<https://cn.vuejs.org/v2/guide/>

www.cnblogs.com/wuhua1/p/6686237.html [js 里面的键盘事件对应的键码](https://www.cnblogs.com/wuhua1/p/6686237.html)

vs插件 bootstrap3 snippets

**vue简介**

**什么是Vue.js**

* vue.js 是目前最火的前端框架，react是最流行的前端框架。（vue语法也可用于手机app的开发，需要借助weex）
* vue.js 是前端的主流框架之一，和angular.js，react.js 一起，并成为前端三大主流框架。
* vue.js是一套构建用户界面的框架，只关注视图层。他不仅容易上手，还便于与第三方库或既有项目整合。（vue 有配套的第三方类库，可以整合起来做大型项目的开发）
* 前端的主要工作？ 负责mvc中的v层，主要工作就是和界面打交道，来制作前端页面效果。

**为什么要学习流行框架**

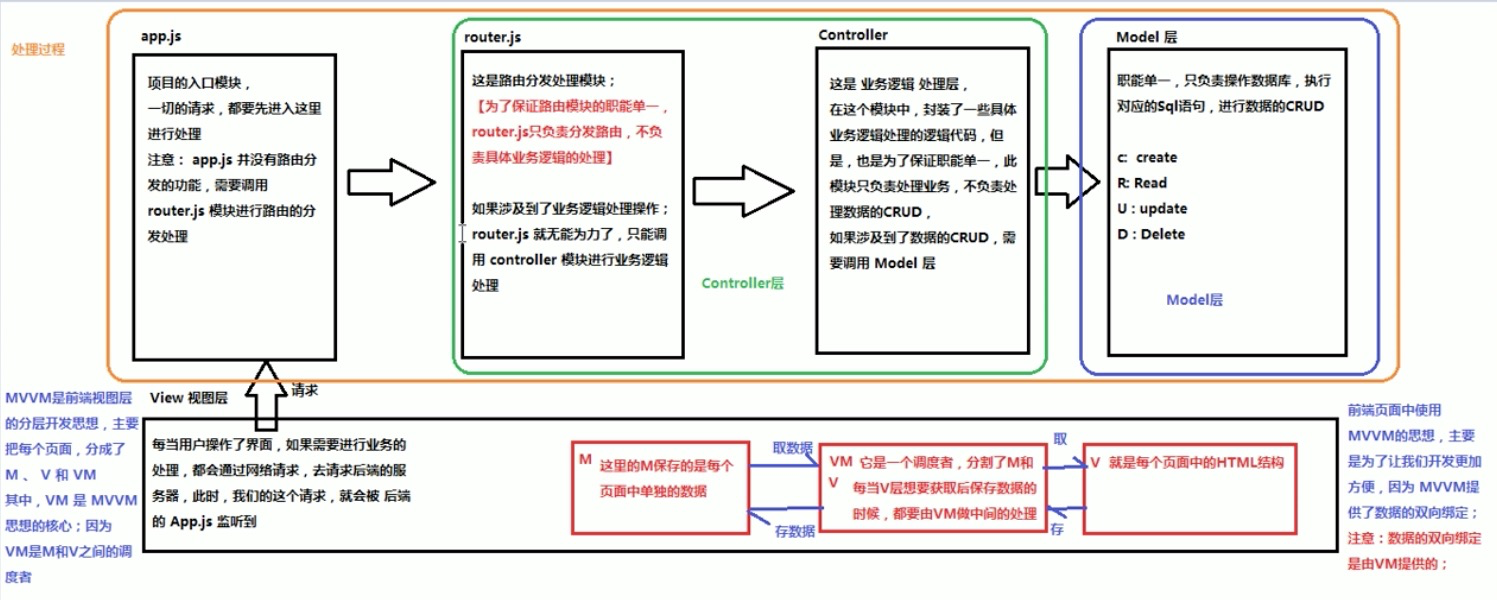
* 企业为了提高开发效率，在企业中，时间就是效率，效率就是金钱。
  + 企业中使用框架可以提高开发效率
* 提高开发效率的发展历史，原生JS-》query之类的类库-》前端模引擎-》angular.js/vue.js
* 在vue中，一个核心的概念，就是让用户不在操作DOM元素，解放了用户的双手，让程序员可以更多的时间去关注业务逻辑
* 增强自己就业时候的竞争力。

**框架和库的区别**

* 框架： 是一套完整的解决方案，对项目的侵入性较大，项目如果需要更换框架，则需要重新架构整个项目。
  + node中的expess；
* 库（插件）： 提供某一个小功能，对项目的侵入性较小，如果某个库无法完成某些需求，可以很容易切换到其他库实现需求。

**node（后端）中的mvc与前端中的mvvm之间的区别**

* mvc 是后端的分层开发概念
* mvvm是前端视图层的概念，主要关注于视图层的分离，也就是说： mvvm把前端的视图层，分为了三部分，model，view，vm，viewmodel
* 为什么有了mvc 还要有mvvm



**vue基本代码和mvvm之间的对应关系**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

<!-- 1. 导入Vue的包 -->

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

</head>

<body>

<!-- 将来new的Vue实例，会控制这个袁术的所有内容 -->

<!-- Vue实例锁控制额这个元素区域，就是我们的V -->

<div id='app'>

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

</div>

<script>

// 2. 创建一个Vue的实例

// 当我们导入包之后，在浏览器内存中，就多了一个Vue构造函数

// 注意，我们new出来的这个vm对象，就是我们mvvm中的vm调度者

var vm = new Vue({

el: '#app', // 表示，当我们new 的这个Vue实例，要控制页面上的哪个区域

// 这里的data就是MVVM中的M，专门用来保存每个页面的数据

data: { //data属性，存放的是el中要用到的数据

msg: '欢迎学习vue' //通过vue提供的指令，很方便的就能把数据渲染到页面上，程序员不再手动操作DOm元素了【前端的Vue之类的框架，不提倡我们去手动操作DOM元素】

}

})

</script>

</body>

</html>

**vue 快速入门**

**安装vue**

参考<https://blog.csdn.net/chenlongjs/article/details/104962275>

**v-cloak v-text v-html**

**v-bind**

**v-on**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

<style>

[v-cloak] {

display: none

}

</style>

</head>

<body>

<div id='app'>

<!-- 使用 v-cloak 能够解决插值表达式闪烁的问题。（当网速慢的时候，页面上会显示出{{ msg }}，再变成123） -->

<p v-cloak> {{ msg }}</p>

<!-- 用v-text也能渲染 跟插值表达式的区别

1. v-text没有闪烁的问题

2. v-text 会覆盖元素原先的内容，但是插值表达式只会替换自己的占位符，不会把整个元素内容清空-->

<h4 v-text="msg"></h4>

<!-- v-html 会输出html，像msg2，插值跟v-text只会输出原本的字符，v-html输出的内容h1 会生效 -->

<div v-html="msg2"></div>

<!-- v-bind: 是vue中，提供的用于绑定属性的指令 title：鼠标悬浮时显示的属性

1. 可以省略v-bind，只保留：

2. v-bindzhong ,可以写合法的js表达式；可以变量+表达式 -->

<!-- <input type="button" value="按钮" v-bind:title="mytitle"> -->

<input type="button" value="按钮" :title="mytitle + '123'" id='btn'>

<input type="button" value="按钮" v-on:click="show" >

<input type="button" value="按钮" @click="show">

</div>

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

<script>

var vm = new Vue({

el: '#app',

data: {

msg: '123',

msg2: '<h1>我是大大的h1</h1>',

mytitle: "这个是我自定义额title",

},

methods: { //这个methods属性中定义了当前Vue实例所有可用的方法

show: function(){

alert('hello')

}

//等效

show(){

alert('hello')

}

}

})

