创建一个vue实例,Vue是一个构造函数

//控制的这个元素区域就是mvvm中的v

<div id=”app”>

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

</div>

<script>

//new出来的这个vm对象就是mvvm中的vm

var vm = new Vue({

el:”#app”, //表示当前我们new的这个Vue实例,要控制页面上的哪个区域

//data就是mvvm中的m

data:{ //data 属性是存放的el中要用到的数据

msg:”Vue”,

msg2:”<h1>2323</h1>”,

mytitle : “asdsd”

},

})

</scritpt>

**Vue指令**

**v-cloak** 解决网络不好情况下用户看到了插值表达式的情况，解决闪烁的问题

<style>

[v-vloak]{

display:none;

}

</style>

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

**v-text**  默认v-text是没有闪烁问题的,但会覆盖原有的内容，但是插值表达式不好覆盖内容，但会有闪烁，这是这两者的区别,两者都不能解析html标签

<p v-text=”msg” >+++</p>

**v-html** 解析html会覆盖原有内容

<div v-html=”msg2”>123</div>

**v-bind** 是vue中提供用于绑定属性的指令

简写 **:title=”mytitle” 可以简写为一个 :**

<input type=”button” value=”提交” v-bind:title=”mytitle”> //还可以用js表达式 mytitle + ‘123’

**v-on** vue中提供了v-on 事件绑定机制

**简写 @click**

<input type=”button” value=”提交” v-bind:title=”mytitle” v-on:click=”show”>

var vm = new Vue({

el:”#app”, //表示当前我们new的这个Vue实例,要控制页面上的哪个区域

//data就是mvvm中的m

data:{ //data 属性是存放的el中要用到的数据

msg:”Vue”,

msg2:”<h1>2323</h1>”,

mytitle : “asdsd”

},

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

show:function(){

alert(‘hello’);

}

}

})

**事件修饰符**

**.stop** 阻止冒泡

<button @click.stop=”showalldiv”></button> //在事件后面加.stop就好

.prevent 阻止默认事件

<a href=”[www.baidu.com](http://www.baidu.com)” @click.prevent=”go”></a> //在事件后面加. prevent就好

.capture添加事件监听器时使用事件捕获模式，实现捕获触发事件机制，从外到内触发事件

<div @click.capture=”div”>

<input @click=”btn”>

</div>

**.self** 你点我 我才触发

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->

<!-- 即事件不是从内部元素触发的 -->

<div v-on:click.self="doThat">...</div>

.once<!-- 点击事件将只会触发一次 -->

<a v-on:click.once="doThis"></a>

**.passive**

<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->

<!-- 而不会等待 `onScroll` 完成 -->

<!-- 这其中包含 `event.preventDefault()` 的情况 -->

<div v-on:scroll.passive="onScroll">...</div>

.stop 和 .self的区别

.stop阻止了所有的冒泡,.self只负责自己的冒泡

**v-model 双向数据绑定**

v-bind 只能实现数据的单向数据绑定,从m自动绑定到v，无法实现数据的双向数据绑定

使用v-model 指令，可以实现表单元素和model中数据的双向数据绑定

注意v-model 只能运用在表单元素中

input(radio,text,address,email…) select checkbox textarea

<input type=”text” v-model:value=”msg”>

var vm = new Vue({

el:”#app”,

data:{

msg:”Vue”,

msg2:”<h1>2323</h1>”,

mytitle : “asdsd”

},

methods:{

show:function(){

alert(‘hello’);

}

}

})

**vue中的class绑定**

.red{

color:red;

}

.green{

color:green;

}

.active{

color:blue;

}

第一种使用方式，直接传递一个数组

<h1 :class=”[‘red’,’green’]”>asd</h1>

第二种可以在数组中使用三元表达式

<h1 :class=”[‘red’,’green’,flag?’active’:’’]”>asd</h1>

var vm = new Vue({

el:’#app’,

data:{

flag: false

}

})

第三种在数组里面写对象

<h1 :class=”[‘red’,’green’,{‘active’:flag}]”>asd</h1> //为true就应用active,比三元表达式少些代码

第四种直接使用对象  
我们可以传给 v-bind:class 一个对象，以动态地切换 class：

<div v-bind:class="{ active: isActive }"></div>

上面的语法表示 active 这个 class 存在与否将取决于数据 property isActive 的 [**truthiness**](https://developer.mozilla.org/zh-CN/docs/Glossary/Truthy)。

