# Vue.js

一、什么是Vue.js

Vue.js 是目前最火的一个前端框架，React是最流行的一个前端框架（React除了开发网站，还可以开发手机App， Vue语法也是可以用于进行手机App开发的，需要借助于Weex）

Vue.js 是前端的主流框架之一，和Angular.js、React.js 一起，并成为前端三大主流框架！

Vue.js 是一套构建用户界面的框架，只关注**视图层**，它不仅易于上手，还便于与第三方库或既有项目整合。（Vue有配套的第三方类库，可以整合起来做大型项目的开发）。

1、前端的主要工作：

主要负责MVC中的V这一层；主要工作就是和界面打交道，来制作前端页面效果；

二、提高开发效率的发展历程

1、原生JS

有兼容性问题，不同的浏览器有自己的实现。

2、jQuery之类的类库

把兼容性都给处理好了，假如要设置表格，就要拼接字符串，然后去操作dom，这样不方便，大家的拼接方式可能不一样。

1. 前端模板引擎

它提供一个模板，我们去填数据就行，解决了频繁操作dom问题。  
 [

{id:1, name:'hello'}

{id:3, name:'hello'}

{id:2, name:'hello'}

{id:4, name:'hello'}

]

点击id这一列，它会重新进行排序，然后模板引擎重新渲染，但是有个问题，1，4的顺序是不用改变的，但是它被重新渲染了，如果有上千百条的数据，性能问题就不得不考虑。

1. Vue.js

能够帮助我们减少不必要的DOM操作；提高渲染效率；双向数据绑定的概念：通过框架提供的指令，我们前端程序员只需要关心数据的业务逻辑，不再关心DOM是如何渲染的了。

在Vue中，一个核心的概念，就是让用户不再操作DOM元素，解放了用户的双手，让程序员可以更多的时间去关注业务逻辑。

三、框架和库的区别

1、框架：是一套完整的解决方案；对项目的侵入性较大，项目如果需要更换框架，则需要重新架构整个项目。

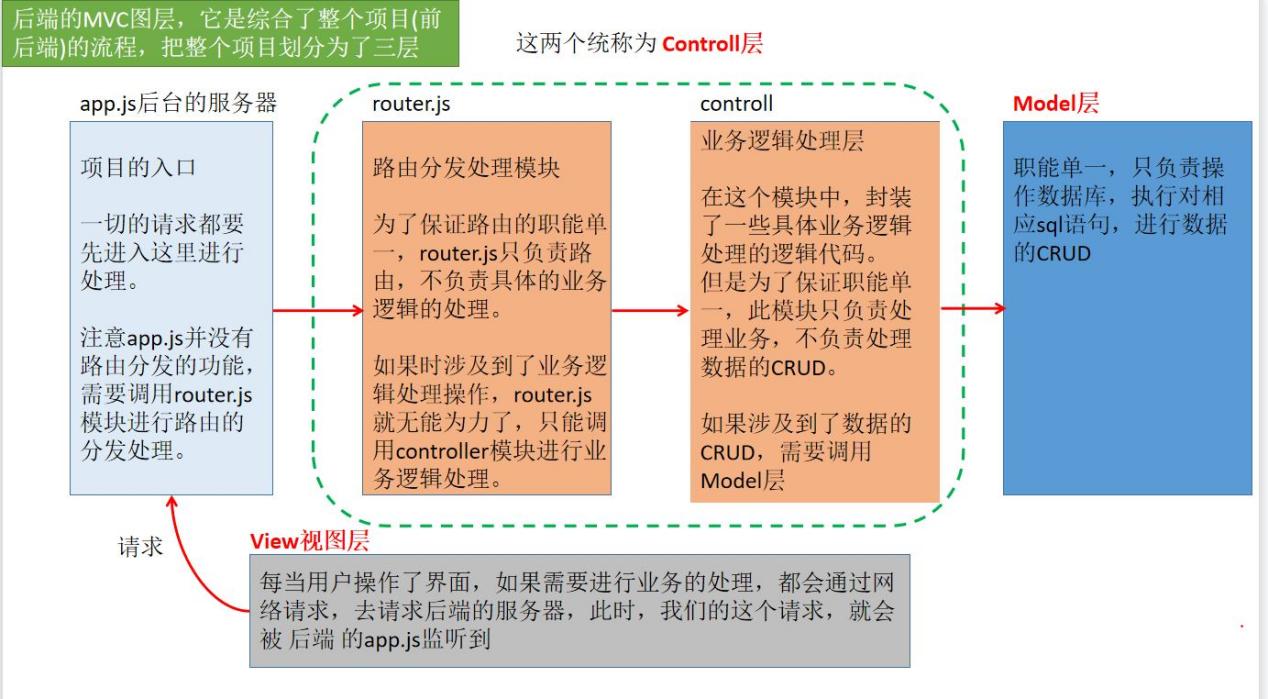
node 中的 express。

2、库（插件）：提供某一个小功能，对项目的侵入性较小，如果某个库无法完成某些需求，可以很容易切换到其它库实现需求。

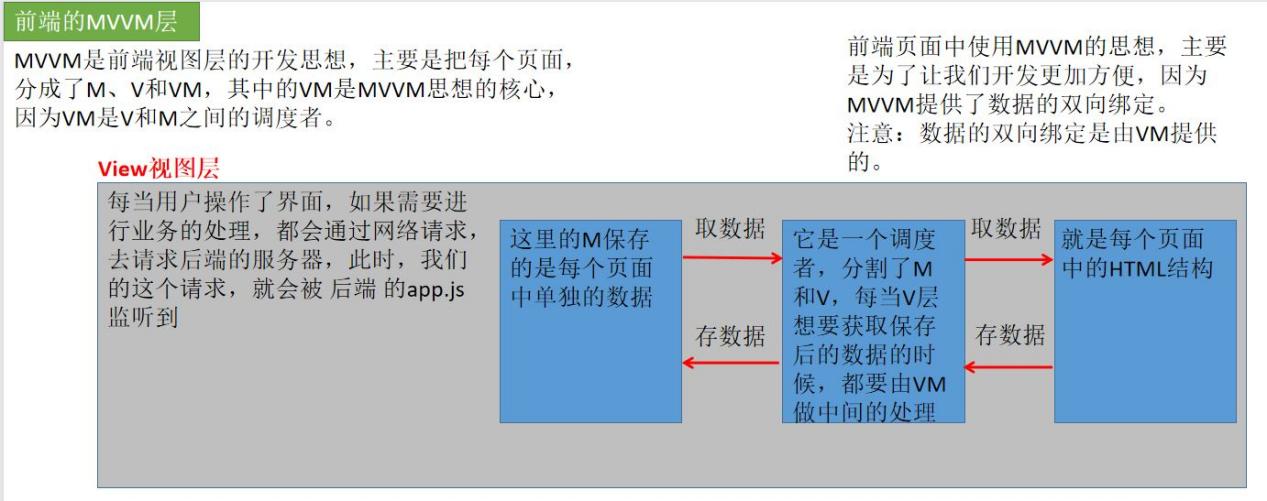
如从jQuery 切换到 Zepto，它们的API基本是大同小异的。

四、Node（后端）中的 MVC 与前端中的 MVVM 之间的区别

1. MVC 是后端（站在整个项目层面上）的分层开发概念。



2、MVVM是前端视图层的概念，主要关注于 视图层分离，也就是说：MVVM把前端的视图层，分为了 三部分 Model、View、VM（ViewModel）



CRUD是指在做计算处理时的增加(Create)、查询(Retrieve)（重新得到数据）、更新(Update)和删除(Delete)几个单词的首字母简写。主要被用在描述软件系统中数据库或者持久层的基本操作功能。

1. Vue.js 基本代码 和 MVVM 之间的对应关系
2. 第一个Vue实例

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue案例</title>  </head>  <body>  <!-- Vue 实例所控制的这个元素区域，就是我们的 V -->  <div class="box">  <p>{{ msg }}</p>  </div>  <!-- 当我们导入包之后，在浏览器的内存中，就多了一个 Vue 构造函数 -->  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  // 创建一个Vue的实例，我们创建出来的这个 vm 对象，就是我们 MVVM 中的 VM 调度者。  var vm = new Vue({  el: '.box', // 表示Vue的实例，要控制页面上的哪个区域，选择器。  // data 就是 MVVM 中的 M，专门用来保存每个页面的数据，它存放的是 el 中要用到的数据。  data: {  msg: '你好啊，我是vue',  // 通过 Vue 提供的指令，很方便就能把数据渲染到页面上，程序员不再手动操作DOM元素了，前端的 Vue 之类的框架，不提倡我们手动操作DOM元素了。  }  });  </script>  </body>  </html>  1 |

2、插值

（1）数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：双大括号会将数据解释为普通文本，而非 HTML 代码。

|  |
| --- |
| <span>Message: **{{ msg }}**</span> |

Mustache 标签将会被替代为对应数据对象上 msg 属性的值。无论何时，绑定的数据对象上 msg 属性发生了改变，插值处的内容都会更新。

（2）v-once

通过使用 v-once 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

|  |
| --- |
| <span v-once>这个将不会改变: {{ msg }}</span> |

（3）使用 JavaScript 表达式

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

|  |
| --- |
| <!-- 这是语句，不是表达式 -->  {{ var a = 1 }}  <!-- 流控制也不会生效，请使用三元表达式 -->  {{ if (ok) { return message } }} |

3、指令

指令 (**Directives**) 是带有 v- 前缀的特殊 attribute。指令 attribute 的值预期是**单个 JavaScript 表达式** (v-for 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。

注意：在一些自定的指令中，v-color = “red”，跟 v-color = “**’**red**’**”，这两者是不一样的，前者会去找data中的red属性，而后者是一个字符串red来着。

（1）v-cloak：能解决 插值表达式 闪烁的问题

在慢速请求的情况下，{{ msg }}还没有渲染出来，它还在加载vue库。它在加载完毕之后，就会把这个指令移除了。刷新快点就能看到

（2）v-text

v-text是不会有闪烁的问题的。如果它标签里边原来有内容，那么原来的内容会闪烁的，但是一般不会有这个情况。加内容似乎没有意义。

v-text解析完毕后，会覆盖元素中原本的内容，**但是 插值表达式 只会替换**自己的这个占位符，不会把整个元素的内容清空的。

（3）v-html

{{ }}和v-text是不会解析标签的，只会把它们解析成普通的文本，它们在没有显示出来是不会占位置。

v-html也不会有闪烁问题。它会覆盖原有的内容

|  |
| --- |
| <style type="text/css">  [v-cloak] {  display: none;  }  </style>  <body>  <div class="box">  <p v-cloak>==========={{ msg }}=========</p>  <p v-text='msg'>啊啊啊啊啊啊啊啊啊</p>  <div v-html='msgWithHTML'>66666666666</div>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  msg: '你好啊，我是vue',  msgWithHTML: '<h1>我是大标题啊</h1>',  }  });  console.log(vm);  </script>  </body> |

（4）v-bind

是Vue中提供用于绑定属性的指令，把属性和变量绑定起来，它把属性值当成js代码来执行，可以写合法的js表达式。

v-bind指令可以被简写为 :要绑定的属性。

|  |
| --- |
| <input type="button" value="按钮" **v-bind**:title="**myTitle**">  var vm = new Vue({  el: '.box',  data: {  **myTitle**: '我是v-bind绑定的属性title',  }  });  如果不是合法的js表达式代码则会报错。 |

（5）v-on

v-on:click=”xxx”，xxx表示函数的名字。后边是能带小括号跟参数的

简写@click=

methods属性中定义了当前的Vue实例所有可用的方法。

|  |
| --- |
| <input type="button" value="按钮" v-bind:title="myTitle" v-on:click='show'>  var vm = new Vue({  el: '.box',  methods: {  show: function () {  console.log('hello i am v-on:click');  }  }  }); |

（6）this的使用，跑马灯小案例

在vm实例中，如果想要获取 data 上的数据或者想要调用 methods 中的方法，就必须通过 this 数据属性名或 this 方法名，来进行访问，这里的this 就表示我们new出来的vm实例对象，它在构造函数里边就用this

Vue它会监听自身data数据的改变，如果数据改变了，那么它会自动同步跟着刷新，从data上同步到页面中去，我们只需关心数据，不需要考虑如何重新渲染DOM页面。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue案例</title>  <style type="text/css">  [v-cloak] {  display: none;  }  </style>  </head>  <body>  <div class="box">  <input type="button" value="跑起来" @click="run"/>  <input type="button" value="停止" @click="stop"/>  <h4 v-text="msg"></h4>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  msg: '跑起来，跑马灯，gogo',  timeId: null,  },  methods: {  run: function () {  // 为什么访问msg没有通过data，访问data却为undefined  // console.log(this.msg);  if (this.timeId) {  return;  // 这种处理方式只能在时间比较快执行下才起作用，400毫秒太慢了，一直点击的话，会出现停止现象  // clearInterval(this.timeId);  // this.timeId = null;  }  this.timeId = setInterval(() => {  var first = this.msg.substring(0, 1);  var end = this.msg.substring(1);  this.msg = end + first;  }, 400);  },  stop: function () {  clearInterval(this.timeId);  this.timeId = null;  }  }  });  </script>  </body>  </html>  这里定时函数，它那箭头函数是属于run的，把它当实参传递进去而已，并不是属于setInterval的。还可以通过bind，不用马上执行。  this.timeId = setInterval(function() {  var first = this.msg.substring(0, 1);  var end = this.msg.substring(1);  this.msg = end + first;  }**.bind(this)**, 400); |

4、事件修饰符

1. .stop：阻止冒泡，从里到外，想从谁开始阻止就给谁加

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>study Vue</title>  <style type="text/css">  .box1 {  padding: 50px;  background-color: pink;  }  .box2 {  width: 200px;  height: 200px;  display: table-cell;  vertical-align: middle;  text-align: center;  background-color: lightskyblue;  }  </style>  </head>  <body>  <div class="box1" @click="box1Click">box1  <div class="box2" @click="box2Click">box2  <input type="button" value="按钮" **v-on:click.stop**="btn">  </div>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box1',  data: {  },  methods: {  box1Click: function () {  console.log('box1 事件发生');  },  box2Click: function () {  console.log('box2 事件发生');  },  btn: function () {  console.log('btn 事件发生');  },  }  });  </script>  </body>  </html> |

（2）.prevent：阻止默认事件

|  |
| --- |
| 如果把a链接写到box1区域之后，那么vue就控制不到了  <div class="box1" @click="box1Click">box1  <div class="box2" @click="box2Click">box2  <input type="button" value="按钮" v-on:click.stop="btn">  </div>  <a href="http://www/baidu.com" **@click.prevent="linkClick"**>前去百度</a>  </div> |

（3）.capture：添加事件侦听器时使用事件捕获模式，从外到里（默认的是冒泡模式）

|  |
| --- |
| <div class="box1" **@click.capture="box1Click"**>box1  <div class="box2" @click="box2Click">box2  <input type="button" value="按钮" v-on:click="btn">  </div>  <a href="http://www/baidu.com" @click.prevent="linkClick">前去百度</a>  </div>  box1 事件发生  btn 事件发生  box2 事件发生 |

（4）.self：只当事件在该元素本身（比如不是子元素）触发时触发回调（如冒泡事件它是被动触发的，只有点击自己才执行）

|  |
| --- |
| <div class="box1" @click="box1Click">box1  <div class="box2" **@click.self="box2Click"**>box2  <input type="button" value="按钮" v-on:click="btn">  </div>  <a href="http://www/baidu.com" @click.prevent="linkClick">前去百度</a>  </div>  btn 事件发生  box1 事件发生 |

（5）.once：事件只触发一次

如果是被冒泡事件触发的，它也是照算执行了一次，后续就不会再触发了

|  |
| --- |
| <div class="box1" @click="box1Click">box1  <div class="box2" @click="box2Click">box2  <input type="button" value="按钮" **v-on:click.once="btn**">  </div>  <a href="http://www/baidu.com" @click.prevent="linkClick">前去百度</a>  </div> |

（6）.self和.stop的区别

.self只会阻止自己身上的冒泡行为的触发，并不会阻止冒泡行为的传递。

5、指令v-model实现双向数据绑定（唯一）

表单元素（input之类的）就能跟用户进行交互。

（1）v-bind 只能实现数据的单向绑定，从 M（数据） 自动绑定到 V（视图） ，无法实现数据的双向绑定。

|  |
| --- |
| 1  2 |

（2）v-model 指令，可以实现表单元素和 Model 中数据的双向绑定

注意：v-model 只能运用在表单元素中去。

input（radio、text、address、email....） select、checkbox、textarea

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>study Vue</title>  </head>  <body>  <div class="box1">  {{ msg }}  <input type="text" v-model="msg">  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box1',  data: {  msg: '你好啊，我是Tom,找Jerry',  },  });  </script>  </body>  </html>  1 |

（3）计算器小案例

select的value属性它用于在选择的下拉列表中指定<option>元素value的值。

|  |
| --- |
| <body>  <div class="box1">  <input type="number" **v-model**="n1">  <select v-model="opt">  <option value='+'>+</option>  <option value='-'>-</option>  <option value='\*'>\*</option>  <option value='/'>/</option>  </select>  <input type="number" **v-model**="n2">  <input type="button" value="=" v-on:click="calc">  <input type="number" **v-model**="result">  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box1',  data: {  n1: 0,  n2: 0,  result: 0,  opt: '+',  },  methods: {  calc: function () {  /\*switch (this.opt) {  case '+':  this.result = parseInt(this.n1) + parseInt(this.n2);  break;  case '-':  this.result = parseInt(this.n1) - parseInt(this.n2);  break;  case '\*':  this.result = parseInt(this.n1) \* parseInt(this.n2);  break;  case '/':  this.result = parseInt(this.n1) / parseInt(this.n2);  break;  }\*/  // 通过拼接字符串来处理结果，eval,在实际中不能常用，耗费性能  var str = 'parseInt(this.n1) '+ this.opt +' parseInt(this.n2)';  this.result = eval(str);  }  }  });  </script>  </body>  1 |

vue双向绑定本质

是监听input事件，通过js去修改input值之后不会触发input事件所以双向绑定并没有生效。你需要每次修改input.value之后，手动触发input事件（**不知道当时是在什么条件下，记录了这一句话，怎么说都不对。这句话是错误的，它的本质是value+input的语法糖），通过js修改是可以的**

|  |
| --- |
| 双向绑定是指绑定一个变量之后，既可以读其值，也可以写其值。不同的框架，对于双向绑定这个事情的实现是不同的。对于 Vue 来说，并不是真正的双向绑定，而是通过只读绑定+事件反馈来实现的。v-model 不过是个语法糖，其他属性值双向绑定可以用语法糖 :value.sync 声明并通过事件反馈（Vue2）。在 Vue3 中直接使用 v-model 带参数的方式让语法变理意义更清楚。  从[官方文档](https://link.segmentfault.com/?url=https://v3.cn.vuejs.org/guide/migration/v-model.html%23_3-x-%E8%AF%AD%E6%B3%95" \t "https://segmentfault.com/q/_blank)中可以看到，Vue3 的 v-model 带参数其实也是个语法糖，如果拆解开，还是一个单向绑定+事件，比如官例： |

|  |
| --- |
| 有时也需要在内联语句处理器中访问原始的 DOM 事件。可以用特殊变量 $event 把它传入方法：  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>study Vue</title>  </head>  <body>  <div class="box1">  {{ msg }}  <input type="text" :value="msg" @input="msg=$event.target.value">  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box1',  data: {  msg: '你好啊，我是Tom,找Jerry',  },  });  </script>  </body>  </html> |

6、在Vue中使用class样式

（1）第一种，直接传递一个数组，**类名需要加引号**，这里的 class 需要使用 v-bind 做数据绑定，这种**不方便使用**，不如原来的方式。

|  |
| --- |
| <h3 v-bind:class="[**'**thin**'**, **'**italic**'**]">{{ msg }}</h3> |

（2）第二种，在数组中使用三元表达式

|  |
| --- |
| <h3 v-bind:class="['thin', 'italic', **flag?'active':''**]" v-cloak>{{ msg }}</h3> |

（3）在数组中使用对象来代替三元表达式，提高代码的可读性

|  |
| --- |
| <h3 v-bind:class="['thin', 'italic', **{'active': flag}**]" v-cloak>{{ msg }}</h3> |

（4）在为 class 使用v-bind 绑定对象的时候，对象的属性是类名，由于 对象的属性可带引号，也可不带引号，属性值 是一个标识符

|  |
| --- |
| <h3 v-bind:class="**classObj**" v-cloak>{{ msg }}</h3>  data: {  msg: 'vue的类样式操作。',  flag: true,  classObj: {  thin: true,  italic: true,  active: true,  }  }, |

7、在Vue使用内联样式

（1）直接在元素上通过 :style 的形式，书写样式对象。

|  |
| --- |
| <h3 :style="{color: 'red', **'font-weight'**: '200',}" v-cloak>{{ msg }}</h3> |

（2）将样式对象，定义到 data 中，并直接引用到 :style 中，引用对象就不用加引号了，相当于是使用变量。

如果对象属性有横线，就要加上单引号，如font-size。

|  |
| --- |
| <h3 :style="styleObj" v-cloak>{{ msg }}</h3>  var vm = new Vue({  el: '.box',  data: {  msg: '你好啊，我是h3',  styleObj: {  color: 'red',  'font-size': '18px',  },  },  }); |

（3）在 :style 中通过数组，引用多个 data 上的样式对象

|  |
| --- |
| <h3 :style="[styleObj, styleObj1]" v-cloak>{{ msg }}</h3>  data: {  msg: '你好啊，我是h3',  styleObj: {  color: 'red',  'font-size': '18px',  },  styleObj1: {  'font-style': 'italic',  }  }, |

8、v-for

迭代跟循环是一样的，说法不同。

（1）迭代数组，如果是数组形式的，那么能第一个参数是它的值，第二个参数是它的索引。

|  |
| --- |
| <div class="box">  <ul>  <li v-for="**item** in **arr**" v-text="**item**"></li>  </ul>  </div>  var vm = new Vue({  el: '.box',  data: {  arr: [1,2,3,4,5,6,7],  },  methods: {  }  });  1 |

（2）迭代对象数组，如果是数组形式的，那么能第一个参数是它的值，第二个参数是它的索引。

|  |
| --- |
| <li v-for="(val, index) in objArr">索引：{{ index }}---------属性值：{{ val.name }}----{{ val.color }}</li>  data: {  objArr: [{ name: 'Tom', color: 'blue', },  { name: 'Jerry', color: 'yellow', },  { name: 'Jon', color: 'sky', }],  },  1 |

（3）迭代对象属性，对象的话v-for有三个属性。

注意，在遍历对象身上的键值对的时候，除了有 val(第一个是属性值)、key(第二个是属性)，在第三个位置还有一个索引（这个索引基本上不会用到）。

|  |
| --- |
| <li v-for="(val, key, index) in objArr">索引index：{{ index }}---------属性key：{{ key }}----属性值val：{{ val }}</li>  1 |

（4）迭代数字

in 后面我们放过 普通数组、对象数组、对象、还可以是数字。

注意：如果使用 v-for 迭代数字的话，前面的 count 值从 1 开始

|  |
| --- |
| <li v-for="count in 10">count: {{ count }}</li>  1 |

1. key

在组件中使用v-for时，要注意key是必须使用的，每次for循环的时候，通过指定key来标识当前循环的这一项的唯一身份。

v-for 循环的时候，key 属性只能使用number，或者 string ，在使用的时候，必须使用 v-bind 属性绑定的形式（因为它的值是变化，不加v-bind就变成了字符串，固定了），指定 key 的值。

在组件中使用 v-for 循环的时候，或者在一些特殊的情况的中，如果 v-for 有问题，就必须指定唯一的**字符串、数字类型**的 key 值。

|  |
| --- |
| 初步简单理解，不加key，你选第五个checkbox，在渲染的时候checkbox只记得是第五个被选，只要是第五就行。  加了key，就记得是item.id为5的那个。保证我们的data数据和页面上的数据，让它强制关联起来，加了key(一定要具有唯一性) id的checkbox跟内容进行了一个关联。是我们想达到的效果  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue study</title>  </head>  <body>  <div class="box">  id:<input type="text" v-model="id">  name:<input type="text" v-model="name">  <input type="button" value="添加" @click="add">  <p v-for="item in list" **v-bind:key="item.id"**>  <input type="checkbox">  id:{{item.id}}-----name: {{item.name}}  </p>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  list: [  {id: 1, name: 'Tom'},  {id: 2, name: 'Jerry'},  {id: 3, name: 'Jon'},  {id: 4, name: 'Mike'},  ],  id: 0,  name: '',  },  methods: {  add: function () {  // this.list.push({id: this.id, name: this.name});  this.list.unshift({id: this.id, name: this.name});  }  },  });  </script>  </body>  </html>  不加key的情况  1 2、 |

1. v-for 也可以去迭代一个函数中的数据，这样能动态切换数据，见品牌案例实例运用

v-for中的数据都是直接从 data 上的list中直接诶渲染过来的，这样是固定的了。

现在，我们自定义一个 search 的方法，同时把所有的关键字，通过传参的形式，传递给了 search 方法。

在 search 方法内部，通过 执行 for循环，把所有符合搜索关键字的数据，保存到一个新数组中，再返回。

9、v-if和v-show

（1）v-if，每次都会重新删除，或创建元素

有较高的切换性能消耗。

如果元素涉及到频繁的切换，最好是不要使用 v-if，推荐使用v-show。

（2）v-show，每次不会重新进行DOM操作，只是切换了元素的 display:none样式

有较高的初始渲染消耗（就是flag为false，它还是会display渲染了，if什么都不会去搞）。

如果元素可能永远也不会被显示出来被用户看到，则推荐使用 v-if，而且不会被频繁切换。

|  |
| --- |
| <div class="box">  <input type="button" @click="toggle" value="切换">  <p v-if="flag">我是v-if控制的</p>  <h3 v-show="flag">我是v-show控制的</h3>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  flag: true,  },  methods: {  toggle: function () {  this.flag = !this.flag;  }  },  });  </script>  1  2 |

10、小总结：

Vue控制的区域不能是body（1.0版本是可以的）

1. 品牌案例

分析：

获取到 id 和 name ，直接从 data 上面获取

组织出一个对象

把这个对象，调用数组的相关方法，添加到当前的data上的list中

在Vue中，已经实现了数据的双向绑定，每当我们修改了 data 中的数据，Vue会默认监听到数据的改动，自动把最新的数据，应用到页面上。

我们更多的是在进行 VM 中的Model 数据的操作，同时，在操作Model数据时候，指定业务逻辑操作

看不到vue data上的数据，可以通过装vue-devtools，在F12中可以看到，数据的变化。

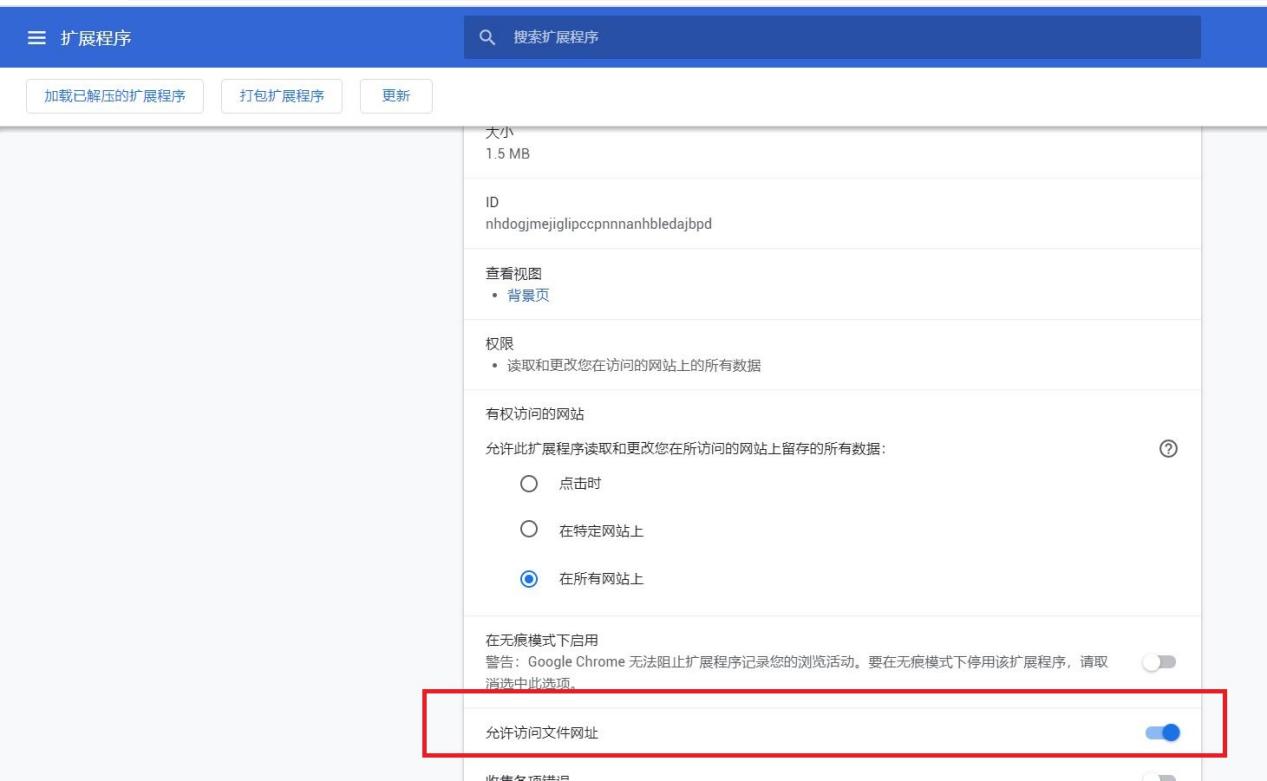
要允许访问文件网址，才能在控制台查看

在写函数的时候，要考虑参数没有传递的情况下，会不会报错的问题，一般给默认值，可以采用ES6的方式加

七、Vue调试工具vue-devtools的安装步骤和使用，它是一个chrome的插件

1、Vue.js devtools ，翻墙安装方式，安装后重启就行。

https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=zh-CN



八、过滤器

1. 概念：

Vue.js 允许你自定义过滤器，可被用作一些常见的**文本格式化**。

过滤器可以用在**两个地方（其他不能用）**：mustache 插值（插值表达式：{{ xxxx }}）和 v-bind 表达式。过滤器应该被添加在 JavaScript 表达式的尾部，通过“管道符”（ | ）形式。

