



vfront开发指引(vfront developmen guide)

序言：环境

工欲善其事，必先利其器。在使用vfront进行开发之前，你需要安装以下开发环境及工具：

0. nodejs && npm
1. vue-devtools
2. visual studio code(vsc)

vsc 中需要安装的辅助插件如下：

- Auto Close Tag - 标签自动补全
- Auto Rename Tag - 标签重命名
- Beautify - 通用代码美化
- Bracket Pair Colorizer - 括号美化
- ESLint - JS语法检测
- JavaScript (ES6) snippets - JS代码提示
- jslint - JS质量检测（相对语法自定义，比ESLint严格）
- markdownlint - MD语法检测
- Path Intellisense - 路径侦测
- Prettier - 代码美化
- TODO Highlight - TODO高亮显示
- vscode-fileheader - 代码头部说明
- Vue.js Extension Pack - vue开发套件

第一章：Mock & Ajax

一、前言

在以往的开发过程中，前端往往依赖服务端提供的数据来进行开发、验证、运行以及测试。如果没有服务端提供的数据，客户端通过制造假数据开发会有以下的麻烦：

0. 制造加数据的过程相当的浪费时间；
1. 制造的假数据难以被清洗，导致发布的UI Server膨胀；
2. 依赖耦合，严重影响到测试；

为了解决以上的问题，在vfront中，提供了数据模拟模块（mock）和网络请求模块（ajax）。

二、Mock

引入模块

首先我们要在项目的mock目录下新建user.js文件，在该文件的第一行应该引入我们已经打包好的mock模块：

```
// vfront/src/mock/user.js
import Mock from 'mock';
```

模拟方式

接着，我们就可以开始按照mockjs官方提供的API来进行数据模拟了，我们有两种方式来模拟数据：

1. 本地化数据模拟（local mocking）

```
// 本地化模拟数据是将数据缓存到变量内存中
let data = Mock.mock({...});
```

2. 非本地化模拟数据（back mocking）

```
// 非本地化模拟数据是将模拟数据直接返回给数据层  
Mock.mock(url, {...});
```

注意：什么时候应该采用**本地化模拟数据**还是**非本地化模拟数据**，这取决于你的用途是什么。如果你仅仅只是为了演示效果，你可以通过**back mocking**的方式直接将模拟数据返回给**数据层**。如果你的用途用于展示完整的交互流程，**local mocking**的方式会是不错的选择，但意味着你要了解更多的业务知识，将服务层的逻辑写在这里。

举例：进行转账业务的时候，需要查询两个用户的资金及信用情况。在不需要模拟真实业务的情况下，可采用**back mocking**的方式将成功信息或失败信息传回去。若要模拟真实业务，则采用**local mocking**的方式将数据缓存下来，并通过模拟业务逻辑将计算出来的值返回回去。

本开发指南中案例介绍的为**local mocking**模式，大家根据自己的需求来确定模拟方式。

mock函数

模拟数据，主要是通过**mock**函数来进行，他可以直接模拟数据，也能将模拟的数据通过拦截HTTP请求将其返回过去，我们的**back mocking**方式就是这种方式。**mock**函数的API及释义如下：

```
Mock.mock(rurl?, rtype?, template | function(options))
```

