

vue续

vuecli3创建项目和目录结构

vue2.5.21 -> vue2.x -> flow-type(facebook)

Vue3.x -> TypeScript(microsoft)

认识Vue CLI3

- 今年8月份刚刚发布



- vue-cli 3 与 2 版本有很大区别

- vue-cli 3 是基于 webpack 4 打造，vue-cli 2 还是 webpack 3
- vue-cli 3 的设计原则是 “0配置”，移除的配置文件根目录下的，build和config等目录
- vue-cli 3 提供了 vue ui 命令，提供了可视化配置，更加人性化
- 移除了static文件夹，新增了public文件夹，并且index.html移动到public中

vue-cli3是基于 **webpack4**打造，vue-cli2还是 **webpack3**

vue-cli3的设计原则是“0配置”，**移除的配置文件根目录下的build和 config等目录**，开发者从目录里面看不到相关的配置文件

vue-cli3**提供了 vue ui 命令，提供了可视化配置，更加人性化**

移除了 static文件夹，新增了 public文件夹（此文件夹功能和static一样，同样是静态文件不作处理直接放到dist文件夹中），并且 **index. html移动到 public中**

创建vuecli3命令行：**vue create 项目名称**

Vue CLI v3.0.4
? Please pick a preset: (Use arrow keys)
> coderwhy (babel) 1.选择配置方式
default (babel, eslint) 默认配置还是手动配置
Manually select features

? Check the features needed for your project: (Press <space> to toggle all, <ctrl> to invert selection)
> Babel
o TypeScript
o Progressive Web App (PWA) Support
o Router
o Vuex
o CSS Pre-processors
● Linter / Formatter
o Unit Testing
o E2E Testing

made by coderwhy
微博: coderwhy

? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? w keys)
> In dedicated config files 3.对应的配置单独生成文件还是放在package.json
In package.json

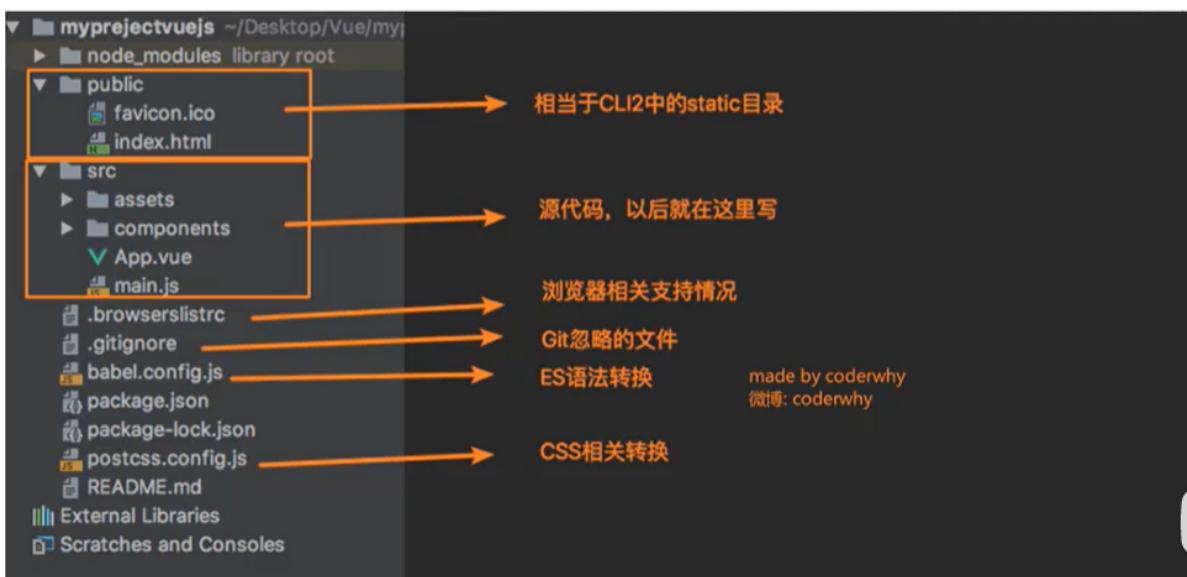
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? w keys)
> Save this as a preset for future projects? (y/N) 4.要不要把刚才自己选择配置保存下来

? Save this as a preset for future projects? Yes
? Save preset as: mypreset

Vue CLI v3.0.4
> Creating project in /Users/xmg/Desktop/Vue/myprojectvuejs.
Initializing git repository...
o Installing CLI plugins. This might take a while...

保存自己的配置作为预设后，如果想删除，在c盘users中administrator中.vuerc文件夹中删除

(很多文件名结尾都是rc: run command里面都是和终端命令相关的)



打开main.js文件

```
package.json × main.js × index.html ×
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 Vue.config.productionTip = false
5
6 new Vue({
7   render: function (h) {
8     return h(App)
9   }
10 }).$mount('#app')
```

Vue.config.productionTip = false，意思是项目构建过程中，会给出构建过程中的一些提示信息，提示构建了什么东西，开发阶段一般是false

跟cli2中不同的是，vue实例中并没有用el来挂载相应的标签id值，而是使用了\$mount来挂载，实现的功能是一样的，通过el来挂载源码和mount是一样的

vuecli3配置文件的查看和修改

修改配置有三种方案

①

启动vue ui服务器（用edge打开，谷歌打开报错）

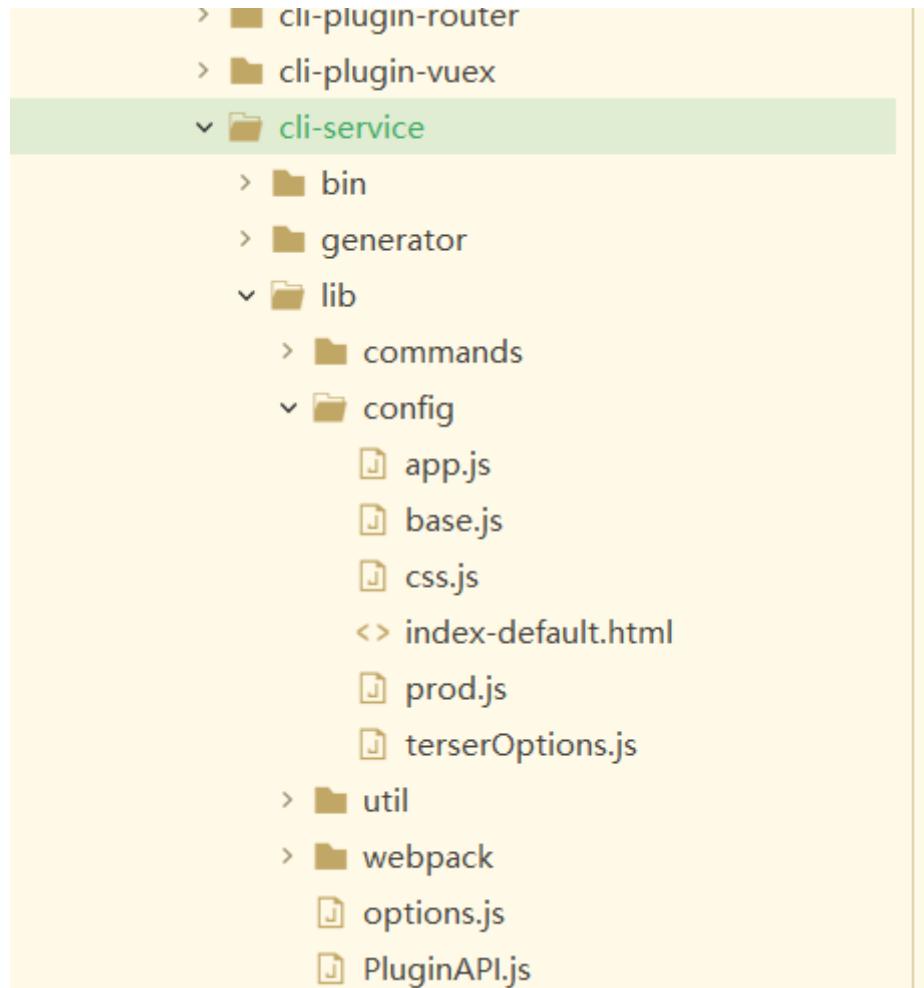
在任何一个目录下面的命令行呼入输入：**vue ui**

不用指定目录，因为启动的是一个服务器，并不是专门顶峰已哪一个目录下面的文件



②

之前说vuecli3设计原则是0配置，根目录下的build、config文件夹被移除，但是可以在项目的**node_modules->@vue->cli-service**里面有一些具体的配置



③

如果真的想修改vue的具体配置，可以自己添加一个文件放在项目的根目录下面

文件的名字是**固定的**: **vue.config.js**，在文件中，将配置用**module.exports导出**的方法进行导出

到时候vue项目会自动在文件中找名为vue.config.js的配置文件，导入到上面提到的cli-service文件夹中的相关配置文件中，**进行合并**

```
module.exports = {}
```

路由

路由简介



什么是路由？

■ 说起路由你想起了什么？

□ 路由是一个网络工程里面的术语。

□ **路由 (routing)** 就是通过互联的网络把信息从源地址传输到目的地址的活动。--- 维基百科

■ 额，啥玩意？没听懂

□ 在生活中，我们有没有听说过路由的概念呢？当然了，路由器嘛。

□ 路由器是做什么的？你有想过吗？

□ 路由器提供了两种机制：路由和转送。

✓ 路由是决定数据包从**来源到目的地**的路径。

✓ 转送将**输入端**的数据转移到合适的**输出端**。

□ 路由中有一个非常重要的概念叫路由表。

✓ 路由表本质上就是一个映射表，决定了数据包的指向。

前端渲染后端渲染/前端路由后端路由

后端路由阶段

早期的网站开发整个HTML页面是由**服务器来渲染**的

服务器通过**jsp或者Java等等直接生产渲染好对应的HTML页面**，返回给客户端进行展示

但是，一个网站，这么多页面服务器如何处理呢？

一个页面有自己对应的网址，也就是**URL**。

URL请求会发送到服务器，服务器会通过**正则**对该URL进行匹配，并且最后交给一个 Controller 进行处理

Controller进行各种处理最终生成HTML或者数据，返回给前端，这就完成了一个IO操作

网页和url这种映射关系由后端处理和保存，这就是**后端路由**：

当我们页面中需要请求不同的路径内容时，交给服务器来进行处理，服务器渲染好整个页面，并且将页面返回给客户端

这种情况下渲染好的页面，不需要单独加载任何的js和cs可以直接交给浏览器展示，这样也有利于SEO的优化

后端路由的缺点

一种情况是整个页面的模块由后端人员来编写和维护的

另一种情况是前端开发人员如果要开发页面，需要通过PHP和Java等语言来编写页面代码

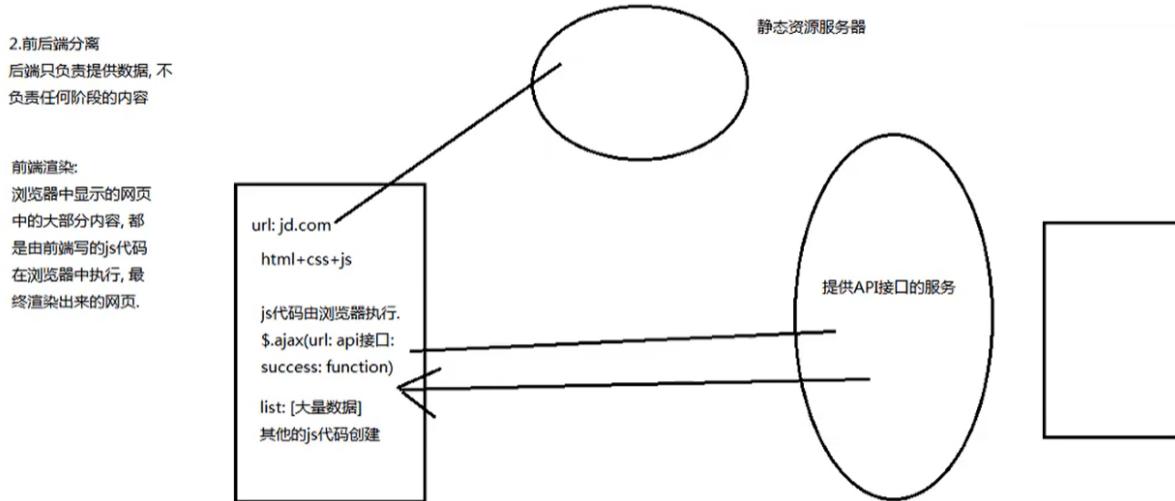
而且通常情况下HTML代码和数据以及对应的逻辑会混在一起，编写和维护都是非常糟糕的事情。

前后端分离阶段

随着[ajax](#)的出现，有了前后端分离的开发模式

后端只提供API来返回数据，[前端通过ajax获取数据](#)，并且可以通过Javascript将数据渲染到页面中
这样做最大的优点就是前后端责任的清晰，[后端专注于数据上，前端专注于交互和可视化上](#)
[并且当移动端\(iOS/ Android\)出现后，后端不需要进行任何处理依然使用之前的一套API即可。](#)

目前很多的网站依然采用这种模式开发。

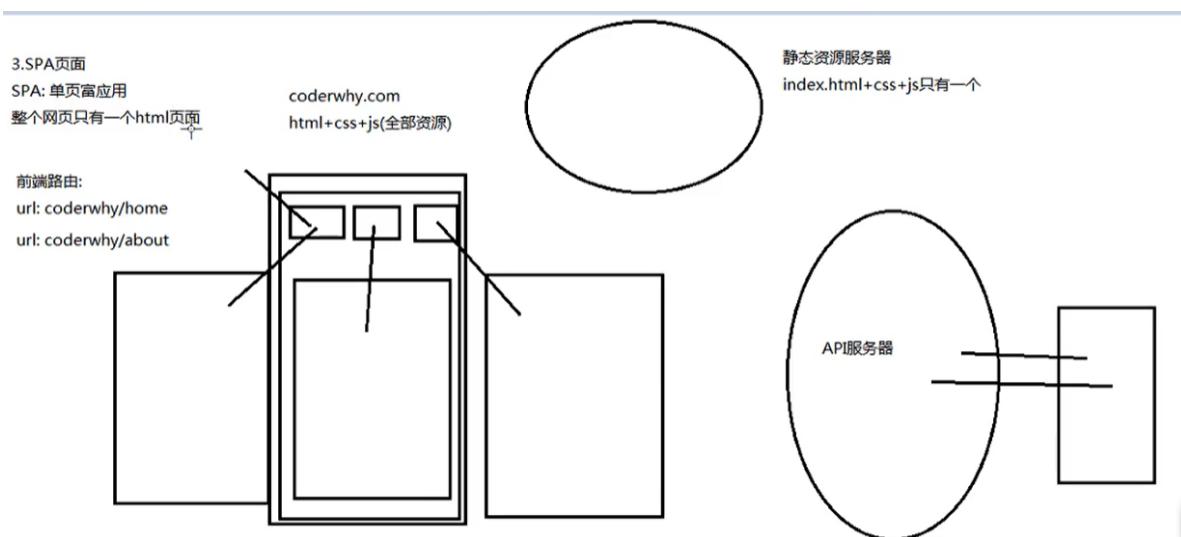


前端路由阶段

spa: 单页面富应用，整个网页只有一个页面

其实SPA最主要的特点就是在[前后端分离的基础上加了一层前端路由](#)

也就是[前端来维护一套路由规则](#)



前端路由阶段和前后端分离阶段不同点在于

前端同样是从静态资源服务器上请求html、css、js，不同的是，[前端路由阶段html+css+js只有一套](#)

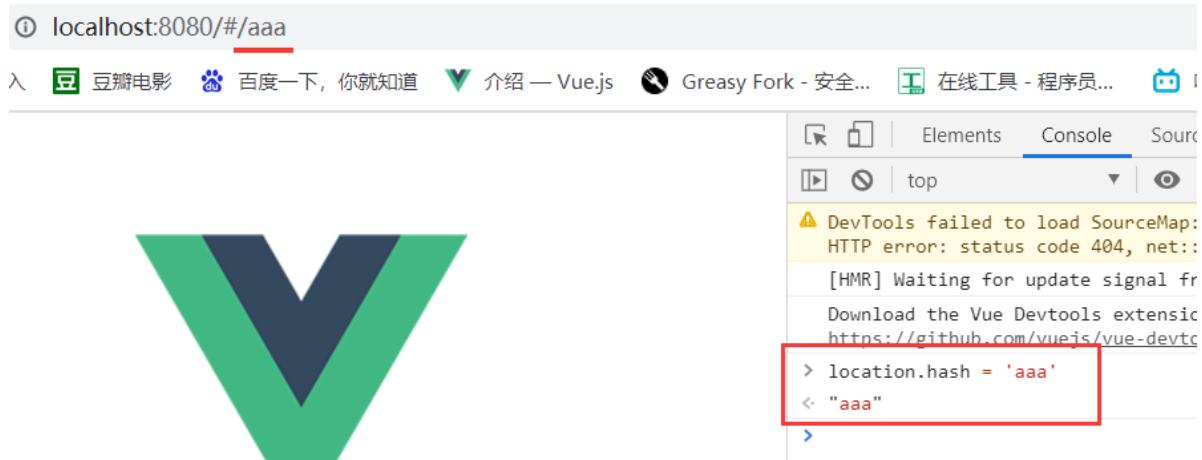
浏览器一次性从服务器上下载下来所有的资源，浏览器再次发起当前网站的不同url时，[就不会向服务器请求数据了](#)，而是通过[前端路由](#)来调用不同的页面（在vue中就是一个个vue组件）

前端路由的核心是：改变url，页面不会进行整体的刷新，不会重新向服务器发出请求

url的hash和html5的history

url的hash

以vue的主页为例，在控制台通过输入：location.hash = 'aaa'将网页的url进行修改，在URL后面加上一条记录，但是网页并不会进行刷新



通过查看network中加载项，发现并没有进行新的内容请求，即网页并没有刷新

html5的history

pushState()、replaceState()、go()、forward()这四个方法都属于js中的BOM

后面三个在之前讲过

pushState()与replaceState()对比详解：<https://blog.csdn.net/yexudengzhidao/article/details/101448168>

history.pushState()方法

用于修改url，同样是在URL后面加上一条记录

该方法接受三个参数，依次为：

state：一个与添加的记录相关联的**状态对象**，主要用于**popstate事件**。该事件触发时，该对象会传入回调函数。也就是说，浏览器会将这个对象序列化以后保留在本地，重新载入这个页面的时候，可以拿到这个对象。如果不需要这个对象，此处可以填null。

title：新页面的标题。但是，现在所有浏览器都忽视这个参数，所以这里可以填空字符串。

url：新的网址，必须与当前页面**处在同一个域**。浏览器的地址栏将显示这个网址。

```
> history.pushState({}, '', 'home')
< undefined
> history.pushState({}, '', 'me')
< undefined
> history.pushState({}, '', 'demo')
```

pushState()方法的原理是每修改一次url就将当前的url参数进行入栈，多次调用就多次入栈

使用**history.back()**方法会将浏览器的页面进行回退，同时进行**出栈操作**，url会按照pushState方法入栈的url参数，先进后出

history.replaceState()方法

History.replaceState()方法用来修改 History 对象的当前记录，而**不是对url参数值进行入栈操作**，**是直接将url进行替换**，并且**不能回退**

其他都与pushState()方法一模一样。

go()方法

它需要一个整数作为参数：

- 1：表示向前跳转一个页面相当于 forward()
- 2：表示向前跳转两个页面
- 1：表示向后跳转一个页面
- 2：表示向后跳转两个页面

forward()方法

可以跳转下一个页面，作用和浏览器的前进按钮一样



- 目前前端流行的三大框架，都有自己的路由实现：
 - Angular的ngRouter
 - React的ReactRouter
 - Vue的vue-router
- 当然，我们的重点是vue-router
 - vue-router是Vue.js官方的路由插件，它和Vue.js是深度集成的，适合用于构建单页面应用。
 - 我们可以访问其官方网站对其进行学习：<https://router.vuejs.org/zh/>

vue-router的安装与配置



认识vue-router

- 目前前端流行的三大框架, 都有自己的路由实现:
 - Angular的ngRouter
 - React的ReactRouter
 - Vue的vue-router
- 当然, 我们的重点是vue-router
 - vue-router是Vue.js官方的路由插件, 它和vue.js是深度集成的, 适合用于构建单页面应用。
 - 我们可以访问其官方网站对其进行学习: <https://router.vuejs.org/zh/>
- vue-router是基于路由和组件的
 - 路由用于设定访问路径, 将路径和组件映射起来.
 - 在vue-router的单页面应用中, 页面的路径的改变就是组件的切换.