// vue 不推荐操作DOM

// document.getElementById('btn').onclick = function(){

// alert('hello')

// }

</script>

</body>

</html>

**跑马灯效果**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<input type="button" value="浪起来" @click="lang">

<input type="button" value="低调" @click="stop">

<h4>{{ msg }}</h4>

</div>

<script>

// 在vm实例中，如果想要获取data上的数据，或者想要调用methods中的方法，必须通过this.来访问。 这里的this，就表示我们new出来的vm实例对象

var vm = new Vue({

el: "#app",

data: {

msg: "猥琐发育，别浪~~！",

intervalId: null // data上定义定时器id

},

methods: {

lang() {

// console.log(this.msg)

// var \_this = this

if (this.intervalId != null) return;

// => 这个表示函数内部也能使用外面的this指针

this.intervalId = setInterval(() => {

var start = this.msg.substring(0, 1)

var end = this.msg.substring(1)

// 重新拼接得到新的字符串，并复制给this.msg

this.msg = end + start

// 注意，vm实例，会监听自己身上的data中所有数据的改变，只要数据一发生变化，就会自动把新的数据，从data上同步到页面中；好处： 程序员只需要关心数据，不需要考虑如何重新渲染DOm

}, 400)

},

stop(){

clearInterval(this.intervalId)

this.intervalId = null

}

}

})

// 分析：

// 1. 给【浪起来】按钮，绑定一个点击事件 v-on

// 2. 在按钮的时间处理函数中，写相关的业务逻辑代码，拿到msg字符串，然后调用字符串的substring来进行字符串的街区操作，把第一个字符街区出来，放到最后一个位置即可

// 3. 为了实现点击一下按钮，自动街区的功能，需要把2步骤中的代码，放到一个定时器中去

</script>

</body>

</html>

**事件修饰符介绍**

* .stop 阻止冒泡
* .prevent 阻止默认事件
* .capture 添加时间侦听器时使用时间捕捉模式
* .self 治党事件在该元素本身（比如不是子元素）触发时触发回调
* .once事件只触发一次

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<style>

.inner {

height: 150px;

background-color: darkcyan;

}

</style>

</head>

<body>

<div id='app'>

<div class="inner" @click="div1Handler">

<!-- 使用stop阻止冒泡：如果眉头stop，会同时触发button、div 的click事件。有了stop，只触发button的click事件 -->

<input type="button" value="戳他" @click.stop="btnHandler">

</div>

<!-- 加了prevent之后就不会跳转到baidu -->

<a href="http://www.baidu.com" @click.prevent="linkClick">有问题，先百度</a>

<!-- 使用.capture实现捕获出发时间的机制 会先执行div 的click事件-->

<div class="inner" @click.capture="div1Handler">

<input type="button" value="戳他" @click.stop="btnHandler">

</div>

<!-- 使用。self实现只有点击当前元素的时候才会触发时间处理函数 点击按钮只会触发按钮的点击事件

.self 只会组织自己身上的冒泡行为，并不会真正组织冒泡行为-->

<div class="inner" @click.self="div1Handler">

<input type="button" value="戳他" @click="btnHandler">

</div>

<!-- 使用。once 支出法一次时间处理函数， prevent once 额顺序没有影响，效果一样 -->

<a href="http://www.baidu.com" @click.prevent.once="linkClick">有问题，先百度</a>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

},

methods: {

div1Handler() {

console.log('这是触发了inner div 的点击事件')

},

btnHandler() {

console.log('这是触发了btn 点击事件')

},

linkClick() {

console.log('这是触发了lnck 点击事件')

}

}

})

</script>

</body>

</html>

**v-model 实现表单元素数据双向绑定**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<h4>{{msg}}</h4>

<!-- v-bind 只能实现数据的单向绑定，从M自动绑定带V,无法实现数据的双向绑定 -->

<!-- <input type="text" v-bind:value="msg" style="width:100%;"> -->

<!-- 使用v-model指定，可以实现变淡元素跟model中数据的双向数据绑定

v-mode 只能运用在表单元素中 -->

<!-- input(radio,text,address,email) select checkbox textarea -->

<input type="text" v-model="msg" style="width:100%;">

</div>

<script>

var vm = new Vue({

el: '#app',

// 在页面控制台，windows.vm 可以找到该对象。 在控制台上，修改vm.msg="修改了的数据",会在页面上显示修改了的数据。称为单向绑定

data: {

msg: "大家都是好学生"

},

methods: {}

})

</script>

</body>

</html>

简易计算器

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

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

<select name="" id="" v-model='opt'>

<option value="+">+</option>

<option value="-">-</option>

<option value="\*">\*</option>

<option value="/">/</option>

</select>

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

<input type="button" value='=' @click='calc'>

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

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

n1: 0,

n2: 0,

opt: '+',

result: 0

},

methods: {

calc() {

// switch (this.opt) {

// case '+':

// this.result = parseInt(this.n1) + parseInt(this.n2)

// break;

// case '-':

// this.result = parseInt(this.n1) - parseInt(this.n2)

// break;

// case '\*':

// this.result = parseInt(this.n1) \* parseInt(this.n2)

// break;

// case '/':

// this.result = parseInt(this.n1) / parseInt(this.n2)

// break;

// }

// 换一种方法.这是投机取巧的方式，正式开发中尽量少用

var codeStr = 'parseInt(this.n1) ' + this.opt + 'parseInt(this.n2)'

this.result = eval(codeStr)

}

}

})

</script>

</body>

</html>

**在Vue中使用样式**

**使用class样式**

1. 数组
2. 数组中使用三元表达式
3. 数组中嵌套对象
4. 直接使用对象

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<style>

.red {

color: red

}

.thin {

font-weight: 200;

}

.italic {

font-style: italic;

}

.active {

letter-spacing: 0.5em;

}

</style>

</head>

<body>

<div id='app'>

<!-- Vue 中不能用原先的 -->

<!-- <h1 class="red thin">这是一个很大很大的h1</h1> -->

<!-- 直接传递数组，注意，这里class需要使用v-bind做数据绑定 -->

<!-- 在数组中使用三元表达式 -->

<h1 :class="['red','thin',flag?'active':'']">这是一个很大很大的h1</h1>

<!-- 在数组中使用对象来代替三元表达式，提高可读性 -->

<h1 :class="['red','thin',{'active':flag}]">这是一个很大很大的h1</h1>

<!-- 在为class 使用v-bind绑定对象的时候，对象的属性是雷鸣，由于对象的属性可带引号，也可不带引号，所以这里没写引号。属性的值是一个标识符 -->

<h1 :class="classobj">这是一个很大很大的h1</h1>

<!-- 属性里面含有-，则必须带单引号 -->

<!-- 对象就是无需键值对的集合 -->

<h1 :style="{color: 'red', 'font-weight': 200 }">Vue样式-style</h1>

<h1 :style="styleobj1">Vue样式-style</h1>

<h1 :style="[styleobj1,styleobj2]">Vue样式-style</h1>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

flag: true,

classobj: {red: true, this: true, italic: false, active: false},

styleobj1: { color: 'blue', 'font-weight': 200 },

styleobj2: {'font-style':'italic'}

},

methods: {}

})

</script>

</body>

</html>

**使用内联样式**

代码见上

1. 直接在元素上通过:style 的形式，书写样式对象
2. 将样式对象，定义到data中，并直接饮用到：style中
3. 在style中通过数组，饮用多个data上的样式

**v-for 和key属性**

**使用注意事项**

**2.2.0+ 的版本里，当在组件里使用v-for时，key是必须的**

当vue.js 用v-for正在更新已渲染的元素列表时，它默认用“**就地复用**”策略。如果数据项的顺序被改变，Vue将**不是移动DOM元素来匹配数据项的顺序**，而是简单复用此处每个元素，并且确保他在特定索引下显示已被渲染过的每个元素。

