vue 一周

数组的迭代方法

for

```
for (var i=0; i<5; i++)
  {
  x=x + "The number is " + i + "<br>";
  }
```

for in

- 语法：for（（val，key） in obj）{}
- obj表示一个对象，val则表示对象中的属性值和方法，key表示属性名和方法名。for…in…循环遍历对象内的属性和方法，

for of（数组，对象，字符串，数组对象）

- 语法：for (var value of myArray) {
  console.log(value);
  }

- 与forEach()不同的是，它可以正确响应break、continue和return语句
-

forEach

- 语法：array.forEach(function(item, index, arr))
- 参数：item 当前对象， index 当前对象索引， arr 当前对象所属的数组对象

map 映射

- map()方法返回一个新数组，数组中的元素为原始数组元素调用函数处理的后值。
- map()方法按照原始数组元素顺序依次处理元素。
- map不会对空数组进行检测
- map不会改变原始数组
- 语法：arr.map(function(currentValue，index，arr),thisValue)

- 参数说明 ：
  - currentValue 必须 当前元素值
  - index 可选 当前元素的索引值
  - arr 可选 当前元素属于的数组对象。
- 例子：

```
<p>点击按钮将数组中的每个元素乘以输入框指定的值，并返回新数组</p>
    <p>
        最小年龄:
        <input type="number" id="age" value="10" />
    </p>
    <button id="btn">点我</button>
    <p id="data">新数组</p>

    <script type="text/javascript">
        var btn = document.getElementById("btn");
        var data = document.getElementById("data");
        var age = document.getElementById("age");
        var numbers = [25,36,121,49];

        function myFunction(num,index,arr){
            console.log('arr',arr);
            console.log('index',index);
            console.log('num',num);
            return num * age.value;
        }

        btn.onclick = function(){
            data.innerHTML = numbers.map(myFunction);
        }
```

some

- 语法：arr.some((item,index)=>{
  console.log(‘q’);
  return index > 2
  });
- 返回值是个 布尔类型
  // 返回值若是 true : 所有的回调函数中，至少有一个 return 的值是 true
  // 返回值若是 false: 所有的回调函数返回值都是 false
- 迭代次数 当碰到return true时，后边的项不再迭代；

every

- 语法：arr.every((item,index)=>{});
- 返回值：只要有一个回调的返回值时false 那么结果就是false
  // 全是true时返回结果才是true；
- 迭代次数 当碰到 return false 时，迭代结束

filter 过滤

- 语法：arr.filter((item,index)=>{
  return index ！== 1 ;
  });
- 返回值是个新数组； 数组中的项决定于 回调函数的return的布尔值
- 若 当前迭代 让return true ;则把当前项放到新数组中；

reduce

- 语法：let res7 = arr.reduce((prev,next)=>{
  return prev + next
  })
- arr.reduce((prev,next)=>{
  console.log(prev,next);
  // 本次的prev 就是 上一次的回调的 return值
  // reduce 有两个参数 ；第一个参数是个回调函数 ；第二个参数可以不写；若写了，则是回调函数的第一个 prev 值；不写的话，回调函数的第一个 prev 是数组中的第一项
  // reduce的返回值是 最后一次回调的 return值
  return next
  })

vue全家桶 vue-router vuex vue-cli

安装包的三种方式

- 1、npm install –》简写 npm i
- 2、cnpm install –》 简写 cnpm i
  - 使用cnpm 的前提是 全局安装了 cnpm
  - 安装命令 npm install -g cnpm –registry=https://registry.npm.taobao.org
- 3、yarn add 包名
  - 使用这个而命令的前提是全局安装了 yarn
  - 安装命令 npm i -g yarn

vue 语法

```
<div id="app">
    <!-- 小胡子
        这里边可以直接写变量
        也可以写表达式
        但是不能写JS语句
     -->
    <!--name = '珠峰培训2018'  这个表达式的返回结果是 '珠峰培训2018'-->

    {{name = '珠峰培训2018'}}
    {{age && 'qqqq'}}
    <!-- 赋值表达式 是有返回值的  返回值就是 赋给变量的 那个值 -->
    <!-- 两次赋值 直接报错？？？ -->
    <h2>{{age > 3 ? name='qqq' : name='www'}}</h2>
    {{name}}
</div>
```

```
let vm = new Vue({
        el:'div',// 决定哪个元素是 vue的html 模板 ;值就是一个css 选择器；但是只对第一个元素起作用
        data:{
            name:"珠峰培训2018"
        }
})
```

