# 学习Vue

## Vue.js安装

### 方式一：直接CDN引入

开发环境版本：

<script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>

生产环境版本：

<script src="https://cdn.jsdelivr.net/npm/vue@2.6.14"></script>

### 方式二：下载和引入

<https://vuejs.org/v2/guide/installation.html（开发版和生产版本）>

引用<script src=”../js/vue.js”></script>

### 方式三：NPM安装

（脚手架就不需要前两种方式）

$ npm install vue

## vue.js的使用

### Vue编程

编程范式：声明式编程

<div id=”app”>{{message}}{{counter}}

<li v-for=”(item,index) in movies”>{{index}}-{{item}}</li> //遍历数组

<button v-on:click=”add”>+</button> v-on简写为@click //实现方法

</div>// 内部进行解析message

创建vue实例 传入一个****options 对象 后面跟属性data****

const app=new Vue{( //用app变量接受实例

el:”#app”, //管理div中的app,挂载要管理的元素

data:{ //定义数据 也可以是网络 服务器上面的

message:’你好呀’,// 数据替换

movies:[‘’],

counter:0

},

methods:{

add:function(){

this.counter++ //this指向app实例对象当前对象

},

}

)}

### MVVM理解

Console控制台中，app.message=’你好’ app.movies.push(‘海王’)数组中添加数据View(div)模块中add 通过ViewModel(new Vue)模块解析 到methons方法中的counter 回调到Model(data)模块中counter 再通过ViewModel(new Vue)解析 回到View(div)模块中的counter

编程范式：声明式编程 MVVM

Model到ViewModel再到View

### options中属性定义

el:

类型：string|HTMLElement 管理哪一个DOM

data:

类型：Object|Function（组件当中data必须是一个函数） 数据对象

methods:

类型：{[key:string]:Function} 定义Vue方法，可以在其他地方调用，指令中使用

方法：method 函数：function

生命周期函数

计算属性computed:

### 生命周期

生命周期：事物从诞生到消亡的整个过程

Vue生命周期（钩子函数）：

beforeCreate:function(){}

created:function(){}

beforeMounte:function(){}

mounted:function(){}

beforeUpdate:function(){}

updated:function(){}

beforeDestory:function(){}

destoryed:function(){}

### **插值mustache语法：**

v-once:

后面不跟任何表达式，元素和组件只会渲染一次，不会随着数据的改变而改变。

v-html:

<h2 v-html=”url”></h2>

url:”<a href=”#”>百度一下</a>”

v-pre不进行解析预格式化

v-clock在vue解析之前，div中有一个属性v-clock，在vue解析之后，div中没有一个属性v-clock。

<style>

[v-clock]{  
display:none;

}

</style>

<div id=”app” v-clock>{{message}}</div>斗篷效果是先显示再消失再显示

### v-bind动态绑定基本属性src href

v-bind除了内容需要动态来决定外，某些属性我们也希望动态绑定

比如动态绑定a元素的href属性，img元素的src属性

v-bind:src v-bind:href

### v-bind动态绑定class

v-bind:class(动态绑定对象（对象可以用函数《在methods中定义函数return对象》代替）包含键值对：类名加布尔值《一般布尔值放到data里，通过methods进行按钮控制取反》) 动态绑定数组 data里面都是变量

### v-bind动态绑定style

传对象

v-bind:style=”{key(属性名):value(属性值)}”属性值必须加上单引号，否则是当作一个变量去解析，属性名为驼峰命名法

传数组

数组里面存对象，数组里面是一个个的元素

### computed计算属性

解决任何复杂重复逻辑,methods没有缓存重复次数多，性能比较低,而computed响应式依赖缓存重复次数只有一次，函数不会被频繁的调用，代码简洁，性能更高，跟函数和属性（函数）

let是有块级作用域而var没有块级作用域定义变量 const是常量

Es5和Es6的区别

computed:{

里面跟函数 例如：fullName:function(){ 对应的就是get方法

return 一个结果 this.firstName+””+this.lastName

} 当成一个属性去使用的{{fullName}}

完整的计算属性 一般没有set方法，只读属性，没有set方法

fullName:{

set:function(newValue){ newValue等价于this.firstName+””+this.lastName

保存const names=newValue.split(‘’); 拿到一个数组

this.firstName=names[0];

this.lastName=names[1];

写

},

get:function(){

return this.firstName+””+this.lastName 返回一个结果 读

}

}

}

比较methods:{

里面跟方法 例如：fullName(){

return 一个结果

} 使用{{fullName()}}

}

### let/var

我们可以将let看成更完美的var，

var可以看作是全局作用域，没有块级作用域引起的问题，if的块级和for的块级的问题，赋值可以被改变，造成了全局污染。解决方案：

闭包处理

(function(i){})(i)函数是有作用域的，或者把var i改成let i

而let是有块级作用域的，不会与外面变量名产生冲突，局部变量

### const

将某个变量修饰为常量，不可以再次赋值，来保证数据的安全性。建议：在ES6研发中，优先使用const，，只有需要改变某个标识符的时候才使用let。

注意一：一旦给const修饰的标识符被赋值之后，不能修改。注意二：必须进行赋值。注意三：常量的含义是指向的对象不能修改（对象内存地址没变，指向属性），但是可以改变对象内部的属性（例子app不能改变，可以改变app里面data中的属性）

### 增强写法

ES5写法

const name=’why’;

const obj={

name:name,

}

get:function(){}

ES6写法

const name=’why’;

const obj={

name,

}

get(){}

### v-on参数

v-on语法糖@，v-on:click,v-on:keyup键盘按下抬起才会有效果

如果该方法不需要额外参数，那么调用方法后的（）可以不添加。但是注意：如果方法本身中有一个参数，那么调用方法不加（）会默认将原来事件event参数传递进去，那么调用方法加（）没有传参结果为undefind。如果方法有两个参数，那么调用方法不加（）会显示结果undefind，加（）传参（123，event）找不到属性event,加入（123，$event）可以获取event对象，传参加引号，不加引号就是变量。

如果需要同时传入某个参数，同时需要event时，可以**通过$event**传入事件

### v-on修饰符

阻止冒泡,v-on:click.stop。

@click.prevent，阻止默认提交。

@keyup监听键盘的键帽，按下抬起才会有效果

@keyup.enter监听enter的点击

@click.native监听组件里的属性

@click.once只触发一次回调

### v-if,v-else-if,v-else

一般简单的判断用v-if和v-else，复杂的判断不采用v-else-if，用方法调用。

### key

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，而不是创建新的元素。

### v-show和v-if的区别

v-if当条件为false时，压根不会有对应的元素在DOM中。

v-show当条件为false时，仅仅是将元素的display属性设置为none而已。

当需要在显示与隐藏之间切片很频繁时，使用v-show。

当只有一次切换时，通过使用v-if。

### v-for遍历数组和对象

遍历数组(item,index) in info

遍历对象(value,key,index) in info

Item,value都是值，index索引号，key键。

### v-for绑定和非绑定key的区别

key:和Vue的虚拟DOM的Diff算法有关系。

ABCDE,在B和C之间加一个F，Diff算法默认执行起来是这样的。即把C更新成F，D更新成C，E更新成D，最后再插入E，效率太低。

所以我们需要使用key来给每一个节点做一个唯一标识。

Diff算法就可以正确的识别此节点。

找到正确的位置区插入新的节点。

key的作用主要是为了高效的更新虚拟DOM。

<li v-for=”item in letters” :key=”item”>{{item}}</li>

### 数组中响应式的方法

pop删除最后一个元素

shift删除第一个元素

unshift在最前面添加一个或者多个元素

push在最后面添加一个或者多个元素

splice(从第几个元素删除，删除几个元素<如果没有传就删除后面所有元素>，替换前面删除元素)

sort排序

reverse反转

function sum(...num){  
}

sum(可以传递任意参数)

注意：通过索引值修改数组中的元素不是响应式的，解决：1.通过splice方法解决，2.Vue.set(修改的对象，索引值，修改后的值)

### v-model及修饰符

Vue中使用v-model指令来实现表单元素和数据的双向绑定。

v-bind:value+v-on:input=v-model,event.target.value获取输入的值(方法函数里使用，方法函数里面可以用this.$emit,传出this.dnumber参数)，event调用方法传参。

<input type=”text” v-bind:value=”dnumber1” @input=”dnumber1=$event.target.value”>

$event传入event事件对象,在Div里用$。

radio,name=”sex”（必须加形成互斥）,用这个v-model不用加，checkbox，select,multiple。

lazy修饰符：

v-mode.lazy输入没有效果，只有敲回车才会有效果。

官方：默认情况下，v-model默认是在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失去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时字母还是数字，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做数字处理。

number修饰符可以让在输入框中输入的内容自动转成数字类型。typeof age。

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。

### input值绑定

<label v-for=”item in originHobbies” :for=”item”>

<input type=”checkbox” :value=”item” :id=”item” v-model=”hobbies”>{{item}}

</label>

hobbies:[]

orginHobbies:[“篮球”,”足球”]

在真实开发中，这些input的值可能是从网络或定义在data中的。

所以我们可以通过v-bind:value动态的给value绑定值。

## 项目结构

### 数组对象（index）

main.js里面创建Vue实例对象，管理div中id的app，定义数据，挂载要管理的数据。Index.html进行展示遍历main.js里面的数组对象，内层遍历对象<拿到对象里面的属性可以进行操作>。

toFixed(2)保留两位小数。

通过索引号来获取对象，调用对象的属性，索引号index用来传参，v-bind:disabled=”item.count<=1”,当数量为的时候不能点击。

splice((index,1),移除当前元素。

### 过滤器

Vue实例中的filters:{

showPrice(price){

return “￥” +price.toFied(2)

}

}

调用{{item.price | 过滤器}}

### 遍历数组对象取值

1.for(let i=0;i<this.books.length;i++){}

2.for(let i in this.books){}中i拿到的也是索引值index。

3.for(let item of this.books){}拿到的是一个对象item,item=this.books[i]

调用函数+返回值

## 组件化

### 组件化思想

如果我们将一个页面中所有的处理逻辑全部放在一起，处理起来就会变得非常复杂，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小的功能块，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得非常容易了。

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颗组件树。

### 组件化使用

组件的使用分成三步骤：

创建组件构造器：构造器常量接受 Vue.extend(构造器对象模板)方法继承，

注册组件：Vue.component(构造器名称《注册组件的标签名》，构造器常量《组件构造器》<语法糖构造器对象模板>)方法。

全局组件，意味着可以在多个Vue的实例下面使用

局部组件，在Vue实例下面创建，components:{

cpn:cpnC<语法糖构造器对象模板> //cpn使用组件时的标签名，cpnC组件构造器

}

使用组件：在Vue实例的作用范围内，构造器名称

root组件 父组件 子组件

注册组件语法糖：主要是省去了调用Vue.extend()步骤，而是可以直接使用一个对象来代替。

### 组件模板抽离

1.<script>标签

<script type=”text/x-template” id=”cpn”>

<div>

</div>

</script>

2.<template>标签

<template id=”cpn”>

<div> //模板根

</div>

</template>

Vue.component(“cpn”,{

template:”#cpn”

})

### 为什么组件data必须是函数

组件是一个单独功能模块的封装，这个模块属于自己的HTML模块，也应该有属于自己的数据data。

如果所有的数据都放到在Vue实例中，Vue实例就会变得非常臃肿。

Vue组件应该有自己保存数据的地方。

data(){

return{

}

}

而且这个函数返回一个对象，对象内部保存着数据。

单独的作用域，一个组件的数据变化，对其他组件数据不会影响，函数的对象不是不同的，地址不同，函数的作用域。

函数返回的对象其实就是一个内存地址（不同变量引用一个对象）。

### 父传子组件通信

父组件通过props(properties)属性向子组件传递数据。

在子组件中，使用选项props来声明需要从父级接受到的数据。

props的值有两种方式：

方式一：字符串数组，数组中的字符串就是传递时的名称。

方式二：对象，对象可以设置传递时的类型，也可以设置默认值等。

<cpn :cmovies=”movies” :cmessage=”message”></cpn>

props:[“cmovies”,”cmessage”]

父组件注册父组件，子组件构造器props用来存储父组件属性，构造器对象模板，使用子组件构造器（将父组件属性赋值给子组件属性）。

props数据验证，支持的数据类型：

方式二：类型限制

String Number Boolean Array Object Date Function Symbol

提供一些默认值

type default required

类型是对象或者数组时，默认值必须是一个函数。

props驼峰标识 v-bind不支持驼峰 props中的驼峰标识cInfo改成 div组件中使用:c-info。

template模板中用div包裹。

app父组件，cpn子组件，div父组件模板，template子组件模板。

### 子传父组件通信

子组件通过$emit发送事件向父组件发送消息。

btnClick(item){

this.$emit(“item-click”,item) item-click子传父定义事件名称，item子组件参数

}

在<cpn @item-click=”cpnClick”></cpn> 默认传递event item

在父组件定义事件

cpnClick(item){

console.log(“”,item)

}

自定义事件：

当子组件需要向父组件传递数据时，就要用到自定义事件了。

v-on不仅仅可以用于监听DOM事件，也可以用于组件间的自定义事件。

流程：

在子组件中，通过$emit()来触发事件。

在父组件中，通过v-on来监听子组件事件。

### 父传子组件通信-双向绑定

props中的对象属性需要初始化属性

data(){

return{

props属性绑定在这里面，才能进行双向绑定

}

}

理解：

data{

num1:1,

num2:0

}

props{

number1:Number,

number2:Number

}

data(){

return{

dnumber1:this.number1,

dnumber2:this.number2

}

}

methods:{

num1input(event){

dnumber1=event.target.value

this.$emit(“num1change”,dnumber1)

}

num2input(){}

}

@num1change=”num1change”

methods:{

num1change(value){

this.num1=value

}

}

子组件进行双向绑定，需要通过data函数进行绑定，用v-bind:value,@input1=num1input,@input2=num2input

也可以通过watch来实现，watch就是监听某个属性的改变，对象类型。

watch:{

dnumber1(newvalue){

this.dnumber2=newvalue\*100;

this.$emit(“num1change”,newvalue);

}

}

### 父访问子-children-refs

父组件访问子组件：使用$children或$refs(refrence引用)。$children的访问：this.$children[0].调用方法是一个数组类型，它包含所有子组件对象,下标拿子组件对象不好，不知道子组件有多少个，一般拿所有子组件。通过一个遍历，取出所有子组件的message状态。组件一般使用双标签。一般用$refs，对象类型，默认是一个空的对象。

<cpn ref=”aaa”></cpn>

调用this.$refs.aaa.name。

### 子访问父-parent-root

子组件访问父组件：使用$parent，用的比较少，访问的是VueComponent。

this.$root,访问的是Vue实例，用的比较少。

## 插槽

### 基本使用

组件的插槽：

组件的插槽也是为了让我们封装的组件更加具有扩展性，可以预留一定的空间，使用同一个组件，可以展示不同的内容，用插槽进行修饰（添加不同内容）。

让使用者可以决定组件内部的一些内容到底是什么。

抽取共性，保留不同。

最好的封装方式就是将共性抽取到组件中，将不同暴露为插槽。

一旦我们预留了插槽，就可以让使用者根据自己的需求，决定插槽中插入什么内容。

是搜索框，还是文字，还是菜单。由调用者自己来决定。

父：<cpn></cpn>组件里面就是插槽的内容(内容替换到插槽里面)，组件模板里面子：<slot></slot>插槽里面是插槽默认的内容，<cpn></cpn>组件里面有内容就替换，没有就是默认内容。

### 具名插槽

组件模板里面有多个插槽，替换第几个插槽默认的内容，给插槽起个名字。

<slot name=”center”></slot>

<cpn></cpn>组件里面<span slot=”center”></span>

### 编译作用域

<cpn v-bind:>一般去 Vue实例里面找，不会去组件里面找。

而在模板中一般去组件里面找。组件的作用域。

父组件模板的所有东西都会在父级作用域内编译；子组件模板的所有东西都会在子级作用域内编译。

### 作用域插槽

父组件替换插槽的标签，但是内容由子组件来提供。可以展示不同的样式。

通过父组件模板插槽获取子组件模板的子组件属性。

<cpn>

<template slot-scope=”slot”> //slot引用了子组件模板

<span v-for=”item in slot.data”>{{item}-}</span>

//<span>slot.data.join(“-”)</span>

</template>

</cpn>

<slot :data=”planguage”> //子组件属性planguage

</slot>

## 模块化

### 模块化的导入和导出

解决全局变量同名的问题。

解决方案ES5：var moduleA=(function(){

Var name=”小明”

Var obj={}

Var obj.name=name;

return obj;

})() //匿名函数的作用域 返回obj对象等价于moduleA对象接受。

CommonJS文件的导出，ES6：module.exports={

name

}

var {name}=require(“相对文件途径”)

var 文件名=require(“相对文件路径”)

var name=文件名.name;

最简单的写法：

模块作用域

<script src=”aaa.js” type=”module”></script>type=”module”解决命名冲突的问题

导出一：

export{

name

}

导出二：

export var name=”小明”;

导出函数或类：

可以先定义变量函数类，再导出。也可以直接导出变量函数类，类的使用，创建类对象，调用类方法。

某些情况下，一个模块包括某个功能，我们并不希望给这个功能命名，而且让导入者可以自己来命名。这个时候就可以用export default。

export default name

Import nam from “.../aaa.js”

默认的只能有一个，nam就是name的导出，nam名字可以随便取。

导入：

Import {name} from “./aaa.js”

我们使用export指令导出了模块对外提供的接口，下面我们就可以通过Import命令来加载对应的这个模块了。

首先，我们需要在HTML代码中引入两个js文件，并且类型需要设置为module。

变量太多，可以通过\*导入模块中所有export变量。

但通常情况下我们需要给\*起一个别名，方便后续的使用。

Import \* as info from “.../aaa.js” info.name info相当于一个导出对象。

## Webpack

### 模块化概念

从本质上来讲，webpack是一个现代的JavaScript应用的静态模块打包工具。

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间的依赖关系。

而且不仅仅是JavaScript文件，我们的CSS、图片、json文件等等在Webpack中都可以被当作模块来使用。

### 打包概念

就是将webpack中的各种资源模块进行打包并合成一个或多个包（Bundle）。

并且在打包的过程中，还可以对资源进行处理，比如压缩图片，将scss转化成css，将ES6语法转成ES5语法，将TypeScript转化成JavaScript等等操作。

Webpack模块化打包，为了可以正常的运行，必须依赖node环境，node环境为了可以正常执行很多代码，必须其中包含各种依赖的包，npm工具（node package manager）包管理工具。

