

vue 总结（进阶篇）

1. 路由篇

1.1 各种写法

1.1.1 组件导入写法

1.1.2 路由懒加载

1.1.3 动态路由匹配

1.1.4 命名路由

1.1.5 命名视图

1.1.6 重定向（三种写法）

1.1.7 嵌套路由

1.1.8 别名

1.1.9 程式化导航（重点！各种写法）（通过 `this.$router` 访问到路由实例）

1.2 路由进阶篇

1.2.1 路由传参的几种模式

1.2.2 路由导航守卫

1.2.3 历史路由待完善

2. Vuex 篇

2.1 先提一下 组件传值（并整理父子组件和兄弟组件的传参）

2.1.1 分析

2.2 bus 的使用（作为交互的中介）

2.2.1 原理：`emit` 可以触发当前实例上的一些事件，`$on` 是给当前实例绑定一个事件监听，然后事件...

2.3 Vuex 基础

2.3.1 state 和组件拿到状态的各种方法

2.3.1.1 组件中获取的各种方式

第一种：通过 `$store` 访问

第二种：使用工具函数 `mapState` 写法：`import { mapState } from 'vuex'` 利用了解构赋值

2.3.1.2 开启命名空间 模块中 添加 `namespace: true`，然后使用 `createNamespacedHelpers`

2.3.1.2 如果想要直接 使用 `mapState` 完成这种效果 需要在对象传入模块名 或者使用路径写法，`mut...`

2.3.2 getters

2.3.3 mutations

2.3.3.1 先提一下自定义属性

2.3.3.2 单值写法和后面参数是对象的写法

2.3.3.3 纯对象写法： 直接传递一整个对象

2.3.3.4 给state 添加一个 版本号的信息 使用vue 身上的set方法（后面解释为什么要用set）

2.3.3.5 使用mapmutations 工具辅助函数

2.3.4.6 还可以路径写法， 如果是拆分细模块且模块相互对应不推荐这样写每次 都this后面带路径， 还...

2.3.3.6 模块中的话 开启和不开启命名空间的情况下（有注意点）

不开启命名空间的情况下,可以不用书写模块名，因为vuex 会把mutations 和 actions 和getters 和根级...

2.3.4 actions 做异步操作 然后触发mutations 之后修改state 里面的数据

1. 在src 的api文件夹下创建 app.js，模拟请求 代码如下
2. actions.js 中 把方法引用过来，并创建自己的方法
3. store.vue 中使用辅助函数把actions 的方法映射到当前, 然后点击的时候触发
4. 开启命名空间后的路径写法 （不太推荐）
5. 直接使用 \$store 身上的dispatch 方法

2.3.5 module 模块

动态注册模块（先写一个 普通版本的 后面书写一个 在模块中的）

在模块中的

总结

2.4vuex 进阶篇

2.4.1 vuex 严格模式 strict: true

3. Ajax 阶段（解决跨域问题，封装axios ， 开发实战中使用）

3.1 解决跨域问题

2. 后端进行配置

1. 路由篇

```
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3 Vue.use(VueRouter)
4
5 const routes = [
```

```

6 ]
7
8 const router = new VueRouter({
9   routes
10 })
11
12 export default router

```

1.1 各种写法

1.1.1 组件导入写法

```

1 import Home from '../views/Home.vue'
2 //path 指定路径, component 指定展示的组件
3 const routes = [
4   {
5     path: '/',
6     component: Home
7   }
8 ]

```

1.1.2 路由懒加载

```

1 const routes = [
2   {
3     path: '/parent',
4     component: () => import('@views/parent.vue'),
5   }
6 ]

```

1.1.3 动态路由匹配

```

1 //第一种 路由中
2 const routes = [
3   {

```

```

4    // 动态路由匹配
5    path: '/argu/:name',
6    component: () => import('@views/argu.vue')
7  }
8 ]
9
10 //组件中
11 <template>
12   <div>
13     路由动态匹配{{ $route.params.name }}
14   </div>
15 </template>

```

1.1.4 命名路由

```

1  const routes = [
2    {
3      path: '/about',
4      name: 'About', //命名路由
5      component: () => import('@views/About.vue')
6    },
7  ]
8
9  //组件中
10 <template>
11   <div id="app">
12     <div id="nav">
13       <!-- 命名路由 to 要变成属性绑定的形式， 属性是一个对象 -->
14       <router-link :to="{ name: 'About' }">About</router-link>
15     </div>
16     <router-view/>
17   </div>
18 </template>

