目录

[VUE介绍 5](#_Toc6748600)

[什么是渐进式 5](#_Toc6748601)

[VUE的特点 5](#_Toc6748602)

[认识MVC开发模式 5](#_Toc6748603)

[MVC开发模式的缺点 6](#_Toc6748604)

[变种的开发模式MVP 6](#_Toc6748605)

[变种的开发模式MVVM 6](#_Toc6748606)

[VUE的两个核心点 7](#_Toc6748607)

[vue起步 7](#_Toc6748608)

[静态数据的装载 7](#_Toc6748609)

[动态数据的装载 7](#_Toc6748610)

[插值表达式{{ 表达式 }}“Mustache”语法（胡子语法） 8](#_Toc6748611)

[vue中的指令 8](#_Toc6748612)

[常用指令 8](#_Toc6748613)

[给元素属性绑定值v-bind 9](#_Toc6748614)

[给元素绑定事件v-on 9](#_Toc6748615)

[v-model实现双向数据响应 10](#_Toc6748616)

[v-for输出数组和对象数据到页面 11](#_Toc6748617)

[指令小结 11](#_Toc6748618)

[事件的处理对象methods 12](#_Toc6748619)

[事件的处理对象methods中的this对象 12](#_Toc6748620)

[数据响应变化 13](#_Toc6748621)

[数组数据的响应 13](#_Toc6748622)

[对象数据的响应 13](#_Toc6748623)

[简易留言板 14](#_Toc6748624)

[选项卡 15](#_Toc6748625)

[components声明实例组件 16](#_Toc6748626)

[开发实例组件的两种形式 16](#_Toc6748627)

[开发简易版头部组件 16](#_Toc6748628)

[开发简易版页面组件 17](#_Toc6748629)

[父组件向子组件传递数据 18](#_Toc6748630)

[子组件向父组件传递数据 19](#_Toc6748631)

[组件与组件传递数据 20](#_Toc6748632)

[全局组件 21](#_Toc6748633)

[总结options对象的属性 21](#_Toc6748634)

[slot内置组件 22](#_Toc6748635)

[无名slot 22](#_Toc6748636)

[具名slot 23](#_Toc6748637)

[过滤器 24](#_Toc6748638)

[全局过滤器 24](#_Toc6748639)

[组件内过滤器 25](#_Toc6748640)

[监视 26](#_Toc6748641)

[watch监视 26](#_Toc6748642)

[watch深度监视 27](#_Toc6748643)

[computed监视多个值的变化 28](#_Toc6748644)

[组件的生命周期 29](#_Toc6748645)

[beforeCreate、created 30](#_Toc6748646)

[beforeMount、mounted 31](#_Toc6748647)

[beforeUpdate、updated 32](#_Toc6748648)

[beforeDestroy、destroyed 33](#_Toc6748649)

[activated、deactivated、keep-alive 34](#_Toc6748650)

[$refs属性 35](#_Toc6748651)

[获取入口组件中的DOM元素 35](#_Toc6748652)

[获取子组件 36](#_Toc6748653)

[关于$属性 的总结 36](#_Toc6748654)

[$nextTick方法 37](#_Toc6748655)

[nextTick案例 37](#_Toc6748656)

[前端路由 38](#_Toc6748657)

[路由原理：原生JS实现 38](#_Toc6748658)

[vue-rotuer插件 39](#_Toc6748659)

[路由的快速启动 39](#_Toc6748660)

[router-link内置组件 40](#_Toc6748661)

[路由规则中的name属性 41](#_Toc6748662)

[获取路由的参数 42](#_Toc6748663)

[嵌套路由 43](#_Toc6748664)

[前端实现权限控制 44](#_Toc6748665)

[meta 44](#_Toc6748666)

[beforeEach钩子函数 44](#_Toc6748667)

[权限控制实例 44](#_Toc6748668)

[单文件组件 46](#_Toc6748669)

[.vue文件 46](#_Toc6748670)

[vue-loader 46](#_Toc6748671)

[webpack 46](#_Toc6748672)

[Vue-cli 47](#_Toc6748673)

[搭建项目骨架 47](#_Toc6748674)

[axios 48](#_Toc6748675)

[基本使用 48](#_Toc6748676)

[axios合并多个请求 49](#_Toc6748677)

[请求的配置 50](#_Toc6748678)

[全局配置defaults：this.$axios.defaults.XXX 50](#_Toc6748679)

[单请求配置options：axios.post(url,data,options); 51](#_Toc6748680)

[拦截器interceptors 52](#_Toc6748681)

[loading动画 52](#_Toc6748682)

# VUE介绍

一套用于构建用户界面的**渐进式框架**

## 什么是渐进式

https://mp.weixin.qq.com/s?\_\_biz=MzUxMzcxMzE5Ng==&mid=2247485737&amp;idx=1&amp;sn=14fe8a5c72aaa98c11bf6fc57ae1b6c0&source=41#wechat\_redirect



1. 声明式渲染
2. 组件系统
3. 客户端路由（单页应用 single page application SPA）
4. 状态管理
5. 构建工具

## 声明式渲染和命令式渲染

声明式：只需要声明在那里，程序会帮我们处理底层逻辑（数组的map方法）

命令式：需要用具体的代码表达在那里，要做什么，怎么实现(for 循环数组)