### webpack安装

查看版本：电脑cmd中输入node -v

全局安装webpack：npm install [webpack@3.6.0](mailto:webpack@3.6.0) -g

局部安装webpack:cd 对应目录 npm install [webpack@3.6.0](mailto:webpack@3.6.0) --save-dev

--save-dev是开发时依赖，项目打包后不需要继续使用。

为什么全局安装后，还要局部安装呢？

在终端直接执行webpack命令，使用的是全局安装的webpack。

当在package.json中定义了script时，其中包含了webpack命令，那么使用的是局部webpack。

### 基本使用过程

dist(distribution发布),打包一般放在这里。

用CommonJs或ES6模块化规范来导出导入Js文件。

用CommonJs或ES6模块化规范来导出导入Css文件。

require(“./css/normal.css”)//依赖css文件

终端使用webpack ./src/main.js ./dist/bundle.js 自动处理模块化的依赖。

在html引用<script src=”./dist/bundle.js”></script>

### 打包简化webpack.config.js配置

终端使用webpack ./src/main.js ./dist/bundle.js通过webpack.config.js改成了终端使用webpack进行**打包**。操作如下：

const path=require(“path”)//依赖node一些包，在全局里找path

module.exports={

entry:”./src/main.js”,

//output:”./dist/bundle.js”,

output:{

path:”path.resolve(\_\_dirname,”dist”)”,//动态获取绝对路径 .dist需要的是绝对路径。

\_\_dirname是node里面的内容，可以获取当前文件路径。

filename:”bundle.js”,

publicPath:”dist/”

},

module: {

rules: [

{

test: /\.css$/, //\.对点进行转义，$结束

//css-loader只负责将css文件进行加载

//style-loader负责将样式添加到DOM中

//使用多个loader时，是从右向左顺序读取的

use: [ 'style-loader', 'css-loader' ]

},

{

test: /\.less$/,

use: [{

loader: "style-loader" // creates style nodes from JS strings

}, {

loader: "css-loader" // translates CSS into CommonJS

}, {

loader: "less-loader" // compiles Less to CSS

}]

},

{

test: /\.(png|jpg|gif|jpeg)$/,//test用来匹配的

use: [ //使用哪一个loader

{

loader: 'url-loader',

options: {

limit: 8192, //图片大小，小于limit时(小于8kb)，会将图片编译成base64字符串形式，大//于limit时(大于8kb)，会使用file-loader模块进行加载。npm install --save-dev file-loader.在//webpack.config.js中配置publicPath:”/dist”

name:”img/[name].[hash:8].[ext]”

},

}

]

}

]

}

}

动态获取路径：

将path放进一个包中，**npm init对该项目进行初始化**，起一个包的名字，包的名字不能有中文和符号，初始化项目后生成package.json文件，npm install，安装package.json等一些依赖。

在package.json文件中”scripts”:{“build”:”webpack”},脚本中添加build，在终端中运行npm run build，进行了打包。

全局webpack与本地webpack，一般用本地webpack，在终端中npm intall [webpack@3.6.0](mailto:webpack@3.6.0) --save-dev安装本地webpack。

在package.json会多出:

“devDependencies”:{ //开发时依赖

“webpack”:”^3.6.0”

},

“dependencies”:{} //与开发时依赖对立

开发时依赖，运行时依赖，只要开发时依赖。

cmd终端和软件终端都是全局webpack,本地webpack在软件终端运行./node\_modules/.bin/webpack，脚本上”build”:”webpack”优先找本地webpack。

package.json中的scripts的脚本在执行时，会按照一定的顺序寻找命令对应的位置。

首先，会寻找本地node\_modules/bin/webpack路径中对应的命令。

如果没有找到，会去全局的环境变量中寻找。

文件和文件夹解析：

dist文件夹：用于存放之后打包的文件

src文件夹：用于存放我们写的源文件

main.js：项目的入口文件。

mainUtils.js：定义了一些数学工具函数，可以在其他地方引用，并且使用。

index.html：浏览器打开展示的首页html。  
package.json通过npm init生成的，npm包管理的文件。

### webpack中css文件配置

loader使用过程：https://www.webpackjs.com/loaders/css-loader/

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

终端中npm install --save-dev css-loader

npm install --save-dev style-loader 在main.js里面进行引用依赖css文件依赖，安装loader，在webpack.config.js进行配置。版本要一致。

### webpack-less文件的处理

在main.js里面进行引用依赖require(“./css/special.less”)，npm run build,进入package.json找脚本里面的build执行webpack,进入webpack.config.js找到对应的入口进行编译，入口是main.js，找到所有对应的依赖less文件依赖。https://www.webpackjs.com/，LOADERS，样式，安装loader，npm install --save-dev less-loader less，在webpack.config.js进行配置。版本要一致。

### webpack-图片文件处理

https://www.webpackjs.com/，LOADERS，文件，安装loader，

npm install --save-dev url-loader

在webpack.config.js进行配置。版本要一致。

我们发现webpack自动帮助我们生成一个非常长的名字

这是一个32位hash值，目的是防止名字重复

但是在真实的开发中，我们可能对打包的图片名字有一定的要求

比如，将所有的图片放在一个文件夹中，跟上图片原来的名称，同时也要防止重复img/name/hash8

所以，我们可以在options中添加如下选项：

img:文件打包到的文件夹

name:获取图片原来的名字，放在该位置

hash:8:为了防止图片名称冲突，依然使用hash，但是我们只保留8位

ext:使用图片原来的扩展名

### webpack-ES6转ES5的babel

ES6语法并没有转成ES5，那么就意味着可能一些对ES5还不支持的浏览器没有办法很好的运行我们的代码。

将ES6转成ES5，那么就需要使用babel。

而在webpack中，我们直接使用babel对应的loader就可以了。

npm install --save-dev babel-loader@7 babel-core babel-preset-es2015

配置webpack.config.js文件

{

test: /\.js$/,

exclude: /(node\_modules|bower\_components)/,

use: {

loader: 'babel-loader',

options: {

presets: ['es2015']

}

}

}

const相关的东西全部改成var类型的。

### webpack-使用Vue的配置过程

现在，我们希望在项目重用Vue.js，那么必然需要对其有依赖，所哟需要先进行安装，注意：用为我们后续实际项目中也会使用vue的，所以并不是开发时依赖。

npm install vue --save 开发、运行时候也依赖。

使用Vue进行开发，安装之后进行依赖 import Vue from “vue”,进行使用Vue实例，在main.js中使用new Vue{}。

runtime-only 代码中，不可以有任何的template

runtime-compiler 代码中，可以有template，因为有compiler可以用于编译template。

在webpack.config.js进行配置，resolve和module并列

resolve:{

//alias别名

alias:{

“vue$”:”vue/dist/vue.esm.js”

//在node\_modules查找，就是选择runtime-compiler

}

}

SPA(simple page web aoolication)->vue-router(前端路由)

如果我们希望将data中的数据显示在界面中，就必须是修改index.html

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html模板在之后的开发中，我并不希望手动的来频繁修改，是否可以做到呢？

在根组件new Vue{}里面定义template属性：

在前面的Vue实例中，我们定义了el属性，用于和index.html中的#app进行绑定，让Vue实例之后可以管理它和其中的内容。

这里我们可以将div元素中的{{message}}内容删掉，只保留一个基本的id为div元素。

但是如果我依然希望在其中显示{{message}}的内容，应该怎么处理呢？  
我们可以再定义一个template属性。template里面的代码会自动替换el-><div id=”app”></div>里面的{{message}}代码。

tempate里面的代码可以通过组件进行抽离

在new Vue{

el:”#app”,

template:”</App>”,

components:{

App

}

}

可以将组件写到另一个文件夹中进行默认导

在main.js中进行导入import App from “./vue/app”

在app.js转化成App.vue，将组件写到App.vue中<template></template>，

js代码写到<script></script>中，import App from “

./vue/App.vue”，app.js可以省略。在App.vue中name:”App”是自动生成的。

App.vue不能被加载编译，这种特殊的文件及特殊的格式要处理，进行安装：

npm install vue-loader vue-template-compiler --save-dev

所有的loader都需要配置。

{

test:/\.vue$/,

use:[“vue-loader”]

}

package.json文件中看版本号

“vue-loader”:”^13.0.0”,^表示大于等于，

改package.json文件，需要重新安装，执行npm install(node\_modules)

解决import App from “./vue/App.vue”，省略.vue，在webpack.config.js进行配置

resolve:{

extensions:[“js”,”css”,”.vue”]

}

### webpack-横幅Plugin的使用

plugin是插件的意思，通常是用于对某个现有的架构进行扩展。

webpack中的插件，就是对webpack现有功能的各种扩展，比如打包优化，文件压缩等等。

loader和plugin的区别

loader主要用于转换某些类型的模块，它是一个转换器。

plugin是插件，它是对webpack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要安装）

步骤二：在webpack.config.js中的plugins中配置插件

我们先来使用一个最简单的插件，为打包的文件添加版权声明

该插件名字叫BannerPlugin，属于webpack自带的插件

按照下面的方式来修改webpack.config.js的文件：

const webpack=require(“webpack”)会在node\_modules查找webpack

plugins与modules并列

plugins:{

new webpack.BannerPlugin(“最终版权归coderwhy所有”)

}

重新打包程序：查看bundle.js文件的头部，看到如下信息

/\*最终版权归coderwhy所有\*/

### webpack-HtmlWebpackPlugin的使用

目前，我们的index.html文件是存在项目根目录下的。

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等文件也就没有意义了。

所以，我们需要将index.html文件打包到dist文件中，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

安装HtmlWebpackPlugin插件

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

使用插件，修改webpack.config.js文件中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:”dist/”属性

否则插入的script标签中的src可能会有问题。

删除index.html中script标签

修改webpack.config.js文件：

const HtmlwebpackPlugin=require(“html webpack plugin”)

plugins:[

new HtmlwebpackPlugin({

template:”index.html”

})

]

### webpack-UglifyjsWebpackPlugin使用

在项目发布之前，我们必然需要对js等文件进行压缩处理。

我们就对打包的bundle.js文件进行压缩，安装插件，版本号1.1.1，和CLI2保持一致。

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

修改webpack.config.js文件，使用插件：

const UglifyJsPlugin=require(“uglifyjs-webpack-plugin”)

plugins:{

new uglifyJsPlugin()

}

### webpack-dev-server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中使用之前需要安装它

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

devserver也是作为webpack中的一个选项，选项本身可以设置如下属性：

contenBase：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：在SPA页面中，依赖HTML5的history模式

webpack.config.js文件配置修改如下：

和module并列

devServer:{

contenBase:”./dist”,

inline:true

}

我们可以再配置另外一个scripts:

--open参数表示直接打开浏览器

在package.json文件中script中配置

scripts:{

“dev”:”webpack-dev-server --open” //--open自动打开

}

终端运行：npm run dev

开发阶段不建议使用丑化插件UglifyjsWebpackPlugin，发布阶段需要

devServer开发时需要，运行不需要

### webpack配置文件的分离

分成三种base.config.js,dev.config.js,prod.config.js。

dev.config.js文件里面放devServer。

prod.config.js文件里面放

const UglifyjsWebpackPlugin=require(“uglifyjs-webpack-plugin”)

plugins:[

new UglifyjsWebpackPlugin()

]

其他代码都放入base.config.js文件中

安装npm install webpack-merge --save-dev

在prod.config.js导入

const webpackMerge=require(“webpack-merge”)

const baseConfig=require(“./base.config”)

module.exports=webpackMerge(baseConfig,{

})

在package.json文件中

“build”:”webpack --config ./build/prod.config.js”

“dev”:”webpack-dev-server --open --config ./build/dev.config.js”

修改base.config.js文件中

output:{

path:path.resolve(\_\_dirname,”../dist”)

}

## 8.vuecli脚手架

### vuecli-脚手架的介绍和安装

如果你只是简单写几个Vue的Demo程序，那么你就不需要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，那么无疑效果比较低，所以通常我们会使用一些脚手架工具来帮助完成这些事情。

CLI是Command-Line Interface,翻译为命令行界面，俗称脚手架。

Vue CLI是一个官方发布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。

使用前提-node

安装NodeJS

可以直接在官方网站中下载安装

网址:http://nodejs.cn/download/

检查安装的版本

默认情况下自动安装Node和NPM

Node环境要求8.9以上或者更高版本

NPM的全称Node Package Manager

是一个NodeJS包管理和分发工具，已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的依赖包。

cnpm安装可以加快安装速率

由于国内直接使用npm官方镜像是非常慢的，这里推荐使用淘宝NPM镜像。

你可以使用淘宝定制的cnpm（gzip压缩支持）命令行工具代替默认的npm：

npm install -g cnpm -registry=https://registry.npm.taobao.org

这样就可以使用cnpm命令来安装模块了：

cnpm install [name]

Vue.js官方脚手架工具就可以使用了webpack模板

对所有的资源会压缩等优化操作

它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

安装Vue脚手架

npm install -g @vue/cli 安装到全局就可以了

注意：上面安装的使Vue CLI3的版本，如果需要想按照Vue CLI2的方式初始化项目时不可以的。

查看脚手架版本：终端vue --version

拉取 2.x 模板（旧版）可以使用Vue CLI2:

Vue CLI >= 3 使用相同的vue二进制文件，因此它会覆盖 Vue CLI 2 ( vue-cli)。如果您仍然需要旧版vue init功能，您可以安装一个全局网桥：

npm install -g @vue/cli-init

vue init now works exactly the same as [vue-cli@2.x](mailto:vue-cli@2.x)

Vue CLI2初始化项目

vue init webpack my-project（项目名字不能是中文）

Vue CLI3初始化项目

vue create my-project

### vuecli-CLI2初始化项目过程

vue init webpack my-project

Project name 项目名字 enter就是默认之前的名字

Project description 描述信息 这些信息都会保存在package.json里面

Author 作者，默认git作者 .gitconfig

Vue build

Runtime +Compile暂时选择

Runtime +only webpack默认 大多数项目用这个 优点 文件小 运行效率更高

推荐使用Runtime + Compile

Install vue-router 可以暂时不安装（可以手动安装）

大型项目要用Vue全家桶 VueCore+vue-router+vuex

Use ESLint to lint your code 限制代码 选择的话yes

Pick an ESLint preset(Use arrow keys)

Standard标准 暂时用标准的

Airbnb爱彼迎 民宿

none可以自己来配置规范

Set up unit tests 单元测试 不要

Setup e2e tests with Nightwatch e2e end to end端到端测试->selenium自动测试 n

Yes,use NPM 选择

Yes,use Yarn 谷歌联合其他公司开发管理工具

No,I will handle that myself

Users->Adminstrator .gitconfig可以修改作者Author

### vuecli-CLI2目录结构解析

build文件目录里：

先看package.json文件，”dev”和”start”效果一样，”lint”对代码做限制的

以前js的运行依赖浏览器，只能跑在浏览器上，现在node为js提供了一个运行环境，node使用C++开发的，有V8引擎（谷歌开发的）

js:js->字节码->浏览器

V8引擎（C++，开源，底层支撑）:js->二进制代码（机器语言识别，效率高）

终端输入node 文件名.js，可以在终端上操作

再看build.js，先删除原来打包，再重新打包，配置文件的分离。

config文件目录里：

index.js里定义了变量，dev变量，build变量。

build文件和config文件都是webpack配置的内容

node\_modules文件目录就是当前项目依赖的很多node的包，去package.json文件看依赖包，分开发时依赖（一个”vue”）和运行时依赖（依赖很多）

src文件写代码，static文件静态资源（原封不动的复制到dist文件中，.gitkeep如果没有的话，可能不会上传到服务器。如果有的话，不管有没有，都会上传到服务器）

.babelrc文件ES6转化成ES5

其中”browsers”:[“>1%”,”last 2 version”,”not ie <=8”]

浏览器市场份额大于1%且最后两个版本且ie小于等于8就不用考虑了

“stage-2”转化的代码只针对这个阶段的代码

.editorconfig文件进行代码的统一，编码，缩进风格，缩进大小，换行，最后插入一行，清除空格。

.eslintignore文件代码不规范忽略

.gitignore文件上传服务器中一些东西不需要上传

.eslintrc.js文件代码检测配置一些东西

.postcssrc.js文件是css进行转化的时候配置的东西

package.json文件版本号是大于等于的，~最后一个可以变，^前面两个可以变

package-lock.json文件锁定版本号，当你再次npm install的时候会自动安装最新版本，但是有lock就按照原先的版本。

README.md文件写一些文档之内的东西，说明书

### 安装CLI错误和过程ESLint规范

安装CLI在终端中以管理员身份运行，如果安装不成功输入npm clean cache --force 或者直接进入C:\Users\Administrator\AppData\Roaming\npm-cache

npm run dev来到package.json文件中”dev”运行dev.config.js，通过dev.config.js找到main.js的入口，main.js文件通过el:”#app”管理index.html文件，导入App组件又通过components:{App}管理组件，又通过template:{</App>}替换index.html中div内容。

App组件相当于根组件，App组件可以导入其他子组件进行管理。

ESLint规范：最后一个代码不需要加,函数空格，必须调用函数，最后一个代码也不要加分号

怎么关闭ESLint

在config文件中找到index.js，找到代码useEslint:true改成false

### runtime-compiler和runtime-only的区别

runtime-compiler中template:”<App/>” components:{App}

template模板传给vue，vue会进行保存，vue保存在vue实例下的options。

进行pass（解析），解析成ast(抽象语法树 abstract syntax tree)，compiler编译成render函数（function）最终翻译成virtual虚拟 dom，进行渲染成真实的dom。

template->ast->render->virtual dom->UI真实dom

runtime-only中render:h=>h(App) render渲染 箭头函数 h(createElement)参数

1.createElement(“标签”,

{标签的属性},

[“Hello World”,createElement(“button”,[“按钮”])  
])

2.传入组件（对象）

箭头函数转换 render:function(h){return h(App)}

将template编译成了render函数

那么.vue文件中的template是由谁处理的了？

是由vue-template-compiler处理的 转成render函数 runtime-only

1.性能更多 2.下面的代码量更少(少6kb就是这么来的)

render->virtual dom->UI 真实dom

### VueCLI3创建项目和目录结构

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

vue-cli3的设计原则是”0配置”,移除的配置文件根目录下的build和config等目录

vue-cli3提供了vue ui命令，提供了可视化配置，更加人性化

移除了static文件夹，新增了puhlic文件夹，并且index.html移动到public中

public文件夹里面的东西也是原封不动的复制到dist里面。

vue create 项目名称

请选择一个配置preset

default(babel,eslint)

