
/// </summary>
/// <param name="input">查询条件</param>
protected override string GetCondition(UserPagedDto input)
{
    //根据条件，构建SQL条件语句
    SearchCondition condition = new SearchCondition();
    if (!input.Role_ID.HasValue)
    {
        condition.AddCondition("ID", input.ID, SqlOperator.Equal)
            .AddCondition("IdentityCard", input.IdentityCard, SqlOperator.Equal)
            .AddCondition("Name", input.Name, SqlOperator.Like)
            .AddCondition("Note", input.Note, SqlOperator.Like)
            .AddCondition("Email", input.Email, SqlOperator.Like)
            .AddCondition("MobilePhone", input.MobilePhone, SqlOperator.Like)
            .AddCondition("Address", input.Address, SqlOperator.Like)
            .AddCondition("HandNo", input.HandNo, SqlOperator.Like)
            .AddCondition("HomePhone", input.HomePhone, SqlOperator.Like)
            .AddCondition("Nickname", input.Nickname, SqlOperator.Like)
            .AddCondition("OfficePhone", input.OfficePhone, SqlOperator.Like)
            .AddCondition("OpenId", input.OpenId, SqlOperator.Like)
            .AddCondition("Password", input.Password, SqlOperator.Like)
            .AddCondition("PID", input.PID, SqlOperator.Like)
            .AddCondition("QQ", input.QQ, SqlOperator.Equal)
```

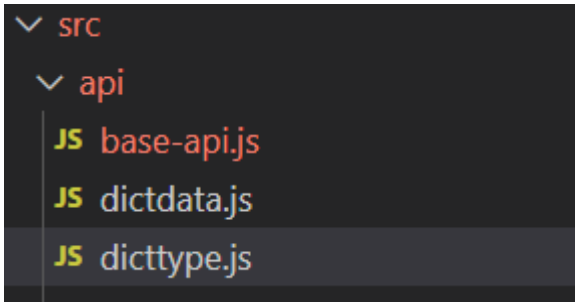
在基于 VUE+Element 前端应用中，上图的 ApiCaller 组件，其实就是我们封装的 API 的 JS 类，同时也有相关的继承封装处理，类似 C#中类的继承关系了。

在 JS 里面引入了 ES6 的语法，JS 函数就可以使用类的方式来实现各种基础接口的封装和子类的继承关系了。

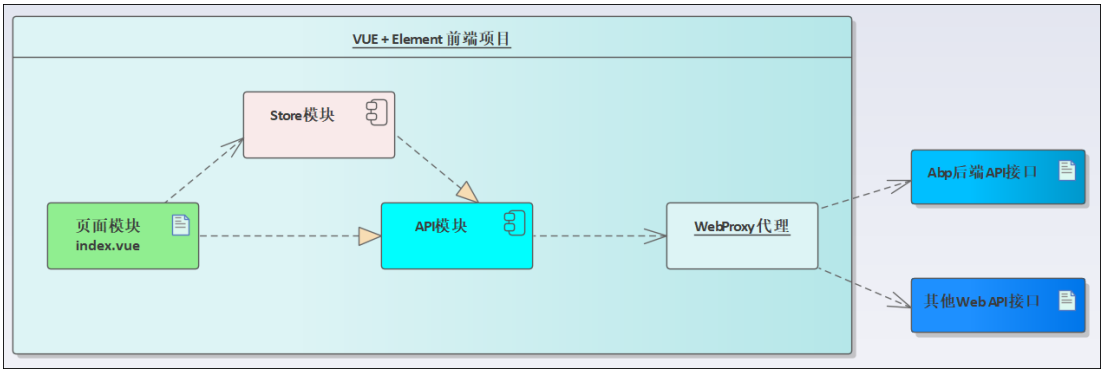
例如我们先定义好 JS 中常规的 API 封装接口 BaseApi 类，然后定义子类有 DIctType、DictData 等业务类，那么这些类继承 BaseApi，就会具有相关的接口了，如下所示继承关系。



这些 API 类的文件视图如下所示。



有了这些业务类的准备，那么我们和后端的 API 接口对接，就很容易了，如下图所示。



一般来说，我们页面模块可能会涉及到 Store 模块，用来存储对应的状态信息，也可能是直接访问 API 模块，实现数据的调用并展示。在页面开发过程中，多数情况下，不需要 Store 模块进行交互，一般只需要存储对应页面数据为全局数据状态的情况下，才可能启用 Store 模块的处理。

通过 WebProxy 代理的处理，我们可以很容易在前端中实现跨域的处理，不同的路径调用不同的域名地址 API 都可以，最终转换为本地的 API 调用，这就是跨域的处理操作。

3.11.3. 前端界面处理

有了上面的一些知识准备，相信对 Vue+Element 的前端对接有了一个大概的认识了。

那么我们前端界面也需要根据这些参数来构造查询界面，我们可以通过部分条件进行处理即可，其中 MaxResultCount 和 SkipCount 是用于分页定位的参数。

前端界面代码如下所示。

