与三阶段无关

框架是什么?

框架可以帮助我们完成与业务（功能）无关的事情 并且其中封装了很多现成api方便我们使用

好处：大大提高我们的开发效率

缺点：你要学习一套新的语法

优雅降级和渐进增强

他们两个就是一种开发过程的小方式

优雅降级：在开发的时候先按照最高版本进行开发 然后再逐渐的向下兼容低版本

渐进增强：在开发的时候先按照最底版本进行开发 然后再逐渐的向上兼容高版本

解构赋值

可以让对象和数组快速取值的一个技术

对象解构

数组解构

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // let obj={
        //     name:"xixi",
        //     age:18,
        //     sex:"男"
        // }
```

```
        // 传统的方式 非常麻烦 去多条很费事
        // let xxx=obj.xxx

        // 解构赋值  让对象的值快速取出
        // let {name,age,sex}=obj;


        // console.log(sex)


        // let arr=[11,2222,3333,4444]

        // let [,,xiaohei,xiangzi]=arr
        // console.log(xiaohei)



        let arr=[11,22,[33,[4,[5,6,[7,[0],[8,[9]]]]]]]


        let [,,[,[,[,,[,[xiaoming],[,[]]]]]]]=arr
        console.log(xiaoming)
    </script>
</body>
</html>
```

扩展运算符

vue

vue是什么?

vue就是一个当下最为主流的前端框架 是一个渐进式的 自底向上增量开发的**MVVM**框架

渐进式（只会做职责之内的使用）：vue可以很方便的和现有的第三方框架进行整合 但是vue不会影响其他框架 因为他只会管理自己范围内的使用

自底向上增量开发：先写一个基本的页面 然后再逐渐的向上添加各种功能

MVVM

M --- model 模型==数据==变量

V --- view 视图==页面==用户可以看见的界面

VM --- viewModel 视图模型==用来关联数据与视图之前的桥梁

HelloWord

1.下载vue库文件 npm init -y npm install --save vue@2

2.新建html页面先把vue引用进来

3.编写如下代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <!-- 1.先引用 -->
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
<!-- M       ---    model          模型==数据==变量

V        ---    view             视图==页面==用户可以看见的界面
```

```
VM    ---    viewModel  视图模型==用来关联数据与视图之前的桥梁 -->


<!-- 2创建视图层   V  vue的根容器  根节点  今后把你的vue内容都写在这个根节点中-->
    <div id="demodiv">
        <h1>{{text}}</h1>
        <h1>{{num}}</h1>
        <h1>{{obj.name}}</h1>
        <h1>{{arr[4]}}</h1>
    </div>



    <script>
        // 3.创建vm层  --》 就是vue实例
        new Vue({
            el:"#demodiv",//关联视图
            data:{//4 m层  模型数据
                text:"我是字符串",
                num:18,
                bool:true,
                obj:{
                    name:"xixi",
                    age:18
                },
                arr:[1111,22222,33333,4444,5555]
            }
        })
    </script>
</body>
</html>
```

{{}}--模板语法

在vue中{{}}被称之为模板语法 双花括号赋值法 vue数据插值。。。。 作用：就是可以在双大括号中写入表达式 并且展示在页面中

语法:在你想展示数据的任何位置 {{表达式}} （表达式 通过计算可以返回结果的公式）

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
```

```
</head>
<body>
        <div id="demodiv">
            <h1>{{num}}</h1>
            <!-- 因为{{表达式}}s所以运算符都可以写 -->
            <h1>{{num*2}}</h1>

            <h1>{{bool?"你好":"你坏"}}</h1>
            <!-- 建议不要在{{}}写太复杂的内容 -->
            <h1>{{text.toUpperCase().slice(1,5)}}</h1>

        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{
                    num:666,
                    bool:false,
                    text:"abcdefghiaslkdjaskldhalk"
                }
            })

        </script>


</body>
</html>
```

指令

什么是html标签的属性?

通过写在html开标签中的 使用属性="属性值" 的这些东西 可以扩展的功能

什么是指令?

就是在vue中给html标签添加的带有v-前缀的特殊属性（在vue中 给html标签添加个一些特殊性功能属性）

v-model

作用： 就是给表单元素进行数据的双向绑定

双向绑定

视图改变模型也会改变

模型变视图也会随之改变

语法: <标签 v-model=""值/>

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">
            <h1>v-model</h1>

            <input type="text" v-model="inputval"/>

            <h1>{{inputval}}</h1>

        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{
                   inputval:""
                }
            })

        </script>


</body>
</html>
```

双向绑定的原理

在vue中基于数据劫持与发布者订阅者模式完成的

数据劫持：数据拦截 就是对data中的数据在初始化的时候监听起来(Object.defineProperty来进行监听）当数据改变之后 vm就会知道 在视图改变 他就会通知模型你要修改了 模型改变了也会通知视图改变

发布者订阅者模式：就是一个一对多的关系 发布者就是数据提供者 订阅者就是页面展示的 一个发布者可以对应无数个订阅者 但是发布者改变了 所有订阅者也会改变

v-show

作用：控制dom元素的显示或者隐藏 v-show的显示和隐藏是通过css的dispaly方式来控制的

语法 v-show="布尔值" true显示 false隐藏

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">
            <input type="checkbox" v-model="bool"/>{{bool?"勾选了":"没勾
选"}}

           <h1 v-show="bool">我是站位的</h1>
        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{
                    bool:false
                }
            })

        </script>


</body>
</html>
```

v-on

作用：就是给vue的dom绑定事件的

语法:

传统写法： v-on:你的事件不加on="函数"　　v-on:click=”函数“

简写写法： @你的事件不加on="函数"　　@click="函数"

注意：函数需要写在 与**el　data**同级位置使用**methods**来进行包裹

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">
            <button v-on:click="fun()">点我触发函数</button>
            <button @click="fun()">点我触发函数简写写法</button>
        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{

                },
                // 函数需要写在与el data同级的位置使用methods来进行包裹
                methods:{
                    fun(){
                        console.log("你好")
                    },

                }
            })

        </script>


</body>
</html>
```

注意 函数中怎么使用**data**数据　使用this.变量名

```
    methods:{
                        fun(){
                            // console.log("你好")
                            console.log(this.text)
                        },

                    }
```

v-for

概念： 用来遍历数据 生成页面内容

语法： v-for="(遍历的值,遍历的下标) in 你要遍历的数据"

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">
            <!-- 语法： v-for="(遍历的值,遍历的下标) in 你要遍历的数据"  -->
            <ul>
                <li v-for="(v,i) in arr">
                    {{v}}------{{i}}
                </li>
            </ul>
        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{
                    arr:["ez","Vn","MF","noc"]
                },

                methods:{


                }
```

```
            })


        </script>


</body>
</html>
```

遍历复杂数据

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">
            <!-- 语法： v-for="(遍历的值,遍历的下标) in 你要遍历的数据"  -->
            <ul>
                <li v-for="(v,i) in arr">
                    {{v}}------{{i}}
                </li>
            </ul>

            <hr/>

            <table border="1">
                <tr v-for="(v,i) in obj">
                    <td>{{v.name}}</td>
                    <td>{{v.age}}</td>
                </tr>
            </table>
        </div>

        <script>

            new Vue({
                el:"#demodiv",
                data:{
                    arr:["ez","Vn","MF","noc"],
                    obj:[
                        {name:"xixi1",age:181},
                        {name:"xixi2",age:182},
```

```
                        {name:"xixi3",age:183},
                        {name:"xixi4",age:184},
                        {name:"xixi5",age:185},
                        {name:"xixi6",age:186}
                    ]
                },

                methods:{


                }
            })


        </script>


</body>
</html>
```

v-bind 这个是初学者最容易忘记的 但是很重要

概念: 就是html的属性插入变量

语法:

传统写法: v-bind:html的属性="值"

简写写法: :html的属性="值"

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.min.js"></script>
</head>
<body>
        <div id="demodiv">

            <a v-bind:href="ahref">{{text}}</a>
            <a :href="ahref">{{text}}</a>

        </div>

        <script>
```

```
            new Vue({
                el:"#demodiv",
                data:{
                   text:"点我去百度",
                   ahref:"http://www.baidu.com"
                }
            })

        </script>


</body>
</html>
```

v-if全家桶

v-if

作用：就是对dom元素进行移除和添加

语法：写在开标签中　true添加　false移除

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-if</h1>
            <input type="checkbox" v-model="bool"/>{{bool}}
            <h1 v-show="bool">我是v-show的标签</h1>


            <h1 v-if="bool">我是v-if的标签</h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    bool:true
                },
                methods:{
```

```
                }
            })
        </script>
</body>
</html>
```

v-if与v-show的区别

1.v-show是使用css的方式对dom元素进行显示和隐藏的 在频繁切换的时候效率更高 在初始化的时候对性能的损耗比较高

2.v-if是直接把这个dom元素移除或者是添加 在频繁切换的时候效率比较低 在初始化的时候对性能的损耗比较低

v-else

语法： 必须配合v-if使用 不能单独使用

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-if</h1>
            <input type="checkbox" v-model="bool"/>{{bool}}
            <h1 v-if="bool">请您登录</h1>
            <h1 v-else>欢迎您尊敬的vip</h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    bool:true
                },
                methods:{

                }
            })
```

```
        </script>
</body>
</html>
```

v-else-if

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-else-if</h1>
            <select v-model="text">
                <option value="吃饭">吃饭</option>
                <option value="睡觉">睡觉</option>
                <option value="在吃饭">在吃饭</option>
                <option value="在睡觉">在睡觉</option>

            </select>


            <h1 v-if="text=='吃饭'">我是吃饭的dom</h1>
            <p v-else-if="text=='睡觉'">我是睡觉的dom</p>
            <em v-else-if="text=='在吃饭'">我是在吃饭的dom</em>
            <h2 v-else-if="text=='在睡觉'">我是在睡觉的dom</h2>
            <b v-else>我什么都没有干</b>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    text:""
                },
                methods:{

                }
```

```
            })



        </script>
</body>
</html>
```

v-html

作用：就是把字符串标签插入到页面中

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-html</h1>
            {{text}}
            <div v-html="text">

            </div>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                   text:"<h1>我是一个h1</h1>"
                },
                methods:{

                }
            })
        </script>
</body>
</html>
```

v-once

作用: 一次性插值 数据插入到页面中就不会被改变了

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-once</h1>
            <input type="text" v-model="text">
            <h1>{{text}}</h1>
            <h1 v-once>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
            <h1>{{text}}</h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    text:"我是默认值"
                },
                methods:{

                }
            })
        </script>
</body>
</html>
```

v-text

作用：向dom中插入文本内容 和{{}}作用一样

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>v-text</h1>
            <h1>{{num}}</h1>

            <h1 v-text="num"></h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    num:"你好么么哒！！！！^_!"
                },
                methods:{

                }
            })
        </script>
</body>
</html>
```

屏幕闪烁

当用户的设备和网络比较慢的时候可能就会在页面中吧{{}}全部展现出来 当网络恢复正常之后 有突然间显示ok

使用{{}}模板语法的话就会出现这个问题

解决方式

1.使用v-text

2.使用v-cloak指令　等待vue实例渲染完成之后在进行

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <!-- <script src="./node_modules/vue/dist/vue.js"></script> -->
    <style>
        [v-cloak]{
            display: none;
        }
    </style>
</head>
<body>
        <div id="demoDiv" v-cloak>
            <h1>v-text</h1>
            <h1>{{num}}</h1>

            <h1 v-text="num"></h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    num:"你好么么哒！！！！^_!"
                },
                methods:{

                }
            })
        </script>
</body>
</html>
```

v-if与v-for能同时使用吗

当 v-if 与 v-for 不推荐同时使用，**v-for** 具有比 **v-if** 更高的优先级，这意味着 v-if 将分别重复运行于每个 v-for 循环中

,影响速度

如果必须要使用的话

解决方式

1.使用computed处理数据在便利

2.使用v-show替代v-if

###

watch监听属性

watch是vue实例的一个属性 他的作用就是用来监听data中的数据 当数据变了watch就会触发 从而调用函数处理一些逻辑

语法：写在与el data methods 同级位置

watch:{

你要监听的data数据(newval,oldval){

}

}

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
    <script src="./node_modules/vue/dist/vue.js"></script>
</head>
<body>
        <div id="demoDiv">
            <h1>watch</h1>

            <input type="text" v-model="text">
            <h1>{{text}}</h1>
        </div>

        <script>
            new Vue({
                el:"#demoDiv",
                data:{
                    text:""
                },
                methods:{
```

```
                },
                // watch监听数据
                watch:{
                    text(newval,oldval){
                        console.log(newval+"---"+oldval)
                    }
                }
            })
        </script>
</body>
</html>
```

watch在初始化能触发吗?

不会触发

想再初始化触发watch怎么办?

watch 的一个特点是，最初绑定的时候是不会执行的，要等到 监听的数据 改变时才执行监听。那我们想要一开始就让他最初绑定的时候就执行改怎么办呢?

watch 属性与方法

handler方法

data监听的数据改变触发的回调函数

```
  //text变量改变的时候。handler方法就会触发
 watch:{
    text:{
      handler(){
           console.log("aaaa");
       },
    },
  }
```

immdiate属性

watch默认绑定，页面首次加载时，是不会执行的。只有值发生改变才会执行。

设置immediate为true后，监听会在被监听值初始化的时候就开始，也就页面上的数据还未变化的时候。

```
 watch:{
    text:{
      handler(){
           console.log("aaaa");
       },
       immediate:true //true就表示会立即执行
    },
  }
```

deep 属性

属性 deep，默认值是 false，代表是否深度监听

观察下面的代码

当我们在在输入框中输入数据视图改变obj.name的值时，我们发现是无效的。

受现代 **JavaScript** 的限制，**Vue** 不能检测到对象属性的添加或删除。

如果我们需要监听obj里的属性name的值呢?