你可以在对象中传入更多字段来动态切换多个 class。此外，v-bind:class 指令也可以与普通的 class attribute 共存。当有如下模板：

<div

class="static"

v-bind:class="{ active: isActive, 'text-danger': hasError }"

></div>

和如下 data：

data: {

isActive: true,

hasError: false

}

结果渲染为：

<div class="static active"></div>

当 isActive 或者 hasError 变化时，class 列表将相应地更新。例如，如果 hasError 的值为 true，class 列表将变为 "static active text-danger"。

绑定的数据对象不必内联定义在模板里：

<div v-bind:class="classObject"></div>

data: {

classObject: {

active: true,

'text-danger': false

}

}

渲染的结果和上面一样。我们也可以在这里绑定一个返回对象的[**计算属性**](https://cn.vuejs.org/v2/guide/computed.html)。这是一个常用且强大的模式：

<div v-bind:class="classObject"></div>

data: {

isActive: true,

error: null

},

computed: {

classObject: function () {

return {

active: this.isActive && !this.error,

'text-danger': this.error && this.error.type === 'fatal'

}

}

}

**vue中的style内联样式**

v-bind:style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS property 名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用引号括起来) 来命名：

<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

data: {

activeColor: 'red',

fontSize: 30

}

直接绑定到一个样式对象通常更好，这会让模板更清晰：

<div v-bind:style="styleObject"></div>

data: {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

当 v-bind:style 使用需要添加[**浏览器引擎前缀**](https://developer.mozilla.org/zh-CN/docs/Glossary/Vendor_Prefix)的 CSS property 时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>

**v-for**

**循环数组**

<p v-for=”item in list”>

{{item}}

</p>

//加索引

<p v-for=”(item,i) in list”>

索引{{i}}---值{{item}}

</p>

var vm = new Vue({

el:’#app’,

data:{

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

}

})

**循环对象数组**

<p v-for=”user in list”>

ID{{user.id}}----name{{user.name}}

</p>

//加索引

<p v-for=”(user,i) in list”>

ID{{user.id}}----name{{user.name}}---索引{{i}}

</p>

var vm = new Vue({

el:’#app’,

data:{

list:[

{id:1,name:’zs1’},

{id:2,name:’zs2’}

]

}

})

**循环对象**

<p v-for=”(val,key) in list”> //注意：第一个是val，第二是key第三是索引

值{{val}}----键{{key}}

</p>

<p v-for=”(val,key,i) in list”> //注意：第一个是val，第二是key第三是索引

值{{val}}----键{{key}}

</p>

var vm = new Vue({

el:’#app’,

data:{

list:{

id:1,

name:’asdd’,

gender:’男’

}

}

})

**迭代数字**

<p v-for=”count in 10”> //**count从1开始**

这是第{{count}}次循环

</p>

**key的使用注意事项**

**v-for循环的时候 key属性只能使用 number或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的同时，指定唯一的字符串/数字类型:key值**

**vi-if 和 v-show**

**v-if 的特点是每次都会删除或创建元素,v-show的特点每次不会重新进行dom的删除和创建，只是切换了display的样式**

**v-if有较高的切换性能消耗**

**v-show有较高的初始渲染消耗**

**如果元素涉及到频繁的切换不要使用v-if，推荐使用v-show**

**如果元素可能永远不会被显示出来被看到，推荐使用v-if**

<button @click=”toggle” type=”button”>切换</button>

<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>

var vm = new Vue({

el:’#app’,

data:{

flag:false

},

methods:{

toggle(){

this.flag = !this.flag

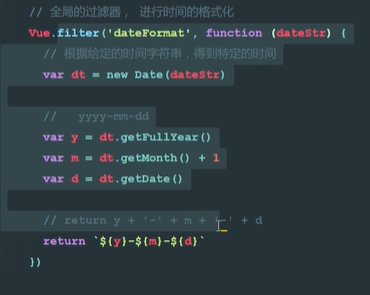
}

}

})

**全局过滤器**

全局过滤器所有的vm都能共享



使用方式



过滤器传参







**私有过滤器**  
let vm = new Vue({

el:’#app2’,

data:{

dt:new Date()

},

metthods:{},

filters:{ //这里写私有过滤器

dateFormat(过滤器名称):function(){ //处理函数

}

}

})

**过滤器调用的时候，采用的是就近原则，如果私有过滤器和全局过滤器名称一致，这时候优先调用私有过滤器**

按键修饰符

以下的系统提供的

.enter 回车

.tab tab

.esc 退出

.space 空格

.up 上

.down 下

.left 左

.right 右

自定义按键修饰符

@keyup.113 F2

你还可以通过全局 config.keyCodes 对象[**自定义按键修饰符别名**](https://vuejs.bootcss.com/api/#keyCodes)：