安装和使用vue-router

- 因为我们已经学习了webpack, 后续开发中我们主要是通过工程化的方式进行开发的.
 - 所以在后续, 我们直接使用npm来安装路由即可.
 - 步骤一: 安装vue-router
 - npm install vue-router --save
 - 步骤二: 在模块化工程中使用它(因为是一个插件, 所以可以通过Vue.use()来安装路由功能)
 - 第一步: 导入路由对象, 并且调用 **Vue.use(VueRouter)**
 - 第二步: 创建**路由实例**, 并且传入路由**映射配置**
 - 第三步: 在**Vue实例**中**挂载**创建的**路由实例**

步骤一: 安装vue-router

命令行: **npm install vue-router--save**

步骤二: 在模块化工程中使用它(因为是一个插件, 所以可以**通过Vue.use()来安装路由功能**)

第一步: **导入**路由对象, 并且调用**Vue.use(Vue Router)**

第二步: **创建****路由实例**, 并且传入**路由映射配置**

第三步: 在**Vue实例**中**挂载**创建的**路由实例**

路由的配置:

```

package.json      index.js      main.js

1 // 配置路由的相关信息
2 import Vue from 'vue' // 导入vue对象
3 import Router from 'vue-router' // 导入vue路由模块中的Router对象
4 import HelloWorld from '@/components/HelloWorld' // 导入组件
5
6 Vue.use(Router) // 通过Vue.use(插件)来安装插件，以后的插件都要进行类似的安装
7
8 // 创建vuerouter对象: routers，并且用export default导出
9 export default new Router({
10   // 配置路由与组件的映射关系，数组中的一个对象是一个映射关系
11   routes: [
12     {
13       path: '/',
14       name: 'HelloWorld',
15       component: HelloWorld
16     }
17   ]
18 })

```

vue实例中进行挂载：

```

package.json      index.js      main.js

1 import Vue from 'vue'
2 import App from './App'
3 /* 导入router数组，将路由与组件映射关系导入
4 这里地址写的时./router，因导入的时候只写文件夹的名字的话
5 会默认从指定文件夹中找到index.js文件，从该文件中进行导入
6 所以index.js可以省略*/
7 import router from './router'
8
9 Vue.config.productionTip = false
10
11 /* eslint-disable no-new */
12 new Vue({
13   el: '#app',
14   // 将路由对象中的映射关系挂载过来
15   router,
16   render: h => h(App)
17 })

```

以上这些步骤在安装脚手架时选择安装路由的话会默认安装好，**到时候只需要配置路由与组件之间的映射关系就行了**

路由通过< router-link>和< router-view>进行跳转

使用vue-router的步骤

第一步：创建路由组件，即编写vue文件，创建组件（因为一个页面就是一个组件）

第二步：配置路由映射：组件和路径映射关系

第三步：使用路由：通过< router-link>和< router-view>

下面是具体如何写两个路由组件并且显示在页面上

例子：

在router文件中的index.js文件中重新配置两个路由与组件的对应关系

```
About.vue      Home.vue      App.vue      main.js      index.js      index.html      pack
1 // 配置路由的相关信息
2 import Vue from 'vue'// 导入vue对象
3 import Router from 'vue-router'// 导入vue路由模块中的Router对象
4
5 import Home from '../components/Home.vue'// 导入组件
6 import About from '../components/About.vue'
7
8 Vue.use(Router)// 通过Vue.use(插件)来安装插件，以后的插件都要进行类似的安装
9
10 // 创建vuerouter对象: routers，并且用export default导出
11 export default new Router({
12   // 配置路由与组件的映射关系，数组中的一个对象是一个映射关系
13   routes: [
14     {
15       /* 这里写path，并不能写url，url是一个完整的路径
16        而这里只是一个相对的路径*/
17       path: '/About',
18       component: About
19     },
20     {
21       path: '/Home',
22       component: Home
23     },
24   ],
25 })
26 })
```

两个组件放在components文件夹中，组件具体内容如下

The screenshot shows a code editor with several tabs at the top: About.vue (selected), Home.vue, App.vue, main.js, and index.html. On the left, there's a file tree for a Vue project:

- HTML教程
- vue
- webpack
- vuecli
 - learnvuerouter
 - build
 - config
 - node_modules
 - src
 - assets
 - components
 - About.vue
 - Home.vue
 - router
 - index.js
 - App.vue
 - main.js
 - static
 - .babelrc

The code for About.vue is displayed in the editor:

```
1 <template>
2   <div>
3     <h3>我是About</h3>
4     <p>hahah哈哈哈哈ahahahahaha</p>
5   </div>
6 </template>
7
8 <script>
9   export default{
10     name: 'About',
11   }
12 </script>
13 <style scoped>
14
15 </style>
16
17 </style>
18
```

首页显示的组件为App.vue，在这个组件中引入创建的两个路由

用**router-link标签**来引入（被渲染成一个a标签，查看网页代码就是一个a标签），用**to属性**来标识路由的名称

用**router-view标签**表示路由放置的位置，放在那里就在哪里像是路由组件，相当于站位

About.vue Home.vue App.vue main.js

```
1 <template>
2   <div id="app">
3     <router-link to="/Home">
4       <button>首页</button>
5     </router-link>
6     <router-link to="/About">
7       <button>关于</button>
8     </router-link>
9     <router-view></router-view>
10   </div>
11 </template>
12 <script>
13   export default{
14     name: 'App'
15   }
16 </script>
17
18 <style>
19
20 </style>
```



创建router实例

The screenshot shows the VS Code interface with the project structure and code editor. The project structure on the left includes files like .editorconfig, .gitignore, package.json, package-lock.json, README.md, and several configuration and build scripts. The code editor on the right contains the following code:

```
import Vue from 'vue'
import VueRouter from 'vue-router'

// 1.注入插件
Vue.use(VueRouter)

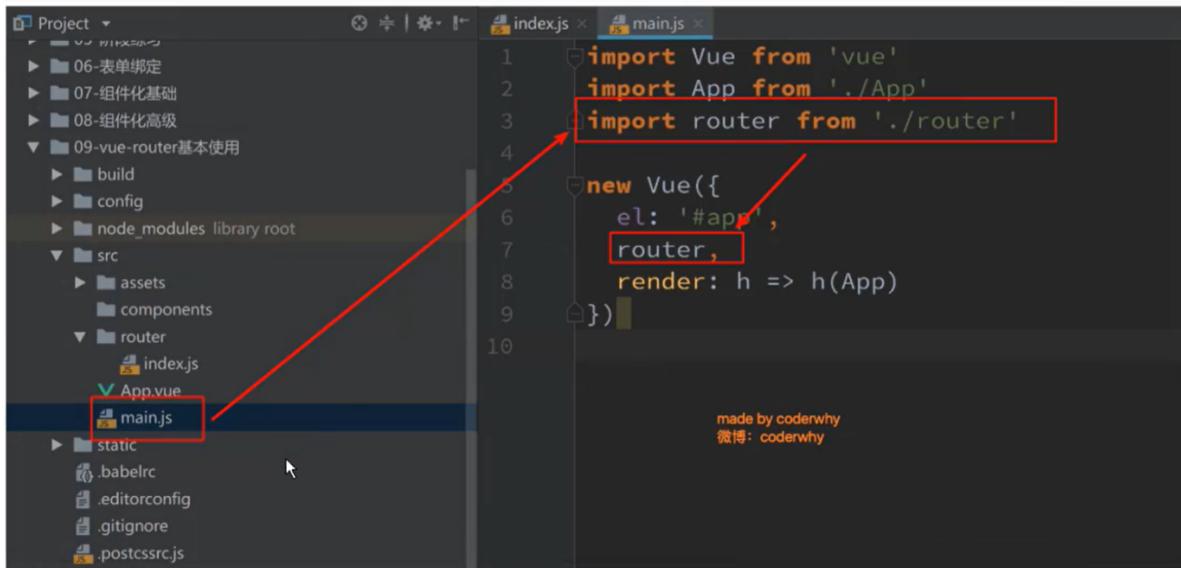
// 2.定义路由
const routes = []

// 3.创建router实例
const router = new VueRouter({
  routes
})

// 4.导出router实例
export default router
```

At the bottom of the editor, there is a note: "made by coderwhy" and "微博: coderwhy".

小码哥教育 挂载到Vue实例中

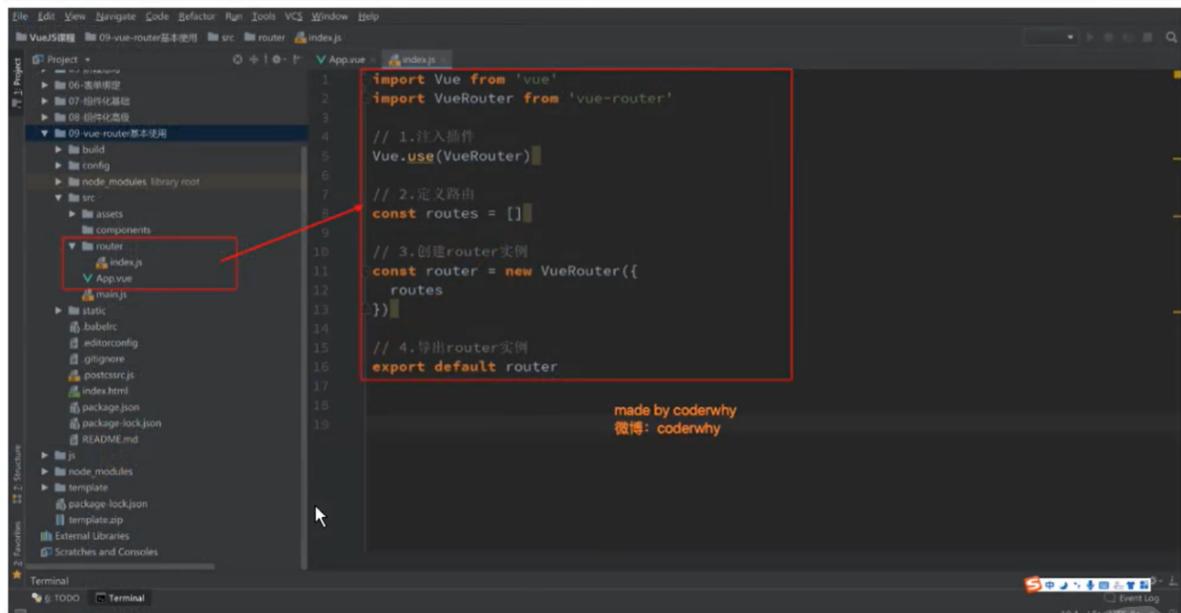


The screenshot shows the project structure on the left with files like index.js, main.js, App.vue, and router/index.js. The main.js file is open in the editor. A red box highlights the line `import router from './router'` and another red box highlights the `router` parameter in the constructor of the new Vue object.

```
index.js x main.js x
1 import Vue from 'vue'
2 import App from './App'
3 import router from './router'
4
5 new Vue({
6   el: '#app',
7   router,
8   render: h => h(App)
9 })
10
```

made by coderwhy
微博: coderwhy

小码哥教育 创建router实例

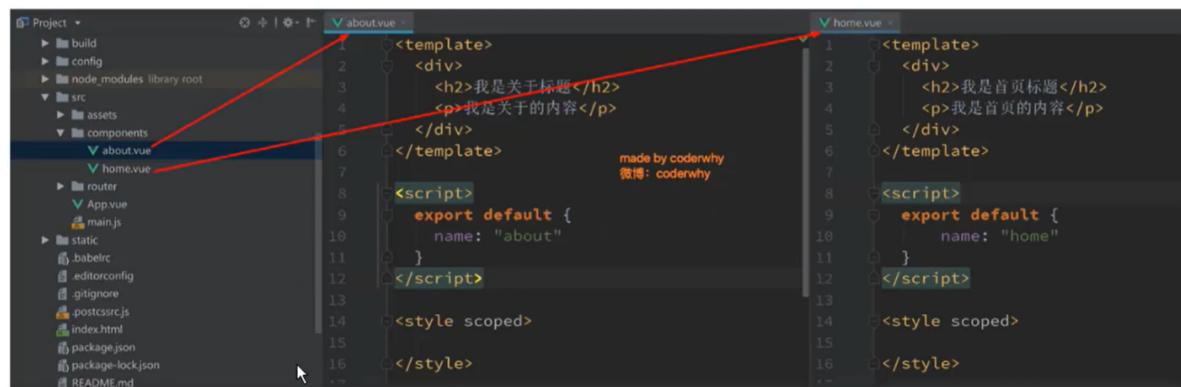


The screenshot shows the project structure on the left with files like index.js, main.js, App.vue, and router/index.js. The router/index.js file is open in the editor. A red box highlights the `main.js` file in the project tree and another red box highlights the code block containing the creation and export of the router instance.

```
index.js
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3
4 // 1.注入插件
5 Vue.use(VueRouter)
6
7 // 2.定义路由
8 const routes = []
9
10 // 3.创建router实例
11 const router = new VueRouter({
12   routes
13 })
14
15 // 4.导出router实例
16 export default router
```

made by coderwhy
微博: coderwhy

步骤一：创建路由组件



The screenshot shows the project structure on the left with files like index.js, main.js, App.vue, router/index.js, and components/about.vue and components/home.vue. The about.vue and home.vue files are open in the editor. A red box highlights the `about.vue` file in the project tree and another red box highlights the code block containing the component definition for 'about'.

```
about.vue
<template>
  <div>
    <h2>我是关于标题</h2>
    <p>我是关于的内容</p>
  </div>
</template>
<script>
  export default {
    name: "about"
  }
</script>
<style scoped>
</style>
```

```
home.vue
<template>
  <div>
    <h2>我是首页标题</h2>
    <p>我是首页的内容</p>
  </div>
</template>
<script>
  export default {
    name: "home"
  }
</script>
<style scoped>
</style>
```

步骤二：配置组件和路径的映射关系

The screenshot shows a code editor with a file structure on the left and the content of `index.js` on the right. A red box highlights the import statements for `Home` and `About` components. Another red box highlights the route definitions in the `routes` array. A third red box highlights the creation of the `router` instance.

```
import Vue from 'vue'
import VueRouter from 'vue-router'

import Home from '../components/home'
import About from '../components/about'

// 1. 注入插件
Vue.use(VueRouter)

// 2. 定义路由
const routes = [
  {
    path: '/home',
    component: Home
  },
  {
    path: '/about',
    component: About
  }
]

// 3. 创建router实例
const router = new VueRouter({
```



步骤三：使用路由.

The screenshot shows a code editor with a file structure on the left and the content of `App.vue` on the right. A red box highlights the `<router-link>` and `<router-view>` components used in the template.

```
<template>
  <div id="app">
    <h1>我是网站的标题</h1>
    <router-link to="/home">首页</router-link>
    <router-link to="/about">关于</router-link>
    <router-view></router-view>
    <h2>我是APP中一些底部版权信息</h2>
  </div>
</template>
<script>
  export default {
    name: 'App',
    components: {
    }
  }
</script>
<style>
```

- `<router-link>`: 该标签是一个 vue-router 中已经内置的组件, 它会被渲染成一个 `<a>` 标签.
- `<router-view>`: 该标签会根据当前的路径, 动态渲染出不同的组件.
 - 网页的其他内容, 比如顶部的标题/导航, 或者底部的一些版权信息等会和 `<router-view>` 处于同一个等级.
 - 在路由切换时, 切换的是 `<router-view>` 挂载的组件, 其他内容不会发生改变.

路由的默认值和修改为history



路由的默认路径

■ 我们这里还有一个不太好的实现:

□ 默认情况下, 进入网站的首页, 我们希望<router-view>渲染首页的内容.

□ 但是我们的实现中, 默认没有显示首页组件, 必须让用户点击才可以.

■ 如何可以让路径默认跳到到首页, 并且<router-view>渲染首页组件呢?

□ 非常简单, 我们只需要配置多配置一个映射就可以了.

```
const routes = [
  {
    path: '/',
    redirect: '/home'
  },
]
```

■ 配置解析:

□ 我们在routes中又配置了一个映射.

□ path配置的是根路径: /

□ redirect是重定向, 也就是我们将根路径重定向到/home的路径下, 这样就可以得到我们想要的结果了.

为主页设置一个默认显示的路由组件

在index.js文件中的创建vuerouter对象的地方

再添加一个路由与组件的映射关系

使用redirect睡醒进行重定向, 当相对地址path为空时, 将地址重定向到Home

```
nvuerouter > src > router > index.js          | 🔍 输入文件名
About.vue      Home.vue      App.vue      main.js      * index.js      index.html | le
7
8 Vue.use(Router)// 通过Vue.use(插件)来安装插件, 以后的插件都要进行类似的安
9
10 // 创建vuerouter对象: routers, 并且用export default导出
11 export default new Router({
12   // 配置路由与组件的映射关系, 数组中的一个对象是一个映射关系
13   routes: [
14     {
15       path: '',
16       // 重定向
17       redirect: '/Home'
18     },
19   ],
20   /*
21   * 这里写path, 并不能写url, url是一个完整的路径
22   * 而这里只是一个相对的路径*/
23   path: '/About'
```

以上对路径的修改默认都是通过url的hash值来进行修改的, 总是地址的地方总是有一个#号

localhost:8080/#/Home

如果想改为history模式

在index.js文件中创建vuerouter对象的地方

之前只有routers数组这一个属性

现在在传一个**mode属性**, 将mod属性命名为history即可

```
10 // 创建vuerouter对象: routers, 并且用export default导出
11 export default new Router({
12   // 配置路由与组件的映射关系, 数组中的一个对象是一个映射关系
13   routes: [
14     {
15       path: '',
16       // 重定向
17       redirect: '/Home'
18     },
19     {
20       /* 这里写path, 并不能写url, url是一个完整的路径
21        而这里只是一个相对的路径*/
22       path: '/About',
23       component: About
24     },
25     {
26       path: '/Home',
27       component: Home
28     }
29   ],
30   mode: 'history' // mode属性，将mod属性命名为history即可
31 }
32 )
33 }
```



我是首页

hahahahahahahahahah

router-link的其他属性的补充

tag属性: tag可以指定<router-link>之后渲染成什么组件, 即使查看网页源码也会显示被渲染的标签, 比如上面的代码会被渲染成一个

- 元素, 而不是

```
vue X
<template>
  <div id="app">
    <h2>我是APP组件</h2>
    <routerr-link to="/home" tag="button">首页</routerr-link>
    <routerr-link to="/about" tag="button">关于</routerr-link>
    <routerr-view></routerr-view>
  </div>
</template>
```



replace属性: replace不会留下 history记录，所以指定 replace的情况下，**后退键返回不能返回到上一个页面中**，该属性不用赋值

```
template>
<div id="app">
  <h2>我是APP组件</h2>
  <routerr-link to="/home" tag="button" replace>首页</routerr-link>
  <routerr-link to="/about" tag="button">关于</routerr-link>
  <routerr-view></routerr-view>
</div>
</template>
```

active-clas属性: 当`<routerr-link>`对应的路由匹配成功时，**会自动给当前元素设置一个 routerr-link-active 的 class**

此特性可以在动态为标签添加样式的时候使用，方便选择器使用

设置 active-clas属性可以**修改属性默认的名称**

The screenshot shows the browser's developer tools inspecting the DOM. It highlights a `<button>` element with a red box. The element has the class `router-link-exact-active router-link-active`. The DOM tree shows the button is part of the `<div id="app">` component, which contains an `<h2>` element with the text "我是APP组件". The browser's address bar shows the URL `/home`, indicating the route is matched.

```
<div id="app">
  <h2>我是APP组件</h2>
  <router-link to="/home" tag="button" replace active-class="active">首页
  </router-link>
  <router-link to="/about" tag="button" replace active-class="active">关于
  </router-link>
  <router-view></router-view>
</div>
</template>

<script>
export default {
  name: 'App'
}
</script>

<style>
  .router-link-active {
    color: #f00;
  }
</style>
```

在进行高亮显示的导航菜单或者底部 tabbar 时，会使用到该类。

但是通常不会修改类的属性，会直接使用默认的 router-link-active

如果真想改的话，可以直接在创建 vuerouter 对象的地方再添加一个属性：linkActiveClass

属性的值就是我们指定的点击时添加在标签上的属性名

```
erouter  src  router  index.js  LS  搜索文件夹
About.vue      Home.vue      App.vue      main.js      * index.js      index.html |
```

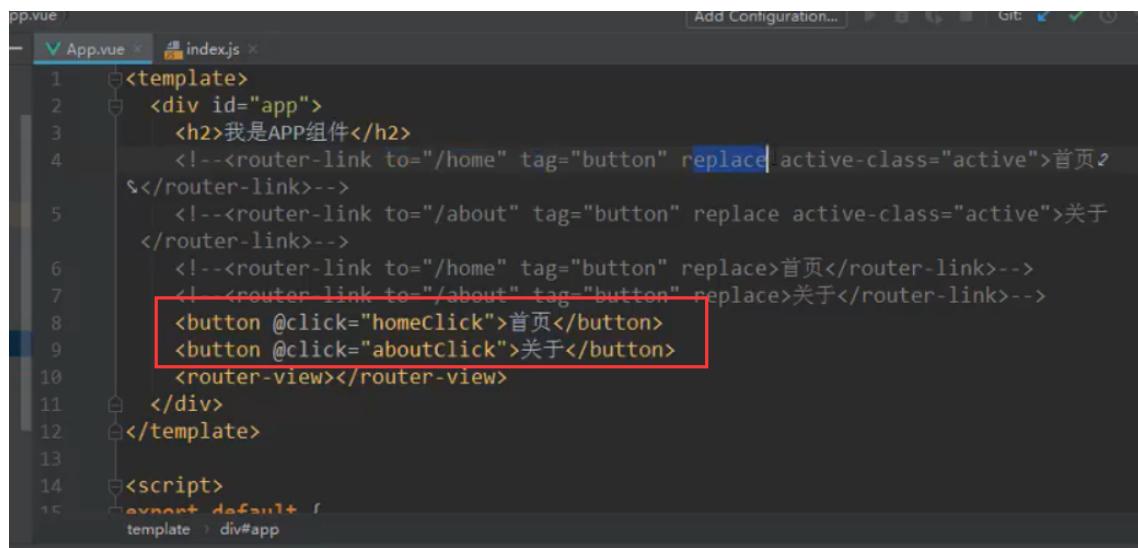
```
16   // 重定向
17   redirect: '/Home'
18 },
19 {
20 /* 这里写path，并不能写url，url是一个完整的路径
   而这里只是一个相对的路径*/
21   path: '/About',
22   component: About
23 },
24 {
25   path: '/Home',
26   component: Home
27 }
28 ],
29 mode: 'history',
30 linkActiveClass: 'active'  // 高亮
31 }
32
33 }
34 }
```