```
<template>
  <div>
    <input type="text" v-model="obj.name">
    <h1>{{obj.name}}</h1>
  </div>
</template>

<script>
export default {
  data(){
    return {
      obj:{
        name:"xixi"
      }
    }
  },
  watch:{
    obj:{
      handler(){
           console.log("aaaa");
       },
       immediate:true //true就表示会立即执行
    },

  }
}
</script>
```

```
<style>

</style>
```

这时候deep属性就派上用场了。

deep的意思就是深入观察，监听器会一层层的往下遍历，给对象的所有属性都加上这个监听器，但是这样性能开销就会非常大了，任何修改obj里面任何一个属性都会触发这个监听器里的 handler。

```
<template>
  <div>
    <input type="text" v-model="obj.name">
    <h1>{{obj.name}}</h1>
  </div>
</template>

<script>
export default {
  data(){
    return {
      obj:{
        name:"xixi"
      }
    }
  },
  watch:{

    obj:{
      handler(){
           console.log("aaaa");
       },
       immediate:true, //true就表示会立即执行
       deep:true
    },

  }
}
</script>

<style>

</style>
```

但是这样性能开销就会非常大了 优化，我们可以是使用字符串形式监听

```
<template>
  <div>
```

```
    <input type="text" v-model="obj.name">
    <h1>{{obj.name}}</h1>
  </div>
</template>

<script>
export default {
  data(){
    return {
      obj:{
        name:"xixi"
      }
    }
  },
  watch:{
    // 字符串方式
    "obj.name":{
      handler(){
           console.log("aaaa");
       },
       immediate:true
    },

  }
}
</script>

<style>

</style>
```

vue/cli 脚手架--5x

脚手架就是项目的开发环境 在脚手架中已经把我们需要开发的一切环境已经配置好了 我们字需要直接写业务代码即可

脚手架创建

1全局安装脚手架　npm install -g @vue/cli

2查看版本　　vue --version

上面的两部 除非你重新装系统或者 重新装node了 否则不需要重复

3。创建项目

（3-1）把你的cmd切换到你要创建项目的文件夹中

（3-2）vue create 你的项目名不要中文空格特殊符号

4 cd到项目名下

5启动项目 npm run serve

6 根据提示打开浏览器访问

拿到空项目怎么办?

1.在src下吧components文件夹下的HelloWord。vue删掉

2.在app.vue中删除掉初始化的内容

```
<template>
  <!-- <img alt="Vue logo" src="./assets/logo.png"> -->
  <!-- 3.使用 -->
  <!-- <HelloWorld msg="Welcome to Your Vue.js App"/> -->

  <div></div>
</template>

<script>
// 1引用
// import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    // 2.调用
    // HelloWorld
  }
}
</script>

<style lang="scss">
// #app {
//   font-family: Avenir, Helvetica, Arial, sans-serif;
//   -webkit-font-smoothing: antialiased;
//   -moz-osx-font-smoothing: grayscale;
//   text-align: center;
//   color: #2c3e50;
//   margin-top: 60px;
// }
</style>
```

扩展----项目怎么启动?

1.cd项目下　npm install 下载依赖

2.启动项目 找到项目下的package.json文件 的scripts节点去查看启动命令

注意 所有单词都是npm run 你的配置　唯独 **start**不一样 因为**start**可以不加**run**

```
"scripts": {
    "diaodeyipi": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
```

扩展----自动开启浏览器与端口修改

1.找到vue。config.js文件写入如下内容

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,
  // 设置浏览器自动开启
  devServer:{
    open:true,//设置自动开启
    port:8888,//修改端口
    host:"localhost"
  }
})
```

组件化

组件的基本概念

组件的本质就是自定义标签

组件其实就是把我们的页面差分成一个个的小模块 分开编写 增加开发效率 降低维度难度　复用性更高

组件的创建

在vue中组件使用.vue文件来进行表示 .vue文件叫做单文件组件

在创建组件的时候 我们是在components文件夹中进行创建的

.vue文件基本页面内容

一个.vue文件中 有三个部分

template -------》写html

script -------》写js逻辑

style -------》写css

```
<template>
  <div>
      我是轮播图
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

组件的分类

全局组件---component

有的时候 一个组件在很多个地方都要被重复使用 那么默认情况下 我们使用局部组件的引用调用使用 每次在这样写很麻烦

全局组件 只需要配置一次main.js 那么就可以在当前项目的任意位置直接使用(全局组件慎用因为全组件可能会造成组件命名污染)

```
import Vue from 'vue'
import App from './App.vue'
```

```
// 1.引用
import AllCom from "./components/AllCom.vue"
// 2.配置全局组件
// Vue.component("给你这个全局组件起个名字",你所要对应使用的组件)
Vue.component("AllCom",AllCom)

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

局部组件---components

局部组件 只能在特定区域使用的组件 谁引用的 谁才能用

1.引用

```
<template>

  <div></div>
</template>

<script>
// 1.引用  组件的名字要大写
import Bottombar from "./components/bottombar/index.vue"
import List from "./components/list/index.vue"
import Slider from "./components/slider/index.vue"

export default {
  name: 'App',

}
</script>

<style lang="scss">

</style>
```

2.调用

```
<template>

  <div></div>
</template>

<script>
// 1.引用 组件的名字要大写
import Bottombar from "./components/bottombar/index.vue"
import List from "./components/list/index.vue"
import Slider from "./components/slider/index.vue"

export default {
  name: 'App',//命名空间    给这个组件起个名字
  // 2。调用
  components:{
    // 名字:你引用的组件
    Bottombar,
    List,
    Slider
  }

}
</script>

<style lang="scss">

</style>
```

3.使用

```
<template>

  <div>
    <!-- 1.使用 -->
    <Bottombar></Bottombar>
    <List></List>
    <Slider></Slider>
  </div>

</template>

<script>
// 1.引用 组件的名字要大写
import Bottombar from "./components/BottomBar/BottomBar.vue"
import List from "./components/ListCom/ListCom.vue"
import Slider from "./components/SliderCom/SliderCom.vue"
```

```
export default {
  name: 'App',//命名空间    给这个组件起个名字
  // 2。调用
  components:{
    // 名字:你引用的组件
    Bottombar,
    List,
    Slider
  }

}
</script>

<style lang="scss">

</style>
```

组件样式隔离--scoped

使用scoped属性可以让 当前样式仅对当前组件生效

```
<style scoped>
 div{
    color:green;
  }
</style>
```

之前学的内容怎么用

之前学过 data methods watch 还有一堆堆指令

除了data以外 剩下的都一样

```
<template>
  <div>
      你是轮播图--{{text}}
      <button @click="fun()">点我</button>
  </div>
</template>

<script>
export default {
//  在组件中data是一个函数return一个对象

  data(){
```

```
    return {
      text:"我是字符串",
      num:666,
      bool:true,
      obj:{name:"xixi"}
    }
  },

  methods:{
    fun(){
      console.log("么么哒")
    }
  }
}
</script>

<style scoped>
 div{
    color:green;
  }
</style>
```

vue组件的data为什么是一个函数?

数据以函数返回值形式定义，这样每复用一次组件，就会返回一份新的data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份data，就会造成一个变了全都会变的结果。

父子组件

组件和组件之间相互嵌套

```
父组件
<template>
  <div>
      fufufuffufufufufufu
      <Zicom></Zicom>
  </div>
</template>

<script>
import Zicom from "./ZiCom"
export default {
```

```
    components:{
        Zicom
    }
}
</script>

<style>

</style>
```

子组件

```
<template>
  <div>
      zizizizizzizizizi
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

app.vue

```
<template>
  <div>
    <FuCom/>
  </div>
</template>

<script>
import FuCom from "./components/NobCom/FuCom.vue"

export default {
  name: 'App',
  components:{
    FuCom
  }

}
</script>
```

```
<style lang="scss">

</style>
```

组件传值

父组件的数据 子组件不能直接使用

子组件的数据 父组件也不能直接使用

组件与组件之间是一个完整地 独立地个体 他们之间的数据 默认是不能相互使用的

正向传值---父组件给子组件数据---props

props是**vue**实例的一个属性 他的作用是用来让组件接受外部传递进来的数据

语法:

写在data methods watch 的同级

props:[接收参数1，接受参数2,.....n]

基本props使用

1.在子组件中 使用props来定义接收参数

```
<template>
  <div>
      <!-- 2.使用 -->
      zizizizizizizzi--{{xiaoming}}---{{xiaohong}}
  </div>
</template>

<script>
export default {
    // 1.定义接收参数
    props:["xiaoming","xiaohong"]
}
</script>