// 可以使用 `v-on:keyup.f1`

Vue.config.keyCodes.f1 = 112

**自定义全局指令**

v-focus

<input type=”text” v-focus>

使用vue.directive()定义全局指令

参数1，指令的名称，注意定义的时候指令的名称前面不需要加v-

但是在调用的时候，必须在指令名称前加上v-

参数2，是 一个对象，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){//每当指令绑定到元素上的时候，会立即执行这个bind函数，只执行一次

注意：在每个函数中，第一个参数永远是el，表示被绑定了指令的那个元素，这个el参数是一个原生的js对象

在元素刚绑定指令的时候，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(){//inserted表示元素 插入到DOM中的时候，会执行inserted函数[触发一次]

el.focus()

},

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

}

})

一个指令定义对象可以提供如下几个钩子函数 (均为可选)：

* 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)。

v-color

<input type=”text”>

自定义样式指令

vue.directive(‘color’,{

//样式只要通过指令绑定给力元素，不管这个元素有没有插入到DOM，这个元素肯定有一个内联样式，将来元素肯定会插入到DOM之后浏览器的渲染引擎必然会解析样式，应用到这个元素

bind:function(el){

el.style.color = ‘red’

}

})

使用钩子函数的第二个binding参数拿到传递的值

vue.directive(‘color’,{

bind:function(el,binding){

el.style.color = binding.value //拿到传递的值

}

})



**自定义私有指令**

let vm = new Vue({

el:’#app’,

data:{},

methods:{},

filters:{},

directives:{ //自定义私有指令

‘fontweight’指令名称:{对象

bind:function(el,binding){

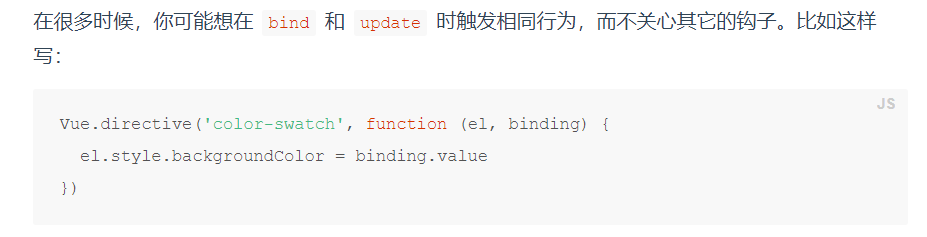
el.style.fontWeight = binding.value

}

}

}

})

自定义指令简写  


let vm = new Vue({

el:’#app’,

data:{},

methods:{},

filters:{},

directives:{ //自定义私有指令

‘fontsize’:function(el,binding){//注意这个function等同于把代码写到了bind和update中去

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

}

}

})

**Vue生命周期函数**

**组件创建期间的4个钩子函数**

<div id=”app”>

<h3 id=’h31’>{{msg}}</h3>

</div>

let vm = new Vue({

el:’#app’,

data:{

msg:’ok’

},

methods:{

show(){

console.log(‘执行了show方法’);

}

},

beforeCreate(){

//这是遇到的第一个生命周期函数，表示实例完全被创建出来之前，会执行它

注意：在beforeCreate生命周期函数执行的时候，data和methods中的数据都还没有初始化

console.log(this.msg); //undefined

this.show();//报错 is not function

},

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，只能在created中操作

console.log(this.msg); //ok

this.show();//执行了show方法

},

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了，但尚未把模板渲染到页面中

console.log(document.getElementById(‘h31’).innerText);//{{msg}}

在beforeMount执行的时候，页面中的元素，还么有真正的替换过来，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，表示内存中的模板已经真实的挂载到了页面中，用户已经可以看到渲染好的页面了

console.log(document.getElementById(‘h31’).innerText);//ok

注意mounted是实例创建期间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，如果没有其他操作，就静静的在内存中

