**VUE.JS**

# 简介

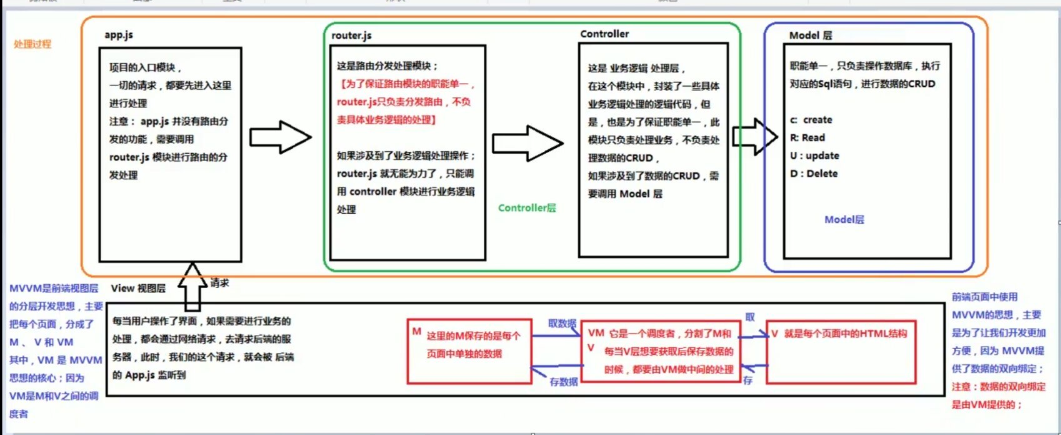
## 什么是Vue.js

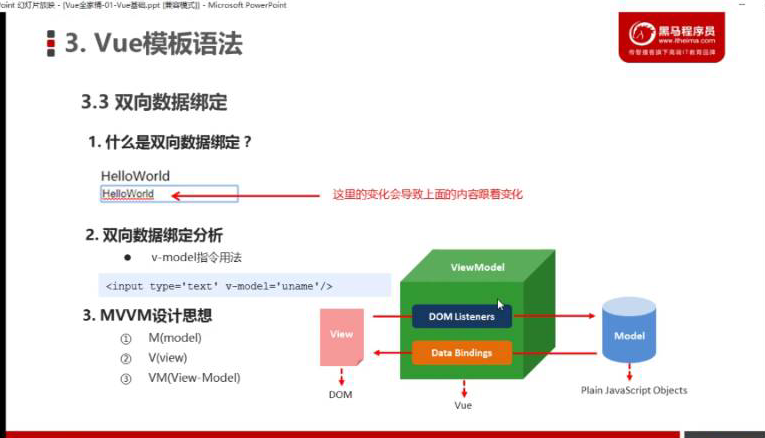
1. Vue是目前最火的前端框架，react是目前最流行的前端框架，都可以制作手机APP
2. 前端三大主流框架，vue.js angular.js,react.js
3. Vue.js是一套构建用户界面的框架，只关注视图层，易上手，兼容性强
4. 前端的主要工作组要是和视图打交道，MVC中的V这一层
5. 使用框架能够提高开发效率
6. 帮助解决我们不必要的dom操作

## 框架与库的区别

1. 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，则需重新构建整个项目
2. 库：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

## 为什么有了MVC还要有MVVM





# Vue的基本代码

 <!-- 将来new的vue实例会控制这个元素中的所有内容 -->

    <!-- vue所控制的这个元素区就是我们的V -->

    <div id="app">

        <!-- 将msg渲染出来的固定语法 -->

        <!--    {{}}规定这样写    -->

        <p> {{msg}}</p>

    </div>

     <!-- 1. 导入vue的包 -->

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

    <script>

        //2. 创建一个Vue的实例

        // 到我们导入包之后在浏览器中就多了一个Vue的构造函数

        // 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表示当前Vue的实例，要控制页面上的哪个区域（app），即指定要控制的区域

            el:'#app',

            // data属性中，存放的是el属性中要用到的数据

            // 这里的data就是MVVM中的M，专门用来保存每个页面中的数据

            data:{

                // 通过vue提供的指令，很方便就能将数据渲染到页面上，程序员不在手动操作dom元素了

                // 前端框架不提倡我们去手动操作dom

                msg:'欢迎学习vue'

            }

        })

    </script>

# Vue指令的使用

     <!-- 创建要控制的区域 -->

    <div id="app">

        <!-- 将数据渲染到页面上 -->

        <!-- 使用v-cloak能够解决插值表达式闪烁的问题，例如网速不够出现漏洞 -->

        <p v-cloak>这是一个值{{msg}}，插值表达式在这里添加内容是可以在view视图上看见的</p>

        <!-- 默认v-cloak是没有闪烁问题的 -->

        <p v-text="msg">如果在这里添加文本，是不会在view试图视图上看见的</p>

        <!-- v-text会覆盖元素中原本的内容 ，但是插值表达式指挥替换自己的这个占位符，不会把整个元素的内容清空-->

        <p v-html='msg1'></p>

        <!-- v-bind:是Vue中，提供的用于绑定自己定义的属性指令 -->

        <!-- 注意:v-bind:指令可以被简写为: -->

        <!-- v-bind中可以书写合法的JS表达式 -->

        <input type="button" value="按钮" v-bind:title="mytitle+'123'">

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

        <!-- 所有的事件绑定,采用v-on指令 -->

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

    </div>

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

   <script>

       var vm=new Vue({

        //    创建一个配置项，写下要控制的元素

        el:'#app',

        data:{

            msg:'123',

            msg1:'<h1>要显示标签里面的东西，就需要v-html,但是v-html会将整个元素的内容清空</h1>',

            mytitle:'这是一个自己定义的title'

        },

        methods:{

            // 这个methods属性中定义了当前vue实例所有可用方法

            show:function(){

                alert("你点击了我");

            }

        }

       })

   </script>

## v-cloak

1. 使用v-cloak能够解决插值表达式闪烁的问题，例如网速不够出现漏洞
2. 默认是没有闪烁问题的
3. {{}}，插值表达式指挥替换自己的这个占位符，不会把整个元素的内容清空

## v-text=’这里填要填充的属性名称’

1. v-text会覆盖元素中原本的内容

## v-bind:

1． Vue提供的属性绑定机制

2. v-bind:指令可以被简写为:

3. v-bind中可以书写合法的JS表达式 ， 例如 :title="mytitle+'123'，最后将计算出来的结果给title

4. 只能实现单项数据绑定，从M到V

## v-on:什么事件=”什么方法”

1. 所有的事件绑定,采用v-on指令，即这个指令是vue提供的事件绑定机制
2. v-on:的缩写是@

## v-for与key

1. 迭代数组
2. 迭代对象中的属性
3. 迭代数字
4. 在Vue2.2.0+版本里，当在组件中使用v-for时，key现在是必须的 ,保证data里面的数据与页面上的数据关联
5. <div id="app">
6. <!-- item是循环出来的每一项 -->
7. <!-- 数组的调用方式 -->
8. <p v-for="item in arr1">第{{item}}个p标签</p>
9. <!-- 上面的和下面的是一样的，下面的i表示每一项的索引值 -->
10. <p v-for="(item,i) in arr1">第{{item}}个p标签  -----索引值为{{i}}</p>

13. <!-- 注意in前面的这个单词，注意下面的应用 -->
14. <!-- 数组对象的调用方式 -->
15. <p v-for="my in arr2">Id:{{my.id}}-----名字是{{my.name}}</p>
16. <p v-for="(user,i) in arr2">Id:{{user.id}}--------名字是{{user.name}}------索引值为{{i}}</p>
17. <!-- 对象键值对的使用如下 -->
18. <!-- 注意：在边离对象键值对的时候，除了有value,key,在第三个位置还有一个索引值 -->
19. <p v-for="(value,key,i) in info">值是：{{value}}-----键是：{{key}}------索引值为{{i}}</p>
20. <!-- 迭代数字 -->
21. <!-- in 之后可以放普通数组，对象数组，对象，数字 -->
22. <!-- 注意：如果使用v-for迭代数字的话，从1开始 -->
23. <p v-for="count in 10">这是第{{count}}次循环</p>
24. </div>
25. <script src="lib/vue.min.js"></script>
26. <script>
27. var vm=new Vue({
28. el:'#app',
29. data:{
30. //    遍历数组
31. arr1:[1,2,3],
32. arr2:[
33. {id:1,name:"张三"},
34. {id:2,name:"李四"},
35. {id:3,name:"王五"},
36. {id:4,name:"柳儿"},
37. {id:5,name:"刘二"}
38. ],
39. //   遍历对象
40. info:{
41. name:"王二",
42. tel:1383838438,
43. gender:"男"
44. }
45. }
46. })
47. </script>

## Key

 <div id="app">

       <div>

           <label >

               Id:

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

           </label>

           <label >

            name:

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

        </label>

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

       </div>

       <!--注意：v-for循环的时候，key属性只能用number来获取string-->

       <!-- 注意：key在使用的时候，必须使用v-bind属性绑定的形式来指定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/数字类型的  :key值 -->

       <p v-for="item in info" :key="item.id">

           <!-- 数组对象的调用方式 -->

        <input type="checkbox">

        ----id:{{item.id}}-------- name:{{item.name}}

        </p>

   </div>

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

   <script>

       var vm=new Vue({

           el:'#app',

           data:{

                info:

                    [

                       {

                        id:1,

                        name:"王二",

                       },

                       {

                        id:2,

                        name:"李麻子",

                        },

                       {

                       id:3,

                       name:"张三",

                       }

                    ],

                id:'',

                name:''

           },

           methods:{

               add(){

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

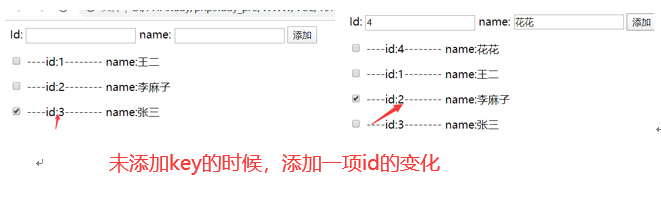
                this.info.unshift({id:this.id,name:this.name});

               }

           }

       })

   </script>





## v-if

特点：①每次都会重新删除和创建元素

②v-if有较高的切换性能消耗

总结：如果元素有平凡的切换，最好不要用v-if,推荐使用v-show

## v-show

特点：①每次不会重新删除和创建dom，只是将元素的样式切换为display：none

②有较高的初始渲染消耗

总结：如果元素创建出来可能永远也不显示，最好不要使用v-show，推荐使用v-if

## v-once

* 1. 只编译一次② 数据显示之后不再具有数据响应功能

## 事件绑定——参数传递

1. 如果事件直接绑定一个函数名称，那么默认会传递事件对象作为函数
2. 如果事件绑定函数调用，名称必须是$event

## 事件修饰符

1. .stop阻止冒泡，阻止所有冒泡
2. .prevent阻止默认行为
3. .capture添加事件侦听器时使用事件捕获模式
4. .self只会阻止自己身上的冒泡，不会阻止其他
5. .once事件只触发一次
6. <div id="app">
7. <div class="inner" @click="divHander">
8. <!-- 用.stop阻止冒泡 -->
9. <input type="button" value="点我" @click.stop="btnHander">
10. </div>
11. <!-- .prevent阻止默认行为 -->
12. <a href="http://baidu.com" @click.prevent="linkClick">默认行为有跳转</a>

