**3.2 指令**

**1. 什么是指令？**

⚫ 指令的本质就是自定义属性

⚫ 指令的格式：以v-开始（比如：v-cloak）

**2. v-cloak指令用法**

⚫ 插值表达式存在的问题：“闪动”

⚫ 如何解决该问题：使用v-cloak指令

⚫ 解决该问题的原理：先隐藏，替换好值之后再显示最终的值

**3. 数据绑定指令**

⚫ v-text 填充纯文本

1. 相比插值表达式更加简洁

⚫ v-html 填充HTML片段

① 存在安全问题

② 本网站内部数据可以使用，来自第三方的数据不可以用

⚫ v-pre 填充原始信息

① 显示原始信息，跳过编译过程（分析编译过程）

        <div v-text='msg'></div>

        <div v-html='msg2'></div>

        <div v-pre>{{msg}}</div>

            data:{

                msg:'hello world',

                msg2:'<h1>qwer</h1>'

            }

**4. 数据响应式**

⚫ v-once 只编译一次

① 显示内容之后不再具有响应式功能

# 3. MVVM设计思想

① M(model)

② V(view)

③ VM(View-Model)

# 3.4 事件绑定

**1. Vue如何处理事件？**

⚫ v-on指令用法 <input type=‘button' v-on:click='num++'/>

⚫ v-on简写形式 <input type=‘button' @click='num++'/>

**2. 事件函数的调用方式**

⚫ 直接绑定函数名称 <button v-on:click='say'>Hello</button>

⚫ 调用函数 <button v-on:click='say()'>Say hi</button>

<input type="button" v-on:click='handle()' value="click2">

 var vm =new Vue({

            el:'#app',

            data:{

                msg:'hello world',

                num:0

            },

            methods:{

           handle:function(){

            console.log(this);

      //  在函数中 想要使用data里面的数据 一定要加this

           this.num++;

                }

            }

        })

**3. 事件函数参数传递**

⚫ 普通参数和事件对象

<button v-on:click='say("hi",$event)'>Say hi</button>

        <button v-on:click='handle1'>点击1</button>

        <button v-on:click='handle(123, 456, $event)'>点击2</button>

            methods:{

                handle:function(p1,p2,event){

                    console.log(p1,p2);

                    console.log(event.target.innerHTML);

                    this.num++;

                },

                handle1:function(event){

                    // 鼠标事件

                    console.log(event);

                    // button

                    console.log(event.target);

                    // 点击1

                    console.log(event.target.innerHTML);

                }

            }

**3.4 事件绑定**

**4. 事件修饰符**

⚫ .stop 阻止冒泡 <a v-on:click.stop="handle">跳转</a>

⚫ .prevent 阻止默认行为 <a v-on:click.prevent="handle">跳转</a>

传统做法：

      <div v-on:click='handle0'>

      <button v-on:click='handle1'>1111</button>

      <a href="http://www.baidu.com" v-on:click='handle2'>百度</a>

        </div>

            methods:{

                handle0:function(event){

                    this.num++;

                },

                handle1:function(event){

                    //阻止冒泡行为

                    event.stopPropagation();

                },

                handle2:function(event){

                    //阻止默认行为

                    event.preventDefault();

                }

        }

**Vue中做法**

<button v-on:click.stop='handle1'>1111</button>

<a href="http://www.baidu.com" v-on:click.prevent='handle2'>百度</a>

<!-- 修饰符可以串联 --> <a v-on:click.stop.prevent="doThat"></a>

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 v-on:click.prevent.self 会阻止****所有的点击****，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

更多修饰符信息查官方文档：https://cn.vuejs.org/v2/guide/events.html

**5. 按键修饰符**

⚫ .enter 回车键 <input v-on:keyup.enter='submit'>

⚫ .delete 删除键 <input v-on:keyup.delete='handle'>

            用户名：<input type="text" v-model='uname' v-on:keyup.delete='clearContent'>

            密码：<input type="text" v-model='upwd' v-on:keyup.enter='handle'>

 var vm = new Vue({

            el:'#app',

            data:{

                uname:'',

                upwd:''

            },

            methods:{

                clearContent:function(){

                    this.uname = ''

                },

                handle:function(){

                    console.log(this.uname);

                    console.log(this.upwd);

                }

            }

        })