},

接下来是运行中俩个事件

beforeUpdate(){

//这时候，表示界面还没更新，但数据更新了

假设修改了msg为no，但页面上的还是ok

console.log(document.getElementById(‘h31’).innerText);//ok

},

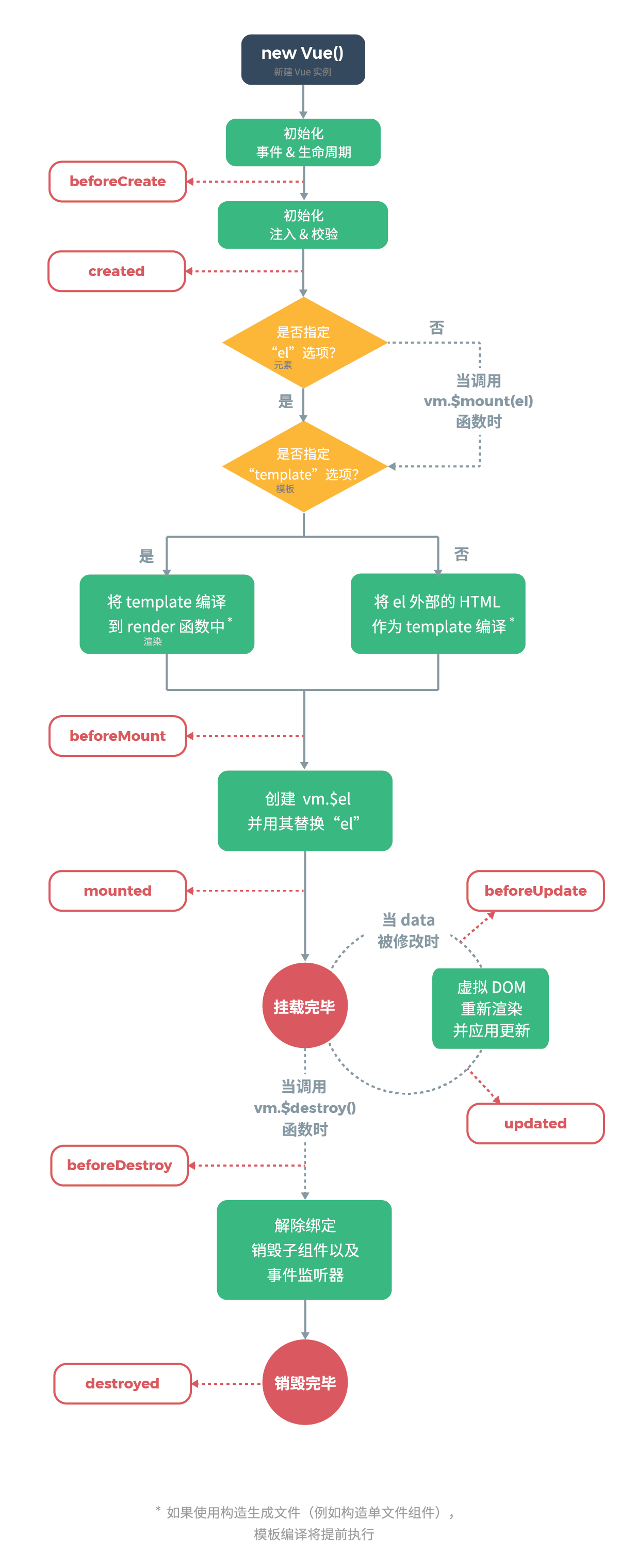
updated(){

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

console.log(document.getElementById(‘h31’).innerText);//no

}

})



这些是组件运行阶段的生命周期函数，只有2个,这俩事件会根据data的改变有选择性的触发0到多次

←这里表示vue开始编译模板，把vue代码中的那些指令进行执行，最终在内存中 生成一个编译好的最终模板字符串然后，把这个模板字符串渲染为内存中的DOM此时，只是在内存中渲染好了模板，并没有把模板挂载到真正的页面中去

当执行beforeDestroy钩子函数的时候，Vue实例已经从运行阶段，进入到了销毁阶段

当执行beforeDestroy的时候，实例身上所有的data和所有的methods，以及过滤器、指令。。。。都处于可用状态，此时，还没有真正执行销毁的过程

当执行到destroyed函数的时候，组件已经被完全销毁了，此时，组件中所有的数据、方法、指令、过滤器。。。。。。都已经不可用了

组件化和模块化的区别

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，前端的组件化，方便UI组件的重用

**Vue组件**

**创建组件(全局)**

第一种

使用vue.extend来创建全局的vue组件

var com1 = Vue.extend({

template:’<h3>这是使用vue.extend创建的组件</h3>’//通过template属性，指定了组件要展示的HTML结构

})

第二种

使用vue.component(‘组件名称’,创建出来的组件模板对象)

Vue.component(‘myCom1’,com1)

Vue.component(‘mycom1’,com1)

Vue.component(‘mycom1’,Vue.extend({

template:’<h3>这是使用vue.extend创建的组件</h3>’

}))