绑定 html模板的方法

- 1、el
- 2、$mount
- 3、直接写 template 属性

用法结构

```
let vm = new Vue({
        el:"#app",//需要操作的对象
        data:{//定义数据
            ary:[],
            name:""
        },
```

```
            created(){//钩子函数
                //实例被创建
                //this是指向当前这个实例
                this.getData();
            },
            methods:{//定义函数
                getData(){
    //    axios请求数据              axios.get("./data.json").then((data)=>
    {
                        console.log(data);
                        this.ary = data || [];
                    }).catch((err)=>{
                        console.log(err);
                    })
                }
            },
            filters:{
                //过滤器
                money(val){//val 就是 管道符| 前边的值
                    return (val/100).toFixed(2)
                }
            }
        })
```

指令

- v-text 等同于{{}} v-text = “name”
- v-html 可以渲染字符串中的标签 v-html = “name”
- v-once 该标签只渲染一次
- v-cloak 解决小胡子显示的问题 结合css属性选择器用
- v-pre 有这个属性的元素 vue不会对元素内的内容编译
- v-model 输入框内容

改变视图内容

- 1.自定义一个无关变量，由这个变量触发视图更新
- 2.创建一个新对象，整个替换
- 3.提前写死（把需要的变量都写全）
- 4.vm.$set(targe,key,value)
- 想要触发视图更新 两个前提：1.该属性有get和set方法 2.该属性在html页面中有用到

v-for

- 是循环指定元素，属性用在哪个标签上就循环换哪个标签
- 可以循环 数组 对象 字符串 数字
- 例子：

```
<ul>
    <li v-for="(val,index) in ary">
        {{val}}{{`索引${index}`}}
    </li>
    <li v-for="(val,key,index) in obj">{{val}}{{`键${key}`}}{{`索引${index}`}}</li>
    <li v-for="val in 10">number:{{val}}</li>
</ul>
```

事件

- keyup.enter 键盘监听事件
  键盘事件中常用键：

  keydown.enter=’show()’ 回车执行

  keydown.up=’show()’ 上键执行

  keydown.down=’show()’ 下键执行
  @keydown.left=’show()’ 左键执行
  @keydown.right=’show()’ 右键执行
- v-on
  - click , mouseenter , mouseover , mouseleave

```
<!--<button v-on:click="fn">按钮</button>-->
<button @click="fn2">按钮1</button>
<!--不加括号默认把事件对象e传给对应的函数-->
<button @click="fn2()">按钮2</button>
<!--加括号时，括号里些什么就给函数传递什么参数，不会默认传事件对象e-->
```

```
let vm = new Vue({
    el:"#app",//优先级高于$mount
    data:{//data里的东西最终都挂在实例vm上
        name:"珠峰",
        /*fn(){//this指向window
            console.log(this);
        }*/
    },
    methods:{//this指向实例vm
        //methods中的属性名不能和data中的属性名重复
        fn2(){
            //console.log(this);
            console.log(arguments);
        }
    }
```

```
})
```

axios 请求数据

- axios 的get 和 post 怎么用？怎么传参 即可
- axios.defaults.baseURL = ” // 设置基础路径
- axios.get(url,{params:参数对象}).then((data)=>{}).catch((err)=>{console.log(err)})
- axios.post(url,参数对象).then().catch()

//请求拦截器
axios.interceptors.request.use((config)=>{
config.data.pc = true;
return config
},(err)=>{
return Promise.reject(err)
})
//响应拦截器
axios.interceptors.response.use((res)=>{
return res.data
},(err)=>{
return Promise.reject(err)
})