15. <div class="inner" @click="divHander">
16. <!-- 用.capture实现捕获触发机制 -->
17. <input type="button" value="点我" @click.capture="btnHander">
18. </div>
19. <!-- 用.self只有点击自己的时候触发 -->
20. <div class="inner" @click.self="divHander">
21. <input type="button" value="点我" @click="btnHander">
22. </div>
24. <!-- .once只允许执行一次 -->
25. <a href="http://baidu.com" @click.once.prevent="linkClick">默认行为有跳转</a>
27. </div>
28. <script src="lib/vue.min.js"></script>
29. <script>
30. var vm=new Vue({
31. el:'#app',
32. data:{
33. msg:"我是div的值",
34. msg1:"我是input的值"
35. },
36. methods:{
37. divHander(){
38. alert(this.msg);
39. },
40. btnHander(){
41. alert(this.msg1);
42. }
43. }
44. })
45. </script>

## 按键修饰符

1. 注意：操作的时候光标要固定在绑定的位置

 <div id="app">

       <form action="">

        <p>用户名：

            <input type="text" @keyup.delete="del" v-model="user">

        </p>

        <p>密  码：

            <input type="text"  @keyup.enter="submit"  v-model="passward">

        </p>

        <input type="button"  value="提交" @click="submit">

       </form>

    </div>

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

    <script>

        var vm=new Vue({

            el:"#app",

            data:{

                user:'',

                passward:''

            },

            methods:{

                submit(){

                    alert(this.user,this.passward);

                },

                del(){

                    this.user='';

                    this.passward='';

                }

            }

        })

    </script>

## 自定义按键修饰符

* + 1. 定义规则：自定义按键修饰符名字是自定义的，但是对应的值必须是按键对应event.keyCode值

如：Vue.config.keyCodes.名字=值

Vue.config.keyCodes.a=65;

## v-model实现数据的双向绑定

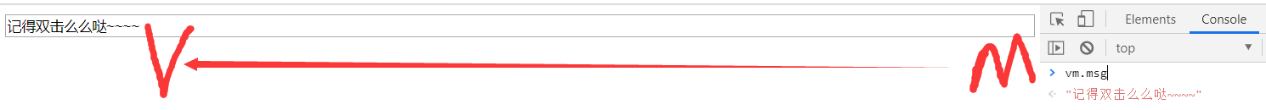
* 等价与v-bind去绑定那个值+v-on去处理这个事件

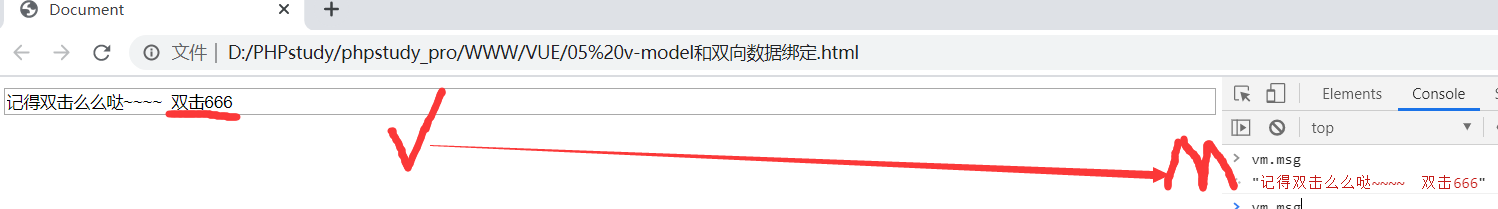
1. 可以实现表单元素和model中数据的双向绑定
2. 注意：v-model只能运用在表单元素中
3. 常见表单元素：input(radio,text,address,email…),select,checkbox,textarea
4. 数据绑定就是将数据填充到标签中

 <div  id="app">

1. <!-- v-model实现数据的双向绑定 -->
2. <input type="text"  v-model="msg">
3. </div>
4. <script src="lib/vue.min.js"></script>
5. <script>
6. var vm=new Vue({
7. el:'#app',
8. data:{
9. msg:"记得双击么么哒~~~~"
10. }
11. })
12. </script>

Viewmodel -> model   数据绑定  
model -> viewmodel  事件监听





## 利用v-model实现简易计算器

 <div id="app">

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

        <select 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="total">

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

    </div>

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

                n1:0,

                n2:0,

                result:0,

                opt:'+'

            },

            methods:{

                // 执行计算方法

                total(){

                    通过判断符号来计算

                    switch(this.opt){

                        case "+":

                        // 为避免把加号理解为连接符

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

                        break;

                        case "-":

                        this.result=this.n1-this.n2;

                        break;

                        case "\*":

                        this.result=this.n1\*this.n2;

                        break;

                        case "/":

                        this.result=this.n1/this.n2;

                        break;

                    }

                    // eval()可以将字符串解析执行

                    // 另外一种做法如下

                    // 注意，下面的这个方法在正式开发的时候尽量不要用，这个是投机取巧

                    // var string="parseInt(this.n1)"+this.opt+"parseInt(this.n2)"

                    // this.result=eval(string);

                }

            }

        })

    </script>

# 跑马灯效果

## 分析

1. 给“滚动“按钮绑定一个事件，用v-on或者@
2. 在按钮的事件处理函数中，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来进行字符串的截取操作，把第一个字符串截取出来放大最后一个位置即可
3. 为了实现点击按钮自动截取的功能，需要将2.中的代码放到一个定时器中去

## 基本代码

 <!-- 创建控制区域 -->

    <div id="app">

        <!-- 绑定监听事件 -->

        <input type="button" value="滚动" @click="go">

        <h3>{{msg}}</h3>

    </div>

    <!-- 1. 导包 -->

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

                msg:"猥琐发育,别浪~~~~~~~"

            },

            methods:{

                // 函数的形式也可以这样写

                go(){

                    // 在vm实例中,如果要获取data身上的属性,或者说调用methods内部的方法,通过this来访问,this直接代表new出来的这个vm实例

                    // 格式为:this.数据属性名或者this.方法名来进行访问

                    // substring()包括开头,不包括结尾

                    // 获取data里面的msg的数据并截取第一个字符

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

                    // 获取data里面的msg的数据并截取后面的所有字符

                    var end=this.msg.substring(1);

                    // 重新拼接得到新的字符串赋值给msg

                    this.msg=end+start;

                    // vm有一个特点,会自动监听自己身上data中所有数据的改变,只要数据一改变,就会自动将新的数据从data上同步到view页面中去(这样的好处是,程序员不需要在操作dom了)

                }

            }

        })

    </script>

## 以前定时器的跑马灯

<!-- 创建控制区域 -->

    <div id="app">

        <!-- 绑定监听事件 -->

        <input type="button" value="滚动" @click="go">

        <h3>{{msg}}</h3>

    </div>

    <!-- 1. 导包 -->

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

                msg:"猥琐发育,别浪~~~~~~~"

            },

            methods:{

                // 函数的形式也可以这样写

                go(){

                    // 在vm实例中,如果要获取data身上的属性,或者说调用methods内部的方法,通过this来访问,this直接代表new出来的这个vm实例

                    // 格式为:this.数据属性名或者this.方法名来进行访问

                    // substring()包括开头,不包括结尾

                    // 在定时器中的this指的是function,想要获取vm上的数据可以事先备份一个

                    var self=this;

                  setInterval(()=>{

                      // 获取data里面的msg的数据并截取第一个字符

                      var start=self.msg.substring(0,1);

                      // 获取data里面的msg的数据并截取后面的所有字符

                      var end=self.msg.substring(1);

                     // 重新拼接得到新的字符串赋值给msg

                     self.msg=end+start;

                    // vm有一个特点,会自动监听自己身上data中所有数据的改变,只要数据一改变,就会自动将新的数据从data上同步到view页面中去(这样的好处是,程序员不需要在操作dom了)

                  },400)

                }

            }

        })

    </script>

## 完整版跑马灯

    <!-- 创建控制区域 -->

    <div id="app">

        <!-- 绑定监听事件 -->

        <input type="button" value="滚动" @click="go">

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

        <h3>{{msg}}</h3>

    </div>

    <!-- 1. 导包 -->

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

                msg:"猥琐发育,别浪~~~~~~~",

                // 在data上定义定时器ID

                IntervalId:null

            },

            methods:{

                // 函数的形式也可以这样写

                go(){

                    // 在vm实例中,如果要获取data身上的属性,或者说调用methods内部的方法,通过this来访问,this直接代表new出来的这个vm实例

                    // 格式为:this.数据属性名或者this.方法名来进行访问

                    // substring()包括开头,不包括结尾

                    // 在定时器中的this指的是function,想要获取vm上的数据可以事先备份一个

                    var self=this;

                    // 判断计时器是不是开启的

                    if(this.IntervalId!=null) return;

                  this.IntervalId=setInterval(()=>{

                    // 获取data里面的msg的数据并截取第一个字符

                    var start=self.msg.substring(0,1);

                    // 获取data里面的msg的数据并截取后面的所有字符

                    var end=self.msg.substring(1);

                    // 重新拼接得到新的字符串赋值给msg

                    self.msg=end+start;

                    // vm有一个特点,会自动监听自己身上data中所有数据的改变,只要数据一改变,就会自动将新的数据从data上同步到view页面中去(这样的好处是,程序员不需要在操作dom了)

                  },400)

                },

                stop(){

                    clearInterval(this.IntervalId);

                    // 每当清除了计时器的时候的时候,需要将setInterval设为null,上面就能判断定时器是不是开启的了.

                    this.IntervalId=null;

                }

            }

        })

    </script>

# 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
5. <div id="app">
6. <!-- 1.直接传递数组，注意：这里需要用v-bind绑定 -->
7. <!-- 在为class使用v-bind绑定的时候，对象的属性是类名，对象那个的属性可带引号也可不带引号， 属性的值是一个标识符-->
8. <h1 :class=['thin','color']>我是H1</h1>
9. <h2 :class=['thin','color',flag?'active':''] @click="show">这是H2</h2>
10. <h2 :class=['thin','color',{'active':flag}] @click="show">这是H2</h2>
11. <h2 :class='obj'  @click="show">这是H2</h2>
12. </div>
13. <script src="lib/vue.min.js"></script>
14. <script>
15. var vm=new Vue({
16. el:'#app',
17. data:{
18. flag:false,
19. obj:{thin:true,fat:false,color:true,active:true}
20. },
21. methods:{
22. show(){
23. this.flag=true;
24. }
25. }
26. })
27. </script>