为了给Vue一个提示，一遍他梦跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一的key属性

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<!-- <p>{{list[0]}}</p> -->

<!-- 循环普通数组 -->

<p v-for="(item,i) in list">索引值：{{i}}-------值：{{item}}</p>

<!-- 循环对象数组 -->

<p v-for="user in list2">{{user.id}}-----{{user.name}}</p>

<p v-for="(user,i) in list2">索引：{{i}}:{{user.id}}-----{{user.name}}</p>

<!-- 遍历对象 除了🈶key，val之外，在第三个位置还有一个索引，从0 开始-->

<p v-for="(val,key,i) in user">{{val}} ----{{key}}</p>

<!-- 迭代数字 -->

<!-- in 后面我们放过普通数组，对象数组，对象，数字 -->

<!-- 注意 如果使用v-for迭代数字的话，前面你的coun值从1 开始-->

<p v-for="count in 10">这是第{{count}}次循环</p>

<!-- key 使用注意事项 -->

<div>

<label>ID:

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

</label>

<label >

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

</label>

<input type="button" value="添加" @click="add">

</div>

<!-- v-for 循环的时候，key属性只能使用number或者string -->

<!-- key在使用的时候，必须使用v-bind属性绑定的形式，指定key的值 -->

<!-- 在组件中，使用v-for循环的时候，或者在一些特殊情况中，如果v-for有问题，必须在使用v-for的同事，指定唯一的字符串/数字类型的key值 -->

<!-- 案例中如果没有key，在勾选了第五个选项，再添加一个新的选项的时候，默认选中的选项会变 -->

<p v-for="item in list3" v-bind:key='item.id'>

<input type="checkbox" >

{{item.id}} {{item.name}}

</p>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

id: '',

name: '',

list: [1, 2, 3, 4, 5, 6],

list2: [{ id: 1, name: 'zs1' },

{ id: 2, name: 'zs2' },

{ id: 3, name: 'zs3' }],

user: {

id: 1,

name: 'tony',

gender: 'male'

},

list3: [

{id: 1,name:'lisi'},

{id:2, name:'yingzhneg'},

{id:3,name: '赵高'},

{id:4, name:'韩飞'},

{id:5,name:'xunzi'}

]

},

methods: {

add(){

// this.list3.push({id:this.id,name: this.name})

// unshift 插入到list前面。

this.list3.unshift({ id: this.id, name: this.name })

}

}

})

</script>

</body>

</html>

**v-if v-show**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<input type="button" value="" @click="toggle">

<!-- v-if 的特点： 每次都会重新创建或者删除元素 -->

<!-- v-show 的特点： 每次不会重新进行DOm的删除和创建操作，只是切换的元素的display:none 样式 在控制台element上可以看到-->

<!-- v-if 有较高的切换性能消耗，如果元素设计到频繁的切换，最好不要使用v-if，而是使用v-show -->

<!-- v-show 有较高的初始渲染消耗 如果元素可能永远也不会被显示出来，给用户看到，则推荐使用v-if-->

<h3 v-if="flag">这是v-if控制的元素</h3>

<h3 v-show="flag">这是v-show控制的元素</h3>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

flag: true

},

methods: {

toggle() {

this.flag = !this.flag

}

}

})

</script>

</body>

</html>

**品牌案例（搜索）**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<div class="panel pannel-primary">

<div class="panel-heading">

<h3 class="panel-title">添加品牌</h3>

</div>

<div class="panel-body form-inline">

<label for="">

Id:

<input type="text" name="" id="" class="form-control" v-model="id">

</label>

<label for="" v-fontweight='1200'>

Name:

<input type="text" class="form-control" v-model="name" @keyup.enter='add'>

<!-- 113 是F2 的键盘值，这样按F2之后也会提交 -->

<!-- <input type="text" class="form-control" v-model="name" @keyup.113='add'> -->

<!-- 添加自定义键值后可以协作这个 @keyup 之后如果没有其他值，会监听所有键盘 -->

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

</label>

<!-- 在vue、中，使用时间绑定机制，为元素指定处理函数的时候，如果加了小括号，就可以给函数传参了 -->

<input type="button" value="添加" class="btn btn-primary" @click="add()">

<label for="">

搜索名称关键字:

<!-- 在Vue中，所有指令都以v-开头，自定义v-focus 也需要保持 -->

<!-- blue 如果不加单引号，则表示data里面的变量。 加上单引号表示是string -->

<input type="text" class="form-control" v-model="keywords" id='search' v-focus v-color="'blue'">

</label>

</div>

</div>

<table class="table table-boardered table-hover table-striped">

<thead>

<tr>

<th v-fontsize='30'> ID</th>

<th>Name</th>

<th>Ctime</th>

<th>Operation</th>

</tr>

</thead>

<tbody>

<!-- 之前，v-for中的数据，都是直接从data上的list中直接渲染过来的。现在，我们自定义了search方法。 在search方法内部，通过执行for循环，

把所有符合搜索关键字的数据，保存到一个新数组中，并返回

-->

<!-- <tr v-for="item in list" :key="item.id"> -->

<tr v-for="item in search(keywords)" :key="item.id">

<td>{{item.id}}</td>

<td v-text="item.name"></td>

<td>{{item.ctime | dateFormat()}}</td>

<td>

<a href="" @click.prevent="del(item.id)">删除</a>

</td>

</tr>

</tbody>

</table>

</div>

<script>

//自定义全局按键装饰器

Vue.config.keyCodes.f2 = 113

// 定义全局的指令

// 参数1： 指定的名称，注意，在定义的时候，不需要加v-前缀，在调用的时候必须在指令名称前加上v-前缀来进行调用

// 全局的过滤器，进行实践的格式化. pattern 如果没有实参，调用toLowerCase会报错，所以给一个默认值

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

Vue.directive('focus', {

bind: function (el) {//每当指定绑定到元素上的时候，会立即执行这个bind函数，只执行一次

// 注意，在每个函数中，每一个参数，永远都是el，表示被绑定了指定的那个元素，这个el参数，是一个原生的JS对象，也就是说可以用原生的方法

// 在元素刚绑定了指令的时候，还没有插入到DOM中去，这时候调用focus方法没有作用。因为一个元素，只有插入DOM之后，才能获取焦点

// el.focus()

},

inserted: function (el) { // inserted 表示元素插入到DOM中的时候，会立即执行inserted函数，【触发一次】

el.focus()

// 和JS行为有关的操作最好在inserted中去执行，防止行为不生效

},

updated: function (el) {//当vnode 更新的时候，会执行updated，可能会触发多次

},

})

// 自定义改变字体颜色的指令

Vue.directive('color', {

// 样式，只要通过指令绑定给了元素，不管这个元素有没有被插入到页面中去，这个元素肯定有了一个内联的样式。

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

bind: function (el,binding) {

// el.style.color = 'red'

console.log(binding.value)

el.style.color = binding.value

// 和样式相关额操作，一般都可以放在bind执行。

}

})

// Vue.directive()

Vue.filter('dateFormat', function (datestr, pattern = '') {

// 根据给定的时间字符串，得到特定的时间

var dt = new Date(datestr)

//yyyy-mm-dd

var y = dt.getFullYear()

var m = (dt.getMonth() + 1).toString().padStart(2, '0')

var d = (dt.getDate()).toString().padStart(2, '0')

// return y +'-' +m+'-'+d

// 模板字符串

if (pattern.toLowerCase() === 'yyyy-mm-dd') {

return `${y}-${m}-${d}`

} else {

var hh = dt.getHours().toString().padStart(2, '0')

var mm = dt.getMinutes().toString().padStart(2, '0')

var ss = dt.getSeconds().toString().padStart(2, '0')

return `${y}-${m}-${d} ${hh}:${mm}:${ss} ~~~`

}

})

var vm = new Vue({

el: '#app',

data: {

id: '',

name: '',

keywords: '',

list: [

{ id: 1, name: '奔驰', ctime: new Date() },

{ id: 2, name: '宝马', ctime: new Date() }

]