Manually(手动) select features 选择

选择加空格 只选择Babel 回车

PWA推送通知 TypeScriptES6语法集合

独立（dedicated）的配置文件 选择

package.json

保存未来项目的配置 yes

保存配置名字coderwhy（之前有两个，保存后有三个） 怎么删除

找到C:\Users\Administrator\.vuerc，找到”presets”删除 rc（run command）

安装依赖时，选择使用的包管理 NPM

VCS->version control system版本控制系统（git/svn），.git默认

git init git add. git commit -m “” git push

main.js中vue配置产品提示关闭

.mount(“#app”)等价el:”#app”

### VueCLI3配置文件的查看和修改

UI方面的配置

启动配置服务器：vue ui也可以创建管理项目

导入一个项目进行管理，看真实版本，也可以进入node\_modules找到vue,vue.js

运行依赖vue 开发依赖vue-template-compiler 版本一致

安装依赖 项目配置 基本配置(webpack) 输出目录dist

运行时编译runtime-compiler

任务 serve 运行

查看build,config文件，在node\_modules中打开@vue,找到cli-service,找到webpack.config.js文件，找到lib，找到Service.js引用其他文件配置

修改 创建一个文件vue.config.js 该文件显示绿色，git commit -m “”

上传文件都是在本地仓库.git中

文件中写修改配置,module.exports={}，修改后的配置会和隐藏的配置进行合并

文件名字固定不能修改 git status 查看修改状态

git add. git status gti commit -m “”

### 箭头函数的使用和this指向

箭头函数：也是一种定义函数的方式

1.定义函数的方式：function

const aaa=function(){}

1. 对象字面量中定义函数

const obj={

aaa:function(){},

aaa(){}

}

1. ES6中的箭头函数

const aaa=(参数列表)=>{} 一个参数不需要括号，两个需要

函数代码块中只有一行代码

const mul=(num1,num2)=>num1\*num2 有返回值的

const mul=()=>console.log(“Hello World”)没有返回值,mul()返回值为undefined

结果作为箭头函数的返回值

什么时候使用箭头函数

函数里面使用其他函数时候

结论：箭头函数中的this是如何查找的了？

答案：向外层作用域中，一层层查找this，直到有this的定义。引用的就是最近作用域中的this。

## 9.Vue-Router

### 什么是路由和其中映射的关系

路由是一个网络工程师里面的术语

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。

路由器提供了两种机制：路由和传送

路由是决定数据包从来源到目的地的路径

传送将输入端的数据转移到合适的输出端

路由表本质上就是一个映射表决定了数据包的指向

映射表：{内网ip1:电脑mac1地址 内网ip2:电脑mac2 地址:电脑}

### 前端路由和后端路由（渲染）

后端渲染：html+css+js(早期没有js，用jsp/php代替)，jsp:java server page

浏览器输入网站发送到服务器

服务器：html css java代码作用是从数据库中读取数据，并且将它动态的放在页面中（已经是一个网页了） 传到浏览器 只有html css 服务端渲染已经完成了

url与网页形成了映射关系

后端路由:**后端处理URL和页面之间的映射关系**

早期的网站开发整个HTML页面是由服务器来渲染的

**服务器直接生产渲染好对应的HTML页面，返回给客户端进行展示**

但是，一个网站，这么多页面服务器如何处理呢？

**一个页面会有自己对应的网址**，也就是URL

URL会发送到服务器，服务器会通过**正则对该URL进行匹配**，并且最后交给一个Controller（控制器）进行处理

Controller进行各种处理，最终生成HTML或者数据，返回给前端

这就完成了一个IO操作。

上面的这种操作，就是后端路由

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

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

后端路由的**缺点**：

一种情况是整个页面的模块由**后端人员来编写和维护的**

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

而且通常情况下HTML代码和数据以及对应的**逻辑**会混合在一起，编写和维护都是非常糟糕的事情。（前后端分离）

前后端分离阶段：

随着**Ajax**的出现，有了前后端分离的开发模式。

**后端只提供API来返回数据**，前端通过Ajax获取数据，并且可以通过JavaScript将数据渲染到页面中

这样做最大的优点就是前后端责任的清晰，**后端专注于数据上，前端专注于交互和可视化上**

并且当移动端（IOS/Android）出现后，后端不需要进行任何处理，依然使用之前的一套API即可

后端只负责提供数据，不负责任何阶段的内容

静态资源服务器（传到浏览器html+css直接渲染 js代码浏览器执行 $.ajax(url:api接口:)） 请求资源 提供API接口的服务 将对应的数据返回到浏览器 其他js代码创建div 浏览器进行渲染

前端渲染：浏览器中显示大网页中的大部分内容，都是由前端写的js代码在**浏览器**中执行，最终渲染出来的网页。

**单页面富应用阶段**：

其实SPA（single page application）最主要的特点就是在前后端分离的基础上加了一层前端路由。

就是前端来维护一套路由规则。

真个网页就只有一个html页面，路由就是替换dom

前端路由：**url->页面,一个url对应一个组件，映射关系**

前端路由的核心

改变URL，但是**页面不进行整体的刷新。**

### url的hash和HTML5的history

在console中改变，location.hash=”aaa”

localhost:8080/# 变成了localhost:8080/#/aaa clear清除

不会刷新网页

location.href（hyper reference 超级引用）会刷新网页

刷新网页所有资源都会请求

URL的hash

URL的hash就是锚点（#），本质上是改变window.location的href属性

我们可以通过直接赋值location.hash来改变href，但是页面不发生刷新。

history.pushState({},””,”home”) localhost:8080/home 栈结构 先进后出

history.pushState({},””,”about”) localhost:8080/about

history.back() localhost:8080/home

history.replaceState({},””,”home”) localhost:8080/home

replaceState与pushState区别，replaceState不能进行前进和后退，不是连续的，不可以点击，替换，不是压栈和出栈，不会处于栈顶中

history.back()等价history.go(-1)

history.forword()等价history.go(1)

（栈结构中）这三个接口等同于浏览器界面的前进后退。

### vue-router安装和配置方式

目前前端流行的三大框架，都有自己的路由实现

Angular的ngRouter

React的ReactRouter

Vue的vue-router

vue-router是Vue.js官方的路由插件，它和vue.js是深度集成呢个的，适合用于构建单页面应用。

我们可以访问其官方网站对其进行学习：https://router.vuejs.org/zh/

vue-router是基于路由和组件的

路由用于设定访问路径，将路径和组件映射起来

在vue-router的单页面应用中，页面的路径的改变就是组件的切换

步骤一：安装 npm install vue-router --save

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

第一步：导入路由对象，并且调用Vue.use(VueRouter)

//执行VueRouter.install

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

使用vue-router的步骤：

第一步：创建路由组件

第二步：配置路由映射：组件和路径映射关系

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

import VueRouter from “vue-router”

import Vue from “vue”

import Home from “../components/Home”

通过Vue.use（插件），安装插件

Vue.use(VueRouter) //安装插件

创建VueRouter对象

const routes=[

映射关系，一个映射关系就是一个对象

{

path:””,

//redirect 重定向 默认打开会重定向到localhost:8080/#/home这个路径

redirect:”/home”

},

{

path:”/home”,

component:Home

},

{}

]

配置路由和组件之间的应用关系

const router=new VueRouter({

routes,

//路径上有#都是hash值（默认）修改成history 消除#

mode:”history” //localhost:8080/home

//history.pushState修改的话模式是hash就会路径出错多出#

})

将router对象传入到vue实例

export default router

在main.js导入并挂载

import router from “./router”

在App.vue中写标签

<router-link to=”/home”>首页</router-link> to和path一致

<router-view></router-view>路由组件展示

最终被渲染成a标签<a href=”#/home” class>首页</a>

如果不想渲染成a标签，在<router-link to=”/home” **tag=”button”**>将会被渲染成button标签

我们只是使用了一个属性：to，用于指定跳转的路径

<router-link>还有一些其他属性

tag:tag可以指定<router-link>之后渲染成什么组件，上面代码渲染成了<button>

replace:replace不会留下history记录，所以指定replace的情况下，后退键返回不能返回到上一个页面中。

active-class:当<router-link>对应的路由匹配成功时，会自动给当前元素设置一个router-link-active的class，设置active-class可以修改默认的名称。

在进行高亮显示的导航菜单或者底部tabbar时，会使用到该类

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

在App.vue中设置样式 .router-link-active:{color:red}

修改router-link-active名字

<router-link active-class=”active”> .active:{color:red}

给每个<router-link>修改router-link-active名字

在路由index.js里面修改

VueRouter实例里面添加属性linkActiveClass:”active”

vue-router注册<router-link><router-view>两个全局组件

<router-link>：该标签是一个vue-router中已经内置的组件，它会被渲染成一个<a>标签。

<router-view>：该标签会根据当前的路径，动态渲染出不同的组件。

网页的其他内容：比如顶部的标签/导航，或者底部的一些版权信息等会和<router-view>处于同一等级

在路由切换时，切换的是<router-view>挂载的组件，其他内容不会发生改变。

### 第二种方式：通过代码跳转路由

<button @click=”homeClick”>首页</button>

在App.vue文件中

methods:{

homeClick(){ //this拿到当前组件 vue-router源码加载$router属性

//$全局变量 push=>pushState

this.$router.push(“/home”) //to=”/home”

//this.$router.replace(“/home”)

}

}

### vue-router动态路由的使用

在index.js中routes数组里添加{

path:”/user/:userId”,

component:User

}

<router-link :to=”’/user/’+userIds”></router-link> //属性绑定变量 v-bind

在App.vue中设置属性

data(){

return{

userIds:”lisi”

}

}

在User.vue中

{{$route.params.userId)}}

//$route获取当前活跃的路由组件 而$router是一个路由对象 params参数

### 路由的懒加载箭头函数（打包文件解析）

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（js文件中），然后当路由被访问的时候才加载对应组件，这样就更高效了。

打包中的js文件我分为了三类

app~.js当前应用程序开发的所有代码（业务代码）

manifest~.js为了打包的代码做底层支撑的 (运行时)

([main.js,mathUtils.js,info.js]) \_\_webpack\_require\_\_(moduleId)

vendor~.js提供商第三方vue vue-router/axios/bs

懒加载：用到时，再加载。

首先，我们知道路由中通常会定义很多不同的界面

这个页面最后会被打包在哪里呢？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。

如何避免这种情况的发生呢？使用路由懒加载就可以了。  
import Home from “../components/Home” 打包只有三个js文件

懒加载形式：

改成{

path:”/home”,

component:()=>import (“../components/Home”) //等价于Home变量

}

打包有多个个js文件 将app~.js文件进行分割成多个js文件

一个懒加载会对应一个js文件

方式一.结合Vue的异步组件和webpack的代码分析

const Home =resolve=>{require.ensure([“../components/Home.vue”],

()=>{resolve(require(“../components/Home.vue”))}//认识即可

)}

方式二.AMD写法

const About=resolve=>require([“../components/About.vue”],resolve)

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

const Home=()=>import(“../components/Home.vue”)

### vue-router路由的嵌套使用

嵌套路由是一个很常见的功能

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

一个路径映射一个组件，访问这两个路径也会分别渲染两个组件

路径/home 组件Home

路径/home/news 组件news

实现嵌套路由有两个步骤：

创建对应的子组件，并且在路由映射中配置对应的子路由

在组件内部使用<router-view>标签

const HomeNews=()=>import (“../componets/HomeNews”)

{

path:”./home”,

componet:Home,

children:[

{

path:””,

redirect:”news”

},

{

path:”news”,

componet:HomeNews

}

]

}

在Home.vue文件中使用

<router-link to=”/home/news”>新闻</router-link>

//to需要完整路径 在url中 /代表网站的根目录

<router-view></router-view>

### vue-router参数传递

第一步：创建新的组件Profile.vue

第二步：配置路由映射

第三步：添加跳转的<router-link>

传递参数主要有两种类型:params和query

params的类型：

配置路由格式：/router/:id

传递方式：在path后面跟上对应的值

传递后形成的路径：/router/123,/router/abc

query类型：

配置路由格式：/router,也就是普通配置

传递的方式：对象中使用query的key作为传递方式

传递后形成的路径：/router?id=123,/router?id=abc

<router-link :to=”{path:”/profile”,query:{name:”why”,age=18,height:1.88}}”>

</router-link>

//没有v-bind to后面只会当成普通字符串{}解析，有v-bind就会真正当成语法对象{}去解析

URL（统一资源定位符Uniform Resource Locator）:scheme协议类型://localhost服务器地址主机:8080port端口号/path资源层级UNIX文件路径?query查询fragment片段

在Profile.vue中{{$route.query.name}}

在App.vue文件中：

<button @click=”ProfileClick”>档案</button>

ProfileClick(){

this.$router.push({path:”/profile”,query:{name:”why”,age=18,height:1.88}})

}

### vue-router-route和route由来

router和组件里面this.$router是同一个。

push,replace,go,back,forword（pushState）方法

而route和组件里面this.$route是同一个

哪一个路由组件活跃就赋值给route

params,query方法

routes属于router

所有的组件都继承自Vue的原型

Vue.prototype.test=functuon(){} 调用this.test()

const obj={}

Object.defineProperty(定义属性)(obj,”age”,18)

等价于obj.age=18

Object.defineProperty(Vue.prototype,”$router”,

{get(){return this.\_routerRoot.\_router}})

等价于Vue.prototype.$router=return this.\_routerRoot.\_router

**即$router=router**

this.$options.router就是VueRouter对象里的属性router

this.routerRoot=this this就是Vue实例

**this.\_router=this.$option.router**

Object.defineProperty(Vue.prototype,”$route”,

{get(){return this.\_routerRoot.\_route}})

等价于Vue.prototype.$route=return this.\_routerRoot.\_route

**即$route=route**

$route和$router是有区别的

$router为VueRouter实例，想要导航到不同URL，则使用$router.push方法

$route为当前router跳转对象里面可以获取name、path、query、params等

Vue.component(“RouterView”,View)

Vue.component(“RouterLink”,Link)

<router-view> <router-link>

### 八个生命周期

created一旦组件被创建出来，就会回调这个生命周期函数

mounted组件创建之后，当我们将template挂载到整个dom上之后，就会回调这个生命周期函数

updated界面{{message}}发生更新的时候，只要发生一次更新，就会回调这个生命周期函数。

destoryed销毁，就会回调这个生命函数

new Vue() ->init Events&Lifecycle(beforeCreate)->init injections&reactivity

创建Vue实例后就会初始化事件和生命周期，回调beforeCreate这个函数，初始化响应式（注射和反应），回调created这个函数。问是否有el这个属性，没有el这个属性就会vm.$mount(el)进行回调。如果有el这个属性，再问有没有template这个属性，如果有template这个属性就会将el这个属性替换掉，没有这个属性的话，说明很多东西都会编译成了render函数了，如果有template这个属性，就会编译template属性变成render函数。beforeMount函数之后，就会真正把el属性替换掉（相当于template上的东西被挂载到dom上了），这个时候就会回调mounted（挂载）这个函数，就会进行一种响应式的检测。数据变化时候就会回调beforeUpdate这个函数，就会通过Virtual DOM重新渲染（re-render）并和真实DOM进行对比修改（patch补丁），调用updated这个函数进行循环。不再停留这个界面了（去下个界面），就会调用beforeDestory这个函数，回收很多东西，最后调用destoryed函数，完全销毁。

### vue-router全局导航守卫

created(){

document.title=”首页”

}

这里的跳转都是路由跳转，对路由进行监听

在index.js文件

{path:”/home”,

component:Home,

meta:{

title:”首页” //meta元数据（描述数据的数据）

}

}

//前置守卫(guard) guard守卫 跳转前回调

router.beforeEach(function(to,from,next){})

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

//从from跳转到to

document.title=to.matched[0].meta.title,//matched针对嵌套路由

next(), //必须调用，下一步

})

导航钩子的三个参数解析：

to：即将要进入目标的路由对象

from：当前导航即将要离开的路由对象

next：调用该方法后，才能进入下一个钩子

如果是后置钩子，也就是afterEach，不需要主动调用next()函数

//后置钩子(hook)，钩子就是回调的含义 跳转后回调

router.afterEach((to,from)=>{

})

以上都称之全局守卫

还有路由独享的守卫，

在routes:{

beforeEnter:(to,from,next)=>{next()}

}

组件内的守卫

const foo={

template:””,

beforeRouteEnter(to,from,next){},

beforeRouteUpdate(to,from,next){},

beforeRouteLeave(to,from,next){}

}

登录失败next(“./login”)

### vue-router-keep-alive

keep-alive也是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。//保证活着

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。

router-view也是一个组件，如果直接被包在keep-alive里面，所有路径匹配到的视图组件都会被缓存：

在App.vue文件中

<keep-alive exclude=”Profile,User”>

在Profile.vue文件中name:Profile(用的比较少) 正则不要随便加空格

<router-view></router-view> //只会创建一次（调用一次created函数）

</keep-alive>

在Home.vue文件中

activated(){

this.$router.push(this.path) //使用keep-alive才能引用

},

//deactivated(){

//this.path=this.$route.path

//}

将redirect路由映射清除

在data()里面设置path:”/home/news”属性

用组件内导航守卫

beforeRouterLeave(to,from,next){

this.path=this.$route.path;

next()

}

首页中使用path属性记录离开时的路径，在beforeRouteLeave中记录

## 10.tabber基本结构的构建

### TabBar实现思路

1. 如果在下方又有一个单独的TabBar组件，你如何封装

自定义TabBar组件，在APP中使用

让TabBar出于底部，并且设置相关的样式

1. TabBar中显示的内容由外界决定

定义插槽

flex布局平分TabBar

1. 自定义TabBarItem，可以传入图片和文字

定义TabBarItem，并且定义俩个插槽：图片、文字。

给两个插槽外层包装div，用于设置样式

填充插槽，实现底部TabBar的效果

assets文件创建css文件（base.css），img文件

在main.js导入css require(“”)不太好

在App.vue中<style>中导入css

@import “”; 最合适

导入基本样式外，

封装组件写组件样式display:flex flex:1;

box-shadow:x y 模糊程度 颜色rgba //x y都为正会往右下走

子组件用插槽，父组件写代码内容，父组件里面的代码内容(孙子组件TarBarItem)会替换子组件里面的插槽，子组件就只管TabBar，不会管TabBarItem。继续封装，创建TabBarItem.vue，在<tab-bar><tab-bar-item></tab-bab-item></tab-bar>