vue声明式渲染：只要初始化vue实例，数据就可以绑定到DOM模板上。

## VUE框架的特点

构建用户界面，只关注View层

简单易学，简洁，轻量、快速

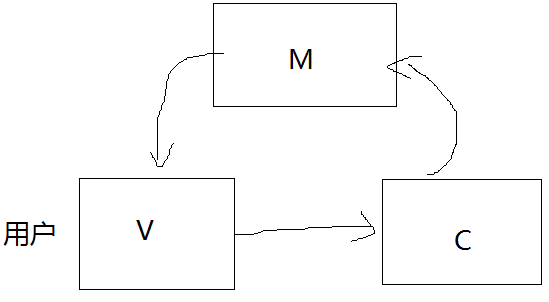
渐进式框架

## 认识MVC开发模式

M----Model----模型----数据（建模：把人的信息抽选出来）

V----View-----视图----表现层（HTML/CSS 用户的入口）

C----Controller---控制器----业务逻辑



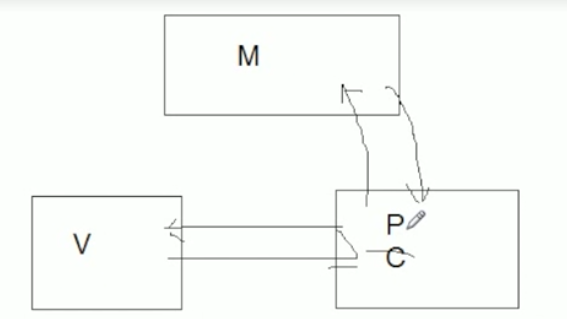
## MVC开发模式的缺点

1、M和V的耦合度高

2、C特别臃肿

## 变种的开发模式MVP

P:presenter（主持人）



优点：如果M或V发生了变化，只要修改P就可以了。M和V没有耦合了

缺点：P会变得特别特别的臃肿

## 变种的开发模式MVVM

VM:viewmodel 视图层+模型层，两个融合

优点： M和V没有耦合了，部分的业务逻辑分给了V，不会特别臃肿

# VUE的两个核心点

1. 响应式的数据绑定

当数据发生变化，视图会自动更新

忘记操作DOM这回事，专注于数据的操作

1. 可组合的视图组件

把视图按照功能划分成若干个组件，组件可以一级一级的组合成应用，形成倒置

的组件树。

使用组件的优点：便于维护、重用性好、便于测试



# vue起步

引包

启动 new Vue(options);

配置options（选项）对象：{el:发生行为的目的地, template:装载数据的模板}

## 静态数据的装载

new Vue({

el:"#app",

template:'<div><h1>hello world</h1></div>'

})

## 动态数据的装载

new Vue({

    el:"#app",

    template:'<div><h1>hello world{{text}}</h1></div>',

data:function(){

return {

text:"大家好"

}

}

})

## 插值表达式{{ 表达式 }}“Mustache”语法（胡子语法）

{{ a?b:c }} {{ {name:’web’,sex:’man’} }}

# vue中的指令

概念：**为了更方便的控制页面中的模板和数据**，**vue提供的一些操作**叫指令。

所有指令都是以**v-xxx**开头的属性，就像是在html中添加**自定义属性**一样。

指令其实是vue封装好操作DOM行为而设置的一些**暗号。**

## 常用指令

v-text 就是innerText

v-html 就是innerHTML

v-if 值为true 元素插入 值false 元素移除

v-else-if

v-else

v-show 值为true 元素显示 值false 元素 display: none;

new Vue({

    el:"#app",

    template:`

<!--必须只能放一个根节点-->

<div>

<span v-text="text"></span>

<span v-html="html"></span>

<button v-if="isExit">测试v-if</button>

<!--以下的3个指令必须是相邻的元素-->

<button v-if=" num==1 ">测试v-if</button>

<button v-else-if=" num==2 ">测试v-if-else</button>

<button v-else>测试v-else</button>

<hr>

<button v-show="isShow">测试v-show</button>

</div>

`,

data:function(){

return {

text:"大家好",

html:"<strong>大家好</strong>",

isExit:true,

num:1,

isShow:false

}

}

})

## 给元素属性绑定值v-bind

两种写法

1. v-bind:属性=”变量”
2. ：属性=”变量”

new Vue({

    el:"#app",

    template:`

<!--必须只能放一个根节点-->

<div>

<input type="text" v-bind:value="myValue" :file=" 'XXX' ">

</div>

`,

data:function(){

return {

myValue:"hello"

}

}

})

## 给元素绑定事件v-on

两种写法

1. v-on:事件属性=”表达式 || 函数名”
2. @事件属性=”表达式 || 函数名”

new Vue({

    el:"#app",

    template:`

<div>

<input type="text" v-bind:value="myValue" :file=" 'XXX' ">

<button v-on:click="myValue='abc'">点击改变myValue的值</button>

<button @click=" myValue='123' ">点击改变myValue的值</button>

</div>

`,

data:function(){

return {

myValue:"hello"

}

}

})

**这里是通过事件修改vue中的数据，再通过v-bind实现单向数据响应**

## v-model实现双向数据响应

    <div id="app">

        <input type="text" **v-model="msg"**>

        <p>{{msg}}</p>

    </div>

    <script>

        var message="hello world";

        var vm=new Vue({

            el:"#app",

            data:{//数据对象上的属性会成为 实例对象上的属性

                msg:message

            }

        })

        console.log(vm);

/\*

            在控制台中输入 vm.msg="abc" 也能修改

         \*/

    </script>

***特别提醒：v-model只能给具有value属性的元素添加***

new Vue({

    el:"#app",

    template:`

<div>

<input type="text" v-model="myValue">

<span v-show=" myValue=='' ">用户名不能为空</span>

</div>

`,

data:function(){

return {

myValue:"请输入用户名"

}

}

})

## v-for输出数组和对象数据到页面

语法形式：

v-for=”item in arr”

v-for=”(item,index) in arr”

v-for=”(value,key,index) in obj”

new Vue({

    el:"#app",

    template:`

<div>

<ul>

<li v-for="item in stus" :class="item.myclass">{{ item.name }}</li>

</ul>

<ul>

<li v-for="(value,key,index) in fruit" :class="index%2==0?'bga':'bgl'">{{ key+':'+value }}</li>

</ul>

</div>

`,

data:function(){

return {

stus:[{name:'张三',myclass:'bga'},{name:'李四',myclass:'bgl'},{name:'王五',myclass:'bga'},],

fruit:{'apple':'10kg','banana':'30kg','orange':'50kg'}

}

}

})

## 指令小结

操纵文本节点：v-text v-html

操纵属性节点：v-bind:属性 或 :属性

操纵事件属性：v-on:事件属性 或 @事件属性

操纵元素：v-if v-else-if v-else（插入移除元素） v-show（显示隐藏元素）

操纵数据：v-mode（双向数据流 元素要有value属性） v-for（多数据的循环展示）

# 事件的处理对象methods

new Vue({

el:"#app",

template:`

<div>

<h1 v-if="isShow">命运的安排</h1>

<button @click="isFnShow">事件处理</button>

</div>

`,

data:function(){

return {

isShow:true

}

},

methods:{

isFnShow(){

alert(1);