2、过滤器的格式

**{{ name | nameOpt }}**，在输出name值之前先调用 nameOpt 进行一下处理，并把处理的结果当做内容进行渲染。nameOpt是过滤器的名称。原来的数据并不会被修改

3、过滤器的定义语法

Vue.filter(‘过滤器的名称’, function (data) {....});

过滤器中的function，第一个参数已经被规定死了，永远都是过滤器-->管道符前面-->传递过来的数据。

疑问：如果要过滤的对象是一个对象呢。

4、使用多个过滤器，先执行第一个过滤器，然后把第一个过滤器的结果，给第二个过滤器，这样子来处理。

|  |
| --- |
| <body>  <div class="box">  {{ msg **|** translate('u') **|** test }}  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  Vue.filter('translate', function (data, str) {  var result = data.replace(/\w/g, str);  return result;  });  Vue.filter('test', function (data) {  return data + 'test double filter go go go go go';  });  var vm = new Vue({  el: '.box',  data: {  msg: 'hello 我是Tom，我喜欢Jerry',  },  methods: {    },  });  </script>  </body> |

5、全局过滤器，就是所有的 VM 的实例都能进行共享的。

|  |
| --- |
| Vue.**filter**('dateFormate', function (data, format = 'yyyy-mm-dd hh:mm:ss') {  ...................  });  filter是不带s的，它实现了一个过滤器功能。 |

6、自定义一个私有的过滤器（局部）

过滤器调用的时候，如果私有过滤器和全局过滤器名称一致，这时候，优先调用私有过滤器。

它是在Vue构造函数的属性 fliters，它可以定义多个过滤器

|  |
| --- |
| var vm = new Vue({  el: '.box',  data: {...},  methods: {...},  // 私有过滤器，过滤器的名称和处理函数  filters: {  dateFormate: function (data, format = 'yyyy-mm-dd hh:mm:ss') {  ...............  },  dateFormate1: function (data, format = 'yyyy-mm-dd hh:mm:ss') {  ...............  },  }  }); |

7、ES6中字符串新方法，头部填充String.prototype.padStart(maxlength, fillstring = ‘ ’)或者尾部填充String.prototype.padEnd(maxlength, fillstring = ‘ ’)，来填充字符串。

|  |
| --- |
| var h = date.getHours().toString().padStart(2, '0');  var m = date.getMinutes().toString().padStart(2, '0');  var s = date.getSeconds().toString().padStart(2, '0'); |

1. 键盘修饰符以及自定义键盘修饰符

在一些网站输入完信息之后，直接回车就可以提交信息了，这个回车，是输入框的一个键盘事件。

如果只是绑定键盘事件，那么在输入信息的时候，也会触发，这个时候，就需要键盘修饰符，来指定你敲了回车键才触发。

1、键盘修饰符

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

|  |
| --- |
| .enter  .tab  .delete (捕获“删除”和“退格”键)  .esc  .space  .up  .down  .left  .right |

2、自定义全局键盘修饰符

（1）通过**Vue.config.keyCodes.**名称 = 按键值。来自定义按键修饰符的别名：

|  |
| --- |
| **Vue.config.keyCodes.f2** = 113; |

（2）使用自定义的按键修饰符：

|  |
| --- |
| <input type="text" class="form-control" v-model="name" **@keyup.f2="add"**> |

3、系统修饰键（来自vue文档，还有其他的组合按键，可以看文档，这里就不复制了）

|  |
| --- |
| 2.1.0 新增  可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。  **.ctrl**  **.alt**  **.shift**  **.meta**  注意：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”。 |
| 例如：  <!-- Alt + C -->  <input v-on:keyup.alt.67="clear">  <!-- Ctrl + Click -->  <div v-on:click.ctrl="doSomething">Do something</div>  请注意修饰键与常规按键不同，在和 keyup 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 ctrl 的情况下释放其它按键，才能触发 keyup.ctrl。而单单释放 ctrl 也不会触发事件。如果你想要这样的行为，请为 ctrl 换用 keyCode：keyup.17。 |

4、js 里面的键盘事件对应的键码--网址

http://www.cnblogs.com/wuhua1/p/6686237.html

http://www.cnblogs.com/kidney/p/6052935.html

注意：Vue中所有的指令，在调用的时候，都以 v- 开头

十、自定义指令

**涉及一些组件，跟钩子函数的知识？？？**

1、使用 Vue.directive()定义全局指令

参数1：指令的名称，在定义的时候，指令的名称前面，**不需要加 v- 前缀**，在调用的时候，必须在指令名称前加上 v- 前缀来进行调用。

参数2：是一个对象，这个对象身上，有一些指令相关的函数，这些函数可以在特定的阶段，执行相关的操作。

**（1）和样式相关的可以在bind函数中。**

**（2）和行为相关的可以在inserted函数中。只要通过指令绑定给了元素，不管这个元素有没有被插入到页面中去，这个元素肯定有了一个内联的样式**

**将来元素肯定会显示到页面中，这时候，浏览器的渲染引擎必然会解析样式，应用给这个元素。**

|  |
| --- |
| // 自定义全局指令  // 在每个函数中，第一个参数永远是 el ，表示被绑定了指令的那个元素，el 参数是一个原生的js对象，那我们就可以使用DOM对象的那些方法和属性了  Vue.directive('focus', {  // 每当指令绑定到元素上的时候，会立即执行这个bind函数，只执行一次。  // 在元素绑定指令的时候，还没有插入到DOM中，这时候，调用focus，方法没有作用，因为元素只有插入DOM之后，才能获取焦点  bind: function (el) {  // el.focus();  },  // inserted 表示元素插入到DOM中的时候，会执行 inserted 函数，只触发一次  inserted: function (el) {  el.focus();  },  // 当VNode(理解成节点值，Vue 编译生成的虚拟节点)更新的时候，会执行updated，可能会触发多次。  updated: function () {  }  }); |

在bind发生的时候，我们的元素还没有加入到DOM树中，这时候还在内存中，获取焦点是获取不到的，还没有进行渲染

2、自定义私有指令（局部指令）

跟filters的定义类似，指令名称：处理函数

|  |
| --- |
| <label for="exampleInputName2" v-color=**"'green'"**>搜索名称关键字:</label>  它加上单引号，表示是一个字符串。不加话，它会去找data里边的green属性，是找不到的。  var vm = new Vue({  el: '.box',  data: {....},  methods: {....},  // 私有过滤器  filters: {....},  // 私有自定义指令  directives: {  color : {  bind: function (el, binding) {  el.style.color = binding.value;  },  },  }, |

3、钩子函数参数

指令钩子函数会被传入以下参数：

|  |
| --- |
| el：指令所绑定的元素，可以用来直接操作 DOM。  binding：一个对象，包含以下 property：  name：指令名，不包括 v- 前缀。  value：指令的绑定值，例如：v-my-directive="1 + 1" 中，绑定值为 2。  oldValue：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。  expression：字符串形式的指令表达式。例如 v-my-directive="1 + 1" 中，表达式为 "1 + 1"。  arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。  modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为 { foo: true, bar: true }。  vnode：Vue 编译生成的虚拟节点。移步 VNode API 来了解更多详情。  oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。 |

除了 el 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 dataset 来进行。

此外还有动态指令参数，通过vue文档查看。

十一、vue实例的生命周期

1、什么是生命周期

从Vue实例创建、运行、到销毁期间，它会创建各种各样的事件，这些事件，统称为生命周期！说白了就是一个事件。

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做****生命周期钩子****的函数，这给了用户在不同阶段添加自己的代码的机会。

1. 生命周期钩子：就是生命周期**事件**的别名而已。等价于生命周期函数。

2、主要的生命周期函数分类（页面F5刷新的时候，会重新执行一遍）

（1）创建期间的生命周期函数：

beforeCreate：实例刚在内存中被创建出来，此时，还没有初始化好 data 和 methods 属性。

created：实例已经在内存中创建OK，此时 data 和 methods 已经创建OK，此时还没有开始 编译模板。

beforeMount：此时已经完成了模板的编译，但是还没有挂载到页面中。

mounted：此时，已经将编译好的模板，挂载到了页面指定的容器中显示。

（2）运行期间的生命周期函数：

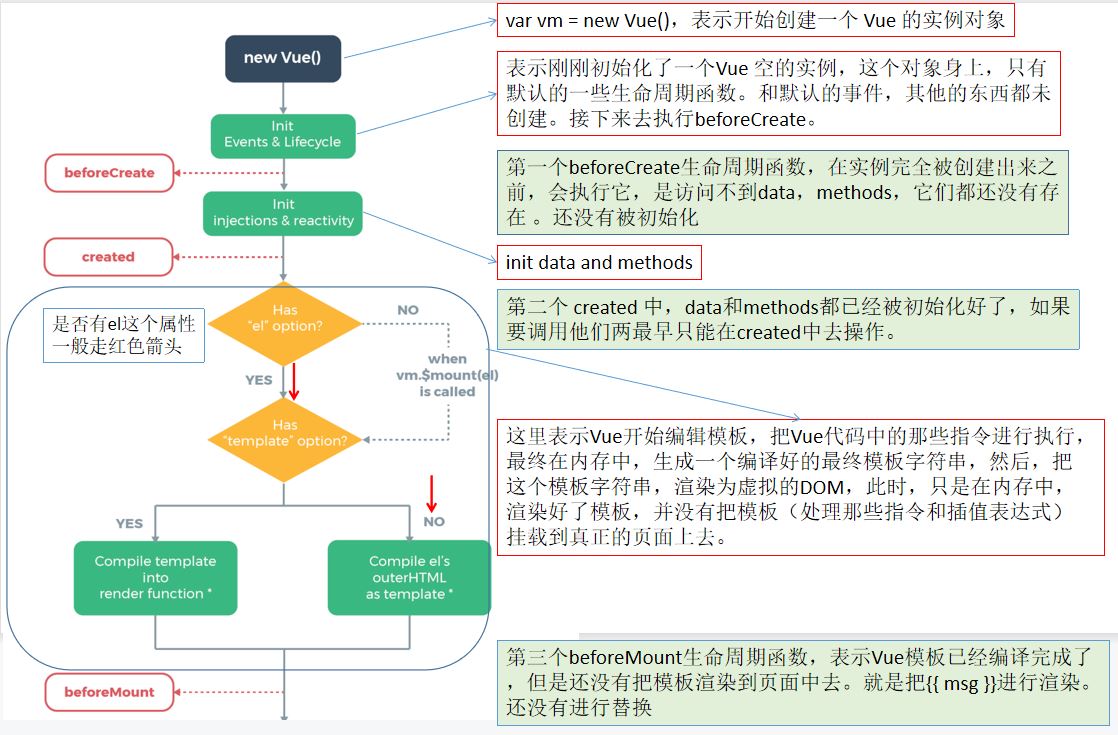
beforeUpdate：状态更新之前执行此函数， 此时 data 中的状态值是最新的，但是界面上显示的 数据还是旧的，因为此时还没有开始重新渲染DOM节点。

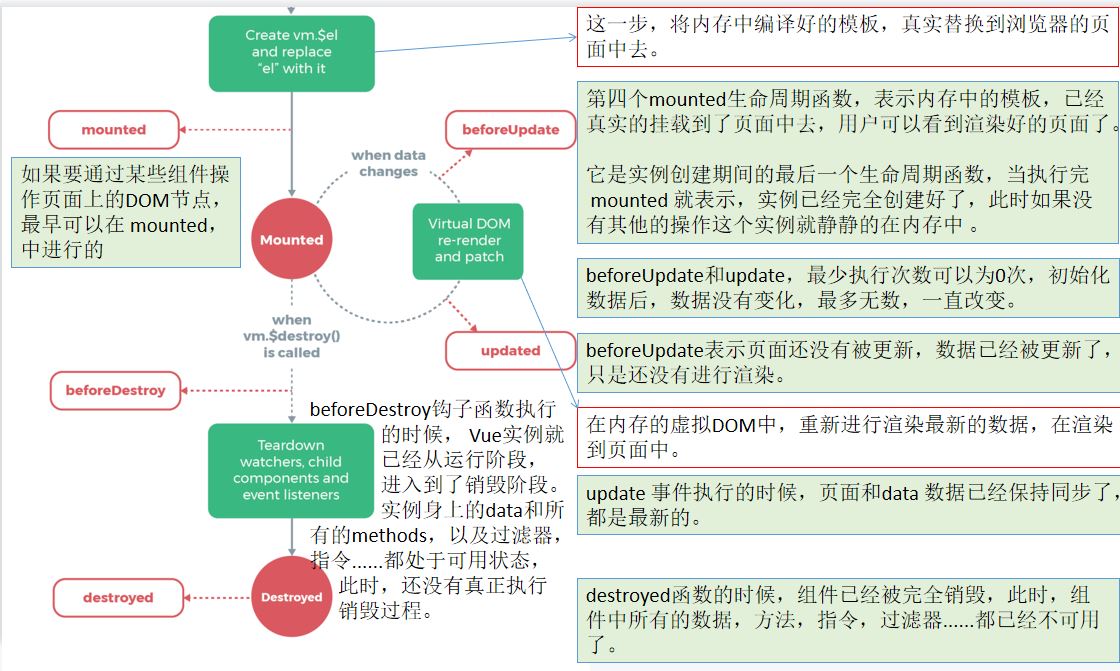
updated：实例更新完毕之后调用此函数，此时 data 中的状态值 和 界面上显示的数据，都已经完成了更新，界面已经被重新渲染好了！

（3）销毁期间的生命周期函数：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。





|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue study</title>  <style type="text/css">  [v-cloak] {  display: none;  }  </style>  </head>  <body>  <div class="box" v-cloak>  <input type="button" value="改变内容" @click='change'>  {{ msg }}  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  msg: 'I am Vue',  },  methods: {  change: function () {  this.msg = 'click change';  }  },  **beforeCreate**: function () {  console.log('beforeCreate', this.msg);  },  **created**: function () {  console.log('created' ,this.msg);  },  **beforeMount**: function () {  var box = document.querySelector('.box');  console.log('beforeMount: box.innerText' ,box.innerText);  },  **mounted**: function () {  var box = document.querySelector('.box');  console.log('mounted: box.innerText' ,box.innerText);  },  **beforeUpdate**: function () {  console.log('beforeUpdate msg = ', this.msg);  var box = document.querySelector('.box');  console.log('beforeUpdate: box.innerText' ,box.innerText);  },  **updated**: function () {  console.log('updated msg = ', this.msg);  var box = document.querySelector('.box');  console.log('updated: box.innerText' ,box.innerText);  }  });  </script>  </body>  </html>  1 |

十二、[vue-resource 实现 get，post，jsonp请求](https://github.com/pagekit/vue-resource)

https://github.com/pagekit/vue-resource

Vue向window挂载Vue，vue-resource向Vue的$http。它是ajax的封装。

在Vue开发中（不提倡去操作DOM），我们不需要jQuery（它主要去操作DOM），它们俩是有一些冲突的，vue-resource是官方开发出来请求。**后来推荐是用 axios 的第三方包来实现数据的请求。好像没人去维护了。**

它用的是ajax请求，可以在F12 network中看请求的类型，xhr。Ajax是原生的自带的，vue-resource只是对它的封装，不单单是jQuery有。一开始以为ajax是jQuery的。

测试的URL请求资源地址：（测试接口用wamp和php也能实现，设置一下php的响应头解决ajax跨域的问题）

get请求地址： [http://www.liulongbin.top:3005/api/getlunbo](http://vue.studyit.io/api/getlunbo)

post请求地址：http://www.liulongbin.top:3005/api/post

jsonp请求地址：http://www.liulongbin.top:3005/api/jsonp

1. get请求使用

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue</title>  </head>  <body>  <div class="box">  <input type="button" value="get请求" @click="getRequest">  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <!-- vue-resource是依赖于vue的，vue要先引入 -->  <script type="text/javascript" src="lib/vue-resource-1.5.3/dist/vue-resource.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  },  methods: {  getRequest: function () {  **this.$http.get**('http://www.myWebServer.com/index.php?username=lxf&password=123')**.then**(function (result) {  console.log(result);  });  },  },  });  </script>  </body>  </html> |
| 允许所有的源访问，方便测试，这样在本地也能访问到  <?php  header("Access-Control-Allow-Origin:\*");  $username = $\_GET['username'];  $password = $\_GET['password'];  // $username = $\_POST['username'];  // $password = $\_POST['password'];  $arr = array('username'=>$username, 'password'=>$password);  echo json\_encode($arr);  ?>  1  2 |

|  |
| --- |
| getRequest: function () { this.$http.get('http://www.liulongbin.top:3005/api/getlunbo').then(function (result) {  console.log(result);  }); |
| 12 |

2、\*支持promise情况下，async和await修改实现数据请求的异步方法，这样可以不要写.then方法

|  |
| --- |
| // 加上async，它从普通的方法修饰为 异步的方法，就可以使用await。再加上解构赋值。  async getRequest() {  const {body, data} = await this.$http.get('http://www.liulongbin.top:3005/api/getlunbo');  console.log(body);  console.log(data);  },  1 |

3、post请求

手动发起的 post 请求，默认没有表单格式，有的服务器可能处理不了，**request body是请求体**的意思。

post都是表单类型的提交，application/x-www-form-urlencoded，提交格式。

|  |
| --- |
| async postRequest() {  const {body} = await this.$http.post('http://www.myWebServer.com/index.php', {username: 'lxf', password: '123'}, **{emulateJSON: true}**);  console.log(body);  }  如果没有提交格式，那么它会出错。  1 |
| async postRequest() {  const {body} = **await this.$http.post**('http://www.liulongbin.top:3005/api/post', **{username: 'lxf', password: '123'}**, {**emulateJSON**: true});  console.log(body);  }  1 |

4、jsonp请求

|  |
| --- |
| async jsonpRequest() {  const { body } = await this.$http.**jsonp**('http://www.liulongbin.top:3005/api/jsonp');  console.log(body);  }  1 |

5、axios请求

支持promise语法，它和vue并没有强制的关系，不用指定引入顺序，但是它不支持发送jsonp的请求，它的数据格式跟vue-resource有点区别

|  |
| --- |
| getRequest: function () {  **axios**.get('http://www.liulongbin.top:3005/api/getlunbo').then(function (result) {  console.log(result);  });  }, |
| 1 |

（1）axios可以挂载到Vue的构造函数的原型上，这样能跟vue-resource无缝衔接

|  |
| --- |
| async postRequest() {  var formData = new FormData();  formData.append('username', 'lxf');  formData.append('password', '123');  // 注意这个不能在全局设置，它会影响get。  const {data} = await this.$http.post('http://www.myWebServer.com/index.php',**formData**,{  headers: {  'Content-Type': 'application/x-www-form-urlencoded',  }  });  console.log(data);  },  1  如果数据放在对象中，它是以对象的形式直接传过去  const {data} = await this.$http.post('http://www.myWebServer.com/index.php', {username: 'lxf', password: '123'},{  headers: {  'Content-Type': 'application/x-www-form-urlencoded',  }  });  1  测试使用的是php做的后台接口  <?php  header("Access-Control-Allow-Origin:\*");  $username = $\_POST['username'];  $password = $\_POST['password'];  $arr = array('username'=>$username, 'password'=>$password);  echo json\_encode($arr);  ?> |

6、jsonp的实现原理

由于浏览器的安全性限制，不允许AJAX访问 协议不同、域名不同、端口号不同的 数据接口，浏览器认为这种访问不安全；

可以通过动态创建script标签的形式，把script标签的src属性，指向数据接口的地址，因为script标签不存在跨域限制，这种数据获取方式，称作JSONP（注意：根据JSONP的实现原理，知晓，JSONP只支持Get请求）；

（1）具体实现过程：

先在客户端定义一个回调方法，预定义对数据的操作；

再把这个回调方法的名称，通过URL传参的形式，提交到服务器的数据接口；

服务器数据接口组织好要发送给客户端的数据，再拿着客户端传递过来的回调方法名称，拼接出一个调用这个方法的字符串，发送给客户端去解析执行；

客户端拿到服务器返回的字符串之后，当作Script脚本去解析执行，这样就能够拿到JSONP的数据

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Jsonp模仿</title>  </head>  <body>  <script type="text/javascript">  function success(data) {  console.log(data, typeof data);  }  </script>  <script type="text/javascript" **src="http://192.168.43.153:3000/?callback=success"**></script>  </body>  </html> |
| Node.js服务器下  const http = require('http');  http.createServer((request, response) => {  const {searchParams} = new URL(request.url, `http://${request.headers.host}`);  const callback = searchParams.get('callback');  console.log(callback);  let people = {  name:'Tom',  age: '43',  };  response.writeHead(200, {  'content-Type': 'text/plain; charset=utf8',  });  **const data = `${callback}(${JSON.stringify(people)})`;**  console.log(data);  response.end(data);  }).listen(3000,'192.168.43.153', (err)=>{  if (err) {  console.log(500);  return;  }  console.log('server running');  }); |

（2）注意点，回传的字符串

`${callback}(${JSON.stringify(people)})`结果是字符串success({"name":"Tom","age":"43"})

这是node服务器返回的结果，people对象转化为字符串之后还跟callback拼接了，它们是整体是一个字符串。 客户端获取，执行时，它实际上就是一个对象。

如果在script中直接写test(JSON.stringify(people));，那么people对象，它是一个字符串传进去，打印也是字符串

如果涉及到了跨域，最好是在服务端的Api接口只中CORS就行了

十三、品牌管理结合axios(具体代码看brand\_axios)

1. input应该把数据前后的空格处理一下。

前边我们都是直接把请求的网址写死了，这样后期修改很麻烦，可以同axios提供

axios.defaults.baseURL

axios.create()方法，调用的返回值，是一个新的 axios 实例，在调用 create 函数的时候，可以初始化一些默认配置项，比如，请求的baseURL地址。

这样的话每个html页面都要写一个axios的实例，都是一样的代码，可以把这段代码提取出来，放到**js文件**中，需要的就引入。

2、接口说明

Vue.js中所有接口更改为：http://www.liulongbin.top:3005/

（1）品牌数据列表

|  |  |
| --- | --- |
| **地址** | http://www.liulongbin.top:3005/api/getprodlist |
| **作用描述** | 获取品牌数据json数据 |
| **请求方式** | Get |
| **传入api的参数** | 无 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（2）删除品牌数据

|  |  |
| --- | --- |
| **地址** | http://www.liulongbin.top:3005/api/delproduct/:id |
| **作用描述** | 删除品牌数据 |
| **请求方式** | Get |
| **传入api的参数** | Id：传入品牌数据id  举例：  vueapi.ittun.com/api/delproduct/8 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（3）添加品牌数据

|  |  |
| --- | --- |
| **地址** | http://www.liulongbin.top:3005/api/addproduct |
| **作用描述** | 添加品牌数据 |
| **请求方式** | Post |
| **传入api的参数** | name : ’品牌名称’ |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

十四、Vue中的动画与过渡

CSS3可以做的比较炫酷，而Vue中的动画、过渡是为了让客户去了解页面。能够提高用户的体验，帮助用户更好的理解页面中的功能。基本是过渡的效果，渐渐显示出来，隐藏等等。

动画与过渡看具体的使用，它们在表现形式上跟动画是类似的。以下一些描述就暂时不做区分。

1、单元素/组件的过渡，**进入和离开就是显示跟隐藏**

Vue 提供了 transition 的封装组件，**在下列情形中**，可以给任何元素和组件添加**进入**/**离开**过渡。过渡结束的时候，是会移除它所有的样式的。

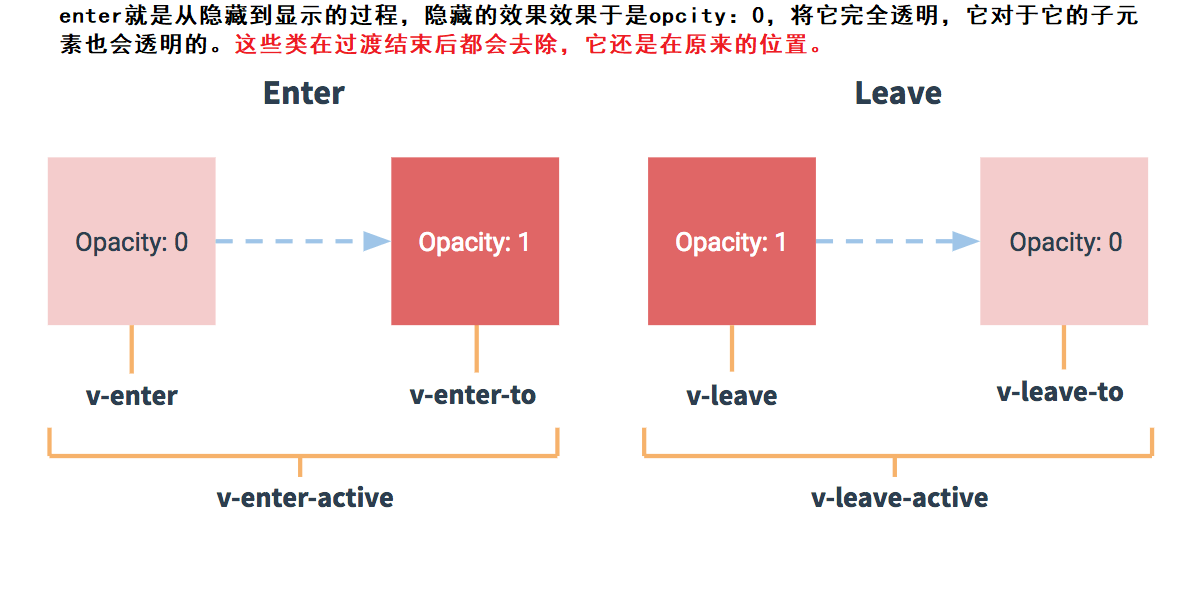
（1）条件渲染 (使用 v-if，可以理解成增加 / 删除，它的本质是添加 / 删除dom，列表过渡时候，可以这么理解)

（2）条件展示 (使用 v-show)

（3）动态组件

（4）组件根节点

（5）可以大致理解成transition过渡是针对一个标签，让他显示、隐藏，里边的数据可以改。



2、Vue把一个完整的过渡，拆分成两个部分（你把它隐藏了，肯定可以让它显示，所有可以分成两部分），入场（从隐藏到显示）过渡和离场（从显示到隐藏）过渡。

（1）入场过渡中，包含了两个状态，是起始状态（v-enter），结束状态（v-enter-to）。

（2）v-enter表示入场过渡的起始状态，如透明度为0，横向偏移量为50px。

（3）v-enter-to表示入场过渡的结束状态，如透明度为1，横向偏移量为 0px。

（4）v-enter-active表示入场过渡的时间段，在这里，可以规定过渡的时长，还有一些过渡相关的效果，如ease等等

v-leave和v-enter是类似的。

3、过渡的类名

在进入/离开的过渡中，会有 6 个 class 切换。

（1）进入

|  |
| --- |
| v-enter：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的**下一帧移除（它的样式也就失效了）**。  v-enter-active：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后**移除**。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。  v-enter-to：2.1.8 版及以上定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除)，在过渡/动画完成之后**移除**。  简单理解：先v-enter，然后v-enter-active，再v-enter-to同移除v-enter，结束所有类移除。 |

（2）离开

|  |
| --- |
| v-leave：定义离开过渡的开始状态。在离开过渡被触发时立刻生效，**下一帧被移除（它的样式就失效了）**。  v-leave-active：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后**移除**。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。  v-leave-to：2.1.8 版及以上定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除)，在过渡/动画完成之后**移除**。 |