rurl: [可选] 表示需要拦截的URL，可以是字符串（String），也可以是正则表达式（RegExp）。

rtype: [可选] 表示需要拦截的Ajax请求类型，例如：**GET**、**POST**、**PUT**、**DELETE**等。

template: [可选] 表示数据模版，可以是对象或字符串。

function(options): [可选] 表示用于生产本次请求的响应函数，可以通过**return**将模拟数据返回。

options: 指向本次请求的Ajax选项集，含有**url**、**type**、**body**三个属性，参见**XMLHttpRequest规范**。

相关案例

这里，我们给出一个用户管理的简单案例，通过案例实现对用户的CRUD操作：

```
// vfront/src/mock/user.js
import Mock from 'mockjs';
import url from 'url';
import qs from 'qs';

const GET = 'get';
const POST = 'post';
const PUT = 'put';
const DELETE = 'delete';
const PATCH = 'patch';
const parse = JSON.parse;

// 创建本地化数据, 可用于CRUD
let users = Mock.mock({
  "users|93": [
    {
      "id": '@guid',
      "name": 'name@increment',
      "passwd": '123456',
      "age|18-65": 100,
      "email": '@email'
    }
  ]
}).users;

// 查询单个数据, 请求地址为"/api/user/:guid"
Mock.mock('/api/user/[0-9a-zA-Z]{8}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{12}/',
  GET, options => {
  let id = options.url.split('/')[3];
  return {
    user: users.find(user => user.id === id)
  };
});

// 查询分页数据, 请求地址为"/api/user?"
Mock.mock('/api/user+(?{0,1}([A-Za-z0-9~]+|={0,1})([A-Za-z0-9~]*|\&{0,1})*)$', GET, options => {
  let urlQuery = url.parse(options.url).query;
  let params = qs.parse(urlQuery);
  let beginIndex = params.offset;
  let endIndex = params.offset + params.limit;
  return {
    users: users.filter((item, index) => {
      return index >= beginIndex && index < endIndex;
    })
  };
});

// 新建数据, 请求地址为"/api/user"
```

```
1 | Mock.mock('/api/user', POST, options => {
2 |   let user = parse(options.body).data;
3 |   user.id = Mock.mock('@guid');
4 |   users.unshift(user);
5 |   return {user: user};
6 | });
7 |
8 | // 删除数据，请求地址为"/api/user/:guid"
9 | Mock.mock('/api/User/[0-9a-zA-Z]{8}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{12}/',
10 | DELETE, options => {
11 |   let id = options.url.split('/')[3];
12 |   let index = users.findIndex(user => user.id === id);
13 |   users.splice(index, 1);
14 |   return {success: true};
15 | });
16 |
17 | // 更新数据（部分更新），请求地址为"/api/user/:guid"
18 | Mock.mock('/api/User/[0-9a-zA-Z]{8}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{4}-[0-9a-zA-Z]{12}/',
19 | PATCH, options => {
20 |   let id = options.url.split('/')[3];
21 |   let newUser = parse(options.body).data;
22 |   let index = users.findIndex(user => user.id === id);
23 |   users.map((user, index) => {
24 |     if(user.id === id){
25 |       for(let i in newUser){
26 |         user[i] = newUser[i];
27 |       }
28 |     }
29 |   });
30 |   return {user: users[index]}
31 | });
32 |
33 | // 查询数据总量，请求地址为"/api/user/count"
34 | Mock.mock('/api/user/count', GET, {
35 |   "count": users.length
36 | });
37 | // 批量业务，请求地址为"/api/user/batch-delete"
38 | Mock.mock('/api/user/batch-delete', POST, options => {
39 |   let data = parse(options.body);
40 |   let ids = data.data;
41 |   ids.forEach(id => {
42 |     let index = users.findIndex(user => user.id === id);
43 |     users.splice(index, 1);
44 |   });
45 |   return {success: true};
46 | });
47 |
48 |
49 |
50 |
51 |
```

其他API

有关如何模拟数据，请参考[mockjs wiki](#)

TODO

目前链路层比较简单，后期会加入更多的功能。比如HTTP中间件、全局跳转等。

0. 将mock模块定义为开发模式自启动，去除api模块对它的依赖；
1. 批处理及相关综合业务操作应当去除action名词；

三、Ajax

弃用jQuery

以往的开发模式中，通常会使用jQuery的Ajax模块发起HTTP请求。jQuery做为老牌开发类库，有着自己不败的地位，然而它更像是WEB农耕时代的利器，不太适用于现代化生产的WEB工程，他的劣势如下：

0. SPA应用导致页面的DOM数量增加、层级加深，使用jQuery查询效率降低；
1. 未生成的DOM导致测试困难；
2. 监听导致性能有所下降；

组件化开发方式能有效隔离开发人员对DOM（原子）的操作，转向组件（组织）的使用，能大大提高效率，解决了以上的问题。然而SPA仍然脱离不了Ajax，这也是前后端分离的宿求。前端模块中，有很多优秀的封装Ajax的包，在vfront架构中，我们选用axios来提供Ajax模块。