```

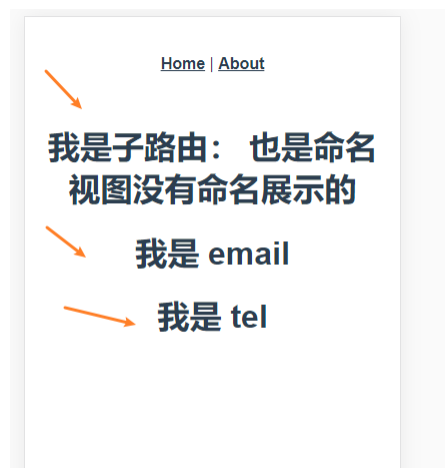
1.1.5 命名视图

```

1 <template>
2   <div id="app">
3     <router-view/>
4     <!-- 如果想在一個頁面上顯示不同的視圖，而且讓指定的視圖顯示在指定的位置，
        那就要用到命名視圖-->
5     <router-view name="email"/>
6     <router-view name="tel"/>
7   </div>
8 </template>
9
10
11 const routes = [
12   {
13     // 命名視圖
14     path: '/named_view',
15     props: true,
16     //注意 這裡要帶 s 因為有多個
17     components: {
18       //沒有指定名字顯示 default 指定的 指定了的話展示各自對應的
19       default: () => import('@/views/child'),
20       email: () => import('@/views/email.vue'),
21       tel: () => import('@/views/tel.vue')
22     }
23   },
24 ]

```

效果如下三個被路由匹配的組件都被展示出來



```
▼ <App>
  <RouterLink>
  <RouterLink>
  <Child> router-view: /named_view
  <Email> router-view: /named_view
  <Tel> router-view: /named_view
```

1.1.6 重定向（三种写法）

```
1 // 重定向（第一种写法）
2 {
3   path: '/main', redirect: '/'
4 }
```

或者这么写

```
1 // 重定向（第二种写法）
2 {
3   path: '/main', redirect: {
4     name: 'Home' // 相当于是 :to 后面跟对象指定跳转地址的方法
5   }
6 }
```

```
1 // 重定向（第三种写法）
2 {
3   path: '/main', redirect: to => {
4     return '/'
5   }
6 }
7 // 根据箭头函数的定义还可以简写成
8 {
9   path: '/main', redirect: to => '/'
10 }
```

1.1.7 嵌套路由

```

1 //默认子路由
2 const routes = [
3   {
4     path: '/',
5     component: Layout,
6     children: [{
7       path: '',
8       name: 'Home',
9       component: () => import('@views/Home')
10    }]
11  },
12 ]
13
14 // home前置页面
15 const routes = [
16   {
17     path: '/',
18     component: Layout,
19     redirect: '/home',
20     children: [{
21       path: 'home',
22       name: 'Home',
23       component: () => import('@views/Home')
24     }]
25   },
26 ]
27

```

1.1.8 别名

```

1 const routes = [
2   {
3     path: '/',
4     //使用alias 取别名, 这样访问 /home_page 直接访问'/' , 是一样的
5     alias: '/home_page',
6     name: 'Home',
7     component: Home
8   }

```

1.1.9 程式导航（重点！！各种写法）（通过 this.\$router 访问到路由实例）

```
1      handleClick () {
2
3          // 后退一页
4          this.$router.go(-1)
5          // 前进一页
6          this.$router.go(1)
7          // 返回
8          this.$router.back()
9          // 跳转到指定页面（写法1）
10         this.$router.push('/named_view')
11         // 跳转到指定页面（写法2）
12         this.$router.push({
13             name: 'named'
14         })
15         //可以携带参数：效果在代码结束后面
16         this.$router.push({
17             name: 'argu',
18             query:{
19                 name: 'aaa'
20             }
21         })
22         //使用params 或者直接使用path 然后用模板字符串拼接：效果在代码结束后面
23         this.$router.push({
24             name: `argu`
25             params: {
26                 name: 'lwq'
27             }
28         })
29         //使用params 另一种写法
30         const name = 'lwq'
31         this.$router.push({
32             path: `/argu/${name}`,
33         })
```



```

34     //重定向（写法1）
35     this.$router.replace('/named_view')
36     //重定向（写法2）
37     this.$router.replace({
38         name: 'named'
39     })
40 }

```

携带参数的效果: 使用 query `st:8080/#/?name=aaa`

使用params `localhost:8080/#/argu/lwq`

1.2 路由进阶篇

1.2.1 路由传参 的几种模式

```

1  const routes = [
2    {
3      path: '/argu/:name',
4      name: 'argu',
5      //路由这边先 props: true, 开始传参 和父子组件传参一样 然后 组件那边接收
6      props: true,
7      component: () => import('@views/argu.vue')
8    }
9  ]
10 //组件为中 先使用props 指定要接受的内容 第一种写法 对象的形式
11 props: {
12   name: {
13     可以指定接收的类型, 和没有传参默认显示的内容
14     type: [String, Number],
15     default: 'lwq'
16   }
17 }
18
19 //第二种 数组的写法
20 //props: ['name'],
21

```

```
22 //然后直接就可以渲染出来
23 <template>
24   <div>
25     路由传参拿到的name 值: {{name}}
26   </div>
27 </template>
```

```
1 //还可以直接指定携带参数      路由中
2 {
3   path: '/about',
4   name: 'About',
5   component: () => import('@views/About.vue'),
6   props: route => {
7     food: route.query.food
8   }
9 },
10
11 //组件中可以指定下类型和默认值
12 props:{
13   food:{
14     type: String,
15     default: "apple"
16   }
17 }
```

