# Vue框架

# 一、邂逅Vue

## 认识Vue.js

### （1）为什么学习Vue

因为项目中要用到，所以需要学习一下这个技术。

### （2）简单认识一下Vue.js

* 读音 同view这个单词，不是读V U E 三个字母的音
* Vue是一个渐进式的框架。关于渐进式：
* 渐进式意味着，你可以将Vue作为你应用的一部分嵌入系统中，带来更丰富的交互体验；
* 或者如果你希望将更多的业务逻辑使用Vue来实现，那么Vue的核心库及其生态；
* 比如 core+vue-router-vuex,也可以满足你的各种需求。
* Vue有很多特点和web开发中的高级功能：
* 解耦视图和数据
* 可复用的组件
* 前端路由技术
* 状态管理
* 虚拟DOM

## 2.Vuejs的安装方式

### (1) CDN引入

* 开发环境下可以使用下面的引入方式。

|  |
| --- |
| <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script> |

* 生产环境下可以使用压缩过后的引入方式，体积更小。

|  |
| --- |
| <script src="https://cdn.jsdelivr.net/npm/vue@2.6.11"></script> |

### (2) 下载和引入

直接下载 vue.js 的压缩包

### (3) NPM 安装管理（推荐使用）

在用 Vue 构建大型应用时推荐使用 NPM 安装。

NPM 能很好地和诸如 webpack 或 Browserify 模块打包器配合使用。

同时 Vue 也提供配套工具来开发单文件组件。

|  |
| --- |
| **npm install vue** |

## 3.Vuejs初体验

### (1) Hello Vue.js

|  |
| --- |
| **简单使用一下Vue,封装了一个对象，绑定了两个数据，插值表达式**  **只写了一个html文件**  **分成了三个步骤进行的操作** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>Hello Vue Demo 01</title>  </head>  <body>  <!--3.create a div element with id div01-->  <div id="div01">  <!-- 3.1 数据绑定，可以使用这种插值表达式的形式-->  <h1>{{message}}</h1>  <h2>{{name}}</h2>  </div>  <!--1.import the lib of vue-->  <script src="../js/vue.js"></script>  <!--2.create a obj with vue-->  <script >  const obj01 = new Vue({ // Vue 这个类就是引入的包中的封装类  el:'#div01', // 参数就是一个对象类型的，  data:{  message:'Hello Vue',  name:'Whc'  }  });  </script>      </body>  </html> |

### (2) Vue列表展示（循环指令 v-for）

|  |
| --- |
| **展示数组中的数据，循环指令 v-for的简单使用** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>简单循环数组</title>  </head>  <body>  <!-- 3.create html code -->  <div id="div02">  <ul>  <!-- 3.1 使用循环指令 v-for 来获取数组中的数据 -->  <li v-for="item in moveis">{{item}}</li>  </ul>  </div>  <!--1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create a object of Vue -->  <script>  let div02 = new Vue({  el:'#div02',  data:{  moveis:['西游记','三国演义','水浒传','红楼梦']  }  }  );  </script>    </body>  </html> |

### (3) 案例：计数器

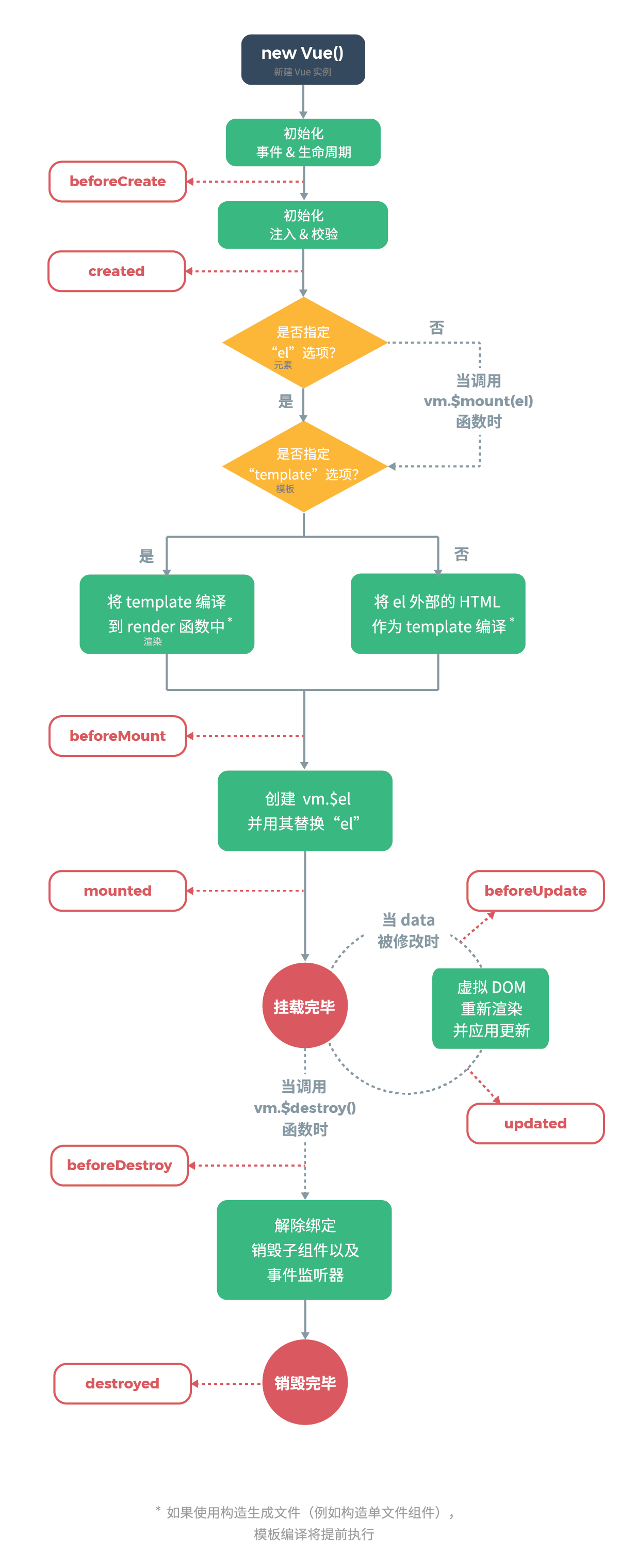
|  |
| --- |
| **1.自定义方法，对变量属性进行操作** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>计数器的小练习</title>  </head>  <body>  <!-- 3.create div03 element -->  <div id="div03">  <h2>当前的数据 ： {{counter}}</h2>  <!-- 3.1 点击事件 第一种写法 ：这种写法实际上是 v-指令的语法糖，就是一种简便的写法 -->  <button @click="jiajia()">+</button>  <!-- 3.2 点击事件 第二种写法 ： 视频中就是这样写的 -->  <button v-on:click="jianjian">-</button>  </div>  <!-- 1.import Vue -->  <script src="../js/vue.js"></script>  <!-- 2.create a object with vue -->  <script>  let div03 = new Vue({  el:'#div03',  data:{  counter:0  },  //下面定义两个方法，一个加加，一个减减  methods:{  jiajia:function(){ // 指定是一个方法 function 类型  this.counter++;  },  jianjian:function(){ // 指定是一个方法 function类型  this.counter--;  }  }  });  </script>    </body>  </html> |

## 4.Vuejs的MVVM

### (1) Vue中的MVVM

**Model – View – ViewModel(Controller)**

## 5.Vuejs.声明周期钩子



# 二、模板相关的语法

## 插值操作

### Mustache语法（双大括号）

|  |
| --- |
| 1. **直接插入变量** 2. **可以进行简单的变量计算** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>Mustache语法</title>  </head>  <body>  <!-- 3.div use mustache -->  <div id="div0201">  <!-- 3.1 可以直接把变量放进来 -->  <h1>{{message}}</h1>  <!-- 3.2 也可以写简单的表达式，进行拼接 -->  <h2>{{firstName + ' ' +secondName}}</h2>  <!-- 3.3 这也是直接把变量放进来 -->  <h2>{{firstName}} {{secondName}}</h2>  <!-- 3.4 可以进行简单的计算 -->  <h2>{{counter \* 2 }}</h2>  </div>  <!-- 1.import vue.js -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0201',  data:{  message:'Hello World!',  firstName:'王',  secondName:'宏昌',  counter:101  }  });  </script>    </body>  </html> |

### 其他指令操作

#### v-once指令 ：不进行响应式的变化

|  |
| --- |
| **只显示当时的内容，当变化的时候不再响应** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>v-once指令</title>  </head>  <body>  <!-- 3.div use mustache -->  <div id="div0202">  <!-- 3.1 可以直接把变量放进来，这个是响应式的，可变的 -->  <h1>{{message}}</h1>  <!-- 3.2 使用指令v-once之后，就不可变了 -->  <h1 v-once>{{message}}</h1>  </div>  <!-- 1.import vue.js -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  const obj = new Vue({  el:'#div0202',  data:{  message:'Hello World!',  }  });  </script>  </body>  </html> |

#### v-html指令 ： 把字符串进行html解析

|  |
| --- |
| **这种情况适用再网络请求返回结果是html的字符串的时候** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-html指令</title>  </head>  <body>  <!-- 3.use vue -->  <div id="div0203">  <h1>{{message}}</h1>  <!-- 3.1 直接展示的效果 -->  <h2>{{aaurl}}</h2>  <!-- 3.2 使用v-html指令进行解析 -->  <h2 v-html="aaurl"></h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create a obj with vue -->  <script>  let obj = new Vue({  el:'#div0203',  data:{  message:'Hello World!',  // 下面的这个属性是一个完整的html元素  aaurl:'<a href="https://www.baidu.com">百度一下</a>'  }  });  </script>  </body>  </html> |

#### v-text指令 ： 展示文本用的（一般不用！）

|  |
| --- |
| **会覆盖原来的文本** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-text指令</title>  </head>  <body>  <!-- 3.use v-text 指令 -->  <div id="div0204">  <h1>{{message}}</h1>  <!-- 3.1 使用 v-text 指令，当时会覆盖所有的原来的内容 -->  <h2 v-text="message">这里是原来的文本！</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create a obj with vue -->  <script>  let obj = new Vue({  el:'#div0204',  data:{  message : 'hello world!',  }  });  </script>  </body>  </html> |

#### v-pre指令 ： 原封不动的展示内容，不解析

|  |
| --- |
| **原封不动的把原来的文本显示出来，不进行Vue解析** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-pre指令</title>  </head>  <body>  <!-- 3.use v-pre 指令 -->  <div id="div0205">  <h1>{{message}}</h1>  <!-- 3.1 使用 v-pre指令直接展示原来的内容 -->  <h1 v-pre>{{message}}</h1>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create obj with vue -->  <script>  let obj = new Vue({  el:'#div0205',  data:{  message:'Hello World!'  }  });  </script>  </body>  </html> |

#### v-cloak指令 ： （基本不用，了解）当无法解析插属性的时候，不显示插值表达式，

|  |
| --- |
| **在vue解析之前，元素中有一个 v-cloak属性**  **在vue解析之后，元素中没有了v-cloak属性** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-cloak属性</title>  <!-- 4.写一个属性的样式来控制 -->  <style>  /\* 属性选择器 \*/  [v-cloak]{  display: none;  }  </style>  </head>  <body>  <!-- 3.use v-cloak 指令 -->  <div id="div0206">  <h1>{{message}}</h1>  <!-- 3.1 在vue解析之前，有一个 v-cloak属性  在vue解析之后，没有了 v-cloak属性  -->  <h1 v-cloak>{{message}}</h1>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  // 2.1 这里延迟一秒中 ，模拟message从没有到有的过程  setTimeout(function(){  const obj = new Vue({  el:'#div0206',  data:{  message:'Hello World!'  }  });  },1000);    </script>  </body>  </html> |

## 绑定属性 v-bind，动态绑定属性

### v-bind基本使用 ： 属性绑定

|  |
| --- |
| **绑定基本的属性，如图片的路径，等等** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-bind基本使用</title>  </head>  <body>  <!-- 3.use v-bind to bind property -->  <div id="div0301">  <h1>{{imageUrl}}</h1>  <img src="D:\EDailyRoutine\毕业相册\IMG\_3306.JPG" alt="图片不存在" width="200px" height="150px">  <!-- 3.1 使用 v-bind 绑定 图片的路径属性 -->  <img v-bind:src="imageUrl" alt="图片不存在" width="200px" height="150px">  <!-- 3.2 使用 v-bind 的语法糖（简写一个冒号即可）形式 绑定图片的路径属性 -->  <img :src="imageUrl" alt="图片不存在" width="200px" height="150px">  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0301',  data:{  imageUrl:'D:\\EDailyRoutine\\毕业相册\\IMG\_3306.JPG'  }  });  </script>  </body>  </html> |

### v-bind 动态绑定 class(1.对象语法)

|  |
| --- |
| **两种方式 ： 1.直接写；2.写方法获取方法的返回值，**  **注意 ： 逗号分割多个，一定是逗号而不是分号！** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-bind 动态绑定class - 对象语法</title>  <!-- 4.create a style script to use -->  <style>  .active{  color: red;  }  .AA{  font-size: 100px;  }  </style>  </head>  <body>  <!-- 3.use v-bind to bind class just like [ngClass] in Angular-->  <div id="div0302">  <h2>你好呀</h2>  <!-- 3.1 v-bind class : 方式一 ： 直接写对象表达式  对象语法 ： 绑定的内容是一个对象的形式 {类名 ： 布尔类型,类名2：布尔值2}  当布尔类型为true的时候，类 生效  当布尔类型为false的时候，类 无效  **注意 ： 当有多个类的时候，使用 逗号进行分割**  -->  <h2 :class="{active:isActive & 1 == 1,AA:true}">你好呀2</h2>  <!-- 3.2 v-bind class : 方式二 ： 通过方法的返回值来获取对象 -->  <h2 :class="getClass()">你好呀3</h2>  <!-- 3.3 我还可以写一个按钮，来控制这个是否生效,  调用方法的那个小括号是可以省略的（之后会讲到这个问题）  （这个是可选项）  -->  <button v-on:click="changeClass">是否生效 ？ {{isActive}}</button>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0302',  data:{  isActive:true  },  methods:{  // 2.1 这个方法用来控制类是否生效的  changeClass:function(){  this.isActive = !this.isActive;  },  // 2.1 把要设置的类的对象的字符串放到方法中返回  getClass:function(){  return {active:this.isActive & 1 == 1}  }  }  });  </script>  </body>  </html> |

### v-bind 动态绑定 class(2.数组语法,用的少)

|  |
| --- |
| **两种方式：1.直接写；2.写方法获取方法的返回值**  **有多个的时候，使用逗号进行隔开，注意 一定是逗号！** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-bind 绑定class - 数组语法</title>  <!-- 4.create style  这里的类名，需要和数据定义的那个是一样的，  这样才能够匹配上，否则是不行的  -->  <style>  .active{  color: green;  }  .fontSize{  font-size: 50px;  font-weight: bold;  }  </style>  </head>  <body>  <!-- 3.use v-bind to bind class -->  <div id="div0303">  <h2>你好呀</h2>  <!-- 3.1 这个地方可以使用数组的方式,把要绑定的属性放进来  通过逗号进行隔开  -->  <h2 :class="['active','fontSize']">你好呀2</h2>  <!-- 3.2 也可以通过方法的返回值的形式获取这个数组 -->  <h2 :class="getClass()">你好呀3</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0303',  // 2.1 定义数据的时候，可以把类的名字放进来备用  data:{  message:'hello world',  class1:'active',  class2:'fontSize'  },  methods:{  // 2.2 这个方法，返回类的名称的数组，在动态绑定类的时候就用上了  getClass:function(){  return [this.class1,this.class2]  }  }  });  </script>  </body>  </html> |

### v-bind + v-for的小练习 循环展示数据；点谁谁变色

|  |
| --- |
| **v-for 可以这样写 v-for=”(item,index) in lists”** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>点谁谁变色的练习</title>  <!-- 4.create style -->  <style>  .fontColor{  color: red;  }  </style>  </head>  <body>  <!-- 3.use v-bind and v-for to show content -->  <div id="div0304">  <!-- 3.1 v-for 循环扩展，通过（item,index）方式来获取循环的下标 -->  <!-- 3.2 v-on:click（@click）事件绑定，并把点中的下标传入方法中，进行处理 -->  <!-- 3.3 v-bind:class(:class) 属性绑定，根据条件进行属性的绑定，动态确定生效的属性 -->  <div v-for="(item,index) in movies"  @click="changeColor(index, event)"  :class="{fontColor:index == colorIndex}">  {{index}} - {{ item }}  </div>    </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0304',  data:{  movies:['火影忍者','钢之炼金术师','海贼王','死神','速度与激情9'],  // 2.1 这个用来控制变颜色的下标，这个是控制颜色的关键操作  colorIndex:0  },  methods:{  // 2.2 这个是点击的变色的方法，也就是改变了一个下标  changeColor:function(index,event){  this.colorIndex = index;  }  }  });  </script>  </body>  </html> |

### v-bind 动态绑定内联样式style(1.对象语法)

|  |
| --- |
| **两种方式 ：1.直接写；2.写方法获取方法的返回值，**  **注意 ：逗号分割多个** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-bind 动态绑定 style - 对象语法</title>  </head>  <body>  <!-- 3.use v-bind to bind style -->  <div id="div0305">  <h2>{{message}}</h2>  <!-- 3.1 内联样式绑定  格式 ：:style="{key(属性名称):value(属性值)}"  注意 ： 1. 属性名称 ：一定要使用 驼峰命名法 才可以 ，例如 fontSize，而不要使用 font-size  2. 属性值 ： 一定要用 单引号 括起来 ，表示这是一个字符串值，而不是一个变量需要去解析  **3. 当放入多个属性的时候，一定是要用 逗号 进行隔离开 ！很关键**  -->  <div :style="{fontWeight:'bold',color:'green',fontSize:'50px'}">{{message}}</div>    <!-- 3.2 通过数据绑定的方式来进行 样式的设定！  color : 直接返回的是字符串，可以直接使用  fontSize:属性中是一个数字，所以需要进行拼接字符串的操作！  -->  <div :style="{fontWeight:'bold',color:divColor,fontSize:divFontSize+'px'}">{{message}}</div>  <!-- 3.3 通过调用方法，获取返回值对象  这样解决了字符串过长的问题  -->  <div :style="getStyles()">{{message}}</div>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0305',  data:{  message:'Hello World!',  divColor:'red', // 2.1 可以放入style样式绑定中去进行绑定  divFontSize:100 // 2.2 设置字体大小的样式，需要使用的时候进行拼接字符串  },  methods:{  //3. 通过方法的方式，把样式的字符串进行返回回去  getStyles:function(){  return {fontWeight:'bold',color:this.divColor,fontSize:this.divFontSize+'px'}  }  }  });  </script>  </body>  </html> |