通过代码跳转路由

上面方法是通过路由组件+router-link+router-view 标签来实现路由跳转

还有一种方法是通过代码跳转路由，同样要用到**router-view**

仅仅在app组件中添加两个按钮，为按钮绑定两个点击事件

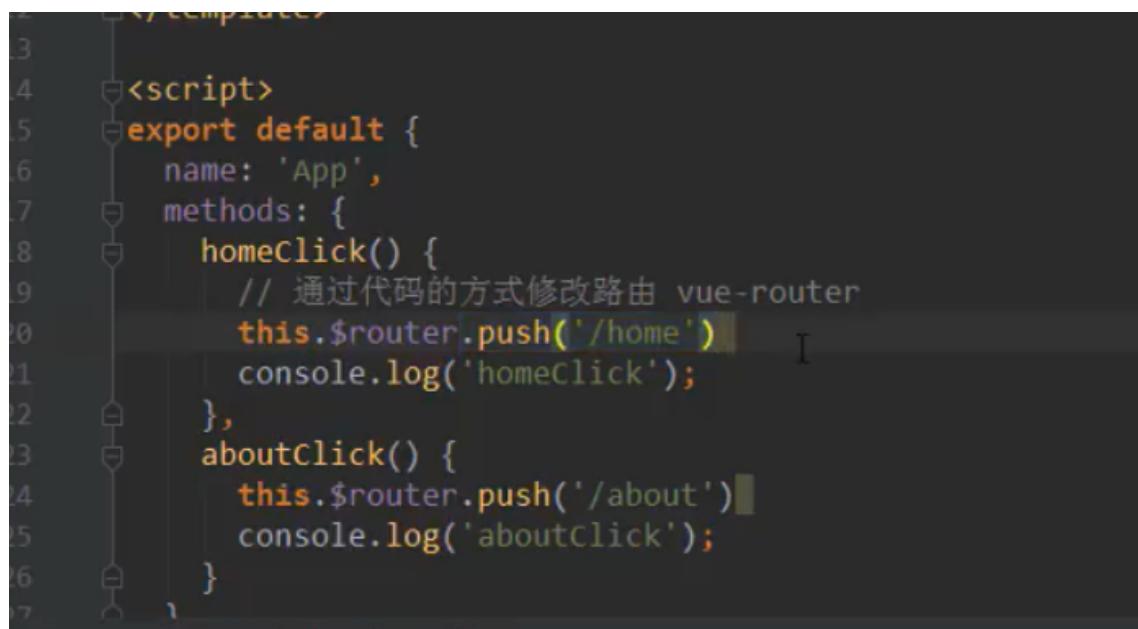


```
App.vue
1 <template>
2   <div id="app">
3     <h2>我是APP组件</h2>
4     <!--<router-link to="/home" tag="button" replace active-class="active">首页</router-link>-->
5     <!--<router-link to="/about" tag="button" replace active-class="active">关于</router-link>-->
6     <!--<router-link to="/home" tag="button" replace>首页</router-link>-->
7     <!--<router-link to="/about" tag="button" replace>关于</router-link>-->
8     <button @click="homeclick">首页</button>
9     <button @click="aboutclick">关于</button>
10    <router-view></router-view>
11  </div>
12 </template>
13
14 <script>
15   export default {
16     template: '#app'
17   }
18 </script>
```

vue-router模块往所有的组件里面添加了一个**\$router**属性

通过**this.\$router.push()**来进行路由的跳转

push内部使用的方法其实是**history.pushState()**, 浏览器可以返回



```
index.js
1 <script>
2   export default {
3     name: 'App',
4     methods: {
5       homeClick() {
6         // 通过代码的方式修改路由 vue-router
7         this.$router.push('/home')
8         console.log('homeClick');
9       },
10      aboutClick() {
11        this.$router.push('/about')
12        console.log('aboutClick');
13      }
14    }
15  </script>
```

对应的还有一个**this.\$router.replace()**方法来跳转路由，

该方法使用的是**history.replaceState()**, 浏览器不能返回

```
App.vue
methods: {
  homeClick() {
    // 通过代码的方式修改路由 vue-router
    // push => pushState
    // this.$router.push('/home')
    this.$router.replace('/home')
    console.log('homeClick');
  },
  aboutClick() {
    this.$router.push('/about')
    this.$router.replace('/home')
    console.log('aboutClick');
  }
}
</script>
```

```
index.js
{
  path: '/user/:id',
  component: User
}
```

动态路由的使用



动态路由

■ 在某些情况下，一个页面的path路径可能是不确定的，比如我们进入用户界面时，希望是如下的路径：

- /user/aaaa或/user/bbbb
- 除了有前面的/user之外，后面还跟上了用户的ID
- 这种path和Component的匹配关系，我们称之为动态路由(也是路由传递数据的一种方式)。

```
<div>
  <h2>{{$route.params.id}}</h2>
</div>
```

```
<router-link to="/user/123">用户</router-link>
```

localhost:8080/user/123

我是APP页面

首页 关于用户

123

某些情况下一个页面的path路径是不确定的，比如：不同用户登录时，大致的url路径是一样的，但是不同用户的url路径最后面可能是不一样的，张三:user/zhangsan，李四:user/lisi

也就是path部分的路径不同，这种**path (相对路径) 和component (组件) 的匹配关系**，称之为**动态路由**

创建动态路由的过程：

①

在router对象中添加一个路由与组件的映射关系

只不过这个映射关系的path的值最后添加一个**冒号+变量 :变量**

冒号标明这个是个变量，可以被替换



```
22     path: '/About',
23     component: About
24   },
25   {
26     path: '/Home',
27     component: Home
28   },
29   {
30     path: '/user/:name',
31     component: User
32   }
33 ],
34 mode: 'history',
35 linkActiveClass: 'active'
36
37 })
```

②

在App组件中，在使用User路由组件的时候，来为动态路由中的变量赋值

在APP组件的**data**中定义一个变量（当然可以从数据库中调用），在router-link的**to**属性中引用这个变量

引用变量的时候，因为是在标签里面，所以要使用**v-bind**，并使用**+加号**与/user/来进行路径拼接



```
1 <template>
2   <div id="app">
3     <router-link tag="button" to="/Home">
4       首页
5     </router-link>
6     <router-link tag="button" to="/About">
7       关于
8     </router-link>
9     <router-link tag="button" :to="'/User/' + userId">
10      用户
11    </router-link>
12    <router-view></router-view>
13  </div>
14 </template>
15 <script>
16 export default{
17   name: 'App',
18   data(){
19     return {
20       userId: '张三'
21     }
22   }
23 }
```

③

上面几部将动态路由地址修改好了，还没有相应的路由组件，新建一个User组件，不同的用户应该有不同的内容，要想在User组件中显示App组件从data中引入的用户名的名字这就用到了之前在**绑定路由与组件时用到的冒号后面的变量**

```
        },
        {
          path: '/User/:uname',
          component: User
        }
```

这个值再App组件中已经被赋值，所以User组件中可以使用这个变量来获取数据

router对象中有一个**\$route属性**（注意与\$route属性区别开），此属性指的是当前活跃的路由组件（就是当前页面显示的是哪一个路由组件，\$router指的是哪一个路由组件）

this.\$route.params: 指的是活跃路由组件的参数，在参数中可以引用uname变量，即
this.\$route.params.uname



```
About.vue      Home.vue      App.vue      main.js      index.js      User.vue
1 <template>
2   <div>
3     <h2>我是用户界面</h2>
4     <p>我是用户的相关信息</p>
5     <h2>我的名字叫: {{userName}}</h2>
6   </div>
7 </template>
8
9 <script>
10 export default{
11   name: "User",
12   computed: {
13     userName(){
14       return this.$route.params.uname
15     }
16   }
17 }
18 </script>
19
20 <style>
21 </style>
```

打包文件的解析

在之前使用webpack进行打包的时候，将所有打包的文件，除了HTML文件时另外打包的，**所有的js与css文件都是打包到了bundle.js文件中**，并进行了**压缩**

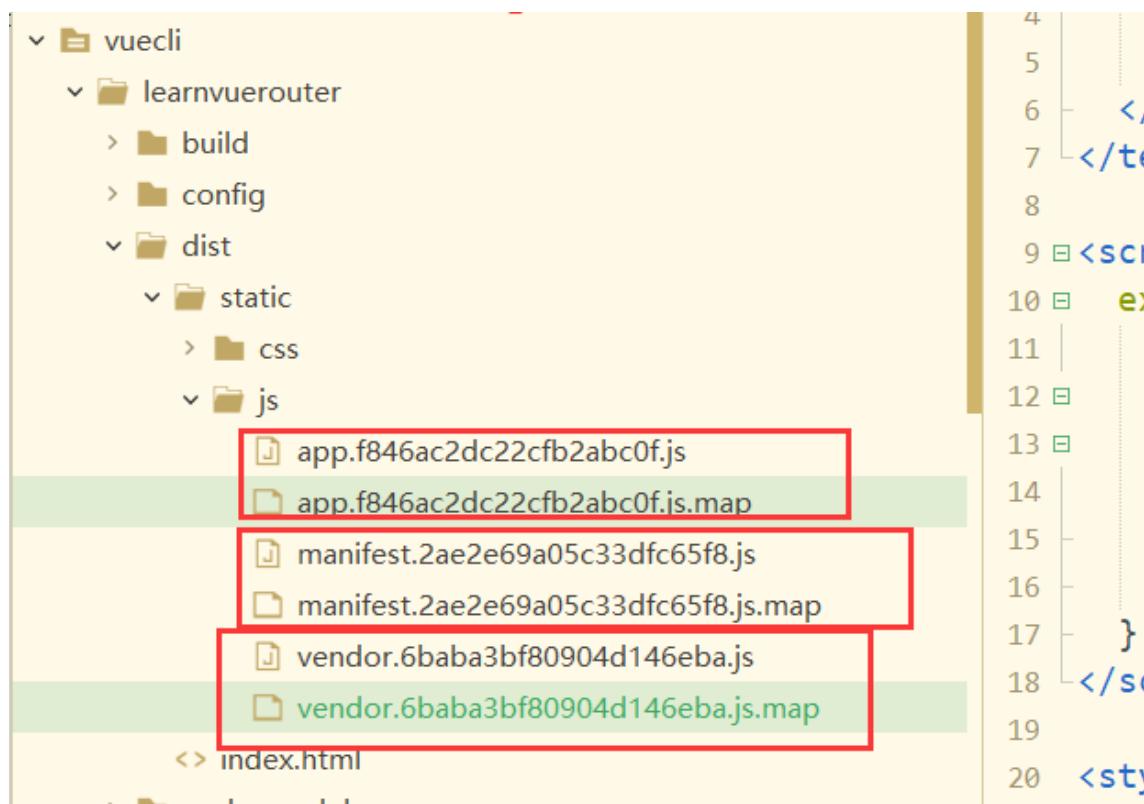
但是在vue中，将文件进行打包后，上传到**dist文件夹**中，**js和css文件进行了分离**，并且**index.html文件也进行了压缩**

其中js文件也并不是一整个，而是进行了划分，**分为三个js文件**，三个文件代表三个层次

第一个：当前应用程序开发的所有代码，即我们自己写的所有代码

第二个：manifest为我们打包的代码作**底层支撑**的，比如我们代码中使用的commonJS或ES6的语法规规范，或者其他的一些复杂的处理，为这些做低层支撑

第三个：vendor第三方/提供商，我们在编写程序时使用的框架、引用的第三方的内容：vue、vue-router、axios、bs等等



路由懒加载的使用

如果项目多的话，页面第一次进行加载，从服务器一次性会把所有的文件进行下载

如果文件过大，页面第一次加载会很卡顿

我们希望的是，每次加载页面，用到了那个路由组件，就从服务器上加载过来，没用到了路由组件先不从服务器上加载过来，之后用到哪个在加载哪个，这就叫**路由懒加载**

之前打包的话，所有我们自己写的代码都打包到了一个js文件，也就是所有的路由组件都打包到了一个文件，现在要实现懒加载，就需要将每个路由组件从这一个js文件中分离出来

路由懒加载的效果

The screenshot shows two columns illustrating the effect of lazy loading.

Left Column (Standard Loading):

```
import Home from '../components/Home'
import About from '../components/About'

Vue.use(VueRouter);

const routes = [
  { path: '/home', component: Home },
  { path: '/about', component: About }
];
```

This column includes a red box around the 'index.html' file, which contains links to several large JS files: 'app.801c2823389fbf98a530.js', 'manifest.2ae2e69a05c33dfc65f8.js', and 'vendor.1748317793fd05195ff8.js'.

Right Column (Lazy Loading):

```
const routes = [
  { path: '/home',
    component: () => import('../components/Home') },
  { path: '/about',
    component: () => import('../components/About') }
];
```

This column includes a red box around the 'index.html' file, which contains links to smaller files: '0.09675c5e37c95c6e65ff.js', '1.266ad04847546fd5bdb2.js', 'app.513c0ad5da30a20ee757.js', 'manifest.f2307a2fbea088ed5ed4.js', and 'vendor.426ef21560bb1458790e.js'.

懒加载的方式

- 方式一: 结合Vue的异步组件和Webpack的代码分析.

```
const Home = resolve => { require.ensure(['../components/Home.vue'], () =>
  { resolve(require('../components/Home.vue')) })};
```

- 方式二: AMD写法

```
const About = resolve => require(['../components/About.vue'], resolve);
```

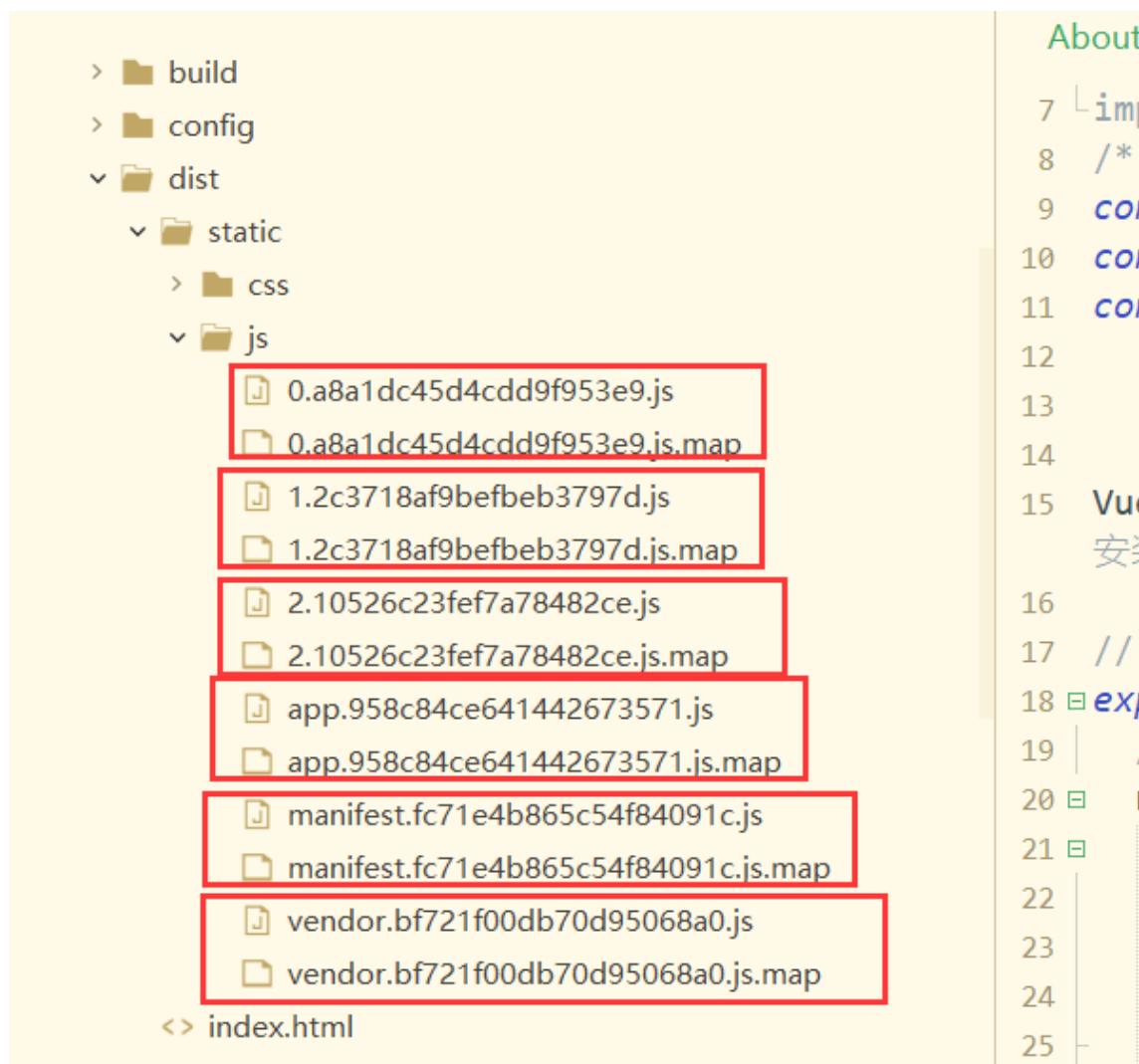
- 方式三: 在ES6中, 我们可以有更加简单的写法来组织Vue异步组件和Webpack的代码分割.

```
const Home = () => import('../components/Home.vue')
```

在router的index.js文件中, 将所有的路由组件进行懒加载

而App.vue 是不需要懒加载的, 因为一开始App就是我们主页面的内容

所以打包后, 除了manifest和vendor这两个文件, 还有四个: 一个App.vue文件, 三个懒加载的文件



路由的嵌套使用

■ 嵌套路由是一个很常见的功能

□ 比如在home页面中, 我们希望通过/home/news和/home/message访问一些内容.

□ 一个路径映射一个组件, 访问这两个路径也会分别渲染两个组件.

■ 路径和组件的关系如下:



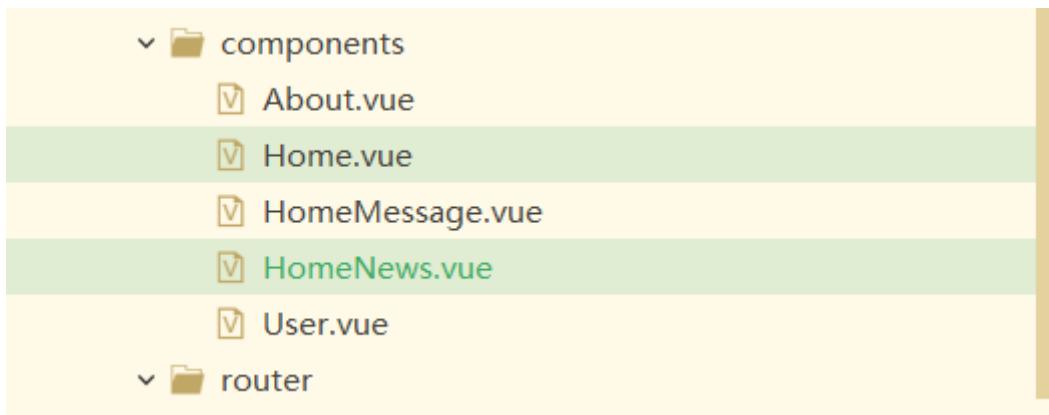
■ 实现嵌套路由有两个步骤:

- 创建对应的子组件, 并且在路由映射中配置对应的子路由.
- 在组件内部使用<router-view>标签.