1.2.2 路由导航守卫

```

/**
 * 1. 导航被触发
 * 2. 在失活的组件（即将离开的页面组件）里调用离开守卫 beforeRouteLeave
 * 3. 调用全局的前置守卫 beforeEach
 * 4. 在重用的组件里调用 beforeRouteUpdate
 * 5. 调用路由独享的守卫 beforeEnter
 * 6. 解析异步路由组件
 * 7. 在被激活的组件（即将进入的页面组件）里调用 beforeRouteEnter
 * 8. 调用全局的解析守卫 beforeResolve
 * 9. 导航被确认
 * 10. 调用全局的后置守卫 afterEach
 * 11. 触发DOM更新
 * 12. 用创建好的实例调用beforeRouterEnter守卫里传给next的回调函数
 */

```

```

1    组件内的守卫：
2    针对组件进行拦截
3    beforeRouteEnter (to, from, next) {
4    // 在渲染该组件的对应路由被 confirm 前调用
5    // 不！能！获取组件实例 `this`
6    // 因为当守卫执行前，组件实例还没被创建
7    next()
8    },
9    beforeRouteUpdate (to, from, next) {
10   // 在当前路由改变，但是该组件被复用时调用
11   // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2
    之间跳转的时候，
12   // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情
    况下被调用。
13   // 可以访问组件实例 `this`
14   },
15   beforeRouteLeave (to, from, next) {
16   // 导航离开该组件的对应路由时调用
17   // 可以访问组件实例 `this`
18   console.log('即将离开about')
19   if(confirm('当前表单没有提交？确定要离开首页？')){
20     next()
21   }
22   }
23

```

```

24 // 挂载路由导航守卫 设置拦截 如果没有token 的话强制跳转回去 login
25 router.beforeEach((to, from, next) => {
26     // to 代表将要访问的路径
27     // from 表示从哪个路径跳转而来
28     // next() 放行
29     // next('./login') 强制跳转到login
30     if (to.path === "/login") return next();
31     // 获取 token
32     const tokenStr = window.sessionStorage.getItem("token");
33     // 如果没有登录强制跳转回login
34     if (!tokenStr) return next("login");
35     next();
36 });
37

```

1.2.3 历史路由待完善

2.Vuex 篇

2.1 先提一下 组件传值（并整理父子组件和兄弟组件的传参）

重点提要：

1. !! 单向数据流：父组件向子组件传值，一定是通过 属性，子组件通过props 接收
2. 而子组件如果要修改父组件传过来的值，一是通过事件的方式，通过emit触发并把需要修改的值以参数的形式传递过去 （详情看例子1）

然后父组件绑定一个事件来知道子组件要改值，然后再子组件里面修改值

3.兄弟组件 （详情看例子 2）（下一篇完善）

例子1：

Ainput.vue 组件中

```

1 // 为了区分自己封装组件最好有统一的前缀
2 <template>
3   <input @input='handleInput' :value="val">
4 </template>

```

```

5 <script>
6 export default {
7   name: "Ainput",
8   components: {},
9   props: {
10     value: {
11       type: [String, Number],
12       default: ''
13     }
14   },
15   data() {
16     return {
17       val: ''
18     };
19   },
20   methods: {
21     handleInput(event) {
22       const value = event.target.value
23       this.$emit('input', value)
24     }
25   },
26 }
27 </script>
28
29 <style scoped lang="less">
30
31 </style>

```

store.vue 中

```

1 <template>
2   <div>
3     <a-input v-model="inputValue" />
4     <h1>{{inputValue}}</h1>
5   </div>
6 </template>
7
8 <script>
9   // _c  这里是配置文件中配置的  相当于 src/components

```

```

10 import AInput from '_c/Ainput.vue'
11 export default {
12   name: "store",
13   components: {
14     AInput
15   },
16   props: {},
17   data() {
18     return {
19       inputValue: ''
20     };
21   },
22 }
23 </script>
24
25 <style scoped lang="less">
26
27 </style>

```

2.1.1 分析

先整体分析 先Ainput 组件中书写了一个文本框，然后 给他绑定了输入事件，触发了 函数，展示的数据是数据绑定的 val， 然后 在store ，显示导入了Ainput 这个组件 然后注册组件，在页面 显示他， 最后在双向数据绑定这个属性就可以了

核心逻辑分析： 为什么直接 双向数据绑定就可以达成效果，

为什么可以直接出现效果： 是因为 v-model 相当于 :value='inputValue' @input='handleInput' 相当于属性绑定加上一个输入事件 ， 然后事件里面，

handleInput (val) {this.inputValue=val} 函数里面有将我们传输过来的val ， 赋值给我们定义的 inputValue 。

例子2：

store.vue 中

```

1 <template>
2   <div>
3
4     <a-input @input="handleInput"/>
5     <a-show :content="inputValue"/>
6     <!-- <h1>{{inputValue}}</h1> -->
7   </div>
8 </template>
9
10 <script>
11 import AShow from '_c/Ashow.vue'
12 import AInput from '_c/Ainput.vue'
13 // import { mapState, mapMutations, mapActions, mapGetters } from
    'vuex'
14 export default {
15   name: "store",
16   components: {
17     AInput,
18     AShow
19   },
20   props: {},
21   data() {
22     return {
23       inputValue: ''
24     };
25   },
26   methods: {
27     handleInput(val){
28       this.inputValue = val
29     }
30   }
31 }
32 </script>
33
34 <style scoped lang="less">
35
36 </style>