```
<el-form ref="searchForm" :model="searchForm" label-width="80px">
  <el-row>
    <el-col :span="6">
      <el-form-item label="字典名称" prop="Name">
        <el-input v-model="searchForm.Name" />
      </el-form-item>
    </el-col>
    <el-col :span="6">
      <el-form-item label="备注" prop="Remark">
        <el-input v-model="searchForm.Remark" />
      </el-form-item>
    </el-col>
  </el-row>
</el-form>
```

主要就是构造一个条件查询的表单如下所示。

其中定义了一个 searchForm 的属性，那么我们可以了解下它的定义。

```
searchForm: { // 查询表单
  name: '',
  remark: '',
  dictType_ID: '',
  dictCode: ''
},
```

然后我们在查询处理的函数 getlist 里面，就可以根据这些条件，以及分页参数进行数据的请求查询了，如下代码所示。

```
getlist() { // 列表数据获取
  var param = { // 构造常规的分页查询条件
    SkipCount: (this.pageinfo.pageindex - 1) * this.pageinfo.pagesize,
    MaxResultCount: this.pageinfo.pagesize,
    // 过滤条件
    Name: this.searchForm.name,
    Remark: this.searchForm.remark,
    DictType_ID: this.searchForm.dictType_ID
  };

  // 获取产品列表，绑定到模型上，并修改分页数量
  this.listLoading = true
  dictdata.GetAll(param).then(data => {
    this.list = data.result.items
    this.pageinfo.total = data.result.totalCount
    this.listLoading = false
  })
},
```

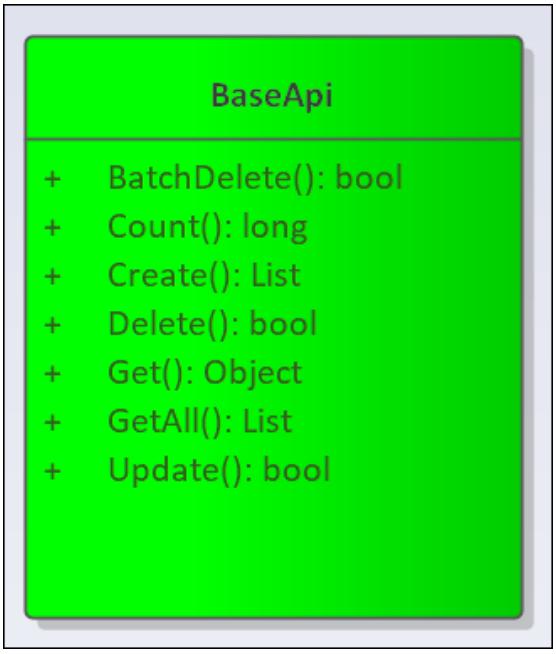
其中的 dictdata.GetAll 就是调用 API 模块里面的函数进行处理，是我们在视图页面类中导入这些 API 类的定义的。

```
// 业务 API 对象
import dicttype from '@/api/dicttype'
import dictdata from '@/api/dictdata'
```

我们对于 DictDataApi 的 JS 类定义如下所示。

```
import BaseApi from '@/api/base-api'
// 业务类自定义接口实现，通用的接口已经在BaseApi中定义
class Api extends BaseApi {
  // 说明：…
  GetDictByTypeIDCached(data) {…
  }
  // 根据字典类型ID获取所有该类型的字典列表集合(Key为名称，Value为值)
  GetDictByTypeID(data) {…
  }

  // 根据字典类型名称获取所有该类型的字典列表集合(Key为名称，Value为值)
  GetDictByDictType(data) {…
  }

  // 根据字典类型ID删除对应下面的字典数据
  DeleteByTypeID(data) {…
  }
}

// 构造Api实例，并传递业务类接口地址
export default new Api('/abp/services/app/dictdata/')
```

其中 JS 类的 BaseApi 具有常规的增删改查接口，如下所示。



其中 list 是列表的数据，pageinfo 是分页信息，那么我们在前端的列表展示界面代码如

下所示。