### v-bind 动态绑定内联样式 style(2.数组语法，用的少)

|  |
| --- |
| **先封装一个属性的对象，然后，把这个对象放入到数组中去，**  **这个真的是没有什么意义** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-bind 动态绑定style-数组绑定</title>  </head>  <body>  <!-- 3.use v-bind to bind style -->  <div id="div0306">  <h2>{{message}}</h2>  <!-- 3.1 使用数组绑定的方式进行样式绑定 ：  这个数组中的内容是 ：  一个预先定义好的对象，然后把这个对象放入进这个数组中  当有多个这样的对象的时候，都放进来就行  -->  <h2 :style="[baseStyle]">{{message}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0306',  data:{  message:'Hello World!',  baseStyle:{color:'green',fontSize:'100px'}  }  });  </script>  </body>  </html> |

## 计算属性（重要）

### 什么是计算属性？

* 我们知道，在模板中可以直接通过插值语法显示一些data中的数据。
* 但是某些情况下，我们可能需要对数据进行一些转化后再显示，或者需要多个数据结合起来进行显示
* 比如我们有firstName和lastName 两个变量，我们需要显示完整的名称
* 但是如果多个地方都需要显示完整的名称，我们就需要写多个{{firstName}}{{lastName}}
* 我们可以将上面的代码换成计算属性
* Ok,我们发现，计算属性是写在实例的computed选项中的

### 计算属性的基本使用

|  |
| --- |
| **1.定义在对象的computed中，写成方法的形式，**  **2.但是声明的时候使用名词形式（一个属性），使用的时候不需要添加小括号** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>计算属性的基本使用</title>  </head>  <body>  <!-- 3.use 计算属性 to show data -->  <div id="div0401">  <!-- 3.1拼接字符串的方式 -->  <h2>{{firstName + ' ' + lastName}}</h2>  <!-- 3.2 两个分开写的方式 -->  <h2>{{firstName}} {{lastName}}</h2>  <!-- 3.3 \*\*\* 使用计算属性的方式！ \*\*\* -->  <h2>{{fullName}}</h2>  <!-- 3.4 使用方法的方式（不常用） -->  <h2>{{getFullName()}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0401',  data:{  firstName:'王',  lastName:'宏昌'  },  //2.1 定义计算属性，也是定义成了一个方法的形式  computed:{  fullName:function(){  return this.firstName+' '+this.lastName;  }  },  //2.2 使用method 的方式也可以实现  methods:{  getFullName:function(){  return this.firstName +' '+this.lastName  }  }  });  </script>  </body>  </html> |

### 计算属性的复杂操作

|  |
| --- |
| **就是数据结构复杂时候的计算** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>计算属性的复杂操作</title>  </head>  <body>  <!-- 3.show complete data with 计算属性 -->  <div id="div0402">  <!-- 3.1 计算书的总价格，这就涉及到了比较复杂的计算  因此，比较适合使用计算属性  -->  <h2>总价格是 ： {{totalPrice}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0402',  data:{  //2.1 定义了一个书的数组，  books:[  {id:1001, name:'西游记',price:100},  {id:1002, name:'三国演义',price:200},  {id:1003, name:'水浒传',price:300},  {id:1004, name:'红楼梦',price:400},  ]  },  computed:{  //2.2 定义一个求书的总价的计算属性  totalPrice:function(){  let t\_price = 0;  for(const book of this.books){  console.log(book);  t\_price += book.price;  }  return t\_price;  }  }  });  </script>  </body>  </html> |

### 计算属性的setter方法和getter方法,

|  |
| --- |
| **计算属性的完整的定义方式，一般情况下set方法不用写，只需要写get方法即可，**  **也就是叫一个只读属性，**  **这里就解释了为什么使用计算属性的时候不需要添加小括号了** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>计算属性的getter setter 方法</title>  </head>  <body>  <!-- 3.use 计算属性 to show data -->  <div id="div0403">  <h2>{{fullName1}}</h2>  <h2>{{fullName2}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0403',  data:{  firstName:'王',  lastName:'宏昌'  },    computed:{  //2.1 计算属性的完整定义方法  fullName1:{  set:function(newValue){  console.log('fullName1 : ',newValue);  },  get:function(){  return this.firstName+' '+this.lastName  }  },  //2.2 一般情况下 set 方法可以省略，所以可以简写成如下形式  // 因此，虽然看上去是个方法，但实际上是一个属性  // 所以，在使用的时候，不需要加小括号  fullName2:function(){  return this.firstName+' '+this.lastName;  }  }  });  </script>  </body>  </html> |

### 计算属性和methods的对比 计算属性的缓存问题，

|  |
| --- |
| **方法使用几次就调用几次，但是计算属性使用多次，只调用一次get方法，内部的缓存，节省了资源** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>计算属性与methods的对比</title>  </head>  <body>  <!-- 3. use 计算属性 to show data -->  <div id="div0404">  <!-- 3.1 调用 5次方法 -->  <h2>{{getFullName()}}</h2>  <h2>{{getFullName()}}</h2>  <h2>{{getFullName()}}</h2>  <h2>{{getFullName()}}</h2>  <h2>{{getFullName()}}</h2>  <!-- 3.2 调用5次 计算属性 -->  <h1>{{fullName}}</h1>  <h1>{{fullName}}</h1>  <h1>{{fullName}}</h1>  <h1>{{fullName}}</h1>  <h1>{{fullName}}</h1>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0404',  data:{  firstName:'王',  lastName:'宏昌'  },  //2.1 使用方法的操作：使用几次调用几次，资源消耗大  methods:{  getFullName:function(){  console.log('调用方法展示全名');  return this.firstName+' '+this.lastName;  }  },  //2.2 使用计算属性的操作 ： 有内部缓存  computed:{  fullName:function(){  console.log('使用计算属性展示全名');  return this.firstName+' '+this.lastName;  }  }  });  </script>  </body>  </html> |

## 事件监听

### v-on的基本使用

|  |
| --- |
| **没有参数的时候，可以省略小括号** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>事件监听的基本用法</title>  </head>  <body>  <!-- 3.use v-on and @ to add listener -->  <div id="div0501">  <h2>{{counter}}</h2>  <!-- 3.0 在添加事件 的方法的时候，如果没有参数可以不用写小括号 -->  <!-- 3.1 使用 “v-on:事件名称” 的方式进行 事件监听-->  <button v-on:click="jiajia()">增加1</button>  <!-- 3.2 使用v-on的语法糖进行简写 “@事件名称” 的方式进行事件监听 -->  <button @click="jianjian">减少1</button>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0501',  data:{  counter:0  },  methods:{  //2.1 ES6 的定义函数的语法，直接写就🆗 了  jiajia(){  this.counter++;  },  jianjian(){  this.counter--;  }  }  });  </script>    </body>  </html> |

### v-on的参数问题

|  |
| --- |
| **函数中定义了参数：**  **1.没有传入参数但是有小括号的时候，则默认值是undefined;**  **2.没有传入参数但是也没有小括号的时候，默认传入的是监听的事件，也就是浏览器默认的那个事件对象；3..正常传入参数，则正常接收参数并使用；**  **4.参数中需要event参数的时候，直接写入$event作为参数即可,固定的写法就是这样的;**  **5.一般情况下都会按照正常的函数参数进行传入，谁会定义了参数又不去传进参数去呢？** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>事件监听的参数问题</title>  </head>  <body>  <!-- 3.use v-on about params -->  <div id="div0502">  <!-- 3.1 调用没有参数的方法 -->  <button @click="btnClick01">没有参数的方法</button>  <!-- 3.2 方法有参数，但是没有传入参数1 ： 写上了小括号 -->  <button @click="btnClick02()">默认是undefined</button>  <!-- 3.3 方法有参数，但是没有传入参数2 ： 没有写小括号 -->  <button @click="btnClick02">默认是触发的事件</button>  <!-- 3.4 方法有多个参数，并且需要事件作为参数：固定写法 ： $event作为事件的那个参数 -->  <button @click="btnClick03('wanghongchang',$event)">事件作为参数$event</button>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0502',  data:{  message:'Hello World!'  },  methods:{  btnClick01(){  console.log('这个是没有参数的点击方法');    },  btnClick02(namestr){  console.log('有一个参数的点击方法 ： 参数是 ：',namestr);    },  btnClick03(namestr,event){  console.log('有两个参数的点击方法');  console.log('第一个参数是 ：',namestr);  console.log('第二个参数是 ：',event);  }  }  });  </script>  </body>  </html> |

### v-on的修饰符

|  |
| --- |
| **1. .stop修饰符 ：可以阻止事件冒泡；**  **2. .prevent 修饰符 ：阻止默认事件（从来没有用到过）**  **3..{keycode|keyAlias} : 当事件是从特定的按键触发时才发生的回调；**  **4.native : 监听组件根元素的原生事件；**  **5.onec : 只触发一次回调** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>事件监听中的几个修饰符</title>  </head>  <body>  <!-- 3.use vue -->  <div id="div0503">  <!-- 3.1 .stop 阻止事件冒泡 -->  <div @click="divClick()">  <h5>this is a div</h5>  <!-- 在时间后面添加 .stop的修饰符 -->  <button @click.stop="btnClick()">事件冒泡的按钮</button>  </div>  <hr>  <!-- 3.2 .prevent 阻止默认事件 -->  <form action="baidu" method="get">  <input type="submit" value="阻止默认的提交动作" @click.prevent="mySubmit()" >  </form>  <hr>  <!-- 3.3 .keyCode | .keyAlias 指定 按键触发事件 -->  <input type="file" name="heihei" id="file01">  <input type="text" @keyup.enter="inputok()" value="">  <hr>  <!-- 3.4 .native 与自定义的组件有关，后面再讲 -->  <!-- 3.5 .onec 只触发一次的限定 -->  <button @click.once="clickonce()">只触发一次</button>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0503',  data:{  message:'Hello World!'  },  methods:{  //2.1 阻止冒泡的操作  divClick(){  console.log('div click');  },  btnClick(){  console.log('btn click');  },  //2.2 阻止默认事件的发生  mySubmit(){  console.log('阻止默认的提交事件的发生');  },  //2.3 指定按键的名称或者按键代码触发的事件  inputok(value){  console.log('输入完成！');  },  //2.4 .native 跳过，后面涉及到  //2.5 限定只触发一次  clickonce(){  console.log('只触发一次的函数');    }  }  });  </script>  </body>  </html> |

## 条件判断（比较简单）

### v-if （相当于\*ngIf)

|  |
| --- |
| **非零数字为true，布尔值，布尔表达式** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-if 条件判断</title>  </head>  <body>  <!-- 3.use v-if to define wheather h2 is show or not -->  <div id="div0601">  <!-- 3.1 v-if = "布尔类型的值 或者 布尔表达式 或者时非零数字！" -->  <h2 v-if="0">{{message}}</h2>  <h2 v-if="true">{{message}}</h2>  <h2 v-if="1 == 1">{{message}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0601',  data:{  message : 'Hello World!'  }  });  </script>  </body>  </html> |

### v-if 和 v-else结合使用

|  |
| --- |
|  |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>v-if 和 v-else 的结合使用</title>  </head>  <body>  <!-- 3.show data using v-if and v-else -->  <div id="div0602">  <h1 v-if="false">{{message1}}</h1>  <!-- 3.1 当上面的那个 v-if 的判断为false 的时候，显示下面的这个 -->  <h2 v-else>{{message2}}</h2>  <!-- 3.2 只和最近的v-if挨着的才有效果，下面的就不行了 -->  <h3 v-else>hello world h3</h3>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0602',  data:{  message1 : 'hello ',  message2: 'hahah'  }  });  </script>  </body>  </html> |

### v-if 和 v-else-if 和 v-else 的使用

|  |
| --- |
| **(当然是不推荐的使用方式)** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>条件判断的操作</title>  </head>  <body>  <!-- 3.show data with 三种条件 -->  <div id="div0603">  <h2 v-if="score >= 90">优秀</h2>  <h2 v-else-if="score >= 70">良好</h2>  <h2 v-else-if="score >= 60">及格</h2>  <h2 v-else>不及格</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0603',  data:{  score:99  }  });  </script>  </body>  </html> |

### 条件判断的小练习

|  |
| --- |
| **使用按钮来切换两种登陆方式 ： 用户名登陆 or 邮箱登陆** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>点击按钮切换登陆状态</title>  </head>  <body>  <!-- 3.show html with v-if and v-else -->  <div id="div0604">  <!-- 3.1 模拟使用用户名登陆的操作 -->  <div v-if="isUser">  <label for="username">用户名</label>  <input type="text" name="userName" id="username" placeholder="请输入用户名">  </div>  <!-- 3.2 模拟使用用户邮箱登陆的操作 -->  <div v-if="!isUser">  <label for="email">用户邮箱</label>  <input type="text" name="email" id="email" placeholder="请输入用户邮箱">  </div>  <!-- 3.3 点击按钮进行登陆方式的切换操作 -->  <button @click="isUser = !isUser">切换登陆状态</button>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0604',  data:{  isUser:true  }  });  </script>  </body>  </html> |

### 条件判断练习中的一个问题

* **小问题**
* 如果我们再有输入内容的情况下切换了输入类型，则输入的内容仍然存在
* 但是，按道理讲，我们因该已经切换到另一个input元素中了
* 在另一个input元素中我们并没有输入内容
* 为什么会出现这个问题呢？
* **问题解答**
* **这是因为Vue在进行DOM渲染时，出于性能考虑，会尽可能的复用已经存在的元素，而不是重新创建新的元素**
* **在上面的案例中，Vue 发现原来的input元素不再使用，所以直接作为下一个input元素来使用了**
* **解决方案**
* **如果我们不希望出现上述的复用问题，可以为相同的元素添加key**
* **并且保证key不同即可**

### v-show的使用

|  |
| --- |
| **1.用法和v-if相似，也是用来决定一个元素是否进行渲染的**  **2.v-if 为 false时，在DOM 中根本不会存在这个元素**  **3.v-show 为false 时，仅仅将元素的display属性设置为none而已，此元素还是存在的** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-show 的使用</title>  </head>  <body>  <!-- 3.use v-show 当为false的时候，dom仍然存在，不过display属性为 none-->  <div id="div0605">  <h2 v-if="isShow" id="aa">{{message}}</h2>  <h2 v-show="isShow" id="bb">{{message}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0605',  data:{  message:'Hello World!',  isShow:true  }  });  </script>  </body>  </html> |

## 循环遍历

### 遍历数组

|  |
| --- |
| 1. **只遍历数组元素；** 2. **同时遍历数组元素和索引** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>循环数组</title>  </head>  <body>  <!-- 3.use v-for to show array -->  <div id="div0701">  <!-- 3.1 只显示 数组中的值 -->  <ul>  <li v-for="movie in movies">{{movie}}</li>  </ul>  <!-- 3.2 显示 数组值和 对应的索引 （value,index）  注意 ： 前面是 值，后面是索引  -->  <ul>  <li v-for="(movie,index) in movies">{{index}} - {{movie}}</li>  </ul>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0701',  data:{  movies:['西游记','水浒传','三国演义','红楼梦']  }  });  </script>  </body>  </html> |

### 遍历对象

|  |
| --- |
| 1. **只遍历对象的属性值** 2. **同时遍历对象的属性值和属性名** 3. **同时遍历对象的属性值和属性名和索引** |
| <!DOCTYPE html>  <html lang="en">  <head>  <title>循环对象</title>  </head>  <body>  <!-- 3.use v-for to show object -->  <div id="div0702">  <!-- 3.1 只显示 对象的属性值 -->  <ul>  <li v-for="value in studentInfo">{{value}}</li>  </ul>  <!-- 3.2 显示对象的属性值和属性名 注意顺序，先属性值在属性名（property\_value,property\_name） -->  <ul>  <li v-for="(value,name) in studentInfo">{{value}} : {{name}}</li>  </ul>  <!-- 3.3 显示对象的属性值和属性名和对应索引，注意顺序 （value,name,index）-->  <ul>  <li v-for="(value,name,index) in studentInfo">{{value}} : {{name}} - {{index}}</li>  </ul>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0702',  data:{  studentInfo:{  name:'whc',  age:18,  school:'清华大学'  }  }  });  </script>  </body>  </html> |

### 绑定key属性提高循环的操作效率(了解)

|  |
| --- |
| 在中间插入数据的时候，为了提高数据的插入效率，可以在循环的时候绑定key属性，  不过，绑定的时候需要注意，绑定的那个内容需要是唯一的！ |
| <ul>  <li v-for="value in studentInfo" ：key=”value‘>{{value}}</li>  </ul>  就想这个value需要是唯一的 |

### 数组中的那些方法是响应式的

|  |
| --- |
| **Push()方法** ：数组最后追加新的元素，可以追加多个元素 |
| **Pop()方法** ：删除数组最后的元素 |
| **Shift()方法** ：删除数组第一个元素 |
| **Unshift()方法** ：在数组最前面添加元素，可以添加多个元素 |
| **Splice()方法** ： 1.删除元素；2.插入元素；3.替换元素 有这三个功能，根据参数来决定的功能 |
| **Sort()方法** ： 排序 |
| **Reverse()方法** ： 数组反转 |
| **通过索引改变该位置的元素就不是响应式的 ： this.movies[2] = ‘叶问4’ ，不是响应式的**  **但是 数组 确实是已经改变了，但是，页面 确实是不会改变的!**  **当有这个需求的时候，**  **方式一 ：就是用splice(start,num,value)的替换功能来实现! this.movies.splice(2,1,’叶问4’)**  **方式二 ：Vue内部的set函数来实现 ：Vue.set(this.movies,2,’叶问4’);**  **Delete函数 Vue.delete(对象，对象属性 或者 数组索引) ：删除对象的属性或者数组中该索引的值，**  **Vue中的 Set 和 delete 是响应式的！** |