}

}

});

## 事件的处理对象methods中的this对象

var vm=new Vue({

    el:"#app",

    data:function(){

        return {//数据对象上的属性会成为 实例对象上的属性

            msg:"hello world"

        }

    },

    methods:{//事件处理对象上的属性也会成为 实例对象上的属性

        isFnShow:function(){

            alert(1);

alert(this.msg);

        }

    }

})

console.log(vm);

console.log(vm.$el);

console.log(vm.msg);//this就是vm

console.log(vm.isFnShow);//this就是vm

# 数据响应变化

vue只会对**计划好的数据**进行响应和渲染，那么**如何对将来要添加的数据**进行动态响应呢？

<div id="app">

<p>{{msg}}</p>

<p>{{abc}}</p>

</div>

<script>

let vm=new Vue({

el:'#app',

data:{

msg:'hello'

}

});

vm.abc='world';//这样没计划好的后添加数据，渲染时会报错

</script>

## 数组数据的响应

<script>

let vm=new Vue({

el:'#app',

data:{

arr:[1,2,3]

}

});

vm.arr.push(4);//这个push是vue改进后的一个方法，所以可以重新渲染

</script>

## 对象数据的响应

let vm=new Vue({

el:'#app',

data:{

obj:{name:'lc'}

}

});

vm.arr.push(4);

vm.$set(vm.obj,'age',30);

# 简易留言板

<div id="app">

<input type="text" v-model='val'>

<input type="button" value="发送" @click="send()">{{val}}

<ul>

<li v-for='ele in list'>{{ele}}</li>

</ul>

</div>

<script>

let list = [];

let vm = new Vue({

el: '#app',

data: () => ({

list,

val: ''

}),

methods: {

send() {

this.list.push(this.val);

}

}

})

</script>

# 选项卡

<div id="app">

<input

type="button"

v-for="item,index in tabData"

:value="item.title"

:class="{active:index===current}"

@click="tab(index)"

>

<div v-for="v,i in tabData" :style="{display:i===current?'block':'none'}">

<p v-for="option,i in v.content">{{option}}</p>

</div>

</div>

<script src="./vue.js"></script>

<script>

let tabData=[

{title:'新闻',content:['教育新闻','科技新闻','娱乐新闻']},

{title:'娱乐',content:['娱乐中国','娱乐世界','娱乐网民']},

{title:'体育',content:['初中体育','高中体育','大学体育']},

]

let vm=new Vue({

el:'#app',

data:()=>({

tabData,

current:0

}),

methods:{

tab(i){

this.current=i;

}

}

});

</script>

# components声明组件

组件是对数据和方法的简单封装, 简而言之，组件就是对象

**组件的类型：入口组件，实例组件，全局组件，内置组件，单文件组件**

Vue选项中template可以设置入口组件（父组件）

1. 要使用其他组件就要先**开发组件**（子组件）
2. 再在父组件（components）中**声明组件**
3. 然后再父组件的template处**使用子组件**。

同理子组件也可以继续开发子组件。

## 开发实例组件的两种形式

var comp1={}//语法糖形式创建组件

var comp2=Vue.extend({})//标准创建组件

## 开发简易版头部组件

var MyHead={//这里是开发子组件代码

template:`

<div>

<h1>我是头部</h1>

</div>`

}

new Vue({

el:"#app",

//下面是声明子组件代码

components:{

"my-head":MyHead

},

//下面是使用子组件代码，入口位置

template:`

<div>

<my-head />

</div>`

})

## 开发简易版页面组件

var MyHead={

template:`

<div>

<h1>我是头部</h1>

</div>`

}

var MyLeft={

template:`

<div>我是身体左边</div>`

}

var MyRight={

template:`

<div>我是身体右边</div>`

}

var MyBody={

components:{

"my-left":MyLeft,

"my-right":MyRight

},

template:`

<div>

<my-left />

<my-right />

</div>`

}

var MyFoot={

template:`

<div>

<h3>我是脚部</h3>

</div>`

}

new Vue({

el:"#app",

components:{

"my-head":MyHead,

"my-body":MyBody,

"my-foot":MyFoot

},

template:`

<div>

<my-head />

<my-body />

<my-foot />

</div>`

})

## 父组件向子组件传递数据

原理：

在父组件中 添加属性并且通过v-bind 绑定动态数据；

在子组件中配置props选项，该选项的值是一个数组，将父元素的属性设置成数组元素，然后通过插值表达式拿到传过来的值。

var MyLeft={

template:`

<div>我是身体左边{{title}}</div>`,

props:['title']

}

var MyRight={

template:`

<div>我是身体右边{{title}}</div>`,

props:['title']

}

var MyBody={

components:{

"my-left":MyLeft,

"my-right":MyRight

},

template:`

<div>

<my-left title="hello" />

<my-right :title="aaa" />

</div>`,

data:function(){

return {

aaa:"父组件中的数据"

}

}

}

## 子组件向父组件传递数据

<script>

let myHead={

data:()=>({

childValue:'子组件中的数据'

}),

template:`

<div>

<input type="button" @click="fn" value="button"/>

</div>

`,

methods:{

fn(){

// pevent是在父组件on监听的事件

// 第二个参数this.childValue是需要传的值

this.$emit('pevent',this.childValue)

}

}

}

let App={

components:{'my-head':myHead},

template:`

<div>

{{zf}}

<my-head @pevent="pmethod" />

</div>`,

data:()=>({

zf:''

}),

methods:{

pmethod(val){

this.zf=val;

}

}

}

var vm=new Vue({

el:'#app',

components:{'app':App},

template:`<app />`

});

</script>

## 组件与组件传递数据

<script>

let Bus=new Vue();//创建一个公共的实例对象

let Item1={

data:()=>({

msg:'item1组件中的数据'

}),

template:`

<div>

<p>这是Item1组件</p>

<button @click="fn1">送数据给给其他组件</button>

<hr>

</div>`,

methods:{

fn1(){

Bus.$emit('gevent',this.msg);

}

}

};

let Item2={

data:()=>({

msg:''