},

methods: {

add() {

console.log("akk")

// 分析：

// 1. 要获取到id跟name，直接从data获取

// 2. 组织出一个对象

// 3. 把这个对象，调用数组的相关方法，添加到当前data上的list中

// 4. 在Vue中，已经实现了数据的双向绑定，每当我们修改了data中的数据，Vue会默认监听到数据的改动，自动把最新的数据，应用到页面上。

// 5. 当我们意识到上面的第四步的时候，就证明大家已经入门VUe了，我们更多的是在进行VM中Model数据的操作，同时，在操作Model数据的时候，指定的业务逻辑操作

var car = { id: this.id, name: this.name, ctime: new Date() }

this.list.push(car)

this.id = this.name = ''

},

del(id) {//根据id删除数据

// 分析： 1. 如何根据Id，找到要删除这一项的索引

// 2. 如果找到索引了，直接调用数组的splice方法

this.list.some((item, i) => {

if (item.id == id) {

this.list.splice(i, 1)

// 在数组some方法中，如果return true，就会立即终止这个数组的后续循环

return true;

}

})

// 换一种方法

// this.list.findIndex(item=>{

// if(item.id == id){

// return true

// }

// })

},

search(keywords) {

// var newlist = []

// this.list.forEach(item=> {

// if(item.name.indexOf(keywords) != -1){

// newlist.push(item)

// }

// })

// return newlist

// 换种方法

// 注意： forEach some filter findIndex 这些都属于数组的新方法，都会对数组汇总的每一项，进行遍历，进行相关的操作

var newlist = this.list.filter(item => {

// 注意： ES6 中，为字符串提供了一个新方法，叫做 string.propoty.includes('要包含的字符串')

// 如果包含，则返回true，否则，返回false

if (item.name.includes(keywords)) {

return item

}

})

return newlist

},

// 过滤器的定义预发

// Vue.filter('过滤器的名称'，function(){})

},

directives:{ //自定义私有指令

'fontweight': { //设置字体粗细的

bind: function (el,binding) {

el.style.fontWeight = binding.value

}

},

'fontsize': function(el,binding){ //注意：这个function 鞥同于把代码写到了bind和update中去

el.style.fontSize = parseInt(binding.value) +'px'

}

}

})

// 让搜索框默认获取焦点. Vue 不推荐直接操作DOM

// document.getElementById('search').focus()

</script>

</body>

</html>

<!-- 过滤器调用时候的格式 {{name | 过滤器名称 }} -->

**Vue-devtool 的两种安装方式**

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

下载后本地安装

**字符串填充**

使用ES6中的字符串新方法 string.prototype.padstart(maxLength,fillString=' ') 或者String.prototype.padEnd(maxLength,fillString='') 来填充

**自定义按键修饰符 （响应键盘回车事件）**

<https://cn.vuejs.org/v2/guide/events.html#%E6%8C%89%E9%94%AE%E4%BF%AE%E9%A5%B0%E7%AC%A6>

**自定义指令**

<https://cn.vuejs.org/v2/guide/custom-directive.html>

钩子函数

* bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
* inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。
* update：所在组件的 VNode 更新时调用，**但是可能发生在其子 VNode 更新之前**。指令的值可能发生了改变，也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新 (详细的钩子函数参数见下)。

**过滤器**

vue.js 允许你自定义过滤器，可被用作一些常见的文本格式化。 过滤器可以用在两个地方，mustache插值和v-bind表达式。 过滤器应该被添加在JavaScript表达式的未卜，由管道符只是；

全局过滤器： 所有的vm实例都能共享

私有过滤器

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<p>{{msg | msgFormat('疯狂') | test}}</p>

</div>

<script>

// 定义一个Vue全局的过滤器，名字叫做msgFormat

Vue.filter('msgFormat', function (msg, arg) {

// replace 方法，第一个参数，除了可以一个字符串之外，还可以用正则

// return msg.replace('单纯','邪恶')

// return msg.replace(/单纯/g, '邪恶')

return msg.replace(/单纯/g, arg)

})

Vue.filter('test', function (msg) {

return msg + "==="

})

var vm = new Vue({

el: '#app',

data: {

msg: '曾经，我也是一个单纯的少年，单纯的我，傻傻的问，谁是世界上最单纯的男人'

},

methods: {},

filters: {//定义私有过滤器，有两个条件【过滤器名称，处理函数】

// 如果跟全局过滤器崇明，优先调用私有的

dateFormat: function (datestr, pattern) {

// 根据给定的时间字符串，得到特定的时间

var dt = new Date(datestr)

//yyyy-mm-dd

var y = dt.getFullYear()

var m = dt.getMonth() + 1

var d = dt.getDate()

// return y +'-' +m+'-'+d

// 模板字符串

if (pattern.toLowerCase() === 'yyyy-mm-dd') {

return `${y}-${m}-${d}`

} else {

var hh = dt.getHours()

var mm = dt.getMinutes()

var ss = dt.getSeconds()

return `${y}-${m}-${d} ${hh}:${mm}:${ss}`

}

}

}

})