<style>

</style>
```

2父组件 开始给子组件传递参数

```
<template>
  <div>
      fuffufufufufufuf
      <!-- 在子组件被调用的地方  把props的参数当成属性进行传值 -->
      <Zicom :xiaoming="text" :xiaohong="num"/>
  </div>
</template>

<script>
import Zicom from "./ZiCom.vue"
export default {
    components:{
        Zicom
    },
    data(){
        return {
            text:"我是字符串",
            num:18
        }
    }
}
</script>

<style>

</style>
```

props验证语法

在上面的例子中 大家会发现 我们给子组件传递任意数据类型都可以 但是如果我们想限制父组件给子组件传递的数据类型时候 那么就要使用props验证

语法:

props:{

  接受参数:{

    type：数据类型

  },

  接受参数2:{

    type：数据类型

  },

}

子组件

```
<template>
  <div>
      <!-- 2.使用 -->
      zizizizizizizzi--{{xiaoming}}---{{xiaohong}}
  </div>
</template>

<script>
export default {
    // 1.定义接收参数
    // props:["xiaoming","xiaohong"]

    // props验证
    props:{
        xiaoming:{
            type:String
        },
        xiaohong:{
            type:Boolean
        }
    }
}
</script>

<style>

</style>
```

逆向传值--- 子组件给父组件数据

$emit自定义事件

逆向传值默认是不被允许的 我们需要使用一些歪门邪道

需要使用自定义事件来完成 **$emit()**

注意：很多同学后期在被问到$emit是什么的时候 总会回到他是逆向传值 这个回答是错的 因为$emit是自定义事件 而逆向传值只是他能完成的一个小功能

1.在子组件中必须必须通过事件来触发自定义事件的抛出

```
<template>
  <div>
      zzizizizizizizizi
      <!-- 1.通过事件调用一个自定义事件的创建 -->
      <button @click="fun()">点我逆向传值</button>
```

```
  </div>
</template>

<script>
export default {
    data(){
        return {
            zitext:"我是子组件的数据么么哒！！！！！"
        }
    },
    methods:{
        fun(){
            // 2.在函数中使用$emit来创建一个自定义事件
            // this.$emit("自定义事件的名字",你要传递的数据)
            this.$emit("zipao",this.zitext)
        }
    }
}
</script>

<style>

</style>
```

2.父组件接收

```
<template>
  <div>
      fufufufufufufuf
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <!-- 在子组件被调用的时候  得到刚才的自定义事件 函数不加（）-->
      <Zicom @zipao="fun"/>
  </div>
</template>

<script>
import Zicom from "./ZiCom.vue"
export default {
    components:{
        Zicom
    },
    methods:{
```

```
            // 有个形参   这个形参接收的就是刚才自定义事件的第二个参数
            fun(val){
                console.log(val)
            }
        }
    }
</script>

<style>

</style>
```

ref

把ref绑定到子组件身上 那么就可以得到当前这个子组件的所有信息 包含他的data数据 从而完成了逆向传值

同胞传值--- 兄弟组件（有一个共同的父组件）传值

跨层级传值 --- 爷爷组件给孙子组件---vuex

因为vue中是单向数据流（组件与组件之间的数据传递只能是单向一层一层的进行传值）那么在多层级传值的时候如果我们一层一层的传值 非常麻烦

vuex是什么?

vuex就是在vue中的统一状态（数据）管理工具

vuex创建

在创建的时候选择vuex即可

vuex5大属性---state---数据源

state属性的作用 就是在vuex中存放数据的 我们今后在vuex的所有数据都写在state中

```
import Vue from 'vue'
```

```
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {//就是数据源  存放数据的地方
    text:"我是字符串",
    num:18,
    bool:true,
    arr:[1111,2222,3333],
    obj:{
      name:"xixi",
      age:18
    }
  },
  getters: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

使用state的数据

因为vuex是统一状态管理 所以在项目下的任何数据 都可以直接使用state的数据

语法： this.$store.state.xxx

方式1 直接读取

```
<template>
  <div class="about">
    <h1>This is an about page---{{this.$store.state.text}}</h1>
  </div>
</template>
```

方式2 使用计算属性间接读取

```
<template>
  <div class="about">
    <h1>This is an about page---{{this.$store.state.text}}</h1>
    <h1>计算属性方式-- {{newbool}}</h1>
  </div>
```

```
</template>
<script>
export default{
    computed:{
      newbool(){
        return this.$store.state.bool
      }
    }
}
</script>
```

vuex5大属性---module---模块

随着项目的体积逐渐增大 那么变量与今后的其他操纵就会在vuex的文件中 增多 导致这个文件中的内容原来越冗余 后期也几乎无法维护

为了避免上述情况 所以我们可以使用vuex的模块把内容进行拆分

1.在store文件夹下创建文件夹用来存放我们拆分的模块

2.创建模块文件 写入如下内容

可以直接从store下的index里面复制过来

```
let aboutm={


    state: {//就是数据源   存放数据的地方
        text:"我是字符串",

        bool:true,

      },
      getters: {
      },
      mutations: {
      },
      actions: {
      },


}

// 必须暴露
```

```
export default aboutm
```

3.在store下的index.js 中关联并使用我们创建的模块

```
import Vue from 'vue'
import Vuex from 'vuex'
// 1.引用模块
import homem from "./modules/homem.js"
import aboutm from "./modules/aboutm.js"

Vue.use(Vuex)

export default new Vuex.Store({

  modules: {//2配置模块
    homem,
    aboutm
  }
})
```

4,如果vuex被拆成了模块的话 那么我们要使用数据 必须使用

this.$store.state.模块名.xxxx

```
<template>
  <div class="about">
    <h1>This is an about page---{{this.$store.state.aboutm.text}}</h1>
    <h1>计算属性方式-- {{newbool}}</h1>
    <h1>{{this.$store.state.homem.num}}</h1>
  </div>
</template>
<script>
export default{
    computed:{
      newbool(){
        return this.$store.state.aboutm.bool
      }
    }
}
</script>
```

vuex5大属性---mutations---修改数据的

mutations的作用就是在vuex中修改state的 如果想修改state的数据必须使用mutations来修改

mutations是一个属性 这个属性中包含的是一个个修改的函数 **mutaitons**想使用 那么我们需要在组件中使用**commit**（）来进行调用

```
let aboutm={


    state: {//就是数据源  存放数据的地方
        text:"我是字符串",

        bool:true,

      },
      getters: {
      },
      mutations: {
          我是修改函数1(){

          },
          我是修改函数1(){

          },
          我是修改函数1(){

          },
          我是修改函数1(){

          },
          我是修改函数1(){

          },
          我是修改函数1(){

          },
      },
      actions: {
      },


}

// 必须暴露
export default aboutm
```

组件内使用commit调用

```
  methods:{
    add(){
      // 调用vuex的修改
      // this.$store.commit("你调用的mutations名字随便写",你想给mutations的数据 可选)
      this.$store.commit("NUM_LIST_ADD_DATA")
    },
    del(){
      this.$store.commit("NUM_LIST_DEL_DATA")
    }
  },
```

创建mutations中的内容

```
let homem={


    state: {//就是数据源  存放数据的地方
        num:666

      },
      getters: {
      },
      mutations: {
        //   state是一个形参可以随便写但是建议写state
        //   这个形参的作用就是代表上面的数据源
        NUM_LIST_ADD_DATA(state){
            state.num++

        },
        NUM_LIST_DEL_DATA(state){
            state.num--
        },
      },
      actions: {
      },


}
// 必须暴露
export default homem
```

mutations的payload

在我们使用commit（）的时候 第一个参数是你要调用的修改动作名 第二个参数是你要给mutations传递的数据

```
 methods:{
    add(){
      // 调用vuex的修改
      // this.$store.commit("你调用的mutations名字随便写",你想给mutations的数据 可选)
      this.$store.commit("NUM_LIST_ADD_DATA",{inputval:this.inputval})
    },
    del(){
      this.$store.commit("NUM_LIST_DEL_DATA",{inputval:this.inputval})
    }
  },
```

