# Vue\_03

# 目标

* 能够定义vue自定义指令
* 使用计算属性和侦听器
* 使用混入完成方法添加
* 掌握生命周期
* 能够使用axios发起网络请求
* 定义组件实现父子组件间传值

# 一、自定义指令

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

除了核心功能默认内置的指令，Vue也允许注册自定义指令。有的情况下，对普通 DOM 元素进行底层操作，这时候就会用到自定义指令绑定到元素上执行相关操作。

自定义指令分为：全局指令和局部指令，当全局指令和局部指令同名时以局部指令为准。

自定义指令常用钩子函数

* bind 第一次绑定到元素时调用
* 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 数据更新时调用
* componentUpdated 指令所在组件的 VNode 及其子 VNode 全部更新后调用。
* unbind 只调用一次，指令与元素解绑时调用。

# 全局

Vue.directive('test',{

bind(el,bind){

console.log(el)

}

})

# 局部

new Vue({

directives: {

test:{

bind(el,bind){}

},

// bind/update

test2(el,bind){}

}

})

注：指令中不能读取Vue实例中的data数据



在指令中获取和操作data数据



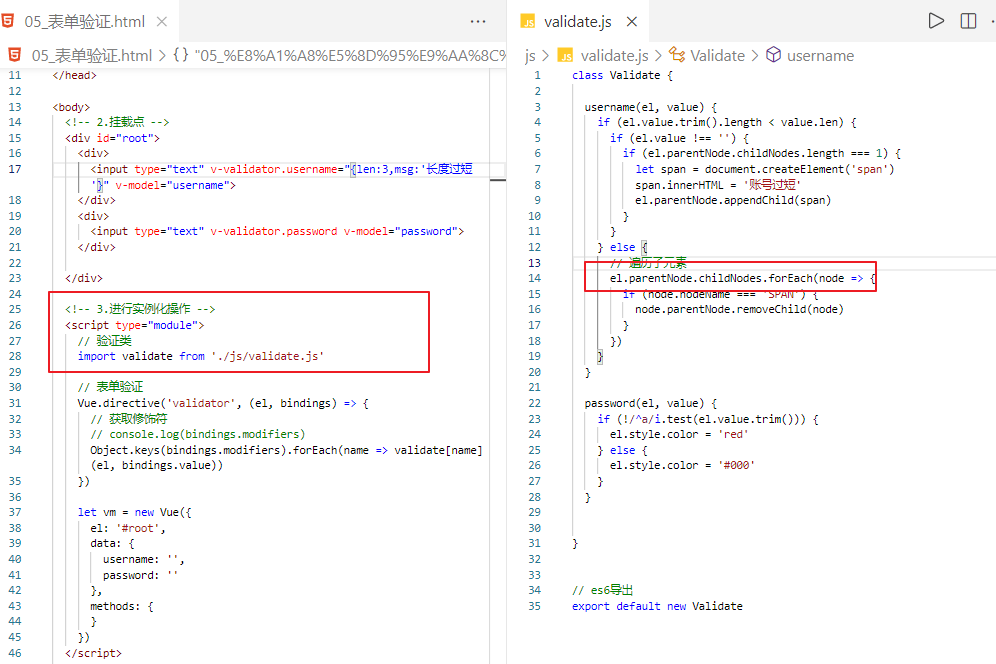
自定义修饰符



使用指令完成表单验证



表单验证封装



使用指令完成按钮级别权限



# 二、计算属性 - computed

模板中放入太多的逻辑会让模板过重且难以维护，使用计算属性可以让模板变得简洁易于维护。计算属性是基于它们的响应式依赖进行缓存的，计算属性比较适合对多个变量或者对象进行处理后返回一个结果值，也就是数多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化。