直接使用Vue.component创建组件

Vue.component(‘mycom2’,{

//无论是哪种方式创建出来的组件，组件的template属性指向的模板内容，必须有且只能有唯一的一个根元素

template:’<h3>这是直接使用Vue.component创建组件</h3>’

})

第三种

<div id=”app2”>

</div>

在被控制的#app外面使用template元素，定义组件的HTML结构

<template id=”app1”>

<div>

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

</div>

</template>

Vue.component(‘mycom3’,{

template:’#tmpl’,

})

**创建组件（私有）**

var vm = new Vue({

el:’#app’,

data:{},

methods:{},

filters:{},

directives:{},

components:{

//定义实例内部私有组件

login:{

template:’<h1>这是私有的login组件</h1>’

或： template:’temp2’

}

},

//生命周期函数

beforeCreate(){},

created(){},

beforeMount(){},

mounted(){},

beforeUpdate(){},

updated(){},

beforeDestroy(){},

destroyed(){}

})

使用组件(全局)

<div id=”app”>

<my-com1></my-com1> 使用的时候用-代替驼峰命名法,如果使用Vue.compoonent定义全局组件的时候，使用纯小写则直接小写调用

<mycom1></mycom1>

<mycom2></mycom2>

<mycom3></mycom3>

</div>

使用组件(私有)

<div id=”app”>

<login></login >

</div>

<template id=”tmp2”>

<h1>私有login组件</h1>

</template>

组件中的data 和methods

<div id=”app”>

<mycom1></mycom1>

</div>

//1.组件可以有自己的data数据

2.组件的data和实例的data不一样，实例中data为对象，但是组件中的data为方法

3.组件中 data除了必须为一个方法外，这个方法内部还必须返回一个对象

4.组件中的data数据使用方式和实例中的data使用方式完全一样

vue.component(‘mycom1’,{

template:’<h1>这是全局组件---{{msg}}</h1>’,

data : function(){

return {

msg:’这是组件中的data定义的数据’

}

}

})

var vm = new Vue({

el:’#app’,

data:{},

methods:{}

})

组件中的data为什么要这样设计呢（fun）？为什么不与实例设计的一样(对象)？

**组件中的data写成函数，数据以函数返回值的形式定义，这样每次复用一次组件就会返回新的data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例管理维护各自的数据，而如果用对象形式，就使得所有的组件实例公用了一个data，就会造成一个改变全部改变的结果**

组件切换 vue提供了 component 来展示对应名称的组件

component是一个占位符, :is属性可以用来指定要展示的组件的名称

<a href=”” @click=”comName=’login’”>登录</a>

<a href=”” @click=”comName=’register”>注册</a>

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

Vue.component(‘login’,{

template:’<h3>登录组件</h3>’

})

data:{

comName:’’ //当前显示的组件名称

}

组件切换动画

<style>

.v-enter,

.v-leave-to{

opacity:0;

transform:translateX(150px);

}

.v-anter-active,

.v-leave-active{

transition:all 0.5s ease;

}

</style>

<a href=”” @click=”comName=’login’”>登录</a>

<a href=”” @click=”comName=’register”>注册</a>

<transition mode=”out-in”> //还有in-out

<component :is=”comName”></component>

</transition>

data:{

comName:’’ //当前显示的组件名称

}

**组件传值**

**父组件向子组件传值**

<div id=”app”>

父组件，可以在引用组件的时候，通过属性绑定的形式，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>

var vm = new Vue({

el : ‘#ap’,

data:{

msg:’这是父组件中的数据’

}

components{

//子组件中，默认无法访问到父组件中的data上的数据和methods中的方法

com1:{

data(){//子组件中的data数据是子组件自己私有的，和父组件无关系

return {

title:’123’,

content:’qqq’

}

}

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

注意组件中的所有props中的数据，都是通过父组件传递给子组件的

props:[‘parentmsg’ ],

//注意这是一个数组,把父组件传递过来的parentmsg属性先在props定义一下这样才能使用这个数据

}

}

})

子组件中的data是可读可写的，props中的数据都是只读的无法重新赋值

父组件把方法传递给子组件

**子组件向父组件传值**

<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些方法来调用传递进去的这个方法了

<com2 v-on:func=”show”></com2>

</div>

<template id=”temp1”>

<div>

<h1>这是子组件</h1>

<input type=”button” value=”这是子组件的按钮，点击它触发父组件传递过来的func方法” @click=”myclick”>

</div>

</template>

//定义一个字面量组件模板

var com2 = {

template:’#tmp1’,

methods:{

myclick(){

this.$emit(‘func’,’123’) //传递参数到父组件的方法

}

}

}

var vm = new Vue({

el : ‘#ap’,

data:{

dataFormSon : null

},

methods:{

show(data){

this.dataFormSon = data

console.log(‘父组件show方法’+data)