**6. 自定义按键修饰符**

⚫ 全局 config.keyCodes 对象 Vue.config.keyCodes.f1 = 112

用法：

1先在script中定义

Vue.config.keyCodes.a= 65

2调用

<input type="text" v-on:keyup.a='handle' v-model='info'>

**案例：简单计算器**

# 3.5 属性绑定

**1. Vue如何动态处理属性？**

⚫ v-bind指令用法

⚫ 缩写形式

<a v-bind:href='url'>跳转</a>

<a :href='url'>跳转</a>

    <div id="app">

        <a v-bind:href="url" v-text='name' target="\_black"></a>

        <button v-on:click="handle">切换</button>

      </div>

        /\*

          属性绑定

        \*/

       var vm = new Vue({

           el:'#app',

           data:{

               url:'http://www.baidu.com',

               name:'百度'

           },

           methods:{

               handle:function(){

                   this.url = 'https://www.bilibili.com/',

                   this.name = '哔哩哔哩'

               }

           }

       })

**2. v-model的底层实现原理分析**

<input v-bind:value="msg" v-on:input="msg=$event.target.value">

<div id="app">

    <div>{{msg}}</div>

    <input type="text" v-bind:value="msg" v-on:input="handle">

    <input type="text" v-bind:value="msg" v-on:input='msg=$event.target.value'>

  </div>

  <script type="text/javascript" src="js/vue.js"></script>

  <script type="text/javascript">

    /\*

      v-model指令的本质

    \*/

    var vm = new Vue({

      el: '#app',

      data: {

        msg: 'hello'

      },

      methods: {

        handle: function(event){

          // 使用输入域中的最新的数据覆盖原来的数据

          this.msg = event.target.value;

        }

      }

    });

  </script>

# 3.6 样式绑定

**1. class样式处理**

⚫ 对象语法

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

   data: {

          isActive:true

      },

      methods: {

        handle: function(){

          // 控制isActive的值在true和false之间进行切换

          this.isActive = !this.isActive;

        }

      }

⚫ 数组语法

<div v-bind:class="[activeClass, errorClass]"></div>

<div v-bind:class='[activeClass, errorClass]'>测试样式</div>

      data: {

        activeClass: 'active',

        errorClass: 'error'

      },

      methods: {

        handle: function(){

          this.activeClass = '';

          this.errorClass = '';

        }

      }

**样式绑定相关语法细节：**

1对象绑定和数组绑定可以结合使用

<div v-bind:class='[activeClass, errorClass, {test: isTest}]'>

2class绑定的值可以简化操作

第一种

<div v-bind:class='arrClasses'></div>

data: {

arrClasses: ['active','error'],

}

第二种

<div v-bind:class='objClasses'></div>

data: {

objClasses: {

active: true,

error: true

}

}

3默认的class如何处理？默认的class会保留

<div class="base" v-bind:class='objClasses'></div>

**2. style样式处理**

⚫ 对象语法

<div v-bind:style="{ border: borderStyle, width: widthStyle }"></div>

data: {

borderStyle: '1px solid blue',

widthStyle: '100px',

},

⚫ 数组语法

<div v-bind:style="[baseStyles, overridingStyles]"></div>

data: {

baseStyles: {

border: '1px solid green',

width: '200px',

height: '100px'

},

overrideStyles: {

border: '5px solid orange',

backgroundColor: 'blue'

}

},

**3.7 分支循环结构**

**1. 分支结构**

⚫ v-if

⚫ v-else

⚫ v-else-if

    <div v-if='score>=90'>优秀</div>

    <div v-if='score>60&&score<90'>一般</div>

<div v-else>比较差</div>

⚫ v-show

    <div v-show='flag'>测试v-show</div>

      data: {

        score: 70,

        flag: false

      }

**2. v-if与v-show的区别**

⚫ v-if控制元素是否渲染到页面

⚫ v-show控制元素是否显示（已经渲染到了页面）

**3. 循环结构**

⚫ v-for遍历数组

     <li v-for='item in fruits'>{{item}}</li>

      <li v-for='(item, index) in fruits'>{{item + '---' + index}}</li>

 data: {

        fruits: ['apple', 'orange', 'banana'],

        myFruits: [{

          id: 1,

          ename: 'apple',

          cname: '苹果'

        }

⚫ key的作用：帮助Vue区分不同的元素，从而提高性能

<li :key='item.id' v-for='(item,index) in list'>{{item}} + '---' {{index}}</li>

<li v-bind:key='item.id' v-for='(item,index) in myFruits'>

<span>{{item.ename}}</span></li>

**4. 循环结构**

⚫ v-for遍历对象

<div v-for='(value, key, index) in object'></div>

⚫ v-if和v-for结合使用

<div v-if='value==12' v-for='(value, key, index) in object'></div>

5.2 表单操作

1. 基于Vue的表单操作

⚫ Input 单行文本

⚫ textarea 多行文本

⚫ select 下拉多选

⚫ radio 单选框

⚫ checkbox 多选框

<input type="checkbox" id="ball" value="1" v-model='hobby'>

        <label for="ball">篮球</label>

3. 表单域修饰符

⚫ number：转化为数值

<input type="text" v-model.number='age'>

console.log(this.age+2);