计算属性定义在Vue对象中，通过关键词computed属性对象中定义一个个函数，并返回一个值，使用计算属性时和data中的数据使用方式一致

<div id="app">

<!-- 当多次调用 cfn计算属性时只要里面的 num值不改变,它会把第一次计算的结果直接返回直到data中的num值改变 计算属性才会重新发生计算 -->

<div>{{ cfn }}</div>

<div>{{ cfn }}</div>

<!-- 调用methods中的方法的时候 他每次会重新调用 -->

<div>{{ fn() }}</div>

<div>{{ fn() }}</div>

</div>

<script type="text/javascript">

const vm = new Vue({

el: '#app',

data: {

num: 10

},

// 方法

methods: {

fn() {

console.log('methods')

return this.num

}

},

// 计算属性

computed: {

cfn() {

console.log('computed')

return this.num

}

}

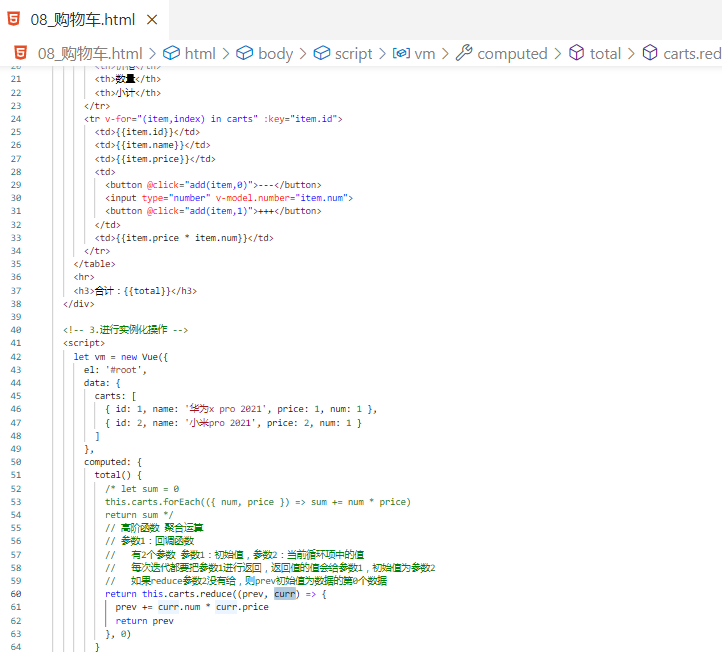
})

</script>

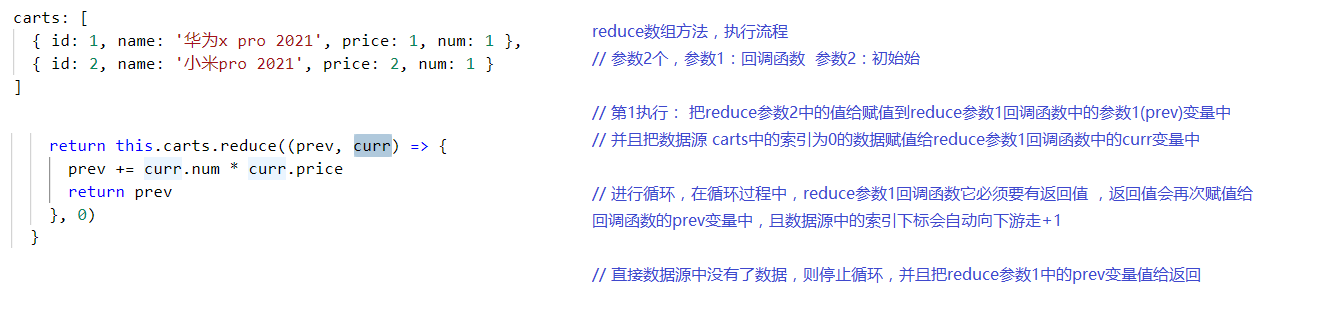
注：只要依赖的数据源不发生改变，我就调用1次，其它使用缓存



购物车



reduce



# 三、侦听器 - watch

使用watch来侦听data中数据的变化,watch中的属性一定是data 中已经存在的数据。

当需要监听一个对象的改变时，普通的watch方法无法监听到对象内部属性的改变，只有data中的数据才能够监听到变化，此时就需要deep属性对对象进行深度监听。

<div id="app">

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

</div>

<script>

const vm = new Vue({

el: '#app',

data: {

username: 'aaaa'

},

watch: {

username(newVal, oldVal) {

console.log(newVal, oldVal)

}

}

})

</script>

-----------------------------------------------------------

# 对象侦听

<div id="app">

<input type="text" v-model="obj.username">

</div>

<script type="text/javascript">

const vm = new Vue({

el: '#app',

data: {

obj: {

username: '张三'