}),

mounted() {

Bus.$on('gevent',(data)=>{

this.msg=data;

})

},

template:`

<div>

<p>这是Item2组件</p>

<button @click="fn2">获取item2组件中的数据</button>

</div>`,

methods:{

fn2(){

alert(this.msg);

}

}

};

var vm=new Vue({

el:'#app',

components:{

item1:Item1,

item2:Item2

},

template:`

<div>

<item1 />

<item2 />

</div>`

});

</script>

# Vue2 模板template的四种写法

<div id="app">

<h1>我是直接写在构造器里的模板1</h1>

</div>

<template id="demo3">

<h1 style="color:red">我是选项模板3</h1>

</template>

<script type="x-template" id="demo4">

<h1 style="color:red">我是script标签模板4</h1>

</script>

<script>

var vm=new Vue({

el:"#app",

data:{

message:1

},

//第2种模板 写在构造器里

//template:`<h1 style="color:red">我是选项模板2</h1>`

//第3种模板 写在<template>标签里

//template:'#demo3'

//第4种模板 写在<script type="x-template">标签里

template:'#demo4'

//第五种 render 渲染

const app = new Vue({

el: '#app',

data: {

count: 0

},

render (createElement) {

return createElement(

'div', {

ref: 'myDiv',

// domProps: {

// innerHTML: '<span>注意：添加该属性会把后面的子节点覆盖</span>'

// },

attrs: {

id: 'test-id',

title: 'test-title'

}

},

[

createElement('item', {

ref: 'item'

},

[

createElement('p', {

ref: 'p'

}, 'this is a slot')

])

])

},

components: {

Item

}

})

# 全局组件

如果一个组件在各个地方都要使用的话，就应该把该组件创建为全局组件

语法：**Vue.component( “组件名” , 组件对象 )**

**使用全局组件，不用声明，直接在组件中使用**

Vue.component("my-btn",{

template:`

<button>全局按钮</button>

`

});

var MyHead={

template:`

<div>

<h1>我是头部</h1>

<my-btn />

</div>`

};

# 总结options对象的属性

new Vue({

el:'#app',

template:'',

data:function(){

return {}

},

methods:{},

components:{},

props:['title']//子组件用来接收父组件传递的数据

})

虚拟DOM

Vue通过建立一个虚拟DOM对真实DOM发生的变化保持追踪

上面的createElement 返回的不是一个实际的DOM元素，更准确的名字是createNodeDescription，它包含的信息

会告知vue页面上需要渲染出上面样的节点，这种节点描述为虚拟节点 Virtual Node ,简称 Vcode,

虚拟DOM是我们对有Vue组件建立起来的整个VNode树的称呼

# slot内置组件

**作用：父组件向子组件传递DOM结构**

**slot插槽的意思（留坑）**

**1.无名slot 2.具名slot**

## 无名slot

需求：实现一个九宫格，放不同的DOM结构

<script>

var MyLi={

template:`

<li>

<slot></slot>

</li>`

}

Vue.component("my-li",MyLi);//注册一个全局组件

var App={

template:`

<div>

<ul>

<my-li><button>button</button></my-li>

<my-li><a href="">slot</a></my-li>

<my-li>3</my-li>

</ul>

</div>`,

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## 具名slot

<script>

var MyLi={

template:`

<li>

<slot name="two"></slot>

<hr>

<slot name="one"></slot>

</li>`

}

Vue.component("my-li",MyLi);//注册一个全局组件

var App={

template:`

<div>

<ul>

<my-li>

<div slot="one"><h1>1111</h1></div>

<div slot="two"><h1>2222</h1></div>

</my-li>

</ul>

</div>`,

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

**总结： slot可以动态传递DOM，props可以动态传递数据**

# component和keep-alive内置组件

component组件可以和is属性配合动态的调用自定义的组件

keep-alive组件可以让我们的组件在切换时保留之前的所有状态

<div id="app">

<input type="button" value="切换到第1个组件" @click="tabComponent(1)"/>

<input type="button" value="切换到第2个组件" @click="tabComponent(2)"/>

<input type="button" value="切换到第3个组件" @click="tabComponent(3)"/>

<keep-alive>

<component :is="current"></component>

</keep-alive>

</div>

<script>

var custom1 = Vue.component("custom1",{

template:`<div @click="changeDivbg($event)">我是第1个组件</div>`,

methods:{

changeDivbg(ev){

ev.target.style.background = "red";

}

}

});

var custom2 = Vue.component("custom2",{

template:`<div>我是第2个组件</div>`

})

var custom3 = Vue.component("custom3",{

template:`<div>我是第3个组件</div>`

})

new Vue({

el:"#app",

data:{

current:custom1

},

methods:{

tabComponent(index){

if(index === 1){

this.current = custom1

}else if(index === 2){

this.current = custom2

}else if(index === 3){

this.current = custom3

}

}

}

})

</script>

# 组件的生命周期

**创建组件过程中产生的事件处理函数(钩子函数)**

beforeCreate：组件创建前

created：组件创建后。这里能拿到数据且可以修改数据，这样实现vue→页面的影响应用：这个位置我们可以发送ajax请求。

beforeMount：vue启动装载数据到页面之前DOM 只执行一次

mounted：vue启动装载数据到页面之后DOM 只执行一次

beforeUpdate：数据改变 影响页面前 获取到原DOM结构 循环执行多次

updated：数据改变 影响页面后 获取到新DOM结构 循环执行多次

beforeDestroy： 对应父组件控制子组件 v-if 为false 销毁组件前

destroyed ： 对应父组件控制子组件 v-if 为false 销毁组件后

销毁是为了做一些其他功能的释放，比如保存数据到localstorage中

**频繁的销毁和创建组件最好使用内置组件keep-alive包裹该组件，这样性能好**

activated：组件被激活了 必须配合keep-alive组件使用

deactivated：组件被停用了 必须配合keep-alive组件使用

## beforeCreate、created

<script>

var MyTest={

template:`

<div>

<h1>我是test组件</h1>

</div>

`,

data:function(){

return {

msg:"hello"

}

},

beforeCreate:function(){

//组件创建前

console.log(this.msg);

},

created:function(){

//组件创建后

console.log(this.msg);