⚫ trim：去掉开始和结尾的空格

<input type="text" v-model.trim='content'>

⚫ : 将input事件切换为change事件

<input type="text" v-model.lazy='msg'>

# 5.3 自定义指令

1. 为何需要自定义指令？

内置指令不满足需求

1. 自定义指令的语法规则（获取元素焦点）

Vue.directive('focus' ,{

inserted: function(el) {

// 获取元素的焦点

el.focus();

}

})

3. 自定义指令用法

<input type="text" v-focus>

1. 带参数的自定义指令（改变元素背景色）

Vue.directive(‘color', {

inserted: function(el, binding) {

el.style.backgroundColor = binding.value.color;

}

})

4. 指令的用法

<input type="text" v-color='{color:"orange"}'>

5. 局部指令

var vm = new Vue({

      el: '#app',

      data: {

        msg:{

          color:'pink'

        }

      },

      methods: {

      },

      directives:{

        color:{

          bind:function(el,binding){

            el.style.backgroundColor = binding.value.color

          }

        },

        focus:{

          inserted:function(el){

            el.focus();

          }

        }

      },

    });

5.4 计算属性

1. 为何需要计算属性？

表达式的计算逻辑可能会比较复杂，使用计算属性可以使模板内容更加简洁

2. 计算属性的用法

 var vm = new Vue({

      el: '#app',

      data: {

        msg: 'Nihao'

      },

      computed:{

          reversedMessage:function(){

              return this.msg.split('').reverse().join('');

          }

      }

    });

 <div>{{reversedMessage}}</div>

**3. 计算属性与方法的区别**

⚫ 计算属性是基于它们的依赖进行缓存的

⚫ 方法不存在缓存

**5.5 侦听器**

**1. 侦听器的应用场景**

数据变化时执行异步或开销较大的操作

**2. 侦听器的用法**

**侦听器案例：验证用户名是否可用**

<body>

  <div id="app">

    <div>

      <span>用户名：</span>

      <span>

        <!-- lazy 作用是将input事件变成change事件，失去焦点后才触发，不然的话，这个input输入框输入一次，就要触发一次，因为监听器监听的是uname所以就要执行一次监听操作 -->

        <input type="text" v-model.lazy='uname'>

      </span>

      <span v-text='tips'></span>

    </div>

  </div>

  <script type="text/javascript" src="js/vue.js"></script>

  <script type="text/javascript">

    /\*

      侦听器

      1、采用侦听器监听用户名的变化

      2、调用后台接口进行验证

      3、根据验证的结果调整提示信息

    \*/

    var vm = new Vue({

      el: '#app',

      data: {

        uname:'',

        tips:''

      },

      methods:{

          checkName:function(uname){

            var that = this;//定时器内部的this指向的是Windows

            setTimeout(() => {

                  if(uname == 'admin'){

                      that.tips = '已经有人注册了'

                  }else{

                      that.tips = '可以注册'

                  }

              }, 2000);

          }

      },

      watch:{

        //监听的对象发生改变时，触发监听函数

          uname:function(val){

              this.checkName(val);

              this.tips='正在验证...'

              console.log(val);

              //val表示变化后的值

          }

      }

    });

  </script>

**5.6 过滤器**

**1. 过滤器的作用是什么？**

格式化数据，比如将字符串格式化为首字母大写，将日期格式化为指定的格式等

**2. 自定义过滤器**

Vue.filter(‘过滤器名称’, function(value){

// 过滤器业务逻辑

})     Vue.filter('upper',function(val){

        return val.charAt(0).toUpperCase() + val.slice(1)

    })

**3. 过滤器的使用**

<div>{{msg | upper}}</div>

<div>{{msg | upper | lower}}</div>

<div v-bind:id=“id | upper "></div>

**4. 局部过滤器**

      filters: {

        upper: function(val) {

          return val.charAt(0).toUpperCase() + val.slice(1);

        }

**5. 带参数的过滤器**

Vue.filter(‘format’, function(value, arg1){

// value就是过滤器传递过来的参数

})