computed 计算属性

```
computed:{
// 计算属性
            name2(){
// 简写方式
// 只要name 不发生改变 那么 本函数就永远不会执行；
// DOM模板中使用的一直都是 name2的缓存值
// name2 完全依赖于 name;
                return this.name.split('').reverse().join('');
            },
            name3(){
//name3 是否依赖于 this.name
// 只要这个函数中用某些变量； 那么这个计算属性就依赖于这些变量
                return 123;
            },
            name4(){
                let a = this.str+'str';
                let b = this.name + 'name';
```

```
                return a+b;
            },
            name5:{
// 这个时全写的 内容 有一个get函数 和 一个set函数
// 简写只相当于有一个get函数
// 着两个函数 可以只写一个 get; 但是不能只写一个set
                get(){
//get 的return值 就是 nam5的值
                    return this.name
                },
                set(val){
 // val 就是设置给 name5的值
                    console.log(val,'set')
                }
}
```

watch 侦听

```
    watch:{
// 侦听 data中的name属性
            name(cur,prev){
//cur  代表新值,prev 代表老值只有新旧两次值不一样时，才能触发该函数
                clearTimeout(this.timer);
                this.timer = setTimeout(()=>{
                    this.str = cur + '123'
                },1000);
this.str2 = cur.split('').reverse().join('');
            },
            question(cur,prev){
                clearTimeout(this.timer2);
                this.timer2 = setTimeout(()=>{
                    axios.get('https://yesno.wtf/api')
                        .then((data)=>{
                            console.log(data);
                            this.answer = data.data.answer;
                            this.pic = data.data.image;
                        }).catch((err)=>{
                        console.log(err)
                    })
                },1000);
            }
        }
```

v-if / v-show

- v-if 是用来决定 该标签是否要加载的 true 代表这个元素要加载；false 代表这个元素不加载；
- v-if v-else v-else-if 这些指令使用时 中间不能掺杂不相干的元素
- 用的时候 所在元素需要紧挨着
- v-show 是控制这个标签是否显示；控制的是CSS属性
- v-if 是控制这个标签要不要加载的;
  `<h2 v-show="isShow">v____show</h2>`

transition 过度动画

```
.fade-enter-active,.fade-leave-active{
/*整个过渡期间的类名下的样式*/}
.fade-enter{
/*动画开始第一帧时的样式*/}
.fade-enter-to{
/*动画的最后一帧时的样式；可以理解成最终的显示状态*/}
```

v-bind 用法

:class=” bg’”
:class=” bg==’bg1’ ? ‘bg2’ : ‘bg1’”
:class=”{bg1:flag, bg2:!flag}”
:class=”[bg]”
:class=”ary”

- {bg1:flag, bg2:!flag} ：就是JS的普通对象
  对象的这种写法 属性名是要添加的类名； 属性值 是布尔值，决定要不要添加这个类名
- 数组的用法 把数组中的每一项都添加给该元素的类名

:style=”{background:’#ccc’,color:col}”
:style=”obj1”
:style=”{…obj1,…obj2}”
:style=”[obj1,obj2]”

绑定style的这种形式 对象的形式
属性名 是要加的 css属性
属性值 是要加的 css属性值 ；可以是变量；但是变量要有对应的值

事件的修饰符

- stop 阻止冒泡
  - click.stop=”fn”
- prevent 阻止默认行为
  - click.prevent=”fn” a标签的默认行为
- self 只有点击绑定的元素才能触发
  - `<div class="center" @click.self="fn1">
    <div class="inner" @click="fn2">

    `
- once 绑定的元素只能点击一次
  - click.once=”fn”
- capture @click.capture —》 addEventListener(‘click’,fn,true) 事件要在捕获阶段触发
- passive 针对onscroll事件 不加这个修饰符；它是先执行事件；再看事件中有没有组织默认行为；没有阻止才会触发默认行为。 加这个修饰符， 他就不管事件中是否有阻止默认，都会直接出发默认行为；
- v-model.number=”name” 可以改变数据格式
- v-model.trim=”name”
  - trim 去除首位空格；即使在输入框看着输入了空格；但是数据层的数据仍然是没有空格的数据

自定义指令

```
<h1 v-color-red="'red'">{{name}}</h1>
<h1 v-color-red>{{name}}</h1>
directives:{
// 自定义指令
colorRed(ele,obj){
ele.style.color = obj.value || ‘red’;
}
}
```

深度watch

v-model=”obj.name”

```
data:{
        name:"123",
        obj:{
           name:456
        }
},
watch:{
       name(cur,prev){
//cur 改变之前的值，prev 改变之后的值
       },
       obj:{
          handler(cur,prev){
// handler 单词是固定的
       },
       deep:true
}
```