引入模块

我们可以在项目的api目录中新建 user.js 文件，表示针对user这一资源的全部接口，在该文件的第一行引入我们已经封装好的Ajax模块：

```
// vfront/src/api/user.js
import {get, del, post, put, patch} from '@/lib/ajax';
```

注意：我们是引入的封装好的模块，而非原始的 axios 模块。

返回接口

api 属于链路层的，它需要两个清晰的目的：提供简洁API，以及返回数据给数据层。按照如下写法将API返回：

```
export default {  
}  
}
```

相关案例

我们给出了用户管理相关的CRUD操作：

```
import {get, del, post, put, patch} from '@/lib/ajax';  
export default {  
    // 获取用户数量  
    getUserCount: () => get('/api/user/count'),  
    // 获取用户列表，带分页查询参数params  
    getUser: (params) => get('/api/user', params),  
    // 增加用户  
    addUser: (user) => post('/api/user', user),  
    // 删除用户  
    delUser: (id) => del('/api/user/' + id),  
    // 更新用户  
    updateUser: (id, newUser) => patch('/api/user/' + id, newUser),  
};
```

调用方法

在其他模块，如果想要调用该啊 api，只需要引用该模块即可使用它的接口：

```
import userAPI from '@api/user';  
// 通过api获取数据库全部用户  
userAPI.getUser();
```

返回结果

api 调用的返回结果都将是异步的，但这并不代表异步的不好用或者不好调试，通过ES6提供的Promise，我们可以轻松的使用：

```
// 准备缓存数据
let userCache;
userApi.getUser()
.then(data => {
  // 将返回的数据存入缓存
  userCache = data;
});
```

有关Promise的其他内容，可以通过百度以查询更多，或者[点击这份简略的帮助](#)。

四、链路层

链路层主要提供了客户端向服务端发起的请求接口以及模拟的数据。`axios`轻量级且封装了Ajax，足以胜任它在架构中的地位。而`mockjs`能拦截HTTP请求，模拟数据并返回，有效降低了客户端与服务端之间的耦合。

TODO

第二章：views & component

一、前言

vue中有7种定义组件模版的写法，这里没有完美的写法，各有各的优势。但为了更好的实现更好的团队协作与组件分离，我们推荐使用**单文件组件**的写法。它允许你将组件相关的所有内容（模版、样式、交互）组织在一个文件中，以`.vue`命名该文件。尽管它有一些劣势：需要预编译，某些IDE不支持`.vue`文件的语法高亮。但不用担心，我们建议采用的编辑器（vs code）有相关插件解决这一问题，webpack中也配置好编译`.vue`文件的配置项，所以不用担心，尽管按照教程去写即可。

一份简单的vue单文件形式应该是这样的：

```
<template>
<!-- 这里放组件的模版代码，即HTML和vue组件标签 --&gt;
&lt;/template&gt;
&lt;style&gt;
/* 这里放组件的样式代码，即CSS、LESS、Sass*/
&lt;/style&gt;
&lt;script&gt;
// 这里放组件的功能或交互代码，即JavaScript
&lt;/script&gt;</pre>
```

注意：有关vue组件的其他写法，不建议在本项目中使用，其他用法应该是作为高级组件开发时可以使用的。虽然不用写，但需要去做了解和学习。

二、模版视图

template 是组件的模版视图，提供了组件的非逻辑组成部分。一份简单的模版视图如下：

```
<template>
<div>
<Button type="primary" @click="onclick"> Primary </Button>
<Table :columns="columns1" :data="data1"> </Table>
</div>
</template>
```

在以上的视图模版中，我们发现，它的写法和HTML的Tag标签写法一摸一样，不同的区别在于：

0. 采用了非HTML标准协议的Tag标签，例如：**<Button>**；
1. 标签属性前有特殊符号，例如：**@** 和 **:**；

这里，我们进行简单的释义，因为这仍要和组件的逻辑部件组合在一起讲解，后续我们会给大家一一指出。

全局组件

<Button> 是一个**全局组件**，我们只需要通过标签来进行引用即可。既然有**全局组件**，也就有**局部组件**。是的，组件在整个组件化开发中就像是某个对象一样，可以被其他模块、视图、组件所嵌套和引用，因此他有全局和局部之分。这并不是我们所关心的，我们只需要使用全局组件即可。我们用的所有的部件来自于**iView**，各位请参阅网站的示例学习和使用即可。