**6. 过滤器的使用**

<div>{{date | format(‘yyyy-MM-dd')}}</div>

**5.7 生命周期**

**1. 主要阶段**

**⚫ 挂载（初始化相关属性）**

① beforeCreate

② created

③ beforeMount

④ mounted

⚫ 更新（元素或组件的变更操作）

① beforeUpdate

② updated

⚫ 销毁（销毁相关属性）

① beforeDestroy

② destroyed

**2. Vue实例的产生过程**

① beforeCreate 在实例初始化之后，数据观测和事件配置之前被调用。

② created 在实例创建完成后被立即调用。

③ beforeMount 在挂载开始之前被调用。

④ 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。

⑤ beforeUpdate 数据更新时调用，发生在虚拟DOM打补丁之前。

⑥ updated 由于数据更改导致的虚拟DOM重新渲染和打补丁，在这之后会调用该钩子。

⑦ beforeDestroy 实例销毁之前调用。

⑧ destroyed 实例销毁后调用。

案例：补充知识（数组相关API）

1. 变异方法(修改原有数据)

⚫ push()

⚫ pop()

⚫ shift()

⚫ unshift()

⚫ splice()

⚫ sort()

⚫ reverse()

2. 替换数组(生成新的数组)

⚫ filter()

⚫ concat()

⚫ slice()

**3. 修改响应式数据**

⚫ Vue.set(vm.items, indexOfItem, newValue)

⚫ vm.$set(vm.items, indexOfItem, newValue)

① 参数一表示要处理的数组名称

② 参数二表示要处理的数组的索引

③ 参数三表示要处理的数组的值

data: {

list: ['apple', 'orange', 'banana'],

info: {

name: 'lisi',

age: 12

}

}

vm.$set(vm.list, 1, 'lemon');

vm.$set(vm.info, 'gender', 'female');

**1.1 现实中的组件化思想体现**

⚫ 标准

⚫ 分治

⚫ 重用

⚫ 组合

Web Components 通过创建封装好功能的**定制元素**解决上述问题

**2.1 全局组件注册语法**

Vue.component(组件名称, {

data: 组件数据,

template: 组件模板内容

})

注册：

    Vue.component('button-count',{

        data:function(){

            return{count:0}

        },

        template:'<button v-on:click="handle">点击{{count}}</button>',

        methods:{

            handle:function(){

                this.count++

            }

        }

    })

    var vm = new Vue({

        el:'#app',

        data:{

        }

    })

使用：

   <div id="app">

        <button-count></button-count>

        <button-count></button-count>

    </div>

**2.3 组件注册注意事项**

1. data必须是一个函数

⚫ 分析函数与普通对象的对比

1. 组件模板内容必须是单个根元素

template: '<div><button @click="handle">点击了{{count}}次</button><button>测试</button></div>',

⚫ 分析演示实际的效果

3. 组件模板内容可以是模板字符串

⚫ 模板字符串需要浏览器提供支持（ES6语法）

**4. 组件命名方式**

⚫ 短横线方式

Vue.component('my-component', { /\* ... \*/ })

⚫ 驼峰方式

Vue.component('MyComponent', { /\* ... \*/ })

   <div id="app">

        <button-count></button-count>

        <button-count></button-count>

        <SayHello></SayHello>

    </div>

  template:`

        <div>

            <button v-on:click="handle">点击{{count}}</button>

            <SayHello></SayHello>

        </div>`,

**2.4 局部组件注册**

外部声明（格式）

    var HelloTom = {

      data: function(){

        return {

          msg: 'HelloTom'

        }

      },

      template: '<div>{{msg}}</div>'

    };

内部调用 （格式）

   components: {

        'hello-world': HelloWorld,

        'hello-tom': HelloTom,

      }

**3.1 调试工具安装**

具体教程看浏览器收藏栏

两种方法1谷歌插件直接安装

2自己下载程序，解析配置安装

**4.1 父组件向子组件传值**

1. 组件内部通过props接收传递过来的值

    /\*

      父组件向子组件传值-基本使用

    \*/

    Vue.component('menu-item',{

        props:[titleName],

        data:function(){

            return{

                msg:'this is a message'

            }

        },

        template:'<div>{{msg +"--"+titleName}}</div>'

    })

2. 父组件通过属性将值传递给子组件

      <menu-item title-name="1234"></menu-item>

      <menu-item :title-name="name"></menu-item>

var vm = new Vue({

      el: '#app',

      data: {

          name:'kity'

      }

    });