```

Ashow 中

```
1 <template>
2   <div>
3     {{ content }}
4   </div>
5 </template>
6
7 <script>
8 export default {
9   name: "Ashow",
10  components: {},
11  props: {
12    content: {
13      type: [String, Number],
14      default: ''
15    }
16  },
17  data() {
18    return {};
19  },
20 }
21 </script>
22
23 <style scoped lang="less">
24
25 </style>
```

2.2 bus 的使用（作为交互的中介）

1. 创建一个 bus 文件夹 然后里面创建一个index .js 书写如下代码

```
1 import Vue from 'vue'
2 const Bus = new Vue()
3 export default Bus
```

2. main.js 中书写


```
1 //导入文件并 将其挂到vue原型上
2 import Bus from './bus'
3 Vue.prototype.$Bus = Bus
```

```
1 //从打印中可以得只相当于 创建一个空的vue 实例 作为交互的中介
2 console.log(this.$Bus)
```

tel.vue 中

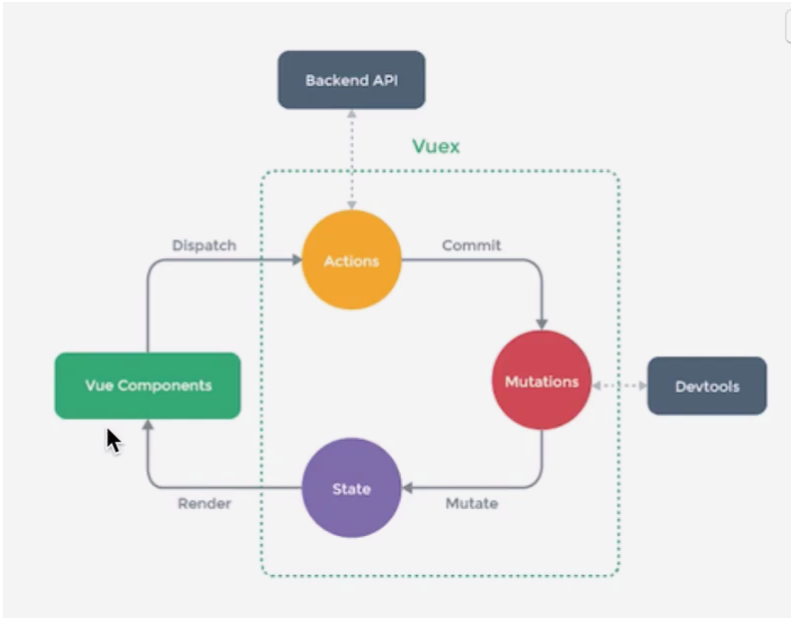
```
1 <template>
2   <div class="tel">
3     <p> {{message}}</p>
4   </div>
5 </template>
6
7 <script>
8 export default {
9   name: "tel",
10  data() {
11    return {
12      message: ''
13    };
14  },
15  mounted() {
16    this.$bus.$on('on-click', mes => {
17      this.message = mes
18    })
19  }
20 }
21 </script>
22
23 <style scoped lang="less">
24 .tel {
25   border: 1px solid red;
26 }
27 </style>
```

emali.vue 中

```
1 <template>
2   <div class="email">
3     <button @click="handleClick">按钮</button>
4   </div>
5 </template>
6
7 <script>
8 export default {
9   name: "email",
10  data() {
11    return {};
12  },
13  methods: {
14    handleClick() {
15      this.$bus.$emit('on-click', 'hello')
16    }
17  },
18 }
19 </script>
20
21 <style scoped lang="less">
22 .email {
23   border: 1px solid green;
24 }
25 </style>
```

2.1.1 原理： emit 可以触发当前实例上的一些事件，\$on 是给当前实例绑定一个事件监听，然后事件是绑定在this.\$bus 上，所以在this.\$bus 上在监听这个事件这样就形成了响应

2.3 Vuex 基础



先分析图片流程： 组件触发actions 做接口的请求等异步，请求后可以触发mutations， 然后通过mutations 去修改state 的状态值，然后state修改后， 触发组件里视图的渲染

```
pre > index.js > default
import Vue from 'vue'
import Vuex from 'vuex'
import state from './state'
import mutations from './mutations'
import actions from './actions'
import getters from './getters'
import user from './module/user.js'
Vue.use(Vuex)

export default new Vuex.Store({
  state,
  mutations,
  actions,
  getters,
  modules: {
    user
  }
})
```

根级别

更加独立的拆分然后放到modules中

2.3.1 state 和组件拿到状态的各种方法

```
1 //state.js 中
2 const state = {
3   appName: 'admin'
4 }
5
6 export default state
```