}

},

watch: {

obj: {

handler(newVal, oldVal) {

console.log(newVal, oldVal)

},

// 深度侦听

deep: true

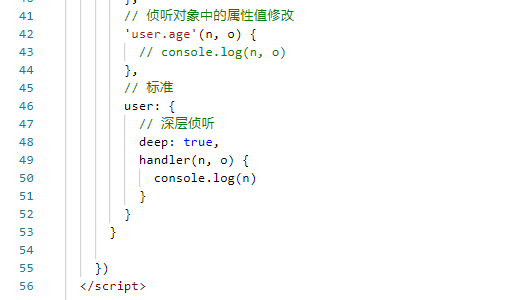
}

}

})

</script>





# 四、过滤器 - filters

在数据被渲染之前，可以对其进行进一步处理，比如将字符截取或者将小写统一转换为大写等等，过滤器本身就是一个方法。

过滤器可以定义全局或局部

# 全局

// 回调函数中的参数1永久是绑定的数据

Vue.filter(过滤器名称,(value,[…args])=>{})

# 局部过滤器定义

此过滤器定义在Vue对象的配置中的filters配置选项中，以方法形式来定义，参数1永久是绑定的数据

new Vue({

filters:{

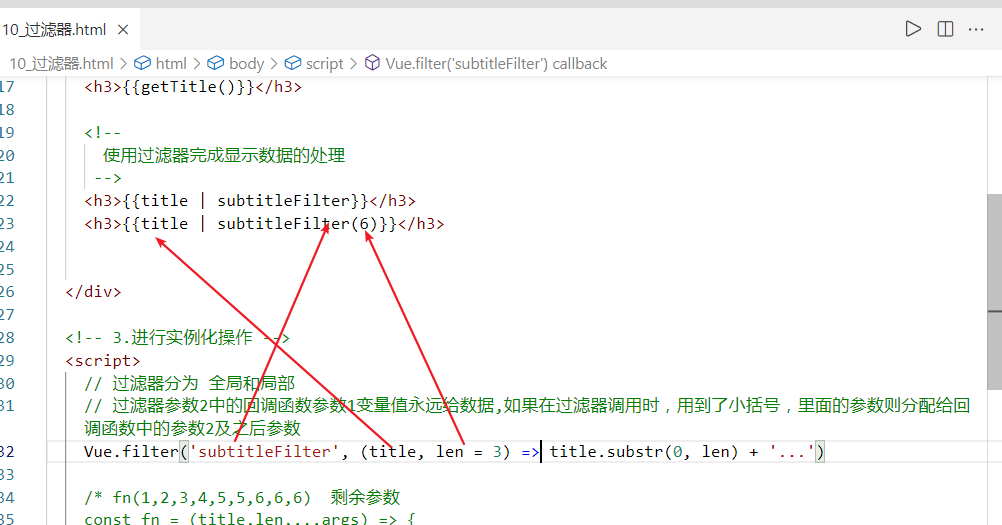
过滤器名(value){

return 数据

}

}

})



# 五、混入 - Mixins

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

混入分为：全局和局部

# 全局混入

Vue.mixin({

created(){},

methods:{}

})

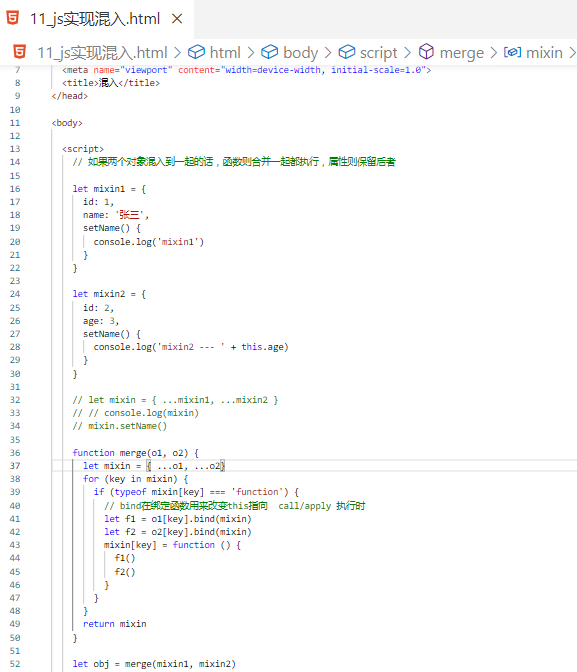
# 局部混入

let mixinObj = {}

new Vue({

mixins:[mixinObj]