1. props属性名规则

⚫ 在props中使用驼峰形式，模板中需要使用短横线的形式

⚫ 字符串形式的模板中没有这个限制

Vue.component(‘menu-item', {

// 在 JavaScript 中是驼峰式的

props: [**‘menuTitle'**],

template: '<div>{{ **menuTitle** }}</div>'

})

<!– 在html中是短横线方式的 -->

<menu-item **menu-title**=“nihao"></menu-item>

**4.1 父组件向子组件传值**

⚫ 字符串 String

⚫ 数值 Number

⚫ 布尔值 Boolean

⚫ 数组 Array

⚫ 对象 Object

 <menu-item :pstr='pstr' :pnum='12' pboo='true' :parrr='parr' :pobj='pobj'></menu-item>

    Vue.component('menu-item', {

      props: ['pstr','pnum','pboo','parrr','pobj'],

      template: `

        <div>

          <div>{{pstr}}</div>

          <div>{{12 + pnum}}</div>

          <div>{{typeof pboo}}</div>

          <ul>

            <li :key='index' v-for='(item,index) in parrr'>{{item}}</li>

          </ul>

            <span>{{pobj.name}}</span>

            <span>{{pobj.age}}</span>

          </div>

        </div>

      `

    });

 data: {

        pmsg: '父组件中内容',

        pstr: 'hello',

        parr: ['apple','orange','banana'],

        pobj: {

          name: 'lisi',

          age: 12

        }

**4.2 子组件向父组件传值**

# **[vue中 关于$emit的用法](https://www.cnblogs.com/darkbluelove/p/11338432.html)**

1、父组件可以使用 props 把数据传给子组件。  
2、子组件可以使用 $emit 触发父组件的自定义事件。

vm.$emit( event, arg ) //触发当前实例上的事件

vm.$on( event, fn );//监听event事件后运行 fn

1. 子组件通过自定义事件向父组件传递信息

<button v-on:click='$emit("enlarge-text") '>扩大字体</button>

Vue.component('menu-font',{

        props:['size'],

        template:`

        <div>

          <button v-on:click='$emit("enlarge-text")'>点下就变大</button>

        </div>

        `

    })

2. 父组件监听子组件的事件

<menu-item v-on:enlarge-text='fontSize += 0.1'></menu-item>

<div id="app">

      <div :style='{fontSize:fontsize+"px"} '>{{content}}</div>

      <menu-font v-on:enlarge-text='handle'></menu-font>

  </div>

3. 子组件通过自定义事件向父组件传递信息

<button v-on:click='$emit("enlarge-text", 0.1) '>扩大字体</button>

     template: `

        <div>

          <button @click='$emit("enlarge-text", 5)'>扩大父组件中字体大小</button>

          <button @click='$emit("enlarge-text", 10)'>扩大父组件中字体大小</button>

        </div>

      `

      methods: {

        handle: function(val){

          // 扩大字体大小

          this.fontSize += val;

        }

      }

4. 父组件监听子组件的事件 （传参）

<menu-item v-on:enlarge-text='fontSize += $event'></menu-item>

    <div :style='{fontSize: fontSize + "px"}'>{{pmsg}}</div>

    <menu-item  @enlarge-text='handle($event)'></menu-item>

在原生事件中，$event是事件对象

在自定义事件中，$event是传递过来的数据

**vm.$emit( eventName, […args] )**

**参数：**

**{string} eventName**

**[...args]**

    <menu-item  @enlarge-text='handle($event)'></menu-item>

      template: `

        <div>

          <button @click='$emit("enlarge-text", [5,15])'>扩大父组件中字体大小</button>

          <button @click='$emit("enlarge-text", [10,-5])'>扩大父组件中字体大小</button>

        </div>

      `

      methods: {

        handle: function(val){

          console.log(val);

          var reval = val[1];

          console.log(reval);

          // 扩大字体大小

          this.fontSize += reval;

        }

**4.3 非父子组件间传值**

1. 单独的事件中心管理组件间的通信

在一个工具文件中创建 事件中心

export const eventBus = new Vue()

2. 监听事件与销毁事件

eventHub.$on('add-todo', addTodo)

eventHub.$off('add-todo')

3. 触发事件

eventHub.$emit(‘add-todo', id)

**a组件中引入**

**import { eventBus } from '../main'**

**methods: {**

**messageSister() {**

**eventBus.$emit('brotherSaid', 'hhhh')**

**}**

**}**

**b组件中引入**

**created() {**

**eventBus.$on('brotherSaid', (message) => {**

**this.fromBrother = message**

**})**

**}**

**5.1 组件插槽的作用**

⚫ 父组件向子组件传递内容

**5.2 组件插槽基本用法**

1. 插槽位置

Vue.component('alert-box',{

        template:`

        <div>

          <strong>Tips:</strong>

          <slot>这里是默认内容</slot>

        </div>

        `

    })