```
<el-table
  v-loading="listLoading"
:data="list"
  border
  fit
  stripe
  highlight-current-row
  :header-cell-style="{background:'#eef1f6',color:'#606266'}"
  @selection-change="selectionChange"
  @row-dblclick="rowDbclick"
>
  <el-table-column type="selection" width="55" />
  <el-table-column label="字典名称">
    <template slot-scope="scope">
      {{ scope.row.name }}
    </template>
  </el-table-column>
  <el-table-column align="center" label="字典值">
    <template slot-scope="scope">
      {{ scope.row.value }}
    </template>
  </el-table-column>
  <el-table-column label="备注">
    <template slot-scope="scope">
      {{ scope.row.remark }}
    </template>
  </el-table-column>
```

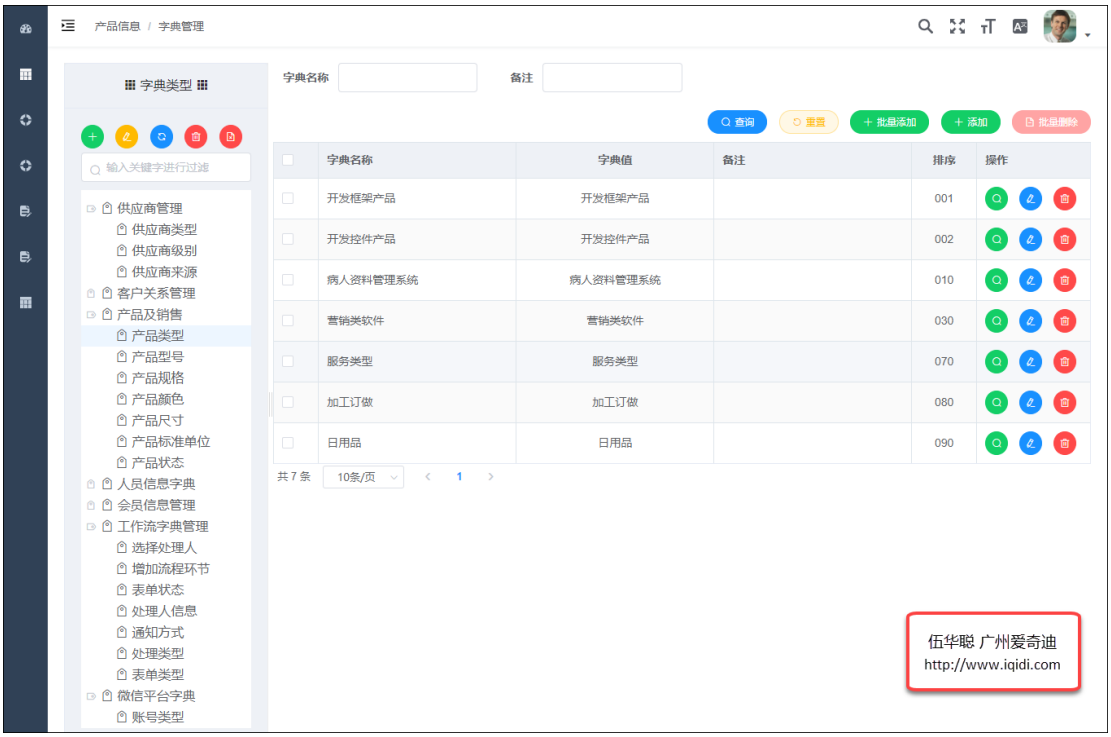
```
  <el-table-column align="center" label="排序" width="80">
    <template slot-scope="scope">
      {{ scope.row.seq }}
    </template>
  </el-table-column>
  <el-table-column label="操作" width="140">
    <template scope="scope">
      <el-row>
        <el-tooltip effect="light" content="查看" placement="top-start">
          <el-button icon="el-icon-search" type="success" circle size="mini" @click="showView(scope.row.id)" />
        </el-tooltip>
        <el-tooltip effect="light" content="编辑" placement="top-start">
          <el-button icon="el-icon-edit" type="primary" circle size="mini" @click="showEdit(scope.row.id)" />
        </el-tooltip>
        <el-tooltip effect="light" content="删除" placement="top-start">
          <el-button icon="el-icon-delete" type="danger" circle size="mini" @click="showDelete(scope.row.id)" />
        </el-tooltip>
      </el-row>
    </template>
  </el-table-column>
</el-table>
<div class="block" style="height:70px;">
  <el-pagination
    :current-page="pageinfo.pageindex"
    :page-size="pageinfo.pagesize"
    :total="pageinfo.total"
    :page-sizes="[10,20,30,40]"
    layout="total, sizes, prev, pager, next"
    @size-change="sizeChange"
    @current-change="currentChange"
  />
</div>
```

这样就可以简单实现列表的查询和展示了。

当然我们一般情况下，可能有一些列表用来进行数据过滤处理的，如这里的字典类型，可以通过树列表的进行展示，从而可以友好的管理不同类型的字典数据，如下是整合了树形

列表的查询处理过程，界面相对复杂一些，不过原理差不多，都是根据条件进行 API 数据的请求，然后展示在列表中即可。

完整的对字典管理界面的实现。



其中包括对字典大类的维护，以及对应字典大类的数据列表数据的添加处理。

3.11.4. 前端附件管理的实现

上传图片或者其他附件信息，我们可以用 Element 组件里面的 el-Upload 组件操作，这个控件基本上能够满足大多数的应用了，参考地址：https://element.eleme.cn/#/zh-CN/component/upload。

如下是其中的界面使用代码：