生命周期

钩子函数

- beforeCreate created
- beforeMount mounted
- beforeUpdate updated
- beforeDestroy destroyed

beforecreated：el和data 并未初始化
created:完成了 data数据的初始化 一般写接口请求，el没有
beforeMount：完成了 el 和 data 初始化
mounted ：完成挂载
所有生命周期钩子函数中的this都指向vm实例
methods/computed/watch 中的this 都是指向当前实例的
directives/filters 中的函数中的this 都是指向当前实例的
data里的值被修改后，将会触发update的操作。
执行了destroy操作，后续就不再受vue控制了（改值不起作用）,但之前渲染的元素在页面上还存在

```
beforeCreate(){
console.group('beforeCreate 初始化前el和data 并未初始化》');
},
created(){
console.group('created 初始化完成data数据的初始化  一般写接口请求，el没有》');
```

```
},
beforeMount(){
console.group('beforeMount 挂载前，完成了 el 和 data 初始化 》');
},
mounted(){
console.group('mounted 挂载完成===============》');
},
beforeUpdate() {
console.group('beforeUpdate 更新前===============》');
},
updated(){
//视图更新时触发
console.group('updated 更新完成===============》');
},
beforeDestroy() {
console.group('beforeDestroy 销毁前===============》');
},
destroyed(){
console.group('destroyed 销毁完成===============》');
}
```

钩子函数用处

- beforecreate : 举个栗子：可以在这加个loading事件
- created ：在这结束loading，还做一些初始化，实现函数自执行
- mounted： 在这发起后端请求，拿回数据，配合路由钩子做一些事情
- beforeDestory： 你确认删除XX吗？
- destoryed ：当前组件已被删除，清空相关内容
  当然，还有更多，继续探索中……

获取元素

- ref
- this.$refs

```
<ul>
        <li v-for="i in n" ref="li">{{i}}</li>
</ul>
```

```
mounted(){
// 获取元素的操作 一般都在 mounted 函数中
// 通过ref 获取元素 ；若是写死的元素；则只能获取最下边的那个元素
// 若是通过v-for循环出来的；那么都能获取到
```

```
// DOM 的渲染是异步的
// console.log(this.$refs)
}
```

- created:在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。
  - 在created的时候，视图中的html并没有渲染出来，所以此时如果直接去操作html的dom节点，一定找不到相关的元素
- mounted:在模板渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作。
  - 在mounted中，由于此时html已经渲染出来了，所以可以直接操作dom节点

dom的异步渲染 $nextTick

```
<ul>
<li v-for="i in n" ref="a">{{i}}</li>
</ul>
```

```
vm.n = 5;
vm.$nextTick(()=>{
console.log(vm.$refs.a)
})
```

组件

- 组件里的data都是函数式的 `data(){}`

定义全局组件

定义

template id=”myName”

```
<template id="myName">
    <div>
        <h1>这是个全局组件</h1>
        {{name}}
        <!--<son></son>-->//子组件
    </div>
    <!--<div></div>--> <!--只能有一个根元素-->
</template>
```

创建

Vue.component

```
Vue.component('my-name',{
        template:"#myName",
        data(){//data要定义成函数
            return{
                name:"珠峰"
            }
        },
        components:{
            son //son:son
        }
    });
```

使用

```
<my-name></my-name>
```

自定义组件

```
<template id="myDialog">
<div></div>
</template>
```

使用

```
<my-dialog :title="til" v-show="flag" @close="close"></my-dialog>
```

定义

```
<template id="myDialog">
    <div class="bgBox">
        <div class="contentBox">
            <h1>{{title}}</h1>
            <div class="bodyBox">
                这是中间部分
            </div>
```

```
            <button @click="close">关闭</button>
        </div>
    </div>
</template>
```

创建

```
let myDialog = {
        template:"#myDialog",
        props:["title"],//接收父组件数据
        methods:{
            close(){
                this.$emit('close')
            }
        }
    };
```

注册

```
let vm = new Vue({
    el:"#app",
    data:{

    },
    components:{
        myDialog//注册自定义组件
    }
})
```