## 内敛样式

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，并直接引用到 :style中。即①在data上定义样式，②在元素中，通过属性绑定的ing是，将样式对象应用到元素中
3. 在 :style中通过数组，引用多个data上的样式对象。即①在data上定义样式，②在元素中，通过属性绑定的ing是，将样式对象应用到元素中
4. <div id="app">
5. <!-- 对象就是一个无序键值对的集合 -->
6. <!-- 注意：如果样式的中间有横线，必须要用单引号包裹起来 -->
7. <h1 :style="{color:'red','font-size':'70px'}">这个是一个H1</h1>
8. <h2 :style=" styleObj1">这个是一个H2</h2>
9. <h2 :style="[ styleObj1, styleObj2]">这个是一个H2</h2>
10. </div>
11. <script src="lib/vue.min.js"></script>
12. <script>
13. var vm=new Vue({
14. el:'#app',
15. data:{
16. // 注意这里是对象，不需要加引号
17. styleObj1:{color:'red','font-size':'70px'},
18. styleObj2:{'background-color':'yellow'}
19. }
21. })
22. </script>

# 品牌管理实例

## 遍历数组的方法

1. forEach：无法被终止
2. some：返回true的时候终止
3. filter：进行过滤，将满足条件的数组返回得到一个新数组
4. findIndex ：找到对应项目的索引

## 代码展示

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

    <link rel="stylesheet" href="bootstrap-3.3.7-dist/bootstrap-3.3.7-dist/css/bootstrap.min.css">

</head>

<body>

    <div id="app">

        <!-- 这个是最顶部的信息 -->

       <div class="panel panel-primary">

             <div class="panel-heading">

                   <h3 class="panel-title">添加品牌</h3>

             </div>

             <!-- 让这些表格处于一行form-inline -->

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

                   <label>

                       id:

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

                   </label>

                   <label>

                    name:

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

                </label>

                <label>

                    <!-- 事件绑定如果添加（），表示可以传参 -->

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

                </label>

                <label>

                    搜索名称关键字：

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

                </label>

             </div>

       </div>

        <!-- 添加一个表格，这个是最顶部的表格 -->

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

            <thead>

                <tr>

                    <th >id</th>

                    <th>name</th>

                    <th>time</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,我们自定义了一个search方法,同时,把所有的关键字通过传参的形式,传递给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返回. -->

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

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

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

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

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

                </tr>

            </tbody>

        </table>

    </div>

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

    <script>

        var vm=new Vue({

            el:"#app",

            data:{

                Id:'',

                Name:'',

                Keywords:'',

                // 每一行信息是一个对象，整体是一个大数组

                list:[

                    {"id":1,"name":"奔驰","time":new Date()},

                    {"id":2,"name":"宝马","time":new Date()},

                    {"id":3,"name":"奥迪","time":new Date()}

                    ]

            },

            methods:{

                // 添加方法

                add(){

                    var car={"id":this.Id,"name":this.Name,"time":new Date};

                    this.list.push(car);

                },

                // 删除方法

                del(Id){

                    //  根据id删除数据

                    // 1.如果根据id找到删除这一项的索引

                    // 2.如果找到索引了直接调用数组的splice方法

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

                    // (item,i)=>{}是一个回调函数，这里的this表示定义这个函数的父级作用域的this。也就是del函数

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

                        // 如果列表里面的id等于我传进来的id,就返回true，终结循环

                        if(item.id==Id){

                            return true;

                        }

                         // splice(从哪里开始删，删除几个，插入的数据)

                         this.list.splice(i,1);

                    });

                    // 方法二：

                    // findIndex用于查找索引

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

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

                    //         return true;

                    //     }

                    // });

                    // this.list.splice(index,1);

                },

                // 根据关键字进行数据的搜索

                search(Keywords){

                //     var newList=[];

                //     // 遍历整个数组

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

                //         // Keywords中的信息在list里面执行

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

                //             // 搜索到匹配的信息以后,将匹配的信息重新添加到数组里

                //             newList.push(item);

                //         }

                //     });

                //     return newList;

                    // 方法二:

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

                        // 在ES6中,为字符串提供了一个新方法,叫做string.prototype.includes('要包含的字符串'),如果包含,返回true,否则返回false

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

                        return item;

                        }

                    })

                   return newList;

                }

            }

        });

    </script>

</body>

</html>

# 表单操作

1．作用：用户数据的交互

* 基于vue的常见表单：

注意：复选框或单选框双向绑定值的时候

1. 两个单选框或复选框需要同时通过v-model双向绑定一个值
2. 每个单选框或复选框必须要有一个value值且value值不能一样
3. 当某个单选框或者复选框被选中的时候，v-model会将当前的value值改变data中的数据
4. 当复选框或者单选框被选中的时候，我们只需要实时监控就可以了

* 获取下拉框和文本框的值
  1. 需要给select通过v-model进行双向绑定一个值
  2. 每个option必须要有value属性，且value值不能一样
  3. 当某个option选中的时候，v-model会将当前的value值改变，data中的值就是选中的值，我们需要实时监控他的值就可以了

1. Radio：单选框：通过v-model来获取单选框中的值
2. Checkbox：复选框：通过v-model来获取复选框中的值，data中需要多选的，要定义成数组，否则无法实现
3. Select：下拉选项
4. Textarea：多行文本，是一个双标签，不需要绑定value的属性，用v-model进行双向数据绑定

## 代码展示

 <!-- 注意，因为这里的action提交表单我们没有设置他的行为，所以下面如果我们采用submit来提交表单是不能的，他只能刷新当前页面 -->

        <form action="">

            <p>

                姓名：

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

            </p>

            <p >

                性别：

                <!-- 单按钮这里的那么值必须是一样的才能只选一个 -->

                <!-- value值必须要有，且值不能一样 ,用于区分那个是哪个-->

                <input type="radio"  id="boy"  value="1" v-model="sex">男

                <input type="radio"  id="girl" value="2" v-model="sex">女

            </p>

            <p >爱好：

                <!-- value值不能一样，值是一个数组,可以多选 -->

                <input type="checkbox" value="1" v-model="hobby"> 唱歌

                <input type="checkbox" value="2" v-model="hobby"> 跳舞

                <input type="checkbox" value="3" v-model="hobby"> 篮球

                <input type="checkbox" value="4" v-model="hobby"> 乒乓球

            </p>

            <p>职业：

                <!-- 如果加multiple设为true则可以选中多个，选中多个的时候，data里的profession是数组 -->

                <select   v-model="profession" multiple="true">

                    <option value="0">

                        请选择职业：

                    </option>

                    <option value="1" >

                        公务员

                    </option>

                    <option value="2">

                        大学教师

                    </option>

                    <option value="3" >

                        小学教师

                    </option>

                    <option value="4" >

                        中学教师

                    </option>

                </select>

            </p>

            <p>个人简历：<textarea name="" id="" cols="30" rows="10" v-model="info"></textarea></p>

            <input type="button" value="提交" @click="submit">

    </form>

    </div>

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

                name:'',

                sex:1,

                hobby:[],

                profession:[1,2,3],

                info:''

            },

            methods:{

                submit(){

                    console.log(this.name,this.sex,this.hobby,this.profession,this.info);

                }

            }

        })

    </script>

## 表单修饰符

    <div id="app">

        <p>

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

        </p>

    </div>

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

    <script>

        var vm=new Vue({

            el:'#app',

            data:{

               age:"hahah"

            }

        })

    </script>

1. .number转换为数值：注意：当开始输入非数字的字符串时，因为vue无法将字符串转换成数值，所以属性值将实时更新成相同的字符串。即使后面输入数字，也将被视为字符串。
2. .trim自动过滤用户输入的首位空白字符：注意：只能去掉首尾的，不能去掉中间的
3. .lazy将input事件切换成change事件：注意：.lazy修饰符延迟了同步更新属性值的时机，将原本绑定在input事件的同步逻辑转变为绑定在change事件上，change事件是失去焦点的时候触发，input事件是输入框信息发生变化的时候触发

# 自定义指令

## 语法规则

<div id="app">

        <!-- 需求，刷新到浏览器页面的时候，自动有焦点 -->

        <p>

            <!-- 在这里使用，定义的时候不需要写v-自定义名称，使用的时候要写 -->

            <input type="text" v-自定义名称>

        </p>

    </div>

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

    <script>

        // 自定义指令

        Vue.directive('自定义名称',{

            inserted:function(el){    //inserted是固定的

            // el表示指令说绑定的元素

            // 事件是聚焦

            el.事件方法;

            }

        })

## 代码展示

 <div id="app">

        <!-- 需求，刷新到浏览器页面的时候，自动有焦点 -->

        <p>

            <!-- 在这里使用，定义的时候不需要写v-自定义名称，使用的时候要写 -->

            <input type="text" v-foc>

        </p>

    </div>

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

    <script>

        // 自定义指令

        Vue.directive('foc',{

            inserted:function(el){    //inserted是固定的

            // el表示指令说绑定的元素

            // 事件是聚焦,focus()内置方法

            el.focus();

            }

        })

        var vm=new Vue({

            el:'#app',

            data:{

            }

        })

    </script>

## 带参数的自定义指令

<div id="app">

        <!-- 背景变色 -->

        <input type="text" v-color="msg">

    </div>

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

    <script>

        // 自定义指令——带参数

        Vue.directive('color',{

            // 这里inserted与bind是一样的效果

            // 第二个参数是形参，可以随意起名字

            bind:function(el,binding){

                // 如果要输出msg的值,语法如下:形参.value(这也叫获取参数)

                // alert(binding.value);

                // 根据参数设置背景色

                // 1.获取到这个元素,设置样式的属性值

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

            }

        })

        var vm=new Vue({

            el:"#app",

            data:{

                msg:{

color: "skyblue"

}

            }

        })

    </script>

## Vue.directive注册全局指令

1. 使用自定义指令只需要在对用的元素中，加上v-的前缀形成类似内部指令
2. 在定义指令中，以驼峰命名的方式定义，如Vue.direction
3. 在html中使用的时候，只能通过v-focus-a来使用

## Vue.directive注册全局指令，带参数

1. V-bind声明周期，只调用一次，指令第一次绑定到元素时调用

## 自定义指令局部指令：

1. 自定义指令需要定义在directives的选项中，用法和全局用法一样
2. 局部指令只能在当前组件中使用
3. 当全局指令和局部指令同名时以局部指令为标准

## 自定义局部指令代码展示：

<div id="app">

        <!-- 背景变色,光标聚焦-->

        <input type="text" v-color="msg" v-focus>

    </div>

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

    <script>

        var vm=new Vue({

            el:"#app",

            data:{

                // 注意,在自定义指令设置背景的时候,这里是对象哦

                msg:{

                color:"skyblue"