在mutations中可以读取这个第二个参数

```
  mutations: {
        //    state是一个形参可以随便写但是建议写state
        //    这个形参的作用就是代表上面的数据源

        // 第二个参数就是payload （载荷） payload就是接受commit的第二个参数
        NUM_LIST_ADD_DATA(state,payload){
            state.num=state.num+payload.inputval

        },
        NUM_LIST_DEL_DATA(state,payload){
            state.num=state.num-payload.inputval
        },
      },
```

扩展---vuex数据修改刷新丢失

监听页面刷新 如果刷新 那么就把vuex的数据存储到本地存储中 然后当页面在此加载得的时候 把本地存储存的原始vuex的数据那出来 替换当前的state数据

```
created () {
    //判断是否有store这个本地存储的数据
    if (sessionStorage.getItem("store") ) {
      // 如果有  那么把vuex的数据替换  把当前的state  和上次刷新存储的state合并起来
```

```
        this.$store.replaceState(Object.assign({},
this.$store.state,JSON.parse(sessionStorage.getItem("store"))))
        sessionStorage.removeItem("store")
    }

    //在页面监听绑定一个beforeunload事件（页面刷新事件）
    // 当页面刷新的时候使用本地存储存一个store的数据  把vuex的数据全部取出来 转成字
符串存起来
    window.addEventListener("beforeunload",()=>{

 sessionStorage.setItem("store",JSON.stringify(this.$store.state))
    })
  }
```

vuex5大属性---actions---异步触发器

actions是vuex的一个属性 他的作用就是在vuex中进行异步操作的触发（很多同学后期总爱说**actions**是异步请求 但是要注意他不是异步请求 他是进行异步操纵的触发 异步请求只是它触发的众多异步操纵的其中一种）

语法：要触发actions使用dispatch （dispath触发actions进行异步触发 把请求来的数据 通过commit交给mutations修改state 在页面读取展示）

1.在组件内使用dispatch（）触发vuex的actions进行异步请求的发送

```
<template>
  <div>
    <button @click="fun()">点我使用vuex发送请求</button>
  </div>
</template>

<script>
export default {
  methods:{
    fun(){
      // 如果我们要使用vuex进行数据的发送
      // this.$store.dispatch("你触发的actions的名字",{参数的key:参数的val})
      this.$store.dispatch("AXIOS_CESHI",{url:"/data/user"})
    }
  }
}
</script>

<style>
```

```
</style>
```

2。需要在对应的actions创建你要触发的异步触发器

```
    actions: {
        // actions中也是一个个的方法 每个方法就是一个异步触发器
        // context代表的就是vuex    store对象
        AXIOS_CESHI(context,payload){
            // 在actions中就可以写请求

            $http({
                url:payload.url,
                method:"get"
            }).then((ok)=>{
                console.log(ok.data)




            })

        }
    },
```

3.把请求来的数据通过context.commit()触发修改

```
    actions: {
        // actions中也是一个个的方法 每个方法就是一个异步触发器
        // context代表的就是vuex    store对象
        AXIOS_CESHI(context,payload){
            // 在actions中就可以写请求

            $http({
                url:payload.url,
                method:"get"
            }).then((ok)=>{
                console.log(ok.data)

                context.commit("AXIOSDATA",{data:ok.data})//把请求来的数据
通过commit触发mutations

            })

        }
    },
```

4创建对应mutations修改state

```
mutations: {


        AXIOSDATA(state,payload){
            state.arr=payload.data
        }
    },
```

vuex5大属性---getters---vuex的计算属性

vue的计算属性是computed　　对data的数据进行依赖 处理之后返回新的计算之后的结果

vuex的getters也是计算属性 只是他和上面的computed最大的区别就是 他处理的数据可以在任何组件直接使用 而vue的computed 处理的数据只能在当前组件使用

```
    getters: {
        // state就是上面的数据源
        newtext(state){
            return state.text.toUpperCase()
        }
    },
```

slot槽口/插槽

用来混合父组件与子组件自己的模板

slot其实就是让组件接受一个外部插入进来的**dom**元素 并且进行显示　通过slot就可以扩展组件的复用性

引子

组件的本质 自定义标签　标签可以是双标签也可以是单标签　那么我们能不能向组件的开关标签内插入dom内容?

　默认情况下 给组件的开关标签中写入**dom**页面不显示原因是因为 组件是一个完整地独立地个体 外部的内容默认插入不进来

```
<template>
  <div>
```

```
        fufufufufuffu
        <!-- 默认情况下   给组件的开关标签中写入dom页面不显示
        原因是因为   组件是一个完整地独立地个体 外部的内容
        默认插入不进来 -->
        <Zicom>
            <h1>你好么么哒</h1>
            <h1>你好么么哒</h1>
            <h1>你好么么哒</h1>
            <h1>你好么么哒</h1>
            <h1>你好么么哒</h1>
            <h1>你好么么哒</h1>

        </Zicom>
    </div>
</template>

<script>
import Zicom from "./ZiCom.vue"
export default {
    components:{
        Zicom
    }
}
</script>

<style>

</style>
```

基本slot

在组件中想接受外部插入的dom位置 直接写slot标签 即可接受外部的dom

```
<template>
    <div>
        <!-- 定义基本的插槽 -->
        <slot></slot>
        zizizizizzzizizi

    </div>
</template>

<script>
export default {

}
</script>
```

```
<style>

</style>
```

具名槽口--带有名字的槽口

语法

在定义slot的时候 使用name属性起个名字

```
<template>
  <div>
      <!-- 定义基本的插槽 -->
      <!-- <slot></slot> -->


      <!-- 定义具名槽口 -->
      <slot name="xiaoming"></slot>
      zizizizizizzizizi
      <slot name="xiaobai"></slot>

  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

在使用的时候使用slot属性 指定那个槽口

```
<template>
  <div>
      fufufufufuffu
      <!-- 默认情况下  给组件的开关标签中写入dom页面不显示
      原因是因为  组件是一个完整地独立地个体 外部的内容
      默认插入不进来 -->
      <Zicom>
          <h1 slot="xiaoming">你好么么哒1</h1>
          <h1>你好么么哒2</h1>
          <h1>你好么么哒3</h1>
```

```
            <h1>你好么么哒4</h1>
            <h1 slot="xiaobai">你好么么哒5</h1>
            <h1>你好么么哒6</h1>

        </Zicom>
    </div>
</template>

<script>
import Zicom from "./ZiCom.vue"
export default {
    components:{
        Zicom
    }
}
</script>

<style>

</style>
```

计算属性--computed

计算属性是vue当中的一个新的属性

计算属性：本质就是一个带有计算（data中的数据）功能的属性

一条数据 在不同位置 展示出不同形态的时候 我们可以使用计算属性

引子

```
<template>
  <div>
      <!-- 一条数据 在不同位置 展示出不同形态的时候  我们可以使用计算属性 -->
      <!-- 因为template区域是展示数据的  我们在这个里面处理数据 不合适 会导致页面的可读性比较差 -->
      <h1>基本展示:{{text}}</h1>
      <h1>大写：{{text.toUpperCase()}}</h1>
      <h1>大写截取：{{text.toUpperCase().slice(1,4)}}</h1>
  </div>
</template>