## 阶段案例 – 购物车案例

### 界面搭建

|  |
| --- |
| **把代码分成了三个文件写，html, js, css文件（常规操作！）** |
| **右边的是效果图** |

### 使用过滤器（新的简单的知识点）

|  |
| --- |
| **过滤器就是 Angular中的管道！**   1. **定义过滤器 filters** 2. **使用过滤器 ，通过 符号 |** |
| **第一个图是定义过滤器**  **第二个图是使用过滤器，把数据进行格式化显示** |

### 改变购买的数量:事件监听

|  |
| --- |
| **这个就是正常的常规操作即可** |

### 源代码

#### (4.1) Html代码

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>购物车</title>  <!-- 3.import style.css file -->  <link rel="stylesheet" href="style.css">  </head>  <body>  <div style="text-align: center;">  <h2> Welcome to Your car!</h2>  </div>  <div id="car">  <table border="1" cellpadding=10 cellspacing=0>  <thead>  <tr>  <th></th>  <th>名称</th>  <th>价格</th>  <th>数量</th>  <th>操作</th>  </tr>  </thead>  <tbody>  <tr v-for="(book,index) in booklist" >  <td>{{book.id}}</td>  <td>{{book.bookName}}</td>  <!-- 2.1保留两位小数 方式一，使用toFixed() 方法 -->  <!-- <td>￥{{book.bookPrice.toFixed(2)}}</td> -->  <!-- 2.1保留两位小数 方式二，通过使用过滤器（就是Angular中的管道的概念） -->  <td>{{book.bookPrice | showPrice}}</td>  <td>  <!-- 3.1 减少一本书,只能减到0，且为0的时候，就不能再点了 -->  <button @click="jianjian(book)" :disabled="book.bookNumber == 0">-</button>  {{book.bookNumber}}  <!-- 3.2 增加一本书 -->  <button @click="jiajia(book)">+</button>  </td>  <!-- 4.1 删除一本书 -->  <td><button @click="deleteOne(index)">删除</button></td>  </tr>  <!-- 4.2 当所有的书都删除完成的时候，显示下面这一行 -->  <tr v-if="booklist.length == 0">  <td colspan="5">当前无选中图书！</td>  </tr>  <!-- 5.使用计算属性和过滤器显示选中的图书的总价格 -->  <tr v-if="booklist.length != 0">  <td colspan="5">图书总价格 ： {{totalPrice}}</td>  </tr>  </tbody>  </table>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.import main.js -->  <script src="./main.js"></script>  </body>  </html> |

#### (4.2) Js代码

|  |
| --- |
| let car = new Vue({  el:'#car',  data:{  booklist:[  {id:1001,bookName:'《Java从入门到精通》',bookPrice:98,bookNumber:1},  {id:1002,bookName:'《Mybatis从入门到精通》',bookPrice:35,bookNumber:1},  {id:1003,bookName:'《Spring原理》',bookPrice:120,bookNumber:1},  {id:1004,bookName:'《Python精讲》',bookPrice:119,bookNumber:1},  {id:1005,bookName:'《鸟哥私房菜》',bookPrice:198,bookNumber:1}  ],  },  //2.过滤器显示需要格式的数据！  filters:{  //2.1 声明成方法的形式，参数就是要格式化的数据  showPrice(price){  //2.2 返回的结果就是你需要的格式  return '￥'+price.toFixed(2);  }  },  //3.事件方法  methods:{  //3.1 减少一本书,同时需要改变总的价格  jianjian(book){  if(book.bookNumber > 0){  book.bookNumber--;  }    },  //3.2 增加一本书，同时需要改变总的价格  jiajia(book){  book.bookNumber++;  },  //4.删除一本书  deleteOne(index){  this.booklist.splice(index,1);  }  },  //5.计算属性 ： 总价格  computed:{  totalPrice(){  let aaprice = 0;  for (const book of this.booklist) {  // 注意 ： 这里计算的时候需要和数量进行一起操作，  //这样当数量发生变化的时候，总价格也会发生对应的变化  aaprice += book.bookPrice\*book.bookNumber;  }  return '￥'+aaprice.toFixed(2);  }  }  }); |

#### (4.3) Css代码

|  |
| --- |
| 本案例并没有写什么css代码，哈哈哈哈 |

### JavaScript扩展必会！：三个高阶函数的用法

|  |
| --- |
| /\*\*  \* Javascript中的三个高阶函数的学习（拓展补充回顾知识点）  \*  \* 三个需求 ：  \* 1.取出数组中所有小于100的数，并返回一个新的数组  \* 2.将所有小于100的数进行转化 ： 全部\*2，并返回一个新的数组  \* 3.将所有小于100的数进行求和，并返回  \*  \* 三个高阶函数 ：  \* filter/map/reduce  \*  \*  \* 1.filter：过滤数据  \* 1.1 参数是一个回调函数，且回调函数的返回值必须是一个布尔值  \* 1.2 返回值就是一个新的数组  \* 1.3 回调函数的返回值为 true : 函数内部会自动将这次回调的n加如新的数组  \* 1.4 回调函数的返回值为false : 函数内部会过滤掉这次的n  \*  \* 2.map:对数据进行二次加工  \* 2.1 参数是一个回调函数，  \* 2.2 返回值就是一个新的数组  \* 2.3 可以将你想要的逻辑操作写在这个回调函数的内部  \*  \* 3.reduce:对数据进行汇总  \* 3.1 参数是一个回调函数 + 开始初始化的初始值 ，有两个参数  \* 3.2 回调函数的参数：(前一个值，当前的值，)  \* 3.3 返回值是一个数  \*  \*/  const nums = [101,58,63,166,189,21,75];  /\*\*  \* 1.使用filter 函数  \* 1.filter函数 传入的参数是一个函数  \* 2.这个函数的参数n,就是nums数组中的每一个元素  \* 3.返回是一个布尔值，当为true的时候，判断成功，这个n,就会被放入newNums1数组中去  \* 4. 当为false的时候，判断失败，这个n,就会被忽略  \*/  let newNums1 = nums.filter((n)=>{  return n < 100  });  console.log(newNums1); //结果为 ： [58,63,21,75]  /\*\*  \* 2.使用map函数  \* 1.map函数 传入的参数是一个函数  \* 2.这个函数的参数n,就是nums数组中的每一个元素  \* 3.对每一个n进行一个逻辑操作，然后把新的结果进行返回  \* 4.注意，哪个数组进行调用就会遍历几遍  \*/  let newNums2 = newNums1.map((n)=>{  if(n < 100){  return n \* 2  }  });  console.log(newNums2);  /\*\*  \* 3.使用reduce函数  \* 1.reduce函数 传入的参数一 是一个回调函数  \* 传入的参数二 是一个开始的初始化值，比如0  \* 2.这个函数的参数n,就是nums数组中的每一个元素  \* 3.对每一个n进行一个逻辑操作，然后把新的结果进行返回  \* 4.注意，哪个数组进行调用就会遍历几遍  \*/  let totalNumbs = newNums2.reduce((preValue,n)=>{  return preValue+n;  },0);  console.log('总数是 ： ',totalNumbs);  /\*\*  \*  \* 高阶函数连续操作1.，一套连招搞定你  \*/  let ttl = nums.filter((n)=>{  return n < 100;  }).map((n)=>{  return n \* 2;  }).reduce((preValue,n)=>{  return preValue + n;  },0);  console.log('总数2 是 ： ',ttl);  /\*\*  \* 一行代码的操作  \*/  let ttl2 = nums.filter(n => n < 100).map(n => n\*2).reduce((preValue,n)=> preValue+n,0)  console.log('一行代码搞定的 三个操作 ：',ttl2); |

## 8.v-model （可以实现双向绑定）

### (1) v-model的基本使用

|  |
| --- |
| **这个v-model 就和 Angular 中的【（ngModel）】是一样的，数据的双向绑定** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-model 基本使用</title>  </head>  <body>  <!-- 3.use v-model -->  <div id="div0901">  <!-- 3.1 就是在这里进行数据的双向绑定 -->  <input v-model="message" type="text" />  <h2>{{message}}</h2>  <hr>  <!-- 3.2 其实自己也可以实现数据的双向绑定  1. 响应式绑定数据 :value="message2"  2. 添加事件监听，当输入的数据发生变化的时候，跟着变  @input="message2 = $event.target.value"  -->  <input type="text" :value="message2" @input="message2 = $event.target.value">  <h2>{{message2}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0901',  data:{  message:'Hello World!',  message2:'abc'  }  });  </script>  </body>  </html> |

### (2) v-model结合radio使用

|  |
| --- |
| **Radio就是单选按钮，基本的输入操作** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-mode 结合 radio 单选按钮的使用</title>  </head>  <body>  <!-- 3.use v-model and radio -->  <div id="div0902">  <!--  v-model = "sex" : 指明了双向绑定的属性  value = "男/女" ：指明了绑定的值  -->  <label for="male">  <input type="radio" name="sex\_group" id="male" v-model="sex" value="男"> 男  </label>  <label for="female">  <input type="radio" name="sex\_group" id="female" v-model="sex" value="女"> 女  </label>  <h2>{{sex}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0902',  data:{  sex:'男' // 这个是要双向绑定的数据,初始的选择  }  });  </script>  </body>  </html> |

### (3) v-model结合checkbox使用

|  |
| --- |
| **复选框分为两种情况 ：**  **1.单个勾选框（比如同意协议的时候）**  **2.多个勾选框（比如爱好的选择）** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-model 结合 checkbox的使用</title>  </head>  <body>  <!-- 3.use v-model and checkbox -->  <div id="div0903">  <!-- 3.1 单个框的时候，选中同意，才可以点击下一步继续操作 -->  <label for="agree">  <input type="checkbox" id="agree" v-model="isAgree"/> 同意本协议  </label>  <h2>是否同意本协议 ： {{isAgree}}</h2>  <button :disabled="!isAgree">下一步</button>  <hr>  <!-- 3.2 多个框的时候， -->  <label for="hobby1">  <input type="checkbox" id="hobby1" v-model="hobbies" value="打篮球">打篮球  </label>  <label for="hobby2">  <input type="checkbox" id="hobby2" v-model="hobbies" value="踢足球">踢足球  </label>  <label for="hobby3">  <input type="checkbox" id="hobby3" v-model="hobbies" value="打羽毛球">打羽毛球  </label>  <label for="hobby4">  <input type="checkbox" id="hobby4" v-model="hobbies" value="吃西瓜">吃西瓜  </label>  <label for="hobby5">  <input type="checkbox" id="hobby5" v-model="hobbies" value="看动画片">看动画片  </label>  <h2>你的爱好有 : {{hobbies}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0903',  data:{  //2.1 单选框的属性 ： 布尔值  isAgree:false,  //2.2 多选框的属性 ： 一个数组  hobbies:[]  }  });  </script>  </body>  </html> |

### (4) v-model结合select 使用

|  |
| --- |
| **也分为单选和多选两种情况** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-model 结合 select 的使用</title>  </head>  <body>  <!-- 3.use v-model and select -->  <div id="div0904">  <!-- 3.1 单选下拉框 -->  <select name="fruitSelect" id="a" v-model="fruit">  <option value="苹果">苹果</option>  <option value="桃子">桃子</option>  <option value="香蕉">香蕉</option>  <option value="梨">梨</option>  </select>  <h2>您选中的水果是 ：{{fruit}}</h2>  <!-- 3.2 多选下拉框 ,需要添加属性 multiple  但是由于标签的特殊性，windows中需要按住ctrl键  Mac中需要按住command键  才能够实现多选的操作  -->  <select name="fruitSelect\_m" id="b" v-model="fruits" multiple="multiple">  <option value="苹果">苹果</option>  <option value="桃子">桃子</option>  <option value="香蕉">香蕉</option>  <option value="梨">梨</option>  </select>  <h2>您选中的水果是 ：{{fruits}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0904',  data:{  // 2.1 单选的属性,只是一个字符串  fruit:'香蕉',  // 2.2 多选的属性，必须是一个数组  fruits:[]  }  });  </script>  </body>  </html> |

### (5) v-model 中的值绑定操作，

|  |
| --- |
| **实际上就是 原来有一个数据（或者是通过网络请求获取的数据），然后在这些数据的基础上进行可选则的操作，而不是直接把数据写成固定的操作** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-model + 值绑定操作</title>  </head>  <body>  <!-- 3.use v-model to 值绑定 -->  <div id="div0905">  <!--  实际上这里就是用了一个循环的操作，  把数据动态的进行了一个绑定 v-bind  而不是直接写成固定的形式  -->  <label v-for="hobby in originHobbies" :for="hobby">  <input type="checkbox" :id="hobby" v-model="hobbies" :value="hobby">{{hobby}}  </label>  <h2>你自己选中的爱好是 ： {{hobbies}}</h2>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0905',  data:{  //2.1 模拟 网络请求过来的数据  originHobbies:['打篮球','踢足球','打羽毛球','看电视','吃冰激凌','看动画片'],  //2.2 这个数组用来保存 自己选中的 爱好  hobbies:[]  }  });  </script>  </body>  </html> |

### (6) v-model的修饰符

* **Lazy修饰符**
* 默认情况下，v-model模式实在input事件中同步输入框的数据的。
* 也就是说，一旦有数据发生改变对应的data中的数据就会自动发生改变。
* Lazy修饰符可以让数据在示取焦点或者回车时才会更新
* **Number修饰符**
* 默认情况下，在输入框中无论我们输入的是字母还是数字，都会被当作字符串类型进行处理。
* 但是如果我们希望处理的是数字类型，那么最好直接将内容当作数字处理。
* Number修饰符可以让在输入框中输入的内容自动转换成数字类型。
* **Trim修饰符**
* 如果输入的内容首尾有很多空格，通常我们希望将其去除
* Trim修饰符可以过滤内容左右两边的空格

|  |
| --- |
| **下面就是 三个修饰符的代码** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>v-model 的三个修饰符</title>  </head>  <body>  <!-- 3.use 三个修饰符 ： .lazy .number .trim -->  <div id="div0906">  <!-- 1. 修饰符 .lazy ：可以在失去焦点或者按enter时同步数据 -->  <input type="text" v-model.lazy="data1">  <h2>{{data1}}</h2>  <hr>  <!-- 2. 修饰符 .number : 可以实现数字的转换-->  <!-- 2.1 虽然 type = "number" ,但是类型仍然是 字符串类型 -->  <input type="number" v-model="data2\_1" >  <h2>使用 type="number" 的方式绑定数据 : {{data2\_1}} - {{typeof data2\_1}}</h2>  <!-- 2.2 使用 .number 修饰符后，类型被转换成了 number数字类型 -->  <input type="text" v-model.number="data2\_2">  <h2>使用.number修饰符 绑定数据 ： {{data2\_2}} - {{typeof data2\_2}}</h2>  <hr>  <!-- 3. 修饰符 .trim ：去除前后的空格，这个需要在控制台查看效果 -->  <input type="text" v-model.trim="data3">  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div0906',  data:{  //2.1 .lazy 的绑定的数据  data1:'',  //2.2 .number 的绑定的数据  data2\_1:0,  data2\_2:0,  //2.3 .trim 的绑定的数据  data3:'',  }  });  </script>  </body>  </html> |

# 三、组件化（十分重要）

## 1. 什么是组件化？

* **人面对复杂问题的处理方式：**
* 任何一个人处理信息的逻辑能力都是有限的。
* 所以，当面对一个非常复杂的问题的时候，我们不太可能一次性搞定一大堆的内容。
* 但是，我们人有一种天生的能力，就是将问题进行拆解。
* 如果将一个复杂的问题，拆分成很多个可以处理的小问题，再将其放在整体当中，你会发现大的问题也会迎刃而解。
* **组件化也是类似的思想**
* 如果我们将一个页面中所有的处理逻辑全部放在一起，处理起来就会变的非常复杂，而且不利于后续的管理以及扩展。
* 但如果，我们将一个页面拆分成一个个的小功能块，每个功能块完成属于自己这部分独立的功能，那么之后整个页面的管理和维护就变的非常容易了。

## 2. Vue的组件化思想

* **组件化是Vue.js中的重要思想**
* 它提供了一种抽象，让我们可以开发出一个个独立可复用的小组件来构造我们的应用。
* 任何的应用都会被抽象成一棵组件树。
* 组件化思想的应用：
* 有了组件化的思想，我们在之后的开发中就要充分的利用它。
* 尽可能的将页面拆分成一个个小的、可复用的组件。
* 这样让我们的代码更加方便组织和管理，并且扩展性也更强。

## 3. 注册组件的基本步骤

调用Vue.extend()方法创建组件构造器

* **组件的使用分成三个步骤：**
* 创建组件构造器

调用Vue.component()方法注册组件

* 注册组件
* 使用组件

在Vue实例的作用返回内使用组件

## 4. 组件化的基本使用过程

|  |
| --- |
| **严格按照（3）中的步骤来进行操作即可** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Vue 组件化的基本使用过程</title>  </head>  <body>  <!-- 3.use vue to show -->  <div id="div1001">  <h2>我是正常的标题</h2>  <!-- 3.3 使用自定义的模板 -->  <my-cpn></my-cpn>  <my-cpn></my-cpn>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  /\*\*  \* 3.1 创建组件构造器Vue.extend()方法  \* 参数是一个对象：对象中有一个属性 template,值为模板字符串  \* 注意 ：  \* 这里为了好看，使用了ES6 的语法，  \* 模板字符串： 左上角的那个小逗号，可以换行的操作  \*/  const cpnComponent = Vue.extend({  template:`  <div>  <h2>我是模板中的标题</h2>  </div>  `  });  /\*\*  \* 3.2 注册组件 Vue.component()方法  \* 两个参数 ：  \* 第一个参数 ： 你自己定义的这个组件的名称  \* 第二个参数 ： 你刚才创建的那个组件构造器  \*  \* 实际上这里注册的就是一个全局的组件  \*/  Vue.component('my-cpn',cpnComponent);  // 3.3 使用组件 ： 当然是在html中使用组件的了  //定义一个交给Vue管理的对象，注意这个位置，一定是要在 组件创建的下面！  let obj = new Vue({  el:'#div1001',  data:{  message : 'Hello World!'  }  });  </script>  </body>  </html> |