（3）进入和离开，之间没有任何关系。因为它们最后都被删除了，执行某个过程，才会去添加对应的类

4、过渡用法以及详细分析（重要）

（1）先用transition标签包裹需要有动画的元素

（2）设置v-enter、v-leave-to样式，跟v-enter-active

过渡知识的补充，例如原本width 200px，给一个按钮 width 300px，那么它由200px变为300px这个过程会有一个过渡

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue动画</title>  <style type="text/css">  .v-enter,  .v-leave-to {  opacity: 0;  transform: translateX(200px);  }  .v-enter-active,  .v-leave-active {  transition: all 3s linear;  }  </style>  </head>  <body>  <div class="box">  <input type="button" value="toggle" v-on:click="flag=!flag"><br />  <transition>  <div v-show="flag">使用Vue动画的h3元素</div>  </transition>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el:'.box',  data: {  flag: **true**,  },  methods: {  },  });  </script>  </body>  </html>  分析如下：  flag为true，h3显示，先分析离开。  1、离开。  首先是先加上v-leave起始状态这个类（在离开时候，一般不添加，因为起始状态就是它原来的位置，假如你设置了left，它突然跳一下，这样体验不好），这个时间非常短。**------->** 然后加上v-leave-to类和v-leave-active。-------> 然后同时把 v-leave移除了。------->接下来opacity：0慢慢透明化，以及transform:translate(100px)慢慢向左移动，-------> 最后把v-leave-to和v-leave-active移除，-------> 然后display：none。  2、进入。  首先是先加上v-enter起始状态这个类，这个时间非常短。**------->** 然后加上v-enter-to类（在进入时候，一般不设置这个类，无论你去到哪里，它最后都会被移除，回到原来的位置）和v-enter-active。-------> 然后同时把v-enter 移除了。------->接下来在初始的条件下opacity：0，transform:translate(100px)，它们已经被移除了，所以会表现为慢慢不透明化，慢慢向右移动，就是恢复原来的样子，-------> 最后把v-leave-to和v-leave-active移除，-------> 然后display：none移除。 |
| 把类移除后，会慢慢恢复原状的测试代码  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style type="text/css">  .box {  width: 200px;  height: 200px;  background-color: red;  transition: all 3s linear;  }  .add {  transform: translateX(100px);  }  </style>  </head>  <body>  <input type="button" id="btn" value="增加transform">  <input type="button" id="btn1" value="去除trans">  <div class="box" id = 'box'></div>  <script type="text/javascript">  var btn = document.getElementById('btn');  var btn1 = document.getElementById('btn1');  var box = document.getElementById('box');  btn.onclick = function () {  box.className = 'box add';  }  btn1.onclick = function () {  box.className = 'box';  }  </script>  </body>  </html> |

5、使用第三方 CSS 动画库，需要加强学习

自定义过渡的类名

我们可以通过以下来自定义过渡类名：

|  |
| --- |
| enter-class  enter-active-class  enter-to-class (2.1.8+)  leave-class  leave-active-class  leave-to-class (2.1.8+) |

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如 Animate.css 结合使用十分有用。

（1）导入动画类库：

（2）定义 transition 及属性：

|  |
| --- |
| <link rel="stylesheet" href="lib/animate.css-4.1.1/animate.css">  <div class="box">  <input type="button" value="toggle" v-on:click="flag=!flag"><br />  <transition enter-active-class="animate\_\_animated animate\_\_bounceInDown" leave-active-class="animate\_\_animated animate\_\_bounceOutDown">  <div v-show="flag" class="animated">使用第三方动画的h3元素</div>  </transition>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script> |

6、使用javascript钩子函数

商品加入购物车的动作，就是只有离开的过渡。只有半场的过渡，过渡类名和动画库都无法实现。

（1）钩子就是transition的生命周期函数，有两组，一组是进入，一组离开。

|  |
| --- |
| <transition  v-on:before-enter="beforeEnter" // 在开始过渡之前的起始状态。  v-on:enter="enter"// 过渡完成之后的结束状态。如果这些样式没有在after-enter中进行清理，那么它会一直存在。  v-on:after-enter="afterEnter" // 过渡完成之后的清理工作，相当于是回调函数，动画完成之后调用这个函数。（这个说法并不是准确的，它源码分了自动，手动区别，看下面分析）  v-on:enter-cancelled="enterCancelled" // 取消入场过渡，这个一般用不到，一般只用前边的三个。  v-on:before-leave="beforeLeave"  v-on:leave="leave"  v-on:after-leave="afterLeave"  v-on:leave-cancelled="leaveCancelled"  >  .........  </transition> |

（2）元素的transitionend事件，<transition>元素</transition>

Vue 为了知道过渡的完成，必须设置相应的事件监听器。它可以是 transitionend 或 animationend。

（3）\*钩子函数的执行时机（个人推测，有初步验证）

点击按钮flag由false变为true，当v-show条件为true的时候，先执行before-enter，初始化过渡的起始状态。再执行enter事件，设置过渡的效果，以及元素移动到哪个位置。

此时如果enter带done参数（2个参数），done（after-enter函数）的时机由用户决定；

如果enter不带done参数（只有el一个参数），那么after-enter函数会在动画结束（transitionend事件发生的时候）被Vue自动调用。

（4）enter钩子函数带done参数，与不带done参数分析，有代码

注意：那个239集视频中讲解是有误的，对done的执行时机是错的。done并不是在动画过渡结束执行的。

看源代码发现，如果形参**不指定 done** ，则表明 用户不会手动控制 动画（过渡）的结束，而转由节点的**transitionEnd控制**或者animationEnd 来标识动画（过渡）结束，开始回调 afterEnter。

具体参考源码:

var expectsCSS = css !== false && !isIE9;

var userWantsControl = getHookArgumentsLength(enterHook);

console.log('enterHook arguments length , ' + userWantsControl);

getHookArgumentsLength 用于判断 enterHook 也就是你配置的

enter(el,done) {

el.offsetWidth

el.style.transform = 'translate(200px,150px)'

el.style.transition = 'all 1s ease'

}

钩子函数的形参的个数。

当形参个数大于1，表示形参中有done, 也就是说明 用户需要手动控制动画何时结束。也就是说一旦你配置了done形参，则转由你告诉框架，动画合适结束。所以你需要在合适的时机调用done，否则 afterEnter接口就没法被调用了。

PS: 函数有几个形参，可以通过 functionName.length 得到。

|  |
| --- |
| 1、带done参数，用户需要手动控制动画何时结束。也就是说一旦你配置了done形参，则转由你告诉框架，动画合适结束。  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue动画钩子</title>  <link rel="stylesheet" href="lib/animate.css-4.1.1/animate.css">  <style type="text/css">  .circle {  width: 20px;  height: 20px;  background-color: red;  border-radius: 50%;  }  </style>  </head>  <body>  <div class="box">  <input type="button" value="toggle" v-on:click="flag=!flag"><br />  <transition  v-on:before-enter="beforeEnter"  v-on:enter="enter"  @after-enter="afterEnter"  >  <div v-show="flag" class="circle"></div>  </transition>  </div>  <script type="text/javascript" src="lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el:'.box',  data: {  flag: false,  },  methods: {  beforeEnter (el) {  // el 就是transition包括的元素,就是一个原生的DOM对象  el.style.transform = 'translate(0,0)';  console.log('1beforeEnter');  },  enter(el, done) {  el.offsetLeft;  el.style.transform = 'translate(300px,300px)';  el.style.transition = 'all 3s ease';  **done();**  **// 在这里可以添加el的监听事件transitionEnd，再去执行done，但是这样不如直接不带done参数来得简单。**  console.log('2enter');  },  afterEnter(el) {  this.flag = !this.flag;  console.log('3afterEnter');  }  },  });  </script>  </body>  </html>  1  整个效果就是。  （1）点击按钮v-show：true触发显示小球。  （2）before-enter初始化小球的样式。  （3）enter加上过渡效果，确认小球最终的样式，同时调用了done，执行了after-enter，flag取反，设置为false，display会隐藏，但它是过渡的效果，到过渡结束的时候，才会马上消失。 |
| 2、不带done参数，由Vue帮我们去搞  enter(el) {  el.offsetLeft;  el.style.transform = 'translate(300px,300px)';  el.style.transition = 'all 3s ease';  console.log('2enter');  },  afterEnter(el) {  el.style.transition = 'none';  this.flag = !this.flag;  console.log('3afterEnter');  }  整体效果：  （1）3不会马上打印，会等到小球过渡结束的时候才打印。  （2）然后等待3秒后才消失，因为过渡效果并没有被处理掉，v-show为false它也需要3秒时间从显示过渡到消失。  （3）如果想马上消失，需要把过渡效果处理掉。  1 |

（5）补充事项

动画钩子函数的第一个参数：el，表示 要执行动画的那个DOM元素，是个原生的 JS DOM对象。

等价于 el 通过 document.getElementById('xxx') 方式获取到的原生JS DOM对象。

enter参数done相当于是afterEnter函数，这个done 就是afterEnter 函数的引用。

这是固定的写法，如果不写el.offsetXXX就无法实现动画效果。这个现象视频没有进行解释

可认为。写这个是为了保证Dom实时刷新，随便写什么都行，只要能保证Dom实时刷新，enter只执行了一遍，比较难解释实时刷新。

（6）贝塞尔曲线cubic-bezier（）可以用来调试动画过渡效果，ease内部也是使用这个

7、transition name属性必要性

如果只使用v-，那么在html文件中有多个需transition的元素，所有的transition的效果的一样的，可以通过指定name，来指定不一样的过渡效果。

8、transition mode过渡模式（在transition包括的多组件<component>中也适用）

一个离开过渡的时候另一个开始进入过渡。这是 <transition> 的默认行为。进入和离开同时发生。

同时生效的进入和离开的过渡不能满足所有要求，所以 Vue 提供了过渡模式：

（1）in-out：新元素先进行过渡，完成之后当前元素过渡离开。

（2）out-in：当前元素先进行过渡，完成之后新元素过渡进入。

|  |
| --- |
| <style type="text/css">          .com-enter {              opacity: 0;              transform: translateX(100px);          }          .com-leave-to {              opacity: 0;              transform: translateX(-100px);          }          .com-enter-active,          .com-leave-active {              transition: all 1s ease;              position: absolute;          }      </style>  <transition name="com" mode="out-in">              <component v-bind:is="comname"></component>          </transition> |

9、v-for的列表过渡，<transition-group>

使用 v-for这种场景中，使用 <transition-group> 组件，它会以一个真实元素呈现：默认为一个 <span>（transition-group渲染成了span标签的意思）。你也可以通过 tag 属性更换为其他元素（tag = “ul”）。span里边可能是li标签，这样就不合理，span行内元素不能包括li块级元素。

过渡模式（待学习）不可用，因为我们不再相互切换特有的元素。

内部元素总是需要提供唯一的 key attribute 值。

**CSS 过渡的类**将会应用在内部的元素中，而不是这个组/容器本身。

（1）v-move

<transition-group> 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 **v-move**class****，它会在元素的**改变定位**的过程中应用。像之前的类名一样，可以通过 name attribute 来自定义前缀，也可以通过 move-class attribute 手动设置。

v-move 对于设置过渡的切换时机和过渡曲线非常有用。

v-move一般与v-leave-active { position:absolute }一起使用。

内部的实现，Vue 使用了一个叫 FLIP 简单的动画队列，使用 transforms 将元素从之前的位置平滑过渡新的位置。

注意：需要注意的是使用 FLIP 过渡的元素不能设置为 display: inline 。作为替代方案，可以设置为 display: inline-block 或者放置于 flex 中。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue 列表过渡</title>  <style type="text/css">  \* {  margin: 0;  padding: 0;  }  ul {  list-style: none;  margin: 100px;  }  .box li {  width: 500px;  height: 30px;  line-height: 30px;  border: 1px dashed #000;  margin: 5px;  }  .box li:hover {  background-color: yellow;  }  .v-enter,  .v-leave-to {  opacity: 0;  transform: translateX(600px);  }  .v-enter-active,  .v-leave-active {  transition: all 2s ease;  }  .v-move {  transition: all 2s ease;  }  .v-leave-active {  position: absolute;  }  </style>  </head>  <body>  <div class="box">  id: <input type="text" v-model="id" />  name: <input type="text" v-model="name" />  <input type="button" value="首行添加" @click="firstAdd">  <input type="button" value="末尾添加" @click="lastAdd">  <transition-group tag="ul">  <li v-for="(item, index) in list" v-bind:key="item.id" @click='del(index)'>{{ item.id }}----{{ item.name }}</li>  </transition-group>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  list: [  {id: 1, name: 'Tom'},  {id: 2, name: 'Mike'},  {id: 3, name: 'Jon'},  {id: 4, name: 'Jerry'},  ],  id: '',  name: '',  },  methods: {  del(index) {  // 删除数据，在Vue渲染时候相当于删除了DOM对象，操作了v-if，产生过渡效果  this.list.splice(index, 1);  },  firstAdd() {  var o = {  id: this.id,  name: this.name,  }  this.list.unshift(o);  },  lastAdd() {  var o = {  id: this.id,  name: this.name,  }  this.list.push(o);  }  },  });  </script>  </body>  </html> |

将要被删除的元素，在执行过渡效果的时候，它原本的位置还是占着的，直到过渡效果结束之后，它才完全被删除。

如果在v-enter-active，加了absolute，那么它一过渡马上就脱离文档流，不占据原来的位置。v-move就能渐渐移动上去。

十五、定义全局Vue组件

1、组件化应用构建

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：



在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。在 Vue 中注册组件很简单：

（1）\*它是一个Vue实例的理解

|  |
| --- |
| // 先得到组件的构造函数  const com = Vue.extend({  template: '<div><h2>我是div中的h2,我是一个组件</h2></div>',  });  // 全局注册组件  Vue.component('my-com', com);  1  在html中使用my-com的时候，就相当于是去调用了com这个构造函数，得到一个Vue的实例，组件的实例。  Vue官网，组件的复用有解释 |

（2）全局注册组件要在new Vue实例之前

2、组件化和模块化的不同：

（1）模块化： 是从代码逻辑的角度进行划分的；方便代码分层开发，保证每个功能模块的职能单一；为了保证大家以相同的方式去封装模块，于是我们就创造了模块化的规范（CommonJS规范）。

模块化的好处，方便项目的开发，和后期的维护与扩展，今后如果需要某些固定的功能模块，则直接拿来引用就行，提高了项目的开发效率。

（2）组件化： 是从UI界面的角度进行划分的；把页面上有重用性的UI结构和样式，单独 的抽离粗来，封装为单独的组件，前端的组件化，方便UI组件的重用；

组件化的好处：随着项目规模的发展，我们手里的组件，会越来越多。这样，我们今后一个页面中的UI，几乎都可以从手中拿现成的组件拼接出来，方便项目的开发和维护。

3、created()和mounted()补充

（1）created()表示data数据和 methods 方法都已经可用了，这时候就看可以去初始化一些数据，例如axios去请求数据，等等渲染页面。

（2）mounted()表示内存中渲染好的DOM树，已经挂载到了真实的页面中。

在网页中，有一些第三方的插件需要初始化，插件封装一般分为两部分，UI结构，JS代码。初始化插件的js代码就能先初始化。

一些第三方的插件，如果要结合Vue使用，而且，这个插件需要手动做一些js的初始化，此时，必须放到mounted，否则，可能无法出现预期的效果。

4、全局组件定义的三种方式

定义之后无论在哪个Vue的实例中都可以去使用。

（1）Vue.extend()返回的结果是组件构造函数。

（2）template属性，表示这个组件UI代码结构。

5、第一种方式Vue.extend 配合 Vue.component （麻烦点）

（1）先调用 Vue.extend()得到组件的构造函数。

1. 通过Vue.component(‘组件的名称’, 组件的构造函数)，来注册全局组件。
2. 把注册好的全局组件名称，以标签的形式引入到页面中即可。

（4）注意，全局组件的命名有以下注意事项。

|  |
| --- |
| 1、组件的命名以小写为主，一般不会出问题  2、如果组件的名称以驼峰式命名，在标签引用的时候，需要在大写字母前加 -小写字母。否则出错。  <mycom></mycom> 正确的写法 <my-com></my-com>  Vue.component('myCom', com);  1  3、官方推荐小写加 - 。  <my-com></my-com>  Vue.component('my-com', com); |

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <my-com></my-com>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  // 先得到组件的构造函数  const com = Vue.extend({  template: '<div><h2>我是div中的h2,我是一个组件</h2></div>',  });  // 全局注册组件  Vue.component('my-com', com);  var vm = new Vue({  el: '.box',  data: {  },  methods: {  },  });  </script>  </body>  </html>  2 |

6、第二种，单Vue.component()

直接使用 Vue.component('组件名称', { 组件模板对象 })

Vue.component的第二个参数，即接收一个组件的构造函数，同时，也接收一个对象。

|  |
| --- |
| // 全局注册组件,采用模板字符串来写template  Vue.component('mycom2', **{**  **template: `**  **<div>**  **<h2>我是第二种组件的创建，用的是模板字符串</h2>**  **</div>`,**  **});** |

<template></template>模板标签，由vue提供，跟<transition>和<transition-group>类似

7、第三种，配合<template>标签使用

定义一个template的标签，定义组件的UI模板结构。这样VScode有智能提示，跟结构比较直观。

|  |
| --- |
| <div class="box">  <mycom3></mycom3>  </div>  **<template id="com3">**  **<div>**  **<h2>hello 我外置的template</h2>**  **</div>**  **</template>**  再进行全局注册    Vue.component('mycom3', {  **template: '#com3',**  }); |

（1）<template>的位置不能随意

如果写在 vm 实例的控制范围，那么它会被解析出来。

|  |
| --- |
| <div class="box">  <mycom3></mycom3>  **<template id="com3">**  **<div>**  **<h2>hello 我外置的template</h2>**  **</div>**  **</template>**  </div>  1 |

如果放在body结束标签后，那么它无法获取到，报错。

8、Vue.component使用的注意点

template的内容不能是纯文本，组件中的DOM结构，有且只能有唯一的根元素（Root Element）来进行包裹！

|  |
| --- |
| Vue.component('mycom2', {  **template: '纯文本',**  });  1 |
| Vue.component('mycom2', {  template: **'<h2></h2><h3></h3>',**  });  1  h2，h3是两根元素了。 |
| const **template** = {  template: '<div><h1>hello</h1></div>',  }  Vue.component('mycom2', **template**);  这种写法也是可以的。 |

9、定义私有组件

|  |
| --- |
| <div class="box"></div>  <div class="box1">  **<private-com></private-com>**  </div>  <template id="**templateCom**">  <div>  <h2>我私有的template</h2>  </div>  </template>    <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">    // 全局注册组件,采用模板字符串来写template  var vm = new Vue({  el: '.box',  data: {  },  methods: {  },  });  // 控制box1的vm  var box1 = new Vue({  el: '.box1',  **components: {**  **// 这里写了个小横线，所有要带单引号**  **'private-com': {**  **template: '#templateCom',**  **},**  **},**  });  </script>  如果在box中引用会报如下的错误  1 |

十六、组件中展示数据和响应事件

1、组件数据和方法

（1）data，return数据对象

在 组件中，可以有自己的私有数据，但是，组件的 data 必须是一个 function，并且函数内部 return 一个数据对象。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <com></com>  </div>  <template id="template">  <div>  <h2 v-on:click="say">{{ msg }}</h2>  </div>  </template>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  Vue.component('com', {  template: '#template',  **// 内部会自己先调用下这个函数，初始化data**  **data: function () {**  **return {**  **msg: 'hello world',**  **}**  **},**  methods: {  say: function () {  console.log('hello');  },  },  });  // 全局注册组件,采用模板字符串来写template  var vm = new Vue({  el: '.box',  data: {  },  methods: {  },  });  </script>  </body>  </html> |

1. 组件的复用

<div id="components-demo">

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

</div>

注意当点击按钮时，每个组件都会各自独立维护它的 count。因为你每用一次组件，就会有一个它的**新实例**被创建。

（1）Vue.extend做了什么事情，看源码的解析，这里只截取一部分，完整的可以搜extendOptions能找到。

|  |
| --- |
| Vue.extend = function(extendOptions) {              extendOptions = extendOptions || {};              // this在这个例子其实就是Vue。              var Super = this;              // Appson这个组件的构造函数，sub是一个函数  **var Sub = function VueComponent(options) {**  **// 这个\_init就是调用的Vue.prototype.\_init**  **this.\_init(options);**  **};**              // 把Vue.prototype生成一个              // { \_\_proto\_\_: Vue.prototype }这样的对象，              // 直接赋值给子组件构造函数的prototype              // 此时子组件构造函数的原型链上就可以拿到Vue的原型链的属性了              Sub.prototype = Object.create(Super.prototype);              Sub.prototype.constructor = Sub;              // 合并Vue.option上的一些全局配置**extendOptions，是我们传进来的配置，会作为构造函数的属性**  **Sub.options** = mergeOptions(Super.options, **extendOptions**);              Sub["super"] = Super;              // 拷贝静态函数              Sub.extend = Super.extend;              Sub.mixin = Super.mixin;              Sub.use = Super.use;              // 返回子组件的构造函数  **return Sub;**          }; |

（2）当在页面上使用组件的时候，创建的实例，它的data数据的来源，根据源码

|  |
| --- |
| 经过一系列的操作，创建好了这个实例，它会经在**initInternalComponent函数**中，把子组件构造函数上保存的 options 再转移到vm.$options.\_\_proto\_\_上（也就是实例上边）。          var opts = (vm.$options = Object.create(**vm.constructor.options**));  **vm.constructor.options的初步解析，vm它是实例，获取构造函数（就是由extend创建的构造函数）中的options，就是我们配置的data和methods。（Sub.options里边就存了我们配置的data，和method）**  注意：组件的构造函数在全局注册的时候就初始化好，后面使用组件的时候，都用这个构造函数去创建组件实例，它们实例中的data和methods都是来自于构造函数中的options，都指向了同一个地址。 |

（3）为什么组件中的data属性必须定义为一个方法（也就是一个函数）

经过（1）（2）的说明，已经知道同一个组件实例（可以用很多次，用一次就有一个新实例被创建）中的data和method，都指向了同一块数据（构造函数的options属性，它是一个对象）。

所以这里用一个简化的例子来说明。

|  |
| --- |
| 1、假设options的data是一个对象，在Vue中组件中的data如果是对象的话，是无法通过编译的。这里只是做一个假设。          var options = {              data: {                  name: 'Tom',              },          };          var instance1 = {};          var instance2 = {};  这里两个实例（这里并不是很准确，要通过new的方式得到的对象才叫实例，这里简单处理），的datas属性都指向了，同一个options对象的地址。  所以两个实例的data会相互影响，instance1如果改变datas.data，那么instance2也会跟着改变，这是我们不想看到的结果。  **instance1.datas= options;**  **instance2.datas= options;**  instance1.options.data.name = 'Jerry';  本来是想instance1的名字叫Jerry结果2也跟着变了。          console.log(instance1.datas.data.name);          console.log(instance2.datas.data.name); |

|  |
| --- |
| 2、采用函数的形式，并返回一个**全新**的对象。  每当我们在页面上引用一次组件，Vue会帮我们去调用这个 data 函数，从而得到一个当前组件的私有数据对象。  一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝（Vue.js官网）  红色部分一定要是一个新的对象，要写成变量的形式（对象的引用），那就跟（1）一样了。  var options = {  data: function () {  **return {**  **name: 'Tom',**  **}**  }  };  var instance1 = {};  var instance2 = {};  instance1.datas = options;  instance2.datas = options;  写成函数要调用一下才能获取到新对象  var d1 = instance1.datas.data();  var d2 = instance2.datas.data();    d1.name = 'Jerry';  console.log(d1.name);// 输出Jerry  console.log(d2.name);// 输出Tom |

3、动态组件

（1）使用flag标识符结合v-if和v-else切换组件

v-if当条件为true的时候显示

v-else是当条件为false的时候显示

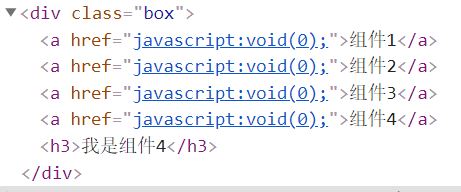
两个组件的话，布尔值还是能使用的，但是三个、四个就用不了，局限性比较大。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <input type="button" value="显示登录" @click="flag=true">  <input type="button" value="显示注册" v-on:click="flag=false">  **// v-if中的flag是vm的数据，不能写到组件的h3中**  <login **v-if="flag"**></login>  <register **v-else="flag"**></register>  </div>  <template id="login" >  <h3>登录组件</h3>  </template>  <template id="register" >  <h3>注册组件</h3>  </template>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  Vue.component('login', {  template: '#login',  });  Vue.component('register', {  template: '#register',  });  var vm = new Vue({  el: '.box',  data: {  flag: true,  },  methods: {  },  });  </script>  </body>  </html>  1 |

（2）通过 Vue 的 <component> 元素加一个特殊的 is 属性来实现：

通过v-bind:is的值(一个js表达式)，可以包括：已注册组件的名字，或一个组件的选项对象。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <a href="javascript:void(0);" v-on:click="comname='com1'">组件1</a>  <a href="javascript:void(0);" v-on:click="comname='com2'">组件2</a>  <a href="javascript:void(0);" v-on:click="comname='com3'">组件3</a>  <a href="javascript:void(0);" v-on:click="comname='com4'">组件4</a>  <component **v-bind:is**="comname"></component>  </div>  <template id="com1" >  <h3>我是组件1</h3>  </template>  <template id="com2" >  <h3>我是组件2</h3>  </template>  <template id="com3" >  <h3>我是组件3</h3>  </template>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  Vue.component('com1', {  template: '#com1',  });  Vue.component('com2', {  template: '#com2',  });  Vue.component('com3', {  template: '#com3',  });  Vue.component('com4', {  template: `<h3>我是组件4</h3>`,  });  var vm = new Vue({  el: '.box',  data: {  comname: 'com1',  },  methods: {  },  });  </script>  </body>  </html> |



十七、父组件、子组件之间传值

1. 父组件向子组件进行传字符串值，通过 props 向子组件传递数据

props 是你可以在组件上注册的一些自定义 attribute。当一个值传递给一个 props attribute 的时候，它就变成了那个组件实例的一个 property。

一个组件默认可以拥有**任意数量的 prop**，**任何值**都可以传递给任何 prop。我们能够在组件实例中访问这个值，就像访问 data 中的值一样。

在Vue中，默认子组件是无法直接获取父组件中的数据。

（1）子组件通过身上的自定义属性，来接收父组件的传递的值，通过 v-bind: 绑定到子组件身上。

**<soncom :valueformfather="value"></soncom>**

（2）在子组件中，不能直接使用父组件传递过来的数据，需要使用props 数组中的自定义属性来接收：

props: [**‘valueformfather’**]

注意：props的时候中的名称，一定要和子组件v-bind绑定的自定义属性，名称保持一致！

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <soncom **:father-value**="value"></soncom>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  value: '我从父组件传给子组件的值',  },  methods: {},  components: {  soncom: {  template: '<h3>父组件的值：{{ fatherValue }}</h3>',  **// props: ['fatherValue'],**  props: {  fatherValue: String,  }  }  }  });  </script>  </body>  </html> |

（3）props注意事项，写的时候不要少了s，官网解释的没有s，容易混淆

|  |
| --- |
| 1、传递静态或动态 Props  （1）可以像这样给 props 传入一个静态的值：（用得比较少）  <blog-post title="My journey with Vue"></blog-post>  （2）prop 可以通过 v-bind 动态赋值。  2、[Props 的大小写 (camelCase vs kebab-case)](https://cn.vuejs.org/v2/guide/components-props.html" \l "Prop-%E7%9A%84%E5%A4%A7%E5%B0%8F%E5%86%99-camelCase-vs-kebab-case" \o "Prop 的大小写 (camelCase vs kebab-case))  HTML 中的 attribute 名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。  当你使用 DOM 中的模板时，camelCase (驼峰命名法) 的 props 名需要使用其等价的 kebab-case (短横线分隔命名) 命名。  如果你使用字符串模板，那么这个限制就不存在了。  3、Props 类型  （1）字符串数组形式列出  （2）通常你希望每个 props 都有指定的值类型。这时，你可以以**对象形式**列出 props，这些 属性的名称和值分别是 props 各自的名称和类型。  props: {  title: String,  likes: Number,  } |

（4）props写错的话是不会报错的，直接在报错是找不到变量

2、父组件向子组件传递对象

|  |
| --- |
| <div class="box">  <soncom **:obj=**"parentObj"></soncom>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  parentObj: {  name: 'Tom',  color: 'blue',  },  },  methods: {},  components: {  soncom: {  template: '<h3>父组件的值：{{ JSON.stringify(**obj**) }}</h3>',  props: [**'obj'**],  }  }  });  </script> |

3、父 向 子传递方法

（1）向子组件传递methos中的方法，则使用 事件绑定的形式 v-on。相当传个引用过去

（2）子组件在自己的methods中去调用this.$emit(‘事件名’)的方式。

注意：名字的命名要注意驼峰命名，事件名字在html中都会被转成小写的，然后你在子组件还是使用驼峰命名是获取不到，要转成全部的小写。

|  |
| --- |
| <div class="box">  <soncom @**funfrom**="say"></soncom>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  },  methods: {  say: function () {  console.log('我是父组件的methods中的方法');  },  },  // 私有组件  components: {  soncom: {  template: '<h3 @click="btnClick">我是子组件</h3>',  methods: {  btnClick: function () {  this.$emit(**'funfrom'**);  },  },  },  }  });  </script> |

4、**子**组件向**父**组件传递数据

就是在父组件给组件传递方法，在那个调用方法时，把子组件的数据，当参数传递，函数把参数赋值给父组件的data中的变量，这样就达到了传值的效果。this.$emit(‘事件名’，参数)。

|  |
| --- |
| <div class="box">  <soncom @tofather="**getSondata**"></soncom>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  value: '',  },  methods: {  getSondata: function (**sonData**) {  **this.value = sonData;**  console.log('子组件传过来的数据：' + this.value);  },  },  // 私有组件  components: {  // 不要把组件的data或者method写到外边了  soncom: {  data: function() {  return {  sonValue: '我是子组件中的data的数据',  };  },  template: '<h3 @click="btnClick">我是子组件</h3>',  methods: {  btnClick: function () {  this.$emit('tofather', **this.sonValue**);  },  },  },  }  });  </script>  12 |