<script>
export default {
    data(){
        return {
            text:"abcdefgalskdhslkdjlk"
```

```
        }
    }
}
</script>

<style>

</style>
```

语法

写在data methods watch同级的位置

computed:{

  你处理好的变量(){

    return 你的处理逻辑

  }

}

```
<template>
  <div>
      <!-- 一条数据 在不同位置 展示出不同形态的时候  我们可以使用计算属性 -->
      <!-- 因为template区域是展示数据的  我们在这个里面处理数据 不合适 会导致页面的可读性比较差 -->
      <h1>基本展示:{{text}}</h1>
      <h1>大写:{{uptext}}</h1>
      <h1>大写截取:{{upslicetext}}</h1>
  </div>
</template>

<script>
export default {
    data(){
        return {
            text:"abcdefgalskdhslkdjlk"
        }
    },
    computed:{
        uptext(){
            return this.text.toUpperCase()
        },
        upslicetext(){
            return this.text.toUpperCase().slice(1,4)
        }
    }
}
```

```
</script>

<style>

</style>
```

计算属性与方法（methods）的区别

计算属性是依赖缓存的 当计算属性处理的数据被多次调用的时候 只执行一次他会把处理好的数据在第一次处理完成之后放到内存中 知道处理的数据不变 那么后续的调用都是从内存中进行读取的 节省性能

函数 憨憨的 只要你调用就会执行 所以函数来处理数据不好 因为浪费性能

```
<template>
  <div>

      <!-- 计算属性与方法的区别
      计算属性是依赖缓存的  当计算属性处理的数据被多次调用的时候  只执行一次
      他会把处理好的数据在第一次处理完成之后放到内存中 知道处理的数据不变 那么
      后续的调用都是从内存中进行读取的    节省性能

      函数 憨憨的  只要你调用就会执行 所以函数来处理数据不好 因为浪费性能 -->


      <h1>计算属性与方法的区别</h1>
      <h1>计算属性:{{newdata}}</h1>
      <h1>计算属性:{{newdata}}</h1>
      <h1>计算属性:{{newdata}}</h1>

      <h1>函数:{{fun()}}</h1>


  </div>
</template>

<script>
export default {
    data(){
        return {
            text:"askduhasdkjashdkasjh"
        }
    },
    methods:{
        fun(){
            console.log("我是函数")
            return this.text.toUpperCase()
```

```
        }
    },
    computed:{
        newdata(){
            console.log("我是计算属性")
            return this.text.toUpperCase()
        }
    }
}
</script>

<style>

</style>
```

计算属性与watch的区别

watch 是异步的 是监听data的数据 当data的数据改变了watch就会收到通知 调用一个函数处理指定的逻辑

computed 是同步的 是依赖data的数据 当data数据改变了 计算属性就会重新计算返回新的计算结果

自定义过滤器

在不改变原始数据的情况下格式化展示内容

内置过滤器

在**vue2x**中 **vue**已经取消了内置过滤器 在**vue1x**中有内置和自定义过滤器

全局过滤器---filter

定义好之后 在整个项目中都能使用

定义在main.js中 使用filter来定义

```
import Vue from 'vue'
import App from './App.vue'

// 1.引用
import AllCom from "./components/AllCom.vue"
// 2.配置全局组件
// Vue.component(“给你这个全局组件起个名字”,你所要对应使用的组件)
```

```
Vue.component("AllCom",AllCom)


// 全局过滤器
// Vue.filter("过滤器的名",(你要过滤的数据会自动传入)=>{
//   return 你的逻辑
// })
// Vue.filter("过滤器的名",(你要过滤的数据会自动传入)=>{
//   return 你的逻辑
// })
Vue.filter("xiaoming",(val)=>{
  return val.slice(0,5)
})

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

局部过滤器---filters

在那个组件中定义的 就只能在这个组件中使用

语法:

写在与data methods等同级位置

filters:{

过滤器的名字(形参就是你要过滤的数据会自动传入){

return 你的过滤逻辑

}

}

```
<template>
  <div>
      <h1 v-for="(v,i) in obj" :key="i">
        {{v.date|xiaoming}}
      </h1>
  </div>
</template>

<script>
export default {
```

```
    data(){
        return {
            obj:[
                {date:"22021-05-14"},
                {date:"22022-05-14"},
                {date:"22023-05-14"},
                {date:"22024-05-14"},
                {date:"22025-05-14"},
                {date:"22026-05-14"},
                {date:"22027-05-14"},
                {date:"22028-05-14"},
                {date:"22029-05-14"}
            ]
        }
    },
    filters:{
        // xiaoming 是过滤器的名  val形参就是你今后要过滤的数据会自动传入
        xiaoming(val){
            return val.slice(0,5)
        }
    }
}
</script>

<style>

</style>
```

过滤器的使用

在你要使用数据的时候 通过 |来完成

语法:

```
{{展示变量|过滤器的名字|过滤器名字2}}
```

事件对象与修饰符

事件对象--$event

谁触发这个事件事件对象就指向谁 （事件对象中包含触发这个事件的元素所有信息）

```
<template>
  <div>
      <h1>事件对象</h1>
      <!-- <button @click="fun($event)">点我</button> -->
      <input type="text" @keydown="fun($event)">
```

```
  </div>
</template>

<script>
export default {
    methods:{
        fun(e){
            // console.log(e)
            if(e.keyCode==90){
                console.log("z被按下了")
            }
        }
    }
}
</script>

<style>

</style>
```

修饰符

通过vue提供的修饰符可以来处理dom事件的一些细节性的内容

按键修饰符

.up .down .left .right .ctrl .space .enter

```
<template>
  <div>

      <input type="text" @keydown.space="fun()">
  </div>
</template>

<script>
export default {
    methods:{
        fun(){
                console.log("aaaaa")
        }
    }
}
</script>

<style>

</style>
```

事件修饰符

1. .stop修饰符　阻止事件传播
2. .prevent修饰符　阻止事件默认行为
3. .captrue修饰符　设置捕获
4. .self 修饰符　只会触发自己范围内的事件 不包含子元素
5. .once修饰符　只触发当前事件一次

```
<template>
  <div>

      <div class="fu" @click="fufun()">
          <div class="zi" @click.stop="zifun()"></div>
      </div>

  </div>
</template>

<script>
export default {
    methods:{
        fufun(){
            console.log("fufufuffufufu")
        },
        zifun(){
            console.log("zizizizziizzi")
        },
    }
}
</script>

<style>
    .fu{
        width: 600px;
        height: 600px;
        background-color: pink;
    }
    .zi{
        width: 300px;
        height: 300px;
        background-color: goldenrod;
    }
</style>
```

自定义指令--directives

在现有内置指令不够用的时候 我们可以自己定义指令来进行使用

语法：写在与data methods watch computed 同级的位置

directives:{

  自定义指令的名字:{

    自定义指令的钩子函数（el代表的就是指定放在那个dom上形参就是谁）{

      你的逻辑

    }

  },

  自定义指令的名字2:{

  },

}

使用 v-自定指令的名字

自定义指令的钩子函数

bind 指令绑定到元素之上的时候执行 但是只执行一次

unbind 指令被移除的时候执行 只执行一次

update 所有组件节点更新的时候执行调用

componentUpdate 指令所在节点以及所有的子节点都更新完成的时候调用

inserted 绑定指令的元素在页面展示的时候调用

```
<template>
  <div>
      <input type="text" v-xiaoming/>
  </div>
</template>

<script>
export default {
    // 创建
    directives:{
        xiaoming:{
```

```
            inserted(el){
                el.focus()
            }
        }
    }
}
</script>

<style>

</style>
```