## 5. 全局组件和局部组件

### (1) 全局组件：

|  |
| --- |
| **1. 意味着可以在多个Vue的实例下面进行使用（但是，一般情况下我们在开发的时候置会创建一个实例）**  **2. Vue.component() 的注册方法就是注册的全局组件。** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>全局组件</title>  </head>  <body>  <!-- 3.use vue -->  <div id="div1002\_a" style="border: 1px solid green;">  <h2>第一个Vue实例</h2>  <my-cpn></my-cpn>  </div>  <div id="div1002\_b" style="border: 1px solid rgb(0, 30, 128);">  <h2>第二个Vue实例</h2>  <my-cpn></my-cpn>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  // 3.1 创建组件的构造器  const cpnC = Vue.extend({  template:`  <div>  <h2>我是全局组件中的标题</h2>  </div>  `  });  // 3.2 注册全局组件  Vue.component('my-cpn',cpnC);  // 3.3 使用全局组件 : 当然是在html中进行使用  //2.1 create object 01  let obj01 = new Vue({  el:'#div1002\_a',  });  //2.2 create object 02  let obj02 = new Vue({  el:'#div1002\_b',  });  </script>  </body>  </html> |

### (2) 局部组件（用的多） ：

|  |
| --- |
| **1.意味着只可以在注册的那个Vue实例中才可以使用**  **2.在具体的某一个实例中添加属性 ： components:{标签名：组件构造器}** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>局部组件</title>  </head>  <body>  <!-- 3.use vue -->  <div id="div1003\_a">  <h2>第一个Vue实例</h2>  <!-- 因为组件注册在了这里面，所以可以使用 -->  <my-cpn></my-cpn>  </div>  <div id="div1003\_b">  <h2>第二个Vue实例</h2>  <!-- 因为组件没有在这里注册，所以不可以使用！ -->  <my-cpn></my-cpn>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  //3.1 创建组件构造器  const conC = Vue.extend({  template:`  <div>  <h2>我是局部组件中的标题</h2>  </div>  `  });  //2.1 create object 01  let obj01 = new Vue({  el:'#div1003\_a',  //3.2 注册局部组件，只能在这个vue实例中才可以使用,components属性！  components:{  // 前面的是 组件的标签名，后面的是标签的构造器  'my-cpn':conC  }  });  //2.2 create object 02  let obj02 = new Vue({  el:'#div1003\_b',  });  </script>  </body>  </html> |

## 6. 父组件和子组件 基本操作

|  |
| --- |
| **1.**定义的顺序一定是 先定义子组件，再定义父组件，也就是父组件一定要在子组件的下面写，否则会找不到子组件；  **2.**在注册组件的时候，**不要用大写字母，**可以种短横线中间连接，这个与vue的驼峰命名法有关，因为它根本就**不支持驼峰命名法**；  **3.在组件的构造器中也可以注册组件，这就是形成父子组件的关键操作1；**  **4.在父组件中使用注册的子组件的标签，这是父组件使用子组件的关键操作2;**  **5.Vue实例可以看成是一个根组件** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>父子组件的基本操作</title>  </head>  <body>  <!-- 3.show data -->  <div id="div1004">  <h2>这个是 父子组件的根div</h2>  <!-- 下面只使用父组件的标签，则，会展示父子组件两个的内容 -->  <cpn-father></cpn-father>  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>    //3.1 创建子组件的构造器  const cpnSon = Vue.extend({  template:`  <div>  <h2>我是儿子组件</h2>  </div>  `  });  //3.2 创建父组件的构造器  const cpnFather = Vue.extend({  template:`  <div>  <h2>我是爸爸组件</h2>  <!-- 在父组件中使用子组件（关键操作2）-->  <cpn-son></cpn-son>  </div>  `,  // 在 组件构造器中也可以注册组件！（关键操作1）  components:{  'cpn-son':cpnSon  }  });  //2.1 创建实例对象并注册 一个 父组件  let obj = new Vue({  el:'#div1004',  data:{  message : 'Hello World!',  },  //这注册了一个爸爸组件  components:{  'cpn-father':cpnFather,  }  });  </script>  </body>  </html> |

## 注册组件的语法糖写法 开发常用

|  |
| --- |
| **实际上是省去了 Vue.extend()的写法，直接把原来的模板对象放到注册对象的位置上** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>注册组件的语法糖</title>  </head>  <body>  <!-- 3.use vue -->  <div id="div1005">  <cpn1></cpn1> <!—全局组件-->  <cpn2></cpn2> <!—局部组件-->  </div>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  //3.1 直接注册一个**全局组件**，把原来Vue.extend()中的对象直接拿过来即可！  Vue.component('cpn1',{  template:`  <div>  <h2>我是全局组件</h2>  </div>  `  })  //2.1 创建vue实例对象，并注册组件  let obj = new Vue({  el:'#div1005',  //3.2 直接注册一个局部组件，将组件这个对象直接写在这里  components:{  'cpn2':{  template:`  <div>  <h2>我是局部组件</h2>  </div>  `  }  }  });  </script>  </body>  </html> |

## 组件的模板抽离写法

|  |
| --- |
| 1. 使用script标签，type=”text/x-template”,并指定id属性，在注册组件的时候，直接给定这个id的属性即可；  2.使用template标签， 并指定id属性 |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>模板分离写法的两种方式</title>  </head>  <body>  <!-- 3.show 模板 -->  <div id="div1006">  <cpn1></cpn1> <!-- script 标签定义模板 的组件 -->  <cpn2></cpn2> <!-- template 标签定义模板 的组件-->  </div>  <!-- 3.1 使用script标签创建模板,一定要指定id属性,且type属性的值要记住！-->  <script type="text/x-template" id="cpn1">  <div>  <h2>这个是script 标签的模板</h2>  </div>  </script>  <!-- 3.2 使用template标签创建模板,一定要指定id属性 -->  <template id="cpn2">  <div>  <h1>这个是 template标签的模板</h1>  </div>  </template>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div1006',  components:{  //3.3 使用语法糖的方式，指定对应的模板id,进行模板绑定的操作  'cpn1':{  template:'#cpn1'  },  'cpn2':{  template:'#cpn2'  }  }  });  </script>  </body>  </html> |

## 组件中的数据存放的问题

|  |
| --- |
| **1.组件中不能使用Vue实例的data属性中的数据；**  **2.组件对象有自己的data属性，也有methods属性；**  **3.组件的data属性必须是一个函数，原因 ： 组件是要重用的，每次调用一个函数就会返回一个新的对象，这样数据就不会出现脏数据！；**  **4.而且这个函数的返回值必须是一个对象，对象的内部就保存着数据。** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>组件的data问题</title>  </head>  <body>  <!-- 本操作封装一个计数的组件，并且重用起来 -->  <!-- 4.展示 可重用的组件 -->  <div id="div1007">  <h1>这里是Vue的实例</h1>  <cpn-jishu></cpn-jishu>  <hr>  <cpn-jishu></cpn-jishu>  <hr>  <cpn-jishu></cpn-jishu>  </div>  <!-- 1.import vue : 注意这个的顺序，这个应该是有文档的加载顺序的-->  <script src="../js/vue.js"></script>  <!-- 3.2 编写组件的模板 -->  <template id="cpn1">  <div>  <h2>当前的计数 ： {{counter}}</h2>  <button @click="jianjian()">-</button>  <button @click="jiajia()">+</button>  </div>  </template>  <!-- 3.1 注册一个组件，使用分离式的写法,为了方便阅读代码，使用全局注册组件 -->  <script>  Vue.component('cpn-jishu',{  template:'#cpn1',  data(){ // 这里的数据需要定义成一个方法的形式，并且返回一个对象  return {  counter:0  }  },  methods:{  jianjian(){  this.counter--;  },  jiajia(){  this.counter++;  }  }  })  </script>      <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div1007',  });  </script>  </body>  </html> |

## 父子组件的通信（简介）

* 前提 ：子组件是不能引用父组件或者Vue实例的数据的。
* 但是，在开发中，往往一些数据确实需要冲上层传递到下层：
* 比如在一个页面中，我们从服务器请求到了很多的数据。
* 其中一部分数据，并非是我们整个页面的大组件来展示的，而是需要下面的子组件进行展示。
* 此时，并不会让子组件再次发送一个网络请求，而是直接让**大组件（父组件）**将数据传递给**小组件（子组件）**。
* **如何进行父子组件间的通信呢？Vue官方提到：**
* **通过props向子组件传递数据**
* **通过事件向父组件发送消息**

## 父传子-props 基本用法（很常用）

* 在组件中，使用选项props来声明需要从父级接收到的数据。
* Props的值有两种方式：
* **字符串数组**，数组中的字符串就是传递时子组件的属性名称，**使用v-bind绑定即可**
* **对象**，对象可以设置传递时的类型，也可以设置默认值等。
* Props的驼峰命名的问题：
* 答：vue中根本不认识 驼峰命名法，
* 假如你在 props中定义了一个驼峰命名法的属性，**myChildName,**
* 那么，你在 绑定传值的时候需要这样来写 **v-bind:my-child-name=”值**
* 这个就是 vue的一个设计，就是这样的，所**以命名的时候可以直接全都写小写就可以了”**

|  |
| --- |
| **这个父子之间的传递的操作确实有点麻烦，不过还是要认真的操作，很常用！** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>父传子 props</title>  </head>  <body>  <!-- 这个练习中，将vue实例作为父组件，自定义的组件作为子组件 -->  <!-- 4. 界面要展示的内容：也就是父组件的模板 -->  <div id="div1008">  <h2>这个是根组件</h2>  <!-- 使用子组件并且将父组件中的数据传入：使用v-bind绑定属性 -->  <cpn-son :sonmessage="message" :sonmovies="movies"></cpn-son>  <!-- 使用默认值的操作！ -->  <cpn-son :sonmessage="message"></cpn-son>  <!-- 使用自己的验证 -->  <cpn-son :sonvalidator="myNumber"></cpn-son>  </div>  <!-- 3.定义子组件的 模板 -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h1>这里是子组件的标签</h1>  <!-- 3.1 在模板中可以 直接使用子组件自己的属性 注册时候 -->  <h2>{{sonmessage}}</h2>  <h3>{{sonmovies}}</h3>  <h3>{{sonvalidator}}</h3>  </div>  </template>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div1008',  //2.1 定义父组件的数据，用来传递给子组件  data:{  message:'你好呀子组件！',  movies:['火影忍者','海贼王','死神','钢之炼金术师'],  myNumber:1000  },  //2.2 注册子组件  components:{  'cpn-son':{  template:'#cpnson',  // 下面的就是子组件声明的属性，可以用来接收父组件传递过来的值  // 2.2.1 方式一 ： 使用数组的方式，数组元素就是组件的属性，开发中**很少用**使用这种形式  //props:['sonmessage','sonmovies']  // 2.2.2 方式二 ： 使用对象的方式，可以 限制属性类型+默认值，开发中用的多  props:{  // 1. 可以限制属性的类型  sonmessage:String,  // 2. 也可以是一个对象  sonmovies:{  // 2.1. 限制属性的类型  type:Array,  // 2.2. 指定默认值，在没有传的情况下会有默认值,  //当类型是 Object 或者 Array 的时候，需要写成函数的形式，直接返回对象或者数组  default:()=>{  return ['a','b']  },  //2.3 require 属性，true的时候是必须要传的  require:false  },  // 3. 对象方式中，也可以自己进行验证，自己写函数  sonvalidator:{  type:Number,  // 验证函数，但是如何获取这个验证的结果呢？还不知道  validator:(value)=>{  // 这个值必须要大于100  return value > 100  }  }  }  }  }  });  </script>  </body>  </html> |

## 子传父-自定义事件 $emit（这个要复杂一点）

|  |
| --- |
| **这个通过事件来执行的操作！也是要必须要掌握的。从上到下都是重点！** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>子组件向父组件传递事件</title>  </head>  <body>  <!-- 4.创建父组件的模板，并使用子组件的模板 -->  <div id="div1009">  <h2>这里是父组件的模板</h2>  <!-- 4.1 在这里进行子组件的事件监听  1.监听的事件名称 ：child-btn-click 是由子组件发射事件的时候自己定义的  2.关于处理方法的参数：  不需要写括号，只写一个名字即可！  不需要写参数，自己就可以传递过来！（这个很难让人接受 啊）  尽管很难让人接受，但确实就是这样写，无论有多少个，在接收的时候，直接接收即可  3.关于处理方法 ： handleChildBtnClick ，这个当然是要定义在父组件中，因为要在负组件中进行处理的  -->  <cpn1 @child-btn-click="handleChildBtnClick"></cpn1>  </div>  <!-- 3.2 创建子组件模板 -->  <template id="soncpn">  <!-- 3.2.1 当有多个标签的时候，都要有一个根标签div,就是注意一下这个写法 -->  <div style="border: 1px solid red;">  <div>这里是子组件模板</div>  <!--  3.2.2 点击button的时候， ---- 这个地方需要进行click监听  向父组件发射一个事件 ---- 这个地方需要进行事件发射操作 $emit    3.2.3 这个事件是子组件中的，所以应该定义在子组件中  -->  <button v-for="item in buttons"  @click="sendMessage(item)">  {{item.name}}  </button>  </div>  </template>  <!-- 3.1 创建子组件对象 -->  <script>  const soncpn = {  template:'#soncpn',  data(){  // 1. 数据是一个对象的数组，  return {  buttons:[  {id:1001,name:'button01'},  {id:1002,name:'button02'},  {id:1003,name:'button03'},  {id:1004,name:'button04'},  {id:1005,name:'button05'}  ]  }  },  // 2. 定义 子组件的事件  methods:{  sendMessage(item){  console.log('子组件要准备发射事件了 ：',item.name);  **// 下面是发射事件的核心写法**:  // 参数一 ： 发射事件的名称，这个需要在父组件模板中进行监听的  // 参数二 ： 事件中要传递的数据  this.$emit('child-btn-click',item,'abc');  }  }  }  </script>    <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create object with vue -->  <script>  let obj = new Vue({  el:'#div1009',  //2.1 注册子组件  components:{  'cpn1':soncpn  },  //2.2 处理监听的子组件的方法  methods:{  handleChildBtnClick(item,msg){  console.log('爸爸已经监听到了子组件中的按钮点击事件');  console.log('传递过来的对象是 ： ',item,msg);  }  }    });  </script>  </body>  </html> |

## 父子组件的通信案例(结合数据的双向绑定)

|  |
| --- |
| 需求：   1. 父组件向子组件传递数据值进行显示：num1,num2 🡪 sonnum1,sonnum2 2. 子组件中要有两个input输入框，分别对应子组件中的两个属性：sonnum1,sonnum2 3. 两个input输入框需要实现数据的**双向绑定**，（**这个地方有一个新的知识点：vue的推荐做法，通过子组件的data来创建数据属性与sonnum1,sonnum2进行绑定，然后进行修改的时候，就不会有警告了**） 4. 当数据双向绑定的时候，同时修改父组件中的num1,num2(子组件向父组件进行传值)   **综上所述，这个需求就是父传子，子传父的一个综合的应用**。 |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>综合案例 ：结合数据的双向绑定</title>  </head>  <body>  <!-- 4. 创建父组件的HTML模板-->  <div id="div1010" style="border: 1px solid green;">  <h1>我是父组件</h1>  <h2>爸爸中的 num1 : {{num1}}</h2>  <h2>爸爸中的 num2 : {{num2}}</h2>  <!-- 直接使用子组件 :  注意 ： 在向父组件发射事件的时候，父组件中的监听函数在这里是不需要写小括号的  只需要写一个名字即可。  -->  <cpnson :sonnum1="num1" :sonnum2="num2"  **@dnumber1-change="number1Change"**  **@dnumber2-change="number2Change"**  ></cpnson>  </div>  <!-- 3.2 创建子组件的HTML模板 -->  <template id="cpnsonhtml">  <div style="border:1px solid red">  <h2>我是子组件</h2>  <h3>子组件 sonnum1 属性 值 ：{{sonnum1}}</h3>  <h3>子组件 sonnum2 属性 值 ：{{sonnum2}}</h3>  <!-- (1) 下面的写法虽然可以实现子组件中的数据双向绑定，  但是会有一个警告出来，vue推荐的做法是：  1.通过data创建新的数据对象  2.数据对象与属性进行绑定  3.数据双向绑定绑定的是 数据对象，而不是属性！  -->  <!--  <input type="text" v-model="sonnum1">  <input type="text" v-model="sonnum2">  -->  <!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->  <!-- (2) 下面的写法虽然实现了双向绑定，  但是，是与dnumber1/dnumber2进行的数据绑定，  而不是 与 sonnum1,sonnum2 实现的绑定  -->  <!--  dnumber1 : <input type="text" v-model="dnumber1">  <br>  dnumber2 : <input type="text" v-model="dnumber2">  -->  <!-- (3) 下面使用 v-bind + input事件绑定的写法进行 实现数据的双向绑定操作 -->  dnumber1 : <input type="text" :value="dnumber1" @input="changednumber1($event)">  <br>  dnumber2 : <input type="text" :value="dnumber2" @input="changednumber2($event)">  </div>  </template>  <!-- 3.1 声明一个子组件对象 -->  <script>  const cpnson = {  template:'#cpnsonhtml',  // 3.1.1 创建接收父组件数据的属性  props:{  sonnum1:Number,  sonnum2:Number  },  // 3.1.2 创建子组件中的数据对象，与属性进行绑定，实现数据的双向绑定操作  data(){  return {  // 直接把 两个属性给这两个数据  dnumber1:this.sonnum1,  dnumber2:this.sonnum2  }  },  // 3.1.3 创建 响应事件方法  methods:{  changednumber1(event){  //1. 修改绑定的值  this.dnumber1 = parseInt(event.target.value);  console.log(this.dnumber1);    //2.向父组件发射事件  this.$emit('dnumber1-change','hello',this.dnumber1);    },  changednumber2(event){  this.dnumber2 = parseInt(event.target.value);  this.$emit('dnumber2-change','hello',this.dnumber2);  }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1010',  data:{  num1:1,  num2:2  },  //2.1 注册子组件  components:{  'cpnson':cpnson  },  //2.2 监听子组件发送过来的事件的方法  methods:{  number1Change(aa,value){  this.num1 = value;  },  number2Change(aa,value){  this.num2 = value;  }  }  });  </script>  </body>  </html> |