属性绑定

对于组件来说，数据是必不可少的，`:`符号就是用来表示动态属性的。`:columns=`是一个指向，`columns1`是一个具体的引用，引用的值会从逻辑部件中取出。因此，它可以是一个简单的值，也可以是一个复杂的对象。

事件绑定

有属性就会有交互，组件的交互，也就是事件是通过`@`符号来表示。同样，`@click=`是一个指向，`onclick`是一个具体的引用，不过它引用的是一个函数。

三、逻辑组件

`script`是组件的逻辑部分，用来指定组件中的数据来源、变换关系、交互。一个简单的逻辑组件代码如下：

```
export default {
  name: 'my-component',
  props: [...],
  data: () => {
    return {
      columns1: [...]
    }
  },
  computed: {
    data1: () => [...]
  },
  methods: {
    onclick: () => {
      // ...
    }
  }
}
```

注册组件

我们通过`export default {}`将组件导出，如果给出了`name`属性，即将该组件注册且命名。这里的命名是全局的，因此`name`属性不能重复。

组件属性

组件有三种属性，外部属性、动态属性、计算属性。外部属性是通过调用组建时传入属性值使之生成，多用于组件化开发，这里我们不多做说明。动态属性和计算属性严格来说都是计算属性，不同的是，前者是从真实数据直接映射出来（属性），后者通过计算得出（函数）。在模版视图的案例中，`:columns=` 指向的 `columns1` 就是这里 `data` 函数返回的 `columns1`，`:data=` 指向的 `data1` 就是这里 `computed` 列出的 `data1` 函数。

组件方法

组件的主要交互逻辑都在这里，在模版视图的案例中，`@click=` 指向的 `onclick` 就是这里 `methods` 列出的 `onclick` 函数。

第三章：router

前言

无论使用SPA应用还是多页面应用，路由的功能都是必不可少的，他通过对URI的解析将特定的页面或者组件引导进来，告诉页面应该如何渲染并使用哪些数据。路由的所有功能都将使用 `vue` 系列的 `vue-router` 来使用，如果你还不了解 `vue-router` 的功能，请点击它的[中文官网](#)学习。

我们在 `vfront` 架构已经添加好了 `vue-router`，你只需要找到 `router.js` 文件，即可按照教程编写：

```
// vfront/src/router.js
const routers = [...]; // 路由配置
```

注意： 我们并不主张所有人都学习路由的功能，因为这些将针对高级开发人员而言。但你仍需要了解一些基础的功能。

路由视图

在 `vue` 组件中，我们用 `<router-view>` 标签来表示一个路由视图，它是路由的一个出口，路由匹配到的组件或者视图都将渲染在这里。路由视图可以不只有一个，如果有多个，就要采用命名视图。

接着，我们要在 `router.js` 中定义路由器的逻辑，这个逻辑很简单，就是URI与Component的一一对应：

```
const routers = [
  { path: '/login', component: (resolve) => require(['./login.vue'], resolve)},
  { path: '/user-list', component: (resolve) => require(['./user-list.vue'], resolve)}
];
```

这里的路由配置中，`path` 对应的是URI，即 `http://localhost:8080/login`，而 `component` 对应的是一个加载函数，这个函数不用去管，我们只需要修改引用（`require`方法）的组件即可，即 `./login.vue` 替换成其他你需要映射的vue组件。

声明式路由

我们可以通过使用 `<router-link>` 标签来创建声明式路由，例如：

```
<router-link :to="/login"> </router-link>
```

它会转换成一个HTML的 `a` 标签，点击查阅更多的[配置及说明](#)。

编程式路由

同样，我们可以借助 `router` 的实例方法，通过编程的方式来实现跳转：

```
router.push('/login');
```

注意： 在Vue实例内部，你可以通过 `$router` 访问路由实例，因此你可以通过 `this.$router.push` 的方法来实现上面的代码。点击查阅更多的[配置及说明](#)