5、评论案例-练习父子组件传值

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style type="text/css">  textarea {  vertical-align: top;  }  </style>  </head>  <body>  <div class="box">  <command @update="updateList"></command>  <ol>  **// 每次循环就处理一个item，而不是直接把整个list丢进去**  <showinfo v-for="(item, index) in list" **:info="item"** :key="index" :index="index" @deldata="deldata"></showinfo>  </ol>  </div>  **<!-- 模板 -->**  **<template id="command">**  **<div>**  **<label>**  **评论人：<input type="text" v-model="name">**  **</label>**  **内容：<textarea v-model="content"></textarea><br />**  **<input type="button" value="提交" @click="postInfo">**  **</div>**  **</template>**  **<template id="showInfo">**  **<li>**  **<h3>评论人：{{ info.name }}</h3>**  **<h5>内容：{{ info.content }}</h5>**  **<a href="javascript:void(0);" @click="del(index)">删除</a>**  **</li>**  **</template>**    <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  // 创建全局组件  Vue.component('command', {  template: '#command',  data: function () {  return {  name: '',  content: '',  }  },  methods: {  postInfo: function () {  **this.$emit('update', {**  **name: this.name,**  **content: this.content,**  **});**  this.name = '';  this.content = '';  }  }  });  Vue.component('showinfo', {  template: '#showInfo',  props: ['info','index'],  methods: {  del: function (index) {  this.$emit('deldata', index);  }  }  });    var vm = new Vue({  el: '.box',  data: {  list: [  {name: 'Tom', content: 'hello'},  {name: 'Mike', content: 'hi'},  {name: 'jerry', content: '你好'},  {name: 'Jon', content: 'good'},  ],  },  methods: {  **updateList(list) {**  **this.list.push(list);**  **},**  deldata: function (index) {  this.list.splice(index, 1);  }  },  });  </script>  </body>  </html>  1  子组件给父组件传值，简单点，就是获得父组件的方法，然后传参，父组件对这个参数在进行处理。 |

6、vue中$refs的使用，获取Dom元素对象和组件

**注意：它是只读的，不能用来设置表单控件（input）value值，会导致问题。（如mui中的数字框），查官网**

**ref** 被用来给元素或子组件注册引用信息。引用信息将会注册在父组件的 $refs 对象上。如果在普通的 DOM 元素上使用，引用指向的就是 **DOM 元素**；如果用在子组件上，引用就指向**组件实例**：

$refs 只会在组件渲染完成之后生效，并且它们不是响应式的。这仅作为一个用于直接操作子组件的“逃生舱”——你应该避免在模板或计算属性中访问 $refs。

想要获取谁，就给它加上ref属性，ref=”xxx”，reference，引用的意思，就给设置一个引用的名字。

（1）获取Dom元素

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <input type="button" value="获取ref的属性值" @click="getRef" ref="btn">  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  },  methods: {  getRef: function () {  console.log(this);  // 获取按钮上的名字  console.log(this.$refs.btn.value);  }  },  });  </script>  </body>  </html> |

（2）获取组件

有时候就不用父组件去传值，直接能修改子组件中的data数据，直接也能去调用子组件中的方法。

|  |
| --- |
| <body>  <div class="box">  <input type="button" value="获取ref的属性值" @click="getRef" ref="btn">  <show ref="**showCom**"></show>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  // 创建全局组件  Vue.component('show', {  template: '<h3>我是组件，{{ msg }}</h3>',  data: function () {  return {  msg: 'hello',  }  },  methods: {  show: function () {  console.log('我是子组件的show方法');  }  },  });  var vm = new Vue({  el: '.box',  data: {    },  methods: {  getRef: function () {  console.log(this);  // 获取按钮上的名字  console.log(this.**$refs.**showCom.msg);  this.**$refs.**showCom.show();  this.**$refs.**showCom.msg = '切换成hi';  }  },  });  </script>  1 |

7、在vue**组件**中data和props的区别

（1）data 在组件中，被定义成function并返回一个新对象。

（2）props 在组件中，被定义成数组，其中，数组的值都是字符串名，表示父组件传递过来的数据。

（3）data 上的数据，都是组件自己私有的，data 上的数据，都是可读可写的。

props 数据，都是外界传递过来的数据，props 中的数据建议是只读（去修改它的值，Vue会给出警告）

（4）props单向数据流

所有的 props 都使得其父子 props 之间形成了一个****单向下行绑定****：父级 props 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外变更父级组件的状态，从而导致你的应用的数据流向难以理解。

额外的，每次父级组件发生变更时，子组件中所有的 props 都将会刷新为最新的值。这意味着你****不****应该在一个子组件内部改变 props。如果你这样做了，Vue 会在浏览器的控制台中发出警告。

|  |
| --- |
| 1、这个 props 用来传递**一个初始值**；**用来初始化**  这个子组件接下来希望将其作为一个本地的 props 数据来使用。在这种情况下，最好定义一个本地的 data property 并将这个 props 用作其初始值：  props: ['initialCounter'],  data: function () {  return {  counter: this.initialCounter  }  }  2、这个 prop 以一种原始的值传入且需要进行转换。  在这种情况下，最好使用这个 prop 的值来定义一个计算属性：  props: ['size'],  computed: {  normalizedSize: function () {  return this.size.trim().toLowerCase()  }  } |

（5）\*在 JavaScript 中**对象和数组**是通过引用传入的，所以对于一个**数组或对象类型**的 props 来说，在子组件中改变变更这个对象或数组本身将会**影响到父组件的状态**

|  |
| --- |
| <body>  <div class="box">  <h1>父组件 {{ fatherInfo }}</h1>  <show :son-info="fatherInfo"></show>  </div>  <script type="text/javascript" src="./lib/vue-2.4.0.js"></script>  <script type="text/javascript">  // 创建全局组件  Vue.component('show', {  template: `<div>  <h3>我是组件data，{{ msg }} </h3>  <h3>我是组件props，{{ sonInfo }} </h3>  <input type="button" value="修改子组件data和props" @click="change">  </div>`,  data: function () {  return {  msg: 'hello',  }  },  props: ['sonInfo'],  methods: {  change: function () {  // 修改data没问题  this.msg = 'hi';  // 父组件传了对象的引用过来，子组件修改属性，会影响父组件  this.sonInfo.name = 'Jerry';  // 直接改了props，会警告  // this.sonInfo = 'gogogo';  },  },  });  var vm = new Vue({  el: '.box',  data: {  fatherInfo: {  name: 'Tom',  age: 34,  },  },  methods: {},  });  </script>  3  12 |

注：（这样子做感觉没有意义，不如直接重新定义一个属性来的快，暂时未找到其运用场景）props 的数据，不要直接拿来修改，如果想要修改，必须在 data 上重新定义一个属性，然后把属性的值 从 this.props 拿过来；

8、兄弟组件通信

对于这种情况可以通过查找⽗组件中的⼦组件实现，也就是 this.$parent.$children，在 $children 中可以通过组件 name 查询 到需要的组件实例，然后进⾏通信。

|  |
| --- |
|  |

9、异步组件

插槽

注意：template只起到包裹元素的作用，不会被渲染为任何标签

在 **2.6.0** 中，我们为具名插槽和作用域插槽引入了一个新的统一的语法 (即 v-slot 指令)。它取代了 slot 和 slot-scope属性。（并不是说<slot></slot>）。

v-slot只能用在组件或者template身上。

1、什么是插槽

（1）在定义子组件的时候，可以使用slot标签定义一个占位符，专业的名字，叫做插槽。

（2）作用：在定义子组件的时候，插槽的位置，不确定要渲染成什么样子的内容，因此，可以放置一个slot进行占位，将来，外界（一般为父组件）使用这个子组件的时候，可以指定具体要渲染的内容，外界指定的内容，就会被渲染到插槽中。

2、单个插槽（匿名插槽--没有name属性）

（1）定义插槽，在子组件作用域中，使用<slot></slot>定义一个插槽

1. 使用插槽：在父组件作用域中使用带有插槽的组件时，子组件内的区域中的内容（<son>hello</son>），hello会被插入到插槽中显示。

（3）注意：在一个组件的定义中，只允许出现一次匿名插槽（测试后使用多次，不会报错，显示一样的内容）。插槽内可以包含任何模板代码，包括 HTML、甚至其它的组件。

|  |
| --- |
| Son.vue  <template>  <div>  <h3>我是子组件</h3>  <slot></slot>  <h3>我上边是插槽</h3>  </div>  </template> |
| <template>  <div>  <h2>我是根组件</h2>  <son>  <mark>我是html高亮显示标签，我是插槽内容</mark>  </son>  </div>  </template>  <script>  import son from './Son.vue'  export default {  components: {  son,  },  }  </script> |

3、多个插槽（具名插槽，组件可能有多个slot）

（1）定义具名插槽：使用name属性为slot插槽定义具体名称；<slot name=”header”></slot>。

1. 使用具名插槽。

在父作用域中使用带有命名的插槽的组件时，需要为内容指定slot=“插槽name”（在2.6版本不适用了，使用v-slot指令来代替）来填充到指定名称的插槽。

<template> 元素中的所有内容都将会被传入相应的插槽。

任何没有被包裹在带有 v-slot 的 <template> 中的内容都会被视为默认插槽的内容。

注意 v-slot 只能添加在 <template> 上 (只有一种例外情况)。

（3）一个不带 name 的 <slot> 出口会带有隐含的名字“default”。

如果你希望更明确一些，仍然可以在一个 <template> 中包裹默认插槽的内容：

<template v-slot:default>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

</template>

|  |
| --- |
| <template>  <div>  <h3>我是子组件</h3>  <slot></slot>  <h3>我上边是插槽</h3>  <slot name="s1"></slot>  <slot name="s2"></slot>  </div>  </template> |
| 这种写法在2.6中废弃了，在vue3中不支持  <template>  <div>  <h2>我是根组件</h2>  <son>  <h2>没有slot属性</h2>  <mark slot="s1">我是html高亮显示标签，我是插槽内容，默认插槽</mark>  <details slot="s2">我是html高亮显示标签，我是插槽内容，默认插槽</details>  </son>  </div>  </template> |
| 2.6版本的写法，注意是冒号，不要写成 = 等于号。  <template>  <div>  <h2>我是根组件</h2>  <son>  <template>  <h2>没有slot属性</h2>  </template>  <h4>我没有在template中</h4>  <template v-slot**:**s1>  <mark>我是插槽1</mark>  </template>  <template v-slot**:**s2>  <mark>我是插槽2</mark>  </template>  </son>  </div>  </template> |

4、作用域插槽（比较重要） 它有什么样的应用场景？？

（1）定义作用域插槽：

在子组件中，使用slot定义插槽的时候，可以通过 属性传值（如果传的值是以变量的形式，那么就要使用v-bind来进行绑定） 的形式，为插槽传递数据（它在父级作用域中渲染的）， 在父级的插槽内容中可用。

（2）使用作用域插槽

在父作用域中，通过定义slot-scope = “scope”属性，接收并使用插槽数据，这个形参 scope名字是任意的，推荐使用 scope 接收。

注意，同一组件中不同插槽的作用域，是独立的。

（3）没有使用template，作用域插槽的作用域，只能接收一次，使用多次的话，后边的会将前面的内容进行覆盖。

解决方式一：如果要接收作用域插槽中的数据，而且渲染为多个标签，则必须在多个标签之外，包裹一个父元素，接收插槽中的数据。

|  |
| --- |
| App.vue  <template>  <div>  <h2>我是根组件</h2>  <son>  <h5 slot-scope="scope">{{ scope.smsg }}</h5> // smsg被sinfo覆盖  <h5 slot-scope="scope">{{ scope.sinfo }}</h5>  <h5 slot-scope="scope" slot="s1">{{ scope.msg }}</h5>  </son>  </div>  </template>  修改方式：  <template slot-scope="scope">  <h5>{{ scope.smsg }}</h5>  <h5 >{{ scope.sinfo }}</h5>  </template> |
| son.vue  <template>  <div class="wrapper">  <h3>我是子组件</h3>  <hr />  <slot smsg="smag" sinfo="hello Jon"></slot>  <slot **:**msg='msg' name="s1"></slot>  </div>  </template> |

（4）2.6版本写法，以上的写法在vue3中不支持了

在一个 <template> 元素上使用 v-slot 指令，并以 v-slot 的参数的形式提供其名称：

|  |
| --- |
| <template>  <div>  <h2>我是根组件</h2>  <son>  <template **v-slot:**default="scope">  <h5>{{ scope.smsg }}</h5>  <h5>{{ scope.sinfo }}</h5>  </template>  <template **v-slot:**s1="scope">  <h5>{{ scope.msg }}</h5>  </template>  </son>  </div>  </template> |

（5）插槽的应用场景。

在element-ui中的一些表格中，我们加上一些自己的结构，就通过“插槽”写进去，例如加上图标按钮。Element-ui中组件中应该是有<slot></slot>这样的插槽的。

作用域插槽：在element中，可以获取到子组件插槽渲染后中的内容。

十八、路由

后端路由就是解析url的路径，根据不同路劲进入不同的处理逻辑。

1、hash(#)相关知识

单页面应用程序，类似于左边栏有很多选项，然后点击后，中心页面显示不同的内容。像是Vue官网那样。



主要通过URL中的hash(#号)（哈希）来实现不同页面之间的切换，锚点。点击的时候，只会在一个页面跳来跳去的，修改了url地址，但不会发出http请求新页面。只是实现客户端页面的定位。

会修改浏览器的历史访问记录。

更详细的说明：https://www.cnblogs.com/joyho/articles/4430148.html

**URL中的hash（井号）**

（1）#的含义

#代表网页中的一个位置，其右边的字符，就是该位置的标识符。比如

http://www.example.com/index.html#print

就是代表index.html中的print位置。浏览器会自动把print位置滚动到页面可视区域内。

设置方法：

step1：设置一个锚点<a href="#print">定位到print位置</a>

step2：在页面需要定位的内容加上id="print"。例如：<div id="print"></div>

测试：step1设置的锚点，step2中id为print的内容会滚动到页面顶端（可观察滚动条的距离）。同时，页面的url末端中会出现#print的哈希值。

（2）HTTP请求不包含#

#号是用来指导浏览器动作的，对服务器端完全无用。所以，HTTP请求中不包含#。

比如，访问下面的网址：

http://jquery.com#hello

浏览器实际发出的请求时这样的：

可以看到，只是请求了http://jquery.com，没有请求"#hello"的部分。

（3）#后面的字符

在第一个#后面出现的任何字符，都会被浏览器解读为位置标识符。这意味着，这些字符都不会被发送到服务器端。

比如，下面的URL的原意是指定一个颜色值：

http://jquery.com/?color=#fff

但是浏览器实际发出的请求是：

可以看到，"#fff"被省略了。只有将#转码为%23，浏览器才会将其作为实义字符处理。也就是说，上面的网址应该被写成：

http://jquery.com/?color=%23fff

（4）改变#不触发网页重载

单单改变#后的内容，浏览器只会滚动到相应位置，不会重新加载网页。

浏览器不会重新向服务器请求页面。

（5）改变#会改变浏览器的访问历史

每一次改变#后的部分，都会在浏览器的访问历史中增加一个记录，使用"后退"按钮，就可以回到上一个位置。

这对于ajax应用程序特别有用，可以用不同的#值，表示不同的访问状态，然后向用户给出可以访问某个状态的链接。

值得注意的是，上述规则对IE 6和IE 7不成立，它们不会因为#的改变而增加历史记录。

（6）window.location.hash读取#值

window.location.hash这个属性可读可写。读取时，可以用来判断网页状态是否改变；写入时，则会在不重载网页的前提下，创造一条访问历史记录。

（7）onhashchange事件

这是一个HTML 5新增的事件，当#值发生变化时，就会触发这个事件。IE8+、Firefox 3.6+、Chrome 5+、Safari 4.0+支持该事件。

它的使用方法有三种：

1.window.onhashchange = func;

2.<body onhashchange="func();">

3.window.addEventListener("hashchange", func, false);

（8）Google抓取#的机制

默认情况下，Google的网络蜘蛛忽视URL的#部分。

但是，Google还规定，如果你希望Ajax生成的内容被浏览引擎读取，那么URL中可以使用"#!"，Google会自动将其后面的内容转成查询字符串\_escaped\_fragment\_的值。

比如，Google发现新版twitter的URL如下：

http://twitter.com/#!/username

就会自动抓取另一个URL：

http://twitter.com/?\_escaped\_fragment\_=/username

通过这种机制，Google就可以索引动态的Ajax内容。

2、前端路由

在单页面应用程序中，这种通过hash改变来切换页面的方式，称作前端路由（区别于后端路由）。

前端的路由：就是根据不同的Hash地址，在页面上展示不同的前端组件。

后续的组件是.vue组件文件。我们在html页面中创建组件，这个组件就跟html关联在一起了，这个并不是符合组件化开发的思想。

需要会配置webpack，才能使用做.vue组件文件，这个后缀vue文件，浏览器解析不了，需要一个工具去帮我们进行处理。

前端路由的目的是为了做单页面的应用程序。就是一个页面中有很多的组件，根据路由的变化显示不同的组件。

三大框架基本上就是为了为单页面应用程序。

实现单页面应用程序的基本前提：要有hash路由，还要有组件。例如阿里云的控制台界面，就是一个后台管理系统。

3、vue-router使用

上边我们需要自己对一些路由进行处理，要写onhashchange事件，这个库就帮我们封装好了这些。

Vue-router是依赖与vue的，所以要后边引入。

（1）在script中创建路由组件

我们之前注册的全局组件就不能够使用了。通过Vue.component创建的组件，它的名字是在html中，当成标签来使用的，无法传递给router实例。

要重新通过component属性给它传一个配置对象，预测它自动帮我们去注册组件了？？？

（2）创建一个路由对象，通过vm的router属性，挂载到 vm 上。

当导入完vue-router之后，在浏览器全局就有了 VueRouter构造函数。

属性routes: [ {path: xxx ,component: xxx}, {}..... ] // 路由规则数组。

hash值和组件做一个对应的关系，每一个对象表示一个路由规则。

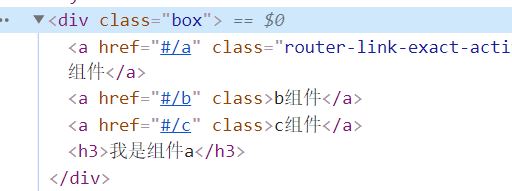
path，hash路径不带#，跟组件名称（变量）。

（3）在html中放<router-view>和放<router-view>

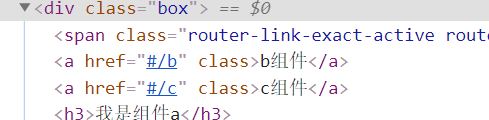
匹配到的路由组件，如果想要展示给用户，需要在页面上放<router-view>，它的作用就是把匹配到的组件，呈现到这个标签中。

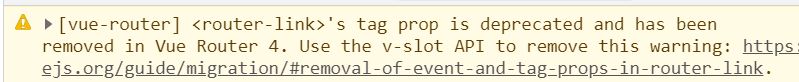
注意：不推荐使用a标签。

使用<router-link to=”路径”>**登录**</router-link>，默认是渲染成a标签。中间记得加上文字，不然是显示不出来的。



能够通过tag属性来让它渲染成别的元素。这个tag属性不能乱指定，它在新版本中已经很不赞成使用了。





（4）会自动在浏览器的url地址添加 #/ ，表示已经被路由管理起来了。

1

（5）const router = new VueRouter({ routes: xxxxxx});中**routes**不要写成**routers**了，这个点很难找到。。。

|  |
| --- |
| <body>  <div class="box">  **<!-- 使用 router-link 组件来导航. -->**  **<!-- 通过传入 `to` 属性指定链接. -->**  **<!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->**  <router-link to="/a">a组件</router-link>  <router-link to="/b">b组件</router-link>  <router-link to="/c">c组件</router-link>  **<!-- 路由出口 -->**  **<!-- 路由匹配到的组件将渲染在这里 -->**  <router-view></router-view>  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  **// 1、创建路由组件**  const coma = {  template: '<h3>我是组件a</h3>',  }  const comb = {  template: '<h3>我是组件b</h3>',  }  const comc = {  template: '<h3>我是组件c</h3>',  }  **// 2、创建路由实例，和路由规则**  const router = new VueRouter({  **//这个容易写成routers，而且很难找到**  **routes**: [{path: '/a', component: coma},  {path: '/b', component: comb},  {path: '/c', component: comc},  ],  });  var vm = new Vue({  el: '.box',  data: {  },  router: router,  // (简写) router相当于 router: router  });  </script>  </body>  </html> |

（6）流程小总结

先触发<router-link>，<router-link>修改hash值，hash值的改变，被路由对象监听到，监听到之后匹配路由规则，找到要显示组件，渲染到<router-view>上边

如果想匹配任意路径，我们可以使用通配符 (\*)：

4、路由切换高亮

（1）直接在router中提供的类样式来，添加高亮的显示就行

1. 在Vue-router官网API说明

|  |
| --- |
| active-class  类型: string  默认值: "router-link-active"  设置链接激活时使用的 CSS 类名。默认值可以通过路由的构造选项 linkActiveClass 来全局配置。 |

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style type="text/css">  .router-link-active {  color: skyblue;  font-size: 20px;  font-weight: bold;  text-decoration: none;  }  .my-active {  color: orange;  font-size: 20px;  font-weight: bold;  text-decoration: none;  }  </style>  </head>  <body>  <div class="box">  **// 忘记写文字，就没有显示出来，找了挺久的**  <router-link to="/login">登录</router-link>  <router-link to="/reg">注册</router-link>  <router-view></router-view>  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  const login = {  template: '<h3>我是登录组件</h3>'  };  const reg = {  template: '<h3>我是注册组件</h3>'  };  const router = new VueRouter({  routes: [{path: '/login', component: login},  {path: '/reg', component: reg},  ],  **linkActiveClass: 'my-active',**  });  var vm = new Vue({  el: '.box',  data: {  },  router: router,  });  </script>  </body>  </html>  1 |

5、redirect重定向

一开始打开页面是根路径，没有显示出任何的组件，可以通过redirect指定一个组件，让其显示。

|  |
| --- |
| routes: [  **{path: '/', redirect: 'login'},**  **// {path: '/', component: login},**                  {path: '/login', component: login},                  {path: '/reg', component: reg},              ],              linkActiveClass: 'my-active', |

6、添加路由切换效果

|  |
| --- |
| .v-enter {  opacity: 0;  transform: translateX(100px);  }  .v-leave-to {  opacity: 0;  transform: translateX(-100px);  }  .v-enter-active,  .v-leave-active {  transition: all 1s ease;  position: absolute;  }  **<transition>**              <router-view></router-view>  **</transition>** |

7、路由组件传参

（1）?参数，形式进行传参

在路由中，使用 **? 传参**，不需要修改对应的路由规则；

组件也有自己的声明周期函数，这些函数，和 vm 实例的声明周期函数一致。

|  |
| --- |
| <router-link to="/login?id=110">登录</router-link>  const login = {  template: '<h3>我是登录组件, 参数为：{{ **$route.query.id** }}</h3>',  created: function () {  console.log(this);  },  };  1 |

（2）/10/lxf（/:id/:name），形式的参数

它需要修改路由规则，规定他们是代表什么意思。

|  |
| --- |
| <router-link to="/login/111/lxf">登录</router-link>  const login = {  template: '<h3>我是登录组件, 参数为：{{ **$route.params.id** }} {{ **$route.params.name** }}</h3>',  created: function () {  console.log(this);  },  };  routes: [  {path: '**/login/:id/:name**', component: login},  {path: '/reg', component: reg},  ],  1 |



（3）使用 props 将组件和路由解耦（推荐使用）---布尔模式

在组件中使用 $route 会使之与其对应路由形成高度耦合（它们之间相互联系，需要我们手动去获取），从而使组件只能在某些特定的 URL 上使用，限制了其灵活性。

布尔模式：如果 props 被设置为 true，route.params 将会被设置为组件属性。

|  |
| --- |
| <router-link to="/login/111/lxf">登录</router-link>  const login = {  template: '<h3>我是登录组件, 参数为：{{ id }}------ {{ name }} </h3>',  created: function () {  console.log(this);  },  **props: ['id', 'name'],**  };  routes: [  {path: '/', redirect: 'login'},  {path: '/login/:id/:name', component: login, **props: true**},  {path: '/reg', component: reg},  ],  1 |

还有对象模式、函数模式，后续用到再进行学习。

8、匹配优先级

有时候，同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：**路由定义得越早，优先级就越高。**

|  |
| --- |
| {path: '/account', component: account},优先级高  **{path: '/account', component: news},** |

9、使用 children 属性实现嵌套路由

<router-view> 是最顶层的出口，渲染最高级路由匹配到的组件。同样地，一个被渲染组件同样可以包含自己的嵌套 <router-view>。

要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置。

要注意，以 / 开头的嵌套路径会被当作根路径。

children属性中的路径不要加上 / ，这样就变成了根路径下。加了 / 就找不到匹配的组件了。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style type="text/css">  .account {  width: 400px;  height: 400px;  border: 1px solid red;  padding-top: 20px;  }  .news {  width: 400px;  height: 400px;  border: 1px solid lightgreen;  padding-top: 20px;  }  </style>  </head>  <body>  <div class="box">  <router-link to="/account">账户页面</router-link>  <router-link to="/news">新闻页面</router-link>  <router-view></router-view>  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  var account = {  template: `  <div class="account">  我是账户组件  **<router-link to="/account/login">登录</router-link>**  **<router-link to="/account/reg">注册</router-link>**  **<router-view></router-view>**  </div>  `,  };  var news = {  template: `<div class="news">我是新闻组件</div>`,  };  var login = {  template: `<h3>我是登录组件</h3>`,  }  var reg = {  template: `<h3>我是注册组件</h3>`,  }  var router = new VueRouter({  routes: [  **{path: '/', redirect:'account'},**  {path: '/account', component: account,  **children: [**  **{path: 'login', component: login},**  **{path: 'reg', component: reg},**  **],**  },  {path: '/news', component: news},  ],  });  var vm = new Vue({  el: '.box',  router: router,  });  </script>  </body>  </html> |

1. 没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个 空的 子路由：  
    这句话的意思就是一个空的路由，什么都不传，下边例子作为说明。它表示得有点歧义。

一开始奔着路由规则修改去了。改错位置了。要改<router-link>的路由

|  |
| --- |
| routes: [  // {path: '/', redirect:'account'},  {path: '/account', component: account,  children: [  {path: 'login', component: login},  {path: 'reg', component: reg},  **{path: '', component: nom},**  ],  },  {path: '/news', component: news},  {path: '', component: nom},  ],  （1）最高级路由匹配，第一层的，黄色部分  访问的url中/html\_study/workplace/Vue\_practice/vue.html#/ 不加hash，什么都不加，就是空路由。nom组件就能生效。  /html\_study/workplace/Vue\_practice/vue.html#/abc，这个什么都不显示，因为没有匹配的组件，现在它不是一个空路由了，所以nom是不被渲染的。  （2） 二层，红色  /html\_study/workplace/Vue\_practice/vue.html#/account，二级路由是空路由，nom对应空路由，就被渲染。  /html\_study/workplace/Vue\_practice/vue.html#/account/hi，二级路由不是空路由，nom不被渲染。 |

（2）如果想匹配任意路径，我们可以使用通配符 (\*)：

这个就跟空路由，有点区别。这个不能写在第一位，这样后续的路由规则都不会有机会匹配了。

10、命名路由<router-view name=”xxx ”>

Node.js还是很重要的，electron做桌面应用的，也是基于node.js去做的。

匹配一个路由的路径要显示两个不同的组件。之前是无法做到的，之前的路由规则，当匹配到一个路由的时候，只能在页面容器中，放置唯一的一个组件。因为一个路由就匹配到了一个组件，所以容器都显示一个组件。路由跟组件是一对一的关系。

（1）当一个路由规则，要展示多个组件的就需要用到命名视图。

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置 (带上 s)：

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style type="text/css">  .header {  height: 100px;  background-color: skyblue;  }  .left {  width: 100px;  height: 200px;  background-color: yellowgreen;  float: left;  }  .right {  height: 200px;  background-color: pink;  /\*  bfc盒子会紧贴浮动盒子的边缘，不加的话，它的div会被挡着  \*/  overflow: hidden;  }  .footer {  height: 200px;  background-color: purple;  }  .test {  width: 100px;  height: 100px;  border: 1px solid #000;  }  </style>  </head>  <body>  <div class="box">  <router-view name="**header**"></router-view>  <div class="content">  <router-view name="**left**"></router-view>  <router-view name="**right**"></router-view>  </div>  <router-view name="**footer**"></router-view>  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  const header = {  template: '<div class="header">头部区域</div>',  }  const left = {  template: '<div class="left">左侧区域</div>',  }  const right = {  template: '<div class="right"><div class="test"></div></div>',  }  const footer = {  template: '<div class="footer">底部区域</div>',  }  const router = new VueRouter({  routes: [  {path: '/', components: {  **header: header,**  **left: left,**  **right: right,**  **footer: footer,**  }  },  ],  });  var vm = new Vue({  el: '.box',  router: router,  });  </script>  </body>  </html>  1 |

11、编程式导航

编程式导航就是使用js代码的方式，来实现路由的跳转。

在组件实例的身上，有this.$route和this.$router。

this.$route是专门用来获取路由中参数的。

this.$router是专门用来实现编程式导航的。

this.$router.push('/home/goodslist/goodsdetail/goodsdesc/'+this.detailContent.id);

// 这里的name是在router.js中配置的

// {path: 'xxxx', xxxx, name: 'goodsComment'},

this.$router.push({name: 'goodsComment', params: {id: this.detailContent.id}});

应用场景：一些引入第三方的组件（如mui的按钮），如果要使用路由跳转，不方便使用router-link，不知道它原来是一个什么样的元素，就可以使用

12、路由导航守卫（简单的理解）

在路由跳转，显示组件之前，判断一下你是否有权限，访问这个组件，没有就显示，守卫做的类似这个事的。在路由规则和组件的显示过程中，做一层拦截，并不是谁都能进行访问组件，提供安全机制。

“导航”表示路由正在发生改变。

（1）router.beforeEach 注册一个全局前置守卫：

这个router实例中的每个路由规则之前，都要先调用一个指定的回调函数，如果函数给你放行了，那么你可以看到你想看组件

通过路由导航守卫，控制有权限页面的访问，只有登录以后，才允许访问高级的页面。

（2）API语法：

参数1：是要去的那个页面路由相关的参数，新地址

参数2：从哪个页面即将离开

参数3：next是一个函数

const router = new VueRouter({ ... })

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

// ...