步骤跟之前创建路由组件大致相同

①

先创建两个子组件放在components文件夹中



②

在router文件夹中的index.js文件中进行路由子组件的映射绑定

子组件的映射绑定添加在父组件的映射对象中就行

在父组件的映射对象中添加一个children数组, 数组中就是子组件的路由与组件的映射绑定

跟之前讲的一样, 同样可以为子组件指定默认显示的组件

注意: 在子组件中, 无论是path中的地址, 还是redirect中的地址, 都不要在前面加/斜杠

```
outer > index.js | Lx 调试入口文件
About.vue   Home.vue   App.vue   main.js   index.js   HomeNews.v
21 // 配置路由与组件的映射关系，数组中的一个对象是一个映射关系
22 routes: [
23   {
24     path: '',
25     // 重定向
26     redirect: '/Home'
27   },
28   {
29     /* 这里写path，并不能写url，url是一个完整的路径
30      而这里只是一个相对的路径*/
31     path: '/About',
32     component: About
33   },
34   ...
35   path: '/Home',
36   component: Home,
37   children:[
38     {
39       path: '',
40       // 重定向，子组件中同样不能写/斜杠
41       redirect: 'news'
42     },
43     {
44       // 路由组件的子组件，path不能写/斜杠
45       path: 'news',
46       component: HomeNews
47     },

```

③

在父组件中导入子路由组件，也就是子组件路由标签的使用放在Home父组件中

同样使用router-link标签和to属性来导入

使用router-view标签来占位

注意： to属性中不能只写子路由组件的path，应当带上父组件的，这样才表明嵌套关系，不然直接就用子路由路径替换了父路由路径，同时注意一定在最前面加/斜杠

```

1 ① <template>
2   <div>
3     <h2>我是首页</h2>
4     <p>hahahahahahahahahah</p>
5     <router-link to="/Home/news">新闻</router-link>
6     <router-link to="/Home/message">消息</router-link>
7     <router-view></router-view>
8   </div>
9 </template>
10
11 <script>
12
13 </script>
14

```

参数传递



小码 哥 教 育
SEEMYGO

传递参数的方式

- 传递参数主要有两种类型: params和query
- **params的类型:**
 - 配置路由格式: `/router/:id`
 - 传递的方式: 在path后面跟上对应的值
 - 传递后形成的路径: `/router/123, /router/abc`
- **query的类型:**
 - 配置路由格式: `/router`, 也就是普通配置
 - 传递的方式: 对象中使用query的key作为传递方式
 - 传递后形成的路径: `/router?id=123, /router?id=abc`

统一资源定位符的标准格式如下：

协议类型:[//服务器地址[:端口号]][/资源层级UNIX文件路径]文件名[?查询][#片段ID]

统一资源定位符的完整格式如下：

协议类型:[//[访问资源需要的凭证信息@]服务器地址[:端口号]][/资源层级UNIX文件路径]文件名[?查询][#片段ID]

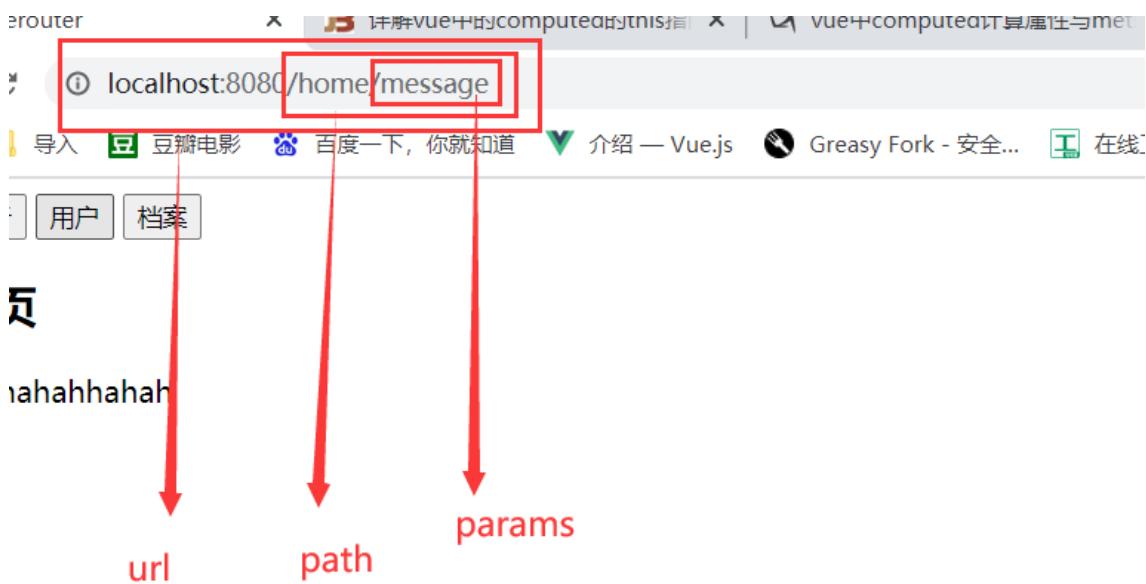
以 <http://zh.wikipedia.org:80/w/index.php?title=Special%E9%A8F%E6%9C%BA%E9%A1%B5%E9%9D%A2> 为例, 其中:

1. **http**, 是协议;
2. **zh.wikipedia.org**, 是服务器;
3. **80**, 是服务器上的网络端口号;
4. **/w/index.php**, 是路径;
5. **?title=Special%E9%A8F%E6%9C%BA%E9%A1%B5%E9%9D%A2**, 是询问。

统一资源定位符 协议://主机:端口/路径?查询#哈希片段

url scheme://host:port/path?query#fragment

关于path、url、params



url就指的是整个的网址

path就指的是网址后面几个/斜杠中的内容, 包括多个路由嵌套

params指的是

query传递参数

参数传递主要有两种类型: **params**、**query**

params类型之前已经使用过了

使用query方法的话, 想从App组件中向子组件传递参数

同样先在App组件中子组件标签的to属性中写入需要传的参数

同样为to属性绑定**v-bind**, **to属性双引号中的内容变成了一个对象**

对象有**两个属性**

path属性: 子组件的地址

query属性: 属性值是一个对象, 里面是需要传递的各种数据

```
<template>
  <div id="app">
    <router-link tag="button" to="/home">首页</router-link>
    <router-link tag="button" to="/about">关于</router-link>
    <router-link tag="button" :to="/user/" + userId>用户</router-link>
    <router-link tag="button" :to="{path: '/profile', query: {name: 'why', height: 18, age: 18}}">档案</router-link>
  </div>
</template>
```

添加query后，url如下，query部分是问号后面的部分

① localhost:8080/profile?name=why&height=18&age=18

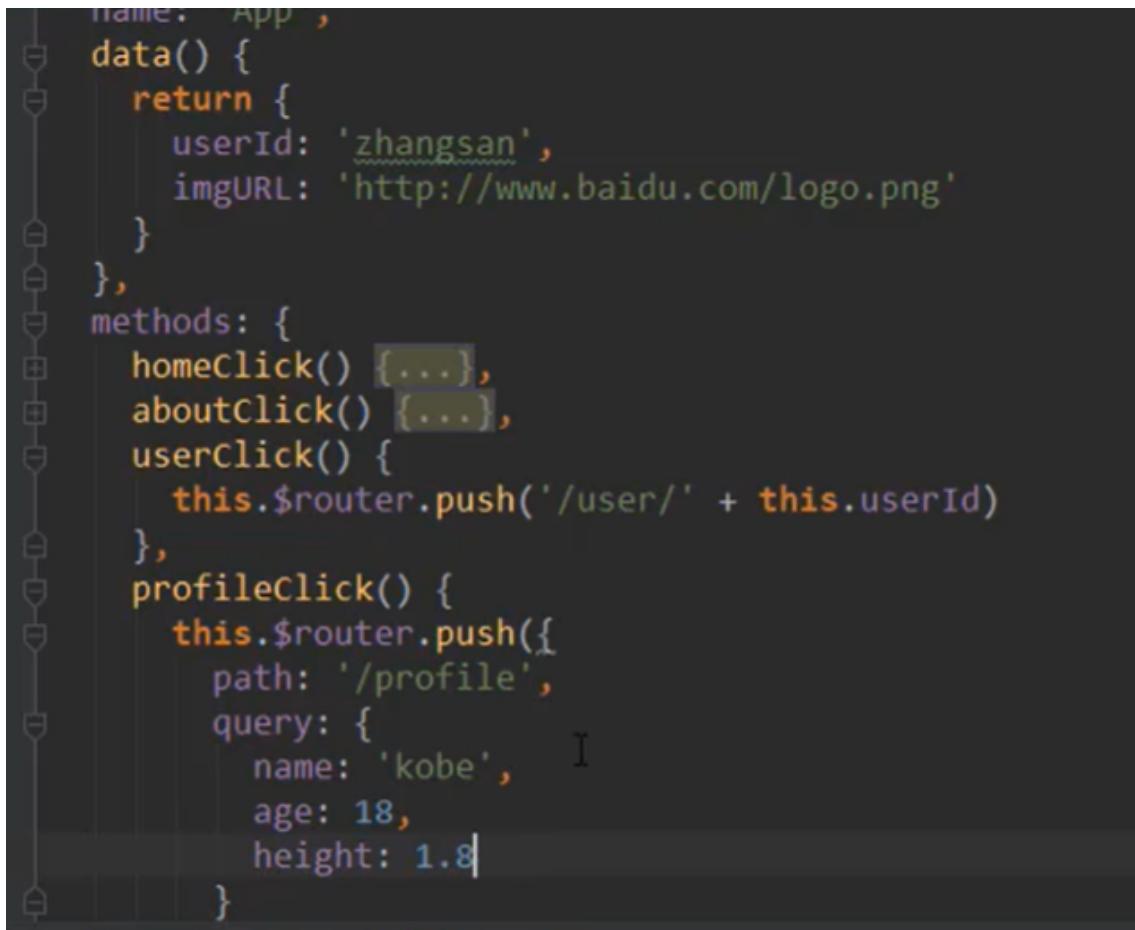
路由子组件收到数值时通过\$route.query.变量来接收

index.js Profile.vue App.vue User.vue

```
1 <template>
2   <div>
3     <h2>我是Profile组件</h2>
4     <h2>{{$route.query.name}}</h2>
5     <h2>{{$route.query.height}}</h2>
6     <h2>{{$route.query.age}}</h2>
7   </div>
```

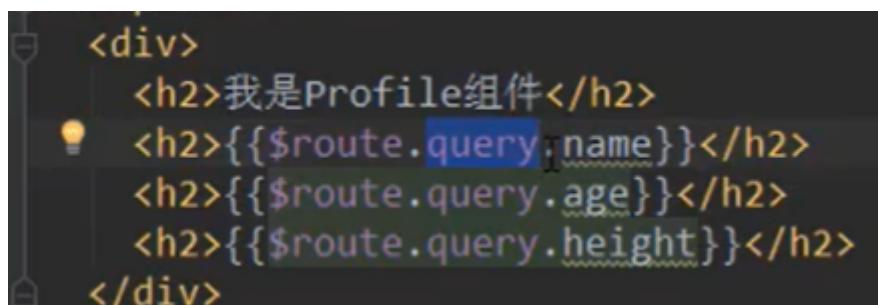
例子：通过代码的方式进行路由跳转，并用query穿传递参数

```
<!-- 档案-->
<button @click="userClick">用户</button>
<button @click="profileClick">档案</button>
```



```

    name: 'App',
    data() {
      return {
        userId: 'zhangsan',
        imgURL: 'http://www.baidu.com/logo.png'
      }
    },
    methods: {
      homeClick() {...},
      aboutClick() {...},
      userClick() {
        this.$router.push('/user/' + this.userId)
      },
      profileClick() {
        this.$router.push({
          path: '/profile',
          query: {
            name: 'kobe',
            age: 18,
            height: 1.8
          }
        })
      }
    }
  
```



```

<div>
  <h2>我是Profile组件</h2>
  <h2>{{ $route.query.name }}</h2>
  <h2>{{ $route.query.age }}</h2>
  <h2>{{ $route.query.height }}</h2>
</div>

```

router和route的由来

\$router

在index.js文件中导出的整个export default new Router

人和组件中使用的\$router拿到的对象是一样的

这个router对象跟\$router所指的对象是一个东西

这个对象有go、back、forward、replace等方法

\$route

\$route指的是当前活跃的路由，也就是在index.js文件中配置的一个个路由

上面这两个属性，都属于vue原型里面的属性

导航守卫

在几个路由之间进行跳转时，如果想做一些监听，在监听函数中做一些事情，就用到了导航守卫

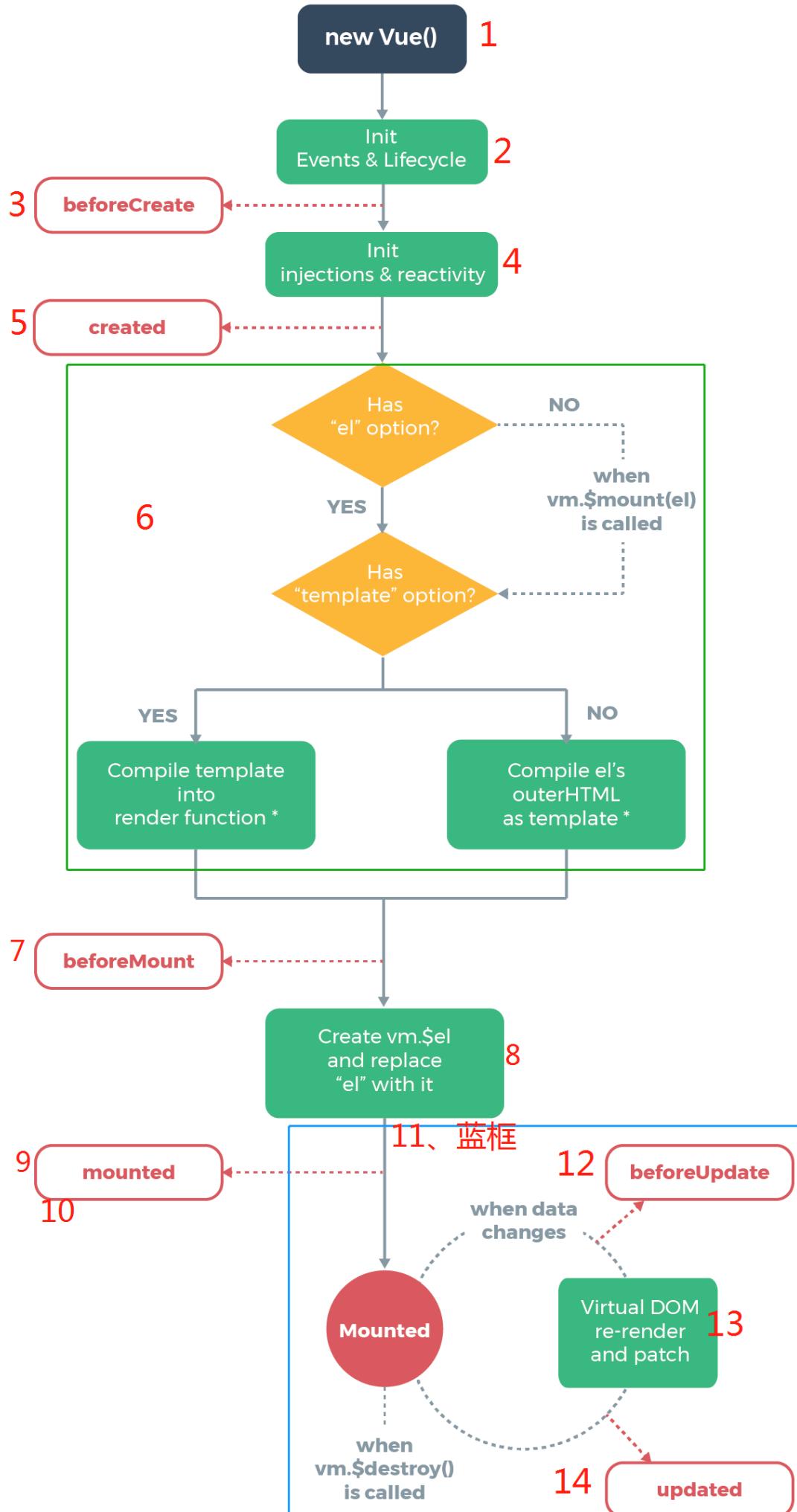
几个生命周期钩子函数

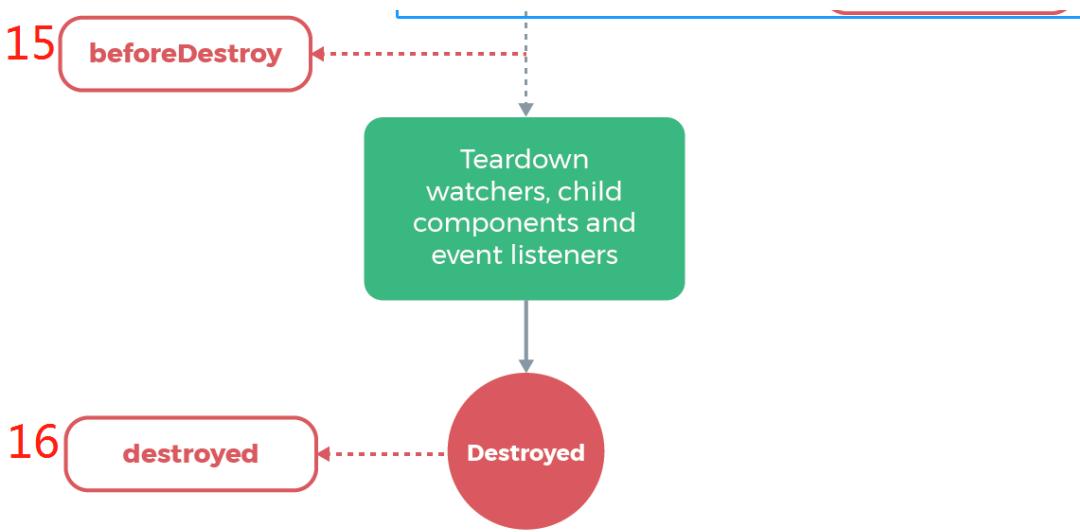
create(): 在组件一被创建出来就回调这个函数，函数内容自定义，可以做一些操作

mounted(): 在组件一被挂载到dom中，就会回调这个函数中的内容，同样函数的内容是自定义的

updated(): 在页面发生刷新的时候，就调用一次这个函数

扩展：





* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

https://blog.csdn.net/sleepwalker_1992

下面解释一一对应图中的数字

- 1、`var vm = new Vue({})` 表示开始创建一个Vue的实例对象
- 2、刚初始化一个空的Vue实例对象，此时，在这个对象上，只有一些默认的生命周期函数和默认的事件，其他的都未创建
- 3、**beforeCreate** 生命周期函数执行时，data和methods中的数据和方法都还没有初始化
- 4、初始化data和methods
- 5、在**created**中，data和methods都已经初始化好了，如果要操作data中的数据或是调用methods中的方法，最早只能在created中操作
- 6、这个绿框中内容表示Vue开始编辑模板，把Vue代码中的那些指令进行执行，最终，在内存中生成一个编译好的最终的模板字符串对象，然后把这个字符串对象，渲染为内存中的DOM，此时，只是在内存中渲染好了模板，并没有把模板挂载到真正的页面中去
- 7、**beforeMount** 函数执行时，模板已经在内存中编译好了，但尚未挂载到页面中去，此时，页面还是旧的
- 8、将内存中编译好的模板，真实的替换到浏览器的页面中区
- 9、**mounted** 是在页面加载完成后执行的函数，如果要通过某些插件操作页面上的DOM节点，最早是在mounted中进行
- 10、只要执行完了mounted，就表示整个Vue实例对象已经初始化完毕了，此时组件已经脱离创建阶段，进入运行阶段。
- 11、蓝框中是组件的运行阶段，运行阶段的生命周期函数只有两个：beforeUpdate和updated，这两个事件会根据data数据的改变，有选择的触发0次到多次
- 12、当执行**beforeUpdate**时，页面中显示的数据还是旧的，此时data中的数据是最新的，页面尚未和最新数据同步
- 13、这一步，先根据data中最新的数据，在内存中，重新渲染出一份最新的内存DOM树，当内存DOM树被更新之后，会把最新的的内存DOM树，重新渲染到真实的页面当中，这时，就完成数据从data (Model层) ->view (视图层) 的更新
- 14、**updated**执行时，页面和data数据已经保持同步，都是最新的

15、当执行**beforeDestroy**钩子函数时，Vue实例就已经从运行阶段进入销毁阶段，此时，组建中所有data、methods、以及过滤器，指令等，都处于可用状态，此时还未真正执行销毁过程

16、当执行**destroyed**函数时，组件已经被完全销毁，此时组建中所有data、methods、以及过滤器，指令等，都已经不可用了

导航守卫

为什么使用导航守卫？

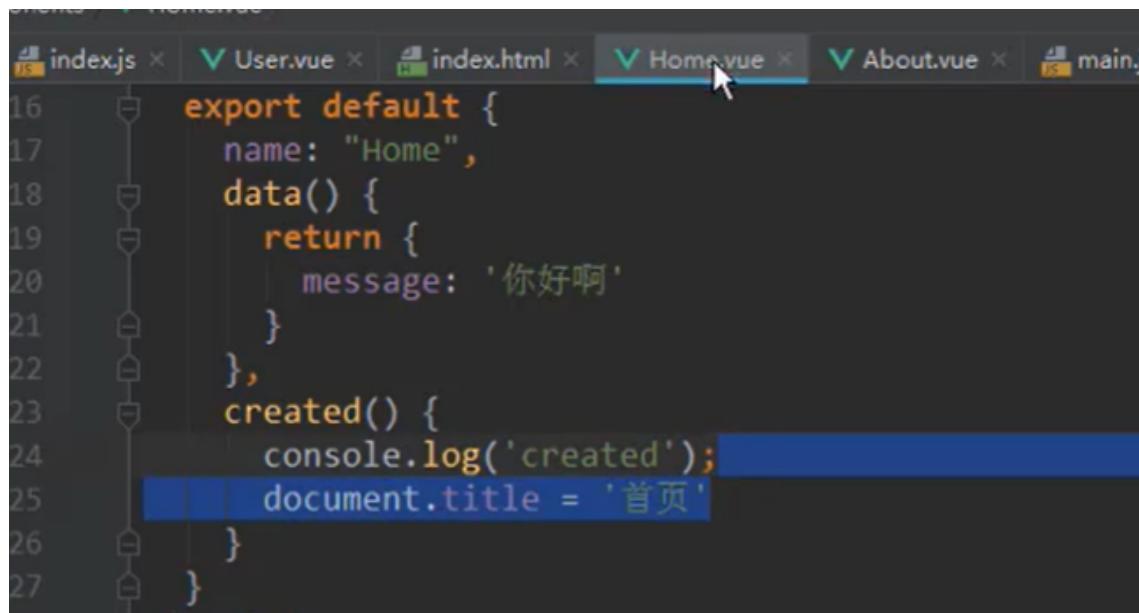
- 我们来考虑一个需求：在一个SPA应用中，如何改变网页的标题呢？
 - 网页标题是通过<title>来显示的，但是SPA只有一个固定的HTML，切换不同的页面时，标题并不会改变。
 - 但是我们可以通过JavaScript来修改<title>的内容>window.document.title = '新的标题'.
 - 那么在Vue项目中，在哪里修改？什么时候修改比较合适呢？
- 普通的修改方式：
 - 我们比较容易想到的修改标题的位置是每一个路由对应的组件.vue文件中。
 - 通过mounted声明周期函数，执行对应的代码进行修改即可。
 - 但是当页面比较多时，这种方式不容易维护（因为需要在多个页面执行类似的代码）。
- 有没有更好的办法呢？使用导航守卫即可。
- 什么是导航守卫？
 - vue-router提供的导航守卫主要用来监听监听路由的进入和离开的。
 - vue-router提供了beforeEach和afterEach的钩子函数，它们会在路由即将改变前和改变后触发。

情景

比如，如果想在每次切换页面的时候，页面的title也发生改变，切换到User界面，title就变成用户，切换到About界面title就变成关于.....