</script>

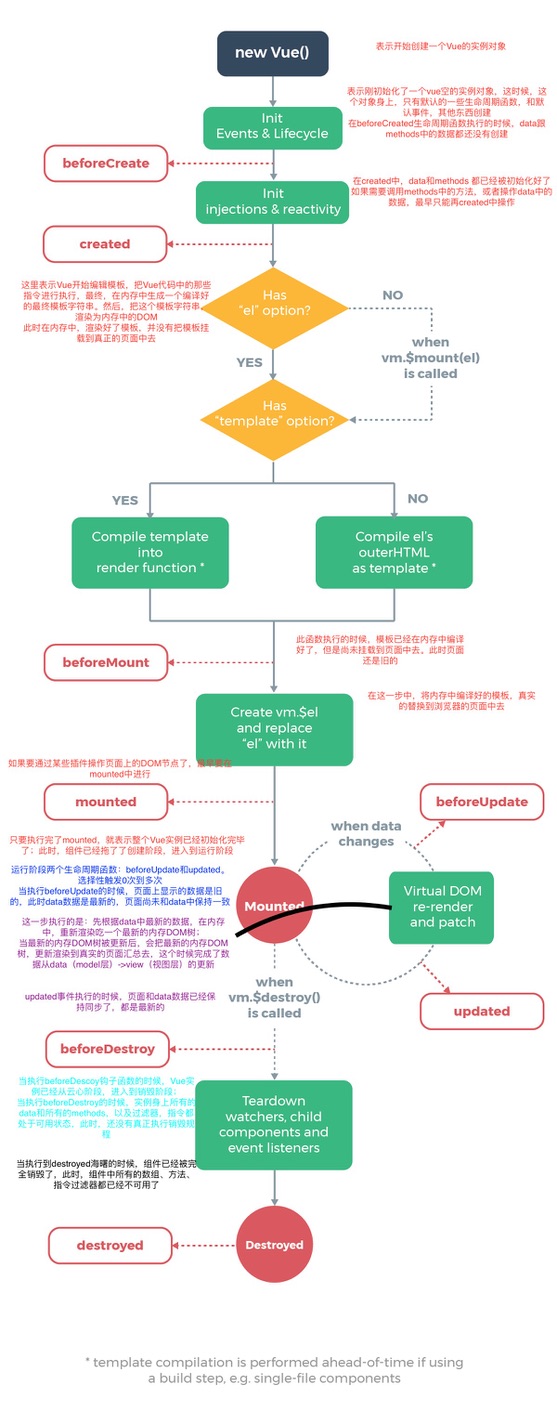
</body>

</html>

**声明周期函数**

<https://cn.vuejs.org/v2/guide/instance.html>

* 什么事声明周期： 从Vue实例创建、运行、到销毁期间，总是伴随着各种各样的时间，这些时间，统称为生命周期
* 声明周期钩子：就是声明周期事件的别名而已
* 生命周期钩子 = 生命周期函数=生命周期事件
* 主要的生命周期函数分类：
  + 创建期间的生命周期函数：
    - beforecreate: 实例刚在内存中被创建出来，此时，还没有初始化 data 跟methods属性
    - created: 实例已经在内存中ok，此时data跟methods已经创建OK，此时还没有开始编译模板
    - beforeMount： 此时已经完成了模板的编译，但是还没有挂载到页面中
    - mounted： 此时，已经将编译好的模板，挂载到了页面指定的容器中显示
  + 运行期间的生命周期函数：
    - beforeUpdate： 状态更新之前执行此函数，此时data中的状态值是最新的，但是界面上显示的数据还是旧的，因此此时还没有开始重新渲染DOM节点
    - updated： 是更新完毕之后调用此函数，此时data中的状态值和界面上显示的数据，都已经完成了更新，界面已经被重新渲染好了
  + 销毁期间的生命周期函数：
    - beforeDestroy： 实例销毁之前调用，在这一步，实例仍然完全可用
    - destroyed： Vue实例销毁后调用，调用后，Vue实例只是的所有东西都会解绑定，所有的时间监听器都会被移除，所有的子实例也会被销毁



<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<input type="button" value="修改msg" @click="msg='NO'">

<h3 id='h3'>{{msg}}</h3>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

msg: "ok"

},

methods: {

show(){

console.log("in show method")

}

},

beforeCreate(){// 这是我们遇到的第一个声明周期函数，表示实例完全被创建出来之前，会执行他

// 在beforeCreate 声明周期函数执行的时候，data和methods 中的数据都还没有被初始化

// console.log(this.msg)

// this.show()

},

created(){// 这是遇到的第二个生命周期函数

// 在created中，data和methods 都已经被初始化好了

// 如果要调用methods 中的方法，或者操作data中的数据，最早只能再created方法中

console.log(this.msg)

this.show()

},

beforeMount(){ //这是遇到的第三个生命周期函数，表示模板已经在内存中编辑完成，但是尚未把模板渲染到页面中

console.log(document.getElementById('h3').innerText) // 打印的是{{msg}}

// 在beforeMount执行的时候，页面中元素，还没有被真正替换过来，只是之前写的一些模板字符串

},

mounted(){// 这是遇到的第四个生命周期函数，表示，在内存中的模板，已经真实的挂载到页面中，用户已经可以看到渲染好的页面

console.log(document.getElementById('h3').innerText) // 打印的是ok

// 注意，mounted是实例创建期间的操作的最后一个生命周期函数，当执行完mounted就表示，实例已经被完全创建好了，此时，如果没有后续操作，页面就一动不动

},

beforeUpdate(){// 这时候，表示我们的界面还没有被更新【数据已经被更新了】

console.log('界面上的元素的内容：'+document.getElementById('h3').innerText)

console.log('data 中的msg：' + this.msg)

// 得出结论： 当执行beforeUpdate的时候，页面上显示的数据是旧的，此时data数据是最新的，页面尚未和data中保持一致

},

updated(){

console.log('界面上的元素的内容：' + document.getElementById('h3').innerText)

console.log('data 中的msg：' + this.msg)

// updated时间执行的时候，页面和data数据已经保持同步了，都是最新的

}

})

</script>

</body>

</html>

**vue-resource 实现get,post,jsonp请求**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<!-- 注意： vue-resource 依赖于vue，所以要注意先后顺序 -->

<!-- this.$http.get -->

<script src="./lib/vue-resource.js"></script>

</head>

<body>

<div id='app'>

<input type="button" value="get请求" @click="getInfo">

<input type="button" value="post请求" @click="postInfo">

<input type="button" value="jsonp请求" @click="jsonpInfo">

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

},

methods: {

// 当发起get请求之后，通过.then 来设置成功的回调函数，失败的回调函数是可选的

getInfo() {

this.$http.get('http://vue.studyit.io/api/getlunbo').then(function (result) {

//通过result.body 拿到服务器返回的成功请求

console.log(result.body)

})

},

postInfo() {

// 手动发起的Post请求，默认没有表单格式，所以，有的服务器处理不了。 这里需要emulateJSON

this.$http.post('http://vue.studyit.io/api/post', {}, { emulateJSON: true }).then(result => {

console.log(result.body)

})

},

jsonpInfo() {

// 手动发起的Post请求，默认没有表单格式，所以，有的服务器处理不了。 这里需要emulateJSON

this.$http.post('http://vue.studyit.io/api/jsonp').then(result => {

console.log(result.body)

})

}

}

})

</script>

</body>

</html>

**jsonp原理**

* 由于浏览器的安全性限制，不允许ajax访问协议不同、域名不同、端口不通的数据接口，浏览器认为这种访问不安全
* 可以通过动态创建script标签的形式，把script标签的src属性，指向数据接口的地址，因为script表现不存在跨域限制，这种数据获取方式，乘坐jsonp。 注意：根据jsonp的实现原理，知晓jsonp只支持Get请求
* 具体实现过程：
  + 现在客户端定义一个回调方法，预定义对数据的操作；
  + 再把这个回调方法的名称，通过URL传参的形式，提交到服务器的数据接口；
  + 服务器数据接口组织好要发送个客户端的数据，再拿着客户端船从过来的回调方法名称，拼接处一个调用这个方法的字符串，发送给客户端去解析执行；
  + 客户端拿到服务器返回的字符串之后，当做script脚本其解析执行，这样就能够拿到jsonp的数据了。

**通过vue-resource改造品牌列表**

**全局配置数据接口的根域名**

全局配置emulatejson选项

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<script src="./lib/vue-resource.js"></script>

<link rel="stylesheet" href="./lib/bootstrap.min.css">

<script src="./lib/bootstrap.min.js"></script>

</head>

<body>

<div id='app'>

<div class="panel panel-primary">

<div class="panel-heading">

<h3 class="panel-title">添加品牌</h3>

</div>

<div class="panel-body form-inline">

<label for="">

Name:

<input type="text" v-mode="name" class="form-control">

</label>

<input type="button" value="添加" @click="add" class="btn-primary">

</div>

</div>

<table class="table table-bordered table-hover table-striped">

<thead>

</thead>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

<th>Ctime</th>

<th>Operation</th>

</tr>

</thead>

<tbody>

<tr v-for="item in list" :key="item.id">

<td>{{ item.id }}</td>

<td>{{ item.name }}</td>

<td>{{ item.ctime }}</td>

<td><a href="" @click.prevent="del(item.id)">删除</a></td>

</tr>

</tbody>

</table>

</div>

<script>

// 如果我们通过全局配置了，请求的数据接口根域名，则，每次单独发起http请求的时候，请求的url路径，应该以相对路径开头，前面不能带/，否则不会启用根路径做拼接

Vue.http.option.root = 'http://vue.studyit.io/'

Vue.http.option.emulateJSON=true

var vm = new Vue({

el: '#app',

data: {

name: "",

list: [ //

{

id: 1, name: 'wuling', ctime: new Date()

},

{ id: 2, name: 'abc', ctime: new Date() }

]

},

created() {// 当vm实例的data和methods初始化完毕之后，vm实例会自动执行created这个生命周期函数

this.getAllList()

},

methods: {

del() {

// 用默认的根路径. 注意这里的路径只能是相对路径

this.$http.get('api/delproduct/' + id).then(result => {

if (result.body.status === 0) {

this.getAllList()