2.3.1.1 组件中获取的各种方式

第一种：通过\$store 访问

```
1    //直接获取
2    <p>{{ $store.state.appName }}</p>
3
4    //或者在计算属性中
5    computed: {
6      appName(){
7        return this.$store.state.appName
8      }
9    },
10   //然后直接拿计算属性中属性
11   <p>{{ appName }}</p>
12   //写在user.js 模块中 在state加上模块的名字
13   userName(){
14     return this.$store.state.user.userName
15   }
```

第二种： 使用工具函数 mapState 写法： import { mapState } from 'vuex' 利用了 解构赋值

```
1 import { mapState } from 'vuex'
2 //相当于：
3 import vuex from 'vuex'
4 const mapState = vuex.mapState
```

在计算属性中使用展开运算符 ... 映射到当前组件

```
1 //使用... 将数据扁平化，映射到当前组件，会有命名冲突的问题！！
2 //（传数组的写法如下）
3   computed: {
4     ...mapState([
5       'appName'
6     ])
```

```

7   }
8   //mapState 最后会返回会是一个对象 , ... 展开运算符, 会把对象的数据扁平化
9
10      //然后就可以直接使用了
11      <p>{{ appName }}</p>
12
13 //也可以传对象 (传对象的写法如下)
14     computed: {
15       ...mapState({
16         appName: state => state.appName,
17         // 模块中的
18         userName: state => state.user.userName
19       })
20     }

```

2.3.1.2 开启命名空间 模块中 添加 namespace: true, 然后使用 createNamespacedHelpers

```

1   //使用createNamespacedHelpers
2   import { createNamespacedHelpers } from 'vuex'
3   const { mapState } = createNamespacedHelpers('user')
4   computed: {
5     ...mapState({
6       userName: state => state.userName
7     })
8   }

```

2.3.1.2 如果想要直接 使用 mapState 完成这种效果 需要在对象传入模块名 或者使用路径写法, mutation 之后在说 不太喜欢这种模式, 可读性较差 每次都要this.[路径] 不方便阅读

```

1   computed: {
2     ...mapState('user', {
3       userName: state => state.userName
4     })
5   }

```

2.3.2 getters

getters.js 中

```
1 const getters = {
2   // 传入state 代表同级别state
3   appNameWithVersion: (state)=>{
4     return `${state.appName}v2.0`
5   }
6 }
7
8 export default getters
```

```
1 //在组件计算属性中
2 computed: {
3   appNameWithVersion(){
4     //直接访问
5     return this.$store.getters.appNameWithVersion
6   }
7 }
8 //展示出来
9 <p>{{ appNameWithVersion }}</p>
10
11 //然后如果state的appName修改就会触发 getters, getters 相当于是全局的计算属性, computed 是局部的计算属性
```

当然也有 工具辅助函数 `mapGetters`

```
1   ...mapGetters([
2     'appNameWithVersion'
3   ])
4
5   //展示出来
6   <p>{{ appNameWithVersion }}</p>
```

```
1   //模块中的引用
```

```
2     ...mapGetters('user', [  
3       'firstLetter'  
4     ])
```

2.3.3 mutations

2.3.3.1 先提一下自定义属性

```
1   computed: {  
2     appName () {  
3       return this.$store.state.appName  
4     }  
5   },  
6  
7   methods: {  
8     handleChangeAppName(){  
9       // 不能直接这样修改 因为计算属性默认只有getter  
10      this.appName = 'newAppName'  
11    }  
12  }
```

会提示计算属性被定义但是他没有setters

因为计算属性默认只有getter，读取计算属性的时候会调用getter，设置的时候会调用 setter 所有会报错

拓展 可以这么写：（不用在意）

```
1   appName: {  
2     set: function(newValue){  
3       this.inputValue = newValue + 'sd'  
4     },  
5     get: function(){  
6       return this.inputValue + 'sdfsdf'  
7     }  
8   }
```

2.3.3.2 单值写法和后面参数是对象的写法

```
1 //组件中
2
3 //结构中
4 <template>
5   <div>
6     <h1>{{appName}}</h1>
7     <button @click="handleChangeAppName">修改state 里面的appName</b
      utton>
8   </div>
9 </template>
10
11   //计算属性中
12   computed: {
13     appName () {
14       return this.$store.state.appName
15     }
16   },
17   //方法中
18   methods: {
19     handleChangeAppName(){
20       // 第一个参数就是方法名， 第二个就是要传递的参数 ，也可以直接传对象过
      去，多值写法后面再写
21       this.$store.commit('SET_APP_NAME', 'newAppName')
22       //对象写法
23       //this.$store.commit('SET_APP_NAME', {
24       //  appName: 'newAppName',
25       // })
26     }
27   }
28
29
30 //mutations.js 中
31 const mutations = {
32   // 第一个参数就是同级的state， 第二个参数是载荷其实就是参数
33   SET_APP_NAME (state, params) {
```



```

34      // 如果是单个值直接 = params 如果是多个需要是对象， 当然也可以直接传对象
      过来
35      state.appName = parmas
36      //传的是对象的话 只改变某一项
37      //state.appName = params.appName
38  }
39 }
40
41 export default mutations