                }

            },

            directives:{

                // 用于放置具体的指令

                // 写法如下:

                color:{

                    bind:function(el,binding){

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

                    }

                },

                focus:{

                    inserted:function(el){

                        el.focus();

                    }

                }

            }

        });

    </script>

# 计算属性 computed

 <div id="app">

        <div>

            <!-- 将msg通过split方法,用''进行分割成数组,用reverse()方法将数组倒置,再通过join()将数组变成字符串 -->

            {{msg.split('').reverse().join('')}}

        </div>

        <div>

            <!-- 计算属性的调用方法 -->

            <!-- 注意:这里不需要加括号 -->

            {{reverseString}}

        </div>

    </div>

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

    <script>

        // 将msg的顺序进行反转

        //方法1.用原生JS中的split方法将字符串分割为数组,在调用reverse方法将数组中的元素进行反转,再通过join()方法让数组变成字符串

        // 方法2.计算属性

        var vm=new Vue({

            el:"#app",

            data:{

                msg:"abcdefghijklmnopq",

            },

            // 注意:计算属性是基于data中的数据来处理的

            computed:{

                // 计算属性的定义方式

                reverseString:function(){

                    // 这里必须要返回值,并且data里面的数据通过this来调用

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

                }

            }

        })

    </script>

## 计算属性与方法的区别：

1. 计算属性是有缓存的（基于data中的数据进行缓存），方法不存在缓存
2. 计算属性当计算量比较大的时候，开销大

# 过滤器

1. 格式化数据。如：将字符串首字母格式化为大写，将日期格式化为指定的格式
2. 过滤器用在两个地方mustache插值和v-bind表达式
3. 可以添加到js表达式的尾部，由管道符号指示
4. 支持级联操作
5. 过滤器不改变真正的data,而只是改变渲染的结果，并返回过滤后的版本
6. 全局注册时是filter，没有s而局部过滤器是fiters,是有s的
7. 格式为{{name|过滤器名称}}
8. 语法：

Vue.filter(‘过滤器的名称’ ，function(第一个参数是过滤器管道符前面传过来的数据){ })

1. 过滤器中一定有返回值，这一外界使用过滤器的时候才能让拿到结果

## 定义过滤器的方式

//过滤器的定义方式,参数value表示要过滤的数据

        Vue.filter('过滤器名称',function(value){

            // 这里需要返回一个处理结果

            return "处理结果"

        })

## 代码展示

 <div id="app">

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

        <!-- 用于显示过滤后的内容，upper是过滤器的名称 -->

        <div>{{msg|upper}}</div>

        <!-- 使用多个过滤器表示把前面处理的结果再进行处理 -->

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

    </div>

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

    <script>

        // 定义过滤器

        Vue.filter('upper',function(value){

            // 将英文首字母大写

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

        });

        Vue.filter('lower',function(value){

            // 将首字母变为小写

            return value.charAt(0).toLowerCase()+value.slice(1);

        })

        var vm=new Vue({

            el:"#app",

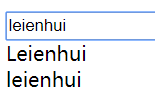
            data:{

               msg:'',

            }

        });

    </script>



## 局部过滤器

// 局部自定义属性

            filters:{

                // upper是过滤器的名称

                upper:function(value){

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

                }

            }

# 过滤器中传递参数

div id="app">

        <!-- 这里的参数在下过滤器的arg中得到 -->

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

    </div>

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

    <script>

        // 使用过滤器格式化日期

        // 过滤器

        Vue.filter('format',function(value,arg){

            if(arg=='yyyy-mm-dd'){

                var str='';

                str+=value.getFullYear()+'-'+(value.getMonth()+1)+'-'+value.getDate();

                return str;

            }

        })

        var vm=new Vue({

            el:"#app",

            data:{

                date:new Date(),

            }

        })

# 侦听器 watch

* 当data中数据一旦发生变化就通知侦听器所绑定的方法
* 数据变化时执行异步或开销大的操作

## 代码展示

    <div id="app">

        <p>

            姓：

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

        </p>

        <p>

            名：

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

        </p>

        <div>{{fullName}}</div>

    </div>

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

    <script>

        // 需求：改变输入框的值，显示的模块马上变值

        var vm=new Vue({

            el:"#app",

            data:{

                firstName:'',

                lastName:'',

                fullName:''

            },

            watch:{

                // 方法1：使用监听器

                // 监听firstName,函数与属性名保持一致

                // value这个参数表示当前的最新值，当firstName的值发生变化，value值就发生变化

                firstName:function(value){

                    this.fullName=value+' '+this.lastName;

                },

                // 侦听lastName

                lastName:function(value){

                    this.fullName=this.firstName+' '+value;

                }

            },

            // 方法2：使用计算属性

            // computed:{

            //     fullName(){

            //         // 使用计算属性的时候。data里面fullName这个属性就可以不用了，用下面的方法进行调用

            //         return this.firstName+''+this.lastName;

            //     }

            // }

        })

    </script>

## 侦听器的应用场景：

需求：验证用户名是否可用，当输入框失去焦点的时候触发侦听

   <div id="app">

        <span>用户名：</span>

        <!-- 用.lazy当表单元素失去焦点的时候，才触发事件 -->

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

        <span> {{tip}}</span>

    </div>

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

    <script>

        //1.  通过v-model实现数据绑定

        //2.  通过侦听器监听输入信息的变化

        // 3. 鼠标拾取焦点的时候触发事件

        var vm=new Vue({

            el:"#app",

            data:{

                name:'',

                tip:'',

            },

            methods:{

                // 参数用于接收传过来的name

                checkName:function(user){

                var self=this;

                    // 接口调用，但是我们可以使用定时任务的方式模拟接口调用

                    setTimeout(() => {

                        // 模拟接口调用

                        if(user=='12123'){

                            self.tip="该用户名存在，换一个名字"

                        }else{

                            self.tip="可以使用用户名"

                        }

                    }, 1000);

                }

            },

            watch:{

                // 采用监听器监听数据用户名是不是变化，如果变化了，那么调用后台接口进行验证，根据验证结果显示提示信息

                name:function(value){

                    // 调用后台接口验证用户名的合法性，将其写到方法里面

                    this.checkName(value);

                    this.tip="正在验证"

                }

            }

        })

    </script>

1. 监听到对象内部属性的改变，只有data中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

# 生命周期

1. 事物从出生到死亡的过程
2. Vue实例从创建到销毁的过程，这些过程中伴随着一些函数的自调用，我们称这些函数为钩子函数

## 常用的钩子函数

1. beforeCreate:在实例化之后，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
2. 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
3. beforeMount：在挂载之前被调用，此时页面上还看不到真实数据，只是一个模板而已
4. mounted: el被新创建的vm.$el替换并挂载到实例上之后调用该钩子，数据已经真实渲染到页面上，在这个钩子函数里面，我们可以使用一些第三方插件
5. beforeUpdate：数据更新时调用，发生在虚拟dom打补丁之前，页面上数据还是旧的
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，页面上的数据已经替换成最新的
7. beforeDestroy:实例销毁之前调用
8. destroyed：实例销毁之后调用

# 数组的变异方法

1. 在vue中，直接修改对象属性的值无法触发响应式，当你直接修改看对象属性的值，你会发现只有数据边离，但是页面上内容并没有改变
2. 变异数组方法：保持数组方法原有功能不变的前提下对其进行功能拓展

## 方法

1. push():往数组最后面添加一个元素，成功返回当前数组的长度
2. pop():删除数组的最后一个元素，成功返回删除元素的值
3. shift():删除数组的第一个元素，成功返回删除元素的值
4. unshift():往数组最前面添加一个元素，成功返回当前数组的长度
5. splice():有三个参数，第一个是想要删除元素的下标（必填），第二个是想要删除的个数（必填），第三个是删除后想要在原来位置替换的值
6. sort():使数组按照字符编码默认从小打大排序，成功返回排序后的数组
7. reverse():将数组倒序，成功返回倒叙后的数组

## 替换数组

* 不会改变原始数组，但是总会返回一个新数组

1. filter():filter()方法创建一个新数组，新数组中的元素是通过检查指定数组中符合条件的所有元素
2. contact():contact()方法用于连接两个或者多个数组，该方法不会改变现有的数组
3. slice():slice()可从已有的数组中返回选定的元素，该方法并不会修改数组，而是返回一个子数组

# 动态数组响应式数据、

## 修改响应式数据

1. Vue.set(vm.item,indexOfItem,newValue)
2. Vue.$set(vm.item,indexOfItem,newValue)
   1. ：参数1表示要处理的数组名称
   2. ：参数2表示要处理的数组的索引
   3. ：参数3表示要处理的数组的值

# 组件的注册

## 全局组件注册语法

Vue.component('组件的名称'，{

            // 里面有两个属性

            // 1.data:组件数据

            // 2.template：组件模板内容

            data:function(){

                // 返回想要的对象

                return  {};

            },

            // 第二个参数是模板内容

            template：'字符串'

        })

## 注册组件的注意事项

* data必须是函数，
* 组件模板内容只有一个根元素
* 模板内容可以是模板字符串（ES6语法）
* // 模板字符串：
* template:`
* <div>
* <p @click="handle">模板字符串了，{{count}}次</p>
* </div>
* `
* 组件的命名方式：
  1. 短横线方式:Vue.compoent(‘my-component’,{})
  2. 驼峰式：Vue.component(‘MyComponent’,{})
  3. 注意：如果使用驼峰式命名组件，那么在使用组件的时候只能在字符串模板中使用驼峰式，在普通标签中必须使用短横线方式且每个单词的首字母都是小写

## 局部组件

    <div id="app">

        <!-- 在这里使用 -->

        <components-one></components-one>

        <components-two></components-two>

        <components-three></components-three>

        <components-four></components-four>

        <components-five></components-five>

    </div>

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

    <script>

        // 就在这里定义吧

        var ComponentsOne={

            data:function(){

                return {

                    msg:"组件1"

                }

            },

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

        };

        var ComponentsTwo={

            data:function(){

                return {

                    msg:"组件2"

                }

            },

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

        };

        var ComponentsThree={

            data:function(){

                return {

                    msg:"组件3"

                }

            },

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

        };

        var ComponentsFour={

            data:function(){

                return {

                    msg:"组件4"

                }

            },

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

        };

        var ComponentsFive={

            data:function(){

                return {

                    msg:"组件5"

                }

            },

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

        }

        var vm=new Vue({

            el:"#app",

            data:{

            },

            // 局部组件

            components:{

                // 在上面定义，在components中注册

                // 组件名称            组件内容

                'components-one':ComponentsOne,

                'components-two':ComponentsTwo,

                'components-three':ComponentsThree,

                'components-four':ComponentsFour,

                'components-five':ComponentsFive,

                // 局部注册的组件只能在他的父组件中使用，其他的是使用不了的

            }

        })