} else {

alert('删除失败！')

}

})

},

add() { //添加品牌列表到后台服务器

//分析： 1. 经过查看数据AOu接口，发现需要发送一个Post请求，this.$http.post

//2. this.$http.post() 中接收三个参数

// 第一个参数： 请求的URL地址

// 第二个参数： 要提交给服务器的数据，要以对象形式提交给服务器{name: this.name}

// 3.3 第三个参数，是一个配置对象，要以哪种表单数据类型提交过去，{emulateJSOn: true}, 以普通表单格式，将数据提交给服务器

// application/x-www-form-urlencoded

// 3. 在post方法中，使用.then 来设置成功的回调函数，如果想要拿到成功的结果，需要result。body

// this.$http.post('http://vue.studyit.io/api/addproduct', { name: this.name }, { emulateJSON: true }.then(result => {

this.$http.post('http://vue.studyit.io/api/addproduct', { name: this.name }).then(result => { //这里使用全局的emulatejson配置

if (result.body.status === 0) {

this.getAllList()

this.name = ''

} else {

alert('添加失败')

}

})

},

getAllList() { //获取所有的品牌列表

// 分析：1. 由于已经导入了vue-resource这个包，所以，可以直接通过 this.$http 来发起数据请求

// 2.根据接口API文档，知道，火气列表的时候，应该发起一个get请求

// 3. this.$http.get（url）.then(function(result){})

// 4. 通过then指定回调函数之后，在回调函数中，可以拿到数据服务器返回的result

// 5. 先判断result.status 是否等于0，如果等于0，就成功了，可以把result.message复制给this.list。 如果不等于0，可以弹框提醒，获取数据失败

this.$http.get('http://vue.studyit.io/api/getprodlist').then(result => {

//注意，通过$http获取的数据都在body里面

result = result.body

if (result.status === 0) {

this.list = result.message

} else {

alert('获取数据失败')

}

})

},

}

})

</script>

</body>

</html>

**动画**

为什么要有动画：动画能够提高用户的，能帮助用户更好的理解

使用过度类名实现动画

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<!-- 2. 自定义两组样式，来控制transition内部元素实现动画 -->

<style>

/\* v-enter 【这是一个时间点】是进入之前，元素的起始状态，此时还没有进入 \*/

/\* v-leave-to【这是一个时间点】是动画离开之后的，离开的重视状态，此时，元素动画已经结束 \*/

.v-enter,

.v-leave-to {

opacity: 0;

transform: translateX(80px);

}

/\* v-enter-active 【进入动画的时间段 v-leave-active 离开动画的时间段 \*/

.v-enter-active,

.v-leave-active {

transition: all 0.8s ease;

}

.my-enter,

.my-leave-to {

opacity: 0;

transform: translateY(80px);

}

/\* v-enter-active 【进入动画的时间段 v-leave-active 离开动画的时间段 \*/

.my-enter-active,

.my-leave-active {

transition: all 0.8s ease;

}

</style>

</head>

<body>

<div id='app'>

<!-- 通过button 控制h3 显示还是隐藏 -->

<!-- <input type="button" value="toggle" @click="flag=!flag">

<h3 v-if="flag">这是一个h3</h3> -->

<!-- 1. 使用transition元素，把需要被控制动画控制的元素宝国起来 -->

<input type="button" value="toggle" @click="flag=!flag">

<transition>

<h3 v-if="flag">这是一个h3</h3>

</transition>

<input type="button" value="toggle6" @click="flag2=!flag2">

<!-- name 字段可以区分不同组之间的动画 -->

<transition name="my">

<h6 v-if="flag2">这是一个h6</h6>

</transition>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

flag: false,

flag2: false

},

methods: {}

})

</script>

</body>

</html>

使用第三方animate.css实现动画

**组件**

什么事组件：组件的出现，就是为了拆分Vue实例的代码量，能够让我们以不同的组件，来划分不同的功能模块，将来我们需要什么样的功能，就可以去调用对应的组件即可。

组件化和模块化的不通：

* 模块化： 是从代码逻辑的角度进行划分的，方便代码分层开发，保证每个功能模块的职能单一。
* 组件化： 是从UI界面的角度进行划分的。方便UI组件的重用。

**组件的定义方式**

**私有组件**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<!-- 如果要使用组件，直接，把组件的名称，以html标签的形式，引入到页面中即可 -->

<!-- 使用的时候不能用小驼峰，得换成-。 如果都是小写，可以直接使用 -->

<!-- <my-com1></my-com1> -->

<mycom3></mycom3>

</div>

<!-- 创建组件的第三种方式 -->

<!-- 在被控制的#app外面，使用template 元素，定义组件的html模板结构 -->

<template id="tmp1">

<div>

<h1>这是通过template元素，在外部定义的组件结构，这个方式，有代码的智能提示和高亮</h1>

<h4>好用</h4>

</div>

</template>

<template id="tmp2">

<h1>这是私有的login2组件</h1>

</template>

<div id='app2'>

<mycom3></mycom3>

<login2></login2>>

</div>

<script>

//1.1 使用Vue.extend 来创建全局的Vue组件

var com1 = Vue.extend({

template: '<h3>这是通过Vue.extend创建的组件</h3>' //通过template 属性，指定了组件要战术的html结构

})

//1.2 使用Vue.component（‘组件名称’，创建出来的组件模板对象）.注册的时候可以用驼峰

Vue.component('myCom1', com1)

// Vue.component 第一个参数，组件的名称，将来在引用组件的时候，就是一个标签形式来引入

// 第二个参数，Vue。extend 创建的组件，其中，template 就是组件将来要展示的html内容

Vue.component('mycom2', Vue.extend({

template: '<h3>这是使用Vue.extend 创建的组件</h3>'

}))

//创建组件的第二种方式

Vue.component('mycom2',{

// template: '<h3>这是使用Vue.extend 创建的组件</h3>',

// 注意：无论哪种方式创建出来的组件，组件的template属性指向的模板内容，必须切只能有唯一的一个根元素

// template: '<h3>这是使用Vue.extend 创建的组件</h3><span>123</span>'

template: '<div><h3>这是使用Vue.extend 创建的组件</h3><span>123</span></div>'

})

// 创建组件的第三种方式

Vue.component('mycom3',{

template: '#tmp1'

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {},

})

var vm2 = new Vue({

el: '#app2',

data: {

},

methods: {},

components: { //定义实例内部私有组件

login: {

template: '<h1>这是私有的login组件</h1>'

},

login2: {

template: '#tmp2'

}

}

})

</script>

</body>

</html>

**组件中的data和methods**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<mycom1></mycom1>

</div>

<script>

Vue.component('mycom1', {

template: '<h1>这是全局组件----{{msg}}</h1>',

// 1. 组件可以由自己的data数据，

// 2. 组件的data 和实例的data有点不一样，实例中的data可以为一个对象，但是组件中的对象必须是一个方法

// 3. 组件中的data 除了必须为一个方法之外，这个方法内部，还必须返回一个对象才行

// 4. 组件中的data 诗句，使用方式和实例中的data使用方式完全一样

data: function () {

return {

msg: '这是组件中的data定义的数据'

}

}

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {}

})

</script>

</body>

</html>

**为什么组件中的data必须是一个function**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<counter></counter>

</div>

<template id="tmp1">

<div>

<input type="button" value="+1" @click=increment>

<h3>{{count}}</h3>

</div>

</template>

<script>

//定义一个外部的对象

// var dataObj = {count: 0}

Vue.component('counter',{

template: '#tmp1',

data: function(){

// 返回外部的对象，如果页面上有多个组件，会互相影响，所以不能用

// return dataObj

// 必须返回内部新创建的对象，多个组件之间不会互相影响

return {count: 0}

},

methods: {

increment(){

this.count++

}

}

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {}

})