```
<el-form-item label="封面图片">
  <el-upload
    ref="upload"
    action="/abp/services/app/FileUpload/PostUpload"
    list-type="picture-card"
    :on-preview="handlePictureCardPreview"
    :on-remove="handleRemove"
    :on-success="onSuccess"
    :on-error="onError"
    accept="image/jpeg,image/gif,image/png,image/bmp"
    :headers="myHeaders"
    :file-list="editForm.fileList"
  >
    <i class="el-icon-plus" />
  </el-upload>
  <el-dialog :visible.sync="dialogVisible">
    <img width="100%" :src="dialogImageUrl" alt="">
  </el-dialog>
</el-form-item>
```

只是我们一般为了简化处理，往往使用一些基于 El-Upload 组件之上封装好的组件，更显方便而已。一般的图片和附件上传界面如下所示。

如我往往喜欢喜欢使用稍加封装，代码量更少的一些第三方组件，如：

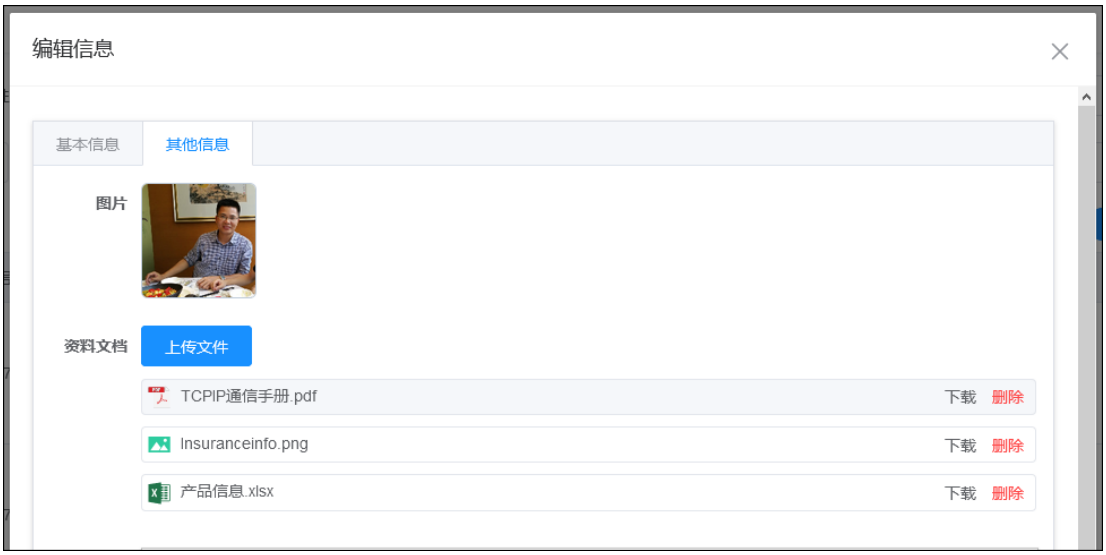
dream2023/vue-ele-upload-image

dream2023/vue-ele-upload-file

等这位仁兄的二次封装的组件来做附件管理，可以更加简洁一些。如下是使用的代码。

```
<el-row>
  <el-col :span="24">
    <el-form-item label="图片" prop="picture">
      <ele-upload-image
        v-model="editForm.picture"
        :is-show-tip="false"
        :size="100"
        action="/abp/services/app/FileUpload/PostUpload"
        title="封面图片"
        :headers="myHeaders"
        :data="{guid:editForm.id, folder:'用户图片'}"
        :crop="true"
      />
    </el-form-item>
  </el-col>
  <el-col :span="24">
    <el-form-item label="资料文档" prop="attachGUID">
      <ele-upload-file
        v-model="editForm.attachGUID_files"
        :response-fn="handleAttachResponse"
        action="/abp/services/app/FileUpload/PostUpload"
        :headers="myHeaders"
        :data="{guid:editForm.attachGUID, folder:'用户图片'}"
        :before-remove="beforeRemoveAttach"
      />
    </el-form-item>
  </el-col>
```

编辑界面下，附件上传界面，可以加载已有的记录展示，如下所示。



而附件列表我们通过调用 WebAPI 后端的 API 即可获取，然后进行绑定即可。

```
// 获取附件文件列表
param = { guid: this.editForm.attachGUID }
fileupload.GetByAttachGUID(param).then(data => {
  if (data.result) {
    this.editForm.attachGUID_files = [...data.result]
  }
})
```

附件上传的操作，我们一般需要通过设置 Header 方式，来传递用户的身份信息，如下 data 部分的代码