    </script>

# 组件间数据交互

## 父组件向子组件传值

* 组件内部通过props属性接收传递过来的值
* 父组件通过属性将值传递给子组件
* props传递数据原则：单项数据流（只允许父组件传值给子组件，不允许子组件传值给父组件）

## props属性名规则

* 若在props中使用驼峰形式，普通模板中需要使用短横线的形式，在使用的时候也是短横线的形式
* 但是在字符串模板中没有这个限制

## 代码展示

    <div id="app">

        <div>{{pmsg}}</div>

        <!-- 将父组件中的pstr转递给子组件 -->

        <!-- 直接传输数值类型 -->

        <!-- 注意这里数值的写法，如果加上“：”那么这个数是number类型，如果不加上，那个这个数字就是string类型 ,bool类型同理，加上:是boolean类型，不加上就是字符串类型-->

        <menu-item :str="pstr" :num="12"  :bool="true" :arr="parr" :obj="pobj"></menu-item>

    </div>

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

    <script>

        Vue.component('menu-item',{

            props:['str','num','bool','arr','obj'],

            // 在下面现实父组件中传递过来的内容

            // 当用到循环的时候都需要通过key来进行唯一区分

            template:`

            <div>

                <div>

                    {{str}}

                </div>

                <div>

                    数值是{{num}}

                </div>

                <div>

                    {{bool}}

                </div>

                <div>

                    <ul>

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

                    </ul>

                </div>

                <div v-for='(item,key) in obj'>{{item}}</div>

             </div>

            `

        })

        var vm=new Vue({

            el:"#app",

            data:{

                pmsg:"父组件中内容",

                pstr:"hello",

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

                pobj:{

                    "name":"翠花",

                    "age":18,

                }

            }

        })

    </script>

## 子组件向父组件传值

* 子组件通过自定义事件向父组件传递信息
* 父组件监听子组件的事件

## 代码演示

 <div id="app">

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

        <!-- 这里的被监听的名称要和下面的一致  enlarge-text -->

        <menu-item  :arr="parr" @enlarge-text="enlarge" ></menu-item>

    </div>

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

    <script>

        // $emit用于触发自定义事件,在父组件中进行事件的监听

        Vue.component('menu-item',{

            props:['arr'],

            template:`

            <div>

                <ul>

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

                </ul>

                <button @click="arr.push('lemon')">点击</button>

                <button @click="$emit('enlarge-text')">子组件按钮</button>

            </div>

            `

        })

        var vm=new Vue({

            el:"#app",

            data:{

                msg:"父组件内容",

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

                FontSize:10,

            },

            methods:{

                enlarge(){

                    this.FontSize+=5;

                }

            }

        })

    </script>

## 真正代码

    <div id="app">

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

        <!-- 这里的被监听的名称要和下面的一致  enlarge-text -->

        <menu-item  :arr="parr" @enlarge-text="enlarge($event)" ></menu-item>

    </div>

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

    <script>

        // $emit用于触发自定义事件,在父组件中进行事件的监听,第二个参数是可以传递给父组件中对应事件的数据，通过$event将其传递给事件,在事件中就能获取对应的值

        Vue.component('menu-item',{

            props:['arr'],

            template:`

            <div>

                <ul>

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

                </ul>

                <button @click="arr.push('lemon')">点击</button>

                <button @click="$emit('enlarge-text',5)">子组件按钮</button>

            </div>

            `

        })

        var vm=new Vue({

            el:"#app",

            data:{

                msg:"父组件内容",

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

                FontSize:10,

            },

            methods:{

                // 用value来接收$event的值，也就是子组件中传递过来的数据

                enlarge(value){

                    this.FontSize+=value;

                }

            }

        })

    </script>

## 非父子组件间传值

* 单独的事件中心管理组件间通信

var eventHub=new Vue();

* 监听事件与销毁事件

eventHub.$on(‘自定义事件的名称’，事件函数)————>事件监听

eventHub.$off(‘要销毁的事件名称’)—————————>事件销毁

* 触发事件

eventHub.$emit(‘要触发的事件(与监听的保持一致)’，’参数2’)

## 代码展示

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body>

    <div id="app">

        <div>父组件</div>

        <div>

            <!--销毁事件 ,在父组件方法中去定义 -->

            <button @click="destroy">销毁</button>

        </div>

        <!-- 在这里使用 -->

        <menu-item1></menu-item1>

        <menu-item2></menu-item2>

    </div>

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

    <script>

        // 1.创建组件并使用

        // 2.创建事件中心

        // 在eventHub内部就可以实现数据监听，但是我们一般在钩子函数mounted中监听，当mounted被触发的时候，模板就已经准备就绪了

        var eventHub=new Vue();

        Vue.component('menu-item1',{

            data:function(){

                return{

                    num:0,

                }

            },

            template:`

            <div>

                <div>num1:{{num}}</div>

                <button @click='num1'> num2变化</button>

             </div>

            `,

            methods: {

                num1(){

                // 触发兄弟组件的事件,value值就是第二个参数

                eventHub.$emit('item2', 5)

                }

            },

            // 钩子函数

            mounted() {

                eventHub.$on('item1',(value)=>{

                    this.num+=value

                })

            },

        });

        Vue.component('menu-item2',{

            data:function(){

                return{

                    num:0,

                }

            },

            template:`

            <div>

                <div>num2:{{num}}</div>

                <button @click='num2'> num1变化</button>

             </div>

            `,

            methods:{

                num2(){

                eventHub.$emit('item1', 10)

                }

            },

            mounted() {

                eventHub.$on('item2',(value)=>{

                    this.num+=value

                })

            },

        })

        var vm=new Vue({

            el:"#app",

            data:{

            },

            methods:{

                destroy(){

                    eventHub.$off('item1');

                    eventHub.$off('item2');

                }

            }

        })

    </script>

</body>

</html>

# 组件插槽

## 组件插槽作用

* 实现父组件向子组件传递模板的内容

## 插槽的位置

* 位于子组件的模板中slot标签中，slot标签属于vue的API

## 插槽的内容（使用组件的时候）

* 通过标签中的内容传递给slot

## 代码演示：

   <div id="app">

        <!-- 在这里使用自定义组件，将父组件中的信息传递到子组件中去 ，下面这些内容都会传递到模板的插槽中-->

        <alert-box>错误信息1</alert-box>

        <alert-box>错误信息2</alert-box>

        <!-- 当自定义组件中没有传递内容时，slot标签中的信息会显示，若传递了，插槽里面的默认信息会被覆盖 -->

        <alert-box></alert-box>

    </div>

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

    <script>

        Vue.component('alert-box',{

            // 注意：下面的错误信息提示是可变的，所以我们要预留一个插槽

            template:`

            <div>

                <strong>error：</strong>

                <slot>插槽默认内容</slot>

            </div>

            `

        })

        var vm=new Vue({

            el:"#app",

        })

    </script>

## 具名插槽用法

* 插槽定义：可以定义多个
* 插槽的内容：通过插槽的名称来匹配，当插槽内容没有对应的插槽名称来匹配的时候，会组件匹配到没有名称的插槽中

## 代码演示1

<div id="app">

        <menu-item>

            <!-- 这里的slot的值要与下面的插槽的name匹配 -->

            <p slot="header">我是头部</p>

            <p slot="footer">我是尾部</p>

            <p>我是中间的内容</p>

        </menu-item>

    </div>

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

    <script>

        // 子组件中定义，父组件中使用

        Vue.component('menu-item',{

            template:`

            <div>

                <header>

                    <slot name="header"></slot>

                </header>

                <main>

                    <slot></slot>

                </main>

                <footer>

                    <slot name="footer"></slot>

                </footer>

            </div>

            `

        })

        var vm=new Vue({

            el:"#app",

        })

    </script>

## 代码演示2

 <div id="app">

        <menu-item>

            <!-- 这里引入了一个新的标签,他的作用是临时性的包裹里面的内容，匹配一下信息，不会渲染到页面 -->

            <template slot="header">

                <!-- 这里可以写我们需要的内容 -->

                <p >我是头部1</p>

                <p >我是头部2</p>

            </template>

            <template slot="footer">

                <!-- 这里可以写我们需要的内容 -->

                <p >我是尾部</p>

            </template>

            <template>

                <p>我是中间的内容</p>

            </template>

        </menu-item>

    </div>

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

    <script>

        // 子组件中定义，父组件中使用

        Vue.component('menu-item',{

            template:`

            <div>

                <header>

                    <slot name="header"></slot>

                </header>

                <main>

                    <slot></slot>

                </main>

                <footer>

                    <slot name="footer"></slot>

                </footer>

            </div>

            `

        })

        var vm=new Vue({

            el:"#app",

            data:{

                pmsg:"父组件中内容",

                pstr:"hello"

            }

        })

    </script>

## 作用域插槽

    <div id="app">

    <!-- 父组件中的操作 -->

    <!-- 插槽内容 -->

        <menu-item :list="arr">

            <!-- template里面的操作决定子组件中某一条数据要显示成什么样子 -->

            <!-- slot-scope这个属性得到子组件的值，他的值就是info对应的值 -->

            <!-- slot-scope="自定义" -->

          <template slot-scope="slotProps">

              <!-- 来自于子组件的数据，并对其加工处理-->

              <strong v-if='slotProps.info.id==2'>

                {{slotProps.info.name}}

              </strong>

              <span v-else>

              {{slotProps.info.name}}

              </span>

          </template>

        </menu-item>

    </div>

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

    <script>

        // 子组件的操作

        Vue.component('menu-item',{

            props:['list'],

            template:`

            <div>

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

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

                </li>

             </div>

            `

        })

        var vm=new Vue({

            el:"#app",

            data:{

                arr:[

                    {"id":1,"name":"apple"},

                    {"id":2,"name":"orange"},

                    {"id":3,"name":"lemon"}

                    ],

            }

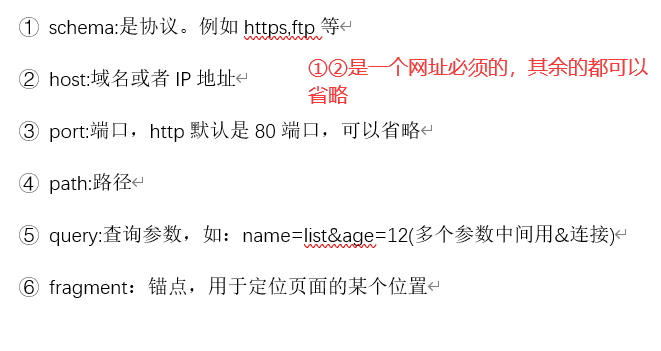
        })