TODO

更多其他的资料，以后会补充，或者参阅[官方文档](#)。

第四章：store & validation

一、前言

如果采用传统的MVC架构，之前的内容就已经可以胜任。我们可以让视图层的组件直接和链路层通信来获取数据。但这会有两个问题：

0. 大量的业务逻辑被嵌在视图层，使得视图过于臃肿，视图对业务的依赖过重，无法分离；
 1. 使得业务逻辑无法被测试，影响整体的正确性；
 2. 组件与组件之间的数据通讯依赖大量的数据通信，组件与组件之间形成强依赖；

为了解决这些问题，我们得将所有的数据抽离出来进行统一管理，做一个专门的层级——数据层。这样做好处和MVVM的体系架构非常契合，以后的开发就变成针对数据的编程，而非视图的编程了。

二、store

vfront框架中使用vuex来组建我们的数据层。vuex特别的简单，它只有四个最基本的概念：

state、getter、mutation、action。它点写法应该形如以下形式：

```
const store = {
  state: { ... },
  getters: { ... },
  mutation: { ... },
  action: { ... }
};
export default store;
```

state

state是vuex的单一状态树，这个对象包含了全部的应用层状态，除了我们的业务数据，还包含样式数据、状态数据等。因此，它能做为我们整个应用的唯一数据源（SSOT）而存在。这也意味着，我们的所有操作都是针对数据的，不再是基于视图的。state就是一群键值队的集合，和普通对象没有区别，但是它经过vuex实例化后就意义非凡了：

```
const store = new Vuex.Store({
  state: {
    list: [ ... ],
    count: 0
  }
});
```

注意：store.state 在外部应该是不可直接调用，更不可直接写入的。它的 get/set 操作都应该由它内部的 getter/mutation/action 来完成。这样一方面形成了数据隔离，便于业务分离和测试，同时对于数据来说也是安全的。

注意：因为 store 是单一数据源（SSOT），所以整个应用程序中有且只应该有一个 new Vuex.Store(...) 的代码。在我们的 vfront 架构中，普通开发人员只需要写配置化的 store 模块即可，不需用此实例化方法。我们这里的，仅仅做为演示。

getter

store.state 中的属性是不可直接调用的，我们需要使用配套的 getter 来获取数据：

```
const store = {
  getters: {
    getList: state => state.list
  }
};
```

getter 中都是函数，是 state 属性中的 get 方法，所有取数据的来源都是从这里获取的。getter 函数的参数中可以获取到 state，可以直接将状态树的值获取并返回。当然，getter 也是可以进行计算的：

```
const store = {
  getters: {
    getList: state => state.list,
    getListCount: state => state.list.length
  }
};
```

getter 的调用非常简单，在 component 中，直接通过 this.\$store 来获取 store，由于 state 是不允许直接获取的，因此我们要这样获取：

```
// 某组件中的代码段
computed: {
  userList () {
    // return this.$store.getters.getList() 效果一致
    return this.$store.getters.getList;
  }
}
```

getter 也是可以传参的，我们可以这样定义：

```
// store代码片段
getters: {
  /**
   * 以下代码等价于：
   * getListById: function (state){
   *   return function (id){
   *     return state.list.find(function (item){
   *       return item.id === id;
   *     });
   *   }
   * }
   * 这就是箭头函数的优势，也是ES6给我们带来的阅读上的便利
   */
  getListById: state => id => state.list.find(item => item.id === id)
}
```

调用的过程就可以使用 `this.$store.getters.getListById(id)` 了。是不是非常的简单和直观。

mutation

更改 state 状态的唯一方式就是通过 `commit` 方法提交 mutation。每个 mutation 都是一个函数，并且接受 state 做为参数：

```
const store = {
  state: { count: 1},
  mutations: {
    increment (state) {
      state.count++;
    }
  }
};
```

调用的过程如下：

```
// 某组件代码段
methods: {
  myClick () {
    this.$store.commit('increment');
  }
}
```