//这里能拿到数据且可以修改数据，这样实现vue→页面的影响

//应用：这个位置我们可以发送ajax请求

}

}

var App={

components:{

"my-test":MyTest

},

template:`<div>

<my-test></my-test>

</div>`,

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## beforeMount、mounted

<script>

var MyTest={

template:`

<div>

<h1>我是test组件</h1>

</div>

`,

data:function(){

return {

msg:"hello"

}

},

beforeMount:function(){

//vue起作用，装载数据到页面之前DOM

//console.log(document.body.innerHTML);

console.log('beforeMount');

},

mounted:function(){

//vue起作用，装载数据到页面之后DOM

//console.log(document.body.innerHTML);

console.log('mounted');

}

}

var App={

components:{

"my-test":MyTest

},

template:`<div>

<my-test></my-test>

</div>`,

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## beforeUpdate、updated

<script>

var MyTest={

template:`

<div>

<h1>我是test组件</h1>

{{msg}}

<button @click="msg=msg+'1'">update</button>

</div>

`,

data:function(){

return {

msg:"hello"

}

},

beforeMount:function(){

//vue起作用，装载数据到页面之前DOM

//console.log(document.body.innerHTML);

console.log('beforeMount');

},

mounted:function(){

//vue起作用，装载数据到页面之后DOM

//console.log(document.body.innerHTML);

console.log('mounted');

},

beforeUpdate:function(){

console.log(document.body.innerHTML);

},

updated:function(){

console.log(document.body.innerHTML);

}

}

var App={

components:{

"my-test":MyTest

},

template:`<div>

<my-test></my-test>

</div>`,

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## beforeDestroy、destroyed

<script>

var MyTest={

template:`

<div>

<h1>我是test组件</h1>

</div>`,

data:function(){

return {

msg:"hello"

}

},

beforeCreate:function(){

//组件创建前

console.log("创建前",this.msg);

},

created:function(){

//组件创建后

console.log("创建后",this.msg);

//这里能拿到数据且可以修改数据，这样实现vue→页面的影响

//应用：这个位置我们可以发送ajax请求

},

beforeDestroy:function(){

//对应父组件 v-if 为false 销毁组件前

console.log("销毁前",this.msg);

},

destroyed:function(){

//销毁组件后

console.log("销毁后",this.msg);

}

}

var App={

components:{

"my-test":MyTest

},

template:`<div>

<my-test v-if="isExist"></my-test>

<button @click="isExist=!isExist">创建销毁子组件</button>

</div>`,

data:function(){

return {

isExist:true

}

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## activated、deactivated、keep-alive

<script>

var MyTest={

template:`

<div>

<h1>我是test组件</h1>

</div>`,

data:function(){

return {

msg:"hello"

}

},

beforeCreate:function(){//组件创建前

console.log("创建前",this.msg);

},

created:function(){//组件创建后

console.log("创建后",this.msg);

//这里能拿到数据且可以修改数据，这样实现vue→页面的影响

//应用：这个位置我们可以发送ajax请求

},

beforeDestroy:function(){//对应父组件 v-if 为false 销毁组件前

console.log("销毁前",this.msg);

},

destroyed:function(){//销毁组件后

console.log("销毁后",this.msg);

},

activated:function(){console.log("组件被激活了");},

deactivated:function(){console.log("组件被停用了");}

}

var App={

components:{

"my-test":MyTest

},

template:`<div>

<keep-alive>//保持组件的生命，实现显示隐藏，不用被反复销毁和创建

<my-test v-if="isExist"></my-test>

</keep-alive>//反复销毁和创建性能就差了

<button @click="isExist=!isExist">创建销毁子组件</button>

</div>`,

data:function(){

return {

isExist:true

}

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

# 过滤器

作用：拿data对象里的数据，根据需求进行加工。

两种过滤器： 全局过滤，组件内部过滤器。

语法：

全局过滤器

Vue.filter(“过滤器名”,function(value,参数1,参数2…){

return 表达式

})

组件内部过滤器

filters:{

过滤器名:function(value,参数1,参数2…){

return 表达式

}…

}

## 全局过滤器

<div id="test">

<p>{{ message | sum }}</p>

<p>{{ message | cal(10,20) }}</p>

</div>

<script src="vue.js"></script>

<script>

Vue.filter("sum", function(value) {

return value + 4;

});

Vue.filter("cal", function(value,begin,xing) {

return value + begin + xing;

});

new Vue({

el:"#test",

data:function(){

return {message:12}

}

});

</script>

## 组件内过滤器

<script>

var App={

template:`

<div>

<input type="text" v-model="message">

<span>{{message | filt("参数1","参数2")}}</span>

</div>`,

data:function(){

return {

message:""

}

},

filters:{

filt:function(val,arg1,arg2){

return arg1+":"+val.split('').reverse().join('')+":"+arg2;

}

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

# 监视

监视可以在数据发生变化时产生一定的行为。（v-model只是实现双向数据流）

## watch监视

<script>

var App={

template:`

<div>

<input type="text" v-model="message">

<span>{{message}}</span>

</div>`,

data:function(){

return {

message:""

}

},

**watch:{**

**message:function(newv,oldv){**

console.log(oldv,newv);

if(newv=="iloveyou"){

alert('我也爱你们');

}

}

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## watch深度监视

<script>

var App={

template:`

<div>

<input type="text" v-model="message">

<span>{{message}}</span>

<hr>

<button @click="stus[0].name='rose'">深度监视</button>

</div>`,

data:function(){

return {

message:"",

stus:[{name:"jack"}]

}

},

watch:{

message:function(newv,oldv){

console.log(oldv,newv);

if(newv=="iloveyou"){

alert('我也爱你们');

}

},

**stus:{**

**deep:true,**

**handler:function(newv,oldv){**

**console.log(oldv,newv);**

**alert("监视成功");**

**}**

**}**

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

## computed监视多个值的变化

<script>

var App={

template:`

<div>

<input type="text" v-model="n1">

+

<input type="text" v-model="n2">

\*

<input type="text" v-model="rate">

={{ result }}

</div>`,

data:function(){

return {

n1:0,

n2:0,

rate:0

}

},

**computed:{//包含原值不变，缓存不调函数的优化机制**

**result:function(){**

**//监视对象写在了函数内部，凡是函数内部有this,相关属性变动，就会触发，当前函数**

**console.log("监视到了");**

**return ((this.n1-0)+(this.n2-0))\*this.rate;**

**}**

**}**

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

})