父组件子组件之间的数据传递

父传子

- 通过 子组件自定义属性； 子组件用过 props接收
  `<son :qqq="父组件数据"></son>`

```
let son = {
        template:"<h1>这是个父组件数据{{name}}</h1>",
        props:['qqq'],//接收父组件的数据
        data(){
            return{
```

```
                name:this.qqq//把父组件的数据赋值给自己
            }
        }
    };
```

子传父

- 父组件调用子组件数据
- 通过 自定义事件；
- 子组件用$emit 触发父组件对应的事件；

```
<my-name @zzz="change"></my-name>//子组件添加自定义事件
```

```
Vue.component('my-name',{
        template:"#myName",
        data(){
            return{
                name:'666'
            }
        },
        methods:{
            fn(){
 this.$emit('zzz',this.name,1234)//$emit把自定义事件和相关数据触发父组件
的事件
            }
        }
    });
```

```
let vm = new Vue({
        el:"#app",
        data:{
            aaa:"珠峰"
        },
        methods:{
            change(name,n){//父组件定义函数接收子组件的数据做相应的事情
                this.aaa = name;
            }
        }
    })
```

Dialog模态框组件

slot 插槽

在子组件使用的地方 用子组件包含的内容；想把这部分内容加入到
子组件的 DOM模板中； 需要使用 slot;
包含的内容标签上 可以添加 slot='xxx'行内属性
在子组件模板中 使用`<slot name='xxx'></slot>`获取 子组件包含的对应内容； name 可以不写 ；默认是default

- 匿名slot

```
<div>
    <h2>我是子组件的标题</h2>
    <slot></slot>    /*这里插入父组件在引用子组件内部的内容*/
</div>
```

- 有名slot
  - 子组件模版

```
<div>
    <slot name="header"></slot>
    <slot name="footer"></slot>
    <slot></slot>
</div>
```

```
- 父组件模板
```

```
<my-component>
    <p>Lack of anonymous slot, this paragraph will not shown.</p>
    <p slot="footer">Here comes the footer content</p>
    <p slot="header">Here comes the header content</p>
</my-component>
```

props传递数据

- 1.传入单数据 `props: ['message']`
- 2.传入多个数据 `props: ['msg','nihao','nisha']`
  - 如果在父组件的模板类添加其他元素或者字符会有：
    ①在最前面加入—每个子组件渲染出来都会在其前面加上
    ②在最后面加入—每个子组件渲染出来都会在其后面加上

③在中间加入—他前面子组件后面加上，后面的子组件后面加上

- 3.动态prop 使用时用v-bind
  - v-bind:my-message=”parentMsg”
- 4.Prop类型绑定

```
props:{
    qqq:{
        // type:String, 限定数据类型是 字符串
        type:[String,Number],// 既可以是字符串，也可以是数字
        default:'默认值',
        // required:true, 是否必须；默认false
    },
    ary:{
        type:Array,
        // default:[]
        default(){
            return []
        },
        validator(val){//validator是验证器
// val 就是传进来的参数或者该项数据的默认值
// 做 自己定义的验证； 需要有返回值 true 或者false
            if(val.length > 3){
                return true
            }else {
                return false
            }
            // console.log(arguments);
        }
    }
}
```

- 5.prop验证
  传入的数据不符合规格，Vue 会发出警告。当组件给其他人使用时，很有用

```
props: {
    msg_null: null,//基础类型检测("null"意思是任何类型都可以)
    msg_string: { //String类型，msg_string必须是定义过的，可以是空字符
串""
        type: String,
        required: true,
    },
    msg_number: {//Number类型，默认值100
        type: Number,
        default: 100,
    },
```

```
        msg_obj: {//Object类型，返回值必须是js对象
            type: Object,
            default: function() {
                return {
                    name: "DarkRanger",
                    age: "18",
                }
            }
        },
        msg_twoway: { //指定这个prop为双向绑定,如果绑定类型不对将抛出一条警告
            type: String,
            twoWay: true,
        },
        msg_validate: { //自定义验证，必须是Number类型，验证规则：大于0
            type: Number,
            validator: function(val) {
                return val > 0;
            }
        },
        msg_number2string: { //将值转为String类型,在设置值之前转换值(1.0.12+)
            coerce: function(val) {
                return val + ""
            }
        },
        msg_obj2json: { //js对象转JSON字符串
            coerce: function(obj) {
                return JSON.stringify(obj);
            }
        },
        msg_json2obj: {//JSON转js对象
            coerce: function(val) {
                return JSON.parse(val);
            }
        },
```

ref

- ref 不仅能获取 DOM ； 也能获取 组件
  `<son ref="qqq"></son>`
  `this.$refs.qqq == that` //$refs用来接收ref