当然，我们也可以向 `commit` 方法中传入参数，即 `mutation payload`，载荷提交：

```
// store代码段
mutations: {
  increment (state, n) {
    state.count += n;
  }
}
// 某组件代码段
this.$store.commit('increment', 10)
```

注意： `mutation` 必须是同步的，这将使得应用变得易于调试。

注意： `mutation` 中不允许直接修改对象类型，如果遇到数组、JSON对象，必须使用 `Vue.set` 方法，相关资料请[查阅这里](#)。

action

`action` 类似于 `mutation`，不同的是：

0. `action` 不主动更新数据，而是通过提交 `mutation` 变相更新；
1. `action` 可以包含任意的异步代码，例如：ajax获取服务端数据；

例如：

```
const store = {
  state: {
    user: []
  },
  mutations: {
    setUser (state, user) {
      // 上面提到过，修改对象要用Vue.set方法
      Vue.set(state, 'user', user || {});
    }
  },
  actions: {
    ajaxLoadUser ({commit}) {
      api.getUser()
        .then(data => {
          commit('setUser', data.user);
        });
    }
  }
};
```

而调用 `action` 是采用分发的方法 `dispatch`：

```
// 某组件代码段
methods: {
  refresh () {
    this.$store.dispatch('ajaxLoadUser');
  }
}
```

和 `mutation` 一样，`action` 也是支持 `payload` 载荷提交。

注意：`action` 和 `mutation` 都能设置状态树的属性值，但 `action` 绝对不操作 `state`，他仅仅做一些异步的交互后，调用 `mutation` 来间接设置状态树的值。

module

`store` 是支持模块化的，毕竟，对于整个应用系统，数据量是非常大的。如果把所有的数据都写在一个文件中，也会变得非常难以维护。我们已经为大家组建了自动模块化，只需要按照上面的需求去写即可。

TODO

0. 加入自动模块化

三、validation

数据验证这块比较简单，我们采用[async-validator](#)来进行数据验证：

```
<template>
<Form ref="formValidate" :model="formValidate" :rules="ruleValidate" :label-width="80">
  <FormItem label="Name" prop="name">
    <Input v-model="formValidate.name" placeholder="Enter your name"> </Input>
  </FormItem>
  <FormItem label="E-mail" prop="mail">
    <Input v-model="formValidate.mail" placeholder="Enter your e-mail"> </Input>
  </FormItem>
  <FormItem>
    <Button type="primary" @click="handleSubmit('formValidate')"> 提交 </Button>
    <Button type="ghost" @click="handleReset('formValidate')" style="margin-left: 8px"> 重置 </Button>
  </FormItem>
</Form>
</template>
<script>
export default {
  data() {
    return {
      formValidate: {
        name: '',
        mail: ''
      },
      ruleValidate: {
        name: [
          { required: true, message: "姓名不能为空", trigger: "blur" }
        ],
        mail: [
          { required: true, message: "邮箱不能为空", trigger: "blur" },
          { type: "email", message: "请输入正确的邮箱格式", trigger: "blur" }
        ]
      }
    };
  },
  methods: {
    handleSubmit(name) {
      this.$refs[name].validate(valid => {
        if (valid) {
          this.$Message.success("Success!");
        } else {
          this.$Message.error("Fail!");
        }
      });
    },
    handleReset(name) {
      this.$refs[name].resetFields();
    }
  }
}
```

```
    }
};

</script>
```

以上代码中，`ruleValidate` 定义了表单的字段验证规则，通过以上案例以及翻阅文档，我们列出 `validation` 的选项释义：

0. `required` - 必要性验证；
1. `type` - 类型验证，常用的还有`string`、`number`、`url`、`date`等，[查阅文档](#)以提供帮助；
2. `pattern` - 正则验证；
3. `range` - 范围验证；
4. `length` - 长度验证；
5. `enum` - 枚举验证；
6. `validator` - 自定义验证，可发起ajax请求向服务器进行验证；
7. `trigger` - 触发验证的交互行为，例如：`blur`（鼠标移开）、`change`（内容变化）；

在组件中，我们通过 `this.$refs` 来获取表单 `form`，表单的命名由组件的 `ref` 属性来确定。获取到表单后，直接调用 `validate` 方法进行验证。