```
myHeaders: { Authorization: 'Bearer ' + getToken() }, // 用于上传文件的身份认证
```

而其中的界面组件中的 data，是指定额外上传的文件附带信息，方便我们区分或者归类文件的。

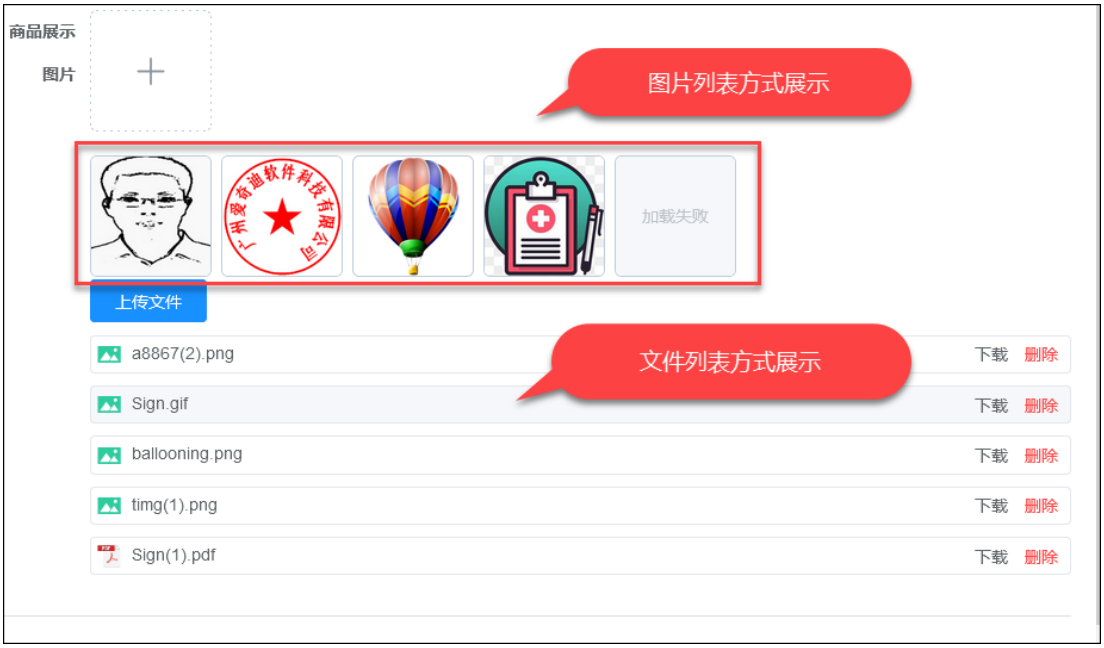
```
<el-form-item label="资料文档" prop="attachGUID">
  <ele-upload-file
    v-model="editForm.attachGUID_files"
    :response-fn="handleAttachResponse"
    action="/abp/services/app/FileUpload/PostUpload"
    :headers="myHeaders"
    :data="{guid:editForm.attachGUID, folder:'用户图片'}"
    :before-remove="beforeRemoveAttach"
  />
```

附件列表，如果需要删除的，那么我们提示确认后，需要调用 WebAPI 后端 API 进行删除

文件。