子组件改父组件数据

- 父组件传给子组件一个引用数据类型； 可以直接在子组件中变异 该引用数据类型；这种方式不建议使用；
- 可以通过 $parent 这种方式直接找到父组件这个实例，进而去修改父组件中的相关内容
  - this.$parent.arr = [333];
  - $parent是父组件vm
- $children[1] 可以找到某个子组件（用来控制子组件）

vue中 关于$emit的用法（子组件改父组件数据）

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。

```
<hello @子组件的自定义事件="父组件methods里的某一函数" />
```

```
chilCall() {//chilCall是子组件methods里的函数、子组件模板要绑定这个函数
this.$emit('子组件的自定义事件', '要传给父组件的子组件数据')
}
```

keep-alive 保持组件不销毁

```
<keep-alive>
    <component :is="type"></component>
</keep-alive>
```

动态组件 component

- component 是内置组件
- 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。
- 动态组件由 vm 实例的属性值 `componentId` 控制
- `<component :is="componentId"></component>` //componentId 是要显示的那个组件

vueRouter

> vue-router是Vue.js官方的路由插件，它和vue.js是深度集成的，适合用于构建单页面应用。vue的单页面应用是基于路由和组件的，路由用于设定访问路径，并将路径和组件映射起来。传统的页面应用，是用一些超链接来实现页面切换和跳转的。在vue-router单页面应用中，则是路径之间的切换，也就是组件的切换。
>
> - 安装：

- npm install vue-router

- 使用：

```
<router-link active-class="qqq" to="/home111" tag="button">首页</rou
ter-link>
<router-link to="/list111" tag="button">列表页</router-link>
<router-view></router-view>
```

- 创建路由映射表

```
let routes = [
    // 路由映射表
    {
    path:'/home',
    name:'home',
    component:home
    },
    {
    path:'/list/:age/:sex',// age sex是属于参数部分的;
    name:'list',
    component:list
    }
]
```

- 创建实例router

```
let router = new VueRouter({//实例
    routes,
    linkActiveClass:'current',  // 统一修改默认选中的类名
    linkExactActiveClass:'current2' // 精确匹配;  路径带着参数完全一样
});
```

- 注册

```
let vm = new Vue({
```

```
    el:"#app",
    router,
    data:{
        obj:{a:12,b:13,c:14}
     }
})
```

`router-link` / `router-view` 都是全局组件 ；这些是 vueRouter定义好的组件

`router-link`是用来控制跳转的
`router-view`是用来控制要显示的组件的
tag属性 可以控制 router-link 转化成的html标签； 默认转成 a标签
router-link-active 当前路由下 对应的标签具有的类名
active-class: 修改默认选中的类名；默认的类名时 router-link-active
路由的来回跳转 会触发组件的销毁；组件的生命周期钩子函数会重新走一遍；
若我们需要组件不销毁时；在 router-view外层套一个 keep-alive标签即可

```
<keep-alive>
    <router-view></router-view>
</keep-alive>
```

- 1、先定义好各个组件
- 2、编写路由映射表
- 3、把路由实例router注册到vm根实例中
- 使用路由切换时 我们不需要进行 组件的注册

path name query params

- path 跳转路径
- name 根path一一对应的一个名字
- query 值是一个对象，里边是要添加到路径上的参数；以?&的形式存在这种方式传参，既可以用path 也可以用 name
- params 值也是个对象，里边的参数(opt)，最终会转化成路径的形式；这种传参，需要映射表中的path 的值 加上 :参数（opt）； 前后这两opt属性名必须保持一致；而且只能用 name 的这种方式；不能用 path;
- 获取参数时；每个组件中 都可以通过 $route 获取相应的参数；
- $route中的存储的是当前路由的所有信息；
    - 不管哪个组件 都能获取到这个属性