## 父子组件的通信案例-watch属性实现（了解）

|  |
| --- |
| **子组件中有个属性叫做watch:** 这个属性可以监听 组件中的属性的变化，一旦发生了变化就会触发事件，详细操作请看 下面的代码即可。 |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>父子组件通信 案例（watch）</title>  </head>  <body>  <!-- 4. 创建父组件的HTML模板-->  <div id="div1010" style="border: 1px solid green;">  <h1>我是父组件</h1>  <h2>爸爸中的 num1 : {{num1}}</h2>  <h2>爸爸中的 num2 : {{num2}}</h2>  <!-- 直接使用子组件 :  注意 ： 在向父组件发射事件的时候，父组件中的监听函数在这里是不需要写小括号的  只需要写一个名字即可。  -->  <cpnson :sonnum1="num1" :sonnum2="num2"  @dnumber1-change="number1Change"  @dnumber2-change="number2Change"  ></cpnson>  </div>  <!-- 3.2 创建子组件的HTML模板 -->  <template id="cpnsonhtml">  <div style="border:1px solid red">  <h2>我是子组件</h2>  <h3>子组件 sonnum1 属性 值 ：{{sonnum1}}</h3>  <h3>子组件 sonnum2 属性 值 ：{{sonnum2}}</h3>    <!-- (2) 下面的写法虽然实现了双向绑定，  但是，是与dnumber1/dnumber2进行的数据绑定，  而不是 与 sonnum1,sonnum2 实现的绑定  -->  dnumber1 : <input type="text" v-model="dnumber1">  <br>  dnumber2 : <input type="text" v-model="dnumber2">  </div>  </template>  <!-- 3.1 声明一个子组件对象 -->  <script>  const cpnson = {  template:'#cpnsonhtml',  // 3.1.1 创建接收父组件数据的属性  props:{  sonnum1:Number,  sonnum2:Number  },  // 3.1.2 创建子组件中的数据对象，与属性进行绑定，实现数据的双向绑定操作  data(){  return {  // 直接把 两个属性给这两个数据  dnumber1:this.sonnum1,  dnumber2:this.sonnum2  }  },  /\*\*  \* 3.1.3 创建 watch 属性，当 属性发生变化的时候，就会触发的事件,  \* 事件名称就叫属性名,  \* 参数一 ： 新的值，  \* 参数二 ： 旧的值（通常不用）  \* 实际上这个属性很方便使用  \*/  watch:{  dnumber1(newValue){  console.log(newValue);  console.log(this.dnumber1);  this.$emit('dnumber1-change','hello',parseInt(this.dnumber1));    },  dnumber2(newValue){  console.log(newValue);  this.$emit('dnumber2-change','hello',parseInt(this.dnumber2));  }  }    }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1010',  data:{  num1:1,  num2:2  },  //2.1 注册子组件  components:{  'cpnson':cpnson  },  //2.2 监听子组件发送过来的事件的方法  methods:{  number1Change(aa,value){  this.num1 = value;  },  number2Change(aa,value){  this.num2 = value;  }  }  });  </script>  </body>  </html> |

## 15父子组件之间的相互访问（简介）

* **有时候，我们需要父组件直接访问子组件，子组件直接访问父组件，或者子组件直接访问根组件。**
* **父组件访问子组件**：使用**$children(获取的是全部的子组件数组)**或者**$refs(给子组件一个ref属性，则获取指定的具体某一个的子组件)**
* **子组件访问父组件**：使用**$parent**

## 16.父组件访问子组件01-$children(不常用)

|  |
| --- |
| **获取到的是所有的子组件的一个 数组**， |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>$children</title>  </head>  <body>  <!-- 4.创建父组件模板 -->  <div id="div1012" style="border: 1px solid green;">  <h2>我是爸爸组件</h2>  <!-- 使用子组件 -->  <cpnson></cpnson>  <cpnson></cpnson>  <cpnson></cpnson>  <!-- 点击按钮调用子组件中的方法 -->  <button @click="callChildren()">调用子组件的方法$children</button>  </div>  <!-- 3.2 创建子组件的模板 -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我就是子组件，来调用我的方法啊</h2>  </div>  </template>  <!-- 3.1 创建一个子组件对象 -->  <script>  const cpnson = {  template:'#cpnson',  methods:{  // 创建一个子组件的方法，供父组件通过子组件进行调用  showMessage(){  console.log('我是子组件cpnson 的方法');  }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el : '#div1012',  // 2.1 注册子组件  components:{  'cpnson':cpnson  },  // 2.2 编写按钮响应事件，调用子组件的方法  methods:{  callChildren(){  // this.$children 是所有的子组件的数组  console.log(this.$children);  // 遍历所有的子组件，并调用方法 showMessage()  for (const son of this.$children) {  son.showMessage();  }  }  }  });  </script>  </body>  </html> |

## 17.父组件访问子组件02-$refs(常用)

|  |
| --- |
| **这个用的简单，并且能够获取到具体的某一个子组件** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>$refs</title>  </head>  <body>  <!-- 4.创建父组件模板 -->  <div id="div1012" style="border: 1px solid green;">  <h2>我是爸爸组件</h2>  <!-- 使用子组件 -->  <cpnson></cpnson>  <!-- 在某一个子组件上面添加ref属性，则this.$refs 就会获取到这个对象了 -->  <cpnson ref="aaa"></cpnson>  <cpnson ref="bbb"></cpnson>  <!-- 点击按钮调用子组件中的方法 -->  <button @click="callChildren()">调用子组件的方法$refs</button>  </div>  <!-- 3.2 创建子组件的模板 -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我就是子组件，来调用我的方法啊</h2>  </div>  </template>  <!-- 3.1 创建一个子组件对象 -->  <script>  const cpnson = {  template:'#cpnson',  methods:{  // 创建一个子组件的方法，供父组件通过子组件进行调用  showMessage(){  console.log('我是子组件cpnson 的方法');  }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el : '#div1012',  // 2.1 注册子组件  components:{  'cpnson':cpnson  },  // 2.2 编写按钮响应事件，调用子组件的方法  methods:{  callChildren(){  // this.$refs 会获取到有ref属性的对象  console.log(this.$refs);  // 并调用有ref=”bbb” 属性的子组件的方法 showMessage()  this.$refs**.bbb**.showMessage();    }  }  });  </script>  </body>  </html> |

## 18.子组件访问父组件 -$parent + $root（开发中不建议使用！）

|  |
| --- |
| 虽然可以操作，但是不建议使用。因为子组件都是独立的，这样就减少了组件的耦合性。 |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>子组件访问父组件</title>  </head>  <body>  <!-- 4.create root template -->  <div id="div1014" style="border: 1px solid green;">  <h2>我是爸爸组件，我也是根组件</h2>  <!-- 使用子组件 -->  <cpnson></cpnson>  </div>  <!-- 3.2 create a son component template-->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我是子组件</h2>  <button @click="btnClick()">获取父组件的操作</button>  </div>  </template>  <!-- 3.1 create a son component object -->  <script>  const cpnson = {  template:'#cpnson',  methods:{  btnClick(){  // 1. 通过 $parent来获取父组件  console.log(this.$parent);  // 2. 通过 $root 来获取根组件  console.log(this.$root);  this.$root.sayHello();    }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1014',  //2.1 创建父组件的方法，用来测试子组件的调用  methods:{  sayHello(){  console.log('Hello 我是爸爸组件');  }  },  //2.2 注册子组件  components:{  'cpnson':cpnson  }  });  </script>  </body>  </html> |

## 19.Slot-插槽的基本使用

* **组件的插槽**：
* 组件的插槽是为了让我们的组件更具有扩展性。
* 让使用者可以决定组件内部的一些内容到底展示什么。
* **例子：移动开发中的导航条**：
* 移动开发 中，几乎每个页面都有导航条，
* 导航栏我们封装成一个组件，如nav-bar,
* 一旦有了这个组件，我们就可以在多个页面中复用了

|  |
| --- |
| 1. **插槽的基本使用，在组件中添加标签<slot></slot>** 2. **也可以指定默认值：<slot><h2>我是插槽</h2></slot>** 3. **当同时写多个<slot></slot>的时候，这多个插槽都会被替换！（在不指定具名插槽的情况下）** 4. **在指定具名插槽的情况下，插槽内的内容跟着名字而绑定** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>slot 的基本使用</title>  </head>  <body>  <!-- 4.create root component template -->  <div id="div1015" style="border: 1px solid green;">  <!-- 4.1 直接使用子组件 -->  <cpnson></cpnson>  <!-- 4.2 在子组件中使用插槽，填充内容 -->  <cpnson>  <div style="border:1px solid rgb(218, 16, 184);width: 50vw;">  <h2>这里是通过插槽显示的div标签</h2>  </div>  <h1>这个是通过插槽写入的h1标签</h1>  </cpnson>  </div>  <!-- 3.2 create a son component template -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我是子组件</h2>  <!-- 下面定义一个插槽的位置 -->  <slot ></slot>  <!-- 下面是定义了一个默认值的插槽位置 -->  <slot ><button>默认值的按钮</button></slot>  </div>  </template>  <!-- 3.1 create a son component object -->  <script>  const cpnson = {  template:'#cpnson',  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obje = new Vue({  el:'#div1015',  //2.1 注册子组件  components:{  'cpnson':cpnson  }  });  </script>  </body>  </html> |

## 20.Slot-具名插槽的使用

|  |
| --- |
| **具名插槽：就是slot标签添加了name属性，**  **使用的时候，在元素中 添加 slot = “slot的名字” 即可**  **通过这种方式进行绑定插槽**  **解决有多个插槽到底放在哪个插槽里的问题。** |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>具名插槽的使用</title>  </head>  <body>  <!-- 4.create root template -->  <div id="div1016">  <h1>我是爸爸组件</h1>  <!-- 4.1 下面使用 默认的子组件 -->  <cpnson></cpnson>  <!-- 4.2 下面使用 具名插槽，指定位置进行替换操作  **通过在标签中添加 slot="插槽名" 的方式指定插槽！**  -->  <cpnson>  <span slot="slot2">我来替换插槽2的原来内容</span>  </cpnson>  </div>  <!-- 3.2 create a son component template -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我是子组件</h2>  <!-- 下面定义三个具名插槽 -->  <slot name="slot1"><span>我是插槽1</span></slot>  <slot name="slot2"><span>我是插槽2</span></slot>  <slot name="slot3"><span>我是插槽3</span></slot>  </div>  </template>  <!-- 3.1 create a son component object -->  <script>  const cpnson = {  template:'#cpnson'  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1016',  //2.1 注册子组件  components:{  'cpnson':cpnson  }  });  </script>  </body>  </html> |

## 21.编译作用域的概念：就是在哪个模板内，用的就是哪个组件或者vue实例的属性。

|  |
| --- |
| **Vue的官方准则：**  父组件模板中的所有东西都会在父级作用域内进行编译；  子组件模板的所有东西都会在子级作用域内进行编译。 |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>编译的作用域</title>  </head>  <body>  <!-- 4.create root template -->  <div id="div1017">  <!-- 1.使用子组件 -->  <cpnson></cpnson>  <!-- 2.使用子组件，并通过 v-show 进行控制  因为这个子组件是在 root 根模板中的，  **所以这个地方的 isShow 的数据是 根组件中的数据，也就是 true**  而不是 子组件中的 那个isShow,因此它是可以展示的  这个地方要特别注意一下，也是一个比较关键的问题。  -->  <cpnson v-show="isShow"></cpnson>  </div>  <!-- 3.2 create a son component template -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我是子组件</h2>  </div>  </template>  <!-- 3.1 create a son component object -->  <script>  const cpnson = {  template:'#cpnson',  // 子组件对象也有自己的数据  data(){  return {  **// 这里起的名字和父组件中的是一样的**  isShow: false  }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1017',  //2.1 创建自己的数据  data:{  // 这里起的名字和子组件中的是一样的  isShow:true  },  //2.2 注册子组件  components:{  'cpnson':cpnson  }  });  </script>  </body>  </html> |

## 22.作用域插槽（有点难理解）

* **一句话总结**：
* 父组件替换插槽的标签，但是内容由子组件来提供。
* **我们先提出一个需求**：
* 子组件中包括一个数据数组：books:[‘西游记’,’三国演义’,’水浒传’,’红楼梦’]
* 需要在多个界面进行展示：
* 某个界面是以水平方向一一展示的
* 某个界面是以垂直方向一一展示的
* 某个界面是直接展示这个数据就🆗了
* 内容在子组件中，希望父组件告诉我们该如何展示，怎么做？
* 此时，使用 作用域插槽slot就可以了。

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>作用域插槽的使用</title>  </head>  <body>  <!-- 4.create root template -->  <div id="div1018">  <h2>我是爸爸组件</h2>  <!-- 1.使用默认的插槽内容 -->  <cpnson></cpnson>  **<!-- 2.使用作用域插槽**  **2.0 子组件中必须要添加一个 template 标签，或者其他的标签也可以**  **2.1 使用 slot-scope="booklist" 来指向子组件的插槽,booklist 这个名字可以随便起**  **2.2 然后通过获取子组件的插槽的自定的那个属性abc ,来获取到数据值**  **2.3 最后，就可以拿着这个数据值进行为所欲为的展示了**  **-->**  <cpnson >  <template slot-scope="booklist\_slot">  <ul>  <li v-for="book in booklist\_slot.abc">{{book}}</li>  </ul>  </template>    </cpnson>  </div>  <!-- 3.2 create a son component template -->  <template id="cpnson">  <div style="border: 1px solid red;">  <h2>我是子组件</h2>  <!-- 下面定义一个插槽  1.给这个插槽设置默认值：直接把books数组展示出来  2.给这个插槽 绑定一个属性，属性名随便起：比如 :abc="books"  属性值 绑定自己的那个books数组  3.这个作用域插槽就定义完成！  -->  <slot :abc="books">  初始化情况 ： books 直接使用自己的属性: {{books}}  </slot>  </div>  </template>  <!-- 3.1 create a son component object -->  <script>  const cpnson = {  template:'#cpnson',  data(){  return {  books:['西游记','水浒传','三国演义','红楼梦']  }  }  }  </script>  <!-- 1.import vue -->  <script src="../js/vue.js"></script>  <!-- 2.create instance with vue -->  <script>  let obj = new Vue({  el:'#div1018',  //2.1 注册子组件  components:{  'cpnson':cpnson  }  });  </script>  </body>  </html> |

# 四、ES6的模块化

## 1.模块化的核心：导入与导出

|  |
| --- |
| **导出 ： export xxx**  **导入 ： import {xxx} from ‘xxx.js’** |
| **Html 文件中直接导入js文件作为模块的**  **<!-- 1.引入 js文件 : 注意要使用模块的类型进行导入 -->**  **<script src="developer01.js" type="module"></script>**  **<script src="developer02.js" type="module"></script>** |

## 2.导出变量：

|  |
| --- |
| /\*\*  \* 知识点一 ： 导出变量 export  \*  \* 方式一 ： 定义时直接导出  \* 方式二 ： 先定义，后直接导出  \*/ |

* **方式一 ： 在定义时导出**

|  |
| --- |
| //方式一 ： 定义时直接导出  export let name = "developer01"  export let age = 18 |

* **方式二 ：同一导出**

|  |
| --- |
| //方式二 ： 先定义变量，后统一导出  let name2 = "developer01"  let age2 = 28  let height = 25.63  export {name2,age2,height} |

## 3.导出函数

|  |
| --- |
| /\*\*  \* 知识点二 ： 导出函数 export  \*  \* 方式一 ： 定义时直接导出  \* 方式二 ： 先定义，后直接导出  \*/ |

* **方式一 ： 定义时导出**

|  |
| --- |
| //方式一 ： 定义时直接导出,这里还时必须要写function关键字  export function sayHello(namestr){  console.log('Hello '+namestr);  }; |

* **方式二 先定义，后导出**

|  |
| --- |
| //方式二 ： 先定义，再导出  function sayHello2(namestr){  console.log('Hello2 '+namestr);  }  export {sayHello2} |

## 4.导出类

|  |
| --- |
| /\*\*  \* 知识点三 ： 导出类 export  \*  \* 方式一 ： 定义时直接导出  \* 方式二 ： 先定义，后直接导出  \*/ |

* **方式一 ： 定义时导出**

|  |
| --- |
| //方式一 ： 定义时直接导出,ES6中可以直接使用class来定义类  export class Student{  id;  name;  age;  run(){  console.log('这个学生会跑');  }  } |

* **方式二 ： 先定义后导出**

|  |
| --- |
| //方式二 ： 先定义，后导出  class School{  name;  sayHello(){  console.log('学校对你say Hello');    }  }  export {School} |

## 5.导出默认 export default

**在某些情况下，一个模块中包含某个功能，我们并不希望给这个功能命名，而且可以让导入者自己来进行命名，**

**此时就可以使用 export default ,**

**并且，一个模块中只能由一个 export default**

**导入时，可以随便起名字**

|  |
| --- |
| /\*\*  \* 知识点四 ： 导出默认 export  \*  \* export default 变量/函数/对象/类 等等的，  \* 都可以进行默认导出  \*/ |

|  |
| --- |
| const baseNumber = 10000  // 下面就是把这个baseNumber给导出默认  export default baseNumber |

|  |
| --- |
| **下面时导入默认的操作：** |
| //3.导入默认的操作，自己随便起个名字就可以  import **bignumber** from './developer01.js';  console.log(bignumber); |

## 6.导入操作 import

|  |
| --- |
| /\*\*  \* 知识点一 ： 导入模块中的内容 import  \* 格式1 ： import {xxx} from 'xxx';  \* 格式2 ： import \* as aaa from 'xxx'  \* 一次性全部导入进来;  \* 使用的时候，通过 aaa.属性名 的方式进行调用  \*  \*/ |

|  |
| --- |
| **import {name,name2,age,age2,sayHello,sayHello2,Student,School} from './developer01.js';**  **import \* as dev01 from './developer01.js';**  //1.直接打印变量进行测试一下  console.log(name);  //2.调用引入的函数进行测试一下  sayHello('xiaohong');  sayHello2('xiaoqiang');  //3.使用引入的类创建对象测试一下  let student01 = new Student();  student01.run();  let school01 = new School();  school01.sayHello();  //3.导入默认的操作  import bignumber from './developer01.js';  console.log(bignumber); |

## \*在调试的时候，出现了一个跨域错误的问题，此问题的解决方案是 ： VSCode安装插件： Live Server ，然后，右键html选择 Open with Live Server即可

# 五、webpack详解

## 1.认识webpack

* Webpack其中一个核心就是让我们可能进行模块化开发，并且会帮助我们处理模块之间的依赖关系。
* 不仅仅是JavaScript文件，css,json,图片等文件，在webpack中都可以作为模块来使用
* 如何理解打包？
* 理解了webpack可以帮助我们进行模块化，并且处理模块间的各种复杂的依赖关系，打包的概念就非常好理解了。
* 就是将webpack中的各种资源模块进行打包合并成一个或多个包（Bundle）。
* 并且在打包构成中对资源进行处理：比如压缩图片，将sacss转换成css,将ts转成js等