组件标签里面的代码替换该对应组件里面的插槽，对应的组件写对应组件的样式，两个插槽可以使用具名插槽。

vertical-align:middle;与图片中间线对齐。

给TabBarItem传入active图片

再定义一个具名插槽用来放入item-icon-active图片

两个具名插槽，通过v-if和v-else进行切换

文字动态绑定class定义颜色red

在插槽中动态绑定class会被App.vue内容替换掉

所以通过div将插槽包裹起来，在div中进行动态绑定

通过div将插槽包裹，可以在div中定义其他属性

components文件（公共抽出来的东西，小组件复用性高的）里面写TabBar、TabBarItem组件。写路由组件时候，创建一个新的文件views，里面写Home、Category、Cart、Profile组件。（组件名首字母都大写，文件名首字母都小写）

### TabBarItem和路由结合效果

实现动态路由跳转路由路径

在TabBarItem.vue中设置@click=”itemClick”

props:{

path:String //通过props动态获取路径 父传子 ，父组件文件中对应该子//组件标签中使用对应的该组件属性

}

data(){

return{

//path:”/home” 不要写死路径

}

}

methods:{

itemClick(){

this.router.replace(this.path)

}

}

在App.vue文件中

<tab-bar-item path:”/home”></tab-bar-item>等四个组件分别设置路径 加:表示传递的是一个变量路径，没加:表示传递的是一个字符串路径。

### 实现TabBarItem的颜色动态控制

在TabBarItem.vue文件中设置

computed:{

isActive(){

//activehome->item(home)=true

return this.$route.path.indexof(this.path)!==-1 就是找到了，等于-1就是没找到

}

}

在App.vue文件中动态决定文字颜色

<tab-bar-item activeColor=”red”></tab-bar-item>//activeColor设置文字颜色

在TabBarItem.vue文件中设置

<div :style=”activeStyle”></div>

//:class清除改成:style，这样可以在App.vue中动态决定图片，文字颜色

props:{

activeColor:{

type:String,

default:”red”

}

}

computed:{

activeStyle(){

return this.isActive?{color:this.activeColor}:{}

}

}

将App.vue文件里面的代码进行抽离，创建MainTabBar.vue文件并将代码放入其中，导入TabBar.vue文件和TabBarItem文件，并在App.vue文件里导入MainTabBar.vue文件，并进行挂载MainTabBar，调用MainTabBar组件。

### tabbar文件路径的引用问题

给路径起一些别名

Vue CLI2中文件build文件下的webpack.base.conf.js文件中找到resolve

resolve:{

extension:[“.js”,”.vue”,”.json”],

alias:{

“@”:resolve(“src”),

“assets”:resolve(“src/assets”),

“components”:resolve(“src/components”),

“views”:resolve(“src/views”),

//脚手架三可以用@/assets

}

}

import 可以直接简写成文件名路径

src 文件名前面必须加~来进行简写

## 11.Promise

### Promise的介绍和基本使用

Promise是异步编程的一种解决方案

那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就是网络请求了

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3+4=7一样是将结果返回。

所以往往我们会传入一个函数，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

回调地狱（回调嵌套）：上一个函数的参数作为下一个调用的函数，下一个调用的函数的参数作为下下个调用的函数。解决：Promise（ES6）可以以一种非常优雅的方式来解决这个问题。

参数->函数(resolve,reject),resolve、reject本身它们又是函数

使用setTimeout

new Promise((resolve,reject)=>{ //data网络封装数据Promise+then

setTimeout((data)=>{//Promise保存了异步操作结果的对象

//成功的时候调用resolve

resolve(“Hello World”) //Hello World->data

//失败的时候调用reject

//reject(“error message”)//就不会来到then

},1000)

}).then((data)=>{

console.log(data)

return Promise((resolve,reject)=>{

setTimeout(()=>{

resolve()

//无限套娃嵌套地狱 抽离出去

},1000)

})

}).then(()=>{

console.log()

return Promise((resolve,reject)=>{

//无限套娃嵌套地狱 抽离出去 setTimeout()当成网络请求

})

}) //在Promise函数执行完后then，在then后面套娃

//.catch((err)=>{ console.log(err)}) error message->err

//什么情况下会使用Promise？

//一般情况下是有异步操作时，使用Promise对这个异步操作进行封装

//new->构造函数(1.保存了一些状态信息 2.执行传入的函数)

//executor执行 在执行传入的回调函数时，会传入两个参数，

//resolve,reject本身又是函数

### Promise三种状态

首先，当我们开发中有异步操作时，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

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，当我们主动回调了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，并且会回调catch()

sync->同步 synchronization

async->异步 asynchronization

asynchronization->wrapped init ->Promise->三种状态

另一种写法：

new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve(“Hello World”)

//reject(“error message”)

},1000)

}).then(data=>{

console.log(data)

},err=>{console.log(err)})

### Promise链式调用

我们在看Promise的流程图时，发现无论是then还是catch都可以返回一个Promise对象。

所以，我们的代码其实是可以进行链式调用的：

这里我们直接通过Promise包装了一下新的数据，将Promise对象返回了

Promise.resolve():将数据包装成Promise对象，并且在内部回调resolve()函数

Promise.reject():将数据包装成Promise对象，并且在内部回调reject()函数

加上setTimeout()才是异步操作’

new Promise((resolve,reject)=>{

setTimeout(()={

resolve(“aaa”)

},1000)

}).then(res=>{

console.log(res,”第一次处理”)

return new Promise(resolve=>{ //简写2return Promise.resolve(res+”111”)

resolve(res+”111”) //简写3 return res+”111” 省略掉Promise.resolve

//简写2rutrn Promise.reject(“error message”)

//throw “error message”

})

}).then(res=>{

console.log(res,”第二次处理”)

return new Promise(resolve=>{

resolve(res+”222”)

})

}).then(res=>{

console.log(res,”第三次处理”)

})

//return new Promise(resolve=>resolve(res+”111”))简写1

### Promise的all方法使用

Promise.all([

new Promise((resolve,reject)=>{

$ajax({

url:”url1”,

success:function(data){

resolve(data)

}

})

}),

new Promise((resolve,reject)=>{

$ajax({

url:”url2”,

success:function(data){

resolve(data)

}

})

})

]).then(results=>{

//results[0] 代表的是url1这个请求

//results[1] 代表的是url2这个请求

})

//iterator可以迭代的 进行模拟

Promise.all([

new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve({name:”why”,age:18})

},1000)

}),

new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve({name:”kobe”,age:18}”)

},2000)

})

]).then(results=>{

console.log(results)

})

## 12.Vuex

### Vuex概念和作用

官方解释：Vuex是一个专为Vue.js应用程序开发的状态管理模式

它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

Vuex也是集成到Vue的官方调试工具devtools.extension,提供了诸如零配置的time-travel调试、状态快照导出等高级调试功能。

状态（变量）管理到底是什么？

状态管理模式、集中存储管理这些名词听起来就非常高大上，让人捉摸不透

其实，你可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面

然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用

响应式：数据发生改变，界面跟着改变

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？

const shareObj={name:”why”} //不是响应式的

Vue.prototype.shareObj=shareObj

所有的组件都继承Vue的原型

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，只是我们先要想想VueJS带给我们最大的便利是什么呢？没错，就是响应式

如果你自己封装实现一个对象能不能保证它里面所有的属性做到响应式呢？当然也可以，只是自己封装可能稍稍麻烦一些

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，有什么状态时需要我们在多个组件间共享呢？

如果你做过大型开放，你一定遇到过多个状态，在多个界面间的共享问题。

比如用户的登录状态、用户名称、头像、地理位置信息等等

比如商品的收藏、购物车中的物品等等

这些状态信息，我们都可以放在统一的地方，对它进行保存和管理，而且它们还是响应式的

### vuex单界面到多界面状态管理切换

View->Actions->State->View

template->@click->data->template

State:不用多说，就是我们的状态。（data）

View:视图层，可以针对State的变化，显示不同的信息

Actions:这里的Actions主要是用户的各种操作：点击、输入等等，会导致状态的改变。

父传子：父组件通过props属性向子组件传递数据

npm install vuex --save

创建store文件index.js

import Vue from “vue”

import Vuex from “vuex”

1.安装插件

Vue.use(Vuex)

2.创建对象

const store=new Vuex.store({

state:{

conuter:1000

},

mutations:{},

actions:{},

getters:{},

modules:{}

})

3.导出store对象

export default store

在main.js文件中

import store from “./store”

//Vue.prototype.$store=store

在new Vue({

store, //store必须小写

})

全局单例模式（大管家）

我们现在要做的就是将共享的状态抽取出来，交给我们的大管家，统一进行管理。

之后，你们每个视图，按照我规定好的规则，进行访问和修改等操作

这就是Vuex背后的基本思想

Vuex状态管理异步）:

State->(Render)Vue Components->(Dispatch分发发布)Actions有异步操作发送网络请求时(<->Backend后端 API)->(Commit)Mutations(<->Devtools Vue开发的一个浏览器插件)->(Mutate) State

前端：frontend

### vuex-devtools和mutations

双核浏览器中找到更多工具，进入扩展程序，点击商店，搜索devtools，添加至chrome。在控制台中点击Vue，进行调试，点击第二个图标进行切换到Vuex。

在store/index.js文件中 methods

state:{

counter:1000

},

mutations:{

//方法

increment(state){

state.counter++ //只要是修改state都是通过mutations来进行修改的

//mutations中increment是事件类型，后面是回调函数

}

}

在App.vue文件中

methods:{

addition(){

this.$store.commit(“increment”)

}

}

### vuex-mutations的携带参数

Mutation状态更新

Vuex的store状态的更新唯一方式：提交Mutation

Mutation主要包括两部分：

字符串的时间类型（type）

一个回调函数（handler），该回调函数的第一个参数就是state

在App.vue文件中

methods:{

addCount(count){

this.$store.commit(“incrementCount”,count)

//payload 负载 参数被称为是mutation的载荷（Payload）

//多个参数通过对象的形式进行传递，也就是payload是一个对象

}

}

在store/index.js文件中

mutations:{

incrementCount(state,count){

state.counter+=count

}

//mutations中方法第二个参数是赋值参数，getters中方法第二个参数是getters

}

还可以进行传对象，在methods里方法中定义变量接受对象进行commit传参。

### vuex-mutation的提交风格

上面的通过commit进行提交是一种普通的方式

Vue还提供了另外一种风格，它是一个包含type属性的对象

methods:{

addCount(count){

//1.普通的提交封装

this.$store.commit(“incrementCount”,count)//这个count就是个数字

//2.特殊的提交封装

this.$store.commit({

type:”incrementCount”,

count //这个count就是个对象里面的属性

})

}

}

mutations:{

incremntCount(state,payload){//payload就是包含type、count属性的对象

state.counter+=payload.count

}

}

### vuex-数据的相应式原理

Vuex的store中的state是响应式的，当state中的数据发生改变时，Vue组件会自动更新。

这就要求我们必须遵守一些Vuex对应的规则：

提前在store中初始化所需的属性。

当给state中的对象**添加**新属性时，使用下面的方式：

方式一:**使用Vue.set(obj,”newProp”,123)**

方式二:**用新对象给旧对象重新赋值**

state里面这些属性都会被加入到响应式系统中，而响应式系统会监听属性的变化，当属性发生变化时，会通知所有界面中用到该属性的地方，让界面发生刷新

在store/index.js文件中mutations

updateInfo(state){

//state.info[“address”]=”洛杉矶” 不是响应式的

Vue.set(state.info,”address”,”洛杉矶”) //对象里面添加属性

//第二个参数修改如果是数组就用Number，如果是对象就用String

//该方式做不到响应式 state删除了，但是界面没有删除

//delete state.info.age

Vue.delete(state.info,”age”)

}

### vuex-mutations的类型常量

在mutation中，我们定义了很多事件类型（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Mutation中的方法越来越多

方法过多，使用者需要花费大量的精力去记住这些方法，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。

在store文件中创建mutations-types.js文件

export const INCREMENT=”increment”

在App.vue文件中导入

import {

INCREMENT

} from “./store/mutations-types”

//导出必须加大括号，默认导出不需要加大括号

在store/index.js中使用

import {

INCREMENT

} from “./store/mutations-types”

[INCREMENT](state){

state.counter++

}

//方便查看方法名称和使用

### vuex-actions的使用详解

通常情况下，Vuex要求我们Mutation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。

如果在mutation中使用异步方法，结果是界面改变了，state里面没有改变。

我们强调，不要在Mutation中进行异步操作

但是某些情况下，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Mutation进行异步操作的。

updateInfo:{

updateInfo(staet){

store.info.name=”coderwhy”

}

}

actions:{

aUpdateInfo(context,payload){ //context相当于store

setTimeout(()=>{

context.commit(“updateInfo”)

console.log(payload.message)

payload.success()

},1000)

}

}

在App.vue中

methods:{

updateInfo(){

//this.$store.commit(“updateInfo”)

//this.$store.dispatch(“aUpdateInfo”,”我是payload”)

//参数或者函数 同时函数和参数 传对象

//参数和对象写到一起不够优雅

this.$store.dispatch(“aUpdateInfo”,{

message:”我是携带的信息”,

success:()=>{

console.log(“里面已经完成”)

}

})

}

优雅的写法：

在App.vue中

methods:{

updateInfo(){

//this.$store.commit(“updateInfo”)

//this.$store.dispatch(“aUpdateInfo”,”我是payload”)

this.$store.dispatch(“aUpdateInfo”,”我是携带的信息”,)

//Promise被返回到这里，通过aUpdateInfo进行过渡

.then(res=>{

console.log(“里面完成了”)

console.log(res)

})

}

}

}

在store/index.js中

mutations:{

updateInfo:{

updateInfo(staet){

store.info.name=”coderwhy”

}

}}

actions:{

aUpdateInfo(context,payload){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

context.commit(“updateInfo”)

console.log(payload)

resolve(“111”)

},1000)

})

}

}

函数返回的值作为这个函数的结果，调用该函数可以使用该函数的方法

### vuex-modules的使用详解

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时，store对象就有可能变得相当臃肿

为了解决这个问题，Vuex允许我们将store分割成模块(Module)，而每个模块拥有自己的state、mutations、actions、getters等

在store/index.js文件中

const moduleA={

state:{

name:”kobe”

},

mutations:{

updateName(state,payload){

state.name=payload

}

},

action:{

aUpdateName(context){

ruturn new Promise((resolve,reject)=>{

setTimeout(()=>{

context.commit(“updateName”,”wangwu”)

resolve()

},1000)

})

}

},

getters:{

fullname(state){

return state.name+”111”

},

fullname2(state,getters){

return getters.fullname+”222”

},

fullname3(state,getters,rootState){

return getters.fullname2+rootState.counter

}, //引用根模块state里面的counter属性，必须在子模块中

}

}

modules:{

a:moduleA

}

在App.vue文件中

<h2>{{$store.state.a.name}}</h2> //将a保存到了state

<h2>{{$store.getters.fullname}}</h2>

<h2>{{$store.getters.fullname2}}</h2>

<button @click=””updateName>修改名字</button>

<button @click=”asyncUpdateName”>异步修改名字</button>

methods:{

updateName(){

this.$store.commit(“updateName”,”lisi”)

},

asyncUpdateName(){

this.$store.dispatch(“aUpdateName”)

}

}

### vuex-store文件夹的目录组织

对象的解构

const obj={

name:”why”,

age:18,

heigth:1.88

}

const {name,age,heigth}=obj;

重构store/index.js文件代码

分别创建mutations.js,actions.js,getters.js文件进行抽离，再建立一个文件夹，modules,moduleA.js文件。将mutations,action,getters,moduleA对象里面的代码放入各个文件中默认导出，再在index.js文件中导入。

### vuex-state单一状态树的理解

英文名称是Single Source of Truth，单一数据源 data

如果你的状态信息是保存到多个Store对象中的，那么之后的管理和维护等等都会变得特别困难。

所以Vuex也是使用了单一状态树来管理应用层的全部状态

单一状态树能够让我们最直接的方式找到某个状态的片段，而且在之后的维护和调试过程中，也可以非常方便的管理和维护。

### vuex-getters的使用详解

state里面的属性经过很多次变化的时候，使用getters(computed)

在store/index.js文件中

getters:{

powerCounter(state){

state.counter\*state.counter

}

}

在App.vue文件中

<h2>{{$store.getters.powerCounter}}</h2>

computed:{

over20Stu(){

return $store.state.student.filter(s=>{ //s=>s.age>20

return s.age>20

})

}

}

//student是一个数组对象

在每个组件里面都要写computed的over20stu方法

所以我们将over20stu方法写到store/index.js里面的getters属性里

getters:{

over20stu(state){

return state.student.filter(s=>s.age>20)

},

over20stuLength(state,getters){ //返回一个结果

return getters.over20stu.length

},

overAgestu(state){ //overAgestu: state=>最好不用

return function(age){

return state.student.filter(s=>s.age>age) //返回一个函数进行传参

} // return age=>state.student.filter(s=>s.age>age)

}

### ES6中高阶函数的使用

filter :先看一下官方的解释：

filter() 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素。

注意： filter() 不会对空数组进行检测。

注意： filter() 不会改变原始数组。

其实这个已经写的很清楚了，就是，返回通过检查的元素，并且会创建一个新的数组。先看代码

1 let arr = [1,2,3,4,5,6,7,8,9,10];

2 let new\_arr = arr.filter((n) => { console.log(n) ; return n > 5})

3 console.log(new\_arr,arr)

这里可以出来，我们的打印，会把1-10都打印出来，但是new\_arr中只有，大于5的元素。arr中数组也不会改变。也就是说，只有当filter中函数，返回true 的是时候，被遍历的元素，才会放到新的数组当中。这样其实就省了我们自己写循环，自己创建数组，自己push了。

map：也是看下官方的解释：

map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值。

map() 方法按照原始数组元素顺序依次处理元素。

注意： map() 不会对空数组进行检测。

注意： map() 不会改变原始数组。

其实这个就是简化了我们的for循环操作。同样的是不对原数组进行操作，会生成一个新的数组，并把元素返回到新的数组中。请看代码：

1 let new\_arr\_dou = new\_arr.map(n => { //这里的new\_arr,就是上一步的到的新数组

2 console.log(n);

3 return n\*2;

4 })

5 console.log(new\_arr\_dou,arr);

同样的是先遍历原数组中所有的元素，然后把return的元素，放到了一个新的数组中，同样的没有操作原数组

reduce：看下官方的解释：

reduce() 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

reduce() 可以作为一个高阶函数，用于函数的 compose。

注意: reduce() 对于空数组是不会执行回调函数的。

这个就是计算累加，需要注意的是，这个里面需要穿两个参数。然后返回的不是一个数组了，就是一个确定的值。废话不多说，直接看代码

1 let new\_arr\_add = new\_arr\_dou.reduce((total,n) => {

2 console.log(total,n);

3 return total + n;

4 },0)