    </script>

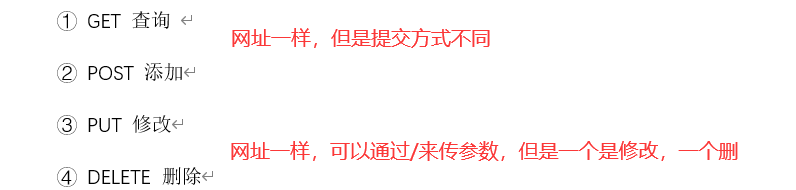
# 前后端交互模式

## URL地址的格式

* 标准（传统）的URL地址格式：schema：//host:port/path?query#fragment



* restful形式的URL：http请求方式



## promise用法

* 解决异步的方案，获取异步操作信息
* promise的好处：
  1. 避免多层异步调用嵌套问题
  2. promise提供了简洁API，使用异步操作更加容易

## promise的基本用法：

* 实例化Promise对象，构造函数中传递参数，该函数中用于处理异步任务
* resolve和reject用于处理成功和失败两种情况，并通过p.then获取处理结果

## 代码展示

<script>

        var  p=new Promise(function(resolve,reject){

            // 这里用于实现异步任务

            setTimeout(function(){

                var flag=false;

                if(flag){

                    resolve('正常情况');

                }else{

                    reject('异常了');

                }

            },1000)

        });

        p.then(function(res){

            // 接收正常信息

            console.log(res);

        },function(rej){

            // 接收异常信息

            console.log(rej);

        })

    </script>

## 基于promise处理Ajax请求

    <script>

        // 封装函数

        function queryData(url){

            var p=new Promise(function(resolve,reject){

                //Ajax发出一个http请求

                var xhr=XMLHttpRequest();

                xhr.onreadystatechange=function(){

                    // 这里处理响应的结果

                    // readystatechange只要发生变化就会被触发，不只被回调一次

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

                        //  正常情况

                        // 拿到服务器响应的数据

                        resolve(xhr.responseText);

                    }else{

                        // 异常情况

                        reject('服务器错误');

                    }

                    if(xhr.readyState!=4){

                    }

                };

                // 准备发送请求的参数，第一个请求是请求数据的方法，第二个参数是请求的地址,第三个参数默认表示异步

                xhr.open('get',url)

                xhr.send(null);

            });

            // 获取结果

            return p;

        }

        // 结果顺序输出

        queryData('请求数据的网址1').then(function(data){

            console.log(data,'这个data指网址1的结果')

            // return的是一个新的promise的实例对象

            // 下一个then的调用就是上一个promise的实例对象，then里面函数的参数用于接收上一个异步任务的结果

            return queryData('请求数据的网址2');

        }).then(function(data){

            console.log(data,'这个data指网址2的结果')

            return queryData('请求数据的网址3');

        }).then(function(data){

            console.log(data,'这个data指网址3的结果')

        })

## then参数中的返回值

* 返回promise实例对象
  1. 下一个then会得到上一个异步任务的结果
* 返回普通值：会在该处产生一个新的promise。以便下一个then来进行链式操作
  1. 返回的普通值会直接传递的下一个then，通过下一个then中的函数的参数接收该值

## promise的常用API

### 实例方法

* p.then()得到异步任务的正确结果。接收resolve传递的信息
* p.catch()获取异常信息，接收reject传递的信息
* p.finally()成功与否否会执行
* function fun1(){
* return new Promise(function(resolve,reject){
* setTimeout(() => {
* resolve(123);
* }, 100);
* });
* };
* fun1()
* .then(function(data){
* console.log(data);
* })
* .catch(function(data){
* console.log(data)
* })
* .finally(function(){
* console.log('finally');
* });

### 对象方法

Promise.all():并发处理多个异步任务，所有任务都执行完成才能得到结果

Promise.race():并发处理多个任务，只要有一个任务完成就能得到结果

 <script>

        // Ajax请求数据

        function queryData(url){

            return new Promise(function(resolve,reject){

                // 1、请求数据

                var xhr=new XMLHttpRequest();

                // 2、状态判断

                xhr.onreadystatechange=function(){

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

                        // 处理正常情况

                        resolve(xhr.responseText);

                    }else{

                        // 处理异常情况

                        reject('服务器错误');

                    }

                };

                // 注意：这是在状态改变这个函数的下面执行这些

                xhr.open('get',url);

                xhr.send(null);

            });

        }

            Promise.all([p1,p2,p3]).then((result)=>{

            //所有任务都完成之后才能得到结果，结果为一个数组，数组的顺序与传进去的一样

            console.log(result);

            });

            Promise.race([p1,p2,p3]).then((result)=>{

            //只能得到最先返回的结果

            console.log(result);

            });

    </script>

# 接口调用

## fetch方法：可以看成Ajax的升级版

* 语法结构
* <script>
* fetch(' 请求的URL地址').then(function(data){
* // 注意：通过then方法得到结果，但是这里的data不可以直接拿到数据，要通过fetch提供的API才能得到
* // 这个text得到的是一个Promise实例对象，通过return出去，让下一个then得到最终的数据
* // text()属于fetchAPI的一部分，它返回一个Promise实例对象，用于获取后台数据
* return data.text();
* })
* .then(function(data){
* // 这个才是真实的数据
* console.log(data);
* })
* </script>

## fetch请求参数

* method(String):http请求方法，默认为GET（GET，POST,PUT,DELETE）
* body(String):http的请求参数，传递实际的数据
* headers(Object):http的请求头，默认为{}，指定请求的类型

## GET，DELETE请求

<script>

        // delete请求方式同理

        fetch(' GET请求的URL地址与格式',{

            method:'get'

        }).then(function(data){

            return data.text();

        })

        .then(function(data){

            // 这个才是真实的数据

            console.log(data);

        })

## POST请求

    <script>

        fetch(' POST请求的URL地址与格式',{

            method:'post',

            body:'name=lisi&age=200',

            headers:{

                // 这个请求头必须设置，否则body里面的数据传递不过来

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

            }

        }).then(function(data){

            return data.text();

        })

        .then(function(data){

            // 这个才是真实的数据

            console.log(data);

        })

        /\*--------------------------------------------------\*/

        // JSON格式的

        fetch(' POST请求的URL地址与格式',{

            method:'post',

            body:JSON.stringify({

                name:'lisi',

                age:'200'

            }),

            headers:{

                // 这个请求头必须设置，否则body里面的数据传递不过来

                'Content-Type':'application/JSON'

            }

        }).then(function(data){

            return data.text();

        })

        .then(function(data){

            // 这个才是真实的数据

            console.log(data);

        })

    </script>

## PUT请求：修改数据

 <script>

        // 这里的地址要到id

        fetch(' PUT请求的URL地址与格式',{

            method:'put',

            body:JSON.stringify({

                'name':'lisi',

                'age':'200'

            }),

            headers:{

                // 这个请求头必须设置，否则body里面的数据传递不过来

                'Content-Type':'application/JSON'

            }

        }).then(function(data){

            return data.text();

        })

        .then(function(data){

            // 这个才是真实的数据

            console.log(data);

        })

    </script>

## fetch响应格式

* text():将返回体处理成字符串类型
* json():返回结果和JSON.parse(responseText)一样----就是将字符串转成JSON形式的对象
* <script>
* // 所有的都要提供一个后台接口，后台接口要对应
* fetch(' 请求的URL地址').then(function(data){
* // 这个text得到的是一个Promise实例对象，通过return出去，让下一个then得到最终的数据
* // json()属于fetchAPI的一部分，它返回一个Promise实例对象，用于获取后台数据
* return data.json();
* })
* .then(function(data){
* // 这个才是真实的数据
* console.log(data);
* })

## axios

* axios是基于Promise用于浏览器和node.js的http客户端
* 特性：
  1. 支持浏览器和node.js
  2. 支持Promise
  3. 能拦截请求和响应
  4. 自动转换json数据

1. <script src="axios/axios.js"></script>
2. <script>
3. axios.get('请求的接口').then(function(res){
4. // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
5. // 注意，下面的data属性是固定的，用于获取后台的实际数据
6. console.log(res.data);
7. })
8. </script>

## axios的常用API

* get:查询数据
* post:添加数据
* put:修改数据
* delete：删除数据

## get/delete传递参数

* 通过URL传递参数
* 通过params选项传递参数
* <script src="axios/axios.js"></script>
* <script>
* // delete方法与get相同
* // url长什么样子取决于后台url是什么格式
* // 用'？id='传递id
* axios.get('请求的接口').then(function(res){
* // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
* // 注意，下面的data属性是固定的，用于获取后台的实际数据
* console.log(res.data);
* })
* // ===============================================================================
* // restful的形式，用'/'传id
* axios.get('请求的接口').then(function(res){
* // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
* // 注意，下面的data属性是固定的，用于获取后台的实际数据
* console.log(res.data);
* })
* // =================================================================================
* // 通过params方式进行传参=========对象传参
* axios.get('请求的接口',{
* params:{
* id:234,
* name:'lisi'
* }
* }).then(function(res){
* // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
* // 注意，下面的data属性是固定的，用于获取后台的实际数据
* console.log(res.data);
* })
* </script>
* 如果前台使用params进行传参，那么后台一般采用query来获取数据

## post传递参数

* 通过选项传递参数，默认传递的是json格式的数据
* 通过URLSearchParams传递参数（application/x-www-form-urlencoded）
* <script src="axios/axios.js"></script>
* <script>
* // post与put类似
* // post请求获取数据的后台要使用body
* axios.post('请求的接口',{
* id:234,
* name:'lisi'
* }).then(function(res){
* // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
* // 注意，下面的data属性是固定的，用于获取后台的实际数据
* console.log(res.data);
* })



* // ========================================================
* var params=new URLSearchParams();
* // 将数据填充到这个实例对象中
* // params.append('键'，'值')
* params.append('name','张三');
* params.append('id','123');
* // 将params作为一个整体传入
* axios.post('请求的接口',params).then(function(res){
* // 注意，上面参数res不是实际的数据，如果想要实际的数据，操作如下
* // 注意，下面的data属性是固定的，用于获取后台的实际数据
* console.log(res.data.name);
* })
* </script>

## axios的响应结果的主要特性

* data:实际响应回来的数据，一般响应回来的是json形式的数据
* header:响应头信息
* status：响应状态码---可以用来判断数据的状态与否
  1. 状态码如果是200的话，表示响应的数据是正常的
* statusText：响应状态信息