在函数体内调用next就能直接去访问组件了

})

to.path表示我们下一刻要访问哪个地址。

from.path表示我们上一刻，所访问的是哪个地址。

next(xx)，表示强制跳转到xxx地址

（3）登录小案例

C:\Users\ASUS\Desktop\html\_study\workplace\Vue\_practice\vue\_webpack\router\_navigation\_guard

如果没有进行登录，就不能访问welcom页面。如果没有导航守卫，那么通过路由是能直接访问到的。加了之后，就不能访问了，会出现空白或者跳转到登录页面，这个是程序设置的。

登录的话，服务器会判断我们输入的信息，成功的话会给我们一个token令牌，这个会保存在sessionStorage中，页面关闭就失效了。如果不关闭的话，就能直接访问，不用多次登录后才能访问。

客户端拿到token之后，在请求其他的页面的时候，会将token一起给服务器，服务器验证成功后在给你访问，否则你可以在本地修改token（在F12中sessionStorage加上token），就能访问了

视频中为什么要return next（）。对不同情况做处理，return就不用执行下边的语句。

|  |
| --- |
| // 如果这里不加路由导航守卫，那么其他的router-link很可能会不经过登录就能够访问  router.beforeEach((to, from, next) => {  if (to.path === '/login') {  next();  return;  }  // 判断是否有登录的令牌,这里加上return，是为了不让程序继续判断下去。  const token = window.sessionStorage.getItem('token');  if (!token) return next('/login');  next();  // console.log(to);  // console.log(from);  }); |

十九、Vue中计算属性和侦听器

1、watch属性的使用（侦听器）

表示要监听指定的数据的改变，他的属性（可以是一个**合法的表达式，能监听当前 vm 实例上所有的属性变化**）是要监听的数据，值是数据变化时的处理函数。

使用watch可以监听虚拟的数据，例如你要监听一个变量a，它可能在某个时刻变化了，他并没有像input那样有change，keyup事件监听，总不能来个循环一直判断。这样就卡死了，js是单线程的。

比较适合监听单个数据

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  姓：<input type="text" v-model="first" />  名：<input type="text" v-model="second" />  姓名：<input type="text" v-model="name" />  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  name: '',  first: '',  second:'',  },  watch: {  first: function (newValue, oldValue) {  console.log(newValue, oldValue);  this.name = this.first + this.second;  },  second: function (newValue, oldValue) {  this.name = this.first + this.second;  },  }  });  </script>  </body>  </html>  1 |

（1）watch监听url，watch在F5刷新的时候，不会被触发，因为所有的数据都重置了，并没有数据的改变。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  <router-link to="/login">登录</router-link>  <router-link to="/reg">注册</router-link>  <router-view></router-view>  <h1> {{ msg }}</h1>  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  const login = {  template: '<h3>我是登录组件</h3>',  }  const reg = {  template: '<h3>我是注册组件</h3>',  }  const router = new VueRouter({  routes: [  {path: '/login', component: login},  {path: '/reg', component: reg},  ],  });  var vm = new Vue({  el: '.box',  data: {  msg: '',  },  router: router,  created: function () {  const path = this.$route.path;  if (path === '/login') {  this.msg = "欢迎登录";  } else if (path === '/reg') {  this.msg = "欢迎注册";  }  },  watch: {  **'$route.path'**: function (newValue, oldValue) {  if (newValue === '/login') {  this.msg = "欢迎登录";  } else if (newValue === '/reg') {  this.msg = "欢迎注册";  }  }  }  });  </script>  </body>  </html> |

2、computed计算属性的基本使用

计算属性，在页面中，使用计算属性（它是一个函数）的时候，永远是把他当作普通的属性来使用。

只要计算属性的function中，所依赖的任何数据发生了变化，则会触发计算属性的重新求值。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  姓：<input type="text" v-model="first">  名：<input type="text" v-model="second">  姓名：<input type="text" v-model="fullname">  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  first: '',  second: '',  },  computed: {  fullname: function () {  return this.first +':'+this.second;  }  },  });  </script>  </body>  </html> |

（1）计算属性缓存 vs 方法

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是**基于它们的响应式依赖进行缓存**的。只在相关响应式依赖发生改变时它们才会重新求值。

这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

|  |
| --- |
| <h1>{{ fullname }}</h1>  <h1>{{ fullname }}</h1>  <h1>{{ fullname }}</h1>  computed: {  fullname: function () {  console.log('computed');  return this.first +':'+this.second;  }  },  1 |

（2）为什么需要缓存？

假设我们有一个性能开销比较大的计算属性 ****A****，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 ****A****。如果没有缓存，我们将不可避免的多次执行 ****A**** 的 getter！如果你不希望有缓存，请用方法来替代。

（3）计算属性 vs 侦听属性

Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：**侦听属性**。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用**计算属性**而不是命令式的 watch 回调。

|  |
| --- |
| 如果是姓名那个案例  computed只需要一个函数就可以  watch需要去监听两个属性，使用两个相同的函数。  这种情况下，计算属性比较合适，但是看具体的应用场景，如路由一般用的watch，它只有hash值的变化，计算属性也行。 |

3、computed计算属性与watch区别

（1）computed用来监控**自己定义的变量**，**该变量不在data里面声明**，直接在computed里面定义，然后就可以在页面上进行双向数据绑定展示出结果或者用作其他处理；

用途：computed比较适合对**多个变量或者对象进行处理后返回一个结果值**，也就是数多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化，举例：购物车里面的商品列表和总金额之间的关系，只要商品列表里面的商品数量发生变化，或减少或增多或删除商品，总金额都应该发生变化。这里的这个总金额使用computed属性来进行计算是最好的选择

（2）watch主要用于监控**vue实例属性**的变化，它监控的变量当然必须在**data里面声明**才可以，它可以监控一个变量，也可以是一个对象

用途：watch一般用于**监控路由**、**input**输入框的值特殊处理等等，它比较适合的场景是**一个数据影响多个数据**

**监控路由：'$route.path': function (newValue, oldValue) {.....}**

（3）computed主要当作属性来使用的，是它函数里边的依赖发生变化（可以多个），才会计算。

而watch是监听某个具体的一个变量（需要观察的表达式）的变化（只有一个）。写法上有点区别。

4、computed中的get和set

（1）简单介绍

在 Vue 中，computed 的属性可以被视为是 data 一样，可以读取和设值，因此在 computed 中可以分成 getter（读取） 和 setter（设值），一般情况下是没有 setter 的，computed 预设只有 getter ，也就是只能读取，不能改变设值。

（2）vue.js计算属性默认只有 get，因为是默认值所以我们也常常省略不写，如下代码：

<div id="demo">{{ fullName }}</div>

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

其实computed里的代码完整的写法应该是：

computed: {

fullName: {

**get**(){

return this.firstName + ' ' + this.lastName

}

}

}

需要注意的是，不是说我们更改了getter里使用的变量，就会触发computed的更新，**前提是computed里的值必须要在模板里**使用才行。

（3）计算属性set

在template 中，我们可以看到，input 是直接绑 v-model="fullName"，如果我们这里直接**修改了fullName的值，那么就会触发setter**，同时也会触发getter以及updated函数。其执行顺序是setter -> getter -> updated。

需要注意的是，并不是触发了setter也就会触发getter，他们两个是相互独立的。

我们这里修改了fullName会触发getter是因为setter函数里有**改变firstName 和 lastName 值的代码**。也就是说我们如果注释掉上边的setter中修改firstName 和lastName的代码后就不会执行getter。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta http-equiv="X-UA-Compatible" content="IE=edge">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <div class="box">  姓：<input type="text" v-model="first">  名：<input type="text" v-model="second">  姓名：<input type="text" v-model="fullname">  </div>  <script type="text/javascript" src="./lib/vue.js"></script>  <script type="text/javascript" src="./lib/vue-router.js"></script>  <script type="text/javascript">  var vm = new Vue({  el: '.box',  data: {  first: '',  second: '',  },  computed: {  fullname: {  get: function () {  console.log('get');  return this.first + ":" + this.second;  },  **set: function (value) {**  **var str = value.split(':');**  **// 修改first和second会触发了get函数，如果把它们注释了就不会。**  **this.first = str[0];**  **this.second = str[1];**  **console.log('set');**  **},**  },  },  });  </script>  </body>  </html> |

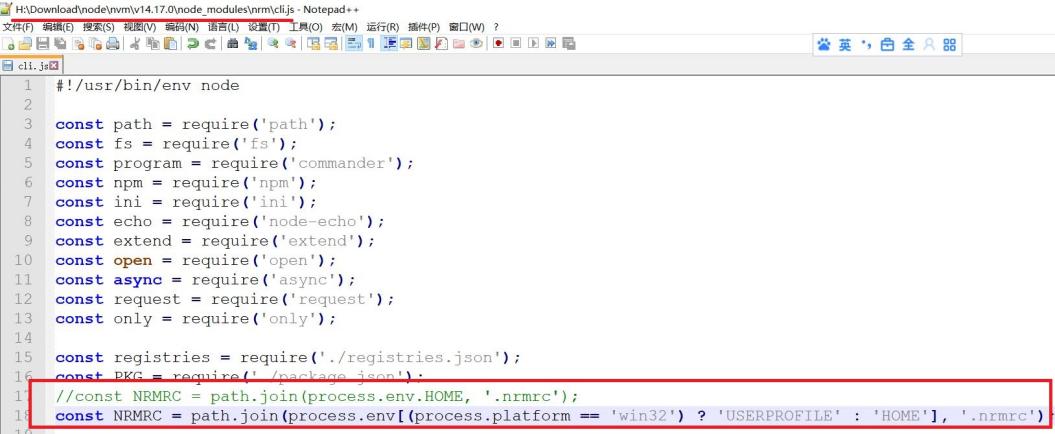
二十、nrm的安装使用（链接node笔记------九、9）

作用：提供了一些最常用的NPM包镜像地址，能够让我们快速的切换安装包时候的服务器地址； 什么是镜像：原来包刚一开始是只存在于国外的NPM服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的NPM服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

运行npm install nrm -g全局安装nrm包；

使用nrm ls查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；

使用nrm use npm或nrm use taobao切换不同的镜像源地址；

****

**const NRMRC = path.join(process.env[(process.platform == 'win32') ? 'USERPROFILE' : 'HOME'], '.nrmrc');**

1、使用cnpm工具下载和安装包

CNPM 是中国 npm 镜像的客户端

大家要区分 cnpm 这个工具 和 nrm 中 cnpm 的区别；

（1）在 nrm 中，cnpm 只是一个镜像的地址而已；

（2）在cnpm 这个工具中，它是一个具体的工具，能够像npm一样，去下载和安装包！

运行npm i cnpm -g 全局按照 cnpm 这个下载包工具可以像使用npm装包一样，去使用cnpm装包；安装jQuery为例子

npm install jquery -S

cnpm install jquery -S

二十一、webpack

1、在网页中会引用哪些常见的静态资源？

（1）样式表

.css、.less、.sass（sass旧版本，它的写法跟新版有区别）、.scss（sass新版本），它们都是css的中间件，最后都会转成css

（2）Javascript脚本文件

.js、.ts（typescript微软的）、.coffee（后边两个也中间件，最后转成的是js）

（3）图片

.jpg、.gif、.png、.bmp

（4）字体文件（Fonts）

.eot、.svg、.ttf、.woff、.woff2

（5）模板文件

.ejs .vue .jade

2、网页中引入的静态资源多了以后有什么问题

（1）加载慢：当静态资源多了以后，会发起很多的**二次资源请求**，这时候，我们的网页加载速度就慢了。

（2）会出现静态资源的前后依赖关系问题；如js文件会有引入前后的关系。

3、如何解决上述两个问题

把图片合并成精灵图、把 CSS 或 JS 合并、压缩一下；

把图片转为 base64 的编码格式；

可以使用即将学到的webpack去解决静态资源的依赖问题；

4、项目中使用 webpack 有什么好处：

能够处理静态资源的 依赖关系；

能够优化项目代码，比如：压缩合并文件，把图片转为base64编码格式；

能够把高级的语法转为低级的语法；（ES6、ES7有些语法浏览器不能识别）

webpack 能够转换一些模板文件； .vue

5、webpack安装的两种方式

一般需要全局（供自己在本地使用webpack命令，同时要装上cli，它是命令行工具），和本地安装，为了别人以后用的跟你同个版本的webpack。两者没有关系

最好是装在我们开发依赖--save-dev

也可以在项目中安装 webpack, 运行cnpm i webpack -D 安装到项目开发依赖即可；

6、初步使用webpack打包构建列表隔行变色案例

（1）相关文件夹说明

src资源放开发的源代码

dist放发布的代码

（2）初步使用webpack方式，ES6导入语法

在浏览器中，在一个js文件中，能否导入另外的一个js脚本文件，默认是不行的。

为了解决浏览器中天生没有模块化的问题，ES6中提出了 import 和export的概念。

注意：在ES6中使用 import 导入其他js模块的语法，chrome浏览器尚未支持。Node环境也不支持，它们俩用的都是v8引擎。

Node遵循commonjs规范（他就对模块化规范，require，export等等），chrome浏览器没有，所以解析不了require，所以需要通过webpack在底层对它们进行转换。

import $ form ‘jquery’，这个是ES6的写法，就是把jquery导出到 $。

const $ = require(‘jquery’)，这是node的导出的写法。

export default {

name: ,

age:,

}

这样子在js模块中向外暴露了一个对象成员。

7、webpack.config.js使用，简化打包时候的命令，直接输入 webpack

在这个配置文件中，必须显示的声明，要处理的文件的路径，和输出文件的路径。视频中的命令行是3.x版本的，高版本已经没有了

注：它是一个js文件，使用node中的语法，向外暴露了一个配置对象，供webpack执行的时候来读取这个配置对象。因为**webpack这个构建工具，底层就是使用了node.js开发出来的**。

涉及到了路径，所以要导入path模块

不要进错目录了。

|  |
| --- |
| const path = require('path');  module.exports = {  entry: path.join(\_\_dirname, './src/main.js'),  output: {  path: path.join(\_\_dirname, './dist'),  filename: 'bundle.js',  },  }; |

webpack入口默认为src/index.js 进入webpack-test文件夹新建文件夹 src，进入src文件新建index.js。

8、webpack-dev-server实时打包编译

开发期间的服务器，它是装在本地的，为什么用就package.json中的script，用npm run能跑起来？？？

运行起来之后，整个目录相当于被托管成一网站，你可以去访问看看

自动编译之后的bundle.js跑哪里去了，webpack-dev-server并把它没有放到实际上的物理磁盘，而是仅仅在内存当中，但是这个bundle.js被托管到了内存中，在以http访问的根目录中，有一个虚拟的、看不见的bundle.js。如果通过file这种协议去访问就没有效果了。

我们经常会使用ctrl+s保存，它就自动去编译，如果存在磁盘，会大量操作磁盘，耗费多，对磁盘也有一定的损耗，在内存处理是最快的，cpu基本不是问题，但是磁盘可能比较慢。

（1）每次修改完 项目代码，都需要手动运行一下 webpack 命令进行打包构建，比较麻烦。

（2）可以在项目本地安装一个webpack-dev-server的工具来实现实时打包构建的效果。

（3）webpack-dev-server这个工具，需要在项目本地安装，同时，它依赖于项目本地的webpack。

npm install webpack --save-dev

npm install webpack-dev-server --save-dev

（4）需要打开package.json这个配置文件，找到 scripts 节点，在其中，新增一个dev脚本命令

|  |
| --- |
| "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "dev": "webpack server" // 注意我这里使用的都是最新版本的，低版本的使用的是 webpack-dev-server  },    "webpack": "^5.40.0",  "webpack-cli": "^4.7.2",  "webpack-dev-server": "^3.11.2" |

（5）只要在终端中，运行npm run dev命令，去执行 dev脚本，启动webpack server这个实时构建编译的服务器。（用npm run 是因为它是装在本地开发环境中，不是全局安装的）

这个dev脚本命令也需要依赖webpack.config.js去获取入口文件，和输出文件的路径。

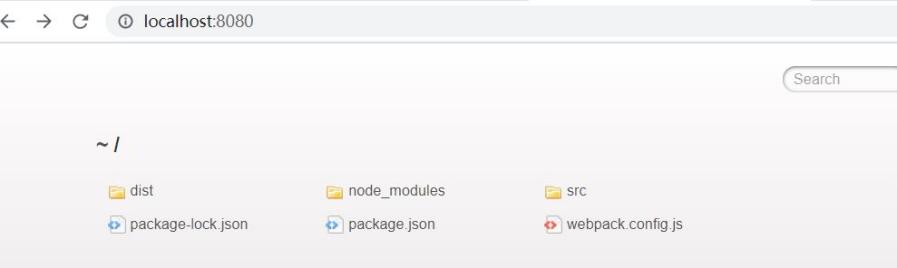
它指的是webpack-dev-server。

它首先会去项目中寻找webpack.config.js这个文件，如果有，它会去读取webpack.config.js（只能用node.js写的，它是运行在node环境的，它是基于node构建的）中，暴露出来的配置对象，webpack去加载指定的配置。运行起来之后，会创建一个小型的服务器，在http://localhost:8080这里。它会监听代码的变化，保存后实时编译

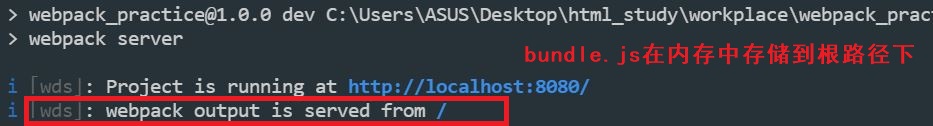
（6）webpack-dev-server打包出来的 bundle.js 并没有存放到实际的物理磁盘，而是托管到了内存中！托管的路径是**项目的根目录**，所以，在引用的时候，script 标签的路径应该这样写：在后边发布的时候，是需要进行修改的。目前只是方便调试（这个不用处理，html插件帮我们直接处理好了）

   <!-- <script type="text/javascript" src="../dist/bundle.js"></script> -->

    <script type="text/javascript" src=**"/bundle.js**"></script>







9、webpack server可以追加一些参数

（1）第一种方式webpack server --open --port 3000 --host 127.0.0.1 --hot

表示编译之后，自动打开浏览器，使用3000这个端口，ip地址，--hot表示你更改的部分以补丁的形式打上去，如果不加的话，那么就全部重新生成的。加上--hot效率会高一些。

--open测试后打开了IE，暂时不去搞了。

--hot需要改动才，能看到补丁（也称之为热更新）。



--port测试是有效的。

--host测试是有效的。

注意：手机与PC在同个网络下，通过配置成pc的IP地址，让手机能访问到网页内容

注意：127.0.0.1这是一个特殊的地址，它是一个本地回环的地址，自己去访问自己，别人访问不了，它既是服务器，又是客户端，回环。

（2）可以装一个自动打开浏览器的插件，可以设置默认浏览器。

（3）第二种方式（不推荐），它与第一种方式，只能选一种，否则会有冲突

修改了webpack.config.js文件，它不会重新加载，因为它在运行时候才去读取的，需要重新运行npm run dev。

5.x版本：

|  |
| --- |
| 在package.json中  {  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "dev": "webpack server",  **"dev1": "webpack server"**  },  }  webpack.config.js添加  module.exports = {  devServer: {  open: false,  port: 3000,  host: '127.0.0.1',  hot: true,  }  }; |

3.x版本

|  |
| --- |
| 1、需要先在 webpack.config.js中，添加devServer 选项：  devServer: { //指令，只能二选一。  open: true, // 自动打开浏览器  port: 3000, // 指定端口号  host: '127.0.0.1', // 指定Ip地址  hot: true // 启用热更新, 这里的 hot 指令，需要配合一个热更新的 webpack 插件才能正常使用  }  2、需要在头部导入webpack：  const webpack = require('webpack')  3、在 plugins节点中，添加一个新的插件：  new webpack.HotModuleReplacementPlugin() |

10、html-webpack-plugin插件配置启动页面

plugin是插件的意思。

没使用之前的存在的问题：它bundle.js是托管在内存中，并没有在物理磁盘上。html也加进去，加快速度？？？这不好理解。还帮我们在调试的时候，避免去切换bundle.js的问题。

（1）先运行cnpm i html-webpack-plugin -D安装到本地开发环境依赖

（2）在webpack.config.js中，导入这个插件：

导入把html页面生成到内存中的插件，导入的得到的结果是一个构造函数。

/ 导入把HTML页面生成到内存中的插件

const htmlWebpackPlugin = require('html-webpack-plugin');

（3）在webpack.config.js的配置对象中，新增一个 plugins 插件节点：

命名成index.html，有个特征，就是访问目录的时候，它默认会打开这个页面。

|  |
| --- |
| module.exports = {  .....  plugins: [  new htmlWebpackPlugin({  // 把指定的html页面，作为模板，复制一份托管到内存中去  template: path.join(\_\_dirname, './src/index.html'),  // 指定，将来在内存中复制出来的页面，名称叫做index.html  filename: 'index.html',  })  ],  }; |

（4）能够在内存中生成的页面中，自动把打包好的 bundle.js 文件，注入为一个 script 标签





（5）这个插件的作用

插件他有一个功能：就是把内存中打包的bundle.js文件，注入到内存中页面中去，路径都帮我们处理好了。这样在调试的时候，我们就不用改物理磁盘上html的引入的bundle.js文件，避免了重复切换的问题。

（6）如果需要引入其他文件，要以根目录为主，因为它是挂载在内存中的。不是以物理磁盘的目录为主。

（7）webpack发布时候，没有用webpack server，那么该插件会把本应该在内存中html文件输出到webpack.publish.config.js中out那里去，就是dist目录，输出对应html文档。

11、了解webpack解决文件之间的依赖关系（5.x版本就不是这样，只保留需要，以下两段话是针对3.6 webpack）。

webpack的内部也是发现这里有个导入了import，那么它会把import所指向的那个文件a，用一个包装器包装到一个闭包（这里需要去验证，从视频看的形式，不像闭包，答：确实是闭包，它里边使用eval函数）当中，向外暴露出一些成员，然后b里面就会通过require，去导入一些模块，去使用那些b暴露出来的东西。

webpack在打包的入口文件的时候，会把这个文件中，所有的依赖的第三方包，或者 js文件，也当做模块，一起打包到bundle.js。

简单粗暴的说法，就是把所有用到js代码，帮我们按顺序组装成一个js文件。

5.x新版打包的bundle.js比3.x的东西，这里是输出了函数，bundle.js所以就比较简单，如果是导入变量，bundle.js会有的代码。

|  |
| --- |
| 入口js文件，a.js  import b from './b.js'  console.log(b.say); |
| b.js  import c from './c.js'  // 采用ES6  export default {  say: function () {  console.log(c.say);  },  } |
| c.js  // 采用ES6  export default {  say: 'hello i from c.js',  } |
| bundle.js  得到了一个自调用函数，输出语句，最后是输出函数  (() => { "use strict"; console.log((function () { console.log("hello i from c.js") })) })(); |

/\*\*\*/ (**(\_\_unused\_webpack\_module, \_\_webpack\_exports\_\_, \_\_webpack\_require\_\_) => {**

**eval(**"\_\_webpack\_require\_\_.r(\_\_webpack\_exports\_\_);\n/\* harmony export \*/ \_\_webpack\_require\_\_.d(\_\_webpack\_exports\_\_, {\n/\* harmony export \*/   \"b\": () => (/\* binding \*/ b),\n/\* harmony export \*/   \"temp\": () => (/\* binding \*/ temp),\n/\* harmony export \*/   \"default\": () => (\_\_WEBPACK\_DEFAULT\_EXPORT\_\_)\n/\* harmony export \*/ });\n// 使用ES6\r\n\r\n// 要跟node的区别\r\nvar a = 10;\r\n\r\n\r\n\r\nlet b = 10;\r\nvar temp = 1;\r\n\r\n/\* harmony default export \*/ const \_\_WEBPACK\_DEFAULT\_EXPORT\_\_ = (a);\n\n//# sourceURL=webpack:///./es6\_study/exportDefault.js?");

/\*\*\*/ **})**,

12、初步总结

整个项目搭建webpack之上，所以一些js文件就能用模块化来开发，使用ES6、7中的语法。

（1）学习Webpack要帮我们压缩、合并常见的静态资源文件。

（2）处理文件的前后依赖关系。（未使用之前需要比较多的script标签去引入，需要一定的顺序）

webpack内部实现了模块化的机制；把我们写ES6，如import打包转成浏览器能识别的（实现模块化的方式，和 Node 中差不多，也是创建了一个模块包装器，把所有的代码都包装起来。）

（3）webpack-dev-server：作用帮我们实现在开发阶段中的实时打包编译和刷新浏览器。

（4）html-webpack-plugin：把HTML首页自动托管到内存中，自动把打包好的bundle.js路径，以 script 标签 注入到页面中。

（5）要注意包的安装，要区别其用途，如果是为了提高效率的，那么应该装在本地开发环境中，使用--save-dev。

二十二、webpack打包其他文件

默认情况下：webpack只能默认打包处理后缀名是 .js 的文件，无法处理其他后缀名的文件。

如果想要使用 webpack 打包处理非js类型的文件，那么，需要在项目中配置合适的loader。

这一些只是为了提高开发的效率，所使用的工具，跟项目的运行没有什么关系。



在配置规则中，如果只需要调用一个loader，则可以把数组简化成一个字符串。

test：匹配的意思

loader：加载器

1、webpack处理css文件

如果想要处理.css后缀名的文件，需要安装并配置两个合适的loader（style-loader和css-loader）。

（1）先安装包，style-loader，css-loader来处理css的第三方loader加载器

npm install style-loader css-loader -save-dev

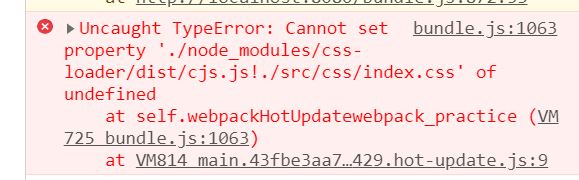
（2）在模块的入口文件main.js中导入css

import myStyle from '**./css/index.css**'

（3）在webpack.config.js中配置非js文件与loader

|  |
| --- |
| module: { // 用来配置非js文件对应的loader  rules: [ // 非js文件和loader之间的对应关系，test后边的是正则表达式，use的处理方式的从右到左的，它会把右边的结果给到左边，依次给过去  {test: /\.css$/, use: ['style-loader', 'css-loader']},  ],  } |

修改css偶尔会出现错误，暂时无解？？？

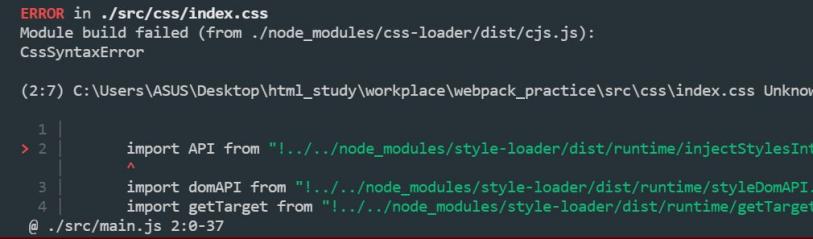


（4）处理css时的注意事项

在webpack中，把所有的静态资源都想象成了一个模块，不管是js文件，还是样式表，图片，字体文件，万物都看成了一个模块，都能够帮你去处理。

非js文件无法向外暴露对象，它们没有这种语法，像css，没有js语法，所以接收了也是空对象（测试后输出了undefined），可以不用变量去接收。

'**style-loader**', '**css-loader**'的引用顺序是需要按照顺序的，style-loader一定要放到第一位。如果颠倒位置则编译出错。



1. 第三方loader和webpack的调用关系

注意：webpack中默认只能处理js文件。

（1）当webpack发现导入文件main.js有一些非js文件需要被处理的时候，就处理不了。webpack会检查“webpack.config.js”中，有没有配置这种文件处理的loader。

（2）如果没有找到对应的 loader 规则，则直接报错：要找合适的loader去处理该文件。You may need an appropriate loader to handle this file type

（3）如果找到了匹配规则，则webpack把这种文件，直接交付给对应的loader去进行处理，如果有多个loader，则load的调用顺序是从后往前（从右往左）。

（4）后面的loader会优先处理这种文件，然后把处理的结果，返回给前面的loader，做进一步的处理，如果最后的一个loader处理完毕了，它需要把结果直接给webpack，让webpack把最后的结果直接打包到bundle.js中。