## 2.webpack的安装

### (1) 需要先安装node环境

**具体的安装步骤，请查阅《02.nodejs安装与配置.docx》文档**

### (2) 全局安装webpack

|  |
| --- |
| **Npm install -g** [**webpack@3.6.0**](mailto:webpack@3.6.0) **(下载慢时使用cnpm)** |

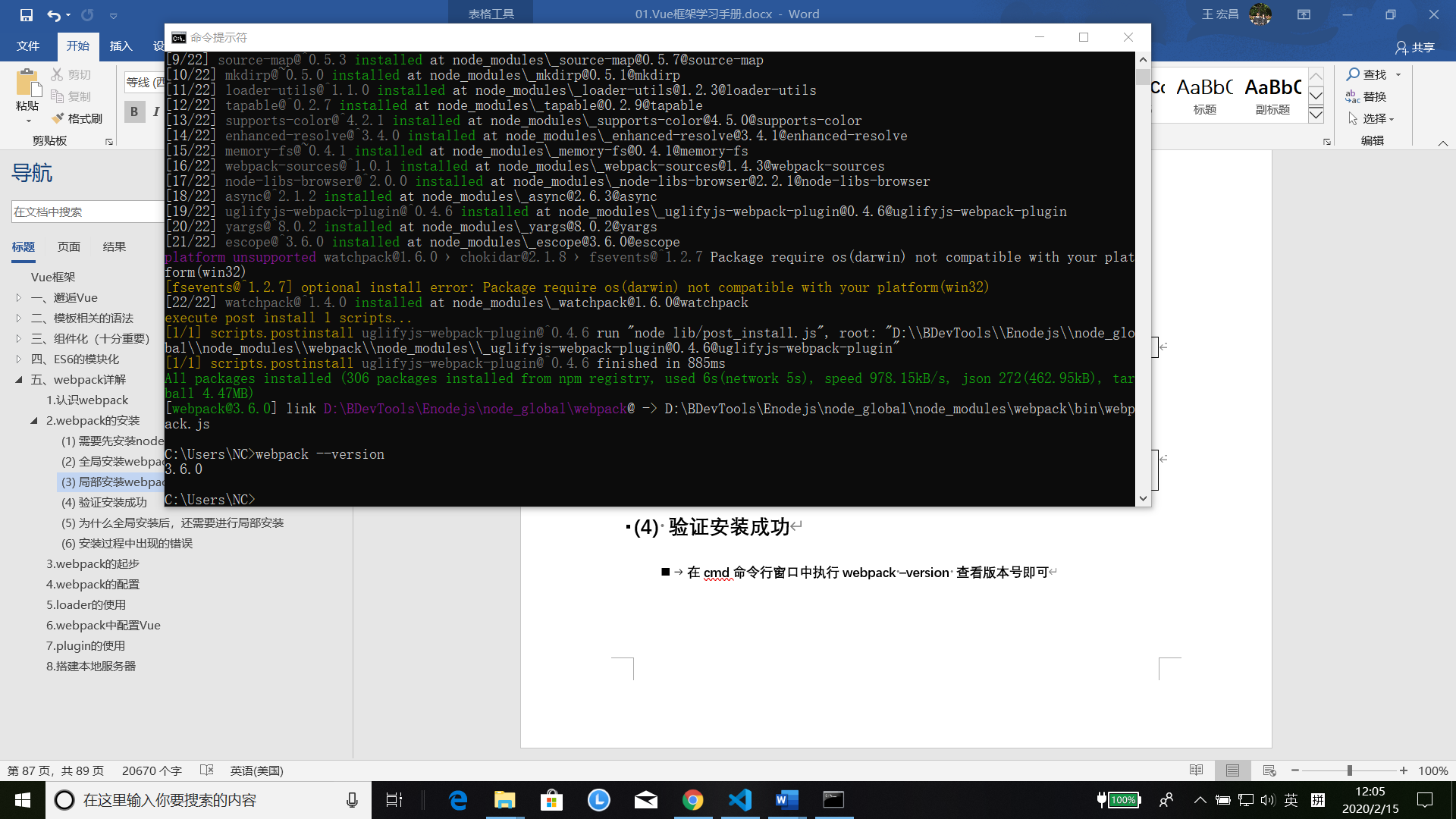
### (3) 局部安装webpack

* **--save-dev 是开发时使用的依赖，项目打包后不需要继续使用的**

|  |
| --- |
| **Cd 到工程目录下面：**  **Npm install –save-dev** [**webpack@3.6.0**](mailto:webpack@3.6.0) **(下载慢时使用cnpm)** |

### (4) 验证安装成功

* **在cmd命令行窗口中执行webpack –version 查看版本号即可**



### (5) 为什么全局安装后，还需要进行局部安装

* 在终端执行webpack命令，使用的是全局安装的webpack.
* 当在package.json中定义了scripts时，其中包含了webpack命令，那么使用的就是局部安装的webpack.

### (6) 安装过程中出现的错误

* **错误1 ：**

|  |
| --- |
| npm ERR! Unexpected end of JSON input while parsing near '...f1d0381ca6f031","tarb' |
| **解决方案：**   1. **清除npm 缓存 npm cache clean --force** 2. **重新执行安装 npm install** [**webpack@3.6.0**](mailto:webpack@3.6.0) **-g** |

* **问题2：**

|  |
| --- |
| **当使用npm进行安装时候，会很慢，甚至安装失败**  **此时直接使用cnpm安装会快很多！** |

## 3.webpack的起步

### (1) webpack基本使用过程

#### 1.1 常用前端工程目录结构介绍（关键）

|  |
| --- |
| **1.根目录下面有一个index.html文件，这个是主页面**  **2.src目录 ：用来存放开发文件，主要的开发位置**  **3.dist目录：用来存放我们打包之后的文件，也就是真正部署使用的文件** |
| **本练习中：**   1. **源代码src目录下 有两个js文件：main.js+mathUtiles.js**   **Main.js中引用mathUtiles.js文件中的内容**   1. **打包目录 下只生成一个js文件：bundle.js** |
|  |

#### 进行正常的模块化开发

|  |  |
| --- | --- |
| **1.mathUtils.js模块开发** | **2.main.js开发** |
|  |  |

#### 1.3将开发的逻辑代码进行打包

|  |
| --- |
| **1.进入到主目录中去** |
|  |
| **2.执行打包命令** |
|  |
| **3.检查dist目录下生成了对应的js文件** |
|  |

#### 1.4在html文件中引入打包完成的js文件

|  |
| --- |
|  |

#### 1.5用浏览器打开html文件，查看效果

|  |
| --- |
|  |

#### 1.6打包过程中遇到的问题

|  |
| --- |
| **这个问题实在vscode的终端中遇到的，**  **应该是一个关于vscode的问题，与webpack是没有关系的** |
|  |

#### 1.7解决打包遇到的问题

|  |
| --- |
| **1.管理员身份运行vscode（关键步骤）**  **2.执行下图中的操作即可**  **3.重新执行打包命令即可成功** |
|  |

### (2) webpack.config.js+package.json文件配置

* 在（1）中我们打包的时候，使用的命令是 **webpack .\src\main.js .\dist\bundle.js**
* **也就是通过指定 入口（.\src\main.js）和 出口（.\dist\bundle.js）的方式进行打包**
* **现在我们想要实现直接输入webpack，就实现打包的功能，这样就很方便了**
* **对此需要进行如下操作：**

#### 2.1工程目录与内容，仍然与（1）中相同，不再赘述

#### 2.2使用npm init 命令自动创建出 package.json文件，

|  |
| --- |
| 1.创建过程中的属性设置可以自己写一下或者直接填写默认即可，  **2.因为这里需要使用到node的环境，所以需要进行这一步的操作** |
|  |

#### 2.3在根目录下创建webpack.config.js文件，名字一定是固定的这个，不要修改！（关键！）

|  |
| --- |
|  |

#### 2.4都创建完成直接执行 webpack 命令，达到与（1）中相同的效果

|  |
| --- |
|  |

#### 2.5 webpack 命令设置成功！

### (3)Package.json文件中定义启动（推荐使用！）

|  |
| --- |
| **1. 这一步的操作在2的基础上进行的（也就是必须先有webpack.config.js文件）**  **2. 目的是 将 webpack 命令 映射成 npm run build 这个命令**  **3. 此时需要修改的就是package.json 文件了**  **4. 关于使用的是全局的webpack还是使用的局部的webpack命令呢？**  **4.1 无论何时，只要是在终端中执行webpack，那都是全局的webpack**  **4.2 当使用npm run build 进行映射的时候，首先会在本地工程中查找是否有对应的webpack命令，**  **如果有，则执行的是本地的webpack命令；**  **如果没有，则执行的是全局的webpack命令。** |
|  |
| 关于安装 局部的webpack  **命令npm install** [**webpack@3.6.0**](mailto:webpack@3.6.0) **–save-dev (或者使用cnpm)**  **-dev 参数表示 只在开发时候用到，而打包部署之后就不会再用到了**  **安装完成之后，package.json文件中会一些内容，如图所示，** |
|  |

## 4.webpack的配置

## 5.loader的使用

* **Loader是webpack中一个非常核心的概念。**
* **Webpack使用来做什么呢？**
* 在我们之前的实例中，我们主要是用webpack来处理我们写的js代码，并且webpack会自动处理js之间相关的依赖。
* 但是，再开发中我们不仅仅有基本的js代码处理，我们也需要加载css、图片，也包括一些高级的将ES6转成ES5代码，将ts转成ES5代码，将scss、less转成css,将jsx、.vue 文件转成js文件等等。
* 对应webpack本身的能力来说，对于这些转化是不支持的。
* 那怎么办呢？给webpack扩展对应的loader就可以了。
* **Loader使用过程：**
* 步骤一：通过npm安装需要使用的loader
* 步骤二：在webpack.config.js中的modules关键字下进行配置。
* **大部分loader我们都可以在webpack的观望中找到，并且学习对应的用法。**

### 使用css文件的配置

#### 1.1编写css文件

|  |
| --- |
|  |

#### 1.2在main.js 文件中将css文件作为模块导出

|  |
| --- |
|  |

#### 1.3下载style-loader和 css-loader两个依赖

|  |
| --- |
|  |
|  |
|  |

#### 1.4在webpack.config.js文件中添加两个loader的配置

|  |
| --- |
|  |

#### 1.5使用npm run build命令进行打包，查看效果，如图：是我们在css文件中写的两个样式！成功！

|  |
| --- |
|  |

### 使用less文件的配置

|  |
| --- |
| **配置方式同 css文件的配置**，  **关于使用哪一个loader，可以在webpack的官网上面直接找到，还是非常方便的。** |

### 图片文件的配置

|  |
| --- |
| 用到的loader:   1. webpack.config.js文件中的配置的时候有一个limit属性值：    1. 当图片的大小小于limit的值的时候，可以**只配置**url-loader    2. 当图片的大小大于limit的值的时候，还需要安装另外一个file-loader,但是这个   **Loader只需要安装，不需要在webpack.config.js中配置** |
| 这个图片的操作，感觉这种手动配置的方式好复杂，如果后面用到了，再回来复习一下把，  视频虽然看完了，但是这个笔记和操作就先不做了，哈哈哈。 |

### ES6转ES5的babel

* 如果你仔细阅读webpack打包的js文件，发现写的ES6语法并没有转成ES5 语法，那么就意味着可能一些不支持ES6的浏览器不能很好的运行我们的代码。
* 如果需要ES6转成ES5,就需要使用babel。

|  |
| --- |
| 这个的babel-loader还需要一些其他的操作  不过这个地方与webpack的版本有很大的关系，我也不进行操作了，  等用到的时候，再进行详细的操作。 |

## 6.webpack中配置Vue

* 后续项目中，我们会使用Vuejs来进行开发，而且会以特殊的文件来组织vue的组件。所以我们接下来学习一下，如何再webpack环境下继承vuejs。
* 现在，我们希望在项目中使用Vuejs,那么必须要有其依赖，首先进行安装（**因为我们在后续的项目中也会使用vue,包括发布之后，所以这里并不是开发时依赖**）
* **安装vue的命令 ： npm install vue –save**
* 那么接下来就可以按照我们之前学过的vue来进行操作了。

### 配置过程

#### 1.1 安装vue npm install vue –save

|  |
| --- |
|  |

#### 1.2在main.js 中引入vue import Vue from ‘vue’

|  |
| --- |
| **1.引入vue** **import Vue from ‘vue’ (特别注意大小写)**  **2.正常开发 vue的实例即可** |
|  |

#### 1.3在html中引入打包的bundle.js文件使用vue进行视图渲染

|  |
| --- |
|  |

#### 1.4错误出现 ：

|  |
| --- |
| 1.配置过程按照正常的配置过程之后，打包之后访问html文件，可能出现错误：  第一类错误 ： runtime-only ->代码中不可以有任何的template模板  第二类错误 ： runtime-complier ->代码中可以允许有template,因为有个编译器给你编译template |
|  |

#### 1.5错误解决方案：

|  |
| --- |
| **配置webpack.config.js文件：添加如下内容：** |
|  |

#### 1.6运行效果

|  |
| --- |
| **正常渲染出界面** |

### el 和 template 的关系

* 在上述配置正常运行的情况下，我们来考虑一个问题：
* 如果我们希望将data中的数据展示在界面上，就必须要修改index.html文件
* 如果我们后面自定义了组件，也必须修改index.html来使用组件
* 但是index.html 在之后的开发中，我们并不希望手动频繁的修改，是否可以做到呢？
* 定义template属性：
* 在前面的Vue实例中，我们定义了el属性，用来与index.html中的元素进行绑定，让Vue实例进行管理元素中的内容
* 如果我们不想在index.html中直接填充内容，那应该怎么办呢？
* 好办：在Vue实例中定义template属性，并将内容填充到这里面即可

#### 2.1在vue实例中创建template模板

|  |
| --- |
|  |

#### 2.2在index.html中直接定义div#app，内容不填

|  |
| --- |
|  |

#### 2.3实际运行中，template模板的内容会完全替换div#app这个标签！

|  |
| --- |
|  |

### (3)简化一：将Vue实例中的template进行抽取成一个组件

|  |
| --- |
| 在2中的代码，Vue实例的template非常的臃肿，  为了让Vue实例更加的简单，可以直接抽取出一个组件，然后实例中直接注册这个组件即可 |
|  |

### (4)简化二：将组件与main.js文件分离开

|  |  |
| --- | --- |
| 在简化一中，组件对象 与 Vue实例同在main.js文件中，这样就显得main.js 文件非常的臃肿，对此，   1. 重新创建组件对象的js文件 ： App.js 2. 将组件对象通过模块导出 3. 在main.js文件中直接模块导入组件对象，注册组件   这样就使得main.js文件结构更简单，而且，组件开发与main.js分开了 | |
| **1.创建App.js文件创建子组件对象并导出** | **2.main.js直接导入使用子组件App** |
|  |  |

### (5)简化三：将子组件的模板和script和样式进行分离

|  |
| --- |
| 我们发现App.js文件还是很臃肿，因为 模板和逻辑代码和css样式仍然是一坨坨，并没有分开。对此：   1. **直接创建.vue文件进行开发**   1.1 Template标签开发模板代码  1.2 Script 标签开发逻辑代码  1.3 Style标签开发样式代码  如此以来，就将各个部分分开的很细致了   1. **为了在打包的时候webpack能够正确解析.vue文件，还需要进行loader的安装**    1. **vue-loader**    2. **vue-template-compiler 这个东西就是将template解析成render函数的**    3. 这里要注意版本的问题！版本太高好像不行 视频中的是13.0.0    4. Vue-loader需要依赖css-loader，所以也要安装一下 |

#### 5.1开发.vue文件

|  |
| --- |
|  |

#### 5.2main.js中引入.vue文件中的子组件

|  |
| --- |
|  |

#### 5.3安装配置loader依赖

|  |  |
| --- | --- |
| **四个依赖loader:**   1. css-loader, 2. style-loader, 3. vue-loader,(**注意这个版本号，太高了会报一个需要插件的错误**) 4. vue-template-compiler | |
| **1.package.json文件配置** | **2.webpack.config.js文件配置** |
|  |  |

#### 5.4打包运行 npm run build

#### 5.5效果图

|  |
| --- |
|  |

## 7.plugin的使用

* **plugin是什么**
* plugin是插件的意识，通常是对某个框架进行扩展。
* webpack的插件，就是对webpack现有功能的扩展，比如，打包优化，文件压缩等。
* **loader和plugin的区别**
* loader主要用于转换某些类型的模块，是一个类型转换器。
* plugin是插件，对webpack的本身的扩展，是一个扩展器。
* **plugin的使用过程**
* 步骤一：通过npm安装需要使用的plugins(某些webpack已经内置的插件不需要安装)
* 步骤二：在webpack.config.js文件中的plugins属性中配置插件。
* **下面，我们就来看看可以通过哪些插件对现有的webpack打包过程进行扩充，让webpack变得更好用。**

### 添加版权的plugin(在我们打包的文件中)

|  |
| --- |
| 该插件的名称叫做 BannerPlugin，为打包的文件添加版权声明  该插件是webpack自带的插件，可以直接引入webpack，然后使用即可 |
| Webpack.config.js文件的配置如下： |
|  |
| Npm run build 打包完成之后，bundle,js文件中的内容变化： |
|  |

### 打包html文件的plugin（重要）

* **目前，我们的index.html是存放在根目录下面的。**
* 我们知道，在真实发布项目时，发布的时dist文件夹中的内容，但是dist文件夹中如果没有index.html文件，那么打包的js等二五年间也就没有意义了。
* 所以，我们需要index.html文件打包到dist文件夹中，这个时候，就可以使用HtmlWebpackPlugin插件。
* **HtmlWebpackPlugin插件可以为我们做这些事情：**
* 自动生成一个index.html文件（可以指定模板来生成）
* 将打包的js文件，自动通过script标签插入到body中，所以在我们开发时候的index.html 文件中就不需要写引入bundle.js文件的这句话了！
* **安装HtmlWebpackPlugin插件**

npm install html-webpack-plugin –save-dev

* **使用插件，修改webpack.config.js文件中的plugins部分的内容如下：**
* 这里的template表示根据什么模板来生成index.html
* 另外，我们需要删除之前在output中添加的publicPath属性
* 否则插入的script标签中的src可能会有问题

|  |
| --- |
| New htmlWebpackPlugin({  template:’index.html’  }) |

#### 2.1安装html-webpack-plugin插件

**npm install html-webpack-plugin –save-dev**

#### 2.2配置html-webpack-plugin插件

|  |
| --- |
|  |

#### 2.3修改index.html文件

|  |
| --- |
|  |

#### 2.4打包测试

|  |
| --- |
| 打包完成之后再dist文件夹里面有了index.html文件，并且文件内容如下： |
|  |

### js文件压缩的plugin

* 在项目发布之前，我们必然要对js等文件进行压缩处理：
* 这里，我们就对打包的js文件进行压缩。
* 我们使用一个第三方的插件uglifyjs-webpack-plugin,并且版本号指定1.1.1，和CLI2保持一致。