</script>

# $refs属性

在入口组件对象的template中给元素设置ref属性和属性值(**ref=”标识符”**)

然后再通过 **this.refs.标识符** 在组件的生命周期中**获取入口组件中的DOM元素**

**在使用子组件的时候**也可以设置ref属性和属性值(**ref=”标识符”**)

然后再通过 **this.refs.标识符.$el** 在组件的生命周期中**获取子组件DOM元素**

## 获取入口组件中的DOM元素

<script>

var App={

template:`

<div>

<button ref="btn">button</button>

</div>

`,

beforeCreate:function(){

console.log(this.$refs.btn);//undefinde

},

created:function(){

console.log(this.$refs.btn);//undefinde

},

beforeMount:function(){

console.log(this.$refs.btn);//undefinde

},

mounted:function(){

console.log(this.$refs.btn);//button

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

});

</script>

## 获取子组件

<script>

var TempComponent={

template:`

<div>

<h1>abc</h1>

</div> `

}

var App={

components:{temp:TempComponent},

template:`

<div>

<temp ref="tmp" />

<button ref="btn">button</button>

</div>`,

mounted:function(){

console.log(this.$refs.btn);

console.log(this.$refs.tmp);

console.log(this.$refs.tmp.$el);

console.log(this.$refs.tmp.$parent);

console.log(this.$refs.tmp.$parent.$children[0]);

console.log(this.$refs.tmp.$parent.$el);

console.log(this.$refs.tmp.$parent.$children[0].$el);

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

});

</script>

## 关于$属性 的总结

$refs 获取组件内的元素 或子组件的对象

$parent 获取当前组件的父组件

$children 获取子组件

$root 获取new Vue()出来的根 实例

$el 获取组件对象DOM元素

# $nextTick方法

nextTick:下一瞬间

语法：

this.$nextTick(function(){

//此处是回调函数，异步拿到DOM元素

})

## nextTick案例

<script>

var App={

template:`

<div>

<input type="text" v-if="isShow" ref="text" />

</div>

`,

data:function(){return {isShow:false}},

created:function(){

console.log(this.$refs.text);//undefined 没拿到元素

},

mounted:function(){

this.isShow=true;

//this.refs.input.focus();//这会报错，因为isShow的数据在vue中还没能确定是否不被修改了,且DOM元素也没拿到(undefined)。所以要想执行focus的话，做如下处理，使用vue的$nextTick()方法

this.$nextTick(function(){//进行异步操作拿到元素

console.log(this.$refs.text);//<input type="text">

this.$refs.text.focus();

});

console.log(this.$refs.text);//undefined 没有isShow的设置是能拿到的

}

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`

});

</script>

# 前端路由

**理解：**

这里"路"——通道 "由"：经过（如：必由之路） 路由：通道

传统开发中根据URL的改变而实现路由，但是这样由于重新加载资源如果资源很多很大的情况下，浏览器加载资源会出现等待，容易白屏。

**前端路由实现原理**

通过锚点值的修改，修改innerHTML内容，实现页面数据更新；

这种也叫***单页应用*（SPA : Single Page Application）**

## 路由原理：原生JS实现

<a href="#login">登录页面</a>

<a href="#reg">注册页面</a>

<div id="content"></div>

<script>

var div=document.querySelector("#content");

window.addEventListener('hashchange',function(){

console.log(location.hash);

switch(location.hash){

case "#login":

div.innerHTML="<h1>登录页面</h1>"; break;

case "#reg":

div.innerHTML="<h1>注册页面</h1>"; break;

default:;break;

}

});

</script>

## vue-rotuer插件

vue的核心插件，在官网可以下载，用来实现前端路由

1. 引包（vue.js 和 vue-router.js）
2. 安装路由插件
3. 创建路由对象
4. 配置路由对象
5. 将配置好的路由对象关联到vue实例中
6. 指定路由改变局部的位置（利用router-view内置组件）

### 路由的快速启动

<div id="app"></div>

<!-- 1.引包 -->

<script src="vue.js"></script>

<script src="vue-router.js"></script>

<script>

//2.安装路由插件

Vue.use(VueRouter);

//4.配置路由对象

var Login={template:`<div>我是登录页面</div>`}

//3.创建一个路由对象

var router=new VueRouter({//4.配置路由对象

routes:[{path:"/login",component:Login}]//路由规则

});

//6.指定路由要改变的局部位置

var App={

template:`<div><router-view></router-view></div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

//5.将配置好的路由对象关联到vue实例中

router:router

})

</script>

**如果想默认就打开一个页面就要配置一个默认路由规则**

**{**path:"/",redirect:"/login"}

### router-link内置组件

<script>

//2.安装路由插件

Vue.use(VueRouter);

//4.配置路由对象

var Login={template:`<div>我是登录页面</div>`}

var Register={template:`<div>我是注册页面</div>`}

//3.创建一个路由对象

var router=new VueRouter({//4.配置路由对象

routes:[

{path:"/login",component:Login},

{path:"/register",component:Register}

]

});

//6.指定路由要改变的局部位置

var App={

template:`<div>

<router-link to="/login">登录去</router-link>

<router-link to="/register">注册去</router-link>

<router-view></router-view>

</div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

//5.将配置好的路由对象关联到vue实例中

router:router

})

</script>

这里有个问题：如果路由规则中的path发生变化，下面的router-link如果有多个的话那么to值也要修改多次。解决：给路由规则配置name属性。

### 路由规则中的name属性

<script>

//2.安装路由插件

Vue.use(VueRouter);

//4.配置路由对象

var Login={template:`<div>我是登录页面</div>`}

var Register={template:`<div>我是注册页面</div>`}

//3.创建一个路由对象

var router=new VueRouter({//4.配置路由对象

routes:[

{name:"login",path:"/mylogin",component:Login},

{name:"register",path:"/myregister",component:Register}

]

});