```
beforeRemoveAttach(file, fileList) { // 移除附件图片
  // 服务端删除文件
  var param = { guid: this.editForm.attachGUID, index: fileList.indexOf(file) }
  fileupload.RemoveAttach(param).then(data => {
    console.log(data.result)
  })
},
```

另外，如果确认附件是全部图片，我们也可以用图片列表控件的方式展示图片信息，如下所示。

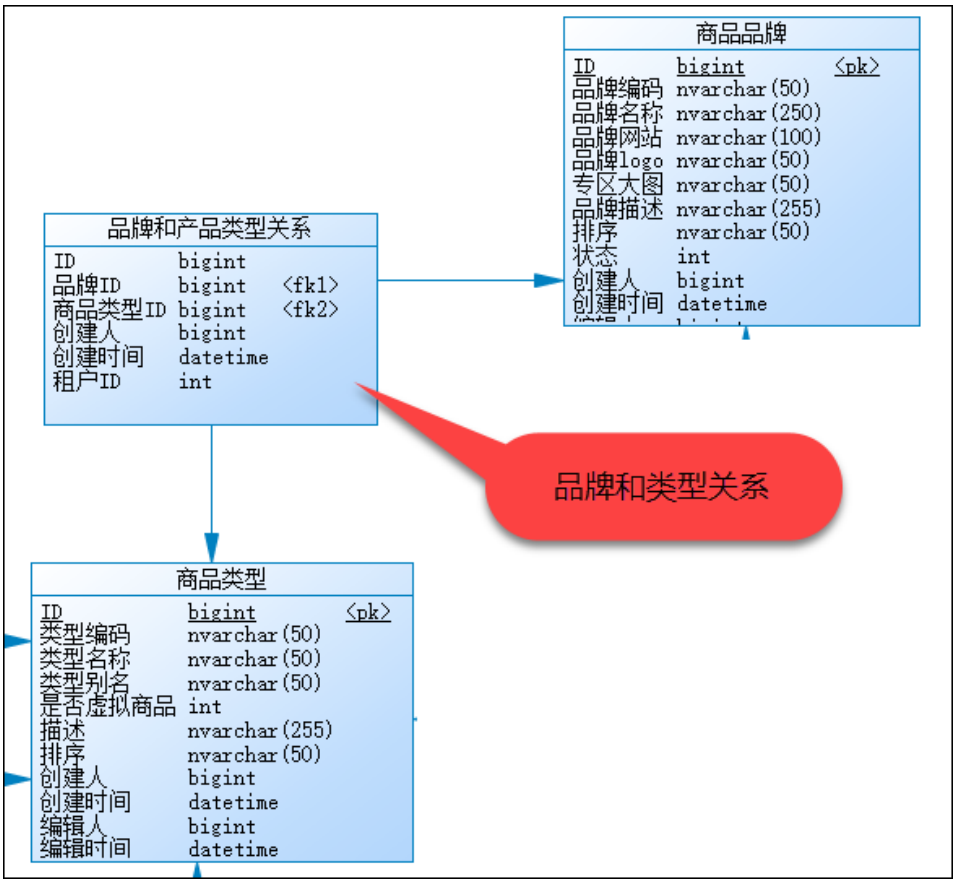


3.11.5. 多对多关系的数据处理

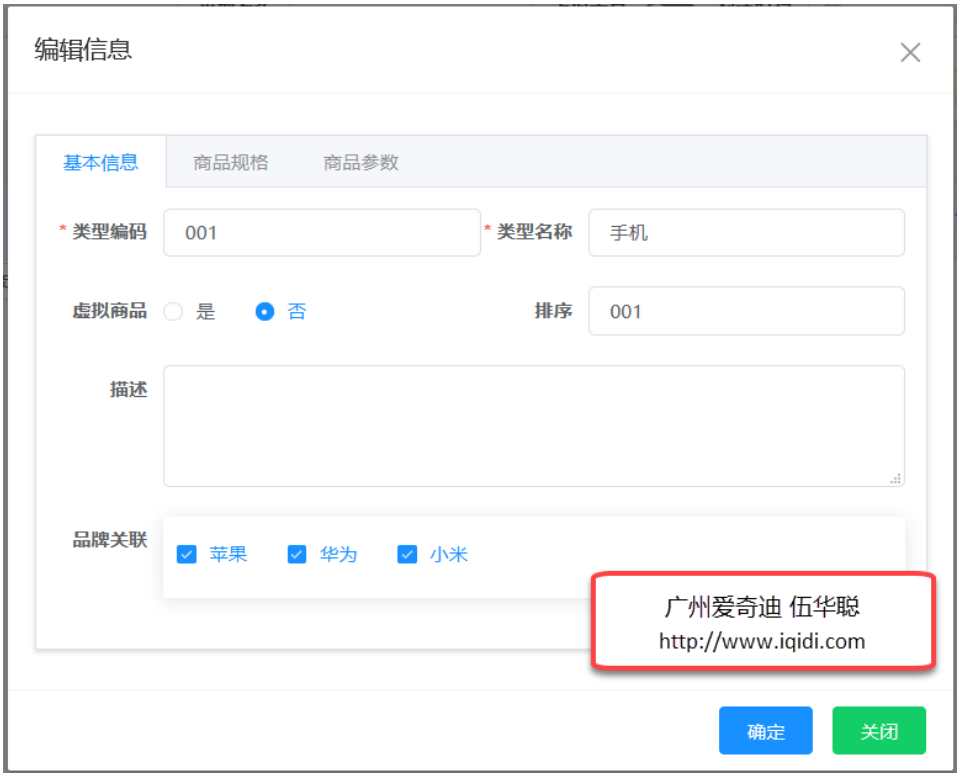
一般多对多的关系是指两个业务表之间存在关联关系，它们通过中间表（包含两个表的外键关系）建立多对多的关系。角色包含菜单资源也是多对多的关系，一般在角色新增或者编辑界面中进行维护。


编辑信息
角色编辑界面中和菜单的多对多关系
基本信息
权限
用户
菜单
输入关键字进行过滤
全选/反选
首页
权限管理
用户管理
机构管理
角色管理
菜单管理
字典管理
功能管理
审计日志
登录日志
商品管理
商品配置
品牌管理
分类管理
规格模板管理
商品类型
商品列表
广州爱奇迪 伍华聪
http://www.iqidi.com
确定
关闭

或者



功能界面设计的时候，就需要考虑和这些表之间的关系维护，如商品类型中，基本信息里面和品牌关系进行绑定。



不管上面的树形列表，还是很后面的复选框组，都是先请求关联主表的数据，然后再请求对应角色或者商品类型下的关系数据，绑定到界面上。

如对于上面的树形列表，通过设置树列表的数据，以及选中的记录就可以实现对应关系的绑定。

```
<el-tree
  ref="tree"
  class="filter-tree"
  style="padding-top:10px"
 :data="treedata"
  node-key="id"
  icon-class="el-icon-price-tag"
  default-expand-all
  highlight-current
  :show-checkbox="showcheck"
  :filter-node-method="filterNode"
:default-checked-keys="checkedList"
>
```

因此，在树形列表绑定的时候，需要请求原有的全部菜单数据，以及属于该角色下的菜单数据，两相整合就可以实现复选框选中已有菜单的效果了。