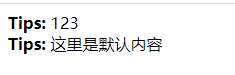
2. 插槽内容

  <div id="app">

      <alert-box>123</alert-box>

      <alert-box></alert-box>

  </div>



**5.3 具名插槽用法**

1. 插槽定义

    Vue.component('base-layout', {

      template: `

        <div>

          <header>

            <slot name='header'></slot>

          </header>

          <main>

            <slot></slot>

          </main>

          <footer>

            <slot name='footer'></slot>

          </footer>

        </div>`

    });

1. 插槽内容

  <base-layout>

      <p slot='header'>标题信息</p>

      <p>主要内容1</p>

      <p>主要内容2</p>

      <p slot='footer'>底部信息信息</p>

    </base-layout>

还有另外一种用法

    <base-layout>

      <template slot='header'>

        <p>标题信息1</p>

        <p>标题信息2</p>

      </template>

      <p>主要内容1</p>

      <p>主要内容2</p>

      <template slot='footer'>

        <p>底部信息信息1</p>

        <p>底部信息信息2</p>

      </template>

    </base-layout>

**5.4 作用域插槽**

⚫ 应用场景：**父组件对子组件的内容进行加工处理**

1. 插槽定义

  /\*

      作用域插槽

    \*/

    Vue.component('fruit-list',{

        props:['listdata'],

        template:`

        <div>

            <li v-bind:key='item.id' v-for='item in listdata'>

            <slot :info='item'>{{item.name}}</slot>

            </li>

        </div>

        `

    })

2. 插槽内容

  <div id="app">

      <fruit-list :listdata='list'>

          <template slot-scope="slotProps">

//id为2的那条li颜色变亮

              <strong v-if='slotProps.info.id ==2' class="current">{{slotProps.info.name}}</strong>

          </template>

      </fruit-list>

  </div>

在作用域插槽内，父组件可以拿到子组件的数据。子组件可以在slot标签上绑定属性值，如：

<slot :nickName="'Tusi'"></slot>

而父组件通过slot-scope绑定的对象下拿到nickName的值。

<template>

<section>

<slot-child>

<template slot-scope="scope">

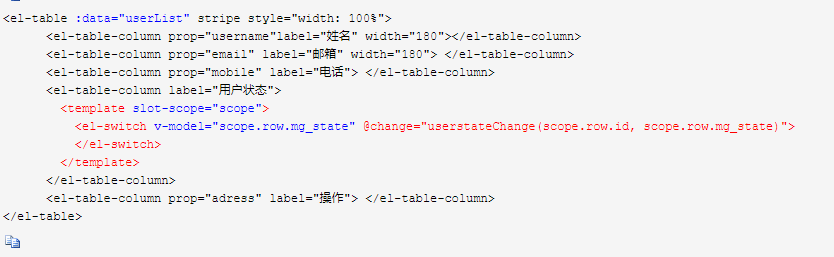
<div>{{scope.nickName}}</div>

</template>

</slot-child>

</section>

</template>



在这里使用ElementUI表格模板渲染数据时,

"当前行数据的获取也会用到插槽,scope相当于一行的数据， scope.row相当于当前行的数据对象,"

这里拿到每一行的index----------scope.$index

这里拿到每一行的数据-----------scope.$row



**Vue在2.6之后更新了插槽的用法，使用案例可以查看 Vue day3 插槽2.6之后用法案例.html**

**11作用域新插槽.html 和原来11作用域插槽.html 可以比对**

**1.路由的基本概念与原理**

**1.1 路由**

路由是一个比较广义和抽象的概念，路由的本质就是对应关系。

在开发中，路由分为：

 后端路由

 前端路由

**1. 后端路由**

 概念：根据不同的用户 URL 请求，返回不同的内容

 本质：URL 请求地址与服务器资源之间的对应关系

**2. SPA（Single Page Application）**

 后端渲染（存在性能问题）

 Ajax前端渲染（前端渲染提高性能，但是不支持浏览器的前进后退操作）

 SPA（Single Page Application）单页面应用程序：整个网站只有一个页面，内

容的变化通过Ajax局部更新实现、同时支持浏览器地址栏的前进和后退操作

 SPA实现原理之一：基于URL地址的hash（hash的变化会导致浏览器记录访问历

史的变化、但是hash的变化不会触发新的URL请求）

 在实现SPA过程中，最核心的技术点就是前端路由

Hash模式和history模式：

**hash：即地址栏URL中的#符号**

**history-利用了HTML5 History Interface 中新增的pushState() 和replaceState() 方法**。pushState()不会触发页面刷新，只是导致history对象发生变化，地址栏会有反应。我们在开发的时候默认使用hash模式(其url后永远带着#号)，那