（5）也就是webpack只关心非js文件的处理结果，并拿到结果直接打包到bundle.js，它并不关心第三方loader的处理过程。

3、webpack中处理less文件

（1）处理less后缀名的文件，需要安装less-loader、less（它是less-loader内置依赖，不需要显示的配置到module）

npm install less-loader --save-dev

npm install less --save-dev

（2）配置处理less的匹配规则，还要使用style-loader，css-loader，因为你less处理完毕后，相当于是一个css，就需要继续处理。

**{test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader']},**

4、待处理，webpack中处理scss文件（注：sass 12.x版本需要装的东西比较多，这里暂时不装，用视频的）

这里跟less选一种就可以，一般不会都使用的。**scss装的包是sass。注意了**

（1）处理scss后缀名的文件，需要安装**sass**-loader、node-**sass**（它是sass-loader内置依赖，不需要显示的配置到module）

npm install sass-loader@6.0.6 --save-dev

npm install node-sass@4.7.2 --save-dev

（2）配置处理scss的匹配规则，还要使用style-loader，css-loader，因为你scss处理完毕后，相当于是一个css，就需要继续处理。

5、在webpack中打包处理样式表的图片路径

（1）处理样式表中的图片路径，需要安装url-loader，file-loader。（file-loader是url-loader的内置依赖项）

npm install url-loader --save-dev

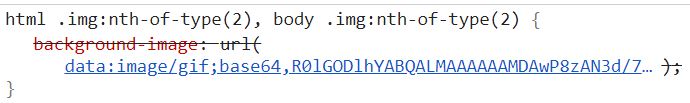
npm install file-loader --save-dev

（2）配置规则

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader']},

这个的正则表达式不能少括号，如果不加括号，就变成了.png，gif了，这样不严谨。

|  |
| --- |
| <div class="img" title="png"></div>  <div class="img" title="gif"></div>  <div class="img" title="webp"></div>  <div class="img" title="jpg"></div>  <div class="img" title="jpeg"></div>  Index.less，注意，如果使用的background，那么background-size一定要写在后面，否则会被覆盖了，无法达到效果。  .img {  display: inline-block;  width: 100px;  height: 100px;  background-size: contain;  background-image: url('../images/kmr2.jpg');  }  .img:nth-of-type(2) {  background-image: url('../images/tianshi.gif');  }  .img:nth-of-type(3) {  background-image: url('../images/small.webp');  }  .img:nth-of-type(4) {  background-image: url('../images/kmr2.jpg');  }  .img:nth-of-type(5) {  background-image: url('../images/kmr.jpeg');  } |



webpack会默认把图片给转成base64的编码，提高效率

（3）url-loader中limit参数的使用

可以使用 ? 给url-loader传递参数，其中，有一个固定的参数，叫做limit，表示图片的大小，需要给定一个数值，它是图片的大小，单位是字节（Byte）。

如果指定了limit参数，当图片的大小，**小于或者等于**给参数值的时候，才会转化为base64格式的图片，否则，就不转换。

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader**?limit=23531**']},



（4）url-loader自动生成图片文件名称

图片的命名之所以要重命名成一串数字字符，是为了防止重命名。

我们物理磁盘不同文件夹下分别存储了名称一样的图片文件。在webpack调试中，它会被url-loader解析到内存中，此时两张图片都是相对放在内存根路径下，就是他们在同个目录下，此时如果设置了不重命名，那么两张图片名字一样，首先加载了第一张图片，然后加载第二张，发现他们俩名字一样，就直接把第一张覆盖了。

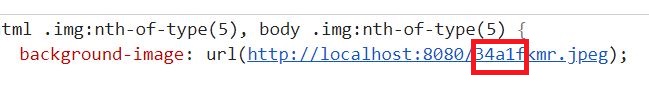
{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader?limit=2353&name=[hash:5][name].[ext]']},

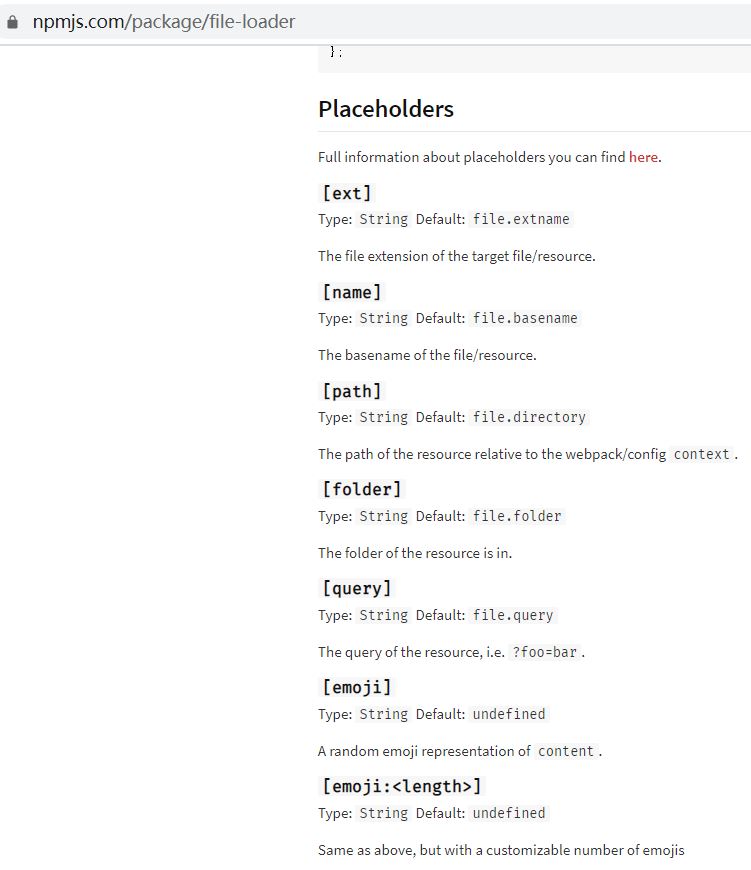
图片保持原来的名字，前提是不能让它转化成base64编码。

[name].[ext]，name表示原来的名字，ext表示原来的后缀名。这种语法是file-loader提供的。可以去npm官网或者github都可以看到其用法。

hash值能够保证图片的唯一性，如果有1像素的变化，它的hash值都会变化。

先得到一个完整的hash（哈希值）然后取前几位就可以，在加上原来的名字就不会重复了。





（5）修改一下图片在内存中的路径，默认是在内存根目录，可以修改为在根目录下的xxx目录下。调试过程不用这么搞，在webpack发布时候可以搞。

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader?limit=2353&name=**images/**[hash:5][name].[ext]']},

（6）也能处理样式中的文件路径，如字体图标路径

webpack默认只能帮我们打包并处理一小部分的ES6新特性，但是无法处理所有的ES6以及ES7+新特性，webpack 5.40已经是支持class中的static，视频的3.6就不行。

二十三、配置babel-loader解析ES6+高级语法

1、需要安装两套相关的包：（webpack5.40如果没问题就不装了）

npm install babel-core babel-loader babel-plugin-transform-runtime --save-dev

npm install babel-preset-env babel-preset-stage-0 --save-dev

以babel-preset-开头的包，叫做babel语法，以babel-plugin开头就做babel插件。

2、打开webpack.config.js配置文件，添加babel的loader配置项：

{test: /\.js$/, use: ['babel-loader'], **exclude: /node\_modules/**},

只要转换自己的代码就行了，node\_modules目录下的包中的js文件都是直接使用的，并不用进行转换。这样做没有意义，所以将其直接忽略 ，如果不忽略，那么会莫名的报错，项目运行不起来。

3、在项目的根目录中，创建一个.babelrc的配置文件，将来，babel-loader在执行的时候，会读取并使用这个配置文件：它是json格式语法。

把我们刚才装的所有的包，去除前缀名称，剩下的名字，把它配置到数组中去。如transform-runtime

{

"plugins": ["transform-runtime"],

"presets": ["env", "stage-0"]

}

4、关于babel配置的说明（chrome基本上是支持ES6了）

比如遇到ES6的static

babel-loader，它只负责提供场地给你转换，它不知道高级ES语法对应哪些低级的ES语法，它通过 .babelrc配置文件找到他们俩来处理这个。

babel-presets-env和babel-presets-stage-0是转换高级语法到低级语法，类似于字典库，它们分别有着不同对应语法。

二十四、ES6

1、export default与export

（1）export default

|  |
| --- |
| 1、export default 是和 import xxx from “xxx.js”搭配使用的。  2、在模块中，使用export default向外暴露成员，只能唯一的一个export default。  3、使用export default向外暴露的成员对象，可以使用任何合法的变量名来接收。  4、export default 和export可以同时使用。  5、使用import xxx from “xxx.js”这种形式导入的得到的变量，默认只能拿到export default导出的成员对象，拿不到export导出的成员对象。 |

1. export

|  |
| --- |
| 1、使用export导出的成员，只能使用 import {成员名} from “xxx.js”，类似解构来进行接收  ，这种 { } 导入成员的形式，叫做按需导入。  2、export导出成员，可以导出多次，没有次数限制。  3、它导出的成员，必须按照导出的名称来接收，如果想要起别名，那么需要在接收名称后边使用 as 关键字，如 import { d as dd } from “xxx.js”。   1. import { d,c } from “xxx.js”它的接收顺序可以任意调整。   注意：export 变量；是不可以的 |
| // 使用ES6  // 要跟node的区别，暂时不考虑node情况，容易混乱  export default {  name: 'Tom',  }  export var a1 = 'a1';  export var a2 = 'a2';    import obj**, {a1, a2}** from './es6\_study/exportDefault.js'    console.log(obj);  console.log(a1);  console.log(a2); |

2、class知识

从ES6开始，{ xx }，大括号就是代表一个块作用域。之前是function有作用域

使用class定义的，叫做类。

（1）什么叫做面向对象

编程的发展历史，面向过程（函数形式编程），接下来面向对象（以对象的形式来组织代码）。

就是把一些功能性的代码，封装到具体的类中，如果需要什么样的功能，就new什么样的类，这样能提高代码的复用性，提高了开发的效率和协作开发的体验。

（2）constructor构造函数的使用

它就类似于之前的构造函数。每当new的时候，必然会优先调用类的constructor。

|  |
| --- |
| **// 没有使用class之前**  function Person(name, age) {  this.name = name;  this.age = age;  }  var p = new Person('Tom', 22);  console.log(p);  **// 使用ES6中的类**  **class Per {**  **// constructor构造函数类似于Person**  constructor(name, age) {  this.name = name;  this.age = age;  console.log('constructor gogogo');  }  }  **var pClass = new Per('Jerry', 23);**  console.log(pClass);  1 |

（3）static静态属性和静态方法

在类中，不用使用new，能直接通过类名点出来的属性（方法），叫做静态属性（方法）。实例是无法使用的。

类的静态属性相当于原生js构造函数的属性。

|  |
| --- |
| // 没有使用class之前  function Person(name, age) {  this.name = name;  this.age = age;  }  Person.hobbies = 'eat';  Person.go = function () {  console.log('gogogo');  }  var p = new Person('Tom', 24);  console.log(p);  // 调用静态方法  console.log(Person.hobbies);  Person.go();  **// ES6 Class**  class Per {  // constructor构造函数类似于Person  constructor(name, age) {  // 通过new之后得到实例对象，this.name跟this.age就叫做实例属性  this.name = name;  this.age = age;  console.log('constructor gogogo');  }  **static hobbies = 'eat class';**  **static go() {**  **console.log('gogogo class');**  **}**  }  var pClass = new Per('Jerry', 24);  console.log(pClass);  // 调用静态方法  **console.log(Per.hobbies);**  **Per.go();**  1 |

（4）实例的属性和方法

通过new出来实例对象，它点出来的属性（方法）叫实例属性（方法）。

类的实例方法，相当于是原生js原型对象的方法。注：是公共的方法，会抽离出来，不会在constructor中，这样浪费内存。

|  |
| --- |
| // 没有使用class之前  function Person(name, age) {  this.name = name;  this.age = age;  }  Person.**prototype**.go = function () {  console.log('gogogo');  }  var p = new Person('Tom', 24);  console.log(p);  p.go();  // ES6 Class  class Per {  // constructor构造函数类似于Person  constructor(name, age) {  // 通过new之后得到实例对象，this.name跟this.age就叫做实例属性  this.name = name;  this.age = age;  console.log('constructor gogogo');  }  static hobbies = 'eat class';  static go() {  console.log('gogogo class');  }  // 实例方法  move() {  console.log('moving class');  }  }  var pClass = new Per('Jerry', 24);  console.log(pClass);  pClass.move();  1 |

class实现面向对象

在class的 { } 作用域中，只能写constructor函数，静态方法，静态属性，实例方法

在class类中，必须有一个constructor，如果没有显示定义，则系统会默认提供一个看不见的constructor。

如果用户显示定义了一个constructor，则会把系统默认的constructor给覆盖了。

new实例对象的时候，一定要有构造函数constructor才行，假如系统没有提供默认的，那么就创建不了。

（5）extends继承（它的实现原理就是原型继承和借用父类构造函数）

super这个函数，为父类的构造函数（原生js中.call借用父类构造函数）。

如果子类通过extends关键字，实现了继承，那么子类的构造函数中，必须先调用一下super这个父类的构造函数，才能使用this。

|  |
| --- |
| class Person {  // constructor构造函数类似于Person  constructor(name, age) {  // 通过new之后得到实例对象，this.name跟this.age就叫做实例属性  this.name = name;  this.age = age;  }  static hobbies = 'eat class';  static go() {  console.log('gogogo class');  }  move() {  console.log('moving class');  }  }    class Student **extends** Person {  constructor(name, age, num) {  **// 这里相当于原生js中借用父类构造函数思想**  **super(name, age);**  this.num = num;  }  study() {  console.log('studying');  }  }  var s1 = new Student('Tom', 22, 100);  console.log(s1);  s1.move();  这个move是在父类的原型上，子类的原型指向了父类的一个实例，就能访问到move  父类的静态属性，就跟子类什么关系了  1 |

3、总结ES6特性，常用的

let、const

箭头函数

Promise

解构赋值

默认形参

函数扩展 function test(...arg)、var arr = [1,2,3]、test(...arr)

模板字符串 `$(name) = hello`

字符串的扩展 .startsWith、.endsWith、.padStart

async await ES7的语法

ES6中的导入和导出，import xxx from “xxx.js”、export default xx 、 export

定义对象的快捷方式

var a = 10

{

a,

show() {

......

}

}

class关键字

4、Fetch API（作用就是来发起Ajax请求的，目的是为了替代传统的XHR对象）

要看请求头的具体信息的话，可以通过火狐浏览器的F12中的网络中，点击其请求头，有相关MDN说明。

|  |
| --- |
| fetch('http://www.liulongbin.top:3005/api/getprodlist').then(response => {  console.log(response);  **// 通过fetch请求的结果，如果成功了，则第一个.then中拿不到最终的数据,数据没有被解析**  **// 需要调用respon.json()得到一个新的Promise对象。**  return response.json();  }).then(data => {  console.log(data);  })  1  它本身（ajax）是不支持跨域的，在服务器去使用CROS，去处理跨域。 |

5、Promise用法，ES6引入的，专门处理异步编程

前后端的交互：前端发送请求调用后端的接口，从而获取特定的数据，涉及到了异步的操作。

在开发中，不会让我们去封装类似与这样的promise方法，只需要调用别人封装好的库，所提供的API方法，人家方法的返回值，就是promise实例对象

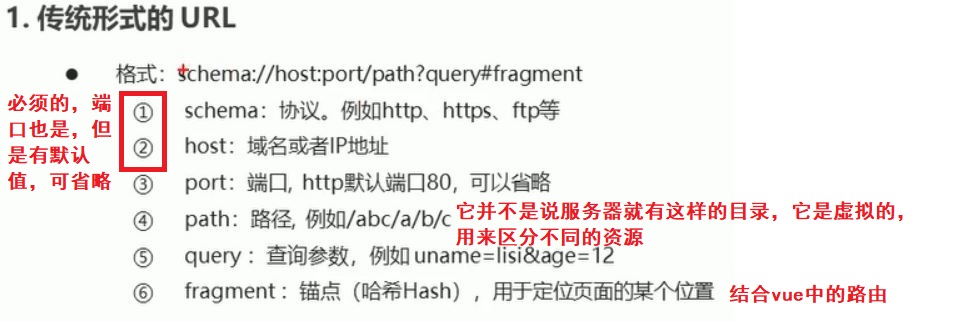
（1）接口的异步调用方式：

原生ajax、

基于jQuery（侧重于DOM操作）的ajax、

fetch（ajax升级版）、

axios（第三方库）



异步调用：定时任务，ajax，事件函数

多次异步调用的结果顺序是不确定的

异步调用的结果如果存在依赖需要嵌套

|  |
| --- |
| $.ajax({  url: 'http://www.myWebServer.com/index.php?username=Jerry&password=123',  success: function (data) {  console.log(data);  $.ajax({  url: 'http://www.myWebServer.com/index.php?username=Tom&password=123',  success: function (data) {  console.log(data);  },  });  },  });  这样子就能保证输出Jerry，然后再是Tom。  但是如果这样写，代码在回调函数中，一层回调函数嵌套一层回调函数，这种代码的风格称为**回调地狱** |
| <?php  //简单跨域处理  header("Access-Control-Allow-Origin:\*");  $username = $\_GET['username'];  $password = $\_GET['password'];  // $username = $\_POST['username'];  // $password = $\_POST['password'];  $arr = array('username'=>$username, 'password'=>$password);  echo json\_encode($arr);  ?> |

（2）Promise概述

Promise是异步编程的一种解决方案，从语法上讲，promise是一个函数对象（是一个函数），从它可以获取异步操作的消息。

1. 它的好处，解决回调地狱：

可以避免多层异步调用嵌套问题（回调地狱）。

Promise函数对象提供了简洁的api，使得控制异步操作更加容易。

console.log(typeof Promise);

console.dir(Promise);



js是单线程执行的，实际上success中的代码还是没有执行的，

（3）Promise基本用法

实例化**promise**对象，构造函数中传递函数，在该函数中用于处理**异步**任务。

当特定的条件满足时候，会触发**resolve**和**reject**两个函数参数用于处理成功和失败的两种情况（把数据丢给resolve或者reject），然后通过p.**then**获取resolve和reject中的数据结果。也可以只写成功的情况就ok

相当于把回调函数给then去处理。

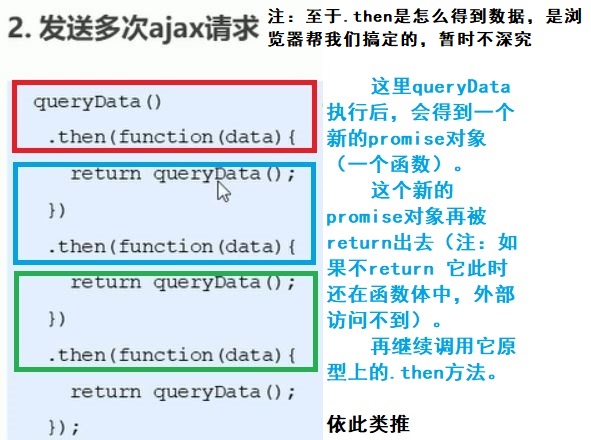


|  |
| --- |
| var p = new Promise(function(resolve, reject){  var flag = false;  // console.log('promise');这个参数的代码会执行一遍  setTimeout(function () {  flag = true;  if (flag) {  resolve('normal');  } else {  reject('nonono');  }  }, 1000);  });  p.then(function (ret) {  console.log(ret);  }, function (ret) {  console.log(ret);  }); |

（4）基于promise处理ajax请求

它在代码的层面上就成了线性的了，没有回调地狱的问题。并且保证了顺序

|  |
| --- |
| <script type="text/javascript">  function queryData(url) {  var p = new Promise(function (resolve, reject) {  var xhr = new XMLHttpRequest();  xhr.open('get', encodeURI(url));  xhr.send(null);  xhr.onreadystatechange = function () {  if (xhr.readyState === 4) {  if (xhr.status === 200) {  resolve(xhr.responseText);  } else {  reject('server error 500');  }  }  }  });  return p;  }  var url1 = 'http://www.myWebServer.com/index.php?username=Tom&password=123';  var url2 = 'http://www.myWebServer.com/index.php?username=Jerry&password=456';  queryData(url1).then(function(data) {  console.log(JSON.parse(data));  **return** queryData(url2);//不要忘记return  }).then(function (data) {  console.log(JSON.parse(data));  });  </script>  1  当请求的页面，服务器没有时，失败了。  2 |



|  |
| --- |
| Post方式，要设置请求头，和传入请求的参数  function postData(url, params) {  var p = new Promise(function (resolve, reject) {  var xhr = new XMLHttpRequest();  xhr.open('post', url);  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');  xhr.send(params);  xhr.onreadystatechange = function () {  if (xhr.readyState === 4) {  if (xhr.status === 200) {  resolve(xhr.responseText);  } else {  reject('server wrong');  }  }  }  });  return p;  }  var url = 'http://www.myWebServer.com/index.php';  var params = 'username=Tom&password=123';// 不要写成对象了  postData(url, params).then(function (data) {  console.log(JSON.parse(data));  }); |

（5）then参数中的函数返回值

1. 返回promise实例对象

返回的该实例对象会调用下一个then。

2. 返回普通值，此时它会产生一个默认的promise对象去帮你调用下一个then，保证可以继续链式操作

返回的普通值会直接传递给下一个then，通过then参数中函数的参数接收该值。

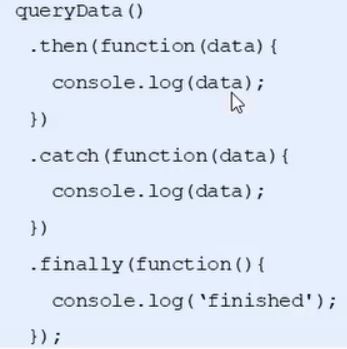
|  |
| --- |
| var p = new Promise(function (resolve, reject) {  setTimeout(function () {  resolve('hello Tom');  }, 1000);  });  p.then(function (data) {  console.log(data);  return 'a common value';  }).then(function (data) {  console.log(data);  });  1 |

（6）常用的实例方法API（原型上的方法）

p.then()得到异步任务的正确结果。

p.catch()获取异常信息。跟then中的第二个参数是效果一致，如果它们同时存在的话，那么catch就不会被调用了。

p.finally()成功与否都会执行。可以用来做提示信息，或者是销毁资源。



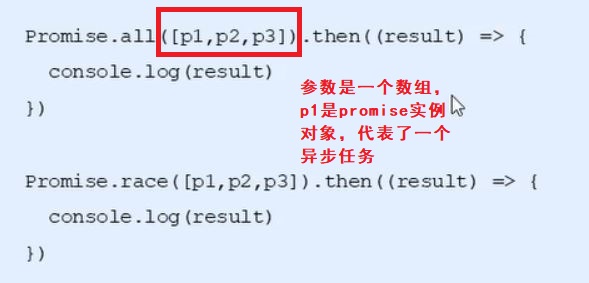
|  |
| --- |
| var p = new Promise(function (resolve, reject) {  setTimeout(function () {  // resolve('hello Tom');  reject('something Wrong');  }, 1000);  });  p.then(function (data) {  console.log(data);  })  .catch(function (data) {  console.log(data);  })  .finally(function (data) {  console.log('finally');  });  等价于  p.then(function (data) {  console.log(data);  }, function (data) {  console.log(data);  })  .finally(function (data) {  console.log('finally');  }); |

（7）常用的Promise方法（Promise函数对象的属性）

Promise.all()并发处理多个异步任务，所有任务都执行完成才能得到结果。

Promise.race()并发处理多个异步任务，只要有一个完成就能得到结果。

|  |
| --- |
| var p1 = new Promise(function (resolve, reject) {  setTimeout(function () {  resolve('hello p1');  }, 1000);  });  var p2 = new Promise(function (resolve, reject) {  setTimeout(function () {  resolve('hello p2');  }, 2000);  });  Promise.all([p1, p2]).then(data=> {  console.log(data);  });  Promise.race([p1, p2]).then(data => {  console.log(data);  });  1  race下输出p1，然后在all按顺序输出所有 |



6、async和await（ES7），注意在node环境catch参数值可能会不一样的

如果某个方法的返回值是 promise 实例对象，那么就可以用 await 关键字来修饰promise实例。

如果某个方法内部，用到了await关键字，那么这个方法，必须被修饰为 async 异步方法。

await只能用在被async修饰的方法中。单一的话会报错。

Uncaught SyntaxError: await is only valid in async function

await结合async使用，就能让异步的方法，按照书写的顺序的去执行。代码风格也比较清晰，能使用这种方式就使用。（至于它内部是怎么处理的，暂时不关心）

当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体后面的语句。

（1）作用：优化promise的调用

|  |
| --- |
| 用来模拟异步操作。  function testAwait(info, time) {  return new Promise(function (resolve, reject) {  setTimeout(function () {  resolve(info);  }, time);  });  }  testAwait('p1', 500).then(function (data) {  console.log(data);  return testAwait('p2', 1000);  })  .then(function (data) {  console.log(data);  }); |
| function testAwait(info, time) {  return new Promise(function (resolve, reject) {  setTimeout(function () {  resolve(info);  }, time);  });  }  async function test () {  const data = await testAwait('p1', 1000);  console.log(data);  const data1 = await testAwait('p2', 500);  console.log(data1);  }  test();  按顺序输出 p1、p2 |

（2）使用catch捕获错误

const data = await postData(url, params).catch(function (data) {

console.log(data);

});

console.log(data); //如果catch中没有返回值的话，那么data是undefined。

（3）async修饰的方法并不是纯粹的异步方法

async修饰的方法中，遇到第一个await之前，所有的代码都是同步执行的（从下往下，一行一行执行。）。遇到await后，就把代码交给异步去处理。

|  |
| --- |
| async function test () {  console.log(3);  const data = await testAwait('p1', 1000);  console.log(data);  const data1 = await testAwait('p2', 500);  console.log(data1);  console.log(4);  }  console.log(1);  test();  console.log(2);  1 |

（4）async返回值

async函数返回一个Promise对象。

async函数内部return语句返回的值，会成为then方法回调函数的参数。

|  |
| --- |
| async function test () {              return 'hello world'          }          const result = test();          console.log(result);          result.then(function (data) {              console.log(data);          });  1 |

补充点

forEach和some循环。

数组的forEach循环会遍历所有的元素，return也不能停止。

some循环，可以在里边进行判断，它适合于，判断数组是否拥有某个元素时候。需要return xxx;

二十五、webpack的发布策略（开发完了，要上线）

webpack打包的bundle.js默认是使用了严格模式。

1、开发阶段配置webpack，提高开发效率

（1）创建webpack.config.js的配置文件

entry 指定项目的入口

output 输出相关的配置

plugins 和插件有关

module 和第三方loader相关

rules 第三方loader的匹配规则

（2）安装了工具webpack-dev-server、html-webpack-plugin

（3）处理第三方文件：安装并配置第三方的loader，【需要安装的包】

style-loader css-loader 【style-loader css-loader】

style-loader css-loader less-loader 【less-loader less】

style-loader css-loader sass-loader 【sass-loader node-sass】

url-loader 【url-loader file-loader】

babel-loader 同时要添加exclude排除项 【babel-core babel-loader babel-plugin-transform-runtime babel-preset-env babel-preset-stage-0】

2、在实际开发中，一般会有两套项目方案：

一套是开发期间的项目，包含了测试文件、测试数据、开发工具、测试工具等相关配置，有利于项目的开发和测试，但是这些文件仅用于开发，发布项目时候需要剔除；

一套是部署期间的项目，剔除了那些客户用不到的测试数据测试工具和文件，比较纯净，减少了项目发布后的体积，有利于安装和部署！

3、为了满足发布策略，需要新建一个配置文件，命名为webpack.publish.config.js，将webpack.config.js的配置拷贝过去，剔除一些开发配置项即可。

（1）将devServer节点删掉（根据你webpack-dev-server，有的是用命令行就没有devServer的配置）

（2）修改url-loader，将图片放入统一的images文件夹之下：

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader?limit=2353&name=**image**s/[hash:5][name].[ext]']},

（3）在package.json中的script节点下新增publish命令，通过--config指定webpack启动时要读取的配置文件：

   "publish": "webpack **--config** webpack.publish.config.js"

4、clean-webpack-plugin，每次重新构建时候删除dist目录

在webpack.publish.config.js中配置，发布的时候，把dist目录整个删了重新建，保证文件最新的。

这些webpack插件直接在npm官网看下用法就行，有时候出npm的包可能会出问题，要重新安装

（1）运行npm install clean-webpack-plugin --save-dev

（2）在**webpack.publish.config.js**中进行配置

const { CleanWebpackPlugin } = require('clean-webpack-plugin');

官方例子的配置

module.exports = {

plugins: [

.......

new CleanWebpackPlugin()

],

};

5、no分离第三方包

引用的jquery、vue、art-template，这些包，如果都打包进去bundle.js，那么这个bundle.js的体积会比较大，可以把他们拆出来，虽然一次性打包进bundle.js，只发出一次请求，但是分出来可能会快点（待确定？？？）

第三方包的名称要根据你npm 安装时候的名称。

The CommonsChunkPlugin has been removed in webpack v4 legato. To learn how chunks are treated in the latest version, check out the [SplitChunksPlugin](https://webpack.docschina.org/plugins/split-chunks-plugin/).