```
async getlist() { // 树列表数据获取
  // 获取全部功能列表
  var param = { SkipCount: 0, MaxResultCount: 1000, Tag: 'web' }
  var treeList = [] // 所有功能列表
  await menu.GetAll(param).then(data => {
    treeList = data.result.items
  })
  // console.log(treeList)

  // 获取角色菜单列表
  var grantedList = []
  if (this.roleId && typeof (this.roleId) !== 'undefined') {
    param = { RoleId: this.roleId, MaxResultCount: 1000, MenuTag: 'web' }
    await role.GetMenusInRole(param).then(data => {
      grantedList = data.result.items
    })
  }
  // console.log(grantedList)
```

当然我们也可以把角色包含菜单数据放在角色对象的 DTO 里面，然后一次性就可以获得菜单集合了，如我这里介绍的商品类型中的包含的品牌列表做法一样。

多对多关系，是我们业务表常见的一种关系，如果是只读的展示，我们直接通过关联关系获得记录展示即可；如果是进行编辑的处理，那么需要获取关联主表的全部记录进行展示，然后根据关联关系，显示复选框勾中的记录展示。

刚才说到，我们商品类型中对于多对多的关系，可以通过后端直接返回对应的数据记录集合的，这种做法可以避免细粒度 API 的请求过程，不过对于太大的数据集合，建议还是通过单独的 API 进行获取。

我们知道，我们所有业务对象提供服务，都是通过对应的应用层服务接口提供，而商品类型这里对应的控制器对象是 ProductTypeController，它继承自 BusinessController 基类对象。

其中为了数据对象的转换方便，我们重写了 Get 和 GetAll 的方法，并提供一个通用的模板方法用来修改对象 DTO 的关系，如下代码所示。

其中 ConvertDto 方法就是我们给子类重写，以便实现数据转换关系的。例如，我们在

子类 ProductTypeController 里面重写了 ConvertDto 方法。

```
/// <summary>
/// 对记录进行转义
/// </summary>
/// <param name="item">dto数据对象</param>
/// <returns></returns>
10 个引用
protected override ProductTypeInfo ConvertDto(ProductTypeInfo item)
{
    //获取关联品牌的对象列表
    var bindedBrands = GetBindedBrands(item.ID).Items;
    //获取关联品牌的ID列表
    var brandIds = bindedBrands.Select(s => s.ID).ToArray();

    //获取关联品牌的对象列表
    var brandDtos = bindedBrands.Select(s=> new BrandItemDto(s.ID, s.BrandCode, s.BrandName)).ToList();

    item.BindBrands = brandIds;     //纯ID集合
    item.BindBrandItems = brandDtos;//ID, BrandName,BrandCode 信息集合
    return item;
}
```

弄好了这些，我们测试接口，可以正确获得对应的记录列表了。

在列表展示界面中绑定已有关系代码如下所示。

```
<el-table-column align="center" label="绑定品牌列表">
  <template slot-scope="scope">
    <el-tag
      v-for="opt in scope.row.bindBrandItems"
      :key="opt.id"
      type="primary"
      :disable-transitions="false"
    >
      {{ opt.brandName }}
    </el-tag>
  </template>
</el-table-column>
```

在编辑界面中绑定已有关系代码如下所示。

```
<el-form-item label="品牌关联" prop="bindBrands">
  <el-checkbox-group v-model="editForm.bindBrands" style="padding:10px;box-shadow: 0 2px 12px 0 rgba(0, 0, 0, 0.1)">
    <el-checkbox v-for="(item, i) in brandList" :key="i" :label="item.id">{{ item.brandName }}</el-checkbox>
  </el-checkbox-group>
</el-form-item>
```

其中 editForm.bindBrands 是我们包含的关系，而 brandList 这是所有品牌列表，这个需要在页面创建的时候，单独获取。