</script>

</body>

</html>

**组件切换 v-if**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<a href="" @click.prevent="flag=true">登录</a>

<a href="" @click.prevent="flag=false">注册</a>

<login v-if="flag"></login>

<register v-else="flag"></register>

</div>

<script>

Vue.component('login',{

template: '<h3>登录组件</h3>'

})

Vue.component('register',{

template: '<h3>注册组件</h3>'

})

var vm = new Vue({

el: '#app',

data: {

flag: false

},

methods: {}

})

</script>

</body>

</html>

**组件切换 component**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<a href="" @click.prevent="comName='login'">登录</a>

<a href="" @click.prevent="comName='register'">注册</a>

<!-- Vue 提供了component，来展示对应名称的组件 -->

<!-- componennt 是一个占位符，:is 属性，可以用来指定要战术的组件名称 -->

<!-- <component :is="'login'"></component> -->

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

<!-- 总结 当前学习了几个Vue提供的标签 -->

<!-- component, template, transition, transitionGroup -->

</div>

<script>

Vue.component('login',{

template: '<h3>登录组件</h3>'

})

Vue.component('register',{

template: '<h3>注册组件</h3>'

})

var vm = new Vue({

el: '#app',

data: {

flag: false,

comName: 'login'

},

methods: {}

})

</script>

</body>

</html>

**父组件向子组件传递数据**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<!-- 父组件可以在引用子组件的时候，通过属性绑定的形式（v-bind），把需要传递给子组件的数据，以属性绑定的形式，传递到子组件内部，供子组件使用 -->

<com1 v-bind:parentmsg="msg"></com1>

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

msg: '123 这是父组件的数据'

},

methods: {},

components: {

com1: {// 默认情况下子组件无法访问到父组件中的data上的数据和methods 中的方法

// template: '<h1>这是子组件---{{msg}}</h1>'

template: '<h1>这是子组件---{{parentmsg}}</h1>',

// 注意，组件中的所有props中的数据，都是父组件传递给子组件的。

// props 中的数据，都是只读的

props: ['parentmsg'] ,//把父组件传递过来的parentmsg属性，现在props数组中定义一下，这样才能使用这个数据

data() { //注意，子组件的data数据，并部署通过父组件传递过来的，而是子组件自身私有的，，比如，子组件通过ajax 请求回来的数据，都可以放在data身上

// data上的数据都是可读可写的

return { title: 123,}

}

}

}

})

</script>

</body>

</html>

**父组件把方法传递给子组件（子组件向父组件传递数据）**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<!-- 父组件向子组件传递方法，使用的是事件绑定机制（v-on）。当我们自定义了一个事件属性之后，那么，子组件就能够通过某系方式，来调用传递进去的这个方法 -->

<com2 @func="show"></com2>

</div>

<template id="tmp1">

<div>

<h1>这是子组件</h1>

<input type="button" value="这是子组件中的按钮，点击他，会触发父组件传递过里啊的方法" @click="myclick">

</div>

</template>

<script>

//定义了一个字面量类型的组件模板对象

var com2 = {

data() {

return {

sonmsg: {name: 'xiaotouerzi',age:6}

}

},

template: '#tmp1', //通过制定了一个id，表示说，要去加载指定id的template元素中的内容，当做组件的html结构

methods: {

myclick(){

// 当点击子组件按钮的时候，如何拿到父组件传递后来的func方法，并调用？

// emit 英文愿意是触发调用、发射的意思，

//

// this.$emit('func',123,345)

this.$emit('func',this.sonmsg)

}

}

}

var vm = new Vue({

el: '#app',

data: {

fromson: {}

},

methods: {

show(data){

// console.log('调用父组件上的show方法----' + data)

// 通过把show 方法传递给子组件，子组件调用的时候，在函数中把子组件的数据赋值到父组件上，这样就实现了子组件向父组件传递数据

this.fromson = data

}

},

components: {

com2

}

})

</script>

</body>

</html>

**组件练习 localStorage的使用**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<!-- <script src="./lib/bootstrap.min.js"></script> -->

<link rel="stylesheet" href="./lib/bootstrap.min.css">

</head>

<body>

<div id='app'>

<ul class="list-group">

<li class="list-group-item" v-for="item in list" :key="item.id">

<span class="badge">评论人： {{item.user}}</span>

{{item.content}}

</li>

</ul>

<cmtbox @func="loadComments"></cmtbox>

</div>

<template id="tmp1">

<div>

<div class="form-group">

<label for="">评论人：</label>

<input type="text" class="form-control" v-model="user">

</div>

<div class="form-group">

<label for="">评论内容：</label>

<textarea class="form-control" v-model="content"></textarea>

</div>

<div>

<input type="button" value="发表评论" class="btn-primary btn" @click="postComment">

</div>

</div>

</template>

<script>

var commentBox = {

template: '#tmp1',

//因为这些都在子组件中使用，所以定义在子组件里

data(){

return {

user: '',

content: ''

}

},

methods: {

postComment(){//发表评论的方法

// 分析： 发表品论的业务逻辑

// 1. 评论数据保存到哪里去了？ 放到了localstorage中

// 2. 先组织出一个最新的品论数据对象

// 3. 想办法，把第二部中得到的评论对象，保存到locaoStorage中：

// 3.1localstorage只支持存放字符串，要先调用JSON.stringify

// 3.2 在保存最新的评论数据之前，要先从localstorage获取到之前的评论数据（string），转换为一个数据对象，然后把最新的评论内容push到数组

// 3.3 如果获取到的localstorage 中的评论字符串为空，则可以返回一个’[]‘，让json.parse去转换

// 3.4 把最新的评论列表数组再次调用JSON.stringify 转为数组字符串，然后调用localStorage.setItem()

var comment= {id: Date.now(),user: this.user,content: this.content}

// 从localstorage中获取所有额评论

var list = JSON.parse(localStorage.getItem('cmts') || '[]')

list.push(comment)

localStorage.setItem('cmts',JSON.stringify(list))

this.user = this.content = ''

this.$emit('func')

}

}

}

var vm = new Vue({

el: '#app',

data: {

list: [

{ id: 1, user: '李白', content: '天生我才必有用' },

{ id: 2, user: '江小白', content: '劝君更尽一杯酒' },

{ id: 3, user: '小马', content: '我姓马，风吹草地现牛羊的马' }

],

},

methods: {

loadComments(){ //从本地的localstorage中，加载评论列表

var list = JSON.parse(localStorage.getItem('cmts')|| '[]')

this.list = list

}

},

created(){

this.loadComments()

},

components: {

'cmtbox': commentBox

}

})

</script>

</body>

</html>

**路由**

1. 后端路由： 对于普通的网站，所有的超链接都是URL地址，所有的URL都对应服务器上对应的资源，
2. 前端路由： 对于单页面应用程序来说，主要同伙URL中的hash（#）来实现不同页面之间的切换，同时，hash有一个特点： http请求中不会包含hash相关的内容，所以，单页面程序中的页面跳转主要用hash实现
3. 在单页面应用程序中，这种通过hash改变来切换页面的方式，称作前端路由。

锚点。就是这里面的hash

参考：www.cnblogs.com/joyho/articles/4430148.html

百度vue-router

**vue-router 基本使用 router-link redirect query传参 params传参**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<!-- 1. 安装vue-router路由模块 -->

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

<!-- 实现选中的标签高亮 router-link-active 是激活的默认类，可以通过路由的构造选项linkActiveClass 来全局配置-->

<style>

.router-link-active {

color: red;

font-weight: 800;

font-size: 40px;