## axios的全局配置

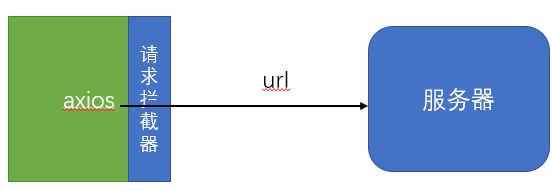
* 超时时间🡺axios.default.timeout=3000;
* 默认地址🡺axios.defaults.baseURL=’http://localhost:3000/app’
* 设置请求🡺axios.defaults.headers[‘mytoken’]=’这里的是字符串’
* <script src="axios/axios.js"></script>
* <script>
* // 配置请求的基准URL地址
* axios.defaults.baseURL='http://localhost:3000/';
* // 配置请求头信息
* axios.defaults.headers['myself']='随意写咯';
* // 把请求的地址简写为axios-json
* axios.get('axios-json').then(function(res){
* console.log(res.data.name);
* })
* </script>

## axios请求拦截器

* 作用：在请求发出之前做一些设置
* <script src="axios/axios.js"></script>
* <script>
* // 添加一个请求拦截器
* axios.interceptors.request.use(function(config){
* // 通过函数的形参config来进行信息的配置
* // 一定要将config   return出来,否则不生效。这些配置在请求发生之前
* config.headers.mytoken='字符串'
* return config;
* },function(err){
* // 这里处理响应错误信息，直接打印出来就可以了
* console.log(err);
* })
* // 下面直接发出请求去请求数据就可以了
* </script>

## axios响应拦截器

* 在获取数据之前对数据进行加工处理



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

<script>

    // 添加一个响应拦截器

    axios.interceptors.response.use(function(res){

        // 形参res表示响应的数据,不是真实的数据。要真实数据就是：res.data

        // 要返回res,否则无效

        res.headers.mytoken='字符串'

        var data=res.data;

        return data;

    },function(err){

        // 这里处理响应错误信息，直接打印出来就可以了

        console.log(err);

    })

    // 下面直接发出请求去请求数据就可以了

    axios.get('网址').then(function(data){

        // 这里的data就是res里面真实的数据

        console.log(data);

    })

## async/await的基本用法

* async/await是ES7引入的新方法，可以更加方便的进行异步操作
* async关键字用于函数上（async函数的返回值是Promise实例对象）
* await关键字用于函数中（await可以得到异步的结果）
* <script src="axios/axios.js"></script>
* <script>
* async function queryData(res){
* // await得到的结果，await后面可以是一个promise实例对象
* const ret=await axios.get('/data');
* // 通过返回值得到异步的结果
* return ret;
* }
* queryData.then(ret=>{
* console.log(ret);
* })

## async/await处理多个异步请求

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

    <script>

        async function queryData(id){

        //await是用来得到异步的结果

        // 用info来接收第一次异步请求的结果

        const info=await axios.get('/async1');

        // 将info作为第二个请求的参数

        const ret=await axios.get('async2?info='+info.data);

        // 这里直接返回明确的值

        return ret.data;

    }

    // 返回的是ret.data所以，用then来得到return出来最终的结果

    queryData().then(ret=>{

        console.log(ret);

    })

# 路由

## 后端路由

* 作用：更具不同的用户URL请求，返回不同的内容
* 后端路由根据不同的URL地址分发不同的资源
* SPA（单页面应用程序）：整个网页只有一个页面，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的过程中，最核心的技术就是前端路由

## 前端路由

* 作用：根据不同的用户事件，显示不同的用户内容
* 本质：用户事件与事件处理函数之间的对应关系
* 前端路由负责事件监听，触发事件后，通过事件函数渲染不同内容

## 实现简易前端路由

* 基于URL的hash实现，点击菜单的实惠改变URL的hash值，根据hash的变化控制组件的切换
* <!-- 被vue实例控制的div区域 -->
* <div id="app">
* <!-- 切换组件的超链接 -->
* <a href="#/zhuye">主页</a>
* <a href="#/keji">科技</a>
* <a href="#/caijing">财经</a>
* <a href="#/yule">娱乐</a>

* <!-- 根据 :is属性指定组件的名称，把对应组件渲染到component标签所在位置，is属性绑定的是哪个属性的名称，就把对应的组件替换到component里面 -->
* <!-- 可以把component标签当成组件的占位符 -->
* <!-- 当点击上面的连接，不同的组件在component中展示 -->
* <!-- ：代表属性绑定,后面是一个字符串 ，就是组件的名称-->
* <component :is="componentName "></component>
* </div>
* <script src="lib/vue.js"></script>
* <script>
* //    region定义要被切换的四个组件
* // 主页组件
* const zhuye={
* template:'<h1>主页信息</h1>'
* }
* // 科技组件
* const keji={
* template:'<h1>科技信息</h1>'
* }
* // 财经组件
* const caijing={
* template:'<h1>财经信息</h1>'
* }
* // 娱乐组件
* const yule={
* template:'<h1>娱乐信息</h1>'
* }
* const vm=new Vue({
* el:'#app',
* data:{
* componentName:'keji',
* },
* components:{
* //将上面四个组件注册为vm的私有子组件
* zhuye,
* keji,
* caijing,
* yule,
* }
* });
* // 监听window的onhashchange事件，根据获取到的最新的hash值，切换要显示的组件的名称
* window.onhashchange=function(){
* // 通过location.hash获取到最新的hash值
* console.log(location.hash);      //这个输出的值是  #/keji...
* // 用slice（1）表示从下表为1的地方开始截取字符串，所以，下面的这个是  /keji
* switch(location.hash.slice(1)){
* // 根据hash值进行如下操作
* case '/zhuye':vm.componentName='zhuye';
* break;
* case '/keji':vm.componentName='keji';
* break;
* case '/caijing':vm.componentName='caijing';
* break;
* case '/yule':vm.componentName='yule';
* }
* }
* </script>

## Vue Router

* Vue Router 是vue.js官方的路由管理器
* 官网：<https://router.vue.js.org/zh/>
* Vue router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：
  1. 支持html5历史模式和hash模式
  2. 支持嵌套路由
  3. 支持路由参数
  4. 支持编程式路由
  5. 支持命名路由…

## Vue Router基本使用步骤

* 引入相关库文件
  1. 按照次序引入，先引入vue.js🡺为全局window对象挂载Vue构造函数
  2. 再引入vue-router.js🡺为全局window挂载VueRouter构造函数
* 添加路由链接
* <router-link to="/uer" >User</router-link>
  1. router-link是vue中提供标签，默认会被渲染为a标签
  2. to属性默认会被渲染为href属性
  3. to属性的值默认会被渲染为#开头的hash值
* 添加路由填充位🡺也叫vue占位符
* <router-view></router-view>
  1. 将来通过路由规则匹配到的组件，将会被渲染到router-view所在位置
* 定义路由组件
  1. 有多少个路由链接定义多少个路由组件
* 配置路由规则并创建路由实例
* // 创建路由实例对象
* const router=new VueRouter({
* // 提供一个数组，里卖弄包含所有路由规则
* // path的值与to属性的值一一对应
* // component的值与组件名称对应,只接受组件对象，不接收字符串
* routes: [
* {path:'/user',component:User},
* {path:'/register',component:Register}
* ]
* 把路由挂载到Vue根实例中

## 整体代码

    <!-- 引包 -->

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

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

    <div id="app">

        <!-- 添加路由连接 -->

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

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

        <!-- 路由占位符 -->

        <router-view></router-view>

    </div>

    <script>

        // 创建组件

        const User={

            template:'<h1>User组件</h1>',

        };

        const Register={

            template:'<h1>Register组件</h1>',

        }

        // 创建路由实例对象

        const router=new VueRouter({

            // 提供一个数组，里面包含所有路由规则

            // path的值与to属性的值一一对应

            // component的值与组件名称对应,只接受组件对象，不接收字符串

            routes: [

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

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

            ],

        });

        const vm= new Vue({

            el: '#app',

            data: {

            },

            methods: {

            },

            // 将上面创建的router挂载到vm实例身上

            // 这个router指向 VueRouter这个实例对象的那个router

            // router:router,

            router,

        });

   </script>

## 路由重定向

* 指用户再访问地址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
* 通过路由的redirect属性，指定一个新的路由地址，可以很方便的设置路由的重定向
* routes: [
* // 想要打开页面的时候显示指定页面，要在头部添加如下规则
* // path中的‘/’表示原先的路径，redirect表示新路径
* {path:'/',redirect:'/user'},
* {path:'/user',component:User},
* {path:'/register',component:Register}
* ],

## 嵌套路由用法

* 嵌套路由功能分析
  1. 点击父级路由链接显示模板内容
  2. 模板内容中又有子级路由链接
  3. 点击子级路由链接显示子级模板内容

## 父路由组件模板

* 1. 添加父级路由链接
  2. 添加父组件路由填充位
  3. 父级路由通过children属性配置子级路由

## 代码展示

 <!-- 引包 -->

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

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

    <div id="app">

        <!-- 添加路由连接 -->

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

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

        <!-- 路由占位符 -->

        <router-view></router-view>

    </div>

    <script>

        // 创建组件

        const User={

            template:'<h1>User组件</h1>',

        };

        // 注意这里模板字符串的标点符号

        // 因为在Register组件下面创建两个子组件，在template里面操作如下

        // 在这里引入子路由链接和添加填充位

        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>

            <router-view></router-view>

            </div>

            `,

        }

        // 创建register下面的两个子组件

        const Tab1={

            template:'<h2>子组件tab1</h2>'

        }

        const Tab2={

            template:'<h2>子组件tab2</h2>'

        }

        // 创建路由实例对象

        const router=new VueRouter({

            // 提供一个数组，里面包含所有路由规则

            // path的值与to属性的值一一对应

            // component的值与组件名称对应,只接受组件对象，不接收字符串

            routes: [

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

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

                // children属性的数组里面表示子路由规则

                {path:'/register',component:Register,children:[

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

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

                ]}

            ],

        });

        const vm= new Vue({

            el: '#app',

            data: {

            },

            methods: {

            },

            // 将上面创建的router挂载到vm实例身上

            // 这个router指向 VueRouter这个实例对象的那个router

            // router:router,

            router,

        });

   </script>

## 动态路由的基本用法

* 应用场景：当路由规则只有某一部分不一样的时候，我们就可以动态的生成某一部分🡺通过路由参数的模式进行路由匹配
* 动态路径参数，以冒号开头，通过对应的组件就能访问这个参数
* 在对应的组件的模板中通过{{$route.params.绑定的参数}}来获取某个组件对应的参数值。

## 代码演示

 <!-- 引包 -->

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

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

    <div id="app">

        <!-- 添加路由连接 -->

        <router-link to="/user/1" >User</router-link>

        <router-link to="/user/2" >User</router-link>

        <router-link to="/user/3" >User</router-link>

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

        <!-- 路由占位符 -->

        <router-view></router-view>

    </div>

    <script>

        // 创建组件

        const User={

            // 点击组件的时候，通过{{$route.params.id}}这个路由参数对象获取匹配到的id

            template:'<h1>User组件ID是{{$route.params.id}}</h1>',

        };

        const Register={

            template:'<h1>Register组件</h1>',

        }

        // 创建路由实例对象

        const router=new VueRouter({

            // 提供一个数组，里面包含所有路由规则

            // path的值与to属性的值一一对应

            // component的值与组件名称对应,只接受组件对象，不接收字符串

            routes: [

                // 这里加一个冒号就是动态路由参数

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

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

            ],

        });

        const vm= new Vue({

            el: '#app',

            data: {

            },

            methods: {

            },

            // 将上面创建的router挂载到vm实例身上

            // 这个router指向 VueRouter这个实例对象的那个router

            // router:router,

            router,

        });