})



# 六、实例生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如需要设置数据监听、编译模板、挂载实例到 DOM、在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，给予用户机会在一些特定的场景下添加他们自己的代码。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | beforeCreate | 在实例初始化之后，数据观测和事件配置之前被调用 此时data 和 methods 以及页面的DOM结构都没有初始化 什么都做不了 执行1次 | | created | 在实例创建完成后被立即调用此时data 和 methods已经可以使用 但是页面还没有渲染出来 执行1次 用this对象 | | beforeMount | 在挂载开始之前被调用 此时页面上还看不到真实数据 只是一个模板页面而已 执行1次 | | mounted | el被新创建的vm.$el替换，并挂载到实例上去之后调用该钩子。 数据已经真实渲染到页面上 在这个钩子函数里面可以进行数据请求等 执行1次 | | beforeUpdate | 数据更新时调用，页面上数据还是旧的 n次 | | updated | 由于数据更新完毕，页面上数据已经替换成最新的 n次 | | beforeDestroy | 实例销毁之前调用 执行1次 | | destroyed | 实例销毁后调用 执行1次 | | activated | keep-alive 组件激活时调用 | | deactivated | keep-alive 组件停用时调用 | | errorCaptured | 当捕获一个来自子孙组件的错误时被调用 | |

# 六、网络请求

## 6.1、XmlHttpRequest

var xmlhttp = new XMLHttpRequest()

xmlhttp.onreadystatechange = function () {

if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {

console.log(xmlHttp.responseText)

}

}

// GET

xmlhttp.open("GET", url, true)

xmlhttp.send(null)

// POST

// xhr.open('POST', 'url', true);

// xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded')

// var data = new FormData();

//data.append('user', 'person');

//xmlhttp.send(data)

## 6.2、fetch

Fetch是新的ajax解决方案 Fetch会返回Promise对象。fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。

参数说明

1、第一个参数是URL:

2、第二个设置请求的参数，是可选参数

3、返回使用了Promise 来处理结果/回调

fetch(url, options).then(res=>res.json()/text()).then(ret=>console.log(ret))

Fetch它给我们提供了很多的方法，如POST，GET，DELETE，UPDATE，PATCH和PUT等。默认的是 GET 请求，可以在设置请求参数中指定请求方法，通过method来设置

// get请求

// 通过url表达式来传递数据

fetch('http://xxx/?id=123')

.then(res => res.json())

.then(data => console.log(data))

// post标准提交

fetch('http://xxxx/post', {

method: 'post',

body: 'uname=lisi&pwd=123',

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

}

}).then(res => res.json()).then(data => console.log(data))

// post提交json数据

fetch('http://localhost:3000/books', {

method: 'post',

body: JSON.stringify({

uname: 'lisi',

pwd: '123'

}),

headers: {

'Content-Type': 'application/json'

}

}).then(res => res.json()).then(data => console.log(data))

## 6.3、axios

文档：<https://www.kancloud.cn/yunye/axios/234845>

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和node.js中。能拦截请求和响应，自动转换JSON数据。axios也是vue作者推荐使用的网络请求库。

**使用语法**

# GET

## 方法1

axios.get('/user?ID=12345').then(res=>{})

## 方法2

axios.get('/user', {

params: {

ID: 12345

}

}).then(res=>{})

# POST

axios.post('/user', {

firstName: 'aa',

lastName: 'bb'

}).then(res=>{})

# 标准写法

axios({

method: 'post',

url: '/user/12345',

timeout: 1000,

headers: {'X-Custom-Header': 'foobar'},

data: {

firstName: 'Fred',

lastName: 'Flintstone'

}

})

# axios 全局配置

# 配置公共的请求头

axios.defaults.baseURL = 'https://api.example.com';