}

}

components:{

com2

}

})

**兄弟组件传值**

可以用过一个vue实例Bus作为媒介，要相互通信的兄弟组件之中，都引入Bus，之后通过分别调用Bus事件触发$emit和监听$on来实现组件之间的通信和参数传递，类似window的全局自定义事件。类似与子传父，只不过是利用一个新的vue示例作为媒介，而不是当前vue示例(this)

// bus.js

import Vue from 'vue';

export default new Vue;

// a.js

<template>

<div class='a'></div>

</template>

<script>

import Bus from 'bus.js' ;

export default {

name: "a",

created() {

// 在需要的传递数据的时候调用sendData方法，这里模拟调用

this.sendData();

},

methods: {

sendData () {

Bus.$emit('listenToA', 'hello');

}

}

}

</script>

// b.js

<template>

<div class='b'></div>

</template>

<script>

import Bus from 'bus.js';

export default {

name: "b",

monted() {

Bus.$on('listenToA', this.getAData);

},

methods: {

getAData (val) {

console.log(`a组件传递过来的数据: ${val}`); // hello

}

}

}

</script>

**vue-router 路由**

<div id=”app”>

<a href=”#/login”>登录</a>

<a href=”#/register”>注册</a>

//这是vue-router提供的元素，专门用来当作占位符的，路由规则匹配到的组件就会展示到这个router-view中去

<router-view></router-view>

</div>

var login = {

template:’<h1>登录组件</h1>’

}

var register = {

template:’<h1>注册组件</h1>’

}

var routerObj = new VueRouter({

routes:[ //路由匹配规则,每个路由规则都是一个对象

path是路由链接地址

component是组件名称,不能是组件的引用名称

{path:’/login’,component:login}，

{path:’/register’,component:register}

]

})

var vm = new Vue({

el : ‘#ap’,

data:{

},

methods:{},

router: routerObj //将路由规则对象，注册到vm实例上，用来监听URL地址变化显示对应的组件

})

路由router-link的使用

//默认渲染为a标签 使用tag=”span”可渲染为span标签

<router-link to=”/login” tag=”span”>登录</router-link>

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

路由重定向

redirect

{path:’/’,redirect:’/login’}

路由高亮

router-link渲染后标签有个.router-link-active class给它设置高亮样式

.router-link-active{

color:red;

}

如果想要更改高亮默认的类名可以在new Vuerouter中添加属性linkActiveClass

linkActiveClass : ‘myactive’ //值是自定义类名

路由规则中定义参数（路由传参）

第一种

//在路径后面加?传参,&链接多个参数

//此方法无需修改路由匹配规则的path属性

<router-link to=”/login?id=10”></router-link>

组件获取传过来的参数

var login = {

template:’<h1>login---{{$route.query.id}}</h1>’,

created(){//组件生命周期函数

//这里打印this.会看到$route这个对象,里面的query就是存放了传递过来的参数

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

}

}

第二种

//传参/10

<router-link to=”/login/10”></router-link>

组件获取传过来的参数

var login = {

template:’<h1>login</h1>’,

created(){

console.log(this.$route.params.id) //注意这里不是query了是params

}

}

//路由规则加了/:id,（多个参数就再加/:name…）

routes:[

{path:’/login/:id’,component:login}

]

路由嵌套

<div id=”app”>

<router-view></router-view>

</div>

<template id=”#app1”>

<router-link to=”/account/login”></router-link>

<router-view></router-view>//在父组件中加router-view

</div>

var account = {

template:’#app1’,

}

var login = {

template:’<h1>login</h1>’,

}

routes:[

{

path:’/account’,

component: account,

//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

children:[

{path:’login’,component:login} //注意没有 /

]

}

]

var vm = new Vue({

el:’#app’

})

命名视图

<router-view></router-view>

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

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

var router = new VueRouter({

routes:[

{

path:’/’,

components:{

‘default’ : header,

‘left’ ：leftBox,

‘main’ : mainBox

}

}

]

})

命名视图是字符串值

**watch监听器**vm实例属性，可以监听data中指定是数据的变化，然后触发这个watch中对应的处理函数

<body>

    <div *id*="app">

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

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

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

    </div>

</body>

<script>

    const vm = **new** *Vue*({

        el: '#app',

        data: {

            firstname: '',

            lastname: '',

            fullname: ''