在每个路由组件中添加create()函数，在函数中将document.title改成对应的值

create函数是在组件data和methods都已经初始化好了之后调用



```
16  export default {
17    name: "Home",
18    data() {
19      return {
20        message: '你好啊'
21      }
22    },
23    created() {
24      console.log('created');
25      document.title = '首页'
26    }
27 }
```

这样每个路由的title都被改掉了

但是这么做效率太低，每个路由都需要调用一次create钩子函数

改进方法就是**全局导航守卫**

全局导航守卫：

在router的index.js文件中，在每个路由项中添加**meta属性（名字不能错）**，属性中可以添加一些**元数据（描述数据的数据）**，在所有的路由项中添加相应的title标题名

```
index.js    Profile.vue    App.vue    User.vue    About.vue

24 日
25     {
26         path: '',
27         // 重定向
28         redirect: '/home'
29 },
30 {
31     /* 这里写path，并不能写url，url是一个完整的路径
32      而这里只是一个相对的路径*/
33     path: '/about',
34     component: About,
35     // 添加元数据
36     meta: {
37         title: '关于'
38     },
39 {
40     path: '/home',
41     component: Home,
42     // 添加元数据
43     meta: {
44         title: '首页'
45     },
46     children: [
47         {
48             path: '',
49             // 重定向，子组件中同样不能写/斜杠
50         }
51     ]
52 }
```

在index.js文件下面拿到router对象，router对象有一个**beforeEach函数**

beforeEach是一个前置钩子

这个beforeEach函数的**参数**是一个函数

这个**作为参数的函数**本身有三个参数：to, from, next

to: 跳转到的路由

from: 从哪个路由进行跳转，也就是要离开的路由对象

next: 是一个函数，这个函数必须在函数中进行调用，否则所有的路由都不会跳转，本来这个方法时router对象自动调用的，这里自定义话，**这个函数必须写上**

在这个函数中将document.title=to.meta.title，函数可以写成箭头函数

```
14
15 router.beforeEach((to, from, next) => {
16
17     // 从from跳转到to
18     document.title = to.meta.title
19     // 必须调用next函数，否则所有的路由都不会跳转，本来这个方法时router对象自动调用的，这里自定义话，必要的一些函数必须写上
20     next()
21 })
```

但是这样做有一个问题，**当有路由嵌套的时候，title不能正常显示**

因为有路由嵌套和没有路由嵌套的to对象不太一样

无路由嵌套：

```
[HMR] Waiting for update signal from WDS... log.js?4244:23
▼ {name: undefined, meta: {...}, path: "/user/%E5%BC%A0%E4%B8%89", hash: "", query: {...}, ...} index.js?3672:90
  fullPath: "/user/%E5%BC%A0%E4%B8%89"
  hash: ""
  ▶ matched: [{}]
  ▼ meta:
    title: "用户"
    ▶ __proto__: Object
    name: undefined
    ▶ params: {id: "张三"}
    path: "/user/%E5%BC%A0%E4%B8%89"
    ▶ query: {}
    ▶ __proto__: Object
```

有路由嵌套：

```
▼ {name: undefined, meta: {...}, path: "/home/news", hash: "", query: {...}, ...} index.js?3672:90
  fullPath: "/home/news"
  hash: ""
  ▶ matched: (2) [{}]
  ▼ meta:
    ▶ __proto__: Object
    name: undefined
    ▶ params: {}
    path: "/home/news"
    ▶ query: {}
    ▶ __proto__: Object
```

发现有路由嵌套的to对象中的meta中没有title属性

但是无论有没有嵌套路由，在to对象中有一个**matched数组**，数组中的第一个对象中，都会有meta属性，并且有定义的title值

```

index.js?367
▼ {name: undefined, meta: {...}, path: "/home/news", hash: "", query: {...}, ...} ⓘ
  fullPath: "/home/news"
  hash: ""
  ▼ matched: Array(2)
    ▼ 0:
      ► alias: []
      ► beforeEnter: undefined
      ► components: {default: {...}}
      ► enteredCbs: {}
      ► instances: {default: VueComponent}
      ► matchAs: undefined
      ▼ meta:
        title: "首页"
        ► __proto__: Object
      name: undefined
      parent: undefined
      path: "/home"
      ► props: {}
      ► redirect: undefined
      ► regex: /^\/home(?:\/(?:$))?$/
      ► __proto__: Object
    ▶ 1: {path: "/home/news", regex: /^\/home\/news(?:\/(?:$))?$/, components: {...}, alias: Array(0), ...}
      length: 2
      ► __proto__: Array(0)
    ▼ meta:
      ► __proto__: Object
      name: undefined
      ► params: {}
      path: "/home/news"
      ► query: {}
      ► __proto__: Object

```

于是路由都从**to对象的matched数组中的第一个对象**获取meta就可以了

```

84
85 router.beforeEach((to, from, next) => {
86
87   // 从from跳转到to
88   document.title = to.matched[0].meta.title
89   // 必须调用next函数，否则所有的路由都不会跳转，本来这个方法时router对象自动调用的，这里自定义话，必要的一些函数必须写上
90   next()
91 })
92

```

导航守卫补充

前面讲到的beforeEach是一个前置钩子，官方更愿称之为守卫 (guard)

对应的也有后置钩子 (hook) : afterEach

afterEach的参数也是一个函数，这个函数的参数只有两个：to、from

因为这是已经跳转完了，并不需要做其他的操作，也就不需要next()函数了

关于next()函数的补充

- `next: Function` :一定要调用该方法来 resolve 这个钩子。执行效果依赖 `next` 方法的调用参数。
 - `next()` :进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 `confirmed` (确认的)。
 - `next(false)` :中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 `from` 路由对应的地址。
 - `next('/')` 或者 `next({ path: '/' })` :跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 `next` 传递任意位置对象，且允许设置诸如 `replace: true`、`name: 'home'` 之类的选项以及任何用在 `router-link` 的 `to prop` 或 `router.push` 中的选项。
 - `next(error)` :(2.4.0+) 如果传入 `next` 的参数是一个 `Error` 实例，则导航会被终止且该错误会被传递给 `router.onError()` 注册过的回调。

确保要调用 `next` 方法，否则钩子就不会被 resolved。

上面我们使用的守卫时全局守卫

除此之外还有路由独享守卫、组件内的守卫

<https://router.vuejs.org/zh/guide/advanced/navigation-guards.html>

路由独享守卫

你可以在路由配置上直接定义 `beforeEnter` 守卫：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

js

这些守卫与全局前置守卫的方法参数是一样的。

在组件中添加`beforeEnter`属性，属性的内容是一个函数，内容、参数和之前的全局守卫一样
同样需要在这个剪头函数中添加一个`next()`函数

组件内的守卫

keep-alive及其其他问题

情景

如下的首页



我是首页

hahahahahahahahahah

新闻 消息

- 消息1
 - 消息2
 - 消息3
 - 消息4

首页中子组件设定默认显示的是新闻，当我们在首页中点击了消息页面，点击用户或者档案路由组件后，再次返回首页，这时候子组件又变成了新闻，不会保留我们之前的状态

keep-alive就是用来保留我们网页状态的一个组件

因为之前的每一个路由组件的打开与关闭都是一个生命周期

每次都会经历create()创建一个新的路由，在经过destroy()函数销毁，每次都是一个新的过程

keep-alive就相当于保持存活的意思

keep-alive是Vue内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。

router-view是**vue-router**内置的一个组件，如果直接被包在**keep-alive**里面，所有路径匹配到的视图组件都会被存

①

在App组件之前的router-view标签前后套上一个keep-alive标签

```
<keep-alive>
  <router-view></router-view>
</keep-alive>
```

②

在index.js文件中之前设置的home路由默认指定的子路由路径取消



```
path: '/home',
component: Home,
// 添加元数据
meta: {
  title: '首页'
},
children: [
  /* {
    path: '',
    // 重定向, 子组件中同样不能写/斜杠
    redirect: 'news'
  }, */
  {
    // 路由组件的子组件, path不能写/斜杠
    path: 'news',
    component: HomeNews
  },
  {
    path: 'message',
    component: HomeMessage
  }
]
```

③

将指定默认子路由的步骤放在Home.vue文件中来做

在组件的data中放一个path变量，用来保存/home/news

使用**activated()函数**（此函数是当前页面活跃时的钩子函数）

（对应的还有一个deactivated函数，即为失去活跃，

注意：这两个函数，只有该组件被保持了状态使用了keep-alive时，才是有效的

）

使用**this.\$router.push方法**将当前地址进行修改

修改为data中存放的path变量

The screenshot shows a code editor with several tabs at the top: index.js, Profile.vue, App.vue, User.vue, About.vue, Home.vue (which is selected and highlighted in green), and main. The main content area displays the Home.vue code:

```
1 <template>
2   <div>
3     <h2>我是首页</h2>
4     <p>hahahahahahahahahah</p>
5     <router-link to="/home/news">新闻</router-link>
6     <router-link to="/home/message">消息</router-link>
7     <router-view></router-view>
8   </div>
9 </template>
10
11 <script>
12   export default{
13     data(){
14       return {
15         path: '/home/news'
16       }
17     },
18     activated(){
19       this.$router.push(this.path);
20     }
21   }

```

Two sections of the code are highlighted with red boxes: the `data` method and the `activated` method.

④

然后在使用一个上面提到的组件内的导航守卫beforeRouteLeave函数

```
beforeRouteLeave(to, from, next) {
  // 导航离开该组件的对应路由时调用
  // 可以访问组件实例 `this`
  ...
  next()
}
```

当在home组件中点击任意一个子路由组件，离开时会调用beforeRouteLeave函数

在函数中将path重新赋值为当前的路径

```
index.js      Profile.vue      App.vue      User.vue      About.vue      Home.vue      main.js
1  <template>
2    <div>
3  |    <h2>我是首页</h2>
4  |    <p>hahahahahahahahahah</p>
5  |    <router-link to="/home/news">新闻</router-link>
6  |    <router-link to="/home/message">消息</router-link>
7  |    <router-view></router-view>
8    </div>
9   </template>
10
11 <script>
12   export default{
13     data(){
14       return {
15         path: '/home/news'
16       }
17     },
18     activated(){
19       this.$router.push(this.path);
20     },
21     beforeRouteLeave(to, from, next){
22       this.path=this.$route.path;
23       next()
24     }
25   }
26 </script>
```

因为此时路由已经处于keep-alive 所以，这个值还会被保留下

再次打开home页面时，由于activated函数又一次调用

地址会被绑定到已经赋过值的path

keep-alive属性介绍

它们有两个非常重要的属性：

include: 字符串或正则表达式，只有匹配的组件会被缓存

exclude: 字符串或正则表达式，任何匹配的组件都不会被缓存

之前在组件的**export default{}中习惯性的写了name属性**

之前一直没用过，现在可以用于include/exclude属性来匹配路由组件

```
index.js × Profile.vue App.vue User.vue /  
1 ⚡<template>  
2 ⚡  <div>  
3 |   <h2>我是Profile组件</h2>  
4 |   <h2>{{$route.query.name}}</h2>  
5 |   <h2>{{$route.query.height}}</h2>  
6 |   <h2>{{$route.query.age}}</h2>  
7 | </div>  
8 </template>  
9  
10 ⚡<script>  
11 ⚡  export default{  
12 |    name: 'Profile',  
13 |  }  
14  
15  
16 </script>  
17  
18 <style>  
19 </style>
```

在keep-alive标签中添加include/exclude属性，属性值就是name值

```
index.js × Profile.vue App.vue User.vue About /  
1 ⚡<template>  
2 ⚡  <div id="app">  
3 |   <router-link tag="button" to="/home">首页</router-link>  
4 |   <router-link tag="button" to="/about">关于</router-link>  
5 |   <router-link tag="button" :to="'/user/' + user.id>用户</router-link>  
6 |   <router-link tag="button" :to="{path: '/profile'}>个人中心</router-link>  
7 ⚡  <keep-alive exclude="Profile">  
8 |   <router-view></router-view>  
9 | </keep-alive>  
10 </div>
```

多个路由组件用逗号分开，注意逗号后面不能加空格

```
<keep-alive exclude="Profile,User">  
  <router-view/>  
</keep-alive>
```

文件别名

找到 **webpack.base.config.js** 文件

在文件的 **resolve** 对象中找到 **alias** 属性 进行文件别名的设置

之前讲过 **resolve** 对象，是专门用来解决文件路径问题的一个对象

之前讲过这里面用来省略文件后缀 的一个属性： **extensions**

alias (别名) 属性中设置文件的别名，使用时可以简化路径书写

并且之后在复制代码时，不用担心路径问题

```
webpack.base.config.js
25 resolve: {
26   extensions: ['.js', '.vue', '.json'],
27   alias: {
28     '@': resolve('src'),
29     'assets': resolve('src/assets'),
30     'components': resolve('src/components'),
31     'views': resolve('src/views'),
32   },
33 },
```

```
MainTabBar.vue
16 <div slot="item-text">购物车</div>
17 </tab-bar-item>
18 <tab-bar-item path="/profile" activeColor="deepPink">
19   
20   
21   <div slot="item-text">我的</div>
22 </tab-bar-item>
23 </tab-bar>
24 </template>
25
26 <script>
27   import TabBar from '@/components/tabbar/TabBar'
28   import TabBarItem from './tabbar/TabBarItem'
29
30 </script>
31
32 <export default {
33   name: "MainTabBar",
34   components: {
35     TabBar,
36     TabBarItem
37   }
38 }>
```

注意：这种修改路径的方法在 js 代码中（基本上就是 **import**）可以直接使用

在 HTML 代码中（像 **src**）就不能直接使用了，需要在路径前面加上波浪符号~

在 **cli2** 中定义路径别名，**后面定义的不能利用之前定义的别名**，也就是上面这个例子，后面别名定义时，不能利用 @ 这个上面定义过的别名

cli3 中可以用

```
<tab-bar>
  <tab-bar-item path="/home" activeColor="pink">
    
    
    <div slot="item-text">首页</div>
```

Vuex



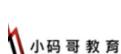
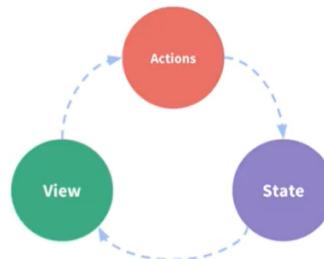
Vuex是做什么的？

- 官方解释：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。
 - Vuex 也集成到 Vue 的官方调试工具 [devtools extension](#)，提供了诸如零配置的 time-travel 调试、状态快照导入导出等高级调试功能。
- 状态管理到底是什么？
 - 状态管理模式、集中式存储管理这些名词听起来就非常高大上，让人捉摸不透。
 - 其实，你可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面。
 - 然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用。
 - 那么，多个组件是不是就可以共享这个对象中的所有变量属性了呢？
- 等等，如果是这样的话，为什么官方还要专门出一个插件Vuex呢？难道我们不能自己封装一个对象来管理吗？
 - 当然可以，只是我们要先想想VueJS带给我们最大的便利是什么呢？没错，就是响应式。
 - 如果你自己封装实现一个对象能不能保证它里面所有的属性做到响应式呢？当然也可以，只是自己封装可能稍微麻烦一些。
 - 不用怀疑，Vuex就是为了提供这样一个在多个组件间共享状态的插件，用它就可以了。



单界面的状态管理

- 我们知道，要在单个组件中进行状态管理是一件非常简单的事情
 - 什么意思呢？我们来看下面的图片。
- 这图片中的三种东西，怎么理解呢？
 - State：不用多说，就是我们的状态。（你姑且可以当做就是 data 中的属性）
 - View：视图层，可以针对 State 的变化，显示不同的信息。（这个好理解吧？）
 - Actions：这里的 Actions 主要是用户的各种操作：点击、输入等等，会导致状态的改变。



单界面状态管理的实现

```
<template>
<div class="test">
  <div>当前计数: {{counter}}</div>
  <button @click="counter+=1">+1</button>
  <button @click="counter-=1">-1</button>
</div>
</template>

<script>
export default {
  name: 'HelloWorld',
  data () {
    return {
      counter: 0
    }
  }
}

</script>

<style scoped>
</style>
```

- 在这个案例中，我们有木有状态需要管理呢？没错，就是个数 counter。
- counter 需要某种方式被记录下来，也就是我们的 State。
- counter 目前的值需要被显示在界面中，也就是我们的 View 部分。
- 界面发生某些操作时（我们这里是用户的点击，也可以是用户的 input），需要去更新状态，也就是我们的 Actions
- 这不就是上面的流程图了吗？



多界面状态管理

- Vue已经帮我们做好了单个界面的状态管理，但是如果是多个界面呢？
 - 多个试图都依赖同一个状态（一个状态改了，多个界面需要进行更新）
 - 不同界面的Actions都想修改同一个状态（Home.vue需要修改，Profile.vue也需要修改这个状态）
- 也就是说对于某些状态(状态1/状态2/状态3)来说只属于我们某一个试图，但是也有一些状态(状态a/状态b/状态c)属于多个试图共同想要维护的
 - 状态1/状态2/状态3你放在自己的房间中，你自己管理自己用，没问题。
 - 但是状态a/状态b/状态c我们希望交给一个大管家来统一帮助我们管理！！！
 - 没错，Vuex就是为我们提供这个大管家的工具。
- 全局单例模式（大管家）
 - 我们现在要做的就是将共享的状态抽取出来，交给我们的大管家，统一进行管理。
 - 之后，你们每个试图，按照我规定好的规定，进行访问和修改等操作。
 - 这就是Vuex背后的基本思想。

Vuex安装配置

Vuex就是一个提供多个组件间共享状态的插件

表明各种状态一般就是通过变量来记录，也就是将多个组件共享的变量放在一个对象中进行存储，并且可以做到响应式



管理什么状态呢？

| 但是，有什么状态时需要我们在多个组件间共享的呢？

- 如果你做过大型开放，你一定遇到过多个状态，在多个界面间的共享问题。
- 比如用户的登录状态、用户名称、头像、地理位置信息等等。
- 比如商品的收藏、购物车中的物品等等。
- 这些状态信息，我们都可以放在统一的地方，对它进行保存和管理，而且它们还是响应式的（待会儿我们就可以看到代码了，莫着急）。

像用户登录状态、用户名称，头像、地理位置、商品的收藏，购物车中的物品，都可以在vuex中
在命令行中输入：**npm install vuex --save** 进行安装