```
created() {
  this.getlist() // 获取并显示列表
  this.getBrandList();// 获取所有品牌列表
},
methods: {
  getBrandList() { // 获取所有品牌列表
    var param = { // 分页条件
      SkipCount: 0,
      MaxResultCount: 1000 }
    brand.GetAll(param).then(data => {
      this.brandList = data.result.items
    })
  },
```

最后，需要介绍一下数据提交的时候，我们需要根据绑定列表关系，修改数据库已有的关联记录，这样实现关联关系的更新。

我们来看看创建商品类型和更新商品类型的时候，对关系数据的处理。

ProductTypeController.cs

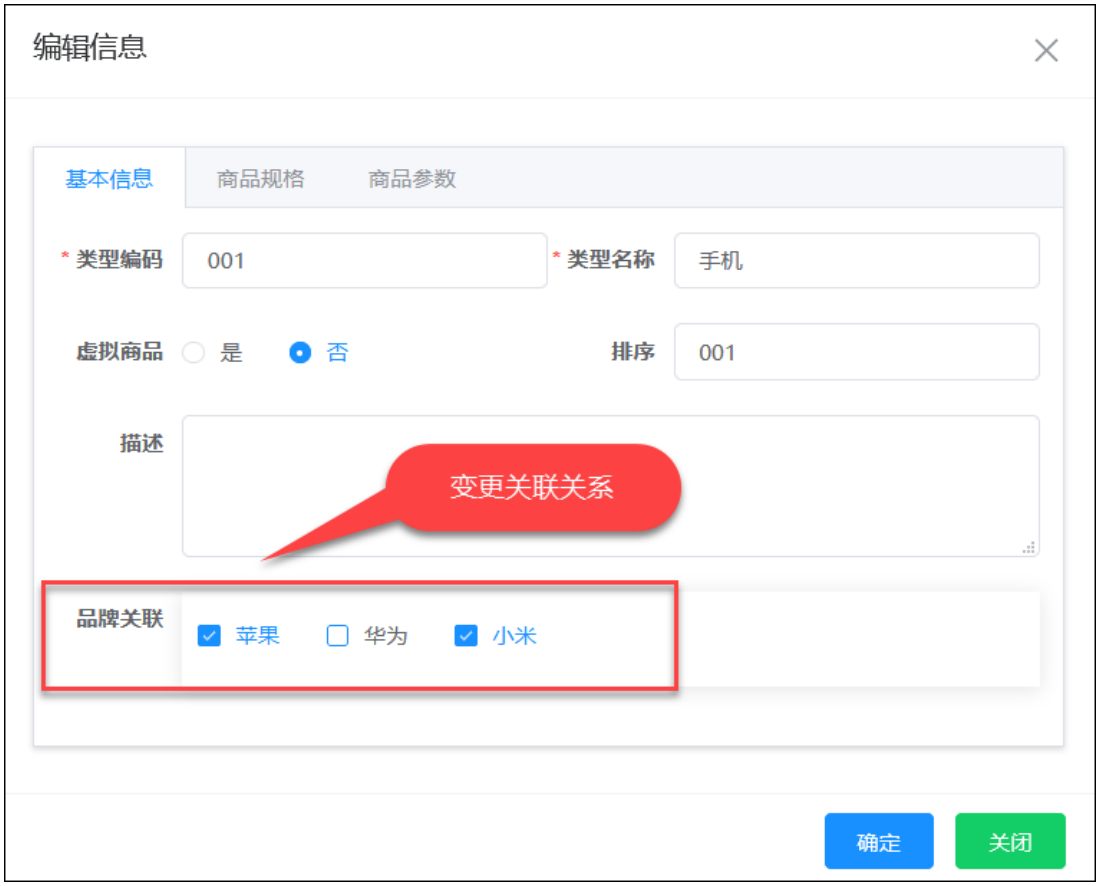
WebAPI　　WebAPI.Areas.Mall.Controllers.ProductTypeController

```
11 个引用
47  public override CommonResult Create(ProductTypeInfo info)
48  {
49      var result = new CommonResult();
50      var obj = base.CreateAndReturn(info);//创建并返回
51      if (obj != null)
52      {
53          //写入中间表关系
54          if (info.BindBrands != null)
55          {
56              var list = new List<BrandTypeInfo>();
57              foreach (var brandId in info.BindBrands)
58              {
59                  list.Add(new BrandTypeInfo()
60                  {
61                      Brand_ID = brandId,
62                      ProductType_ID = obj.ID,
63                      Creator = CurrentUser.ID.ToInt64()
64                  });
65
66              }
67              //增加新增的
68              result.Success = BLLFactory<BrandType>.Instance.InsertRange(list);
69          }
70      }
71      return result;
72  }
```

更新处理代码如下所示

```
/// 产品类型信息的控制器
/// </summary>
0 个引用
public class ProductTypeController : BusinessController<ProductType, ProductTypeInfo, ProductTypePagedDto>
{
    #region 写入数据前修改部分属性
    69 个引用
    protected override void OnBeforeInsert(ProductTypeInfo info)
    {
        //留给子类对参数对象进行修改
        info.CreateTime = DateTime.Now;
        info.Creator = CurrentUser.ID.ToInt64();
        base.OnBeforeInsert(info);
    }

    99+ 个引用
    protected override void OnBeforeUpdate(ProductTypeInfo info)
    {
        //写入中间表关系
        if (info.BindBrands != null)
        {
            var brandsDto = new BrandsToProductTypeDto() { BrandIds = info.BindBrands, ProductTypeId = info.ID };
            AddBrandToType(brandsDto);
        }

        //留给子类对参数对象进行修改
        info.EditTime = DateTime.Now;
        info.Editor = CurrentUser.ID.ToInt64();
        base.OnBeforeUpdate(info);
    }
```

前端界面效果如下所示。



以上就是关于中间表的常见处理操作，希望对你学习框架或者 Element 前端界面有所帮助。