钩子函数

自动执行的函数叫做钩子函数

生命周期的钩子函数

vue实例从创建到销毁的过程中被自动执行的函数

写在与 data methods watch computed directives 同级的位置

作用

就是给程序提供一个自动执行逻辑的场所

8大钩子

实例创建

实例创建之前-----beforeCreate

实例创建之后-----created

模板渲染

模板渲染之前-----beforeMount

模板渲染之后-----mounted

数据更新

数据更新之前-----beforeUpdate

数据更新之后-----updated

实例销毁

实例销毁之前-----beforeDestory

实例销毁之后-----destoryed

```
<template>
  <div>
      <input type="text" v-model="text">
  </div>
</template>

<script>
export default {
    data(){
        return {
            text:"我是默认值"
        }
    },
    methods:{

    },
    computed:{

    },
    watch:{

    },
    directives:{

    },
    // 生命周期的钩子函数
    beforeCreate(){
        console.log("实例创建之前")
    },
    created(){
        console.log("实例创建之后")
    },
    beforeMount(){
        console.log("模板渲染之前")
    },
    mounted(){
        console.log("模板渲染之后")
    },
    beforeUpdate(){
        console.log("数据更新之前")
    },
    updated(){
        console.log("数据更新之后")
    },
    beforeDestory(){
        console.log("实例销毁之前")
    },
```

```
    destoryed(){
        console.log("实例销毁之后")
    },

}
</script>

<style>

</style>
```

什么是什么周期的钩子函数?

vue实例从创建到销毁的过程中被自动执行的函数

生命周期第一次执行那些?

实例创建之前-----beforeCreate

实例创建之后-----created

模板渲染之前-----beforeMount

模板渲染之后-----mounted

生命周期几个阶段

4大阶段 8个钩子

第一次页面加载触发那些

实例创建前后 模板渲染前后

dom在那个阶段渲染完毕?

mounted

请您介绍一下生命周期的每个钩子? (请您给我说一下实例创建的流程与原理)

实例创建

实例创建之前-----beforeCreate 数据的观测与事件的初始化 属性的创建 还没有进行

实例创建之后-----created 在此时vue实例已经创建完毕 所以 数据的观测 属性 方法等内容都已经创建完毕（el属性还没有挂载）

模板渲染

模板渲染之前-----beforeMount 在页面挂载前调用的 所以在此阶段 页面还没有进行渲染与模板的编译 程序在此时会把数据绑定到页面上 但是页面并没有显示

模板渲染之后-----mounted 页面已经渲染出来了 html的内容会在dom中进行加载展示

数据更新

数据更新之前-----beforeUpdate 在此时数据会不停的在dom中进行修改

数据更新之后-----updated 把修改之后的dom内容已经在页面成功的展示了

实例销毁

实例销毁之前-----beforeDestory 此时vue实例还能用

实例销毁之后-----destoryed 什么都没有了 vue实例等内容都没了

父子组件的生周期顺序是什么?

父beforeCreate -> 父created -> 父beforeMount -> 子beforeCreate -> 子created -> 子beforeMount -> 子mounted -> 父mounted->父beforeUpdate->子beforeUpdate->子updated->父updated->父beforeDestroy->子beforeDestroy->子destroyed->父destroyed

前后台交互

什么是前台什么是后台 什么是前端什么事后端?

前端指的是数据展示

后端指的是数据处理

分类

1.原生ajax

2.jqueryAjax 对上面的XHR对象进行了封装 方便使用

（1）下载jquery　npm install --save jquery

（2）引用jquery

```
<template>
  <div>
      <h1>jqueryajax请求数据</h1>
  </div>
</template>

<script>
// 引用jquery
import $ from "jquery"
export default {

}
</script>

<style>

</style>
```

（3）使用jquery

```
<template>
  <div>
      <h1>jqueryajax请求数据</h1>
      <h1>{{data}}</h1>
  </div>
</template>

<script>
// 引用jquery
import $ from "jquery"
export default {
    data(){
        return {
            data:{}
        }
    },

    mounted(){
```

```
        $.ajax({
            url:"/user_list/ceshidemo",
            type:"GET",
            dataType:"json",
            success:(ok)=>{
                console.log(ok)

                this.data=ok
            }
        })
    }

}
</script>

<style>

</style>
```

3.axios 也是对XHR对象进行封装 当时它是使用符合当下的promise来进行的封装

(1)下载 cnpm install --save axios

(2)引用

```
// 引用axios
import $http from "axios"
```

(3)使用

```
$http({
            // 地址
            url:"/movie/list/data_list",
            // 方式 有的同学写的时候写成methods了 有s也可以  原因是因为默认get他不认识你带s的这个属性所以执行默认了
            method:"get"
        }).then((ok)=>{
            console.log(ok.data.subjects)
            // 把请求来的数据赋值给arr
            this.arr=ok.data.subjects

        }).catch((err)=>{
            console.log(err)
        })
```

4.fetch fetch和上面三个都不一样 因为他没有使用XHRajax对象 而是es最新的请求标准 但是既然是最新的 那么兼容性有很大问题

数据请求封装与axios拦截器

拦截器

每次发送请求或者请求相应的时候 都会经过拦截器 才会进入到我们的程序（就是对我们的请求和相应进行发送前或者获取前的一个拦截 ）

请求拦截

发送请求的时候会经过请求拦截 我们就可以在请求拦截上携带每次都要给后台的数据（用户的登录状态）

相应拦截

每次相应数据的时候都会经过相应拦截 （我们就可以在相应拦截的时候对我们的错误或者成功做出反应）

编写拦截器

1.在src下新建一个util文件夹（工具文件夹 用来放置一些项目中有了更好没有也无所谓的工具库）在创建对应的文件用来容纳拦截器

```
// 引用axios
import axios from "axios"
let service=axios.create()
```

```
// 添加请求拦截器
service.interceptors.request.use(function (config) {
    // 在发送请求之前做些什么
    return config;
  }, function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  });

// 添加响应拦截器
service.interceptors.response.use(function (response) {
    // 对响应数据做点什么
    return response;
  }, function (error) {
    // 对响应错误做点什么
    return Promise.reject(error);
  });

//    暴露
export default service
```

数据请求的封装

尽量符合最新的es标准promise来进行封装

1.在src下新建一个api的文件夹（就是容纳数据请求的）新建一个文件容纳封装的请求

```
// 1.引用拦截器
import service from "@/utils/service.js"
// 2.开始使用promise进行封装
let getlink=(url,method='get')=>{
    // resolve成功
    // reject失败
   return new Promise((resolve,reject)=>{
        // 就可以执行你所要封装的操纵
        service.request({
            url,
            method
        }).then((ok)=>{
            resolve(ok)
        }).catch((err)=>{
            reject(err)
        })
    })
}

// 3.暴露
export default getlink
```