这个分离包的，还搞不懂？？？

压缩优化javascript

6、装mini-css-extract-plugin，抽离样式表到单独文件中

（1）运行npm install mini-css-extract-plugin --save-dev

自己跟着文档配置的，初步能达到效果

（2）webpack-publish-config.js中的配置

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

plugins: [

new MiniCssExtractPlugin({

**filename: 'css/style.css',**

})

],

module: { // 用来配置非js文件对应的loader

rules: [ // 非js文件和loader之间的对应关系

{test: /\.css$/, use: [

{

loader: MiniCssExtractPlugin.loader,

options: {

publicPath: '../',

},

},

'css-loader',

],

},

{test: /\.less$/, use: [

{

loader: MiniCssExtractPlugin.loader,

**options: {**

**publicPath: '../',**

**},**

},

'css-loader',

'less-loader',

],

}

7、压缩抽取出来的CSS文件

最原始大小： asset css/style.css 752 bytes [emitted] (name: main)

压缩后：asset css/style.css 622 bytes [emitted] (name: main)

压缩就是去除空格，所有的样式都在一行上。

（1）运行npm install optimize-css-assets-webpack-plugin --save-dev

（2）在配置文件头部导入插件：

var OptimizeCssAssetsPlugin = require('**optimize-css-assets-webpack-plugin**');

（3）在plugins节点下新增插件：

plugins: [

......

**new OptimizeCssAssetsPlugin(),**

],

8、优化压缩HTML文件

直接把属性节点值的双引号直接是去掉是没有问题，浏览器有一定容错机制，会帮我们去处理。

在plugins节点下的htmlWebpackPlugin插件中，添加minify子节点：

new htmlWebpackPlugin({

// 把指定的html页面，作为模板，复制一份托管到内存中去

template: path.join(\_\_dirname, './src/index.html'),

// 指定，将来在内存中复制出来的页面，名称叫做index.html

filename: 'index.html',

**minify: {**

**removeComments: true, // 移除注释**

**collapseWhitespace: true, // 合并空格**

**removeAttributeQuotes: true, // 移除双引号**

**}**

}),

这样发布策略基本可以了，把dist文件夹拉出去就行了。 一般都是高级的工程师去发布。

二十六、Vue render函数

Vue规定了，render必须是一个函数，同时，render的参数列表中，有个形参，叫做createElements，将来，Vue在调用的render的时候，会给我们这个传递这个形参。

注意：Vue给我们传递的这个createElements是一个方法，接收一个要渲染的组件。

render函数会把之前的内容清除掉，然后页面上只能显示这个被render的组件，不能有其他的内容和这个组件并存。（视频讲得似乎有简单？？？需要后续把它补上，一些是vue2.4没有的，如插槽，视频就没有讲）

|  |
| --- |
| <body>  <div class="box">  hello,是会被清除了  </div>  <script type="text/javascript" src="./lib/vue.js"></script>    <script type="text/javascript">  **const com1 = {**  **template: '<h3>我是组件使用render渲染来的</h3>'**  **}**  var vm = new Vue({  el: '.box',  /\* render: function (createElements) {  return createElements(com1);  } \*/  /\* render: (createElements) => {  return createElements(com1);  } \*/  /\* render: createElements => {  return createElements(com1);  } \*/  // 简化到最后的版本  **render: createElements => createElements(com1)**  });  </script>  </body> |

1、特点：可以在页面指定的容器中渲染一个组件（不用注册），和传统的渲染组件的方式相比，render只能渲染这唯一的组件，其他的内容都会被覆盖掉。

即：在vm实例中，指定了 render 函数来渲染组件，那么 el 区域，就会被 render 中的渲染的组件替换掉全部。

使用render渲染出来的组件叫做根组件（不管浏览器中的页面如何切换，根组件永远都会在页面上显示），可以在它里边加其他的子组件。

二十七、Vue在webpack中开发（SPA）

快速装包，npm install

1、什么是SPA

SPA英文全称是Single Page Application，单页面应用程序。

通俗的理解是：只有一个web页面的网站，网站的所有功能都在这个唯一的页面上进行展示与切换。

（1）特点：

只有一个页面。

浏览器一开始就请求这个页面，必须加载对应的HTML、CSS、JavaScript。

用户所有操作，都在这唯一的页面上完成。

页面数据都是用Ajax请求回来的。

（2）好处：

实现了前后端分离开发，各司其职，提高了开发效率

用户体验好、快、内容的改变不需要重新刷新加载整个页面

（3）缺点：

对SEO不是很友好，因为页面数据是Ajax渲染出来的（ajax返回来的数据不会被搜索引擎抓取到，百度它搜索机制通过爬虫，它只是爬取html，并不会去爬取js代码）；可以通过（SSR）（Server Side Render）服务器端渲染对SEO进行优化。 。

刚开始加载的速度可能比较慢，项目开发完毕之后，可以单独对首屏页面的加载时间做优化。

页面复杂度比较高，对程序员要求较高。

1、组件的使用方式不一样

注意：在项目入口文件中，导入的Vue，不支持之前使用定义组件的方式，只支持使用 .vue 模板文件来定义的组件。它们俩使用vue.js文件是不一样的，webpack引入的包package.json它指向的是 "main": "dist/vue.runtime.common.js"。

如果你想要在import方式中使用之前定义组件的方式，就手动去修改。有三种修改方式：

|  |
| --- |
| "main": "dist/vue.runtime.common.js",  "module": "**dist/vue.runtime.esm.js**",  在node\_modules中vue中的package.json改成这样  "module": "dist/vue.js", |
| import Vue from '../node\_modules/vue/dist/vue.js' |
| 使用别名的方式  import Vue from 'vue'  在webpack.config.js中跟entry同级添加  resolve: {  alias: {// 别名  'vue$': 'vue/dist/vue.js'  }  } |

官方不建议这样操作，在webpack中构建的Vue项目中使用之前定义组件的方式，没有把组件的思想发挥极致，应该抽离出去。就不按上边的操作去搞了

2、.vue文件的使用（单文件组件---只能在render运行？？？具体怎么使用还要学习，单文件组件有嵌套组件吗）

一个标准的.vue文件包括<template>、<script>、<style>

|  |
| --- |
| <template>  <div>  <h3>我是.vue文件中的组件</h3>  </div>  </template>  <script>  在webpack环境下，它导出的是全部的内容，看截图import的结果  export default {  name: 'WorkspaceJsonApp',  components: { },  directives: { },  data() {  return {};  },  mounted() {},  methods: {},  };  </script>    <style>  h3 {  color: orange;  }  </style>  1 |

如果要在webpack中使用vue组件，必须先安装对应的loader加载器，否则webpack处理不了。

（1）运行npm install vue --save将vue安装为运行依赖；

（2）运行npm i vue-loader@15.9.5 vue-template-compiler --save-dev 将解析转换vue的包安装为开发依赖；

vue-loader不能装最新的，它用的是vue3，匹配不了。

（3）配置webpack.config.js

const VueLoaderPlugin = require('vue-loader/lib/plugin');

plugins: [

new VueLoaderPlugin(),

],

module: { // 用来配置非js文件对应的loader

rules: [ // 非js文件和loader之间的对应关系

{test: /\.vue$/, use: ['vue-loader']},

],

},

3、vue-router结合webpack模块化使用

npm install vue-router --save(装到生产环境下)

如果在一个模块化工程中使用它，必须要通过 Vue.use() 明确地安装路由功能：

import Vue from 'vue'

import VueRouter from 'vue-router'

Vue.use(VueRouter)

如果使用全局的 script 标签，则无须如此 (就是通过script src这种方式引入

4、style标签中的scoped属性

scoped属性是为这个组件的css样式，设置作用域的。它就会给这个组件中html加上自定义的属性，它的样式就变成属性选择器

|  |
| --- |
| main.js  import Vue from 'vue'  import app from './components/App.vue'  import VueRouter from 'vue-router';  Vue.use(VueRouter);  **// 这是单文件组件**  import Login from './components/account/Login.vue'  import Reg from './components/account/Reg.vue'  **// app不是路由，所有这里不是嵌套路由,它们是单文件组件**  var router = new VueRouter({  routes: [  {path: '/', redirect: 'Login'},  {path: '/Login', component: Login},  {path: '/Reg', component: Reg},  ],  });  var vm = new Vue({  el: '.box',  render: c => c(app),  router: router,  }); |
| App.vue  <template>  <div>  <h3>我是.vue文件中的组件</h3>  <router-link to="/Login">登录</router-link>  <router-link to="/Reg">注册</router-link>  <router-view></router-view>  </div>  </template>    <script>  export default {  name: 'WorkspaceJsonApp',  components: { },  directives: { },  data() {  return { };  },  };  </script>    <style scoped>  h3 {  color: orange;  }  div {  border: 1px solid royalblue;  }  </style> |
| Login.vue  <template>      <div>          <h3>我是注册组件</h3>      </div>  </template>  <script>  </script>  <style **scoped**>      div {          border-radius: 20px;      }  </style>  12 |

5、把路由做成一个js router模块

用来显示UI叫组件，js代码的实现叫做模块。

|  |
| --- |
| console.log('ok vue');  import Vue from 'vue'  import app from './components/App.vue'  import VueRouter from 'vue-router';  Vue.use(VueRouter);  import router from './router.js'  var vm = new Vue({  el: '.box',  render: c => c(app),  router: router,  }); |
| 这里要重新引入一下，因为VueRouter  import VueRouter from 'vue-router';  // 这是单文件组件  import Login from './components/account/Login.vue'  import Reg from './components/account/Reg.vue'  // app不是路由，所有这里不是嵌套路由,它们是单文件组件  var router = new VueRouter({  routes: [  {path: '/', redirect: 'Login'},  {path: '/Login', component: Login},  {path: '/Reg', component: Reg},  ],  });  console.log('GOGOGO');  export default router;  gogogo会比ok先打印？？？ |

6、问题

（1）哪种情况下需要Vue.use()，如mint-ui，vue-router，它们都要使用，它们都是基于vue的。而mui，axios，并不用，它们离开了vue也能使用。也可以通过查看官方文档。

（2）package.json里边的main属性是index.js，但是我们写的这个入口文件是main.js。

package.json是node.js的包管理配置文件，包一般都会有自己的一个文件夹，在文件夹中，有很多的相关文件，对于一个包来说，package.json是必须的，其中有三个必须有的三个属性，main、name、version

如果我们开发的是一个项目，那么我们使用package.json的作用，主要是为了管理包工具，记录我们装了哪些包，主要用到了scripts、devDependencies、dependencies，这三个属性，main属性，几乎没有什么作用。

如果我们开发的是一个正规的包，则一定要main属性，指定一个合法的入口路径。

二十八、基于Vue.js的Mint UI--移动端的组件库

Mobile UI elements for Vue 2.0，是饿了么封装的一套东西，Element UI（PC端）也是它封装的

1、安装运行 npm install mint-ui -S

（1）导入所有MintUI组件，也可按需导入：

import MintUI from 'mint-ui';

Vue.use(MintUI);

（2）导入样式表：

import 'mint-ui/lib/style.css';

（3）使用举例，在App.vue中添加官网的示例

    <mt-button type="danger">danger</mt-button>

2、Toast的使用结合bootstrap

（1）安装bootstrap并在main.js中引入

npm install bootstrap@3.4.1 -S

import '../node\_modules/bootstrap/dist/css/bootstrap.css'

（2）使用bootstrap字体图标，并在webpack.config.js中配置字体图标路径处理

在App.vue的script标签中

// 放在脚本标签里边，这是一个独立单文件组件，不引入的话，Toast是未定义的

import { Toast } from 'mint-ui';

......

Toast({

......

iconClass: 'glyphicon glyphicon-flag flag-color'

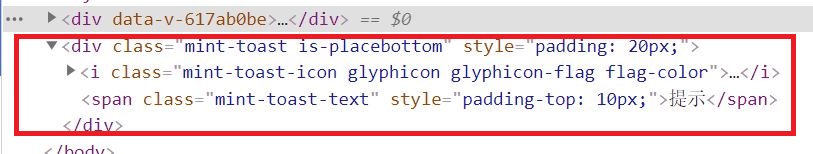
// 这里不用引入bootstrap，是因为它是一个类名，相当于是一个字符串

});

{test: /\.(eot|woff2|woff|ttf|svg)$/, use: ['url-loader']},

（3）在App.vue的style scoped中给字体图标添加样式无效问题

style scoped里边的样式只在app组件内有效，它会带有一个data-v-xxx的属性选择器，但是mint-ui它生成图标不在app组件内，所以flag-color没有找到对应的样式。



解决方式：就是重新写一个全局的样式文件，然后再进行导入，这样就ok

二十九、Vuex的使用

在vue的项目中，要处理各种各样的数据，但是，这些数据虽然多，从本质上来划分，分为两种数据

组件内部私有的数据（组件之间不会共享私有的数据）

组件之间共享的数据（父组件要共享和子组件的数据、子组件要共享给父组件的数据、兄弟组件之间传值，之前都是使用事件绑定，或者props传值。但是兄弟之间的传值就无法使用了）

1、Vuex介绍

Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到**高效更新**。

不能直接改变 store 中的状态。改变 store 中的状态的**唯一途径**就是显式地**提交 (commit) mutation(这也是为什么actions也通过提交)**。这样使得我们可以方便地跟踪每一个状态的变化。

（1）为什么需要Vuex

我们如果在组件之间共享数据，最好的方式，就是在应用程序全局，配置一个共享的数据对象，将来，任何组件想要拿到共享数据的，不必再通过，父子组件的传值，直接从全局对象上拿，这就方便多了，于是Vuex就出现了。

（2）Vuex功能

在全局环境中，为所有的组件之间共享数据，使用Vuex全局存储共享数据的话，好处是只要改变了全局的数据，所有引用了全局数据的组件，也会同步的进行更新数据。

（3）什么样的数据需要存储在Vuex中

组件之间共享的数据，才存储到vuex中，其他的就存储在自身组件的data中就行。

（4）兄弟组件间的传值

子组件的操作，调用的是父组件的方法。这样的逻辑在复杂场景下，容易混乱。

2、Vuex的使用

这个store中的state，可以想象成组件对象的data属性。

在这个state中存储数据，就是我们要全局共享的状态

将创建的共享状态对象，挂载到Vue实例中去，这样所有的组件，就可以从store中获取

（1）运行npm i vuex -S

（2）导入 vuex，安装Vuex

import Vuex from 'vuex'

Vue.use(Vuex)

（3）创建store公共状态对象，将 创建的 store 挂载到 vm 实例上

const store = new Vuex.Store({

state: { // state 中存放的，就是全局共享的数据

count: 0

}

})

new Vue({

el: '#app',

render: c => c(app),

store // 将 创建的共享状态对象，挂载到 Vue 实例中，这样，所有的组件，就可以直接从 store 中获取 全局的数据了

})

（4）this.$store.state在组件中访问全局的数据

this.$store.state.全局数据名称

3、在组件通过mapState中访问store中state上的属性

mapState 函数返回的是一个**对象**。

（1）在子组件中**computed**添加mapState 辅助函数，这样写会导致计算属性无法使用

mapState 函数返回的是一个对象。

在子组件xxx.vue中，按需导入 mapState 辅助函数：

import { mapState } from "vuex";

在子组件xxx.vue中，创建一个computed属性，通过mapState把需要的状态映射到组件的计算属性中：

computed: mapState(**[**"count"**]**)

（2）\***computed**结合 ... 展开运算符，把需要的状态映射到组件的计算属性中，这样能够让自己的计算属性和store中的属性并存。注意使用中括号。

通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性，展开运算符帮我们搞定这个。

import { mapState } from "vuex";

computed: {

newMsg: function () {

return this.msg + '111111111';

},

...mapState(**[**'count'**]**),

}

4、mutations（提交mutation）修改store中state上的值，它是一个方法。

mutations是专门用来修改state中的数据的，如果要修改state中的数据，必须调用mutations提供的方法。

它里边的第一个参数，永远都是state。它只能带另外的一个参数，在调用的时候，只能**传递唯一的一个参数（可以是简单数据类型）**。如果要传递多个参数，可以以数组或对象的形式传递进去

var store = new Vuex.Store({

state: {

count: 0,

},

mutations: {

add(state) {

state.count++;

},

}

});

（1）commit方式在子组件中调用

this.$store.commit('add');

不推荐使用下面的方式，去修改state。

this.$store.state.count++，这样使用的话，很多组件都能去改，这样有问题，不知道找谁了，通过mutations，这样大家通过一个这样的接口去使用。

（2）\*使用mapMutaions来映射方法到methods中

先导入

import { mapState, mapMutations } from 'vuex';

相当于是在methods中有add方法，它是映射过来的。

var store = new Vuex.Store({

state: {

count: 0,

},

mutations: {

add(state,step) {

state.count += step;

},

sub(state, step) {

state.count -= step;

}

}

});

注意中括号

methods: {

...mapMutations(**[**'sub'**]**),// 将 this.sub() 映射为 this.$store.commit('sub')

},

<input type="button" value="-1" @click="**sub(3)**">

5、使用getters（做下包装），它类似计算属性

getters不会修改原数据，只是把数据按照我们自己的需求，做了重新的组织和包装。

Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。就像计算属性一样，getter 的**返回值会根据它的依赖被缓存起来**，且只有当它的**依赖值**发生了改变才会被重新计算。

import { mapState, **mapGetters** } from 'vuex';

var store = new Vuex.Store({

.....

**getters**: {

infoAdd(state) {

**return** '---------'+state.count+'-------';

}

},

});

（1）通过 this.$store.getters.名称 来访问

<h3>{**{ $store.getters.**infoAdd }}</h3>

（2）\*使用mapGetters 来映射为计算属性：

export default {

computed: {

...mapState(['count']),

**...**mapGetters(**[**'infoAdd'**]**),

}

}

<h3>{{ infoAdd }}</h3>

6、actions 来提交异步的操作

有异步操作使用actions，同步就使用mutations。

在mutation中写异步的方法，能实现效果，但是在F12的Vue调试工具Vuex中捕获不到。

Vuex官方虽然说，不要在mutations中写异步（如setTimeout）的操作，但是如果不按照官方的走，在mutations中定义的异步方法，可以正常执行，但是在vue-tools工具，无法监视到数据的变化。

在actions中，可以定义一些行为来提交对应的mutations方法，它不能直接操作 state，如果想要修改state值，只能在actions中提交对应的mutations。

（1）参数context。

Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，它并不是store实例对象。

|  |
| --- |
| mutations: {  add(state,step) {  state.count += step;  },  **sub(state, step) {**  state.count -= step;  }  },  actions: {  newAdd(context, step) {  context.commit('add', step);  },  newSub(context, step) {  setTimeout(() => {  **context.commit('sub', step);**  }, 1000);  }  }  import { mapState, mapActions } from 'vuex';  methods: {  **...mapActions(['newAdd']),**  }, |

7、相关代码位置

C:\Users\ASUS\Desktop\html\_study\workplace\Vue\_practice\vue\_webpack\src\components\vuex\sub.vue

三十、Vue综合项目（涉及MUI）

固定的代码片段，跟写好样式。跟bootstrap一样，导入样式，跟html例子片段就可以，就是做移动端的东西。

git 忽略文件中 node\_modules，.vscode

在mint-ui中没有合适的底部导航栏的组件，所以使用mui。

1、底部导航栏的做法：

使用mui来做，在example找tabbar.html，看它源码使用的html结构，样式，直接引入。发现没有购物车icon图标，找icons.html开头的html文件查看，找到后引入，它需要另外的css跟font，跟着路径放到对应的位置。

（1）前端路由的设置：创建好路由对象(一般要做路由重定向，redirect)，导入组件，在路由对象中做hash跟组件的对应，设置组件的在哪里显示，router-view。最后要挂载到vm实例

（2）注意路由高亮的做法，它是可以直接设置配置的。（这里可以把它原来的class设置的类，直接去掉，这种一开始没有想到的。。。。）

先实现功能，再去抽离。

导航栏使用的a标签，建议改成router-link，使用

2、轮播图

axios获取图片，视频中的接口地址为'http://api.cms.liulongbin.top/'（试试看wamp能不能用php搞一下？？？暂时不搞，涉及的东西有点多）。

（1）axios注意事项

先在main.js中在Vue构造函数原型上注册axios.create实例对象，一个组件是一个实例，所以home.vue是能得到，在它的方法里边去使用，还有要熟悉async的用法。还不是很熟悉，还写错了。

axios.create，没有加d，不是created



（2）如果轮播图插件样式有差异的话，使用参数来进行区分，vue中可以绑定类，然后通过参数来确认是否要使用这个类样式。

3、九宫格布局

使用mui中的grid-default.html。

添加图标并设置宽高。

将背景色设置为白色。

去掉了九宫格的边框。

修改了字体大小为13px。

（1）webpack打包后 html 中的 img标签 的 src属性成了img src=【object Module】

使用webpack打包图片资源时, 使用 url-loader 可以处理less/css/styles文件中引入的图片资源。

但是当在html中直接使用img标签 引入图片时，webpack是不会识别,

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: ['url-loader?limit=2353&**esModule=false**&name=[hash:5][name].[ext]']},

4、路由切换效果问题

实现基本的动画的效果之后，需要解决一些问题（花了三天时间解决了）

1. 设置即将离开的页面运动方向为 -100%
2. 切进来的页面从下边跳上来的问题，需要设置绝对定位。
3. 当组件切换的时候，一瞬间，页面的宽度会变成正常宽度的2倍，此时需要隐藏超出屏幕宽度的部分。

为什么会超出2倍

浏览器的页面它会向右边扩大，顶部fixed会随着

fixed的问题，position: fixed，决定元素（left，top）相对与页面定位，并且height，width百分比也相对页面。

绝对定位元素不会因为父元素的overflow: hidden;而隐藏。

在PC端没有像移动端那样双击缩放的效果，PC端要想出现滚动条，那么你的内容要超出浏览器窗口的大小。

在移动端，如果在html中，meta标签中，不设置user-scalable=no，那么它具有双击缩放的效果。当内容超过浏览器窗口宽度的时候，它的高度也多出同样的比例。（暂时无解）

但是如果只超过浏览器窗口的高度，水平方向上没有滚动条。这个目前在思否看是否有答案？？？

这个现象具体的理解，看后续。

5、新闻模块：

moment，格式化时间的模块。JavaScript 日期处理类库。在官网上有各种用法。

（1）后台接口的使用

|  |  |
| --- | --- |
| **地址** | /api/getnewslist |
| **作用描述** | 主要用来获取点击首页上的“新闻资讯”后进入到的图文列表页面的数据，大家在做的时候可以使用ajax请求这个地址获取json数据后展示到页面上 |
| **请求方式** | Get |
| **传入api的参数** | 无 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（2）展示新闻列表的数据

当页面加载完毕之后，就要去获取页面数据，因此，应该在 created 钩子函数中调用获取数据的方法

在 methods 中，定义个 getNewsList方法来获取数据this.$http.get()

在获取数据的时候，为了操作方便一些，我们用到 async 和 await 特性

当数据获取回来之后，如果想要通过v-for指令循环渲染到页面上，需要把 数据挂载到 data 上。

（3）定义全局的过滤器来格式化时间字符串

经过分析，整个项目中，许多组件都需要格式化时间，因此，推荐定义为全局的时间过滤器；

定义全局过滤器的语法：Vue.filter('dateFormat', function(dataStr){})

调用过滤器的语法：<span>{{ item.add\_time | dateFormat }}</span>

这个时间过滤器，是借助与第三方的moment插件。

在设计时间过滤器的时候，为了提高用户体验度（可能有的用户想要 最全的年-月-日 时：分：秒， 有的用户只想要YYYY-MM-DD），此时，我们可以为过滤器提供一个可选的 格式字符串；

6、分离模块

import ‘xxx.js’会执行xxx.js的代码

导入全局的配置模块，在这个模块中的代码，和main.js中的 vm 实例，没有关系，只是全局的Vue配置，因此，抽离出去之后，可以让项目结构更加的清晰

7、新闻列表 到 新闻详情的 跳转

（1）添加新闻详情组件。

改造每一个 新闻列表项的 链接，为 router-link，其中， to 属性中的 id 值，需要动态拼接给每一个 router-link, 因此，需要把 to属性，改造成 :to，然后，属性值中的 统一路径前缀：/home/newsinfo/ 应该被 单引号包裹起来，后面，在拼接上 item.id 即可。

（2）添加路由规则

当实现了以上三步之后，我们想在页面中，获取路由传递过来的参数：

方式1： $route.params.id 来获取，比较麻烦，是老的方式。

方式2： 使用路由的props属性来传递参数：

改造路由规则，在对应的规则中，添加 props: true 的属性。

在 组件中，定义props: [] ，数组中定义的 名称，就是 路由规则中，对应参数的名称占位符（要保持一致）。

（3）渲染新闻详情页面

获取数据，挂载数据到 data 上，渲染页面。

（4）新闻详情接口

|  |  |
| --- | --- |
| **地址** | /api/getnew/:newid |
| **作用描述** | 根据资讯newid,获取期详细描述信息 |
| **请求方式** | Get |
| **传入api的参数** | newid:资讯id，传入url写法： /api/getnew/43 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

8、评论子组件

（1）获取评论信息后台接口

|  |  |
| --- | --- |
| **地址** | /api/getcomments/:artid?pageindex=1 |
| **作用描述** | 根据资讯id获取它的评论的分页信息 |
| **请求方式** | Get |
| **传入api的参数** | artid: 资讯id，  pageindex: 分页的页码，表示当前第几页  传入url写法： /api/getcomments/43?pageindex=1 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（2）post提交评论信息

|  |  |
| --- | --- |
| **地址** | /api/postcomment/:artid |
| **作用描述** | 给某条资讯进行评论 |
| **请求方式** | Post |
| **传入api的参数** | artid: 资讯id，  content: 评论的内容  传入url写法：/api/postcomment/43  请求报文体Body格式:  content=评论内容 |
| **返回数据格式** | Json |
| **返回数据格式样例** | {  "status":0,  "message":"评论提交成功"  } |

（3）获取评论内容

前提： 在评论组件中，需要获取到新闻的id;

通过---属性传递的形式，父组件为子组件传递新闻的Id；

定义根据新闻Id和页码，获取评论数据的方法，并在 created 钩子中，调用此方法，获取数据；

把获取到的数据，挂载到 data 上的 cmtlist 中；并渲染页面；

（4）点击加载更多的评论

为加载更多按钮，绑定点击事件；

在事件中，让 page 页码值 自增 + 1

当页码值+1之后，重新调用 获取 评论内容的方法；

如果重新调用获取更多评论内容的方法，刚获取到的那一页的数据， 会覆盖之前的评论数据；

为了防止上述的问题，我们在获取到评论数据之后，不应该直接 this.cmtlist = data.message，而是应该this.cmtlist = this.cmtlist.concat(data.message)

（5）实现发表评论的功能

在 点击发表评论按钮的时候，处理函数中，做一下非法值校验，防止评论内容为空！

如果校验通过，则调用API接口提交评论数据到后端；

当评论成功后，需要在客户端，组织出一个新的评论对象，手动 unshift 到 评论列表cmtlist中；

（6）bug

BUG：在用户发表完n条评论后，并且在未刷新页面的情况下点击加载更多按钮时，此时从服务器获取的数据在与this.cmtlist拼接后会出现评论重复显示的问题。

因为在提交评论后并未从服务器重新获取评论数据而是手动拼接了评论列表在本地进行显示。

因此需要合理设定this.page，并且在对this.cmtlist进行重新拼接前对尾部数据进行删除操作，以防止出现评论重复问题。（这个解决方案比较麻烦，服务器每次只能取十个数据，假如用户输入超过十次，那么你就得请求多次服务器数据）。

暂时不处理，不知道做成什么样子。需求不明确，每次加载，并没有说要加载多少个数据，不好做。

9、图片分享区域接口

（1）图片列表数据

|  |  |
| --- | --- |
| **地址** | /api/getimages/:cateid |
| **作用描述** | 用于获取点击首页上的“图片分享”后进入到的列表数据 |
| **请求方式** | Get |
| **传入api的参数** | cateid: 图片的类别id，传入url写法： /api/getimages/23 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（2）图片分类数据

|  |  |
| --- | --- |
| **地址** | /api/getimgcategory |
| **作用描述** | 用于获取图片列表页面上面的分类信息获取 |
| **请求方式** | Get |
| **传入api的参数** | 无 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |
|  | 注意：在后台接口中返回分类数据，不包含“全部”分类，需要前端向message数组中，使用unshift方法手动添加对象，如{ title: 全部, id: 0 } |

（3）图片分享详情中的缩略图数组（商品详情页面中轮播图的API和缩略图共用一个API）

|  |  |
| --- | --- |
| **地址** | /api/getthumimages/:imgid |
| **作用描述** | 用于获取图片详情页面上部的图片列表数据 |
| **请求方式** | Get |
| **传入api的参数** | imgid: 图片id，传入url写法： /api/getthumimages/43 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（4）图片详情

|  |  |
| --- | --- |
| **地址** | /api/getimageInfo/:imgid |
| **作用描述** | 用于获取图片详情页面中的详细描述信息 |
| **请求方式** | Get |
| **传入api的参数** | imgid: 图片id，传入url写法： /api/getimageInfo/43 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

10、缩略图vue-preview插件的使用

（1）安装npm i vue-preview -S

（2）导入import VuePreview from 'vue-preview'

Vue.use(VuePreview)

<template>

<vue-preview :slides="slide1" @close="handleClose"></vue-preview>

</template>

close如果没有用到，可以不用

<script>

export default {

data () {

return {

slide1: [

{

// 放大的图片资源位置

src: 'https://farm6.staticflickr.com/5591/15008867125\_68a8ed88cc\_b.jpg',

// 小图的资源位置，如果不写，缩略图就没有显示

msrc: 'https://farm6.staticflickr.com/5591/15008867125\_68a8ed88cc\_m.jpg',

alt: 'picture1',

title: 'Image Caption 1',

w: 600,

h: 400

},

]

}

},

methods: {

handleClose () {

console.log('close event')

}

}

}

</script>

（3） /deep/知识：无法修改vue-preview样式问题

当前组件中引用了第三方组件，需要在当前组件中局部修改第三方组件的样式。而当<style>标签有scoped属性是，它的css只作用于**当前组件**的元素。所以从我这个 .vue单组件文件 去修改 第三方组件 不会生效。

方式一：把scoped去掉,但是这样的话不仅是自己会影响别的元素,别的元素也会影响自身,所以这个方法一般不会使用。

方式二：使用深度选择器。

解决方法：使用深度选择器 /deep/

**/deep/** .my-gallery{ //deep深层作用选择器

........