然后进行模块的导入（import）和使用（Vue.use）

但是配置的工作一般不放在main.js文件中

像router一样，重新创建一个文件夹，然在**store**文件夹下面

一般这个文件夹约定俗成都叫**store**，在store文件夹中创建**index.js**文件

在**index.js**文件中进行安装插件、创建对象、导出对象

注意这里创建的对象不是Vuex，而是Vuex.store，使用的时候使用的也是store

App.vue index.html * index.js main.js

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3 // 安装插件
4 Vue.use(Vuex)
5 // 创建对象
6 const store = new Vuex.Store({
7   state:{},
8   mutations:{},
9   actions:{},
10  getters:{},
11  modules:{},
12  ...
13  },
14  ...
15  },
16  ...
17  },
18  },
19  ...
20  ...
21  }
22 })
23 // 导出store对象
24 export default {
25   store
26 }
```

然后再**main.js文件**中进行挂载，在vue构造函数中添加store

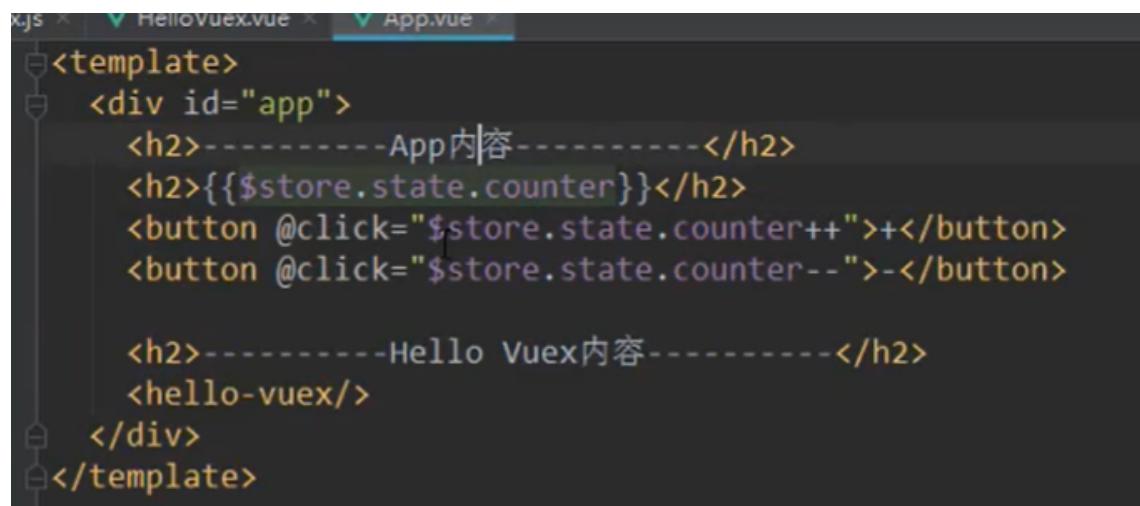
这样相当于在**Vue的原型中添加了\$store**，这样早其他组件中就可以使用store的相关东西

```
App.vue      index.html      index.js      main.js      HelloVue
1 import Vue from 'vue'
2 import App from './App'
3 import store from './store/index.js' highlighted
4
5
6 Vue.config.productionTip = false
7
8 /* eslint-disable no-new */
9 new Vue({
10   el: '#app',
11   store, highlighted
12   render: h => h(App)
13 })
14
```

Vuex对象中一般有图中几个对象

state: 记录各个组件中状态的一些变量

配置好后，在各个组件中就可以通过`$store.state`来访问vuex中的变量，当然也可以直接进行修改，如图：

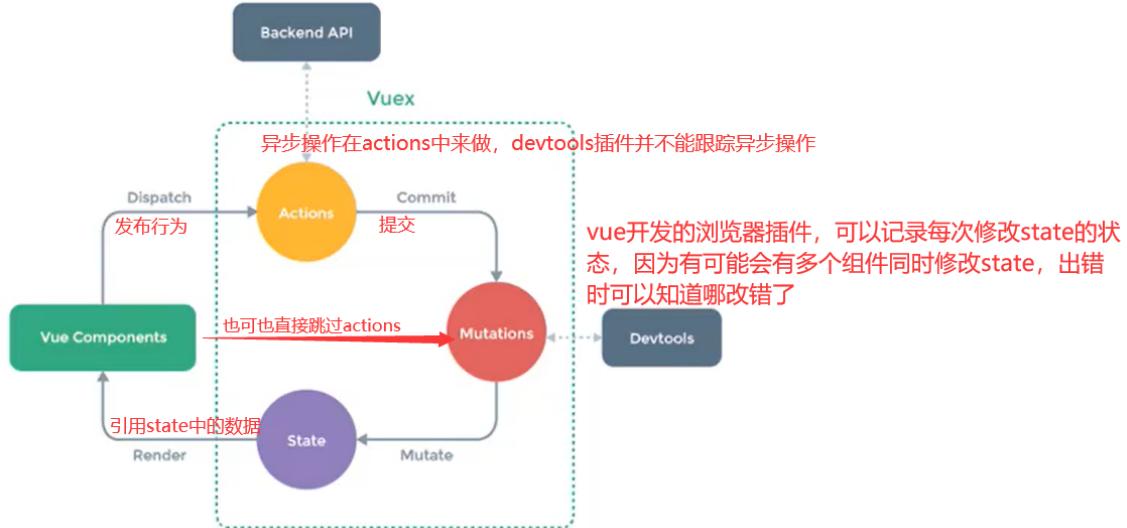


```
<template>
  <div id="app">
    <h2>-----App内容-----</h2>
    <h2>{{$store.state.counter}}</h2>
    <button @click="$store.state.counter++">+</button>
    <button @click="$store.state.counter--">-</button>

    <h2>-----Hello Vuex内容-----</h2>
    <hello-vuex/>
  </div>
</template>
```

单数官方不建议这样修改，因为官方建议按照制定好的规定来修改state

下面是官方对于修改state操作的一个流程图



devtools和mutations

devtools是Chrome浏览器上的一个插件，安装上就可以使用devtools工具来查看vue项目的一些情况

之前说过在创建的Vuex.store对象中有几个对象，其中有一个mutations

以后定义方法就可以在mutations中**定义一些方法**

以后只要是想修改state，都**至少通过mutations**来修改的，有**异步操作在通过actions**

定义的方法默认有一个**参数state**，就是上面**表示状态的对象**

```

App.vue      index.html    index.js      main.js      HelloVuex.vue
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3 // 安装插件
4 Vue.use(Vuex)
5 // 创建对象
6 const store = new Vuex.Store({
7   state: {
8     counter: 1000
9   },
10  mutations: {
11    // 定义方法, 定义的方法默认有一个参数state, 就是上面表示状态的对象
12    // 以后只要是想修改state, 都至少通过mutations来修改的, 有异步操作在通过actions
13    increment(state){
14      state.counter++
15    },
16    decrement(state){
17      state.counter--
18    }
19  },
20  actions: {

```

然后在**App组件**中调用mutations方法时，**通过this.\$store拿到score对象**，然后**通过commit**（对应上面流程图中提交这一步）**来调用score对象的mutations中定义的方法**

this.\$store.commit("方法")

```

App.vue index.html index.js main.js HelloVuex.vue
1 <template>
2 <div id="app">
3   <h2>App中的内容-----</h2>
4   <h2>{$store.state.counter}</h2>
5   <button @click="addition">+1</button>
6   <button @click="subtraction">-1</button>
7   <h2>HelloVuex中的内容-----</h2>
8   <hello-vuex></hello-vuex>
9 </div>
10 </template>
11
12 <script>
13   import HelloVuex from './components/HelloVuex.vue'
14   export default {
15     name: 'App',
16     components: {
17       HelloVuex
18     },
19     methods: {
20       addition() {
21         // 通过this.$store拿到score对象，然后通过commit（对应上面流程图中提交这一步）来调用score对象的mutations中定义的方法
22         this.$store.commit("increment")
23       },
24       subtraction() {
25         this.$store.commit("decrement")
26       }
27     }

```

打开devtools插件后，也能看到状态state中的变量的实时变化

而上一节中绑定click事件的方法state并没有实时变化

state单一状态树的理解

即一个项目只用一个store



State单一状态树

- Vuex提出使用单一状态树，什么是单一状态树呢？
 - 英文名称是Single Source of Truth，也可以翻译成单一数据源。
- 但是，它是什么呢？我们来看一个生活中的例子。
 - OK，我用一个生活中的例子做一个简单的类比。
 - 我们知道，在国内我们有很多的信息需要被记录，比如上学时的个人档案，工作后的社保记录，公积金记录，结婚后的婚姻信息，以及其他相关的户口、医疗、文凭、房产记录等等（还有很多信息）。
 - 这些信息被分散在很多地方进行管理，有一天你需要办某个业务时（比如入户某个城市），你会发现你需要到各个对应的工作地点去打印、盖章各种资料信息，最后到一个地方提交证明你的信息无误。
 - 这种保存信息的方案，不仅仅低效，而且不方便管理，以及日后的维护也是一个庞大的工作（需要大量的各个部门的人力来维护，当然国家目前已经完善我们的这个系统了）。
- 这个和我们在应用开发中比较类似：
 - 如果你的状态信息是保存到多个Store对象中的，那么之后的管理和维护等等都会变得特别困难。
 - 所以Vuex也使用了单一状态树来管理应用层级的全部状态。
 - 单一状态树能够让我们最直接的方式找到某个状态的片段，而且在之后的维护和调试过程中，也可以非常方便的管理和维护。



Getters使用详解

store中的getters跟组件中的computed很像

可以将他理解为计算属性

功能时将state中的一些数据做一些变化、操作再进行返回

跟mutations一样，getters定义的函数默认有一个参数state，就是上面表示状态的对象

下图例子，获取学生年龄大于20的人数

□ 获取学生年龄大于20的个数。

```
const store = new Vuex.Store({
  state: {
    students: [
      {id: 110, name: 'why', age: 18},
      {id: 111, name: 'kobe', age: 21},
      {id: 112, name: 'lucy', age: 25},
      {id: 113, name: 'lilei', age: 30},
    ]
  }
})
```

之前要是在computed中写的话

```
computed: {
  getGreaterAgesCount() {
    return this.$store.state.students.filter(age => age >= 20).length
  }
},
```

现在是利用store中的getters来定义函数，整合数据

```
getters: {
  greaterAgesCount: state => {
    return state.students.filter(s => s.age >= 20).length
  }
}
```

在APP.vue 中要调用getters中的函数的话，需要通过\$store.getters.函数名，函数名后面不需要加括号

上面这个函数还可以改写一下，在getters对象中的函数中可以传递一个getters参数，这个getters就是当前的getters对象本身，也就意味着在getters中的函数中可以调用上面定义过的函数作为参数，方便后面函数的编写，如图

```
getters: {
    powerCounter(state) {
        return state.counter * state.counter
    },
    more20stu(state) {
        return state.students.filter(s => s.age > 20)
    },
    more20stuLength(state, getters) {
        return getters.more20stu.length
    }
},
```

也就是说getters有两个参数，state、getters这两个，别的参数不会接收，要想传入参数，就在函数内部调用state就行了，不能将多余的参数传入

但是如果想传入参数呢？

可以在getters函数中返回一个函数，返回的函数中可以传入一个参数

App组件中在调用getters中函数时，可以传入一个参数，这个参数会被传入到函数返回的函数中去，这样就可以利用传过来的参数进行计算了

```
<h2>{{$store.getters.moreAgeStuLength}}</h2>
<h2>{{$store.getters.moreAgeStu(12)}}</h2>

},
moreAgeStu(state) {
    return function (age) {
        return state.students.filter(s => s.age > age)
    }
},
```

mutations的携带参数

vuex的store状态更新的唯一方式：提交Mutations

- Vuex的store状态的更新唯一方式：提交Mutation
- Mutation主要包括两部分：
 - 字符串的事件类型 (type)
 - 一个回调函数 (handler), 该回调函数的第一个参数就是state。
- mutation的定义方式：

```
mutations: {  
  increment(state) {  
    state.count++  
  }  
}
```

- 通过mutation更新

```
increment: function () {  
  this.$store.commit('increment')  
}
```

之前讲过mutations的基本用法，现在讲组件调用mutations中函数时，同时传入参数
在score中的**index.js**中的mutations中定义函数，除了参数state，另外的参数一同写到函数的括号中

```
mutations: {  
  // 方法  
  increment(state) {  
    state.counter++  
  },  
  decrement(state) {  
    state.counter--  
  },  
  incrementCount(state, count) {  
    state.counter += count  
  }  
},  
actions: [ ]
```

在App.vue中的methods中调用index.js中定义的函数，将参数作为commit的第二个参数传入就行，不是放在引用的函数中，语法：**this.\$store.commit("函数",参数")**

```
methods: {
    addition() {
        this.$store.commit('increment')
    },
    subtraction() {
        this.$store.commit('decrement')
    },
    addCount(count) {
        this.$store.commit('incrementCount', count)
    }
}
</script>
```

参数被称为mutations的**载荷 (payload)**



Mutation传递参数

- 在通过mutation更新数据的时候, 有可能我们希望携带一些额外的参数
 - 参数被称为是mutation的载荷(Payload)
- Mutation中的代码:

```
decrement(state, n) {
    state.count -= n
}
```

```
decrement: function () {
    this.$store.commit('decrement', 2)
}
```

- 但是如果参数不是一个呢?
 - 比如我们有很多参数需要传递.
 - 这个时候, 我们通常会以对象的形式传递, 也就是payload是一个对象.
 - 这个时候可以再从对象中取出相关的信息.

```
changeCount(state, payload) {
    state.count = payload.count
}
```

```
changeCount: function () {
    this.$store.commit('changeCount', {count: 0})
}
```

mutations的提交风格

在App组件中, 上面通过commit提交是一种普通的方式

还有一种方式, commit提交的就不是函数和参数了

commit提交的是一个对象, 对象里第一个属性**type**是函数名, 第二个属性是**参数**

```
// addCount(count) {
  // payload. 负载
  // 1.普通的提交封装
  this.$store.commit('incrementCount', count)
}

// 2.特殊的提交封装
this.$store.commit({
  type: 'incrementCount',
  count: count
}),
},
```

但是这时候count就不再是一个数字了，**而是一个对象**，可打印出来进行查看

```
[HMR] Waiting for update signal from WDS...
▼ {type: "incrementCount", count: 5} ⓘ
  count: 5
  type: "incrementCount"
▶ __proto__: Object
```

在mutations中使用参数的时候，就要将对象中的**count**取出来在进行计算

```
incrementCount(state, payload) {
  // console.log(count);
  state.counter += payload.count
},
```

Vuex数据响应式原理

(注意：以下提到的不能进行响应式的方法，在vuecli4中，进行了更新，部分内容变得可以了，根据实际情况进行判断)

在之前的mutations中修改state中的属性，vue会发生响应式更新

但是响应式需要遵守一些规则

所有响应式的**前提是所有的属性在store中已经初始化好了**

不能像这样下面这样，添加属性进行赋值

```

31     incrementCount(state, payload) {
32         // console.log(count);
33         state.counter += payload.count
34     },
35     addStudent(state, stu) {
36         state.students.push(stu)
37     },
38     updateInfo(state) {
39         // state.info.name = 'coderwhy'
40         state.info['address'] = '洛杉矶'
41     }

```

虽然查看动态state中的属性，info中确实新增了address属性，但是**不会实时进行响应，页面不会发生变化**

-----App内容: info对象的内容是否是响应式-----

```
{ "name": "kobe", "age": 40, "height": 1.98 }
```

[修改信息](#)

-----App内容-----

1000

+ - +5 +10 [添加学生](#)

-----App内容: getters相关信息-----

1000000

```
[ { "id": 111, "name": "kobe", "age": 24 }, { "id": 112, "name": "james", "age": 30 } ]
```

2

```
[ { "id": 110, "name": "why", "age": 18 }, { "id": 111, "name": "kobe", "age": 24 }, { "id": 112, "name": "james", "age": 30 } ]
```

-----Vue DevTools Inspection-----

Base State

updateInfo

Filter mutations

state

- counter: 1000
- info: Object
 - address: "洛杉矶"
 - age: 40
 - height: 1.98
 - name: "kobe"
- students: Array[4]

Filter inspected state

前面vue关于**数组响应式**的讲解中提到过，通过**索引值修改数组元素的方法不是响应式的**数组修改要做到响应式，应该用**Vue.set方法**或者**splice方法**

```

// 注意：通过索引值修改数组中的元素
this.letters[0] = 'bbbbbbb'; I
// this.letters.splice(0, 1, 'bbbbbbb')
// set(要修改的对象，索引值，修改后的值)
Vue.set(this.letters, 0, 'bbbbbbb')
}
}

```

在这里，set方法同样适用于对象

语法: `Vue.set(要修改的对象,索引值/属性名,修改后的值)`

```
updateInfo(state) {
    // state.info.name = 'coderwhy'
    // state.info['address'] = '洛杉矶'
    Vue.set(state.info, 'address', '洛杉矶')
}
```

这样就是响应式的了, `set`内部会将新增的属性添加到响应式系统里面

同样, 想要删除某个属性直接用`delete 对象.属性`是删不掉的

```
updateInfo(state) {
    // state.info.name = 'coderwhy'
    // state.info['address'] = '洛杉矶'
    // Vue.set(state.info, 'address', '洛杉矶')
    // 该方式做不到
    delete state.info.age
}
```

同样需要用到Vue中的方法: `Vue.delete方法`

```
updateInfo(state) {
    // state.info.name = 'coderwhy'
    // state.info['address'] = '洛杉矶'
    // Vue.set(state.info, 'address', '洛杉矶')
    // 该方式做不到响应式
    // delete state.info.age
    Vue.delete(state.info, 'age')
}
```

mutations的类型常量

官方推荐了一种方法, 为了防止在mutations中定义的函数在组件中使用提交commit时这些函数名或变量写错, 如下

index.js文件中

```

1 export const UPDATE_INFO = 'UPDATE_INFO'
2
3 import Vuex from 'vuex'
4 import Vue from 'vue'
5 import * as types from './mutation-types'
6
7 Vue.use(Vuex)
8
9 const store = new Vuex.Store({
10   state: {
11     info: {
12       name: 'why', age: 18
13     }
14   },
15   mutations: {
16     [types.UPDATE_INFO](state, payload) {
17       state.info = {...state.info, 'height': payload.height}
18     }
19   }
20 })
21
22 export default store

```

组件中进行commit提交时

```

<script>
  import {UPDATE_INFO} from "./store/mutation-types"
  ...
  export default {
    name: 'App',
    components: {
    },
    computed: {
      info() {
        return this.$store.state.info
      }
    },
    methods: {
      updateInfo() {
        this.$store.commit(UPDATE_INFO, {height: 1.88})
      }
    }
  }
</script>

```

actions使用详解

上面使用mutations时进行的都是**同步操作**，在满足响应式操作时，所有的修改都会实时地体现在state中

但是如果在mutations中使用的是异步操作，devtools工具不会实时地跟踪状态的变化

异步操作的函数将state中的变量进行修改，页面中发生变化，但是**devtools中监测的值并不改变**

```

1
2   updateInfo(state) {
3     // state.info.name = 'coderwhy'
4     setTimeout(() => {
5       state.info.name = 'coderwhy'
6     }, 1000)

```

-----App内容: info对象的内容是否是响应式-----

+
{ "name": "coderwhy", "age": 40, "height": 1.98 }
修改信息

-----App内容-----

1000

+ - +5 +10 添加学生

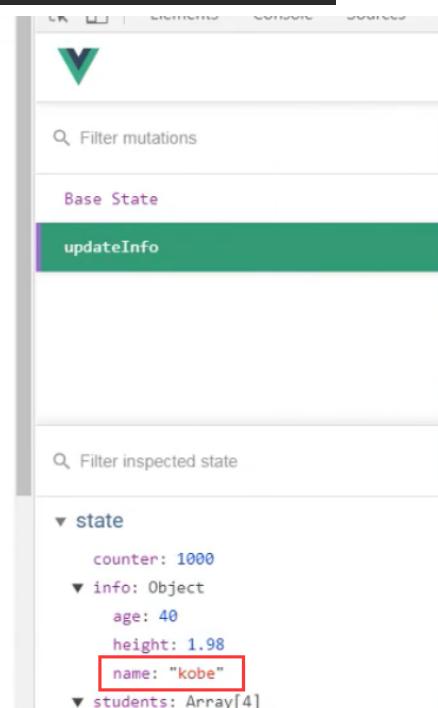
-----App内容: getters相关信息-----

1000000

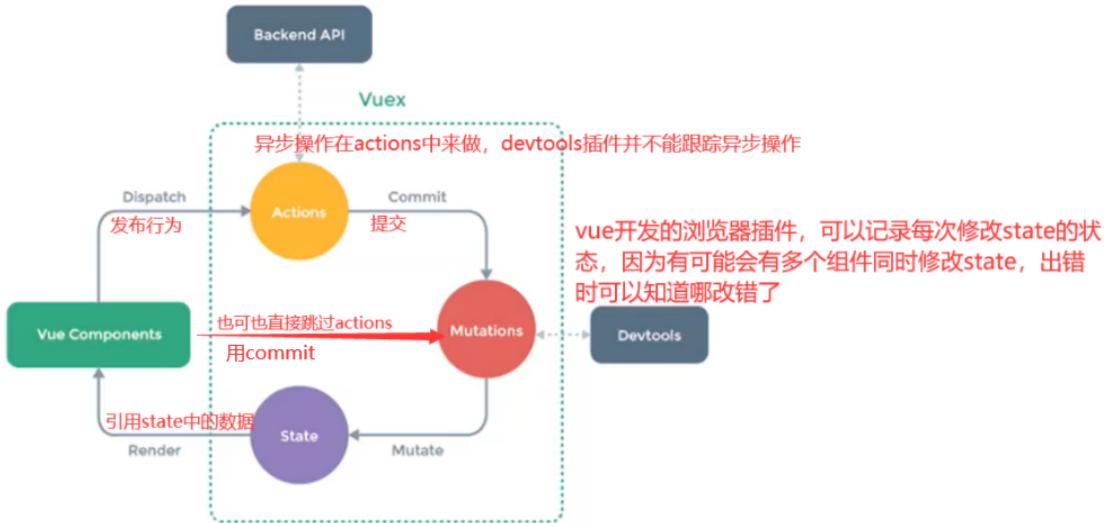
[{ "id": 111, "name": "kobe", "age": 24 }, { "id": 112, "name": "james", "age": 30 }]

2

[{ "id": 110, "name": "why", "age": 18 }, { "id": 111, "name": "kobe", "age": 24 }, { "id": 112,



所以异步操作官方要求我们放到actions中去做



注意：

之前只用mutations时（中间横着这条红线），相当于组件中使用commit提交到mutations中，mutations中函数完成对state中变量的修改

现在相当于中间加上一个actions，组件中使用dispatch来发布给actions，actions中的函数通过commit进行提交给mutations，然后mutations中的函数进行state的修改

在score中的index.js文件中定义actions

actions中定义异步操作的函数

跟mutations中函数默认参数state相似，actions中函数也有一个默认参数：context(上下文)

在这里context代表的是score，然后在actions中的函数中进行commit提交

异步操作的流程

(这里为了理解将这一步放在前面，实际上流程是先dispatch再commit)

在mutations中定义好修改state的一个函数