# 配置 超时时间 毫秒数(0 表示无超时时间)

axios.defaults.timeout = 1000;

# 配置公共的请求头

axios.defaults.headers.token = 'token值';

# 七、vue组件

## 7.1、概述

组件 (Component) 是 Vue.js 最强大的功能之一，它是html、css、js等的一个聚合体。

组件化

* 将一个具备完整功能的项目的一部分分割多处使用
* 加快项目的进度
* 可以进行项目的复用

组件注册分为：全局注册和局部注册

## 7.2、全局注册

Vue.component('组件名称', { })第1个参数是标签名称，第2个参数是一个选项对象。全局组件注册后，任何vue实例都可以用。

**组件注意事项：**

* 构造 Vue 实例时传入的各种选项大多数都可以在组件里使用，只有一个例外：data必须是函数，同时这个函数要求返回一个对象
* 组件模板必须是单个根元素
* 组件模板的内容可以是模板字符串

Vue.component('HelloWorld', {

data: function(){

return {

msg: 'HelloWorld'

}

},

template: '<div>{{msg}}</div>'

});

## 7.3、局部注册

只能在当前注册它的vue实例中使用，通过某个 Vue 实例/组件的实例选项 components 注册仅在其作用域中可用的组件

var Child = {

template: '<div>A custom component!</div>'

}

new Vue({

components: {

// <my-component> 将只在父组件模板中可用

'my-component': Child

}

})

## 7.4、组件间传值

### 7.4.1、父组件向子组件传值

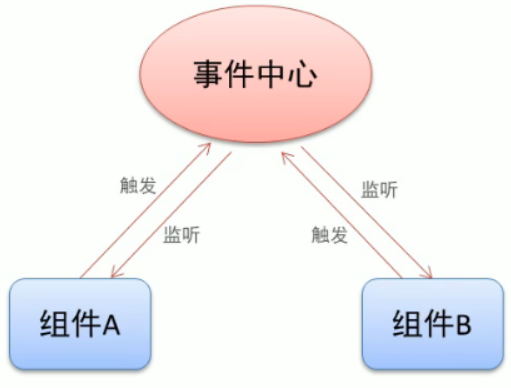
* 父组件以属性的形式绑定值到子组件身上
* 子组件通过使用属性props接收(props是单向绑定的(只读属性)：当父组件的属性变化时，将传导给子组件，但是反过来不会)

### 7.4.2、子组件向父组件传值

* 子组件用$emit()定义自定义事件，$emit()第一个参数为 自定义的事件名称 第二个参数为需要传递的数据
* 父组件用v-on(@)绑定子组件定义的自定义事件名，监听子组件的事件，实现通信

### 7.4.3、事件总线

非父子组件或更多层级间组件间传值，在Vue中通过单独的事件中心来管理组件间的传值。



* 建立统一的事件中心

const bus = new Vue()

* 传递数据方，通过一个事件触发bus.$emit(方法名，传递的数据)
* 接收数据方，在生命周期函数中，通过bus.$on(方法名,[params])来监听
* 销毁事件，在接受数据方，通过bus.$off(方法名)销毁之后无法监听数据

### 7.4.4、Ref

ref被用来给元素或子组件注册引用信息。引用信息将会注册在父组件的 $refs 对象上。如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素；如果用在子组件上，引用就指向组件实例

# 普通dom元素上

<p ref="p">hello</p>

# 子组件上

<child-comp ref="child"></child-comp>

# 调用

this.$refs.引用名

## 7.5、动态组件

通过使用保留的 <component> 元素，动态地绑定到它的 is 特性，我们让多个组件可以使用同一个挂载点，并动态切换

<component v-bind:is="currentView"></component>

<script>

var vm = new Vue({

el: '#root',

data: {

currentView: 'home'

},

components: {

home,

posts

}

})

</script>

* keep-alive

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。

<keep-alive>

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

</keep-alive>

<script>

var vm = new Vue({

el: '#root',

data: {

currentView: 'posts'

},

components: {

home,

posts

}

})

</script>

<!-- 子组件中 -->

<script>

export default {

activated: function () {

console.log("home组件被激活了");

},

deactivated: function () {

console.log("home组件被停用了");

},

};

</script>