#### 3.1安装

**npm install** [**uglifyjs-webpack-plugin@1.1.1**](mailto:uglifyjs-webpack-plugin@1.1.1) **–save-dev**

#### 3.2配置

|  |
| --- |
|  |

#### 3.3打包测试

|  |
| --- |
|  |

## 8.搭建本地服务器

* Webpack提供了一个可选的本地开发服务器，这个本地服务器基于node.js搭建，内部使用了express框架，可以实现我们想要的让浏览器自动刷新显示我们修改后的结果。
* 不过它是一个单独的模块，在webpack中使用之前需要先安装它。
* **npm install** [**webpack-dev-server@2.9.1**](mailto:webpack-dev-server@2.9.1) **–save-dev**
* devserver也是作为webpack中的一个选项，选项本身可以设置如下属性：
* contentBase:为哪一个文件夹提供本地服务，默认是根文件夹，我们这里要填写./dist
* port:端口号
* inline：页面实时刷新
* historyApiFallback:再SPA页面中，依赖HTML5的history模式
* 我们可以再配置另外一个scripts: --open参数表示直接打开浏览器

### (1)安装webpack-dev-server

**npm install** [**webpack-dev-server@2.9.1**](mailto:webpack-dev-server@2.9.1) **–save-dev**

### (2)配置webpack.config.js文件

|  |
| --- |
|  |

### (3)在package.json文件中添加启动脚本（--open参数）

|  |
| --- |
|  |
|  |

### (4)执行命令启动本地服务器

|  |
| --- |
|  |

### (5)在浏览器中查看效果

|  |
| --- |
|  |

### (6)出现的错误及解决：

|  |
| --- |
| 在我们执行npm run start命令启动服务器的时候，一般情况下会报出如下异常。  他说，找不到这个模块，这可真让人头大啊。 |
|  |
| **原因**：这是因为，这个服务器需要使用本地的webpack依赖，而我们并没有在这个工程本地进行webpack的安装。 |
| 因此：我们执行命令安装一下即可 ：  **npm install** [**webpack@3.6.0**](mailto:webpack@3.6.0) **–save-dev**  **然后重新执行启动命令，就不会报错了！** |

## webpack配置文件的分离写法

### (1)安装工具webpack-merge

**npm install webpack-merge –save-dev**

### (2)将原来的webpack.config.js分成三个文件

|  |
| --- |
| **基础的公共文件，开发时文件，生产时文件**  **在我们开发调试过程中使用的时候，基础的公共文件+开发时文件**  **在我们打包部署的时候使用的是 ： 基础的公共文件+生产时文件** |
| **1.下面是基础公共文件：** |
|  |
| **2.下面是开发时用到的文件：** |
|  |
| **3.下面是打包部署时用到的文件：** |
|  |

|  |
| --- |
| **下面是这三个文件的目录位置：** |
|  |

### (3)修改package.json文件中的构建脚本和启动脚本，手动指定我们自己创建的配置文件

|  |
| --- |
|  |

**注意：**我们还需要注意基础的公共文件的路径问题，否则打包出来的内容不一定是在根目录下面，因为path是根据配置文件的路劲进行获取的，这一点要特别注意。

# 六、Vue CLI

## 1.什么是VueCLI

* 如果你只是简单写几个Vue的Demo程序，那么不需要Vue CLI
* 如果你在开发大型项目，那么你需要，并且必然需要使用Vue CLI
* 使用Vue.js开发大型应用时，我们需要考虑代码目录结构，项目结构和部署、热加载、代码单元测试等事情。
* 如果每个项目都要手动完成这些工作，那无疑效率比较低效，所以通常我们会使用一些脚手架工具来帮助我们完成这些事情。
* CLI是什么意思？
* CLI是Command-Line Interface ,翻译为命令行界面，但是俗称脚手架。
* Vue CLI 是一个官方发布的vue.js项目脚手架。
* 使用vue-cli可以快速搭建Vue开发环境以及对应的webpack配置。
* 脚手架长什么样子？

## 2.Vue CLI使用前提-Node

* 安装Node.js
* 参考《02.Node,js安装配置手册》-我自己写的
* 检测安装的版本
* node -v
* npm -v
* Node环境要求8.9及以上
* 什么是NPM?
* NPM 的全称是Node Package Manager
* 是一个Nodejs包管理和分发工具，已经成为了非官方的发布Node模块（包）的标准
* 后续我们会经常使用NPM来安装一些开发过程中依赖包

## 3.Vue CLI使用前提-Webpack

* Vue.js官方脚手架工具就使用了webpack模板
* 对所有的资源会压缩等优化操作
* 他在开发过程中提供了一套完整的功能，能够使得我们开发过程中变得高效
* Webpack的全局安装
* npm install webpack -g

## 4.Vue CLI 的安装

### (1)安装vue cli

* **npm install @vue/cli -g**
* **vue --version 查看版本信息**

|  |
| --- |
|  |

### (2)拉取2.x版本的vue cli 模板

* **Vue CLI >= 3 和旧版使用了相同的 vue 命令,所以 Vue CLI 2 (vue-cli) 被覆盖了。**
* **如果你仍然需要使用旧版本的 vue init 功能，你可以全局安装一个桥接工具：**
* **npm install @vue/cli-inti -g**
* **初始化工程的命令是 ：vue init webpack my-project(这个是工程的名字)**

|  |
| --- |
|  |

### (3)创建工程的命令

* **脚手架>=3** ： **vue create my-project**
* **脚手架2** ： **vue init webpack my-project**

## 5.VueCLI2初始化项目

### (1)执行初始化项目命令

**vue init webpack VueStudy03**

### (2)按照提示的步骤一步一步的进行合适的选择

|  |
| --- |
|  |
|  |

### (3)生成之后的项目目录如下：

|  |
| --- |
|  |

### Runtime-compiler 和 runtime-only的区别

|  |
| --- |
|  |
|  |
| **所以最终直接使用runtime-only 即可！** |

## 6.VueCLI3创建Vue项目

### (1)VueCLI2和3版本的区别

* Vue-cli 3 是基于webpack4打造。Vue-cli2还是webpack3.
* Vue-cli 3 的设计原则是“0配置”，移除的配置文件根目录下的，build和config等目录。
* Vue-cli 3提供了vue ui 命令，提供了可视化配置，更加人性化。
* 移除了static文件夹，新增了public文件夹，并且index.html 移动到public中

### (2)VueCLI3创建项目

* **Vue create my-project**
* **主要就是体会一下创建项目的过程!**
* **项目名称全都是要小写！**

### 关于VueCLI3的配置文件的相关信息

#### 3.1界面操作

|  |
| --- |
| **在终端中执行vue ui命令，即可打开配置用户界面** |
|  |

#### 3.2查看默认的配置文件的位置

|  |
| --- |
|  |

#### 3.3添加自己的配置文件

|  |
| --- |
| **在根目录下面添加文件 vue.config.js 这个名字是固定的，一定不能变！** |
|  |

# 七、Vue路由

## 1.认识路由

### (1)什么是路由？

* **说起路由你想起了什么？**
* 路由是一个网络工程里面的术语。
* 路由（routing）就是通过互联的网络把信息从源地址传输到目的地址的活动。
* **啥？没听懂？**
* 在生活中，我们有没有路由的概念？当然了，就是路由器嘛。
* 路由器是做什么的？
* 路由器提供了两种机制：**路由和转送**。
* **路由就是决定数据包从来源到目的地的路径。**
* **转送将输入端的数据转移到合适的输出段**。
* 路由中有一个非常重要的概念叫**路由表**。
* **路由表本质上就是一个映射表，决定了数据包的指向。**

### (2)修改url而使页面不发生刷新的两种方式

* **方式一：url的hash值**

|  |
| --- |
|  |

* **方式二：HTML5的history.pushState()方法（.back(),.go(),.forward()；.replaceState()）**

|  |
| --- |
|  |

### (3)重点概念

* **Vue-router是vue.js官方的路由插件**，它和vue.js是深度继承的，适合用于构建单页面应用。
* **Vue-router是基于路由和组件的**
* 路由用于设定访问路径，将路径和组件映射起来。
* 在vue-router的单页面应用中，页面的路径的改变就是组件的切换。

## 2.Vue-router基本使用

### (1)安装vue-router

* 因为我们已经学习了webpack，后续开发中我们主要是通过工程化的方式进行开发的。
* 所以在后续，我们直接使用npm来安装路由即可。
* 步骤一：**安装vue-router**
* **npm install vue-router –save**

|  |
| --- |
|  |

* 步骤二：**在模块化工程中使用它（因为它是一个插件，所以可以通过Vue.use()来安装路由功能）**
* 第一步：导入路由对象，并调用Vue.use(VueRouter)
* 第二步：创建路由实例，并且传入路由映射配置
* 第三步：在Vue实例中挂载创建的路由实例

|  |
| --- |
|  |
|  |

### (2)路由的映射配置和呈现

#### 2.1创建两个组件home和about

|  |
| --- |
|  |

#### 2.2配置两个组件对应的前端路由

|  |
| --- |
|  |

#### 2.3在App组件模板中添加链接和呈现位置

|  |
| --- |
|  |

#### 2.4运行结果

|  |  |
| --- | --- |
|  |  |

### (3)路由的默认值的配置

|  |
| --- |
| 当我们只输入服务器的地址的时候，默认导航到Home页面上面 |
|  |

### (4)修改成为history模式的路由

|  |  |
| --- | --- |
| 默认为hash模式的路由 | 下面是修改成history模式 |
|  |  |

### (5)router-link的属性相关

* **to属性:指向路由中配置的路径**

|  |
| --- |
|  |

* **tag属性：可以指定<router-link/>标签渲染成什么标签，默认是<a/>标签**

|  |
| --- |
|  |

* **replace属性：可以设置路由的方式是 html5中的history的replaceStack的方式，也就是不能返回的方式**

|  |
| --- |
| **不能后通过点击浏览器导航栏左边的后退按钮了就** |

* **active-class:当路由匹配成功的时候，会自动给这个router-link添加一个类样式router-link-active**
* **活跃的时候会添加这个属性**

|  |
| --- |
|  |

* **你可以修改这个属性中的内容，比如：修改字体的颜色**

|  |  |
| --- | --- |
|  |  |

* **你可以给这个类改个名字：（一般情况下是不会进行修改的！）**
  + **方式一：直接在router-link中改，只改这一个**

|  |  |
| --- | --- |
|  |  |

* + **方式二：在路由配置属性中改：这样所有的都会改**

|  |  |
| --- | --- |
|  |  |

### (6)通过代码跳转路由的操作

|  |
| --- |
| 比如我们现在想要通过**点击按钮**进行页面的跳转 |
|  |
|  |

### (7)动态路由

* + 在某些情况下，一个页面的path路径可能是不确定的，比如我们进入用户界面时，希望时如下的路径：
    - /user/zhangsan 或者 /user/lisi
    - 即：**除了有前面的/user之外，后面还跟上了用户的ID**
    - 这种path和component的匹配关系，我们称之为动态路由（也就是路由传递数据的一种方式）

#### 7.1创建组件

|  |
| --- |
|  |

#### 7.2配置路由

|  |
| --- |
|  |

#### 7.3路由跳转的位置传入参数

|  |
| --- |
|  |

#### 7.4运行效果

|  |
| --- |
|  |

### (8)路由的懒加载

* + **官方给出了解释**：
    - 当打包构建应用时，JavaScript包会变得非常大，影响页面加载。
    - 如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候菜价在对应组件，这样就更加高效了
  + **官方在说什么呢？**
    - 首先，我们知道路由中通常会定义很多不同的页面。
    - 这个页面最后被打包在哪里呢？一般情况下，是放在一个js文件中。
    - 但是，页面这么多放在一个js文件中，必然会造成这个页面非常的大。
    - 如果我们一次性从服务器请求下来这个页面，可能需要花费一定的时间，甚至用户的电脑上还出现了短暂空白的情况。
    - 如何避免这种情况？使用路由懒加载就可以了。

#### 8.1未使用懒加载前的打包效果

|  |
| --- |
|  |

#### 8.2使用懒加载后的打包效果

|  |
| --- |
|  |

#### 8.3路由懒加载的写法（必会）

|  |
| --- |
|  |

## 3.Vue-router嵌套路由

* + 嵌套路由是一个很常见的功能
    - 比如在home页面中，我们希望通过/home/news和/home/message分别访问一些内容
    - 一个路径映射一个组件，访问这两个路径也会分别渲染两个组件。
  + 实现嵌套路由的步骤
    - 创建对应的子组件，
    - 在路由映射中配置对应的子路由
    - 在组件内部使用<router-view>标签

### 创建子组件

|  |
| --- |
|  |

### (2)配置子路由（懒加载方式）

|  |
| --- |
|  |
|  |

### (3)组件内部进行跳转配置

|  |
| --- |
|  |

### (4)运行效果

|  |
| --- |
|  |

## 4.Vue-router参数传递（必会）

* **参数传递主要有两种方式：params和 query**
* **params的类型（2（7） 的动态路由就是）**
* 配置路由格式 ： /home/:id
* 传递参数的方式：在path后面跟上对应的值
* 传递后形成的路径 ：/home/news /home/message
* **query的类型：(我喜欢用这种方式)**
* 配置路由的格式：/home ,也就是普通配置
* 传递参数的方式：对象中使用query的key作为传递方式
* 传递后形成的路径：/home?id=news /home?id=message

### (1)操作一：常规操作

#### 1.1创建组件

|  |
| --- |
|  |

#### 1.2配置路由

|  |
| --- |
|  |
|  |

#### 1.3添加router-link跳转并传递对象参数

|  |  |
| --- | --- |
|  |  |

#### 1.4运行效果

|  |
| --- |
|  |

### (2)操作二：通过js代码跳转并传参

|  |
| --- |
|  |

## 5.Vue-router导航守卫

* **需求**：

当实现页面跳转的时候，页面的标题显示跳转到的页面的名称

* **实现步骤：**
* **在路由映射中添加meta元数据对象**

|  |  |  |
| --- | --- | --- |
|  |  |  |

* **重写router路由对象的beforeEach(()=>{})方法**

|  |
| --- |
|  |

* **查看运行效果**

|  |
| --- |
|  |

* **看一下整体的路由结构**

|  |
| --- |
|  |

* **导航守卫的补充**
* **如果是后置钩子，也就是afterEach(),不需要调用next()方法**
* **上面我们使用的守卫，称之为全局守卫。**
* **还有路由独享的守卫，和 组件内的守卫**

**详情请了解官方文档 ：**<https://router.vuejs.org/guide/advanced/navigation-guards.htm> 这个是路由守卫的这一部分的详解，可以切换中文语言

## 6.Keep-alive

### (1)生命周期方法的回顾

|  |
| --- |
|  |

### (2)Keep-alive的基本使用

|  |  |
| --- | --- |
|  |  |

### (3)使用组件内的守卫解决跳回到Home页仍然是Message的问题

|  |  |  |
| --- | --- | --- |
|  |  |  |

这样写是可以实现这个功能的，但是会报一个错误，不过效果能实现。

这个地方碰见的这个问题有点奇怪，而且，一般的话在开发中不会这样的！

### (4)Keep-alive 的一些其他属性：

* Keep-alive 是Vue内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染
* 它有两个非常重要的属性：
* include : 字符串或正则表达式，只有匹配的组件会被缓存
* exclude : 字符串或正则表达式，任何匹配的组件都不会被缓存
* 组件中有一个name的属性，这里就可以使用这个name属性
* 例如 exclude=”User,About” :可以有多个，但是不要加空格！
* Router-view也是一个组件，如果直接被包在keep-alive里面，所欲匹配到的视图组件都会被缓存。

## 7.案例

工程详见**D:\DDevWorkspace\workspaceVue\vuestudy06tabbar**

# 八、Promise

## 1.什么是promise？

* **Es6中一个非常重要和好用的特性就是promise**
* 但是初次接触promise会一脸懵逼，这是个什么东西？
* 看看官方或者一些文章对他的介绍和用法，也是一头雾水。
* **Promise到底是做什么的呢？**
* **Promise是异步编程的一种解决方案。**
* **那么什么时候我们回来处理异步事件呢？**
* 一种很常见的场景应该是网络请求了
* 我们封装一个网络请求的函数，因为不能立即拿到结果，所以不能像简单的1+1=2一样将结果返回。
* 所以我们往往会传入另一个函数，在数据请求成功时，将数据通过传入的函数回调出去。
* 如果只是一个简单的网络请求，那么这种方案不会给我们带来很大的麻烦。
* **但是，当网络请求非常复杂时，就会出现回调地狱**。
* 我们来一个案例说明。
* **回调地狱：我们来考虑如下的情况：**
* 我们需要通过一个url1从服务器加载一个数据data1,data1中包含了下一个请求的url2.
* 我们需要通过data1取出url2,从服务器加载数据data2,data2中包含了下一个请求的url3.
* 我们需要通过data2取出url3,从服务器加载数据data3,data3中包含了下一个请求的url4.
* 发送网络请求url4,得到最终的数据data4
* **Promise可以以一种非常优雅的方式来解决这个问题。**

## 2.简单使用

|  |
| --- |
|  |
|  |

## 3.Promise的三种状态

* **首先，当我们开发中有异步操作的时候，就可以给异步操作包装一个promise**
* **异步操作之后会有三种状态**
* **我们一起来看一下这三种状态：**
* **pending:等待状态，比如正在进行网络请求，或者定时器没有到时间**
* **fulfill:满足状态，当我们主动毁掉了resolve时，就处于该状态，并且会回调then()**
* **reject:拒绝状态，当我们主动回调了reject时，就处于该状态，并且回调catch()**