```

2.3.3.3 纯对象写法： 直接传递一整个对象

```

1  handleChangeAppName(){
2      // type 指定方法名， appName 是参数
3      this.$store.commit({
4          type: 'SET_APP_NAME',
5          appName: 'newAppName',
6      })

```

2.3.3.4 给state 添加一个 版本号的信息 使用vue 身上的set方法（后面解释为什么要用set）

```

1 //导入 vue
2 import Vue from 'vue'
3 const mutations = {
4     // 第一个参数就是同级的state， 第二个参数是载荷其实就是参数
5     SET_APP_NAME (state, params) {
6         state.appName = params.appName
7     },
8     //定义一个SET_APP_VERSION 方法
9     SET_APP_VERSION (state) {
10         Vue.set(state, 'appVersion', 'v2.0')
11         // 错误的 会发现没有更新视图
12         state.appVersion = 'v2.0'
13     }
14 }

```

为什么要使用 set，使用 `tate.appVersion = 'v2.0'` 会发现没有更新视图，

根据vue响应式原则： 在创建state实例初期，定义的状态， 通过mutations 修改， 他会触发视图的更新，但是如果一开始没有， 在创建实例的时候不会添加set， get方法就不会触发视图的更新

所以使用set方法把新的值添加到state上，并且会添加上set， get方法，这样就会触发视图的更新

2.3.3.5 使用mapmutations 工具辅助函数

```
1  methods: {
2    //映射到当前组件
3    ...mapMutations(['SET_APP_NAME']),
4    //就可以直接用this使用
5    this.SET_APP_NAME('newAppName')
6  }
7
8
9  //对象的写法
10 methods: {
11   ...mapMutations(['SET_APP_NAME']),
12   this.SET_APP_NAME({
13     appName: 'newAppName',
14   })
15 }
16 //mutations.js 里面获取的方式记得改
17 SET_APP_NAME (state, params) {
18   state.appName = params
19 },
20
21 SET_APP_NAME (state, params) {
22   // 对象
23   state.appName = params.appName
24 },
```

2.3.4.6 还可以路径写法， 如果是拆分细模块且模块相互对应不推荐这样写每次 都this后面带路径， 还是传入模块名或者使用命名空间辅助函数

```
1 methods: {
2     ...mapmutations(['user/setUserName'])
3 }
4 //调用要这样写 模块我们进行了namespaced: true, 所以引用mapmutations时需要带上 user/, 并且在使用该方法时, 直接使用 this['user/setUserName']
5 this['user/setUserName'](this.UserName)
6 也可以直接触发 使用 $store 上的 dispatch 方法
```

2.3.3.6 模块中的话 开启和不开启命名空间的情况下（有注意点）

不开启命名空间的情况下,可以不用书写模块名, 因为vuex 会把mutations 和 actions 和 getters 和根级别的全部注册到当前

```
1 //不开启命名空间的情况下,可以不用书写模块名, 因为vuex 会把mutations
  和 actions 和getters 和根级别的全部注册到当前
2     ...mapMutations([
3     'SET_APP_NAME',
4     'SET_APP_USERNAME'
5     ]),
6     //开启命名空间的情况下, 必须要用, 传入模块的方式, 使用 命名空间辅助函
  数或者路径写法 (懒得写了跳过)
7     ...mapMutations('user',[
8     'SET_APP_NAME',
9     'SET_APP_USERNAME'
10    ]),
```

2.3.4 actions 做异步操作 然后触发mutations 之后修改state 里面的数据

1. 在src 的api文件夹下创建 app.js, 模拟请求 代码如下

```

1 export const getAppName = ()=> {
2   return new Promise((resolve, reject) => {
3     const err = null
4     setTimeout(() => {
5       if (!err) resolve({ code: 200, info:{ appName: 'newAppName'
6     }})
7       else reject(err)
8     }, timeout);
9   })
10 }

```

2. actions.js 中 把方法引用过来，并创建自己的方法

```

1 // action 做异步操作 不能直接在 mutations 做异步任务
2 import { getAppName } from '@api/app'
3 const actions = {
4
5   //注意参数是对象形式，commit是用于提交一个mutations。当然不只有commit
   这一种，
6   //commit 提交一个mutations，state 指代当前state实例，rootState代表
   根级别的state，
7   //dispatch 触发actions，如果同级还有一个actions 用 dispatch('xx
   x','xxx') 后面继续深入
8
9   updateAppName ({ commit }) {
10     getAppName().then(res => {
11       console.log(res)
12       //commit 触发mutations的方法和对应的属性并传递参数
13       commit('SET_APP_NAME', res.info.appName)
14       //利用解构赋值也可以写成
15       const { info: { appName }} = res
16       commit('SET_APP_NAME', appName)
17     }).catch(err => console.log(err))
18   }
19 }
20
21 export default actions