        },

        methods: {

        },

        watch: {

            firstname:function(*newVal*,*odlVal*){

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

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

            },

            lastname:function(*newVal*){

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

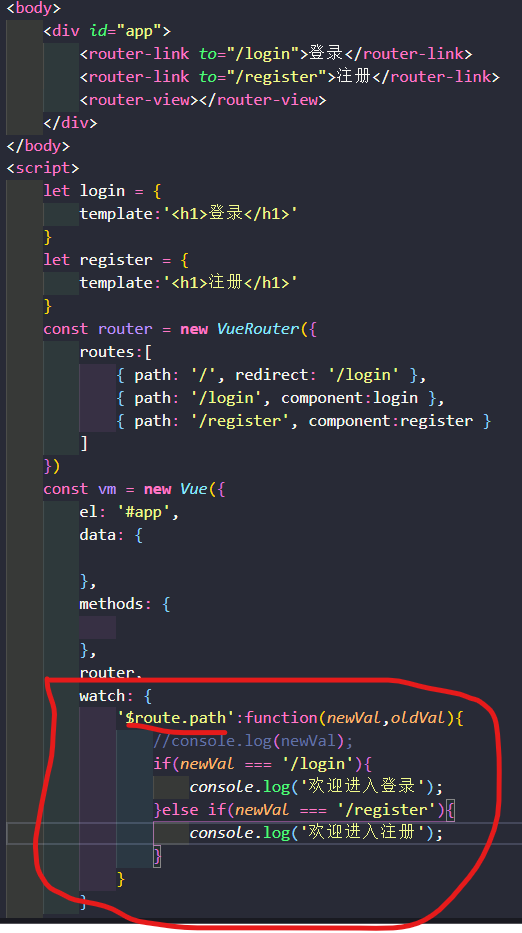
            }

        }

    })

</script>

watch监听路由的改变



**computed计算属性**

<div *id*="app">

        <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: '',

                fullname: ''

            },

            methods: {},

            computed: { //在computed中可以定义一些属性，这些属性叫做【计算属性】，

                //计算属性的本质就是一个方法，只不过在使用的时候，直接当作属性来使用

                //注意1，计算属性在引用的时候，一定不要加()去调用，直接把它当作普通属性区使用就好

                //注意2，只要计算属性这个function内部，所用到的任何data中的数据发生了变化，就会立即重新计算这个计算属性的值

                //注意3，计算属性的求值结果，会被缓存起来，方便下次直接使用；如果计算属性方法中，所有来的任何数据都没有发生变化则不会重新计算属性求值

                'fullname':function(){

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

                }

            },

        });

    </script>

render函数渲染组件

<body>

    <div *id*="app">

    </div>

    <script>

        var login = {

            template: '<h1>这是登录组件</h1>'

        }

        var vm = **new** *Vue*({

            el: '#app',

            data: {},

            methods: {},

            render(*createElements*){ //createElements是一个方法调用它，

            //能够把指定的组件模板渲染为html结构

                return *createElements*(login);

                //这里return的结果会替换页面中el指定的那个容器的innerHML

            }

        });

        /\*

            render和传统的方法唯一区别就是会替换原来容器中的内容，传统渲染的组件是在容器的内容中增加组件

        \*/

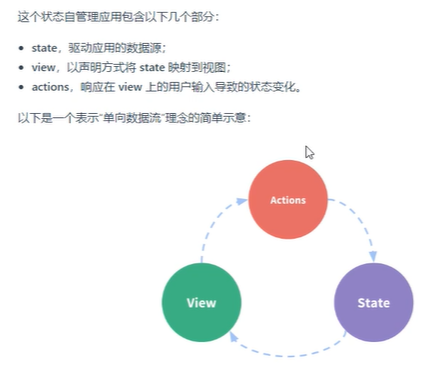
    </script>

</body>

**VueX状态管理模式**

**单向数据流的概念：数据触发视图的更改，视图跟用户交互触发事件动作更改数据这一整个的环形数据的流动就叫单向数据流。**

VueX是适用于在Vue项目开发时使用的状态管理工具。试想一下，如果在一个项目开发中频繁的使用组件传参的方式来同步data中的值，一旦项目变得很庞大，管理和维护这些值将是相当棘手的工作。为此，Vue为这些被多个组件频繁使用的值提供了一个统一管理的工具——VueX。



什么时候用VueX?