## 4.链式调用

### (1)第一种调用;常规思路

|  |
| --- |
|  |

### (2)第二种调用，把then拿出来

|  |
| --- |
|  |

### (3)第三种调用，简化一下第二种

|  |
| --- |
|  |

### (4)第四种调用，简化一下第三种

|  |
| --- |
|  |

## 5.All方法(很好用)

|  |
| --- |
|  |

# 九、VueX详解

## 1.VueX概念和作用解析

* **官方解释：Vuex是一个专为Vue.js应用程序开发的状态管理器。**
* 它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。
* Vuex也是集成到Vue的官方调试工具devtools.extension,提供了诸如零配置的time-travel调试。状态快照导入导出等高级调试功能。
* **状态管理到底时什么？**
* 状态管理模式，集中式存储管理这些名词听起来就非常高大上，让人琢磨不透。
* 其实，你可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面。
* 然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用。
* 那么，多个组件是不是就可以共享这个对象中的所有变量属性了呢?
* **等等，如果是这样的话，为什么官方还要专门出一个插件Vuex呢？难道我们不能自己封装一个对象来管理吗？**
* 当然可以，只是我们要先想想vue带给我门的最大的便利是什么？没错，就是响应式。
* 如果我们自己封装一个对象能不能保证它里面所有的属性做到响应式呢？当然也可以，只是自己封装可能稍微麻烦一些。
* 不用怀疑，vuex就是为了提供这样一个在多个组件间共享状态的插件，用它就可以了。
* **但是，有什么状态是需要我们在多个组件间共享的呢？**
* 如果你做过大型开放项目，你一定遇到过多个状态，在多个界面间的共享问题。
* Virus用户的登陆状态，用户名称，头像，地理位置等信息。
* 比如商品的收藏，购物车中的物品等等。
* 这些状态信息，我们都可以放在统一的地方，对它进行保存和管理，而且他们还是响应式的。
* **Ok,从理论上理解了状态管理之后，让我们从实际代码再来看看状态管理。**
* 毕竟，**Talk is cheap,Show me the code.(Linus)**

## 2.Vuex的使用的准备工作

### (1)安装vuex 的插件

|  |
| --- |
| **npm install vuex --save** |
|  |

### (2)像router一样给vuex建一个目录的文件,注意：Store这个方法是大写的，下面的截图中写的有

|  |
| --- |
|  |

### (3)在main.js的vue实例中注册 vuex对象

|  |
| --- |
|  |

## 3.Vuex的简单使用一:state对象

### (1)新建两个组件 app1 和 app2

|  |
| --- |
|  |

### (2)在vuex对象中的state状态属性对象中添加一个counter属性

|  |
| --- |
|  |

### (3)在app1 和 app2 中分别使用这个counter属性

|  |
| --- |
|  |

### (4)在app1 中添加两个按钮，一个点击使counter+1,一个点击使counter-1

|  |
| --- |
|  |

### (5)在App组件中注册两个子组件

|  |
| --- |
|  |

### (6)并运行查看效果

|  |
| --- |
|  |

## 4.Vuex的简单使用二：mutations

**这里面可以用来定义方法，实现对state中定义的共享变量的操作。**

**这也是官方推荐的做法。**

### (1)在mutations中添加方法

|  |
| --- |
|  |

### (2)修改App1组件中的按钮点击方法

|  |
| --- |
|  |

### (3)通过vue-devtools的chrome浏览器插件查看运行效果

|  |
| --- |
|  |

## 5.Vuex的Getters详解

**可以类比组件中的计算属性，computed，即，当需要使用state中的共享数据时，但是又需要将这个数据进行一些复杂的操作的时候，就可以使用getters**

### (1)在state中定义一个students的数组

|  |
| --- |
|  |

### (2)在getters中定义一个变量more20student，来获取student的age大于20的人

|  |
| --- |
|  |

### (3)在组件中通过$store.getters.more20student,来获取处理过后的数据

|  |
| --- |
|  |

### (4)效果展示

|  |
| --- |
|  |

### (5)Getters中的参数二：getters本身

|  |
| --- |
| **1.把getters传进去** |
|  |
| **2.在组件中直接使用** |
|  |
| **3.看效果** |
|  |

### (6)传入其他的参数的操作\*

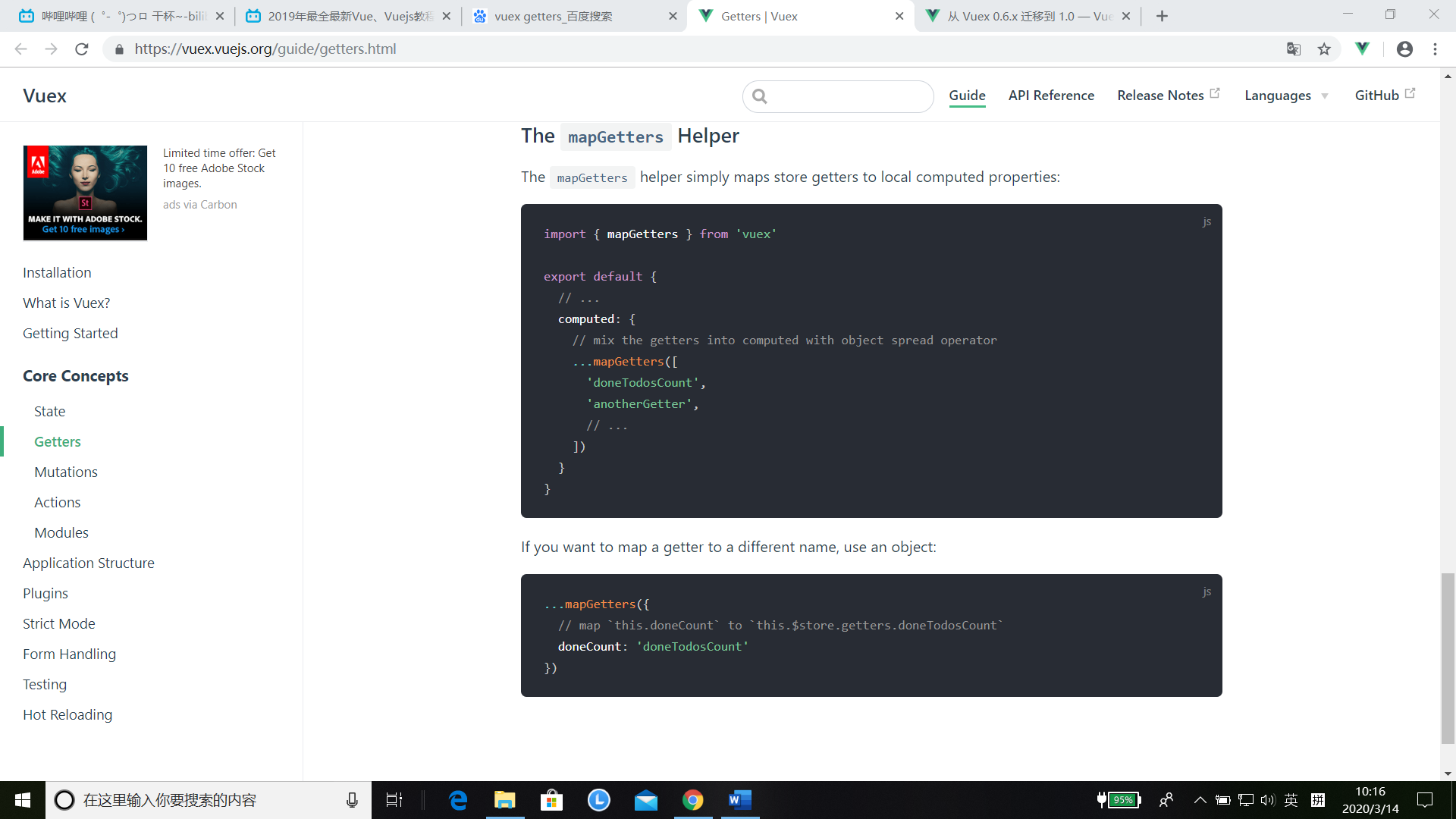
**Getters中的方法的参数;第一个是state,第二个是getters本身，**

**那如果你想传入其他的内容作为参数时该怎么办呢？**

**好办，直接返回一个函数即可！！！，这个思路一下子就清晰了！**

|  |
| --- |
| **1.定义一个返回值是函数的 getters中的对象** |
|  |
| **2.在组件中使用这个对象** |
|  |
| **3.效果图** |
|  |

### (7)Getters转为组件中的计算属性的操作



## 6.Vuex的mutations的详解

### (1)再传入一个参数：

* **Vuex的store状态的更新唯一方式：提交mutation**
* **Mutation主要包括两部分：**
* 字符串的事件类型（type）
* 一个回调函数（handler）,该回调函数的第一个参数就是state
* **通过mutation更新数据的时候，有可能我们希望携带一些额外的参数**
* 参数被称为mutation的**载荷（payload）**
* **但是如果参数不是一个呢**？
* 这个时候，我们通常会**以一个对象的形式进行传递**。
* 而且，当有多个参数的时候**只能通过对象**的形式来进行传递，否则会报错
* **也就是说，mutation中的方法，的参数最多就有state和另外一个！**

|  |
| --- |
| **1.在mutation中创建一个函数** |
|  |
| **2.在组件中调用mutation中的函数，进行数据更新** |
|  |
| **3.效果图** |
|  |

### Mutation的提交风格

* **上面的commit的提交是一种普通的方式**
* **Vue还提供了另外一种提交方式，它是一个包含type属性的对象**

|  |
| --- |
| **1.普通的提交风格的参数的观察：传的是什么就是什么** |
|  |
|  |
|  |
|  |
| **2.第二种风格的参数的观察** |
|  |
|  |
|  |
|  |

### (3)响应式的规则(重点理解)

* **Vuex的store中的state是响应式的，当state中的数据发生变化时，vue组件会自动更新。**
* **但是，这需要要求我们遵守一些 vuex对应的规则**
* **提前再store中初始化好所需要的属性**
* **当给state中的对象添加新的属性时，使用下面的方式;**
* **方式一：使用Vue.set(obj,’newProp’,value)**

**Vue.delete(obj,’propName’)**

* **方式二：用新对象给就对象重新赋值**

**State.info = […state.info,’newProp’:value]**

### (4)Mutation常量类型(官方推荐的写法)

* 我们来考虑下面的问题：
* 在mutation中，我们定义了很多事件类型（type,也就是其中的方法名称）
* 当我们的项目增大时，vuex管理的状态越来越多，需要更新状态的情况越来越多，那么意味着mutatioin中的方法越来越多
* 方法过多，使用者需要花费大量的精力取记住这些方法，甚至是多个文件间来回切换，查看方法名称，甚至如果不是复制的时候，可能还会出现写错的情况。
* 那么我们有什么好的办法可以解决这个事情吗？
* 有！
* 那就是把这个type(也就是方法名称)统一的抽取成一个文件，在这个文件中同一进行管理。
* 在使用的时候，再导入即可。

|  |
| --- |
| **1.我们先来回顾 一下之前的写法** |
|  |
| **2.将我们的类型名称统一定义到一个文件中** |
|  |
| **3.在mutations中使用统一定义的类型名称** |
|  |
| **4.在组件中提交的时候，也使用统一定义的类型名称** |
|  |
| **5.运行效果** |
|  |

## 7.Vuex的actions的详解

* **通常情况下，Vuex要求我们的mutatioins中的方法必须是同步方法。**
* 主要的原因是当我们使用devtools时，devtools可以帮助我们捕捉mutation的快照。
* 但是如果执行的是异步操作，那么devtools将不能很好的追踪这个操作什么时候会被完成。
* **我们强调，不要在mutations中进行异步操作。**
* 但是某些情况，我们确实希望在vuex中进行一些异步操作，比如网络请求，必然是异步的，这个时候怎么处理呢？
* **Action类似于Mutation，但是是用来代替Mutations进行异步操作的。**

### (1)简单使用一下action的思路

|  |
| --- |
| **1.组件中点击按钮触发事件：使用dispatch提交到actions中** |
|  |
| **2.在actions中定义对应的方法，并模拟执行异步操作** |
|  |
| **3.查看运行效果** |
|  |
|  |

### (2)当在action中想给调用的位置返回一个信息的时候的操作：Promise

|  |
| --- |
| **1.在actions中添加promise作为返回值** |
|  |
| **2.在组件中接收promise的返回值** |
|  |
| **3.运行效果** |
|  |

## 8.Modules详解

* **Module是模块的意思，为什么在vuex中我们要使用模块呢？**
* Vue使用单一状态树，那么也意味着很多状态都会交给vuex来管理
* 当应用变得非常复杂是，store对象就有可能变得相当臃肿
* 为了解决这个问题，vuex允许我们将store分割成模块module,而每个模块拥有自己的state，mutations，actions，getters等

### (1)Module的组织结构

|  |
| --- |
|  |

### (2)Module中的state

|  |
| --- |
| **1.在module中定义state的内容** |
|  |
| **2.在组件中使用module中的state的内容（注意state的结构树）** |
|  |
| **3.运行效果** |
|  |

### (3)Module 中的mutations

|  |
| --- |
| **1.module中的mutationns添加方法** |
|  |
| **2.组件中commit到mutations的方法** |
|  |
| **3.运行效果** |
|  |

### (4)Module 中的getters

|  |
| --- |
| **1.定义getters注意有三个参数的情况** |
|  |
| **2.组件中使用getters** |
|  |
| **3.运行效果** |
|  |

### (5)Module 中的actions

|  |
| --- |
| **1.创建module中的actions,注意 参数** |
|  |
| **2.组件中dispatch到module中的actioins** |
|  |
| **3.看效果** |
|  |
|  |

# 十、Axios框架详解

## 1.Axios的基本使用

### (1)安装axios依赖

|  |
| --- |
| **npm install axios --save** |
|  |

### (2)我自己用springBoot准备了一个服务端，用来接收请求的(这个操作作为一名java程序员来讲，应该是个小操作吧)

|  |
| --- |
|  |

### (3)使用前先引入axios的依赖

|  |
| --- |
| **常规操作，无需解释** |
|  |

### (4)使用方式一：axios({config}) 无参

|  |
| --- |
| **1.直接请求数据** |
|  |
| **2.查看效果** |
|  |

### (5)使用方式一：axios({config}) 有参

|  |
| --- |
| **1.请求数据的操作** |
|  |
| **2.查看效果** |
|  |

### (6)使用方式二：axios.get(url) 无参

|  |
| --- |
| **1.请求** |
|  |
| **2.查看请求结果** |
|  |

### (7)使用方式二：axios.get(url,options?) 有参

|  |
| --- |
| **1.请求** |
|  |
| **2.查看请求结果** |
|  |

### (8)使用方式n:类比get即可，post,put,delete都是一样的使用的

## 2.Axios并发请求的操作 axios.all

### (1)直接获取到结果的操作

|  |
| --- |
| **1.请求** |
|  |
| **2.查看结果** |
|  |

### (2)将结果结构的操作

|  |
| --- |
| **1.请求** |
|  |
| **2.查看结果** |
|  |

## 3.Axios全局配置信息的操作

|  |
| --- |
| **1.体会一下baseUrl的配置** |
|  |
| **2.看一下运行结果，是一样的** |
|  |

## 4.Axios实例

* 为什么要创建axios的实例呢？
* 当我们从axios模块中导入对象时，使用的实例是默认的实例。
* 当给该实例设置一些默认配置时，这些配置就被固定下来了。
* 当时后续开发中，某些配置可能会不太一样。
* 比如某些请求需要使用特定的baseURL或者timeout或者content-Type等。
* 这个时候，我们就可以创建新的实例，并且传入术语该实例的配置信息。
* 下面看我如何操作axios的实例。

|  |
| --- |
| **1.创建新的实例并发送请求** |
|  |
| **2.查看一下效果** |
|  |

## 5.自己封装请求模块

* **整体思路：**
* 1.自己封装一个请求模块
* 2.在组件中使用自己封装的模块
* **优势**：
* 1.安全
* 2.便于整个系统的维护
* 3.面向我自己的封装进行开发

### (1)封装方式一：return new Promise()

|  |
| --- |
| **1.新建目录，创建封装文件** |
|  |
| **2.在组件中使用自己封装的请求** |
|  |
| **3.查看效果** |
|  |

### (2)封装方式二：直接return

|  |
| --- |
| **1.原因解释** |
|  |
| **2.自己封装** |
|  |
| **3.组件使用** |
|  |
| **4.效果查看** |
|  |

### (3)封装三：同理可得的并发请求的封装（我自己写的）

|  |
| --- |
| **1.原理解释** |
|  |
| **2.封装操作** |
|  |
| **3.组件使用** |
|  |
| **4.效果查看** |
|  |

## 6.Axios的拦截器

* **Axios为我们提供了拦截器，用于我们每次在发送请求，或者获取响应之后进行对应的处理。**
* **因此拦截器共有以下四种类型：请求成功，请求失败；响应成功，响应失败。**

|  |
| --- |
|  |

# 十一、项目中的补充

## 1.vue中的监听 图片 加载完成的操作

|  |
| --- |
| **1.img标签中添加监听 @load=”function()”** |
|  |
| **2.实现监听的方法** |
|  |
| **3.这样做就OK了** |

## 2.vue2.x中的事件总线（不常用）

|  |
| --- |
| **1.在main.js中添加Vue的原型中添加一个属性作为事件总线** |
|  |
| **2.在一个组件中通过事件总线发射事件** |
|  |
| **3.在另一个组件中通过事件总线监听事件** |
|  |
| **4.完成** |
| **5.这个操作适用在组件关系复杂却又想监听事件的时候** |

## 3.防抖函数的操作

|  |
| --- |
| **1.创建防抖动的函数** |
|  |
| **2.使用防抖动的函数** |
|  |
| **3.完成** |
| 函数执行的次数明显的就变少了！  取消的方法是 ：this.$bus.$off(‘事件名称’，要取消的那个监听方法) |

## 4.获取组件中的元素的方法（offsetTop的值）

|  |
| --- |
| **1.通过组件的$el属性就可以拿到组件对应的div元素了** |
|  |
| **2.查看效果** |
|  |
| **获取组件的offsetTop的值 ： this.$ref.tabController.$el.offsetTop ,就是这样了** |

## 混入

这个是高级操作，不过也很简单，可以直接看看官方文档。

主要的作用就是，当多个组件中有相同的代码的时候，可以抽取出来，然后达到共用的目的。

## 6.Vue中可以自己封装插件！

用到的时候，可以再去查官方的文档