}

</style>

</head>

<body>

<div id='app'>

<!-- 这是vue-router提供的元素，专门用来当做占位符的，将来，路由规则匹配到的组件，就会展示到router-view中去 -->

<!-- 所以，我们可以把router-view认为是一个占位符 -->

<!-- 可用，但不推荐 -->

<!-- <a href="#/login">登录</a>

<a href="#/register">注册</a> -->

<!-- 通过params 传参 -->

<router-link to="/login/11/lisi/">登录</router-link>

<!-- router-link 默认渲染为a标签 加上tag 可以渲染为其他元素-->

<!-- 通过query传参 -->

<!-- 如果在路由中，使用查询字符串，给路由 127.0.0.1#/login?id=10, 这个可以直接访问到-->

<router-link to="/register?id=10&name=zs" tag="span">注册</router-link>

<router-view></router-view>

</div>

<script>

var login = {

// 这里￥route可以没有this

template: '<h1>登录组件 ---{{$route.params.id}}---{{$route.params.name}}</h1>',

data(){

return {

msg: 123

}

},

created(){ //组件的生命周期函数

// 这里必须要有this

console.log(this.$route)

console.log(this.$route.query.id)

}

}

var register = {

// 这是通过query传参

template: '<h1>注册组件---{{this.$route.query.id}} --- {{this.$route.query.name}}</h1>'

}

// 不能这样定义，用在routes里面

// Vue.component('login',{

// template: '<h1>登录组件</h1>'

// })

// 2. 创建一个路由对象，当导入vue-router包之后，在window全局对象中，就有了一个路由的构造函数，叫做VueRouter

// 在new路由对象的时候，可以为构造函数，传递一个配置对象

var routerObj = new VueRouter({

// route //在配置对象中的toute表示【路由匹配规则】的意思

routes: [ //路由匹配规则

// 每个路由规则，都是一个对象，这个规则对象身上，必须有两个必须的属性:

// 属性1是path，表示监听那个路由链接地址

// 属性2是component，表示，如果路由是前面匹配到的path，则展示component属性对应的组件

// 注意： component的属性值，必须是一个组件的模板对象，不能是组件的引用名称

{ path: '/login/:id/:name', component: login },

{ path: '/register', component: register },

{ path: '/', redirect: '/login' } //默认显示登录组件

],

// router-link-active是默认class

linkActiveClass: 'router-link-active'

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {},

// 3. 将路由规则对象，注册到vm实例上，用来监听url地址变化，然后展示对应的组件

router: routerObj

})

</script>

</body>

</html>

**使用children属性实现子路由**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

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

</head>

<body>

<div id='app'>

<router-link to='/account'>Account</router-link>

<router-view></router-view>

</div>

<template id="tmp1">

<div>

<h1>这是Account组件</h1>

<router-link to='/account/login'>登录</router-link>

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

<router-view></router-view>

</div>

</template>

<script>

var account = {

template: "#tmp1"

}

var login = {

template: "<h1>登录</h1>"

}

var register = {

template: "<h1>注册</h1>"

}

var router = new VueRouter({

routes: [{

path: '/account',

component: account,

children: [

// 使用children属性实现子路由，同时，子路由的path前面不要带/,否则永远以根路径开始请求，这样不方便我们用户去理解URL地址

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

{ path: "register", component: register }

]

}

]

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {},

router

})

</script>

</body>

</html>

**路由命名视图实现经典布局**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

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

<style>

html, body{

margin: 0;

padding: 0;

}

.header{

background-color: orange;

height: 80px;

}

h1{

margin: 0;

padding: 0;

font-size: 16px;

}

/\* 这里flex 让left 跟main 显示在同一行 \*/

.container {

display: flex;

height: 600px

}

.left{

background-color: lightgreen;

/\* 这里flex 大概意思是让占20% \*/

flex: 2

}

.main{

background-color: red;

flex: 8

}

</style>

</head>

<body>

<div id='app'>

<!-- 命名视图 -->

<router-view></router-view>

<div class="container">

<router-view name="left"></router-view>

<router-view name="main"></router-view>

</div>

</div>

<script>

var header = {

template: "<h1 class='header'> Header区域</h1>"

}

var leftBox = {

template: "<h1 class='left'> left侧边栏区域</h1>"

}

var mainBox = {

template: "<h1 class='main'> main主区域</h1>"

}

var router = new VueRouter({

routes: [

// 属性名会自动加上单引号

{path: '/',components: {

default: header,

'left': leftBox,

main: mainBox

}},

// { path: '/left', component: leftBox },

// { path: '/main', component: mainBox }

]

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {},

router

})

</script>

</body>

</html>

**watch**

computed: 属性的结果会被缓存，除非依赖的响应是的属性变化才会被重新计算，主要当做属性来使用

methods方法，表示一个具体的操作，主要书写业务逻辑

watch 一个对象，键是需要观察的表达式，值是对应回调函数。 主要用来监听某些特定的数据的变化，从而进行某些具体的业务逻辑操作；可以当错是computed 和methods 的结合体

**watch 基本使用& 计算属性computed**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

</head>

<body>

<div id='app'>

<!-- 方式1： 使用keyup时间监听text 改变 -->

<!-- <input type="text" v-model="firstname" @keyup="pinjie"> +

<input type="text" v-model="lastname" @keyup="pinjie"> =

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

<!-- 方式2，用watch 方法3 计算属性-->

<input type="text" v-model="firstname"> +

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

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

</div>

<script>

var vm = new Vue({

el: '#app',

data: {

firstname: '',

lastname: '',

// 方式1，方式2

// fullname: ''

},

methods: {

// 方式1

// pinjie(){

// this.fullname = this.firstname + " " +this.lastname

// }

},

// watch: {

// // 方式2，使用watch

// //使用这个属性，可以见识data中指定数据的变化，然后出发这个watch中对应的function处理函数

// //对象的属性名是需要加单引号的，但是框架会帮我们自动加。 如果属性名有-， 如first-name,则不需加单引号

// firstname: function (newVal, oldVal) {

// // console.log('监视到了firstname变化')

// // this.fullname = this.firstname + '-' +this.lastname

// console.log(newVal + '====' + oldVal)

// this.fullname = newVal + '-' + this.lastname

// },

// 'lastname': function (newVal) {

// this.fullname = this.firstname + '-' + newVal

// }

// }

computed: {

//在computed中，可以定义一些属性，叫做计算属性，计算属性的本质，就是一个方法；只不过，我们在使用这些计算属性的时候，是把他们的名称

// 直接当做属性使用的，并不会把计算属性，当做方法去调用。

// 注意，计算属性，在引用的时候，一定不要加（），直接把他当做普通属性去使用就好了

// 注意： 只要计算属性这个function内部所用到的任何data中的数据发生变化，就会立即重新计算这个属性的值

// 注意： 计算属性的求职结果，会被缓存起来，方便下次直接使用；如果计算属性方法中，所依赖的任何数据，都没有发生过变化，则不会重新对计算属性求值

'fullname': function(){

return this.firstname + '-' + this.lastname

}

}

})

</script>

</body>

</html>

**watch 监视路由变化**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>document</title>

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

<!-- 1. 安装vue-router路由模块 -->

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

</head>

<body>

<div id='app'>

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

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

<router-view></router-view>

</div>

<script>

var login = {

template: '<h1>登录组件</h1>',

data() {

return {

msg: 123

}

},

}

var register = {

// 这是通过query传参

template: '<h1>注册组件</h1>'

}

var routerObj = new VueRouter({

routes: [

{ path: '/login', component: login },

{ path: '/register', component: register },

{ path: '/', redirect: '/login' } //默认显示登录组件

],

})

var vm = new Vue({

el: '#app',

data: {

},

methods: {},

router: routerObj,

watch: {

'$route.path': function(newVal,oldVal){

console.log(newVal +'-----' + oldVal)

}

}

})

</script>

</body>

</html>