}

/deep/深度选择器用于将全局的组件或者类的样式转换为局部的类从而达到在scoped内部添加样式后起作用的目的。

比如要给scoped内部的.sys\_info\_box的子类组件添加样式，而.el-card\_\_header属于全局组件，无法在局部scoped中起作用，所以可以在.el-card\_\_header前加上/deep/

方法三：给单文件组件的根元素设置类名，作为标识，然后把其他元素作为它的后代并移除scoped。这样就不会跟全局其他样式名字冲突。

11、商品展示

（1）获取商品列表数据

|  |  |
| --- | --- |
| **地址** | /api/getgoods?pageindex=number |
| **作用描述** | 获取商品列表展示页面中的所有数据 |
| **请求方式** | Get |
| **传入api的参数** | pageindex: 页码  获取第一页数据写法：/api/getgoods?pageindex=1 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（2）商品详情页相关api

（3）商品图文介绍

|  |  |
| --- | --- |
| **地址** | /api/goods/getdesc/:id |
| **作用描述** | 获取商品的图文介绍数据 |
| **请求方式** | Get |
| **传入api的参数** | Id：商品数据的id  Url ： /api/goods/getdesc/87 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（4）获取购物车页面数据

|  |  |
| --- | --- |
| **地址** | /api/goods/getshopcarlist/:ids |
| **作用描述** | 获取购物车列表数据 |
| **请求方式** | Get |
| **传入api的参数** | Ids：商品id 字符串，多个id之间用逗号分隔  Url ： /api/goods/getshopcarlist/87,88,89 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

（5）获取商品参数区和价格，标题等数据

|  |  |
| --- | --- |
| **地址** | /api/goods/getinfo/:id |
| **作用描述** | 获取商品参数区和价格，标题等数据 |
| **请求方式** | Get |
| **传入api的参数** | Id:商品主键值  /api/goods/getinfo/100 |
| **返回数据格式** | Json |
| **返回数据格式样例** |  |

12、购物车页面

购物车页面组件把id传递给数字框组件，数字框组件跟store中的数据关联在一起。

重新更新一些store时，会同时更新到本地，它在加入购物车时才生效，所以要删除原来的localstorage，再重新加入购物车。

13、使用Hbuilder软件打包成一个apk，当成app使用

14、还存在的问题

（1）css的压缩，js文件的抽离、压缩

（2）新版的mui，并不会出现，严格模式的问题

[Intervention] Unable to preventDefault inside passive event listener due to target being treated as passive. See <URL>

加上：

.mui-slider {

touch-action: pan-x;

}

正常操作就不会。

如果拉的幅度大，偶尔会出现[Intervention] Ignored attempt to cancel a touchstart event with cancelable=false, for example because scrolling is in progress and cannot be interrupted.

（3）滑动图片分享的tab栏，底部的图标无法切换，顶部的tab栏，mui它的类名内部和js冲突了。我使用的时候并没有出现问题。

还存在的问题：

Tab栏点击之后是没有反应的。测试的文件也是如此。

（4）图片懒加载看不到效果。。

15、细节问题

（1）数量框的问题，如果使用变量的话，变量的值是通过ajax异步获取到的，它在渲染传值的时候，服务器可能还没来得及反应，所以传了undefined。

（2）子组件不能动态的接收值吗

在vue看数据已经是更新的了，但是并没有同步到视图上。子组件已经在创建的时候，用js去获取这个值，然后处理。需要用watch去监控，调用对应的函数去进行处理。

如果只是简单的使用的它的值，例如：{{ xxx }}，这样子是能同步的。

如果要使用变量的值的话，就需要使用v-bind进行绑定。

（3）小球动画：

Vue不推荐使用DOM操作，是针对于双向数据绑定的，如表单，其他的一些功能，能适当使用。

小球动画和双向数据绑定没有任何关系，所以能直接使用普通的DOM操作。DOM操作也有优势，不论要操作的DOM元素是属于哪个组件，只要这个元素属于document，那么就能够直接获取到。

Element.getBoundingClientRect()

Element.getBoundingClientRect() 方法返回元素的大小及其相对于视口的位置。

如果是标准盒子模型，元素的尺寸等于width/height + padding + border-width的总和。如果box-sizing: border-box，元素的的尺寸等于 width/height。

返回的结果是包含完整元素的最小矩形，并且拥有left, top, right, bottom, x, y, width, 和 height这几个以像素为单位的只读属性用于描述整个边框。除了width 和 height 以外的属性是相对于视图窗口的左上角来计算的。

尽量仅使用 left, top, right, 和 bottom.属性是最安全的。

（4）数字框的加减传值给父组件

使用v-model实现不了，因为它数字框的加减不是通过input事件去处理的，所以无法用双向数据的方式来处理，只能通过绑定事件方式。

（5）Vuex的使用：文本框数字字符，转成数字类型

（6）对输入框的处理一般要注意去除前后的空白符（就是空格）

16、还存在的疑问

在谷歌浏览器中使用ctrl+shift+t，可以重新打开不小心关闭的页面。

Page应该是不能随便加的的，视频中说没有影响。要注意下它后台是怎么存的

文字太长的话会跟下边的文字重叠了

（1）{path: '/home/news', component: news},这个home怎么会生效

三十一、vue-cli的安装和使用（一些配置还要看视频，语法约束很多，空格，加不加逗号）

关于旧版本

Vue CLI 的包名称由 vue-cli 改成了 @vue/cli。 如果你已经全局安装了旧版本的 vue-cli (1.x 或 2.x)，你需要先通过 npm uninstall vue-cli -g 或 yarn global remove vue-cli 卸载它。

安装之后，你就可以在命令行中访问 vue 命令。你可以通过简单运行 vue，看看是否展示出了一份所有可用命令的帮助信息，来验证它是否安装成功。

你还可以用这个命令来检查其版本是否正确：

vue --version

目前装的4.4.6，4.5以上是支持vue3的

1、为什么要使用vue-cli创建项目

在终端运行一条简单的命令，就可创建出标准的vue骨架项目(复制html、vue编译、loader等等，都帮你配置好了)

不必自己手动去搭建vue项目基本结构，省时省力

不必关心webpack如何配置，只关注于项目代码的开发

2、安装vue-cli

全局安装脚手架工具

npm install -g @vue/cli

3、初始化项目：

（1）2.x版本

vue init webpack xxx（项目目录名称-------cli 2.x旧版本）

前三个选项，默认回车

接下来，它提供两个Vue版本，一个功能齐全的版本，一个缩减版

ESLint是语法校验工具，建议选，提供了一种语法规范，统一代码风格

选择语法的风格，选标准语法。

单元测试可以不用，选n

e2e黑盒测试选n

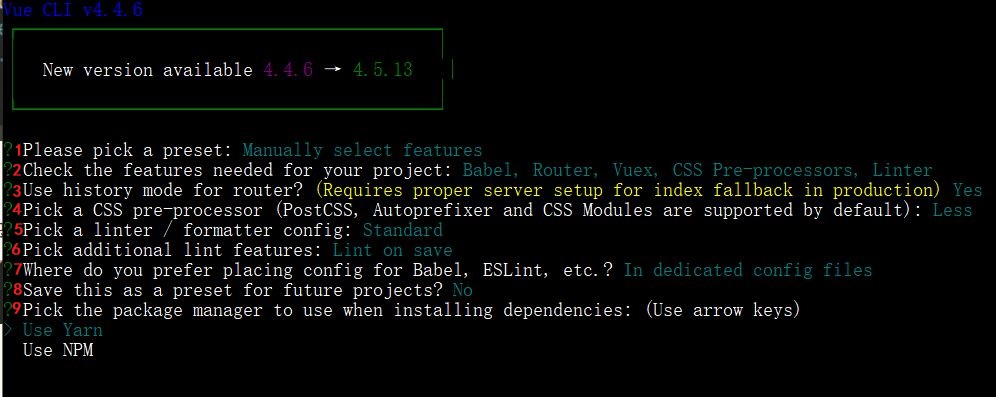
选自己装包就ok

记得手动装包。不要直接运行

（2）4.4.6本版本（注意，使用npm安装的依赖，暂时不使用yarn）

vue create xxx

注意要联网，它需要在联网



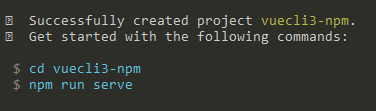
|  |
| --- |
| 上下方向键时控制上下移动的，空格键是选择对应项的，然后回车，继续执行   1. 手动选择配置，这里选择手动选择（推荐） 2. router、vuex、babel、eslint、css预处理 程序 3. 是否使用路由的history模式，看个人及项目，最好一开始就选择history模式，以后不用再更改，不过更改也不麻烦，就直接选择了history模式。   注意：推荐使用hash哈希的路由，选择**n**   1. 选择一个要用的预处理语言，之前一直使用less。 2. 选择哪个ESLint自动化代码格式化检测，分别为，只预防、airbnb配置、标准配置、最高配置，这里选择标准配置。（推荐是标准模式）   6、 选择代码检测的时机，这里选择在保存代码时检测  7、将单独配置的文件放到pakage.json还是放在单独的文件，这里选择单独的文件即可，清晰  8、是否把以上设置过的预设配置保存起来作为模板，这里不需要，每次创建项目重新选择一遍不麻烦，很快就选择好了，输入n  9、拍回车，就开始创建项目了，耐心等待，安装依赖  然后跳转到先创建的xxx文件夹下，运行npm run serve即可。 |

（3）npm卸载的时候，加名字就好，不要加版本号

4、如何更改安装时的包管理器为yarn或npm

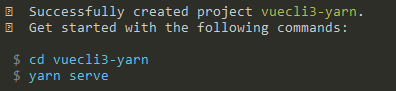
（1）npm

在执行 vue create project 后如果显示如下 npm run serve 则表示你使用的是npm创建的项目。

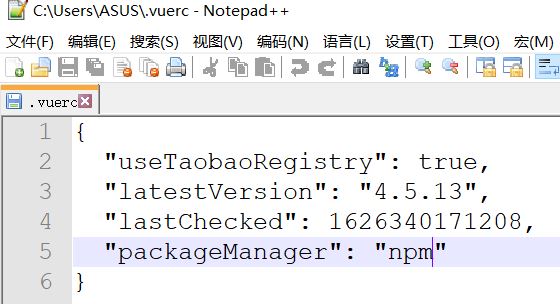


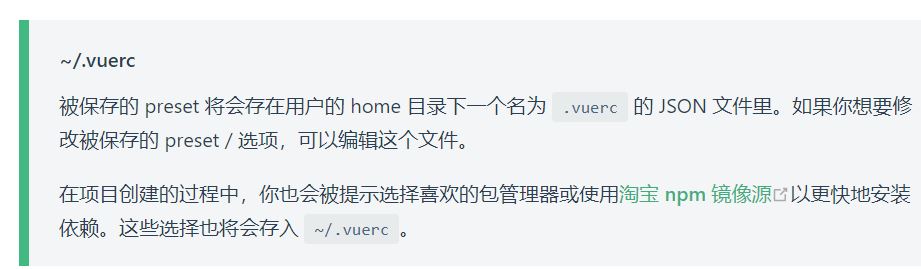
（2）yarn

如果显示如下 yarn serve 则表示此项目为yarn创建。



（3）切换包管理器





windows环境，则存在了 C:/user/administrator/ 下：

只需手动更改配置内容npm为yarn，即可更改创建项目时的包管理器了（亦可删除 .vuerc 文件重新运行 vue create xx 选择配置）

注意：.vuerc 文件是在初次使用 vue create xx 时进行提示选择配置：

IMG_256

之后就会按照第一次选择的配置进行安装，不再重复提示选择包管理器。卸载cli后.vuerc文件还会存在。

5、ESLint语法检查规范

（1）声明但是未使用的变量会报错

（2）空行不饿能连续大于等于2

（3）在行结尾处，多余的空格不允许

（4）多余的分号，不允许

（5）字符串要使用单引号，不能使用双引号

（6）在方法名和形参列表的小括号之间，必须要有一个空格

（7）在单行注释 // 之后，必须有一个空格

（8）在每一个文件的结尾处，必须要有一个空行

（9）import语句必须放到最顶部

在Vue-cli中直接会报出错误的原因

6、vscode格式化代码的配置



把空格的ESLinkd的空格报错问题给禁用了



7、脚手架项目进行自定义配置的两种方式（只能选用一种）

（1）通过package.json配置

不推荐使用这种配置方式，因为package.json主要用来管理包的配置信息，为了方便维护，推荐将Vue脚手架相关的配置，单独定义到vue.config.js配置文件中。



（2）单独的配置文件配置项目vue.config.js

在项目的根目录下创建文件vue.config.js，在该文件中进行相关的配置，从而覆盖默认配置。

// 配置脚手架,用到的是node语法

module.exports = {

devServer: {

port: 8800,

// 自动打开浏览器

open: true,

},

}

输入框的防抖

1. 什么是防抖

防抖策略，是当事件被触发后，延迟n秒后再执行回调，如果在这n秒内事件又被触发，则重新计时。

就例如王者荣耀的回城，它要在原地几秒后才回城。如果在读条内被打断了，再回城，重新要几秒后才执行。

就是当我们频繁去触发事件的时候，能保证事件就执行一次，减少不必要的回调。提高性能。

2、防抖的应用场景

用户在输入框中输入一串字符串时，可以通过防抖策略，只在输入完后，才执行查询的请求，这样可以有效减少请求的次数，节约请求资源。不能输入一个字符就请求一次

3、实现输入框的防抖

（1）防抖动的timer

（2）定义防抖函数

（3）在触发keyup事件是，立即清空timer

|  |
| --- |
| <body>  输入用户名：<input type="text" id="txt">  <script type="text/javascript">  var timer = null;  function defendBounce(name) {  timer = setTimeout(function () {  postName(name);  }, 1000);  }  function postName(name) {  var data = `username=${name}&password=123`;// 不要写成对象  var xhr = new XMLHttpRequest();  xhr.open('post', 'http://www.myWebServer.com/index.php');  xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');  xhr.send(data);  xhr.onreadystatechange = function () {  if (xhr.readyState === 4) {  if (xhr.status === 200) {  console.log(xhr.responseText);  }  }  };  }  // 一般对输入框的空格需要进行处理  var txt = document.getElementById('txt');  txt.addEventListener('keyup', function () {  // postName(this.value);  if (this.value.trim() <= 0) {  return;  }  clearTimeout(timer);  defendBounce(this.value.trim());  });  </script>  </body> |

4、缓存搜索的建议列表

例如：输入apple的时候，会得到服务器数据，此时搜索apple mac，然后又把mac删除，这时候apple，又会打出请求，此时请求是么样必要的，可以通过第一次把apple的结果用键值对形式保存下来，obj[apple] = result。在请求时候先判断是否有请求过了。

节流

1、什么是节流

节流策略，可以减少一段时间内事件的触发频率。

事件第一次被触发，它需要一定的执行时间，在这一段时间内去触发是无效的。只能等待执行完毕。防止被无限的触发。

2、节流的应用场景

（1）鼠标连续不断的触发某事件，（如点击），只在单位时间内触发一次。

（2）懒加载时要监听计算滚动条的位置，但不必每次滑动都触发，可以降低计算的频率，而不必要去浪费CPU资源。

3、节流阀（标志位）

节流阀为空，表示可以执行下次操作，不为空，表示不能执行下次操作。

当前操作执行完毕后，必须将节流阀**重置**为空，表示可以执行下次操作了。

每次执行操作之前，必须先判断节流阀是否为空。

4、鼠标跟随效果

|  |
| --- |
| function getScroll() {  var scrollTop = document.body.scrollTop || document.documentElement.scrollTop;  var scrollLeft = document.body.scrollLeft || document.documentElement.scrollLeft;  return {  scrollTop: scrollTop,  scrollTLeft: scrollLeft,  }  }  function getPage(e) {  var pageX = e.pageX || e.clientX + getScroll().scrollLeft;  var pageY = e.pageY || e.clientY + getScroll().scrollTop;    return {  pageX: pageX,  pageY: pageY,  }  }  var timeFlag = null;  document.onmousemove = function (e) {  if (timeFlag) return;  timeFlag = setTimeout(function () {  var image = document.getElementById('image');  image.style.left = getPage(e).pageX + 'px';  image.style.top = getPage(e).pageY + 'px';  console.log('hello');  timeFlag = null;  }, 20);  }; |

5、防抖和节流的区别

防抖：如果事件被频繁的触发，防抖能保证**只有第一次触发生效**，接下来n多次的触发都会被忽略。

节流：如果事件被频繁的触发，节流能够**减少事件触发的频率**，因此节流是**有选择性**的执行一部分事件。

三十二、webpacka使用的问题

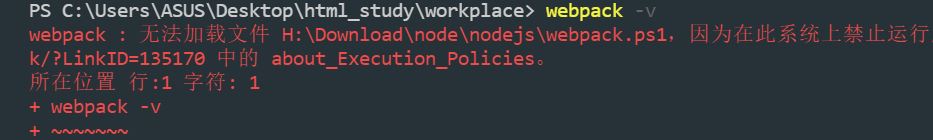
我直接用了最新版的5

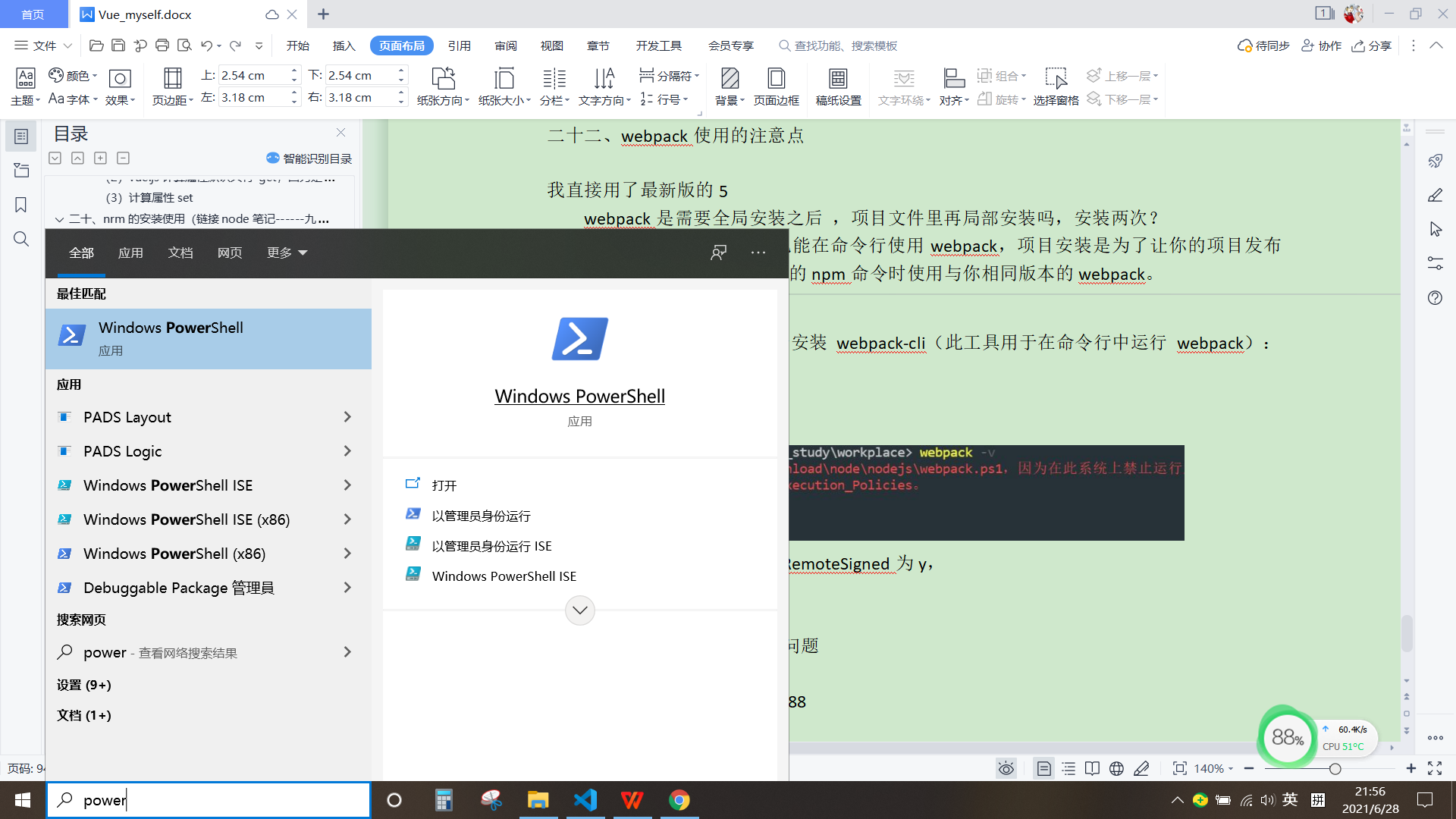
1、webpack是需要全局安装之后 ，项目文件里再局部安装吗，安装两次？

全局安装是为了让你自己能在命令行使用webpack，项目安装是为了让你的项目发布后，其他人可以在直接使用你的npm命令时使用与你相同版本的webpack。

1. 输入webpack -v报禁止运行脚本错误

以管理员的身份运行，直接运行是不可以的。输入 Set-ExecutionPolicy RemoteSigned为y。





3、webpack-dev-server中的问题，设置script: dev webpack-dev-server，执行npm run dev出现的报错问题，报如下错误 。

解决：webpack-dev-server 命令改成了 webpack serve。针对于高级版本。

internal/modules/cjs/loader.js:888

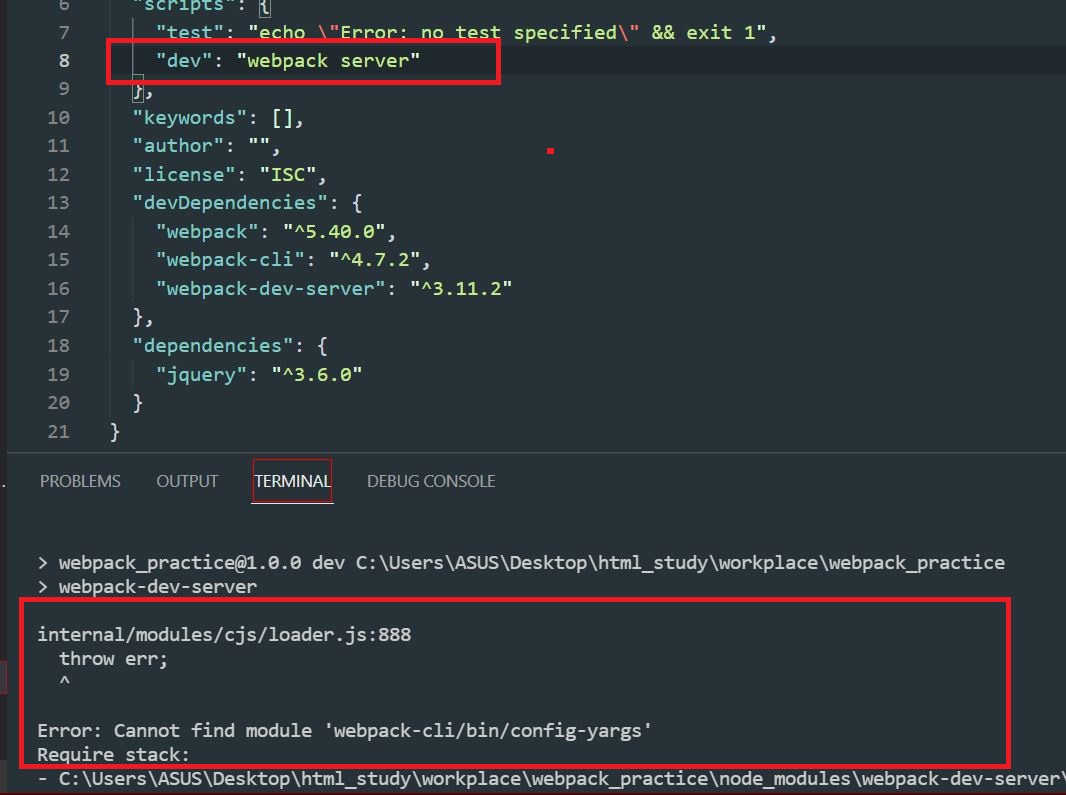
throw err;

^

Error: Cannot find module 'webpack-cli/bin/config-yargs'

Require stack:

- C:\Users\ASUS\Desktop\html\_study\workplace\webpack\_practice\node\_



4、5.40.0版本，警告的mode设置后好像速度快了很多

这个是我自己的，不然老是有警告影响调试,好像设置了之后速度快了很多

mode: 'development',

5、scss的包的编译问题

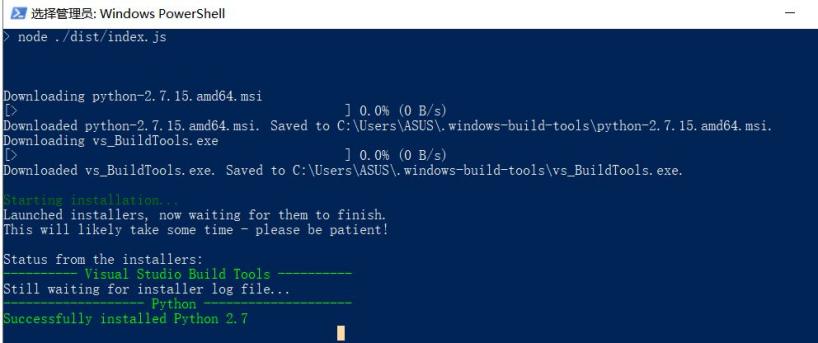
原因：windows平台缺少编译环境

解决方案：

第一步：安装node-gyp

npm install node-gyp -g

第二步： 安装windows-build-tools（前两步已经做了，可能重启电脑生效，暂时不搞）



npm install --global --production windows-build-tools

第三步： 卸载之前安装的node-sass

npm uninstall node-sass -D

第四步：重新安装node-sass

npm install node-sass -D

6、在html中，img src引用【object Module】，的问题

{test: /\.(png|gif|jpeg|jpg|webp)$/, use: [

{

// file-loader也可以

loader: 'url-loader',

options: {

esModule: false, // 这里设置为false，默认是true

}

}

]},

1、全局安装

哈希校验， 把这个打包出来的bundle.js，利用工具去进行哈希求值。如果它的值跟显示的哈希值一样，那么说明没有被修改，没错误。

webpack不要命名成文件名称

li\*10{这是第$个li}

问题：

1、如果看到.then，那么代表前面的那个方法是用promise方法封装的。Promise后边再学习，它的返回值是一个promise的，这个暂时没有学到。

2、那个测试接口怎么搞通的，好像涉及到了跨域。

答：服务器设置了响应头就行。允许所有的源访问。它视频的接口也是这么处理的。

3、jsonp直接传回来匿名函数吗？？？还是说是服务器规定的。

4、它们似乎是通过http请求去处理的。

答：它用的就是ajax，它是一个封装。

1. async和await的学习 ES7的内容

6、FormData是什么样的数据

7、服务器只能传输字符串，数字吗

8、Vue动画如果用transform:translate

除了 vue-resource 之外，还可以使用 axios 的第三方包实现实现数据的请求

问题：

在对象中ES6的语法：lang(){.........}

1、为什么访问msg没有通过data，访问data却为undefined（暂时没有完全解决，看后续学习有没有遇到）

答：百度vue method访问data。

实例创建之后，可以通过 vm.$data 访问原始数据对象。Vue 实例也**代理了 data 对象上所有的属性**，因此访问 vm.a 等价于访问 vm.$data.a。

2、v-for key中的渲染问题，跟diff算法的理解，Vue它渲染的机制，目前理解不了。只能有一个简单的认识。

1. 数组的forEach、some、filter、findIndex，这些都属于数组的新方法，都会对数组中的每一项，进行遍历，执行相关的操作，但是里边有不同，待理解

还有要总结下，包括字符串的方法有哪一些，还是挺多的。

js some()方法

语法

arr.some(callback(element[, index[, array]])[, thisArg])

参数

callback

用来测试每个元素的函数，接受三个参数：

element

数组中正在处理的元素。

index 可选

数组中正在处理的元素的索引值。

array可选

some()被调用的数组。

thisArg可选

执行 callback 时使用的 this 值。

返回值

数组中有至少一个元素通过回调函数的测试就会返回true；所有元素都没有通过回调函数的测试返回值才会为false。

如果用一个空数组进行测试，在任何情况下它返回的都是false。some() 被调用时不会改变数组。

4、CDN是什么东西？？？