5 console.log(new\_arr\_add,new\_arr\_dou);

 红色框中就是total，也就是当前的已经累加的值，然后黑色框就是，咱们遍历的元素的值。可以看到，最后返回的是一个值，也不会对原来的数组进行任何操作。

## axios

### 网络请求模块的选择-axios

Vue中发送网络请求有非常多的方式，那么在开发中，如何选择呢？

选择一:传统的Ajax是基于XMLLHttpRequest(XHR)

为什么不用它呢？

非常好解释，配置和调用方式等非常混乱

编码起来看起来就非常蛋疼

所以真实开发中很少直接使用，而是使用jQuery-Ajax

选择二:在前面学习中，我们经常会使用jQuery-Ajax

相对于传统的Ajax非常好用

为什么不选择它呢？

首先，我们先明确一点：在Vue的整个开发中都是不需要使用jQuery了

那么，意味着为了方便我们进行网络请求，特意引用一个jQuery，你觉的合理吗？

jQuery的代码1w+行

Vue的带啊才1w+行

完全没有必要为了用网络请求就引用这个重量级的框架

选择三:官方在Vue1.x的时候，推出了Vue-resource

Vue-resource的体积相对于jQuery小很多

另外Vue-resource是官方推出的

为什么不选择它呢？

在Vue2.0退出后，Vue作者在GitHub的Issues中说明了去掉vue-resource，并且以后也不会再更新

那么意味着以后vue-resource不再支持新的版本时，也不会再继续更新和维护

对以后的项目开发和维护都存在很大的隐藏

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

axios有非常多的优点，并且维护起来也非常方便

在前端开发中，我们一种常见的网络请求方式就是JSONP

使用JSONP最主要的原因往往是为了解决跨域访问的问题

JSONP的原理是什么呢？

JSONP的核心在于通过<script>标签的src来帮助我们请求数据

原因是我们的项目部署在domain1.com服务器上时，是不能直接访问domain2.com服务器上的资料的

这个时候，我们利用<script>标签的src帮助我们去服务器请求到数据，将数据当作一个javascript的函数来执行，并且执行的过程中传入我们需要的json

所以，封装jsonp的核心就在于我们监听window上的jsonp进行回调时的名称

为什么选择axios？作者推荐和功能特点

功能特点

在浏览器中发送XMLHttpRequest请求

在node.js中发送http请求

支持Promise API

拦截请求和响应

转换请求和响应数据

补充：axios:ajax i/o system

### axios框架的基本使用

安装，npm install axios --save

在main.js文件中导入

import axios from “axios”

axios({

url:”http://123.207.32.32:8000/home/multidata”,

//本身返回Promise

methods:”get”

}).then(res=>{

console.log(ress)

})

axios({

url:”http://123.207.32.32:8000/home/data”,

//本身返回Promise

//专门针对get请求的参数拼接

params:{

type:”pop”,

page:1

}

}).then(res=>{

console.log(ress)

})

//Vue.use()时加入插件用的 而axios是框架

浏览器中测试跨域问题：

<http://123.207.32.32:8000/home/multidata?callback=jsonp123>

进行网络请求

http://123.207.32.32:8000/home/data?type=sell&page=1该接口不能使用

### axios发送并请求

有时候，我们可能需求同时发送两个请求

使用axios.all可以放入多个请求的数组

axios.all([])返回的结果是一个数组，使用axios.spread可将数组[res1,res2]展开为rest1,rest2

axios发送并发请求

axios.all([

axios({

url:”http://123.207.32.32:8000/home/multidata”,

}),

axios({

url:”http://123.207.32.32:8000/home/data”,

params:{

type:”sell”,

page:1

}

})

]).then(results=>{ //results是一个数组 results[0].results[1]

console.log(results) //打印一个数组

})

axios.all([

axios({

url:”http://123.207.32.32:8000/home/multidata”,

}),

axios({

url:”http://123.207.32.32:8000/home/data”,

params:{

type:”sell”,

page:1

}

})

]).then(axios.spread((res1,res2)=>{

console.log(res1)

console.log(res2) //打印的两个对象

}))

数组的解构

const names=[“why”,”kobe”,”james”]

const [name1,name2,name3]=names;

### axios的配置信息相关

在上面的示例中，我们的BaseURL是固定的

事实上，在开发中可能很多参数都是固定的

这个时候我们额可以进行一些抽取，也可以利用axios的全局配置

axios.defaults.baseURL=”123.207.32.32:8000”

axios.defaults.headers.post[“Content-Type”]=”application/x-www-form-urlencoded”

单个axios配置每个axios都需要进行一次配置

所以我们进行全局配置

axios.defaults.baseURL=”http://123.207.32.32:8000”

axios.defaults.timeout=5000

axios.all([

axios({

url:”/home/multidata”,

}),

axios({

url:”/home/data”,

params:{

type:”sell”,

page:1

}

})

]).then(axios.spread((res1,res2)=>{

console.log(res1)

console.log(res2) //打印的两个对象

}))

请求地址

url:”/user”,

请求类型

method:”get”,

请求路径

baseURL:”http://www.mt.com/api”,

请求前数据处理

transformRequest:[function(data){}],

请求后数据处理

transformResponse:[function(data){}],

自定义请求头

headers:{“x-Requested-With”:”XMLHttpRequest”},

URL查询对象

params:{id:12}, 与get请求是对应的

request body

data:{key:”aa”},与post请求是对应的

超时设置s

timeout:1000,

查询对象序列化函数

paramsSerializer:function(){}

### axios的实例和模块封装

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

但是在后续的开发中，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，并且传入属于该实例的配置信息

上面代码是使用全局的axios和对应的配置在进行网络请求

解决方案：

在main.js文件中创建对应的axios的实例

const instance1=axios.create({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

instance1({

url:”/home/multidate”

}).then(res=>{

console.log(res)

})

instance2({

url:”/home/data”,

params:{

type:”sell”,

page:1

}

}).then(res=>{

console.log(res)

})

const instance2=axios.create({

baseURL:”http://222.111.33.33:8000”,

timeout:5000

})

每一个实例都有自己的配置，之前都是使用全局的配置，没有自己独有的配置

进行封装（抽离），每个组件都需要进行网络请求，耦合度太高（对axios依赖太高）和（$router,$store一样进行封装处理，公共文件，引用一个文件，维护和修改时候，只需要修改一个文件即可，$全局变量）

const Vue.prototype.$axios=axios

解决方案，创建一个network文件夹，创建request.js文件

方式一：

import axios from “axios”

export function request(config,success,failure){

//1.创建axios实例

const instance=axios({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

//发送真正的网络请求 用config接受axios res接受axios后面的网络请求

instance(config).then(res=>{

//console.log(res)

success(res)

}).catch(err=>{

//console.log(err)

failure(err)

})

}

方式二：

export function request(config){

//1.创建axios实例

const instance=axios({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

//发送真正的网络请求 用config接受axios res接受axios后面的网络请求

instance(config.baseConfig).then(res=>{

//console.log(res)

config.success(res)

}).catch(err=>{

//console.log(err)

config.failure(err)

})

}

方式三：

export function request(config){

return new Promise((resolve,reject)={

//1.创建axios实例

const instance=axios.create({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

//发送真正的网络请求

instance(config).then(res=>{

resolve(res)

}).catch(err=>{ //前面都是回调函数方式，resolve reject(Promise方式)

reject(err)

})

})

}

方式三改进：

export function request(config){

//1.创建axios实例

const instance=axios.create({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

//发送真正的网络请求

return instance(config)// create 本身就返回Promise对象

}

在main.js文件进行调用

import {request} from “./network/request”

方式一：

request({

url:’/home/multidata’,

timeout:5000

},res=>{

console.log(res) //结果传给success(res)调用自身函数进行回调

},err=>{

console.log(err)

})

服务器路径(随时要维护和修改)和文件路径（自定义）分离

方式二：

request({

baseConfig:{},

success:function(res){}, //通过对象赋值给config取出success函数进行回调

failure:function(err){}

})

方式三：

request({

url:”/home/multidata”,

}).then(res=>{

console.log(res)

}).catch(err=>{

console.log(err)

})

实参传参给形参=>回调函数

### axios的拦截器的使用

axios提供了拦截器，用于我们在发送每次请求或者得到响应后，进行对应的处理

如何使用拦截器呢？

//配置请求和响应拦截

instance.interceptors.request.use(config=>{

console.log(“来到了request拦截success中”);

return config //拦截到请求数据还要进行返回

//请求拦截的作用

//1.比如config中的一些信息不符合服务器的请求

//2.比如每次发送网络请求时，都希望在界面中显示一个请求的图标（动画）

//3.某些网络请求（比如登录（token）），必须携带一些特殊的信息

},err=>{

console.log(“来到了request拦截failure中”)

return err

})

instance.interceptors.response.use(res=>{

console.log(“来到了response拦截success中”);

return res.data //返回data就没有其他东西了

},err=>{

console.log(“来到了response拦截failure中”)

return err

})

## 项目实战(supermall 首页)

### 项目创建和GitHub托管VueCLI3

终端中执行vue create supermall

运行项目npm run serve

登录https://github.com

点击+符号，Repository name supermall、

Description a vue.js.supermall Public 不需要README

Add a License 许可证 选择 MIT

将远程仓库和项目联系起来

复制仓库地址

终端中git clone 仓库地址 //项目名称不一样进行复制项目

在电脑文件里面进行拷贝 原项目.git、node\_modules文件夹不需要拷贝，其他文件拷贝到复制项目里面

终端中cd到复制项目里面 git status 查看状态 //没有进行管理报红

终端中继续输入git add . 再执行git commit -m “初始化项目”

这次提交是提交到本地 需要提交到服务器

再执行 git push 登录GitHub账号密码

方式二：不用拷贝 将项目push到远程仓库里面

终端中使用

git remote add origin 仓库地址

git push -u origin master

登录GitHub账号密码

### 划分项目结构（src）

该项目下终端中npm install

asserts图片、css资源

components公共组件 common（完全公共的组件下一个项目还能使用） content（业务相关公共组件下一个项目不能使用）

views组件视图（首页、分类、购物车、我的）

router路由相关的router

store全局变量相关的Vuex

network网络相关的axios

common公共的js文件 const.js公共的常量进行抽取 utils.js公共方法抽取 mixin.js混入

### css文件的引入

在asserts文件下css文件下创建normalize.css文件（初始化很多属性重置，在GitHub上搜索normalize下载）

创建base.css文件设置body、消除浮动属性，导入normalize.css文件

在App.vue中导入base.css文件

:root->获取根元素的html 里面定义变量--large-size:50px;

使用font-size:var(--large-size)

tintColor背景颜色

### vue.config和editorconfig

脚手架2是在base.config.js文件里面修改

脚手架3创建一个vue.config.js文件

@内部已经配置好了src

*module*.*exports* = {

  configureWebpack: {

    resolve: {

      alias: {

        'components': '@/components',

        'content': 'components/content',

        'common': 'components/common',

        'assets': '@/assets',

        'network': '@/network',

        'views': '@/views',

      }

    }

  }

}

router和store是插件不需要配置别名，network里面axios是框架需要配置别名

在.editorconfig文件里

root = true

[\*]

charset = utf-8

indent\_style = space

indent\_size = 2

end\_of\_line = lf

insert\_final\_newline = true

trim\_trailing\_whitespace = true

### tabbar引入和项目模块划分

将tabbar文件里面（TarBar.vue,TarBarItem.vue）拷贝到components/common文件里面（里面全是插槽可以在下面一个项目使用）在MainTarBar.vue进行导入

将mainTabbar文件里面(MainTarBar.vue)复制到components/conten文件里面(里面是一些图片文字具体代码进行挂载到子组件插槽里面)在main.js文件进行导入

拷贝img下tabbar文件到assets

安装路由npm install vue-router --save

在router/index.js写入router映射关系的代码

导入vue 导入vue-router 安装VueRouter 创建router实例 导出router

在main.js里面导入并挂载router懒加载导入views里面home,category,cart,profile

文件里面Home.vue,Category.vue,Cart.vue,Profile.vue文件

### 小图标的修改以及路径问题

在public文件里面修改图片即可

 <link *rel*="icon" *href*="*<*%= BASE\_URL %>logo.png">

在index.html当前文件所在的路径找找logo.png BASE\_URL动态获取路径

jsp的语法之后会被打包到dist文件夹 进行转换

### 首页导航栏的封装和使用

在components/common文件下创建navbar/NavBar.vue文件

里面分别是左中右三个插槽

scoped是代表css只对当前文件起效果

在home/Home.vue文件导入

插槽里面没有内容 line-height撑不起来 添加height属性

NavBar.vue组件里面的东西都是公共的包括css属性，设置插槽大小布局

背景、字体颜色可以设置到Home.vue里面

TarBar通过路由实现只有一个（所有样式写在各自的子组件里，父组件无样式），而导航栏通过组件实现有多个（公共样式写在子组件里，特殊样式写在父组件里）

### 请求首页的多个数据(轮播图)

先拿到服务器的数据再进行展示

在network文件创建axios.js文件，将网络封装的代码写入其中

一个函数对应一个网络请求数据，可以进行多个网络请求（服务器不同）

安装axios npm install axios --save

再在network文件下创建一个home.js文件进行封装

home.js文件可以封装多个首页中的网络请求(服务器相同路径不同) 对所有请求的层url进行统一管理

在Home.vue中进行导入home.js文件

函数里面所有东西都是局部变量 一旦被调用执行完，里面所有定义的变量东西都会被内存回收掉。函数调用是压入到栈里面 叫函数栈所有东西都是临时的（保存函数调用过程中所有变量->函数调用结束->弹出函数栈(释放函数所有的变量)

在data()里面定义的属性result去接受res函数，一旦函数执行完，res函数里面的所有东西会被回收，rusult不会被回收 指针理解

Promise进行异步操作 函数赋值可能还没完全执行 打印可能为空

MainTabBar.vue和Home.vue是同一级

res就是Promise函数里面网络请求数据的结果data对象（拿到数据 保持数据 使用数据 响应拦截里面返回了一个data 所以只用res.data 而不是res.data.data）

在components/common下创建swiper,创建index.js(导入Swiper.vue、SwiperItem.vue组件并且以对象进行导出)、Swiper.vue(大的轮播框架里面有很多Item->SwiperItem.vue)、SwiperItem.vue文件，再在views/home文件下创建childComps文件，继续创建HomeSwiper.vue组件，在HomeSwiper.vue中从swiper文件导入Swiper,SwiperItem对象。在Home.vue中导入HomeSwiper.vue组件。

swiper/index.js文件就是将两个组件以对象的方式导出，以对象的方式导入

轮播图的展示

在HomeSwiper.vue里面

<swiper-item v-for=”(item,index) in banners” :key=”index”>

<a :href=”item.link”>

<img :src=”item.image” alt=””>//加: 动态绑定变量banners里面某个属性

</a> //banners是个数组 item遍历数组

</swiper-item>

类型是对象或者数组时，默认值必须是个函数

 props: {

      banners: {

        type: *Array*,

        default(){

          return []

        }

      }

    },

### 热卖信息的展示

在views/home/childComps文件里面创建一个FeatureView.vue

在Home.vue中导入

<div *class*="feature-item" *v-for*="(item, index) in features">

      <a :*href*="item.link">

        <img :*src*="item.image" *alt*="">

        <div>{{item.title}}</div>

      </a>

    </div>

{{item.title}}mustache语法

### RecommendView的封装

在views/home/childComps文件里面创建一个RecommendView.vue

在Home.vue中导入

### TabControl的封装

在Components/content文件下创建tabControl,在创建TabControl.vue文件

该首页里面的TabControl.vue可以在另一个界面使用，只是文字不一样（不建议使用插槽 像导航栏NavBar.vue不一样的东西太多了采用插槽）

在Home.vue中导入

<cpn :></cpn>默认传递的是字符串 加:可以编译语法 而不是字符串

data: function () {

      return {

        currentIndex: 0

      }

    },

 methods: {

      itemClick: function (*index*) {

        // 1.改变currentIndex

*this*.currentIndex = *index*;

        // 2.发出事件

      //  *this*.$emit('itemClick', *index*)

      }

    }

position:sticky;

top:44px;//没有达到该位置之前默认为sticky，达到之后浏览器会自动将属性sticky改为fixd 很多浏览器都不适用

### 保存商品的数据结构设计（请求和保存）

变量中存储：流行/新款/精选

流行的数据 页数

在Home.vue中

data(){

return {

goodsList: {

          'pop': {page: 1, list: []},

          'new': {page: 1, list: []},

          'sell': {page: 1, list: []}

        },

}

}

map是键值对的集合

属性名加不加引号都可以正常定义和访问，如果是数字开头的属性名必须加引号

在network/home.js里面

export function getHomeData(*type*, *page*) {

  return axios({

    url: '/home/data',

    params: {

      type,

      page

    }

  })

}

三个type pop new sell 页数page

import {getHomeMultidata, getHomeData, RECOMMEND, BANNER} from "network/home";

对getHomeMultidata,getHomeData两个函数进行封装，封装在methods下的getMultiData(),etHomeProducts(type)函数下面,在created下调用this.getMultiData(),this.getHomeProducts(POP),this.getHomeProducts(NEW),this.getHomeProducts(SELL)

created代码通过函数抽取到methods里面

getHomeProducts(*type*) {

        getHomeData(*type*, *this*.goodsList[*type*].page).then(*res* => {

          //console.log(res)

          const goodsList = *res*.data.list;

*this*.goodsList[*type*].list.push(...goodsList)

*this*.goodsList[*type*].page += 1

*this*.$refs.scroll.finishPullUp()

        })

      }

将服务器里面请求的数据res.data.list push到goodsList里面

...goodsList是对数组的一种解析，可以将数组里面的元素一个个取出来

...可以传很多个数据

### 首页商品数据的展示

在components/content文件下创建good，再创建GoodsList.vue,GoodsListItem.vue组件，（在一个项目出了首页能使用，其他地方也能使用）

GoodsList.vue是GoodsListItem.vue的父组件

 <goods-list-item *v-for*="(item, index) in goodsList" :*key*="index" :*goods*="item"></goods-list-item>

父组件中遍历循环，将item数据传递给子组件，父子组件需要传递数据

不需要传递数据一般采用插槽

GoodsListItem.vue代码

<div *class*="goods" @*click*="goToDetail">

    <img *v-lazy*="getImg" :*key*="getImg" *alt*="">

    <div *class*="goods-info">

      <p>{{goods.title}}</p>

      <span *class*="price">¥{{goods.price}}</span>

      <span *class*="collect">{{goods.cfav}}</span>

    </div>

  </div>

30个图片数据用数组接受（下滑动加载图片），一个图片数据用对象来接受

数组里面有对象

display:flex;

flex-wrap:wrap; 所有内容都显示在一行

justify-content:space-around; 进行均等分

### TabControl点击切换商品

在TabControl.vue中将事件发送出去

*this*.$emit('itemClick', *index*)