//6.指定路由要改变的局部位置

var App={

template:`<div>

<router-link :to="{ name:'login' }">登录去</router-link>

<router-link :to="{ name:'register' }">注册去</router-link>

<router-view></router-view>

</div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

//5.将配置好的路由对象关联到vue实例中

router:router

})

</script>

## 获取路由的参数

参数的形式有两种

query：url?name=value 查询字符串形式

params: url/value path形式

<script>

//2.安装路由插件

Vue.use(VueRouter);

//4.配置路由对象

var Login={

template:`<div>我是登录页面</div>`,

created:function(){

console.log(this.$route.query)//获取参数

}

}

var Register={

template:`<div>我是注册页面</div>`,

created:function(){

console.log(this.$route.params)//获取参数

}

}

//3.创建一个路由对象

var router=new VueRouter({//4.配置路由对象

routes:[

{name:"login",path:"/mylogin",component:Login},

{name:"register",path:"/myregister/:qq",component:Register}

]

});

//6.指定路由要改变的局部位置

var App={

template:`<div>

<router-link :to="{ name:'login',query:{id:1} }">登录去</router-link>

<router-link :to="{ name:'register',params:{qq:'abc'} }">注册去</router-link>

<router-view></router-view>

</div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

//5.将配置好的路由对象关联到vue实例中

router:router

})

</script>

## 嵌套路由

由于上面的路由配置的是整版页面的切换，能否做到局部的切换呢？

解决：嵌套路由

**1：router-view中包含router-view**

**2：路由规则中嵌套子路由children规则**

<script>

var Woman={template:`<div>女人登录</div>`};

var Man={template:`<div>男人登录</div>`};

var Login={

template:`<div>

我是login显示的内容，下面是子路由显示的内容

<router-view></router-view></div>`

}

var router=new VueRouter({

routes:[

{ name:"login",path:"/mylogin",component:Login,children:[

{ name:"login.woman",path:"woman",component:Woman },

{ name:"login.man",path:"man",component:Man }]

}

]

});

var App={

template:`<div>

<router-link :to="{name:'login.woman'}">女人</router-link>

<router-link :to="{name:'login.man'}">男人</router-link>

<router-view></router-view>

</div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

router:router

})

</script>

## 前端实现权限控制

路由meta元数据：meta是对于路由规则是否需要验证权限的配置

路由钩子：权限控制的函数执行时期（用户在什么时期能访问）

**总结：meta带路由的数据，钩子做权限控制**

### meta

meta元数据在路由对象规则中和name属性同级

**{ meta:{ isChecked:true } }**

带这个要check，不带的放过

### beforeEach钩子函数

每次路由匹配后，渲染组件到router-view之前 beforeEach钩子函数

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

to:到哪里去（是个对象） from:从哪里来（是个对象）

next：放不放走（是个方法）

next():不带参数是放行的意思

next(false):取消用户导航行为

next({ name:”login” }):重定向导航（这个参数也可以写成一个字符串路径）

### 权限控制实例

<script>

Vue.use(VueRouter);

var isLogin=false;

var Login={ template:`<div>我是登录页面</div>`,

created:function(){ isLogin=true; }

}

var Music={template:`<div>我的音乐页面</div>`}

var router=new VueRouter();

//可以多次追加路由规则，动态的获取路由规则

router.addRoutes([

{path:"/",redirect:{name:'login'}},

{name:"login",path:"/mylogin",component:Login},

//meta 给未来路由做权限控制，也叫全局路由守卫，做参照条件，是否放行

{

name:"music",path:"/mymusic",component:Music,

meta:{isChecked:true}

}

]);

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

//console.log(to);

//console.log(from);

//next();//当你写了钩子后，不调next函数的话，容易卡住

if(!to.meta.isChecked){

next();

}else{

//想听歌，要先判断是否登录了。

if(isLogin){//实际开发中，要用localstorage判断或发ajax请求

next()

}else{

alert('请先登录');

//next('/mylogin');

next({ name:"login" });

}

}

});

var App={

template:`<div>

<router-link :to="{ name:'login' }">登录去</router-link>

<router-link :to="{ name:'music' }">去听歌</router-link>

<router-view></router-view>

</div>`

}

new Vue({

el:"#app",

components:{app:App},

template:`<app />`,

router:router

})

</script>

# 单文件组件 webpack nodeJS

## .vue文件

.vue文件，称为单文件组件，是Vue.js自定义的一种文件格式，一个.vue文件就是一个单独的组件，在文件内封装了组件相关的代码：html、css、js

.vue文件由三部分组成：<template>、<style>、<script>

<template>

html

</template>

<style>

css

</style>

<script>

js

</script>

## vue-loader

浏览器本身并不认为.vue文件，所以必须对.vue文件进行加载解析，此时需要vue-loader类似的loader还有许多，如：html-loader、css-loader、style-loader、babel-loader等

需要注意的是vue-loader是基于webpack的

## webpack

webpack是一个前端资源模板化加载器和打包工具，它能够把各种资源都作为模块来使用和处理，实际上，webpack是通过不同的loader将这些资源加载后打包，然后输出打包后文件

简单来说，webpack就是一个模块加载器，所有资源都可以作为模块来加载，最后打包输出

为什么打包（减少对服务器的URL请求，提高性能）

依赖：模块的导入导出关系

入口：

出口：

# Vue-cli

https://www.npmjs.com/package/vue-cli

CLI（command-line interface，命令行界面）是指可在用户提示符下键入可执行指令的界面，它通常不支持鼠标，用户通过键盘输入指令，计算机接收到指令后，予以执行。

vue的脚手架工具，快速的帮我们搭建项目的骨架（常规项目中的依赖包都会给我们配置好）

**安装**

**npm install –g vue-cli**

**使用**

**vue init <template-name><project-name>**

**template-name:模板名称，vue提供了如下六个模板**

* **webpack** - 一个功能齐全的Webpack + vue-loader设置，具有热重载，linting（语法检查），测试和css提取功能。
* **webpack-simple** - 一个简单的Webpack + vue-loader设置，用于快速原型设计。
* **browserify** -全功能Browserify + vueify设置用热重装载，linting（语法检查）＆单元测试。
* **browserify -simple**- 一个简单的Browserify + vueify设置，用于快速原型设计。
* **pwa** - 基于webpack模板的vue-cli的PWA模板
* **simple** - 单个HTML文件中最简单的Vue设置