**1、为什么要使用history模式？**

history模式没有#号，看起来更规范更正常，适合推广宣传

开发app的适合会有分享页面，将这个页面分享到第三方的app里时，有的app不允许url带有#符号

但是，

**2、history模式也存在一个问题**

在访问二级页面的时候，做刷新操作，会出现404错误，此时需要后端人员配置一下代理服务器(apache或者nginx)的url重定向，重定向到你的首页路由上。history模式下，前端的url必须和后端发起请求的url一致，如<http://www.baidu.com/a/> ，如果后端缺少对/a的处理，将返回404错误。

（因为你刷新的话，url还是当前的url，相当于地址栏输入按回车，所以会发送一次get请求，而服务器端是没有之前那个url的路径的，因为那个是前端路由，在服务器那就找不到了）

3、hash模式不会返回404错误，为什么？

hash模式下，仅hash符号之前的内容会被包含在请求中，因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回404错误。即hash虽然出现在URL中，但不会被包含在HTTP请求中，对后端完全没有影响，因此改变hash不会重新加载页面。

**2. 前端路由**

 概念：根据不同的用户事件，显示不同的页面内容

 本质：用户事件与事件处理函数之间的对应关系

**3. 实现简易前端路由**

 基于URL中的hash实现（点击菜单的时候改变URL的hash，根据hash的变化控制组件的切换）

        <!-- 根据 :is 属性指定的组件名称，把对应的组件渲染到 component 标签所在的位置 -->

        <!-- 可以把 component 标签当做是【组件的占位符】 -->

        <!-- component渲染一个“元组件”为动态组件。依 is 的值，来决定哪个组件被渲染。 -->

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

**1.2 Vue Router**

**Vue Router**（官网：https://router.vuejs.org/zh/）是 Vue.js 官方的**路由管理器**。

它和 Vue.js 的核心深度集成，可以非常方便的用于SPA应用程序的开发。

Vue Router 包含的功能有：

 支持HTML5 历史模式或 hash 模式

 支持嵌套路由

 支持路由参数

 支持编程式路由

 支持命名路由

**2.1 基本使用步骤**

1. 引入相关的库文件

2. 添加路由链接

3. 添加路由填充位

4. 定义路由组件

5. 配置路由规则并创建路由实例

6. 把路由挂载到 Vue 根实例中

**1. 引入相关的库文件**

<!-- 导入 vue 文件，为全局 window 对象挂载 Vue 构造函数 -->

<script src="./lib/vue\_2.5.22.js"></script>

<!-- 导入 vue-router 文件，为全局 window 对象挂载 VueRouter 构造函数 -->

<script src="./lib/vue-router\_3.0.2.js"></script>

**2. 添加路由链接**

<!-- router-link 是 vue 中提供的标签，默认会被渲染为 a 标签 -->

<!-- to 属性默认会被渲染为 href 属性 -->

<!-- to 属性的值默认会被渲染为 # 开头的 hash 地址 -->

<router-link to="/user">User</router-link>

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

**3. 添加路由填充位**

<!-- 路由填充位（也叫做路由占位符） -->

<!-- 将来通过路由规则匹配到的组件，将会被渲染到 router-view 所在的位置 -->

<router-view></router-view>

**4. 定义路由组件**

        //定义路由组件

        var User = {

            template:'<h1>this user 组件</h1>'

        }

        var Home = {

            template:'<h1>this home 组件</h1>'

        }

**5. 配置路由规则并创建路由实例**

// 创建路由实例对象

var router = new VueRouter({

// routes 是路由规则数组

routes: [

// 每个路由规则都是一个配置对象，其中至少包含 path 和 component 两个属性：

// path 表示当前路由规则匹配的 hash 地址

// component 表示当前路由规则对应要展示的组件

{path:'/user',component: User},

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

]

})

**6. 把路由挂载到 Vue 根实例中**

        var vm = new Vue({

            el:'#app',

            data:{},

            // 为了能够让路由规则生效，必须把路由对象挂载到 vue 实例对象上

            router:router

        })