```

```

1 // action 做异步操作 不能直接在 mutations 做异步任务
2 import { getAppName } from '@api/app'
3 const actions = {
4   //使用async 和 await 优化
5   async updateAppName({ commit }) {
6     try {
7       const { info: { appName } } = await getAppName()
8       commit('SET_APP_NAME', appName)
9     } catch (error) {
10      console.log(error)
11    }
12  }
13 }
14
15 export default actions

```

3. store.vue 中使用辅助函数把actions 的方法映射到当前, 然后点击的时候触发

```

1 methods: {
2   ...mapActions([
3     'updateAppName'
4   ]),
5   //在点击事件的时候触发
6   handleChangeAppName(){
7     this.updateAppName()
8   },
9 }

```

4. 开启命名空间后的路径写法 (不太推荐)

```

1 methods: {
2   ...mapActions(['user/login'])
3 }
4 //调用要这样写 模块我们进行了namespaced: true, 所以引用action时需要带上 user/, 并且在使用该方法时, 直接使用 *this['user/login'] , 使用this.user/login 语法是错误的

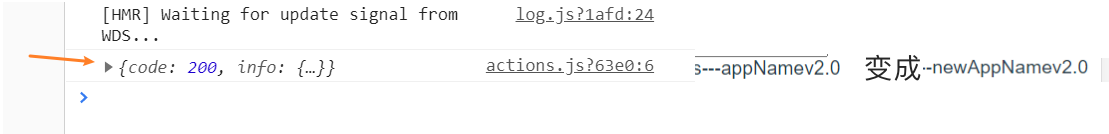
```

```
5 await this['user/login'](this.loginForm)
6 也可以直接触发 使用 $store 上的 dispatch 方法
```

5. 直接使用 \$store 身上的dispatch 方法

```
1 this.$store.dispatch('需要触发的action函数', 需要传递的参数)
```

效果如下



逻辑分析：app.js中封装了异步方法， actions 把方法导入， 进行正确和错误的处理， 正确的话使用 commit 触发mutations 里面的SET_APP_NAME方法， 并传入参数， stor.vue 中使用了辅助函数把方法映射到当前， 点击的时候触发

不在赘述写在模块里上面提到： 不开启命名空间的情况下,可以不用书写模块名， 因为vuex 会把 mutations 和 actions 和getters 和根级别的全部注册到当前。

2.3.5 module 模块

当应用变得非常复杂时，store 对象就有可能变得相当臃肿。Vuex 允许我们将 store 分割成模块（module）。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割

```
1 // 如果模块再嵌套模块， 并开启命名空间的话， 怎么使用方法 实例：
2 // 模块名/下一级模块名
3 ...mapActions('user/next', [
4   'updateAppName'
5 ])
```

动态注册模块（先写一个 普通版本的 后面书写一个 在模块中的）

```
1 <button @click="regiterModule">动态注册模块</button>
2 <p v-for="(li, index) in todoList" :key="index">{{ li }}</p>
3 // 计算属性中
```

```

4    ...mapState({
5      todoList: (state) => (state.todo ? state.todo.todoList : []
6    },
7    // 触发的方法中
8    registerModule() {
9      this.$store.registerModule("todo", {
10        state: {todoList: ["学习mutations","学习actions"]},
11      })

```

在模块中的

```

1    <button @click="registerModule">动态注册模块</button>
2    <p v-for="(li, index) in todoList" :key="index">{{ li }}</p>
3      // 计算属性中
4    ...mapState({
5      //改成访问 state.user.todo
6      todoList: (state) => (state.user.todo ? state.user.todo.to
7    doList : []),
8    },
9    // 触发的方法中
10   registerModule() {
11     this.$store.registerModule(['user', 'todo'], {
12       state: {todoList: ["学习mutations","学习actions"]},

```

我们调用的是Vuex中子模块的action，该模块我们进行了namespaced: true，所以引用action时需要带上 `user/`，并且在使用该方法时，直接使用 `this['user/login']`，使用 `this.user/login` 语法是错误的

总结

vuex 五种状态 State , Getters, Mutations , Actions , Modules

辅助函数: mapState、mapActions、mapMutations, mapGetters ,
createNamespacedHelpers (命名空间辅助函数)

`{ commit, state, rootState, dispatch }` : `commit` 提交一个mutations, `state` 指代当前state实例, `rootState`代表根级别的state, `dispatch` 触发actions, 如果同级还有一个actions 用 `dispatch('xxx','xxx')`

```
1 const actions = {
2   updateAppName ( { commit, state, rootState, dispatch } ) {
3   }
4 }
```

`state` => 基本数据

`getters` => 从基本数据派生的数据

`mutations` => 提交更改数据的方法, 同步!