```
index.js
1 // ...
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

```
App.vue
1 <template>
2   <div>
3     <h1>Hello World</h1>
4     <p>{{ count }}</p>
5     <button @click="decrement">-
6     <button @click="incrementCount">+</button>
7     <br>
8     <ul>
9       <li>{{ student }}</li>
10    </ul>
11  </div>
12</template>
13<script>
14 import { mapState, mapMutations } from 'vuex'
15
16 export default {
17   computed: mapState(['count']),
18   methods: mapMutations(['decrement', 'incrementCount'])
19 }
20</script>
```

然后在actions中将上面mutations中定义的函数进行commit提交

```
index.js
1 // ...
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

```
App.vue
1 <template>
2   <div>
3     <h1>Hello World</h1>
4     <p>{{ count }}</p>
5     <button @click="decrement">-</button>
6     <button @click="incrementCount">+</button>
7     <br>
8     <ul>
9       <li>{{ student }}</li>
10    </ul>
11  </div>
12</template>
13<script>
14 import { mapState, mapMutations } from 'vuex'
15
16 export default {
17   computed: mapState(['count']),
18   methods: mapMutations(['decrement', 'incrementCount'])
19 }
20</script>
21<script setup>
22 import { ref } from 'vue'
23 import { useStore } from 'vuex'
24
25 const store = useStore()
26
27 const address = ref('北京')
28 const age = ref(18)
29 const info = ref({ name: 'coderwhy', address, age })
30
31 const student = ref('张三')
32
33 const count = ref(0)
34
35 const mutations = {
36   decrement(state) {
37     state.count--
38   },
39   incrementCount(state, payload) {
40     // console.log(count);
41     state.count += payload.count
42   },
43   addStudent(state, stu) {
44     state.students.push(stu)
45   },
46   updateInfo(state) {
47     // state.info.name = 'coderwhy'
48     setTimeout(() => {
49       state.info.name = 'coderwhy'
50     }, 1000)
51   }
52 }
53
54
55
56
57
58
59
60
61
62
63
64
65
66
```

在组件中如果调用这个函数的话，需要使用dispatch进行发布（之前是commit提交）

```
index.js
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

```
App.vue
type: 'INCREMENTCOUNT',
count
})
},
addStudent() {
const stu = {id: 114, name: 'alan', age: 35}
this.$store.commit('addStudent', stu)
},
updateInfo() {
// this.$store.commit('updateInfo')
this.$store.dispatch('aUpdateInfo')
}
```

参数传递

同样，actions也可以携带参数，

在组件中调用函数传实参时，**将实参和函数名作为dispatch的两个参数**，这样来传参

在actions中定义函数，可以像正常函数接收形参在括号里接收就行

同样可以传一个对象作为参数

```
index.js
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
```

```
App.vue
type: 'INCREMENTCOUNT',
count
})
},
addStudent() {
const stu = {id: 114, name: 'alan', age: 35}
this.$store.commit('addStudent', stu)
},
updateInfo() {
// this.$store.commit('updateInfo')
this.$store.dispatch('aUpdateInfo', '我是payload')
}

</script>

actions: {
// context: 上下文
aUpdateInfo(context, payload) {
setTimeout(() => {
context.commit('updateInfo')
console.log(payload);
}, 1000)
}
},
```

修改state成功的提示

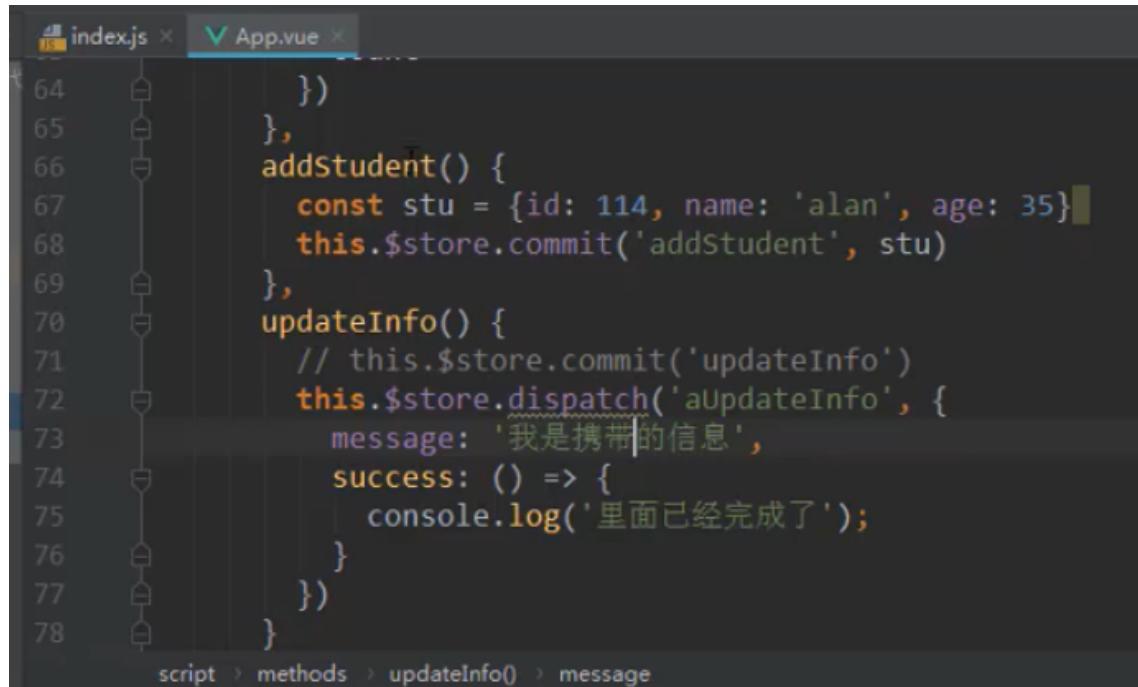
组件在使用dispatch将state进行修改，如果修改成功，我们希望可以得到一个告知

一般我们认为只要调用index.js文件中mutations中的函数，执行了其中的commit就算修改成功了

可以这么做：

在组件中除了函数名，再往dispatch中传递一个对象作为参数

对象中有message属性作为数据，还有一个函数方法



```
index.js x App.vue x
64      })
65    },
66    addStudent() {
67      const stu = {id: 114, name: 'alan', age: 35}
68      this.$store.commit('addStudent', stu)
69    },
70    updateInfo() {
71      // this.$store.commit('updateInfo')
72      this.$store.dispatch('aUpdateInfo', {
73        message: '我是携带的信息',
74        success: () => {
75          console.log('里面已经完成了');
76        }
77      })
78    }
}

script > methods > updateInfo() > message
```

在index.js中的mutations函数中进行如下定义，



```
index.js x App.vue x
55      // delete state.info.age
56      // Vue.delete(state.info, 'age')
57    },
58  },
59  actions: {
60    // context: 上下文
61    aUpdateInfo(context, payload) {
62      setTimeout(() => {
63        context.commit('updateInfo')
64        console.log(payload.message);
65        payload.success()
66      }, 1000)
67    },
68  },
}

commit提交
使用数据进行一些操作
调用参数对象中的函数，用来打印执行成功的消息
```

这样在组件中调用actions中函数，actions中再通过commit调用mutations中函数，完成这一步，就会打印相关信息，即可得到通知

更加优雅的写法：利用Promise

这里先不要管mutations，否则容易乱

在actions中将异步操作放到Promise中，并且通过return将Promise对象作为函数返回值返回。

Promise中有一个执行成功后的操作，通过resolve()函数和then()函数完成

在setTimeout内部前几行，先commit给mutations，然后写**resolve()函数**表示执行成功，执行成功后的操作放在**then()函数**中去做

但是**then**函数在这里并不是写在**index.js**中的，而是写在**App组件中**，在用**dispatch**调用**actions**后，就可以跟上.then()，then函数中进行执行成功消息的打印

为什么**then**可以写在这里呢？

因为**actions**中的这个函数是有返回值的，在**App组件**中调用了这个函数，自然获得了这个**Promise对象**作为返回值，原本**then()**就是接在**Promise对象**后面的

```
index.js x App.vue x
62     //   setTimeout(() => {
63     //     context.commit('updateInfo')
64     //     console.log(payload.message);
65     //     payload.success()
66     //   }, 1000)
67   },
68   aUpdateInfo(context, payload) {
69     return new Promise((resolve, reject) => {
70       setTimeout(() => {
71         context.commit('updateInfo');
72         console.log(payload);
73
74         resolve('1111111')
75       }, 1000)
76     })
}
store > actions > aUpdateInfo() > callback for Promise() > callback for setTimeout()
```

获得一个Promise对象作为调用actions中函数后的返回值

Promise后面本来就是接then()了
这里也就不难理解为什么可以接then()了

```
index.js x App.vue x
71   // this.$store.commit('updateInfo')
72   // this.$store.dispatch('aUpdateInfo', {
73   //   message: '我是携带的信息',
74   //   success: () => {
75   //     console.log('里面已经完成了');
76   //   }
77   // })
78   this.$store
79     .dispatch('aUpdateInfo', '我是携带的信息')
80     .then(res => {
81       console.log('里面完成了提交');
82       console.log(res);
83     })
84   }
85 }
```

modules使用详解



认识Module

■ Module是模块的意思, 为什么在Vuex中我们要使用模块呢?

- Vue使用单一状态树,那么也意味着很多状态都会交给Vuex来管理.
- 当应用变得非常复杂时,store对象就有可能变得相当臃肿.
- 为了解决这个问题, Vuex允许我们将store分割成模块(Module), 而每个模块拥有自己的state、mutations、actions、getters等

■ 我们按照什么样的方式来组织模块呢?

- 我们来看左边的代码

```
const moduleA = {  
  state: { ... },  
  mutations: { ... },  
  actions: { ... },  
  getters: { ... }  
}  
  
const moduleB = {  
  state: { ... },  
  mutations: { ... },  
  actions: { ... }  
}  
  
const store = new Vuex.Store({  
  modules: {  
    a: moduleA, ↴  
    b: moduleB  
  }  
})  
  
store.state.a // -> moduleA 的状态  
store.state.b // -> moduleB 的状态
```

一般就两层就可以了, 多了反倒太复杂

像实例中那样, 一般不会在modules中详细定义各个模块, 一般都将模块对象抽离出来, 在外面定义

然后这些模块通过**属性名:模块名**, 作为modules的属性放到modules里面

```

const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态

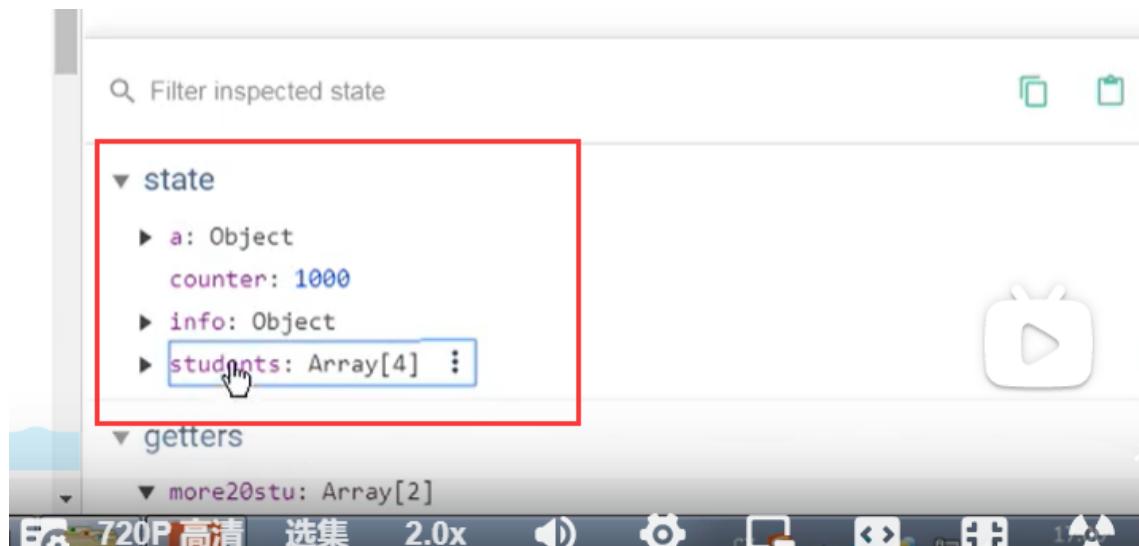
```

模块中的state的使用

上图中，要想拿某个模块中的state，不是通过`store.a.state`，而是**通过`store.state.a`**来拿到不同模块中的state

为什么呢？因为modules在封装state时，其实是将他们都放在里一个state当中（仍然保持了单一状态树）

state中多了几个对象，每个对象就代表不同模块，每个对象里面是不同模块中的状态变量

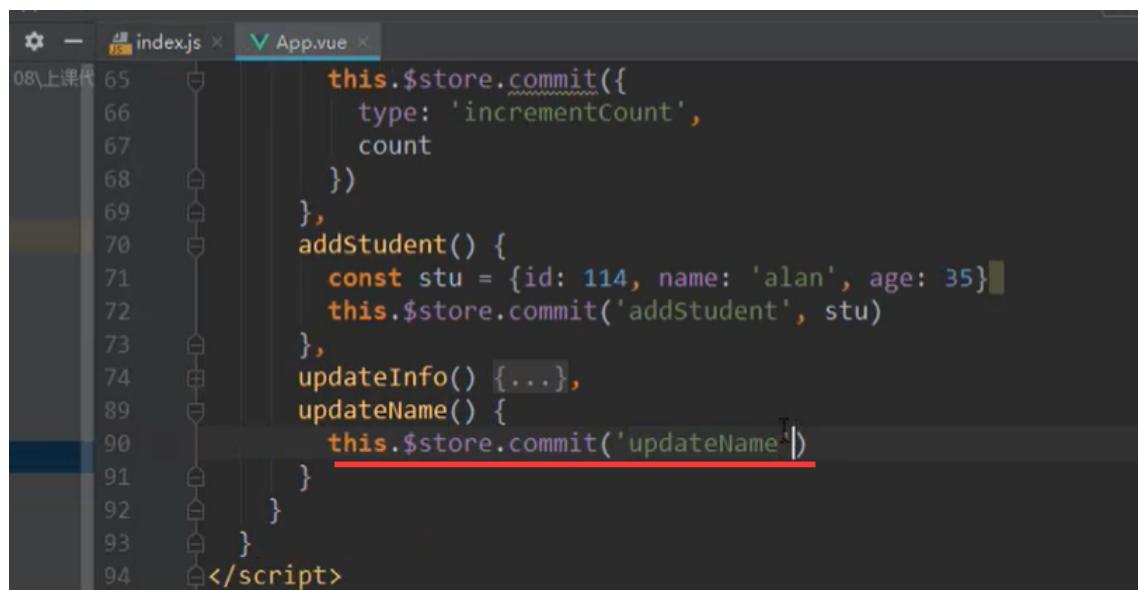


但是除了state，

模块的mutations中函数的调用

mutations中定义的函数，在组件通过commit调用时，就按正常调用就可以了，不用区分模块，因为函数名是不允许一样的，他只是优先在store外层找，找不到就去模块中找

```
// 2. 创建对象
const moduleA = {
  state: {
    name: 'zhangsan'
  },
  mutations: {
    updateName(state, payload) {
      state.name = payload
    }
  },
  actions: {},
  getters: {}
}
```



```
08\上课代码
index.js
65 this.$store.commit({
66   type: 'incrementCount',
67   count
68 })
69 },
70 addStudent() {
71   const stu = {id: 114, name: 'alan', age: 35}
72   this.$store.commit('addStudent', stu)
73 },
74 updateInfo() {...},
89 updateName() {
90   this.$store.commit('updateName')
91 }
92 }
93 }
94 </script>
```

模块中getters的使用

getters也没有模块的特殊用法，和mutations一样，正常使用就可以了，不同模板中getters中函数名也不重复就行

特殊的地方在于，在模块的getters函数中，还可以有**第三个参数：rootState**

这个参数是用来在getters函数中使用store外层state中变量，**rootState代表的时外层的state**

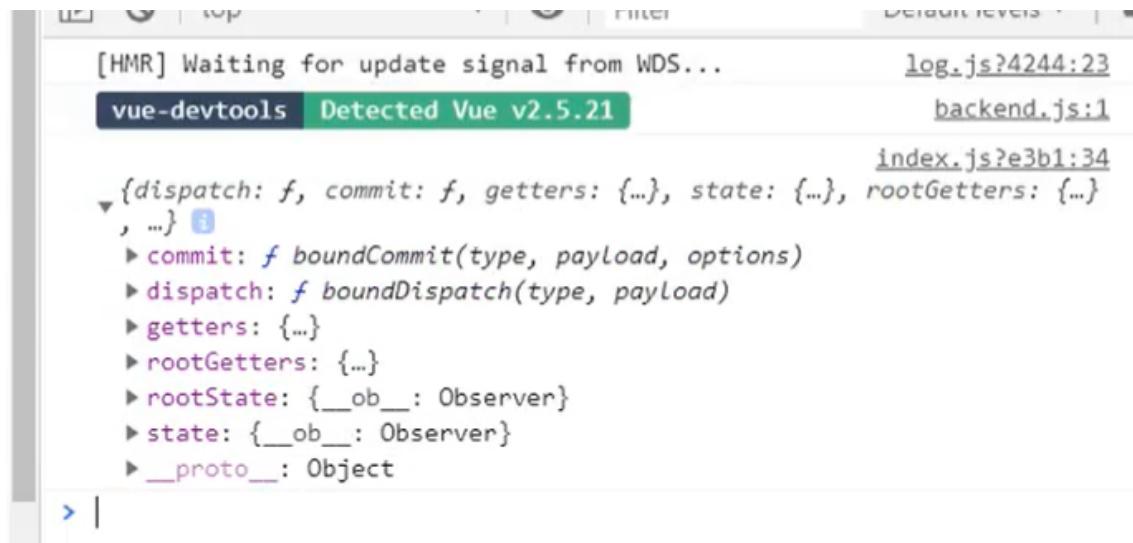
```
App.vue
getters: {
  fullname(state) {
    return state.name + '1111'
  },
  fullname2(state, getters) {
    return getters.fullname + '2222'
  },
  fullname3(state, getters, rootState) {
    return getters.fullname2 + rootState.counter
  }
},
actions: {}
```

模块中actions的使用

在模块中的actions中定义函数，默认的参数context，这是**context就不是store对象了**

而是本模块对象，就比如通过context.commit调用的mutations中的函数不是外层的mutations，而是模板本身的mutations

下面是模板中context打印出来



context.rootGetters拿的时外层的getters对象

context.rootState拿的时外层state对象

ps：一个例子

默认actions中的函数参数是**context**，将函数参数的地方**用{state, commit, rootState}来代替**，就相当于将context进行**解构**，取出里面的三个属性

■ actions的写法呢? 接收一个context参数对象

- 局部状态通过 context.state 暴露出来, 根节点状态则为 context.rootState

```
const moduleA = {
  // ...
  actions: {
    incrementIfOddOnRootSum ({ state, commit, rootState }) {
      if ((state.count + rootState.count) % 2 === 1) {
        commit('increment')
      }
    }
  }
}
```

store文件夹的目录组织



- 当我们的Vuex帮助我们管理过多的内容时, 好的项目结构可以让我们的代码更加清晰.

```
├── index.html
├── main.js
├── api
|   └── ... # 抽取出API请求
├── components
|   ├── App.vue
|   └── ...
└── store
    ├── index.js      # 我们组装模块并导出 store 的地方
    ├── actions.js    # 根级别的 action
    ├── mutations.js  # 根级别的 mutation
    └── modules
        ├── cart.js    # 购物车模块
        └── products.js # 产品模块
```

如果以后代码量多了后, 所有的js代码全写到score中的index.js文件中有点太乱

官方建议将各个部分进行抽离

state在store对象外面进行定义, 然后将定义的对象放到store中, 不要在store对象中定义state中的内容, 但是并**不用抽离到另一个文件中去**

modules、actions、getters、mutations这些部分都抽离成单独的文件

其中modules做成文件夹, 文件夹里面每个模块一个js文件

```
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 import mutations from './mutations' I
5 import actions from './actions'
6 import getters from './getters'
7 import moduleA from './modules/moduleA'
8
9 // 1.安装插件
10 Vue.use(Vuex)
11
12 // 2.创建对象
13 const state = {
14   counter: 1000,
15   students: [
16     {id: 110, name: 'why', age: 18},
17     {id: 111, name: 'kobe', age: 24},
18     {id: 112, name: 'james', age: 30},
19     {id: 113, name: 'curry', age: 10}
20   ],
21   info: {
22     name: 'kobe',
23     age: 40,

```

```
const store = new Vuex.Store({
  state,
  mutations,
  actions,
  getters,
  modules: {
    a: moduleA
  }
})

// 3.导出store独享
export default store
```

网络模块的封装



选择什么网络模块?