**2.2 路由重定向**

路由重定向指的是：用户在访问地址 A 的时候，强制用户跳转到地址 C ，从而展示特定的组件页面；

通过路由规则的 redirect 属性，指定一个新的路由地址，可以很方便地设置路由的重定向：

var router = new VueRouter({

routes: [

// 其中，path 表示需要被重定向的原地址，redirect 表示将要被重定向到的新地址

{path:'/', redirect: '/user'},

{path:'/user',component: User},

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

]

})

**3.1 嵌套路由用法**

**1. 嵌套路由功能分析**

 点击父级路由链接显示模板内容

 模板内容中又有子级路由链接

 点击子级路由链接显示子级模板内容

**2. 父路由组件模板**

 父级路由链接

 父组件路由填充位

**3. 子级路由模板**

 子级路由链接

 子级路由填充位

const Register = {

template: `<div>

<h1>Register 组件</h1>

<hr/>

<router-link to="/register/tab1">Tab1</router-link>

<router-link to="/register/tab2">Tab2</router-link>

<!-- 子路由填充位置 -->

<router-view></router-view>

</div>`

}

**4. 嵌套路由配置**

 父级路由通过children属性配置子级路由

const router = new VueRouter({

routes: [

{ path: '/user', component: User },

{

path: '/register',

component: Register,

// 通过 children 属性，为 /register 添加子路由规则

children: [

{ path: '/register/tab1', component: Tab1 },

{ path: '/register/tab2', component: Tab2 }

]

}

]

})

**4.1 动态匹配路由的基本用法**

**应用场景：通过动态路由参数的模式进行路由匹配**

var router = new VueRouter({

routes: [

// 动态路径参数 以冒号开头

{ path: '/user/:id', component: User }

]

})

const User = {

// 路由组件中通过$route.params获取路由参数

template: '<div>User {{ $route.params.id }}</div>'

}

**4.2 路由组件传递参数**

$route与对应路由形成高度耦合，不够灵活，所以可以使用props将组件和路由解耦

**1. props的值为布尔类型**

const router = new VueRouter({

routes: [

// 如果 props 被设置为 true，route.params 将会被设置为组件属性

{ path: '/user/:id', component: User, props: true }

]

})

const User = {

props: ['id'], // 使用 props 接收路由参数

template: '<div>用户ID：{{ id }}</div>' // 使用路由参数

}

**2. props的值为对象类型**

const router = new VueRouter({

routes: [

// 如果 props 是一个对象，它会被按原样设置为组件属性

{ path: '/user/:id', component: User, props: { uname: 'lisi', age: 12 }}

]

})

const User = {

props:['id','uname','age'],

props: ['uname', 'age'],

template: ‘<div>用户信息：{{ uname + '---' + age}}</div>'

}

**3. props的值为函数类型**

const router = new VueRouter({

routes: [

// 如果 props 是一个函数，则这个函数接收 route 对象为自己的形参

{ path: '/user/:id',

component: User,

props: (route) => ({ uname: 'zs', age: 20, id: route.params.id })}

]

})

const User = {

props: ['uname', 'age', 'id'],

template: ‘<div>用户信息：{{ uname + '---' + age + '---' + id}}</div>'

}

**6.1 命名路由的配置规则**

为了更加方便的表示路由的路径，可以给路由规则起一个别名，即为“命名路由”。

const router = new VueRouter({

routes: [

{

path: '/user/:id',

name: 'user',

component: User

}

]

})

<router-link :to="{ name:'user',params: { id: 123 }}">User</router-link>

router.push({ name: 'user', params: { id: 123 }})

**5.1 页面导航的两种方式**

 声明式导航：通过点击链接实现导航的方式，叫做声明式导航

例如：普通网页中的 <a></a> 链接 或 vue 中的 <router-link></router-link>

 编程式导航：通过调用JavaScript形式的API实现导航的方式，叫做编程式导航

例如：普通网页中的 location.href

**5.1 编程式导航基本用法**

常用的编程式导航 API 如下：

 this.$router.push('hash地址')

 this.$router.go(n)

const User = {

template: '<div><button @click="goRegister">跳转到注册页面</button></div>',

methods: {

goRegister: function(){

// 用编程的方式控制路由跳转

this.$router.push('/register');

}

}

}

**5.2 编程式导航参数规则**

**router.push() 方法的参数规则**

// 字符串(路径名称)

router.push('/home')

// 对象

router.push({ path: '/home' })

// 命名的路由(传递参数)

router.push({ name: '/user', params: { userId: 123 }})

// 带查询参数，变成 /register?uname=lisi

router.push({ path: '/register', query: { uname: 'lisi' }})