`actions` => 像一个装饰器, 包裹mutations, 使之可以异步。

`modules` => 模块化Vuex

2.4vuex 进阶篇

制定一个持久化存储的插件, 其实就是一个函数

1. 创建 plugin文件夹 里面创建 `saveInLocal.js` 然后书写如下代码

```
1 // 定义一个插件, 其实就是一个函数, 只有一个参数就是 store
2 export default store => {
3   // 刷新浏览器 第一次就做的操作写在这里, 使用store.subscribe() 两个参数分
   别是mutation 和 state
4   // 如果有 localStorage.state 这个字段, 说明配置过了, 怎么替换掉实例中得s
   tate 不可以直接store.state = ''这样不行需要使用store提供的replaceState方
   法
5   // 把字符串转换成对象, 替换掉store里面的state
6   if (localStorage.state) store.replaceState(JSON.parse(localStorage.state))
7   store.subscribe((mutation, state) => {
8     // 每次提交mutation 的时候都会触发这里
9     // console.log('提交了mutation 修改state');
10    // 进行一下本地存储
11    localStorage.state = JSON.stringify(state)
12  })
13 }
```


在store 的跟文件， index.js 中导入 并 加载

```

1 import savueLocation from './plugin/saveInLocal'
2 export default new Vuex.Store({
3   state,
4   mutations,
5   actions,
6   getters,
7   modules: {
8     user
9   },
10  // 加载插件：注意不要漏掉s，   plugins 就是插件的意思   把我们创建的插件放进
    去，当然要先import 导入它
11  plugins: [ saveLocation ]
12 })

```

2.4.1 vuex 严格模式 strict: true

```

1 export default new Vuex.Store({
2   //strict 为true开启严格模式， 为false 不开启
3   strict: true,
4   state,
5   mutations,
6   actions,
7   getters,
8   modules: {
9     user
10  },
11  plugins: [ saveLocation ]
12 })

```

在严格模式下，无论何时发生了状态变更且不是由 mutation 函数引起的，将会抛出错误。这能保证所有的状态变更都能被调试工具跟踪到。

~~~~~跳过 表单处理查阅vuex 文档

# 3. Ajax 阶段（解决跨域问题，封装axios ， 开发实战中使用）

## 3 .1 解决跨域问题

| URL                                                      | 说明                  | 是否允许通信                   |
|----------------------------------------------------------|---------------------|--------------------------|
| http://www.d.com/d.js<br>http://www.d.com/w.js           | 同一域名下               | 允许                       |
| http://www.d.com/lab/a.js<br>http://www.d.com/src/b.js   | 同一域名下不同文件夹          | 允许                       |
| http://www.d.com:3333/a.js<br>http://www.d.com:4444/b.js | 同一域名不同端口            | 不允许                      |
| http://www.d.com/a.js<br>http://46.33.22.44/b.js         | 域名和域名对应 IP          | 不允许                      |
| http://www.d.com/a.js<br>http://script.d.com/b.js        | 主域相同，子域不同           | 不允许                      |
| http://www.d.com/a.js<br>http://d.com/b.js               | 同一域名，不同二级域名<br>(同上) | 不允许 (cookie 这种情况下也不允许访问) |
| http://www.d.com/a.js<br>http://www.v.com/b.js           | 不同域名                | 不允许                      |

1. 在开发中，通常开发的时候配置devServer.proxy，实现请求代理，避免跨域。更具体的可以看webpack的devServer配置。

```
1 // 在本地开发的时候开启一个开发环境，然后本地启一个node服务， 端口肯定不一样的，
2 // 同一域名不同端口会出现跨域问题，可以在这里配置一个代理， 会被代理到同一个端口下，
3 // 相当于是在配置的域下进行的请求，这样就没有跨域问题了
4 module.exports = {
5   devServer: {
6     // 配置反向代理 这里配置只是开发环境的跨域，是只存在这个电脑上的， 上线后是部署环境的跨域
7     proxy: {
8       // 这里的api 表示如果我们的请求地址有/api的时候,就出触发代理机制
9       // localhost:8888/api/abc => 代理给另一个服务器
10      // 本地的前端 => 本地的后端 => 代理我们向另一个服务器发请求 （行得通）
11      // 本地的前端 => 另外一个服务器发请求 （跨域 行不通）
12      // 当我们的本地的请求 有/api的时候，就会代理我们的请求地址向另外一个服
```

务器发出请求

```
13     '/api': {  
14         target: 'http://adm-java.test.net/', // 跨域请求的地址（按照  
            自己的接口文档来配置这个请求的地址）  
15         changeOrigin: true // 只有这个值为true的情况下 才表示开启跨域  
16     }  
17 },  
18 },  
19 }
```

如果前端没有设置基础路径会 自动拼接上localhost:8080

```
✖ GET http://localhost:8080/getUserInfo 404 (Not Found) xhr.js?b50d:177  
✖ Uncaught (in promise) Error: Request failed createError.js?2d83:16  
  with status code 404  
    at createError (createError.js?2d83:16)  
    at settle (settle.js?467f:17)  
    at XMLHttpRequest.handleLoad (xhr.js?b50d:62)
```

## 2. 后端进行配置

允许访问的域，允许访问的请求头，允许访问的方法，配置可这三个字段前端就不需要配置代理也可以访问

```
res.header('Access-Control-Allow-Origin', '*')  
res.header('Access-Control-Allow-Headers', 'X-Requested-With,Content-Type')  
res.header('Access-Control-Allow-Methods', 'PUT,POST,GET,DELETE,OPTIONS')
```