■ Vue中发送网络请求有非常多的方式,那么,在开发中,如何选择呢?

■ 选择一: 传统的Ajax是基于XMLHttpRequest(XHR)

■ 为什么不用它呢?

- 非常好理解,配置和调用方式等非常混乱.
- 编码起来看起来就非常笨重.
- 所以真实开发中很少直接使用,而是使用jQuery-Ajax

■ 选择二: 在前面的学习中,我们经常会使用jQuery-Ajax

- 相对于传统的Ajax非常好用.

■ 为什么不选择它呢?

- 首先,我们先明确一点:在Vue的整个开发中都是不需要使用jQuery了.
- 那么,就意味着为了方便我们进行一个网络请求,特意引用一个jQuery,你觉得合理吗?
- jQuery的代码1w+行.
- Vue的代码才1w+行.
- 完全没有必要为了用网络请求就引用这个重量级的框架.

■ 选择三: 官方在Vue1.x的时候,推出了Vue-resource.

- Vue-resource的体积相对于jQuery小很多.
- 另外Vue-resource是官方推出的.

■ 为什么不选择它呢?

- 在Vue2.0退出后,Vue作者就在GitHub的Issues中说明了去掉vue-resource,并且以后也不会再更新.
- 那么意味着以后vue-resource不再支持新的版本时,也不会再继续更新和维护.
- 对以后的项目开发和维护都存在很大的隐患.

■ 选择四: 在说明不再继续更新和维护vue-resource的同时,作者还推荐了一个框架: axios为什么不用它呢?

- axios有非常多的优点,并且用起来也非常方便.
- 稍后,我们对他详细学习.



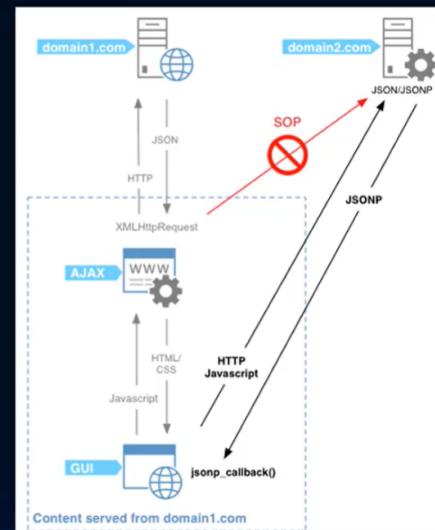
jsonp

■ 在前端开发中,我们一种常见的网络请求方式就是JSONP

□ 使用JSONP最主要的原因往往是为了解决跨域访问的问题.

■ JSONP的原理是什么呢?

- JSONP的核心在于通过<script>标签的src来帮助我们请求数据.
- 原因是我们的项目部署在domain1.com服务器上时,是不能直接访问domain2.com服务器上的资料的.
- 这个时候,我们利用<script>标签的src帮助我们去服务器请求到数据,将数据当做一个javascript的函数来执行,并且执行的过程中传入我们需要的json.
- 所以,封装jsonp的核心就在于我们监听window上的jsonp进行回调时的名称.



```
let count = 1;
export default function originPJSONP(option) {
  // 1. 从传入的option中提取URL
  const url = option.url;

  // 2. 在body中添加script标签
  const body = document.getElementsByTagName('body')[0];
  const script = document.createElement('script');

  // 3. 内部生产一个不重复的callback
  const callback = 'jsonp' + count++;

  // 4. 监听window上的jsonp的调用
  return new Promise((resolve, reject) => {
    try {
      window[callback] = function(result) {
        body.removeChild(script);
        resolve(result);
      }
      const params = handleParam(option.data);
      script.src = url + '?callback=' + callback + params;
      body.appendChild(script);
    } catch(e) {
      body.removeChild(script);
      reject(e);
    }
  })
}
```

```
function handleParam(data) {
  let url = '';
  for (let key in data) {
    let value = data[key] !== undefined ? data[key] : '';
    url += `&${key}=${encodeURIComponent(value)}`;
  }
  return url
}
```



为什么选择axios?

这是也个

■ 为什么选择axios? 作者推荐和功能特点

尤小右 ★ 4
2016-11-3 22:38 来自 微博 weibo.com
公告一下：以后 vue-resource 不再是官方推荐的 ajax 库了，推荐用 axios。详见
英文博客：[网页链接](#)

收藏 62 | 转 35 | 评论 61

■ 功能特点:

- 在浏览器中发送 XMLHttpRequests 请求
- 在 node.js 中发送 http请求
- 支持 Promise API
- 拦截请求和响应
- 转换请求和响应数据
- 等等

■ 补充: axios名称的由来? 个人理解

- 没有具体的翻译.
- axios: ajax i/o system.



axios请求方式

■ 支持多种请求方式:

- axios(config)
- axios.request(config)
- axios.get(url[, config])
- axios.delete(url[, config])
- axios.head(url[, config])
- axios.post(url[, data[, config]])
- axios.put(url[, data[, config]])
- axios.patch(url[, data[, config]])

axios的基本使用

axios框架的安装: **npm install axios --save**

安装好后main.js中import导入

然后就可以在axios中直接使用了

在axios中传入相关的配置即网络请求 (config)

相关的配置和网络请求是一个对象

```
// axios内部支持promise, 可以使用resolve、then等等
// 内部的resolve封装好了
// 可以直接在拿到数据后执行then()函数
axios({
  url: 'http://123.207.32.32:8000/home/multidata',
  // 请求的类型
  methods:'get'
}).then(res => {
  console.log(res);
})

axios({
  url:'http://123.207.32.32:8000/home/data',
  // 专门针对get请求的参数拼接
  params:{
    type:'pop',
    page:'1'
  }
}).then(res => {
  console.log(res);
})
```

axios发送并发请求

之前讲到过，如果有多次网络请求，想要在多次网络请求都收到后在进行后续操作

使用promise可以完成这一要求：Promise.all

axios框架已经提供了发送并发请求并且都到达后进行相应处理的**API**

使用**axios.all方法**，在方法中传入两个（或多个）axios请求

然后在最后使用then()函数进行完成请求后的操作

返回的结果是一个数组

```
axios.all([
  //第一次请求
  axios({
    url:'http://123.207.32.32:8000/home/multidata',
  }),
  //第二次请求
  axios({
    url:'http://123.207.32.32:8000/home/data',
    params:{
      type:'sell',
      page:'5'
    }
})
```

```
]).then(result => {
  console.log(result);
})
```

上面返回的结果是一个数组，介个网络请求的结果写在了一起

如果想在收到返回结果时**将数组中的内容分开**，可以在then中使用**axios.spread方法**

```
axios.all([
  axios({
    //路径拼接
    baseURL: 'http://123.207.32.32:8000',
    url: '/home/multidata',
  }),
  axios({
    //路径拼接
    baseURL: 'http://123.207.32.32:8000',
    url: '/home/data',
    params: {
      type: 'sell',
      page: '5'
    }
  })
]).then(axios.spread((res1, res2) => {
  console.log(res1);
  console.log(res2);
}))
```

配置信息相关

在下面的示例中，我们的 baseURL、timeout是固定的

事实上，在开发中可能很多参数都是固定的

这个时候我们可以进行些抽取也可以利用**axios的全局配置**

```
axios.all([
  axios({
    baseURL: 'http://123.207.32.32:8000',
    timeout:5000,
    url: '/home/multidata',
  }),
  axios({
    baseURL: 'http://123.207.32.32:8000',
    timeout:5000,
    url: '/home/data',
    params: {
      type: 'sell',
      page: '5'
    }
  })
]).then(axios.spread((res1, res2) => {
```

```
        console.log(res1);
        console.log(res2);
    }))
}
```

axios中可以用**axios.defaults对象**来定义全局配置

往axios.defaults中添加属性就可

```
//添加全局变量
axios.defaults.baseURL='http://123.207.32.32:8000';
axios.defaults.timeout=5000;
axios.all([
    axios({
        url: '/home/multidata',
    }),
    axios({
        url: '/home/data',
        params: {
            type: 'sell',
            page: '5'
        }
    })
]).then(axios.spread((res1,res2) => {
    console.log(res1);
    console.log(res2);
}))
```

常见的全局配置选项



常见的配置选项

其实可以混着用，后端接收的时候
http请求中
post

- 请求地址
 - url: '/user',
- 请求类型
 - method: 'get',
- 请根路径
 - baseURL: 'http://www.mt.com/api',
- 请求前的数据处理
 - transformRequest:[function(data){}],
- 请求后的数据处理
 - transformResponse: [function(data){}],
- 自定义的请求头
 - headers:{'x-Requested-With':'XMLHttpRequest'},
- URL查询对象
 - params:{ id: 12 },
- 查询对象序列化函数
 - paramsSerializer: function(params){ }
- request body
 - data: { key: 'aa'},
- 超时设置
 - timeout: 1000,
- 跨域是否带Token
 - withCredentials: false,
- 自定义请求处理
 - adapter: function(resolve, reject, config){},
- 身份验证信息
 - auth: { uname: "", pwd: '12'},
- 响应的数据格式 json / blob /document /arraybuffer / text / stream
 - responseType: 'json',

axios实例

上面的例子使用的都是**全局的axios对应的配置进行网络请求**

但很多时候，不同的axios请求要求的配置不一样，只用同一个全局配置就不合适了

这时候需要**创建多个axios实例**

axios中有一个**create方法**可以创建实例

在每一个实例中传入属于这个实例统一的**配置对象**

每次使用这个实例时都传入其他具体的配置

具体操作如下：

```
// 创建实例1
const instance1 = axios.create({
  baseURL: 'http://123.207.32.32:8000',
  timeout: 5000
})
// 使用实例
instance1({
  url: '/home/multidata'
}).then(res => {
  console.log(res)
})
instance1({
  url: '/home/data',
  params: {
    type: 'pop',
    page: 1
  }
}).then(res => {
  console.log(res)
})
// 创建实例2
const instance2 = axios.create({
  baseURL: 'http://222.111.33.33:8000',
  timeout: 1000
})
```

axios模块封装

下面一个简单的例子，也就是我们以后进行网络请求的简陋雏形，只不过没有进行封装

App组件的进行网络请求

使用**生命周期函数create()函数**，在组件被创建出来出来，且data和methods初始化好了，在这个钩子函数中进行网络请求，并在请求成功后在**then函数**中将返回的数据赋值给data中的变量，即可完成一个简单的网络请求

```
1 <template>
2   <div id="app">
3     <div>{{result}}</div>
4   </div>
5 </template>
6
7 <script>
8   import axios from 'axios'
9
10  export default {
11    name: 'App',
12    components: {
13      },
14    data() {
15      return {
16        result: ''
17      }
18    },
19    created() {
20      axios({
21        url: 'http://123.207.32.32:8000/home/multidata'
22      }).then(res => {
23        console.log(res);
24        this.result = res;
25      })
26    }
27  }
28 </script>
```

来子组件中同样进行网络请求和相关操作

```
1 <template>
2   <h2>{{categories}}</h2>
3 </template>
4
5 <script>
6   import axios from 'axios'
7
8   export default {
9     name: "HelloWorld",
10    data() {
11      return {
12        categories: ''
13      }
14    },
15    created() {
16      axios({
17        url: 'http://123.207.32.32:8000/category'
18      }).then(res => {
19        this.categories = res;
20      })
21    }
22  }
23 </script>
```

将子组件引用到父组件中

```
App.vue
1 <template>
2   <div id="app">
3     <div>{{result}}</div>
4     <h2>-----|</h2>
5     <hello-world/>
6   </div>
7 </template>
8
9 <script>
10    import HelloWorld from './components/HelloWorld'
11    import axios from 'axios'
12
```

但是这样做有很大问题，以后组件多了，么个组件需要进行网络请求，每次都要引入axios框架
以后框架如果不再进行支持，项目重构的时候变得异常艰难

所以建议将axios进行一个封装

可以在src文件夹中创建**network文件夹**，在network中创建js文件：**request.js文件**

在request.js文件中**导入axios模块**

然后**创建一个函数并导出**，在这个**函数里面创建axios实例**，然后**使用创建的实例进行真正的网络请求**，通过函数参数传递过来的配置信息，来配置网络请求

```
request.js
1 import axios from 'axios'
2
3 export function request(config, success, failure) {
4   // 1. 创建axios的实例
5   const instance = axios.create({
6     baseURL: 'http://123.207.32.32:8000',
7     timeout: 5000
8   })
9
10  // 发送真正的网络请求
11  instance(config)
12    .then(res => {
13      // console.log(res);
14      success(res);
15    })
16    .catch(err => {
17      // console.log(err);
18      failure(err)
19    })
20}
```

为了在请求成功后进行进一步的操作，将请求结果回调回去，这里有几种方法，**①、②不经常用，
③是标准做法**

①

上面这个图就是第一种方法，在函数中再传进来两个参数，**两个参数是两个函数，分别是请求成功时执行的函数，和请求失败时执行的函数**

此**函数内容在App父组件的mian.js文件中进行定义**，当然前提是**要导入封装的request模块**，这样才可以使用导出的request函数

```
// baseURL: 'http://222.111.33.33:8000',
// timeout: 10000,
// // headers: {}
// })

// 5.封装request模块
import {request} from "./network/request";

request({
  url: '/home/multidata',
}, res => {
  console.log(res);
}, err => {
  console.log(err);
})
```

②

或者函数值传过来一个config参数，组件调用函数的时候**将执行成功与失败的函数放在config对象里面传进来就行**

配置信息再单独放到对象的baseConfig属性中

在request.js文件中进行网络请求时，**函数中参数是config，网络请求的参数是config.baseConfig**

```
request({
  baseConfig: {
    },
    success: function (res) {
      },
    failure: function (err) {
      }
})
```

```
1 import axios from 'axios'
2
3 export function request(config) {
4     // 1. 创建axios的实例
5     const instance = axios.create({
6         baseURL: 'http://123.207.32.32:8000',
7         timeout: 5000
8     })
9
10    // 发送真正的网络请求
11    instance(config.baseConfig)
12        .then(res => {
13            // console.log(res);
14            config.success(res);
15        })
16        .catch(err => {
17            // console.log(err);
18            config.failure(err)
19        })
20    }
21 }
```

③

上面两个操作不太常用，真正标准做法，还是使用Promise来将请求结果返回给组件中（之前也使用过这个方式）

在request.js文件中，在导出的函数中，将网络请求的异步操作放在Promise对象中，通过函数的返回值返回

```
2
3     export function request(config) {
4         return new Promise((resolve, reject) => {
5             // 1. 创建axios的实例
6             const instance = axios.create({
7                 baseURL: 'http://123.207.32.32:8000',
8                 timeout: 5000
9             })
10
11            // 发送真正的网络请求
12            instance(config)
13                .then(res => {
14                    resolve(res)
15                })
16                .catch(err => {
17                    reject(err)
18                })
19
20        })
21    }
```

然后在父组件的main.js文件中导入模块并调用request函数，传入config参数，并且会有返回值，可以在request函数后面紧接着then()函数和catch()函数，

但教程中这种方法我觉得有个问题，instance中的then里面还有一个resolve，catch里面还有一个reject，这明显混乱了

```
133 //     baseConfig: {
134 //     },
135 //     success: function (res) {
136 //     },
137 //     failure: function (err) {
138 //     }
139 //   }
140 // }
141 // })
142 // })
143
144 request({
145   url: '/home/multidata'
146 }).then(res => {
147   console.log(res);
148 }).catch(err => {
149   console.log(err);
150 })
151
```

改进：

发现一个问题，前面已经讲过了，通过axios.create创建的axios实例本身就支持Promise。前面使用实例时也用到了，也就是说instance实例对象一直作为一个函数来使用的，其返回值就是一个Promise对象。

所以上面这种将网络请求放在Promise中太多此一举了。

request函数直接返回instance就行了。

```
1 import axios from 'axios'
2
3 export function request(config) {
4   // 1. 创建axios的实例
5   const instance = axios.create({
6     baseURL: 'http://123.207.32.32:8000',
7     timeout: 5000
8   })
9
10 // 发送真正的网络请求
11 return instance(config)
12 }
```

在main.js文件中发送网络请求时，直接如下图：

```
09\课堂代 133 //     baseConfig: {  
134 //  
135 //     },  
136 //     success: function (res) {  
137 //  
138 //     },  
139 //     failure: function (err) {  
140 //  
141 // }  
142 // })|  
143  
144 request({  
145     url: '/home/multidata'  
146 }).then(res => {  
147     console.log(res);  
148 }).catch(err => {  
149     console.log(err);  
150 })|
```

axios拦截器

有四个拦截器，**请求成功、请求失败、响应成功、响应失败**

拦截器就是在这些过程中做一个拦截，每个阶段可以做一些额外的操作



如何使用拦截器？

- axios提供了拦截器，用于我们在发送每次请求或者得到相应后，进行对应的处理。
- 如何使用拦截器呢？

```
// 配置请求和响应拦截  
instance.interceptors.request.use(config => {  
    console.log('来到了request拦截success中');  
    return config  
}, err => {  
    console.log('来到了request拦截failure中');  
    return err  
})  
  
instance.interceptors.response.use(response => {  
    console.log('来到了response拦截success中');  
    return response.data  
}, err => {  
    console.log('来到了response拦截failure中');  
    return err  
})
```

拦截器可以**拦截全局的axios**, 直接**axios.interceptors**

但一般是使用**实例的拦截器**, 上面创建的实例instance, **instance.interceptors**

语法:

请求拦截: **实例.interceptors.request.use(config =>{},err =>{})**

响应拦截: **实例.interceptors.response.use(response =>{},err =>{})**

请求拦截

其中箭头函数的这些变量名称都是可以自己命名的

第一个箭头函数表示**请求成功的返回函数**, **默认参数**就是一些详细的配置信息, 所以参数一般写config, 打印config如下

第二个函数表示**请求失败的返回函数**, 默认参数是**错误信息**

[request.js?846](#)

```
▼ Object ⓘ
  ► adapter: f xhrAdapter(config)
  baseURL: "http://123.207.32.32:8000"
  ► headers: {common: {...}, delete: {...}, get: {...}, head: {...}, post: maxContentLength: -1
  method: "get"
  timeout: 5000
  ► transformRequest: {0: f}
  ► transformResponse: {0: f}
  url: "/home/multidata"
  ► validateStatus: f validateStatus(status)
  xsrfCookieName: "XSRF-TOKEN"
  xsrfHeaderName: "X-XSRF-TOKEN"
  ► __proto__: Object
```

但是请求拦截器使用时, **一定要记得将拦截到的config在通过函数返回出去**, 不然就没有响应, 网络请求没有真正拿到结果

响应拦截获取的响应结果一直是**undefined**

所以拦截器第一个箭头函数一定记得返回config

下面图就是最基本的一个做法

```

instance.interceptors.request.use(config => {
  console.log('来到了request拦截success中');
  return config
}, err => {
  console.log('来到了request拦截failure中');
  return err
})

instance.interceptors.response.use(response => {
  console.log('来到了response拦截success中');
  return response.data
}, err => {
  console.log('来到了response拦截failure中');
  return err
})

```

下面是拦截器的基本具体应用场景：

config中的一些信息不符合服务器要求，要做一些修改

每次发送网络请求时，希望在界面显示一个请求图标，等响应成功时再将其隐藏起来

某些网络请求必须携带一些特殊的信息，比如登录（携带token令牌），如果没有携带token就跳转到登录页面

响应拦截

第一个箭头函数表示**响应成功的返回函数**，**默认参数**是相应的结果，所以参数一般写**response**或**result**，下面时打印的结果。

```

[HMR] Waiting for update signal from WDS... log.js?4244:23
request.js?8468:26
{
  data: {...}, status: 200, statusText: "OK", headers: {...}, config: {...}
  , ...} i
    ▶ config: {adapter: f, transformRequest: {...}, transformResponse: {...}}
      data:
        ▶ data: {banner: {...}, dKeyword: {...}, keywords: {...}, recommend: {...}}
          returnCode: "SUCCESS"
          success: true
        ▶ __proto__: Object
      ▶ headers: {content-type: "application/json"}
      ▶ request: XMLHttpRequest {onreadystatechange: f, readyState: 4, ti...
        status: 200
        statusText: "OK"
      ▶ __proto__: Object

```

但axios框架还会将一些其他的信息放进返回结果，所以使用时可以**response/result.data**来获取真正的数据

或者再请求拦截的时候return时，直接returnconfig.data，这样也可以

第二个函数表示**请求失败的返回函数**，默认参数是**错误信息**