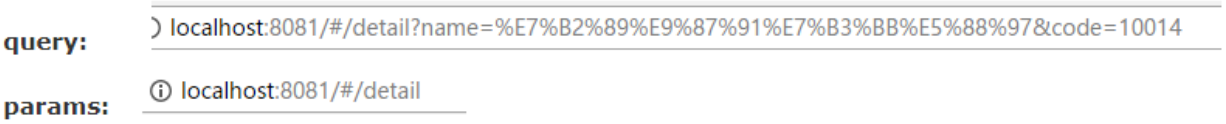
vue 路由传参 params 与 query两种方式的区别

1、用法上的：

- query要用path来引入，
- params要用name来引入，
- 接收参数类似，分别是`this.$route.query.name`和`this.$route.params.name`
- **注意接收参数的时候，是`$route`不是`$router`**

2、展示上的:

- query类似于ajax中get传参
  - 在浏览器地址栏中显示参数
- params类似于post，
  - 在浏览器地址栏中不显示参数



编程式路由切换

编程式的导航，即js控制跳转

- 声名式 `<router-link :to="...">`
- 编程式 `this.$router.push('home')`
  该方法的参数可以是一个字符串路径，或者一个描述地址的对象

```
字符串   router.push('home')

对象   router.push({ path: 'home' })

命名的路由router.push({ name: 'user', params: { userId: 123 }})

带查询参数，变成 /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})
```

如果提供了 path，params 会被忽略 你需要提供路由的 name 或手写完整的带有参数的 path：

```
const userId = 123
router.push({ name: 'user', params: { userId }}) // -> /user/123
router.push({ path: `/user/${userId}` }) // -> /user/123
// 这里的 params 不生效
```

```
router.push({ path: '/user', params: { userId }}) // -> /user
```

插件

- npm install yarn -g
- npm install XXX yarn add XXX
- npm uninstall xxx yarn remove xxx
- npm install -g cnpm –registry=https://registry.npm.taobao.org
- cnpm install XXX

es6模块

- CMD sea.js 和 AMD require.js
- es6模块：（浏览器环境下）es6module
- 引入 import
- 导出 export

node模块：(node环境下) commonJS规范

- 引入：require 内置模块 第三方模块 自定义模块
- 导出：module.exports={} 和 exports.xxx
- npm install babel-cli -g
- npm install babel-loader babel-core babel-preset-es2015 babel-preset-stage-0 style-loader css-loader less less-loader style-loader css-loader file-loader url-loader html-withimg-loader url-loader uglifyjs-webpack-plugin html-webpack-plugin
- Babel 是一个 JavaScript 编译器。
  - npm install babel-cli -g;
  - 配置文件 .babelrc
  - 预设(presets) babel-preset-env es6
  - babel-preset-stage-0 es7
- babel x.js 直接编译
- babel x.js -o x2.js 将编译的结果在x2文件中输出

webpack

- 安装 cnpm install webpack -D
- src （源文件）原始工程文件，上线时不用
- dist （打包编译后的文件,是自动生成的，只需要配置下即可）//上线是使用
- webpack配置在本地得配置下
  “scripts”: {

“build”:”webpack –mode development”
},
npm run build 相当于执行webpack命令
npm run dev 是启动服务器，项目可以自动更新

新建webpack的配置文件 webpack.config.js

解析js文件

webpack不能将es6/es7代码编译成es5代码，如果想要实现这个功能得借助于babel
安装 babel-core babel-loader

解析css文件

style-loader css-loader
css解析完后放在style标签

解析less文件

style-loader css-loader less-loader less

解析图片

file-loader url-loader
url-loader依赖于file-loader
url-loader将图片解析成base64编码格式

html-withimg-loader 处理页面通过路径引入图片的方式

html-webpack-plugin

设置打包后的静态页面，并且将打包后的js文件自动插入到静态页面

webpack-dev-server

创建web服务，当代码发生改变时，自动打包，自动打开页面，自动刷新页面

运行node配置

- 1.npm init 初始化项目
- 2.配置packge.json 文件

```
"scripts": {
    "build": "webpack --mode development",//编译
    "dev":"webpack-dev-server --mode development --open"//自动编译更新并打开端口
  },
```

- npm run build 启动编译
- npm run dev 启动实时编译