定义一个当前类型用来接受动态类型

data(){return{

 currentType: POP,

}}

通过点击事件动态获取当前类型的数据传递给子组件

*this*.goodsList[*this*.currentType].list

通过index和0，1，2比较获取当前点击的类型

tabClick(*index*) {

        switch (*index*) {

          case 0:

*this*.currentType = POP

            break

          case 1:

*this*.currentType = NEW

            break

          case 2:

*this*.currentType = SELL

            break

        }

      },

### Better-Scroll的安装和使用

方式一：

<https://github.com>

搜索iscroll不再维护了 搜索better-scroll核心是iscroll的实现

找到最新的Tag 找到dist进行下载bscroll.js

方式二：

在终端中npm install better-scroll@1.13.2 --save

导入better-scroll

<div class=”wrapper”> 给wrapper设置固定的高度和背景颜色

<button class=”btn”></button>

<ul class=”content”><li></li></ul>

</div>

data(){return{

scroll:null

}}

mounted(){ (created)dom还没有加载出来 应该用mounted

this.scroll=new BScroll(“.wrapper”,{

})

}

cmd中输入ipconfig找到该无线网络上的ipv4 192.168.0.102或者 192.168.1.8

手机浏览器中输入192.168.0.102:8080/home

overflow-y:scroll;实现局部滚动 在移动端是非常的卡顿的 通过方式二解决

方式一基本使用解析

导入bscroll.js

//默认情况下BScroll是不可以实时监听滚动的位置

在<script></script>里面const bscroll=new BScroll(“.wrapper”,{

probeType:3,

click:true,

pullUpLoad:true

//probeType侦测默认为0

// 0和1不侦测实时的位置

//2手指滚动的过程中侦测 手指离开后的惯性滚动过程中不侦测

//3只要是滚动，都侦测

})

bscroll.on(“scroll”,(position)=>{ //监听事件的类型，带参方法获取实时位置

})

bscroll.on(“pullingUp”,()=>{ //监听事件的类型，带参方法

//发送网络请求，请求更多页的数据

//等数据请求完成，并且将新的数据显示出来后

setTimeout(()=>{

bscroll.finishPullUp() //再次进行上拉加载更多

},1000)

})

docunment.querySelector(“.btn”).addEventListener(“click”,function(){

})

给wrapper设置高度和背景颜色

overflow:hidden;

在Home.vue中导入better-scroll

在div中包裹<home-swiper/><good-list/>

耦合度太高 对BScroll进行封装

在components/common文件下创建scroll/Scroll.vue文件

 <div *ref*="wrapper">

    <slot></slot>

  </div>

在Home.vue中导入Scroll文件

在Scroll.vue中导入better-scroll

ref/children可以拿到组件或者元素

ref如果是绑定在组件中的，那么通过this.$refs.refname获取的是一个组件对象

ref如果是绑定在普通的元素中，那么通过this.$refs.refname获取到的是一个元素对象 一定是准确的

<style scoped>//scoped作用域 只能在该组件起作用

Home.vue中

*#home* {

    height: 100vh;

  }

vh->viewport height百分之百视口

方案一：动态计算

.content{

height:calc(100% - 93px);

//calc()中表达式的运算符前后需要加空格

overflow:hidden;

top:44px;

}

方案二：定位

*.content* {

    position: absolute;

    top: 44px;

    bottom: 49px;

    left: 0;

    right: 0;

  }

### BackTop组件的封装和使用

在components/content下创建backTop/BackTop.vue文件

在Home.vue文件中导入BackTop组件

BackTop.vue组件中

 <back-top @*backTop*="backTop" *class*="back-top" *v-show*="showBackTop">

      <img *src*="~assets/img/common/top.png" *alt*="">

    </back-top>

方式一：BackTop组件要监听Scroll组件要通过父组件Home组件

方式二：直接监听组件原生事件native<back-top @click.native=”backTop”>

修饰符.native修饰什么时候使用？

在我们监听一个组件的原生事件时，必须给对应的事件加上.native修饰符，才能进行监听

Home.vue组件中

 backTop() {

*this*.$refs.scroll.scrollTo(0, 0, 300)

//拿到了scroll组件对象（this）调用了srollTo方法

      },

父组件通过$refs访问子组件(都在父组件操作)

<scroll ref=”scroll”></scroll>

Scroll.vue组件中封装了srollTo方法

scrollTo(*x*, *y*, *time*) {

*this*.scroll && *this*.scroll.scrollTo && *this*.scroll.scrollTo(*x*, *y*, *time*)

      }

### BackTop的显示与隐藏

Scroll组件中click属性中

//无论是否设置click:false,button都可以点击

//必须设置click:true,那么div才能监听点击

在Scroll组件里面

1.创建BScroll对象

2.监听滚动的位置

有时候需要实时监听 有时候不需要实时监听（优化性能）

 props: {

      probeType: {

        type: *Number*,

        default: 1

      },

}

动态实现实时监听（封装的思想）

在Home.vue中

<sroll :probe-type=”3”></scroll>不加:传递是字符串 加:传的是数字

在Scroll.vue中

将内部的position(pos)实时监听的滚动的位置传出去

// 2.将监听事件回调

*this*.scroll.on('scroll', *pos* => {

*this*.$emit('scroll', *pos*)

        })

在Home.vue中之定义事件

<scroll @*scroll*="contentScroll"></scroll>

contentScroll(*position*) {

        // 1.决定tabFixed是否显示

*this*.isTabFixed = *position*.y < -*this*.tabOffsetTop

        // 2.决定backTop是否显示

*this*.showBackTop = *position*.y < -BACKTOP\_DISTANCE

      },

返回到顶部是Home组件用BackTop组件原生事件访问($refs)Scroll组件调用方法

BackTop的显示和隐藏是通过Scroll组件($emit)将position属性传出，posion.y和顶部距离的判断来决定BackTop的显示和隐藏。

Home组件和Scroll组件进行属性和方法的传递和调用

<back-top v-show=”showBackTop”></back-top> //默认showBackTop为false

### 完成上拉加载更多

在Scroll组件里

props:{

pullUpLoad:{

type:Boolean,

default:false

}

}

*this*.scroll = **new** BScroll(*this*.$refs.wrapper, {

          probeType: *this*.probeType,

          click: true,

          pullUpLoad: *this*.pullUpLoad

        })

 // 3.监听上拉到底部

*this*.scroll.on('pullingUp', () => {

          console.log('上拉加载');

*this*.$emit('pullingUp')

        })

Scroll组件决定上拉加载更多和BackTop的显示与否

BackTop组件决定是否返回顶部

<scroll @pullingUp=”loadmore”></scroll>

loadmore(){

this.getHomeProducts(this.currentType)

}

*this*.$refs.scroll.finishPullUp()

   getHomeProducts(*type*) {

        getHomeData(*type*, *this*.goodsList[*type*].page).then(*res* => {

          //console.log(res)

          const goodsList = *res*.data.list;

*this*.goodsList[*type*].list.push(...goodsList)

*this*.goodsList[*type*].page += 1

*this*.$refs.scroll.finishPullUp()

        })

      }

图片都是异步加载的 网络都是异步请求的

item->不包含图片 异步加载过来图片会被撑高

获取数据->列表高度发生变化->betterscroll重新计算高度

loadmore->this.$refs.scroll.scroll.refresh()

### 滚动区域的Bug分析和refresh解决

Better-Scroll在决定有多少区域可以滚动时，是根据scrollerHeight属性决定

scrollerHeight属性是根据放Better-Scroll的content中子组件的高度

但是我们的首页中，刚开始在计算scrollerHeight属性时，是没有将图片计算在内的

所以，计算出来的结果是错误的

后来图片加载进来之后有了新的高度，但是scrollHeight属性并没有进行更新。

所以滚动出现了问题

如何解决这个问题了？

监听每一张图片是否加载完成，只要有一张图片加载完成了，执行一次refresh()

如何监听图片加载完成了？

原生的js监听图片：Img.onload=function(){} Vue中没有必要

Vue中监听：@load=方法

调用scroll的refresh()

父访问子this.$refs调用子组件里面的方法

子访问父this.$parents调用父组件里面的方法

父传子props将父组件属性传递给子组件

子传父$emit将子组件的参数通过事件发送给父组件

子组件和root组件之间的通信

如何将GoodListItem.vue中的事件传入到Home.vue中

因为涉及到非父子组件的通信，所以这里我们选择了事件总线

bus->总线

Vue.probetype.$bus=new Vue()

this.$bus.$emit(“事件名称”,参数)

this.$bus.$on(“事件名称”,参数)

方式一：通过Vuex进行转化

方式二：事件总线

在子组件里面通过this.$bus.emit(“aaa”)将数据发送给事件总线

root组件通过this.$bus.$on(“aaa”)来接受事件总线传递的数据

在GoodListItem组件里发送

<img @load=”imageLoad”>

imageLoad(){

this.$bus.$emit(“itemImageLoad”)

}

在Home组件里监听

created(){

this.$bus.$on(“itemImageLoad”,()=>{

this.$refs.scroll.scroll.refresh()

})

}

在main.js中使用原型

Vue.prototype.$bus=new Vue()

refresh函数找不到的bug处理

在Home.vue组件中created函数里面调用Scroll.vue组件中mounted函数里面的refresh方法时候，scroll对象初始化是在mounted里面，初始化之前还是一个null，请求图片来到太快了，造成null.refresh()，refresh函数找不到

官方解释：$refs只会在组件渲染完之后生效，并且它们不是响应式的。这仅作为一个用于直接操作子组件的的”逃逸仓“-你应该避免在模板或计算属性中访问$refs

解决方案：逻辑与

*this*.scroll && *this*.scroll.refresh && *this*.scroll.refresh()

在created最好不要使用this.$refs document.querySelector()查询数据可能为空

解决方案：写到mounted函数里面

 mounted(){

*this*.$bus.$on("itemImageLoad",()=>{

*this*.$refs.scroll.refresh()

      })

    },

### 刷新频繁的防抖动函数处理

对于refresh非常频繁的问题，进行防抖操作

没有防抖函数refresh需要调用30次

有防抖函数refresh可能只需要一次

防抖debounce/节流throttle

在methods中

debounce(func,delay){

let timer=null

//const refresh=funcrion(...args)就是debounce函数调用的结果

return function(...args){

if(timer){

clearTimeout(timer)

}

timer=setTimeout(()=>{

func.apply(this,args)

//func=this.$refs.scroll.refresh.apply(this,args)

},delay)

}

}

//refresh(“11111”,”222”)//因为...args可以传多个参数 this->vue

定时器是异步操作，所以会被拎出来放到事件队列里(定时器是macrotask)等待执行栈执行完以后再执行 js异步编程：单线程+事件队列 空闲时执行

在mounted中

const refresh=this.debounce(this.$refs.scroll.refresh,500)

//refresh不会被销毁 因为this.$bus.$on函数里面调用了refresh产生了闭包

在this.$bus.$on中调用refresh()

防抖函数起作用的过程：

如果我们直接执行refresh，那么refresh函数会被执行30次

可以将refresh函数传入到debounce函数中，生成一个新的函数

之后在调用非常频繁的时候，就使用新生成的函数，而新生成的函数，并不会非常频繁的调用，如果下一次执行来的非常快，那么会将上一次取消掉

第一次timer为null，不会执行if(timer),将setTimeout函数赋值给了timer

第二次timer有了第一次的timer，执行if(timer),将setTimeout函数赋值给了timer

第三十次就会执行func.apply(this,args),refresh函数

将debounce函数抽取到common/utils.js里面作为一个功能函数

### tabControl的offsetTop吸顶效果

必须知道滚动到多少时，开始有吸顶效果，这个时候就需要获取tabControl的offsetTop

但是，如果直接在mounted中获取tabControl的offsetTop，那么值是不正确的

如何获取正确的值了？

监听HomeSwiper中的img的加载完成

加载完成后，发出事件，在Home.vue中，获取正确的值

补充：

为了不让HomeSwiper多次发出事件

可以使用isLoad的变量进行状态的记录

注意：这里不进行多次调用和debounce的区别

监听滚动，动态改变tabControl的样式

问题：东台湾改变tabControl的样式时，会出现两个问题：

问题一：下面的商品内容会突然上移

问题二：tabControl虽然设置了fixed，但是也是随着Better-Scroll一起滚出去了

其他方案来解决停留问题

在最上面，多复制了一份PlaceHolderTabControl组件对象，利用它来实现停留效果，当用户滚动到一定位置时，PlaceHolderTabControl显示出来

当用户滚动没有达到一定位置时PlaceHolderTabControl隐藏起来

<tab-control ref=”tabControl”></tab-control>

在Home组件

data中 tabOffsetTop:0

一开始tabOffsetTop值很小，因为图片没有加载完

在mounted里面 this.tabOffsetTop=this.$refs.tabControl.$el.offsetTop

//获取tabControl的offsetTop

//所有组件都有一个属性$el:用于获取组件中的元素

等图片加载完之后，给出最终的tabOffsetTop

必须知道滚动到多少时，开始有吸顶效果

Bug:mounted后面不需要加,Unexpected token (92:4)

将TabControl组件复制一份放到Scroll组件的上面

在HomeSwiper组件里面 对轮播图片进行监听

@load->imageLoad->swiperImageLoad->swiperImageLoad->

this.$refs.tabControl.$el.offsetTop,得到加载完轮播图片TabControl和顶部的距离

在使用浏览器原生滚动时，为了让导航不跟随一起

.navbar不需要设置position:fixed

直接使用的是better-Scroll的局部滚动

给Scroll组件之外的TabControl组件设置样式

*.fixed* {

    position: fixed;

    top: 44px;

    left: 0;

    right: 0;

  }

让该组件脱离标准流

<tab-control *v-show*="isTabFixed" *class*="fixed" @*itemClick*="tabClick"

                 :*titles*="['流行', '新款', '精选']"></tab-control>

isTabFixed默认为false,等该组件到达Scroll里面的TabControl时候让它显示出来

TabControl和NavBar都是标准流,而TabControl下面的图片也是脱离标准流的，所以Scroll外的TabControl会显示出来（偷天换日）

对两个属性设置ref=”TabControl1” ref=”TabControl2”

在Home组件点击事件tabClick(index){

this.$refs.TabControl1.currentIndex=index;

this.$refs.TabControl2.currentIndex=index;

}

### Home离开时记录状态和位置

让Home保持原来的状态

让Home不要随意销毁掉 keep-alive

从Category点击到Home，让Home中的内容保持原来的位置

离开时，保存一个位置信息saveY

进来时，将位置设置为原来保存的位置saveY信息即可

注意：最好回来时，进行一次refresh()

用导航守卫

在Home组件里

activated: function () {

*this*.$refs.hSwiper.startTimer()

*this*.$refs.scroll.scrollTo(0,*this*.saveY,0)

*this*.$refs.scroll.refresh()

    },

    deactivated: function () {

*this*.$refs.hSwiper.stopTimer()

*this*.saveY=*this*.$refs.scroll.getScrollY()

    },

this.$refs.hSwiper是HomeSwiper组件对象

在scroll组件里

getScrollY(){

        return *this*.scroll?*this*.scroll.y:0

      }

刷新一次不会出现saveY值变化

## 15.项目实战(supermall 详情页)

### 跳转到详情页并且携带id

监听GoodsListItem组件的点击div->goToDetail()

配置详情页路由在router/index.js

创建views文件下detail/Detail.vue文件

在GoodsListItem组件里跳转路由

goToDetail: function () {

        // 1.获取iid

        let iid = *this*.goods.iid;

        // 2.跳转到详情页面

*this*.$router.push({path: '/detail', query: {iid}})

      }

在Detail.vue中

created中

const iid = *this*.$route.query.iid

路由里面的iid 封装成函数 用data()里的iid进行保存

### 详情页-导航栏的封装

在detail创建childComps/DetailNavBar.vue 导入NavBar

### 详情页-数据请求以及轮播图展示

在network文件下创建detail.js文件 导出

在Detail.vue中进行导入

创建childComps/DetailSwiper.vue文件

在Detail.vue中data()里面topImages:[]进行保存服务器的topImages

父传递Props传递给子组件DetailSwiper.vue进行展示

子组件导入{Swiper,SwiperItem}

<keep-alive *exclude*="Detail">

不会点击图片显示同样的详情页

### 详情页-商品基本信息的展示

抽离组件需要的数据

ES6中使用类

class Person(){

this.name=name;

this.age=age;

}

const p=new Person(“why”,18)

console.log(p.name)

在Detail.js文件中 对服务器的数据进行整合再进行展示

export class Goods {

  constructor(*itemInfo*, *columns*, *services*) {

*this*.title = *itemInfo*.title;//标题

*this*.desc = *itemInfo*.desc;//新款

*this*.newPrice = *itemInfo*.price;//现在价钱

*this*.oldPrice = *itemInfo*.oldPrice;//原来价钱

*this*.discount = *itemInfo*.discountDesc;//新品促销

*this*.columns = *columns*;//销量 收藏

*this*.services = *services*;//退货 全国 七天 72小时

*this*.nowPrice = *itemInfo*.highNowPrice;//现在价钱

  }

}

创建类 构造函数

在Detail.vue文件中

// 2.3.获取商品信息

*this*.goods = **new** Goods(data.itemInfo, data.columns, data.shopInfo.services);



创建childComps/DetailBaseInfo.vue文件 Detail.vue的子组件

在该组件中

<div *v-if*="*Object*.keys(goods).length !== 0" *class*="base-info">

先判断goods对象里面是否有数据

const obj={}

Object.keys(obj).length===0 对象里面没有数据

API服务器服务（数据不会存放到一个对象里面）

前端 浏览器web网页 IOS服务器 Android

### 详情页-店铺信息的解析和展示

export class Shop {

  constructor(*shopInfo*) {

*this*.logo = *shopInfo*.shopLogo;//图标

*this*.name = *shopInfo*.name;//名字

*this*.fans = *shopInfo*.cFans;

*this*.sells = *shopInfo*.cSells;//销量

*this*.score = *shopInfo*.score;//描述 价格 质量

*this*.goodsCount = *shopInfo*.cGoods//宝贝

  }

}

创建店铺信息的对象

*this*.shop = **new** Shop(data.shopInfo);