1. 很多组件的情况下
2. 很多的组件需要共享数据的情况下

安装VueX

npm install vux –save

<https://vuex.vuejs.org/zh/guide/state.html>

**state**

### 在 Vue 组件中获得 Vuex 状态

那么我们如何在 Vue 组件中展示状态呢？由于 Vuex 的状态存储是响应式的，从 store 实例中读取状态最简单的方法就是在[计算属性](https://cn.vuejs.org/guide/computed.html)中返回某个状态：

// 创建一个 Counter 组件

const Counter = {

template: `<div>{{ count }}</div>`,

computed: {

count () {

return store.state.count

}

}

}

### mapState 辅助函数 对象展开运算符

mapState 函数返回的是一个对象。我们如何将它与局部计算属性混合使用呢？通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性。但是自从有了[对象展开运算符](https://github.com/tc39/proposal-object-rest-spread)，我们可以极大地简化写法：

computed: {

localComputed () { /\* ... \*/ },

// 使用对象展开运算符将此对象混入到外部对象中

...mapState({

// ...

})

}

**getter**

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

Getter 接受 state 作为其第一个参数：

const store = new Vuex.Store({

state: {

todos: [

{ id: 1, text: '...', done: true },

{ id: 2, text: '...', done: false }

]

},

getters: {

doneTodos: state => {

return state.todos.filter(todo => todo.done)

}

}

})

### [#](https://vuex.vuejs.org/zh/guide/getters.html#%E9%80%9A%E8%BF%87%E5%B1%9E%E6%80%A7%E8%AE%BF%E9%97%AE)通过属性访问

Getter 会暴露为 store.getters 对象，你可以以属性的形式访问这些值：

store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]

Getter 也可以接受其他 getter 作为第二个参数：

getters: {

// ...

doneTodosCount: (state, getters) => {

return getters.doneTodos.length

}

}

store.getters.doneTodosCount // -> 1

我们可以很容易地在任何组件中使用它：

computed: {

doneTodosCount () {

return this.$store.getters.doneTodosCount

}

}

注意，getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的。

### [#](https://vuex.vuejs.org/zh/guide/getters.html#%E9%80%9A%E8%BF%87%E6%96%B9%E6%B3%95%E8%AE%BF%E9%97%AE)通过方法访问

你也可以通过让 getter 返回一个函数，来实现给 getter 传参。在你对 store 里的数组进行查询时非常有用。

getters: {

// ...

getTodoById: (state) => (id) => {

return state.todos.find(todo => todo.id === id)

}

}

store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }

注意，getter 在通过方法访问时，每次都会去进行调用，而不会缓存结果。

### [#](https://vuex.vuejs.org/zh/guide/getters.html#mapgetters-%E8%BE%85%E5%8A%A9%E5%87%BD%E6%95%B0)mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性：

import { mapGetters } from 'vuex'

export default {

// ...

computed: {

// 使用对象展开运算符将 getter 混入 computed 对象中

...mapGetters([

'doneTodosCount',

'anotherGetter',

// ...

])

}

}

如果你想将一个 getter 属性另取一个名字，使用对象形式：

...mapGetters({

// 把 `this.doneCount` 映射为 `this.$store.getters.doneTodosCount`

doneCount: 'doneTodosCount'

})

**mutation**

更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutation 非常类似于事件：每个 mutation 都有一个字符串的 **事件类型 (type)** 和 一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

const store = new Vuex.Store({

state: {

count: 1

},

mutations: {

increment (state) {

// 变更状态

state.count++

}

}

})

**使用常量替代 Mutation 事件类型**

使用常量替代 mutation 事件类型在各种 Flux 实现中是很常见的模式。这样可以使 linter 之类的工具发挥作用，同时把这些常量放在单独的文件中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

**// mutation-types.js**

**export const SOME\_MUTATION = 'SOME\_MUTATION'**

**// store.js**

**import Vuex from 'vuex'**

**import { SOME\_MUTATION } from './mutation-types'**

**const store = new Vuex.Store({**

**state: { ... },**

**mutations: {**

**// 我们可以使用 ES2015 风格的计算属性命名功能来使用一个常量作为函数名**

**[SOME\_MUTATION] (state) {**

**// mutate state**

**}**

**}**

**})**

用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。但如果你不喜欢，你完全可以不这样做。

### Mutation 必须是同步函数

**action**

Action 通过 store.dispatch 方法触发：

store.dispatch('increment')

**module 模块**

Vuex 允许我们将 store 分割成**模块（module）**。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块

默认情况下，模块内部的 action、mutation 和 getter 是注册在**全局命名空间**的——这样使得多个模块能够对同一 mutation 或 action 作出响应。

如果希望你的模块具有更高的封装度和复用性，你可以通过添加 namespaced: true 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名。