   </script>

## 路由组件传递参数

* props是布尔值
* <!-- 引包 -->
* <script src="lib/vue.js"></script>
* <script src="vuerouter/vue-router.js"></script>
* <div id="app">
* <!-- 添加路由连接 -->
* <router-link to="/user/1" >User</router-link>
* <router-link to="/user/2" >User</router-link>
* <router-link to="/user/3" >User</router-link>
* <router-link to="/register">Register</router-link>
* <!-- 路由占位符 -->
* <router-view></router-view>
* </div>
* <script>
* // 创建组件
* const User={
* // user组件想访问id的时候，直接用props来接收参数,在模板里面就可以直接使用
* props:['id'],
* template:'<h1>User组件ID是{{id}}</h1>',
* };
* const Register={
* template:'<h1>Register组件</h1>',
* }
* // 创建路由实例对象
* const router=new VueRouter({
* // 提供一个数组，里面包含所有路由规则
* // path的值与to属性的值一一对应
* // component的值与组件名称对应,只接受组件对象，不接收字符串
* routes: [
* // 这里加一个冒号就是动态路由参数，后面的props设为true
* {path:'/user/:id',component:User,props:true},
* {path:'/register',component:Register}
* ],
* });
* const vm= new Vue({
* el: '#app',
* data: {
* },
* methods: {
* },
* // 将上面创建的router挂载到vm实例身上
* // 这个router指向 VueRouter这个实例对象的那个router
* // router:router,
* router,
* });
* </script>
* props是对象类型
* <!-- 引包 -->
* <script src="lib/vue.js"></script>
* <script src="vuerouter/vue-router.js"></script>
* <div id="app">
* <!-- 添加路由连接 -->
* <router-link to="/user/1" >User</router-link>
* <router-link to="/user/2" >User</router-link>
* <router-link to="/user/3" >User</router-link>
* <router-link to="/register">Register</router-link>
* <!-- 路由占位符 -->
* <router-view></router-view>
* </div>
* <script>
* // 创建组件
* const User={
* // 路径里面的props对象里面有什么数据，那么组件中才能接收什么数据
* props:['name','age'],
* // 这里的这个id是没用的
* template:'<h1>User组件ID是{{id}}{{name+---+age}}</h1>',
* };
* const Register={
* template:'<h1>Register组件</h1>',
* }
* // 创建路由实例对象
* const router=new VueRouter({
* // 提供一个数组，里面包含所有路由规则
* // path的值与to属性的值一一对应
* // component的值与组件名称对应,只接受组件对象，不接收字符串
* routes: [
* // 这里加一个冒号就是动态路由参数，后面的props是对象
* {path:'/user/:id',component:User,props:{name:'lisi',age:18}},
* {path:'/register',component:Register}
* ],
* });
* const vm= new Vue({
* el: '#app',
* data: {
* },
* methods: {
* },
* // 将上面创建的router挂载到vm实例身上
* // 这个router指向 VueRouter这个实例对象的那个router
* // router:router,
* router,
* });
* </script>
* props传递一个函数类型的参数——即可以接收动态绑定的值，也可以接收props属性
* <!-- 引包 -->
* <script src="lib/vue.js"></script>
* <script src="vuerouter/vue-router.js"></script>
* <div id="app">
* <!-- 添加路由连接 -->
* <router-link to="/user/1" >User</router-link>
* <router-link to="/user/2" >User</router-link>
* <router-link to="/user/3" >User</router-link>
* <router-link to="/register">Register</router-link>
* <!-- 路由占位符 -->
* <router-view></router-view>
* </div>
* <script>
* // 创建组件
* const User={
* // 路径里面的props对象里面有什么数据，那么组件中才能接收什么数据
* props:['id','name','age'],
* template:'<h1>User组件ID是{{id}},姓名：{{name}},年纪：{{age}}</h1>',
* };
* const Register={
* template:'<h1>Register组件</h1>',
* }
* // 创建路由实例对象
* const router=new VueRouter({
* // 提供一个数组，里面包含所有路由规则
* // path的值与to属性的值一一对应
* // component的值与组件名称对应,只接受组件对象，不接收字符串
* routes: [
* // 这里加一个冒号就是动态路由参数，后面的props里面是一个函数，最后函数返回一个参数对象
* {path:'/user/:id',component:User,props:route=>({name:'lisi',age:18,id:route.params.id})},
* {path:'/register',component:Register}
* ],
* });
* const vm= new Vue({
* el: '#app',
* data: {
* },
* methods: {
* },
* // 将上面创建的router挂载到vm实例身上
* // 这个router指向 VueRouter这个实例对象的那个router
* // router:router,
* router,
* });
* </script>

## vue-router命名路由：

* 为了更加方便的表示路由路径，可以给路由规则起一个别名，即为“命名路由”。

## 命名路由的配置规则

 <!-- 引包 -->

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

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

    <div id="app">

        <!-- 添加路由连接 -->

        <router-link to="/user/1" >User</router-link>

        <router-link to="/user/2" >User</router-link>

        <!-- 命名路由需要在to属性这里绑定一个对象 -->

        <!-- name表示要跳转的路由，params表示要携带的参数，这里的参数要和规则名的一样 -->

        <router-link :to="{name:'user',params:{id:3}}">User</router-link>

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

        <!-- 路由占位符 -->

        <router-view></router-view>

    </div>

    <script>

        // 创建组件

        const User={

            // 路径里面的props对象里面有什么数据，那么组件中才能接收什么数据

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

            template:'<h1>User组件ID是{{id}},姓名：{{name}},年纪：{{age}}</h1>',

        };

        const Register={

            template:'<h1>Register组件</h1>',

        }

        // 创建路由实例对象

        const router=new VueRouter({

            // 提供一个数组，里面包含所有路由规则

            // path的值与to属性的值一一对应

            // component的值与组件名称对应,只接受组件对象，不接收字符串

            routes: [

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

                {

                    // 命名路由

                    // 通过命名路由的name属性可以实现页面的跳转，

                    name:'user',

                    path:'/user/:id',

                    component:User,

                    props:route=>({

                        name:"lisi",

                        age:18,

                        id:route.params.id

                    })

                },

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

            ],

        });

        const vm= new Vue({

            el: '#app',

            data: {},

            methods: {},

            // 将上面创建的router挂载到vm实例身上

            // 这个router指向 VueRouter这个实例对象的那个router

            // router:router,

            router,

        });

   </script>

## 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

## 常用编程式导航API

* this.$router.push()的参数规则

1. 字符串🡺this.$router.push(‘/home’)，hash地址
2. 对象🡺 this.$router.push({path:’/home’})
3. 命名路由参数🡺 this.$router.push({name:’/home’,params:{id:1}})
4. 带查询参数🡺 this.$router.push({path:’/register’,query:{name:’lisi’})

* this.$router.go(n),表示向前（n是正数）或者向后（n是负数）的，n是一个数字，向前几位n就是正几，后退几位就是负几

    <!-- 引包 -->

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

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

    <div id="app">

        <!-- 添加路由连接 -->

        <router-link to="/user/1" >User1</router-link>

        <router-link to="/user/2" >User2</router-link>

        <!-- 命名路由需要在to属性这里绑定一个对象 -->

        <!-- name表示要跳转的路由，params表示要携带的参数，这里的参数要和规则名的一样 -->

        <router-link :to="{name:'user',params:{id:3}}">User3</router-link>

        <router-link to="/user/4" >User4</router-link>

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

        <!-- 路由占位符 -->

        <router-view></router-view>

    </div>

    <script>

        // 创建组件

        const User={

            // 路径里面的props对象里面有什么数据，那么组件中才能接收什么数据

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

            // 注意，有多个标签的时候要有一个根标签

            template:`

                <div>

                    <h1>User组件ID是{{id}},姓名：{{name}},年纪：{{age}}</h1>

                    <button @click='goRegister'> 到注册组件页面</button>

                </div>

                `,

                methods:{

                     goRegister(){

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

                     }

                }

        };

        const Register={

            template:`

               <div>

                    <h1>Register组件</h1>

                    <button @click="goUser1">点击回到User1</button>

               </div>`,

               methods:{

                   goUser1(){

                       this.$router.go(-1);

                   }

               }

        }

        // 创建路由实例对象

        const router=new VueRouter({

            // 提供一个数组，里面包含所有路由规则

            // path的值与to属性的值一一对应

            // component的值与组件名称对应,只接受组件对象，不接收字符串

            routes: [

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

                {

                    // 命名路由

                    // 通过命名路由的name属性可以实现页面的跳转，

                    name:'user',

                    path:'/user/:id',

                    component:User,

                    props:route=>({

                        name:"lisi",

                        age:18,

                        id:route.params.id

                    })

                },

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

            ],

        });

        const vm= new Vue({

            el: '#app',

            data: {},

            methods: {},

            // 将上面创建的router挂载到vm实例身上

            // 这个router指向 VueRouter这个实例对象的那个router

            // router:router,

            router,

        });

   </script>

## 后台管理路由案例

需求：点击左边列表，右边显示信息



用到的路由技术：

* 嵌套路由
* 路由重定向
* 路由传参
* 编程式导航

## 路由导航守卫控制访问权想

* 如果用户没有登录，但是之既然通过URL访问特定页面，需要重新导航到登陆页面
* const router = new VueRouter({ routes })
* // 挂载一个路由导航守卫
* // 为路由对象添加beforeEach导航守卫
* // to代表将要访问的页面的路径,from代表从那个路径跳转而来，next代表一个放行的函数
* // next() 放行，next('/login')强制跳转到login
* router.beforeEach((to, from, next) => {
* // 如果用户访问登录页，直接放行
* if (to.path === '/login') return next()
* // 获取token,并从session Storage中获取到保存的值
* const tokenStr = window.sessionStorage.getItem('token')
* // 没有token强制跳转到登录页
* if (!tokenStr) return next('/login')
* next()
* })
* // 将路由挂载到默认路径
* export default router

# 退出功能的实现原理

* 基于token的方式实现推出比较简单，只需要销毁的token即可，这样后续的请求就不会携带token,必须重新登录生成一个新的token之后才可以访问页面
* logout () {
* // 清空token
* window.sessionStorage.clear()
* // 跳转到登陆页面
* this.$router.push('/login')
* }