封装一个独立的组件

创建detail/childComps/DetailShopInfo.vue

### 详情页-加入滚动的效果和TabBar隐藏

导入Scroll，给组件的#detail设置了背景颜色将TabBar给遮住了

局部滚动，就是内容在滚动

*#detail* {

    height: 100vh;

    position: relative;

    z-index: 1;

    background-color: #fff;

  }

### 详情页-商品详情数据展示

*this*.detailInfo = data.detailInfo

创建detail/childComps/DetailGoodsInfo.vue

发送事件不用防抖，接受事件用防抖 用if判断

imgLoad(){

        if(++*this*.counter===*this*.imagesLength){

*this*.$emit("imageLoad")

        }

      }

方式一：只会发送一次 所有图片都加载完了，那么进行一次回调就可以了

在mixin.js中加防抖

*this*.newRefresh=debounce(*this*.$refs.scroll.refresh,100)

直接在Detail组件methods中掉用this.newRefresh()即可

方式二：防抖函数 设置时间等最后一张图片加载完了，进行一次刷新就可以了

debouce函数适合非父子组件

事件总线 不需要在方法里面调用

父子之间通信需要在方法里面调用

watch作用监听某一个属性的变化

### 详情页-商品参数信息的展示

export class GoodsParam {

  constructor(*info*, *rule*) {

    // 注: images可能没有值(某些商品有值, 某些没有值)

*this*.image = *info*.images ? *info*.images[0] : '';

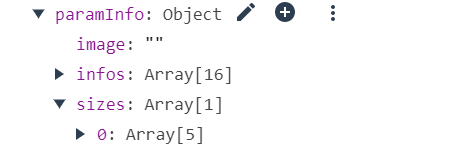
*this*.infos = *info*.set;

*this*.sizes = *rule*.tables;

  }

}

*this*.paramInfo = **new** GoodsParam(data.itemParams.info, data.itemParams.rule);



创建detail/childComps/DetailParamInfo.vue

版本问题 Better-Scroll

'~'（波浪符号)：更新到当前中间位数字中最新的版本，例："vue": "^2.5.2"这个库会去匹配更新到2.5.x中最新版本，如果存在一个新的版本为2.6.0，则不会升到该版本。

'^'（插入符号)：更新到当前第一位数字中最新的版本。例："vue": "^2.5.2",这个库会去匹配更新到2.x.x中最新的版本，如果存在一个新的版本为3.0.0，则不会升到该版本。

### 详情页-商品评论信息的展示

创建detail/childComps/DetailCommentInfo.vue

1970.1.1元年为起点，返回对应的时间戳 1535694719

问题:如何将时间戳转成时间格式化字符串

1.将时间转成Date对象

const date=new Date(1535694719\*1000)

1. 将date进行格式化 转成对应的字符串

date->Format(String)

fmt.fromat(date,”yyyy-MM--dd hh:mm:ss”)

filters: {

      showDate: function (*value*) {

        let date = **new** *Date*(*value*\*1000);

        return formatDate(date, 'yyyy-MM-dd')

      }

    }

对评论时间进行了格式化

创建common/utils.js文件 正则表达式

正则表达式用来干什么的？

字符串匹配 利器 规则太多了

+y->表示一个或者多个y

\*y->0个或者多个y

y?->0个或者一个y

if (/(y+)/.test(*fmt*)) {

*fmt* = *fmt*.replace(*RegExp*.$1, (*date*.getFullYear() + '').substr(4 - *RegExp*.$1.length));

  }

匹配传进来的年份格式 当前年份转成字符串进行截取

### 详情页-商品推荐数据的展示

创建detail/childComps/DetailRcommendInfo.vue

在detail.js文件中

export function getRecommend() {

  return axios({

    url: '/recommend'

  })

}

在Detail.vue中获取数据

\_getRecommend() {

        getRecommend().then((*res*, *error*) => {

          console.log(*res*)

          if (*error*) return

*this*.recommendList = *res*.data.list

        })

      }

 import GoodsListItem from 'views/home/childComps/GoodsListItem'

在GoodsListItem.vue组件中 解决首页中的图片和推荐中的图片名称问题

 computed: {

      getImg() {

        return *this*.goods.img || *this*.goods.image || *this*.goods.show.img

      }

    }

### 详情页-首页和详情页监听全局事件和mixin使用

详情页和首页的商品展示用的是同一个组件，详情页中的商品展示完之后是向Home.vue组件请求刷新的，这不合理。

方式一：

在GoodsIistItem.vue组件中

imgLoad(){

if(this.$route.path.indexOf(“/home”)){

this.$bus.$emit(“homeItemImageLoad”)

}else(this.$route.path.indexOf(“/detail”){

this.$bus.$emit(“detailItemImageLoad”)

})

}

方式二：

在Home.vue组件中

mounted(){

this.itemImgListener=()=>{

refresh()

}

this.$bus.$on(“itemImageLoad”,this.itemImgListener)

}

deactivated(){

this.$bus.$off(“itemImageLoad”,this.itemImgListener)

}

在Detail.vue组件中

导入debounce函数

拿到scroll组件的refresh

mounted(){

let refresh=debounce(this.$refs.scroll.refresh,100)

this.itemImgListener=()=>{

refresh()

}

this.$bus.$on(“itemImageLoad”,this.itemImgListener)

}

//deactivated(){

//this.$bus.$off(“itemImageLoad”,this.itemImgListener)

//}

Detail组件里面没有做缓存，排除了导航守卫 不能使用deactivated

所以要用destoryed

destory(){

this.$bus.$off(“itemImageLoad”,this.itemImgListener)

}

详情页到首页没有定位，首页到详情页有定位

由于Detail、Home组件里面的mounted代码雷同

所以引入混入这个概念

创建common/mixin.js文件

继承是减少类的重复代码

混入是减少对象的重复代码

mounted里面不需要,符号

在Home组件中使用

export const itemListenerMixin={

  data(){

    return{

      itemImgListener:null

    }

  },

  mounted(){

*this*.itemImgListener=()=>{

      refresh()

     }

     const refresh=debounce(*this*.$refs.scroll.refresh,100)

*this*.$bus.$on("itemImageLoad",*this*.itemImgListener)

  }

}

 mixins:[itemListenerMixin],

    data() {

### 详情页-滚到对应内容显示标题

#### 联动效果

父传子

组件里面属性topImages 传入值top-images

子传父事件函数名一样用驼峰命名首字母小写 下一个首字母大写

组件名首字母都要大写

文件名首字母都要小写

获取顶部标题各个组件位置到顶部的距离 在methods里面

created用来异步获取数据 组件加载后

进行赋值的操作进行防抖，在methods中图片加载函数中进行调用，不会过于频繁的调用，提高性能

mounted用来刷新数据 dom加载后还没有渲染 图片没有加载完

在mounted中使用：

$nextTick(()=>{})可以进行渲染//根据最新的数据，对应的DOM是已经被渲染出来，但是图片依然是没有加载完 offsetTop值不对的时候，都是因为图片的问题，所以在mounted中使用

没有$nextTick(()=>{})值不对原因：this.$refs.params.$el压根没有渲染

有$nextTick(()=>{})值不对原因：图片没有计算在内

updated用来数据加载刷新渲染图片加载完后更新

methods调用方法

computed缓存依赖性调用方法

获取数据后还要进行渲染的过程 渲染数据后还要进行图片的加载

在updated中调用methods中的方法

 \_getOffsetTops() {

*this*.themeTops = []

*this*.themeTops.push(*this*.$refs.base.$el.offsetTop)

*this*.themeTops.push(*this*.$refs.param.$el.offsetTop)

*this*.themeTops.push(*this*.$refs.comment.$el.offsetTop)

*this*.themeTops.push(*this*.$refs.recommend.$el.offsetTop)

*this*.themeTops.push(*Number*.MAX\_VALUE)

      },

在detail中监听标题的点击，获取index

滚动到对应的标题：

获取所有主题的offsetTop

问题：在哪里才能真正获取到正确的offsetTop

1. created肯定不行，压根不能获取元素
2. mounted也不行，数据还没有获取到
3. 获取到的数据回调中也不行，DOM还没有渲染完
4. $nextTick也不行，因为图片的高度没有被计算在内
5. 在图片加载完成后，获取的高度才是正确的

滚动内容显示对应标题

在detail组件里

<scroll @*scroll*="contentScroll" :*probe-type*="3"

</scroll>

监听scroll组件中position属性的方法scroll

高级做法：方案二给数组多加了一个值，不遍历最后一个最大的值

i<length-1,就不需要加||后面的判断了

contentScroll(*position*) {

        // 1.监听backTop的显示

*this*.showBackTop = *position*.y < -BACKTOP\_DISTANCE

        // 2.监听滚动到哪一个主题

*this*.\_listenScrollTheme(-*position*.y)

      },

\_listenScrollTheme(*position*) {

        let length = *this*.themeTops.length;

        for (let i = 0; i < length; i++) {

          let iPos = *this*.themeTops[i];

          /\*\*

           \* 判断的方案:

           \*  方案一:

           \*    条件: (i < (length-1) && currentPos >= iPos && currentPos < this.themeTops[i+1]) || (i === (length-1) && currentPos >= iPos),

           \*    优点: 不需要引入其他的内容, 通过逻辑解决

           \*    缺点: 判断条件过长, 并且不容易理解

           \*  方案二:

           \*    条件: 给themeTops最后添加一个很大的值, 用于和最后一个主题的top进行比较.

           \*    优点: 简洁明了, 便于理解

           \*    缺点: 需要引入一个较大的int数字

           \* 疑惑: 在第一个判断中, 为什么不能直接判断(currentPos >= iPos)即可?

           \* 解答: 比如在某一个currentPos大于第0个时, 就会break, 不会判断后面的i了.

           \*/

          if (*position* >= iPos && *position* < *this*.themeTops[i+1]) {

            if (*this*.currentIndex !== i) {

*this*.currentIndex = i;

            }

            break;

          }

        }

      },

将字符串i转化成数字

1. i\*1
2. parseInt(i)

内容滚动，显示正确的标题

普通做法：

let iPos = *this*.themeTops[i];

 (this.currentIndex!==i)&&(i < (length-1) && currentPos >= iPos && currentPos < this.themeTops[i+1]) || (i === (length-1) && currentPos >= iPos),

this.$refs.nav.currentIndex=this.currentIndex

条件成立：this.currentIndex=i

条件一：防止赋值的过程过于频繁

条件二： (i < (length-1) && currentPos >= iPos && currentPos < this.themeTops[i+1]) || (i === (length-1) && currentPos >= iPos),

条件1： (i < (length-1) && currentPos >= iPos && currentPos < this.themeTops[i+1])

判断区间：在0和某个数字之间(i<length-1)

条件2：(i === (length-1) && currentPos >= iPos),

判断大于等于：i===length-1

刷新会取消停留效果，所以客户没有刷新按钮

#### 底部工具栏，点击加入购物车

创建detail/childComps/DetailBottomBar，在Detail组件中导入

#### 回到顶部

导入BackTop组件 导入common/const.js常量

contentScroll(*position*) {

        // 1.监听backTop的显示

*this*.showBackTop = *position*.y < -BACKTOP\_DISTANCE

        // 2.监听滚动到哪一个主题

*this*.\_listenScrollTheme(-*position*.y)

      },

Detail组件和Home组件里面代码重复太多 引入mixin.js

export const backTopMixin = {

  components:{

    BackTop

  },

  data: function () {

    return {

      showBackTop: false

    }

  },

  methods: {

    backTop: function () {

*this*.$refs.scroll.scrollTo(0, 0, 300);

    }

  }

}

传值不能抽取 生命周期里面的函数可以 methods不可以

### 详情页-将商品添加到购物车中

addToCart() {

        // 1.创建对象

        const obj = {}

        // 2.对象信息

        obj.iid = *this*.iid;

        obj.imgURL = *this*.topImages[0]

        obj.title = *this*.goods.title

        obj.desc = *this*.goods.desc;

        obj.newPrice = *this*.goods.nowPrice;

        // 3.添加到Store中

*this*.$store.commit('addCart', obj)

      },

安装Vuex npm install vuex --save

将商品添加到store中

在store中创建index.js文件

导入vue、vuex

1.安装插件

Vue.use(Vuex)

2.创建Store对象

new Vuex.Store({

state:

},)

3.挂载Vue实例上

导出

在main.js中进行导入和挂载

在store/index.js

const state = {

  cartList: []

}

通过数组来存储商品

修改任何state中的属性都要通过mutations

创建store/mutations.js

const mutations = {

   addCounter(*state*,*info*){

*info*.count++

  },

  addToCart(*state*,*info*){

*state*.cartList.push(*info*)

  }

}

oldInfo属性是有值就加1

而info属性是没有值就默认为1

const oldInfo = *state*.cartList.find(*item* => *item*.iid === *info*.iid)

可以用for循环来理解

let oldProduct=null;

for(let item of state.cartList){

if(item.iid===payload.iid){

oldProduct=item;

}

}

mutations唯一的目的就是修改state中状态

mutations中的每个方法尽可能完成的事件比较单一一点

有判断逻辑的代码也应该放到actions中 这样更好跟踪mutations中属性的改变

创建store/actions.js

const actions = {

    addCart(*context*, *info*) {

        console.log(*info*);

        // 1.查看是否添加过

        const oldInfo = *context*.state.cartList.find(*item* => *item*.iid === *info*.iid)

        // 2.+1或者新添加

        if (oldInfo) {

          //oldInfo.count+=1

*context*.commit("addCounter",oldInfo)

        } else {

*info*.count = 1

*info*.checked = true

          //context.state.cartList.push(info)

*context*.commit("addToCart",*info*)

        }

      }

}

创建mutations-type.js文件

export const ADD\_COUNTER="add\_counter"

export const ADD\_TO\_CART="add\_to\_cart"

在actions.js和mutations.js中导入

import { ADD\_COUNTER, ADD\_TO\_CART } from "./mutations-types";

actions.js中

*context*.commit(ADD\_COUNTER,oldInfo)

mutations.js中

[ADD\_COUNTER](*state*,*info*){

*info*.count++

  },

## 16.项目实战(supermall 购物车)

### 购物车-导航栏的实现

创建cart/Cart.vue

导入NavBar组件

导航数量展示

创建getters.js文件

const getters = {

  cartList(*state*) {

    return *state*.cartList

  },

  cartCount(*state*, *getters*) {

    return *getters*.cartList.filter(item=>item.checked).length

  }

}

在Cart组件中

computed: {

      cartList() {

        return *this*.$store.getters.cartList

      },

      cartCount() {

        return *this*.$store.getters.cartCount

      }

    }

代码量还是太多 解决

maGetters辅助函数仅仅是将store中的getters映射到局部计算属性

将getters转成计算属性computed

import {mapGetters} from “vuex”

computed:{

//两种语法

//...mapGetters([“cartList”,”cartCount”])

...mapGetters({

//给方法起名字

count:”cartCount”,

list:”cartList”

})

}

官网vuex.vuejs.org/zh

### 购物车-商品列表展示

创建childComps/CartList.vue

方式一：可以通过父子组件来传递数据

方式二：也可也通过mapGetters来获取store中的数据

创建childComps/CartListItem.vue

创建childComps/CheckButton.vue

### 购物车-底部工具的封装和应用

创建childComps/CartBottomBar.vue

在computed中

 totalPrice() {

        const cartList = *this*.$store.getters.cartList;

        return cartList.filter(*item* => {

          return *item*.checked

        }).reduce((*preValue*, *item*) => {

          return *preValue* + *item*.count \* *item*.newPrice

        }, 0).toFixed(2)

      },

计算价格

0取返为true，10取返为false

判断是否全部选中

checkBtnClick: function () {

        // 1.判断是否有未选中的按钮

        // let isSelectAll = this.$store.getters.cartList.find(item => !item.checked);

        // 2.有未选中的内容, 则全部选中

        if (*this*.isSelectAll) { //全部选中

*this*.$store.state.cartList.forEach(*item* => {

*item*.checked = false;

          });

        } else { //部分没选中

*this*.$store.state.cartList.forEach(*item* => {

*item*.checked = true;

          });

        }

      }

购物车-全选按钮的点击效果

checkBtnClick: function () {

        // 1.判断是否有未选中的按钮

        let isSelectAll = *this*.$store.getters.cartList.find(*item* => !*item*.checked);

        // 2.有未选中的内容, 则全部选中

        if (isSelectAll) {

*this*.$store.state.cartList.forEach(*item* => {

*item*.checked = true;

          });

        } else {

*this*.$store.state.cartList.forEach(*item* => {

*item*.checked = false;

          });

        }

      }

### Vuex-actions返回Promise-mapActions

Toast（吐司）

dispatch返回一个Promise

Toast封装-普通方式的封装

创建components/common/toast/Toast组件

在Detail下导入挂载

在actions.js中

addCart(*context*, *info*) {

      return **new** *Promise*((*resolve*,*reject*)=>{

         // console.log(info);

       // 1.查看是否添加过

       const oldInfo = *context*.state.cartList.find(*item* => *item*.iid === *info*.iid)

       // 2.+1或者新添加

       if (oldInfo) {

         //oldInfo.count+=1

*context*.commit(ADD\_COUNTER,oldInfo)

         resolve("当前商品数量+1")

       } else {

*info*.count = 1

*info*.checked = true

         //context.state.cartList.push(info)

*context*.commit(ADD\_TO\_CART,*info*)

         resolve("添加了新的产品")

       }

      })

     }

在Detail组件中

*this*.addCart(obj).then(*res*=>{

          //console.log(res);

          // this.show=true;

          // this.message=res;

          // setTimeout(()=>{

          //   this.message=""

          //   this.show=false

          // },1500)

        })

### Toast封装-插件方式的封装

创建components/common/toast/index.js

在main.js中导入

Vue.use(toast)

会立刻执行obj.install函数

在index.js中

import Toast from '@/components/common/toast/Toast'

const obj={}

obj.install=function(Vue){

  //console.log("---",Vue);

  //Vue.extend(Toast)//组件构造器

  //document.body.appendChild(Toast.$el)//加载模板内容

  //1.创建组件构造器

  const toastContrustor=Vue.extend(Toast)

  //2.new的方式，根据组件构造器，可以创建出来一个组件对象

  const toast=**new** toastContrustor()

  //3.将组件对象，手动挂载到某个元素上

  toast.$mount(document.createElement("div"))

  //4.toast.$el对应的就是div

  document.body.appendChild(toast.$el)

  Vue.prototype.$toast=toast;//对象加载到原型上

}

export default obj

插件之所以可以直接用就是被定义在原型上面的

混合是针对于共用一个对象

插件是针对于共用一个组件

Toast组件中

data(){

      return{

        message:"",

        isShow:false

      }

    },

    methods:{

      show(*message*="默认文字",*duration*=2000){

       // duration=duration||2000

        //console.log("---");