## 搭建项目骨架

1. npm i vue-cli -g
2. vue init webpack cms\_project\_1
3. 根据需求，按照命令行进行选择（这中间 可能会出现系统上的一些错误，按提示解决）

**ESLint** 是一个语法规则和代码风格的检查工具,可以用来保证写出语法正确、风格

统一的代码

**unit**单元测试，站在开发人员角度测试

**e2e**端对端测试，站在用户角度测试(功能测试黑盒测试)

1. 配置好了，运行 npm run dev 让项目跑一边看下是否正常

# axios

## 基本使用

API: https://segmentfault.com/a/1190000008470355

**语法形式1：**

**axios.method(url,[data],[options])**

**.then(function(res){})**

**.catch(function(err){})**

<script>

//console.log(axios);

var App={

template:`<div>

<button @click="aaa">发送get请求</button>

</div>`,

methods: {

aaa:function(){

this.$axios.get("axios.php")

.then(function(res){

console.log(res);

console.log(res.data);

}).catch(function(err){

console.log(err);

})

}

},

}

//组件内的每一个this对象,都是Vue的孩子

//vue原型上的东西都会共享给所有的孩子

Vue.prototype.$axios=axios;

new Vue({

el:"#app",

template:`<app />`,

components:{app:App}

});

</script>

## axios合并多个请求

**语法形式2：（应用：同时拿到省和市两地的数据…）**

**var q1=axios.get(url,[data],[options]);**

**var q2=axios.post(url,[data],[options]);**

**axios.all([q1,q2])**

**.then( axios.spread( function(res1,res2){} ); )**

**.catch(function(err){})**

<script>

var App={

template:`

<div>

响应1：{{res1}}

<br>

响应2：{{res2}}

<br>

<button @click="join">发送合并请求</button>

</div>`,

data:function(){

return{

res1:'a',

res2:'b'

}

},

methods: {

join:function(){

var q1=this.$axios.get("http://loaclhost/axios/axios.php");

var q2=this.$axios.post("http://loaclhost/axios/axios2.php","a=1");

//合并这两个请求，并处理成功和失败

this.$axios.all([q1,q2])

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

//这个位置要用箭头函数，this才是vue。如果写成拉姆达函数this就是window

console.log(this);

this.res1=res1.data;

this.res2=res2.data;

})).catch(function(err){

console.log(err);

})

}

},

}

Vue.prototype.$axios=axios;

new Vue({

el:"#app",

template:`<app />`,

components:{app:App}

});

</script>

## 请求的配置

### 全局配置defaults：this.$axios.defaults.XXX

<script>

var App={

template:`

<div>

响应1：{{res1}}<br>响应2：{{res2}}<br>

<button @click="join">发送合并请求</button>

</div>`,

data:function(){

return{ res1:'a', res2:'b' }

},

methods: {

join:function(){

//设置基础的全局URL

this.$axios.defaults.BaseURL=<http://loaclhost/axios/>

//this.$axios.defaults.header={};

//设置请求报文头，这样请求头比较多，但可以覆盖默认的请求头

this.$axios.defaults.headers.accept="abc";

var q1=this.$axios.get("axios.php");

var q2=this.$axios.post("axios2.php","a=1");

this.$axios.all([q1,q2])

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

this.res1=res1.data;

this.res2=res2.data;

})).catch(function(err){

console.log(err);

})

}

},

}

Vue.prototype.$axios=axios;

new Vue({

el:"#app",

template:`<app />`,

components:{app:App}

});

</script>

### 单请求配置options：axios.post(url,data,options);

**options：**

baseURL 基础URL路径

headers请求头信息

params 查询字符串(对象)

transformRequest：function(post请求传递的数据) { }转换请求体数据

transformResponse：function(res){ 自己转换相应回来的数据 }转换响应体数据

data请求体数据

timeout请求超时，请求多久以后没有响应算是超时(毫秒)

<script>

var App={

template:`

<div>

响应1：{{res1}} <br> 响应2：{{res2}}<br>

<button @click="join">发送合并请求</button></div>`,

data:function(){ return{ res1:'a', res2:'b' } },

methods: {

join:function(){

var q1=this.$axios.get( "axios.php",{ params:{id:1}, transformResponse:function(data){//对拿回来的数据进行处理

data=data+"处理了"

console.log(data);

**return data**;

}

} );

var q2=this.$axios.post("axios2.php","a=1",{timeout:1000,

transformRequest:function(a){//对要送走的数据进行处理

**return "a=2";**

}

} );

this.$axios.all([q1,q2])

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

this.res1=res1.data; this.res2=res2.data;

})).catch(function(err){

console.log(err);

})

}

},

}

Vue.prototype.$axios=axios;

new Vue({ el:"#app", template:`<app />`, components:{app:App} });

</script>

## 拦截器interceptors

分两种拦截器：

请求之前的拦截器：this.$axios.interceptors.request（全局拦截）

响应之后的拦截器：this.$axios.interceptors.response（全局拦截）

### loading动画

<script>

var App={

template:`<div>

<!--从网上下载一个cssloading动画-->

<div class="cssload-thecube" v-show="isShow">

<div class="cssload-cube cssload-c1"></div>

<div class="cssload-cube cssload-c2"></div>

<div class="cssload-cube cssload-c4"></div>

<div class="cssload-cube cssload-c3"></div>

</div>

<input type="button" value="拦截器" @click="send" />

</div>`,

data:function(){ return { isShow:false } },

methods: {

send:function(){

//在请求之前拦截下来做一些事情，use方法可以多次使用，表示做多个事情

this.$axios.interceptors.request.use((config)=>{

console.log(config);

this.isShow=true;

return config;

});

//在响应之后拦截下来做一些事情

this.$axios.interceptors.response.use((res)=>{

console.log(res)//res.config

this.isShow=false;

return res;

});

this.$axios.get('axios3.php')

.then(function(res){

console.log('响应回来了',res);

})

.catch(function(err){});

}

}

};

Vue.prototype.$axios=axios;

new Vue({

el:"#app",

template:`<app />`,

components:{

app:App

}

});

</script>