*this*.isShow=true;

*this*.message=*message*

        setTimeout(()=>{

*this*.isShow=false;

*this*.message=""

        },*duration*)

      }

    },

Detail组件中使用

*this*.$toast.show(*res*,1500)

1、混合

混合 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混合对象可以包含任意组件选项。以组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项

当组件和混合对象含有同名选项时，这些选项将以恰当的方式混合

其调用的数据和方法都是重复的 组件还是要挂载到dom上

2、插件

Vue.js 的插件应当有一个公开方法 install 。这个方法的第一个参数是 Vue 构造器 , 第二个参数是一个可选的选项对象

将组件作为一个公共对象来使用，方便调用它的方法和属性 组件挂载到原型上

### fastclick-解决移动端300ms延迟

安装npm install fastclick --save

百度搜索fastclick用法

main.js导入

FastClick.attach(document.body)

### 图片懒加载-vue-lazyload框架

什么是图片懒加载？

图片需要现在在屏幕上时，再加载这张图片

使用vue-lazyload

安装 npm install vue-lazyload --save

在main.js中导入

使用懒加载插件

Vue.use(VueLazyLoad, {

  preLoad: 1,

  loading: require('assets/img/common/placeholder.png')

})

修改img:src->v-lazy

### px2vw-css单位转化插件

npm install postcss-px-to-viewport --save-dev

retina 1点2个像素

1. >15px
2. >30px

设计稿：iPhone6(750x667 1334)

创建postcss.config.js文件

*module*.*exports* = {

  plugins: {

    autoprefixer: {},

    "postcss-px-to-viewport": {

      viewportWidth: 375,  // 视窗的宽度，对应的是我们设计稿的宽度.

      viewportHeight: 667, // 视窗的高度，对应的是我们设计稿的高度.(也可以不配置)

      unitPrecision: 5, // 指定`px`转换为视窗单位值的小数位数（很多时候无法整除）

      viewportUnit: 'vw', // 指定需要转换成的视窗单位，建议使用vw

      selectorBlackList: ['tab-bar', 'tab-bar-item','shopping-cart-bottom-bar'], // 指定不需要转换的类

      minPixelValue: 1, // 小于或等于`1px`不转换为视窗单位.

      mediaQuery: false, // 允许在媒体查询中转换`px`

exclude:[/TabBar/]

    },

  }

}

1. 在js中使用正则：/正则相关规则/
2. exclude中存放的元素必须是正则表达式
3. 按照排除的文件写对应的正则
4. 正则的规则：^abc：表示匹配的内容，必须以什么内容开头（以abc开头）
5. abc$：表示匹配的内容，必须以什么内容结尾（以abc结尾）

### nginx-项目在window下的部署

服务器问题：一台电脑（没有显示器，主机），24小时开着，为用户提供服务

大多数公司都没有自己的服务主机，租借阿里云/华为云/腾讯云（配置）

主机->操作系统->window(.net)/Linux->tomcat/nginx(软件/反向代理)

第一：将自己的电脑作为服务器->window->nginx（服务器）

第二：远程部署(Mac)

官方网站:https://nginx.org/

Mainline version:Mainline是Nginx目前主力在做的版本，可以说是开发版

Stable version:最稳定版本，生产环境上建议使用的版本

Legacy version:遗留的老版本的稳定版

npm run build

### 响应式原理-依赖技术的分析和学习

面试题：

如何理解Vue生命周期

如何进行非父子组件通信

Vue响应式原理

不要认为数据发生改变，界面跟着更新，并不是理所当然

app.message修改数据，Vue内部是如何监听message数据的改变

Object.defineProperty->监听对象属性的改变 观察者

name->Dep对象->Subs->[watcher1,watcher2]

const obj={

message:”哈哈哈”,

name:”why”

}

Object.keys(obj),forEach(key=>{

let value=obj[key]

Object.defineProperty(obj,key,{

set(newValue){

//告诉了谁？谁用告诉谁？谁在用了？{{message}}张三李四王五

//根据解析html代码，获取到那些人有用属性 发布

value=newValue

},

get(){

//张三，李四，王五 get 订阅 ->update

return value

}

})

})

//发布者

class Dep{

constructor(){

this.subs=[]

} //2.遍历watcher->通知变化(Dep) <-Observer<-data数据newVue()

addSub(watcher){

this.subs.push(watcher)

}

notify(){ //观察者data数据 订阅者el数据 发布者更新数据

this.subs.forEach(item=>{

item.update()

})

}

}

//订阅者

class Watcher(){

constructor(name){

this.name=name;

} //1.new Vue()el数据->Compile订阅数据变化，绑定更新函数->Watcher->Dep

update(){

}

}

const dep=new Dep()

const w1=new Watcher(“张三”)

dep.addSub(w1)

当数据发生改变，Vue是如何直到要通知哪些人，界面发生刷新

发布订阅者模式

## 17.Js语法

### 数组的方法有哪些(18个)

push 末尾添加

pop 末尾删除

shift 首部删除

unshift 首部添加

concat 数组合并

join 数组元素 通过连接符 连接

reverse 数组反转

sort 数组排序

map 原数组的每一项执行函数后，返回一个新的数组。原数组不变。（注意该方法和forEach的区别）。

forEach 针对每一个元素执行提供的函数，对数据的操作会改变原数组。（注意该方法和map的区别，若直接打印Array.forEach，结果为undefined）。

filter 过滤数组中，符合条件的元素并返回一个新的数组。

indexOf 检测当前值在数组中第一次出现的位置索引，未找到返回-1

includes 判断一个数组是否包含一个指定的值

slice 从start开始，end之前结束，不到end；如果不给end值，从start开始到数组结束。start可以给负值，-1表示数组最后位置，-2表示倒数第二个，以此类推，顾前不顾后。

splice 删除元素并添加元素，从index位置开始删除howmany个元素，并将arr1、arr2...数据从index位置依次插入。howmany为0时，则不删除元素。原数组改变。

对数组进行增删改

增加：ary.splice(n,0,m)从索引n开始删除0项，把m或者更多的内容插入到索引n的前面

返回空数组

修改：ary.splice(n,x,m)从索引n开始删除x个，m替换删除的部分

把原有内容删除掉，然后用新内容替换掉

删除：ary.splice(n,m) 从索引n开始删除m个内容

（如果第二个参数省略，则从n删除到末尾）

返回删除的新数组，原有数组改变

reduce() 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

some 对数组中的每一项进行判断，若都不符合则返回false，否则返回true。

every 对数组中的每一项进行判断，若都符合则返回true，否则返回false。

### 函数的多种写法

1.常规写法 具名函数

　　// 函数写法

function funName () {

alert('常规写法');

}

// 调用

funName();

2.将方法作为一个对象 对象函数

　　// 作为对象方法

var obj = {

funName:function(){

alert('这个必须放在一个对象内部，放在外边会出错！');

}

}

// 调用方法

obj.funName();

3.匿名函数

// 匿名函数

var myfun = function(){

alert('这是一种声明函数的方式，左边是一个变量，右边是一个函数的表达式，意思就是把一个匿名的函数表达式复制给了变量myfun，只是声明了一个变量指向了一个函数对象。');

}

// 调用

myfun();

4.构造函数中给对象添加方法，通常在构造函数中用到。

javascript中的每个对象都有prototype属性，Javascript中对象的prototype属性的解释是：返回对象类型原型的引用。

　　// 给对象添加方法

var funName = function(){}

funName.prototype.myfun = function(){

alert('这是在funName函数上的原始对象上加了一个myfun方法，构造函数中用到');

}

// 调用

var funname = new funName();// 创建对象

funname.myfun();

在给对象添加方法时可以用一下方式添加多个方法：

　　// 给对象添加多个方法

var funName = function(){}

funName.prototype = {

fun1:function(){

alert('fun1');

}

,fun2:function(){

alert('fun2');

}

}

// 调用

var funname = new funName();// 创建对象

funname.fun1();

funname.fun2();

# 知识回顾

## 一、邂逅vue.js

### 1.1.认识Vue.js

为什么要学习vue.js

Vue的读音

Vue的渐进式

Vue的特点

### 1.2.安装Vue

CDN引入

下载引入

npm安装

### 1.3.Vue的初体验

Hello Vuejs

Mustache->体验vue响应式

Vue列表展示

v-for

后面给数组追加元素的时候，新的元素也可以在界面中渲染出来

Vue计数器小案例

事件监听：click->methods

### 1.4.Vue中的MVVM

### 1.5创建Vue时，options可以放哪些东西

el:

data:

methods:

生命周期函数：

## 插值语法

mustache语法

v-once

V-html

v-text

v-pre{{}}

v-cloak

## 三、v-bind

### 3.1.v-bind绑定基本属性

v-bind:sr

:href

### 3.2.v-bind动态绑定class

对象语法：作业:class=”{类名:boolean}”

数组语法

### 3.3v-bind动态绑定style

对象语法

数组语法

## 四、计算属性

案例一：firstname+lastname

案例二：books->price

4.1.计算属性的本质

fullname:{set(),get()}

4.2.计算属性和methods对比

计算属性在多次使用时，只会调用一次

它是由缓存的

## 事件监听

### 5.1.事件监听基本使用

### 5.2.参数问题

btnClick

btnClick(event)

btnClick(abc,event)->$event

### 5.3.修饰符

stop

prevent

.enter

.native

## 条件判断

### 6.1.v-if/v-else-if/v-else

### 6.2.登录小案例

### 6.3.v-show

v-show和v-if区别

## 七、循环遍历

### 7.1.遍历数组

### 7.2.遍历对象

value

value,key

value,key,index

### 7.3.数组哪些方法是响应式的

### 7.4.作业完成

## 八、书籍案例

## 九、v-model的使用

### 9.1.v-model的基本使用

v-model=>v-bind:value v-on:input

### 9.2.v-mode和radio/checkbox/select

### 9.3.修饰符

lazy

number

trim

## 组件化开发

### 10.1.认识组件化

### 10.2.组件的基本使用

### 10.3.全局组件和局部组件

### 10.4.父组件和子组件

访问

children/refs

parent/root

### 10.5.注册的语法糖

### 10.6.模板的分离写法

script

template

### 10.7.数据的存放

子组件不能直接访问父组件

子组件中有自己的data，而且必须是一个函数

为什么必须是一个函数

### 10.8.父子组件的通信

父传子:props

子传父:$emit

### 10.9.项目

npm install

npm run serve

### 10.10.slot的使用

基本使用

具名插槽

编译的作用域

作用域插槽

## 前端模块化

### 11.1.为什么要使用模块化

简单写js代码带来的问题

闭包引起代码不可复用

自己实现了简单的模块化

AMD/CMD/CommonJS

### 11.2.ES6中模块化的使用

export

import

## Webpack

### 12.1.什么是webpack

webpack和gulp对比

webpack依赖环境

安装webpack

### 12.2.webpack的起步

webpack命令

webpack配置:webpack.config.js/package.json(scripts)

### 12.3.webpack和loader

css-loader/style-loader

less-loader/less

url-loader/file-loader

Babel-loader

### 12.4.webpack中配置vue

vue-loader

### 12.5.webpack和plugin

### 12.6.搭建本地服务器

webpack-dev-server

### 12.7.配置文件的分离

## Vue CLI

### 13.1.什么是CLI

脚手架是什么东西

CLI依赖webpack，node，npm

安装CLI3->拉去CLI2模块

### 13.2.CLI2初始化项目的过程

### 13.3.CLI2生产的目录结构的解析

export(导出)/import(导入)

.vue

dist->distribution(发布)

Webpack ./src/main.js ./dist/bundle.js

### 13.4.runtime-compiler和runtime-only的区别

ESLint到底是什么？

template->ast->render->vdom->真实DOM

render:(h)=>h,->createElement

### 13.5.Vue CLI3

如何通过CLI3创建项目

CLI3的目录结构

配置文件：1.Vue UI 2.隐藏的配置 3.自定义vue.config.js

## Vue-Router

### 14.1.什么是前端路由

后端渲染\后端路由

前后端分离

SPA\前端路由

### 14.2.路由的基本配置

安装vue-router

Vue.use->创建VueRouter对象->挂载到Vue实例上

配置映射关系：1.创建组件 2.配置映射关系 3.使用router-link/router-view

### 14.3.细节处理

默认路由：redirect

mode:history

router-link->tag/replace/active-class

### 14.4.动态路由

/user/:id

this.$route.params.id

### 14.5.参数的传递

params

query->URL

URL:

协议://主机:端口/路径?/查询

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

### 14.6.路由嵌套

children:[]

### 14.7.导航守卫

全局导航守卫

路由独享守卫

组件内守卫

### 14.8.Keep-alive

### 14.9.1TarBarItem的封装过程

## Promise

### 15.1.Promise的基本使用

如何将异步操作放入到promise中

(resolve,reject)=>then/catch

### 15.2.Promise的链式调用

### 15.3.Promise的all方法

## 十六、Vuex

### 16.1.什么是状态管理

### 16.2.Vuex的基本使用

state->直接修改state（错误）

mutations->devtools

### 16.3.核心概念

state->单一状态树

getters->

mutations->

actions->异步操作（Promise）

modules

### 16.4.目录组织方式

## 十七、络请求封装（axios）

### 17.1.网络请求方式为的选择

### 17.2.axios的基本使用

### 17.3.axios的相关配置

### 17.4.axios的创建实例

### 17.5.axios的封装

## 十八.项目开发

### 18.1.划分目录结构

### 18.2.引用两个css文件

### 18.3.vue.config.js和editorconfig

### 18.4.项目的模块划分：tabbar->路由映射关系

### 18.5.首页开发

#### navbar的封装

#### 网络数据的请求

#### 轮播图

#### 热卖信息

#### 推荐信息RecommendView

独立组件封装RecommendView

div>a>img

TabControl

独立组件的封装

props->titles

div>根据titles v-for遍历div->span{{title}}

css相关

选中哪一个tab，哪一个tab的文字颜色变色，下面border-bottom

currentindex 将点击的index赋值给currentindex

再用class=”active{currtentindex===index}”来设置点击样式

#### 首页商品数据的请求

设计数据结构，用于保存数据

goods:{

pop:page1/list[30]

new:page1/list[30]

sell:page1/list[30]

}

发送数据请求

在home.js中封装getHomeGoods

在Home.vue中，又在methods中getHomeGoods(type)

在created调用getHomeGoods(“pop”)getHomeGoods(“new”)getHomeGoods(“sell”)

page:动态的获取对应的page

获取到数据:res

this.goods(type).list.push(...res.data.list)

this.goods[type].page+=1

#### 对商品数据进行展示

封装GoodsList.vue组件

通过点击事件获取当前数据类型去获取当前商品的数据请求 传递给该组件

props:goods->list[30]

v-for:goods->GoodListItem[30]

GoodListItem(组件)->GoodsItem(数据)

props:{

goods:{

type:Array,

default(){

return []

}

}

}

封装GoodsListItem.vue组件

props:goosItem

goodsitem取出数据，并且使用正确的div/span/img基本标签进行展示

Home组件传递一个数组给GoodList组件，GoodList组件v-for取出数组里面的每一条数据传递给GoodListItem组件，GoodListItem组件v-for取出数据对象里面的每一个属性进行展示

#### 对滚动进行重构:Better-Scroll

在index.html中使用Better-Scroll

const bscroll=new BScroll(el,{})

注意：wrapper->content->很多内容

1. 监听滚动

probetype:0/1/2手指滚动/3只要是滚动

bscroll.on(“scroll”,position=>{})

1. 上拉加载

pullUpLoad:true

bscroll.on(“pullingUp”,()=>{})

1. click:false

button可以监听点击

div不可以

在Vue项目中使用Better-Scroll

在Profile.vue中简单的演示

对Better-Scroll进行封装:Scroll.vue

Home.vue和Scroll.vue之间进行通信

Home.vue将probeType设置为3

Scroll.vue需要通过$emit实时将事件发送到Home.vue

#### 回到顶部

对BackTop.vue组件进行封装

如何监听组件的点击

直接监听back-top的点击，但是可以直接监听？

不可以，必须添加修饰.native

回到顶部

scroll对象，scroll.scrollTo(x,y,time)

this.$refs.scroll.scrollTo(0,0,500)

#### BackTop组件的显示和隐藏

isShowBackTop:false

监听滚动，拿到滚动的位置：

-opsition.y>1000->isShowBackTop:true

isShowBackTop=-opsition.y>1000

#### 解决首页中Better-Scroll可滚动区域的问题

### 18.6.详情页开发

#### 点击商品进入详情页

GoodListItem点击

点击之后获取商品的IID，跳转到详情页，并且传入IID

#### **解决-首页保持位置状态**

deactivated：记录离开时的位置

activated：通过scrollTo函数，设置离开时位置

bug:Better.Scroll最新的1.15.0有scrollTo的位置

解决方案：进行版本的回退：1.13.2

详情页的导航栏实现

返回按钮

标题列表的展示：center

#### **请求详情的数据**

接口:/detail?iid=

#### **轮播图的实现**

Swiper/SwiperItem

#### **商品基本信息的展示**

数据来自四面八方

对数据进行汇总：一个对象当中

一个对象传入到子组件中

#### **店铺信息的展示**

#### **商品图片的展示**

#### **参数信息的展示**

#### **评论信息的展示**

时间格式化

服务器返回的时间戳->date->格式化

yyyy-MM-dd hh:mm:ss

#### **推荐数据的展示**

请求推荐数据

GoodList展示数据

#### mixin的使用

创建混入对象：const mixin={}

组件对象中：mixins:[mixin]

### 18.7购物车开发

#### 点击加入购物车

监听加入购物车按钮的点击，并且获取商品信息

监听

获取商品信息：iid/price/image/title/desc

#### 将商品添加到Vuex中

安装Vuex

配置Vuex

定义mutations，将商品添加到state.cartList

重构代码：

将mutations中的代码抽取actions中定义两个mutations

将mutations/actions单独抽取到文件中

#### 购物车的导航栏的展示

#### 购物车商品的展示

CartList->Scroll滚动问题

CartListItem->CheckButton

#### 商品的选中和不选中切换

修改模型对象，改变选中和不选中

#### 底部工具栏的汇总

全选按钮

计算总价格

去计算

#### 购物车的全选按钮

显示状态

判断是否有一个不选中为true，取反为false

点击全选按钮

如果原来都是选中，点击一次，全部不选中

如果原来都是不选中（某些不选中），全部选中

#### 添加购物车弹窗

Vuex的补充

Actions可以返回一个Promise

mapActions的映射关系

Toast（吐司）的封装

普通的封装方式

插件的封装方式

补充细节

fastClick减少点击延迟

移动端300ms的延迟