使用封装的请求

1.在你想用的地方先引用 在使用

```
<template>
  <div class="about">
    <h1>This is an about page</h1>
  </div>
</template>
<script>
// 1.引用
import getlink from "@/api/getapi.js";
export default {
  created(){
    getlink("/api/data/cityinfo/101320101.html").then((ok)=>{
      console.log(ok)
    }).catch((err)=>{
      console.log(err)
    })
  }
};
</script>
```

axios get发送数据

axios使用params发送数据

```
import service from "@/utils/service.js"
import { values } from "core-js/core/array"

let link=(url,method="get",params)=>{
    return new Promise((resolve,reject)=>{
        service.request({
            url,
            method,
            params//get发送数据的方式
        }).then((ok)=>{
            resolve(ok)
        }).catch((err)=>{
            reject(err)
        })
    })

    }

export default link
```

axios post发送数据

axios使用 data发送数据

```
import service from "@/utils/service.js"


let link=(url,method="get",params,data)=>{
    return new Promise((resolve,reject)=>{
        service.request({
            url,
            method,
            params,//get发送数据的方式
            data//post发送数据
        }).then((ok)=>{
            resolve(ok)
        }).catch((err)=>{
            reject(err)
        })
    })

    }

export default link
```

axios默认的content-type是application/json,即json格式，后台可以使用字符串进行接收，然后再解析即可

注意

如果后台不是用json格式接收 的话那么 post发送数据后台可能接收不到所以我们可以使用*qs*库序列化数据

1.下载 npm install --save qs

2.引用 import qs from “qs”

3.在传递数据的时候使用qs序列化

```
let key=qs.stringify({
            key:val
})
```

vue环境部署与baseurl配置

环境变量

在开发的时候一般会有是三个环境：开发环境 测试环境 线上（生产）环境

vue 中有个概念就是模式，默认先**vue cli** 有三个模式

- `development`开发环境模式用于 vue-cli-service serve
- `production`生产环境模式用于 vue-cli-service build 和 vue-cli-service test:e2e
- `test`测试环境模式用于 vue-cli-service test:unit

但是往往开发的时候可能不止有三种：

- 本地环境（local）
- 开发环境（development）
- 测试环境（devtest）
- 预发布环境（beta）
- 生产环境（production）

创建不同环境变量文件

通过为.env文件增加后缀来设置某个模式下特有的环境变量。

1.在项目根路径下设置 新建对应文件 .env.development（开发环境文件） .env.production（生产环境文件）.env.devtest（测试环境文件）

2.在每个文件写入如下内容(VUE__APP__随便写)

```
VUE_APP_XIAOMING = "开发模式"
```

package.json环境对应的执行语句

```
  "scripts": {
    "serve": "vue-cli-service serve",//开发模式
    "build": "vue-cli-service build",//生产模式
    "dev_test_build": "vue-cli-service build --mode development_test",//
测试模式
    "lint": "vue-cli-service lint"

  },
```

使用变量process.env.你的内容即可得到

```
mounted()=>{

    console.log(process.env.VUE_APP_XIAOMING);



}
```

打包生产环境

1.npm run build 会生成一个dist文件夹 我们点开之后像运行html一样运行项目

2.配置生产环境 （1）在vue.config.js中设置 publicPath:"./" (2)把路由模式设置为hash

3.重新build

默认请求地址baseurl

因为我们可以配置不同的环境变量 那么我们就可以在设置请求的时候 根据不同的环境来设置不同的请求地址

1.在开发环境文件中配置我们的请求（今后开发的时候可以在其他配置文件中配置你的请求）

```
VUE_APP_XIAOMING = "开发模式"
VUE_APP_API = "http://localhost:8888"
```

2.在封装的拦截器文件中配置baseURL

```
let axiosurl = ""
// 如果为开发模式的话执行url
if(process.env.NODE_ENV === 'development' ){
  axiosurl=process.env.VUE_APP_API
}
else{
  // 否则设置成其他的模式（这里今后有很多个判断）
  axiosurl=process.env.VUE_APP_API
}

// 创建axios 赋值给常量service
const service = axios.create({
  baseURL: axiosurl
});
```

3.在今后的请求中直接写请求的路由地址就行

```
    // 发送请求 url中请求地址前缀已经在baseURL中配置了
      link("/one").then((ok)=>{
        console.log(ok)
      })
```

url封装

方便后期维护

1、在api文件夹下新建一个url.js（文件的名字和位置随便写）

2. 把所有的请求地址写在里面

```
let apiurl={
    // 小明
    ceshi:"/link"
    // 小黑
}

export default apiurl
```

3.在想使用的地方引用使用

```
引用
import urls from "@/apis/urls.js"


使用
AXIOS_LINK(context){
//使用url
            link(urls.ceshi).then((ok)=>{
                console.log(ok)

                context.commit("UPARR",{data:ok.data})
            })
        }
    },
```

请求拦截器

有的时候我们在发送请求的时候后端需要每次我们携带一些数据（比如token）但是添加在每个请求非常麻烦 所以我们可以添加在请求头中 那么这个配置我们可以在拦截器中进行设置

通常我们在登陆成功之后后端会给我们返回一个token 所以我们可以把它存储在本地存储中 在拦截器中从本地存储中获取token 然后添加在请求头中

```
// 添加请求拦截器
service.interceptors.request.use(function (config) {
    // 在发送请求之前做些什么

    // 获取token
    const token = localStorage.getItem("user_token");
    // 判断是否存在 然后发送token
    token && (config.headers.token = token);

    return config;
  }, function (error) {
    // 对请求错误做些什么
    return Promise.reject(error);
  });
```

响应拦截器

响应拦截器的作用是在接收到响应后进行一些操作

响应拦截器也是一样如此，就是在请求结果返回后，先不直接显示错误，而是先对响应码等等进行处理，处理好后再导出给页面一个错误提醒

1.在util文件夹下的service.js中打印相应信息（需要关闭后台或者修改下错误路径）会发现得到了不同的错误信息

```
service.interceptors.response.use(function (response) {
    return response;
  }, function (error) {
     //  打印下看看当错误的时候干什么
  console.log("aaaaaa",error);

    return Promise.reject(error);
  });
  export default service
```

2.得到失败响应的status(状态码)需要在response中获得 error.response.status

```
    console.log("aaaaaa", error.response.status);
```

3 可以对错误信息进行设置

```
        //  打印下看看当错误的时候干什么
    console.log("aaaaaa", error.response.status);
    switch (error.response.status) {
      case 404:
        console.log("url信息有误")
        break;
      case 500:
        console.log("服务器有问题")
        break;

      default:
        console.log("未知错误")
        break;
    }
```

跨域

因为浏览器的安全机制 同源策略 不同端口不同域名不同协议 就会造成跨域

jsonp

面试的时候千万不要先说jsonp 太low

面试的时候千万不要先说jsonp 太low

面试的时候千万不要先说jsonp 太low

面试的时候千万不要先说jsonp 太low

代理跨域

代理：造成跨域的问题是浏览器的安全机制 因为有了这个安全机制我们才要解决跨域

我现在不让浏览器帮我发送请求了 而是让我项目的服务器帮助我绕开浏览器发送请求

nginx反向代理

devServer代理跨域

devServer就是vue脚手架中那个内置的微型开发小服务器

1.在项目的根路径下 创建一个vue.config.js

2.写入如下内容

```
module.exports={
    devServer: {
        proxy: {  //配置跨域
          '/api': {
            target: 'http://www.weather.com.cn/',  //需要解决跨域的地址
            pathRewrite: {
              '^/api': ''
            }
          },
        }
      },
}
```

3.修改请求路径

```
<template>
  <div>
      <h1>大纲要修写法</h1>
        <button @click="fun()">点我请求数据</button>
  </div>
</template>

<script>
import $http from "axios"
export default {
    methods:{
        fun(){
           $http({
            //     修改名字
               url:"/api/data/cityinfo/101320101.html",
               method:"get",//没有s
           }).then((ok)=>{
               console.log(ok)
           }).catch((err)=>{
               console.log(err)
           })
        }
    }
}
</script>

<style>

</style>
```

4.千万不要忘了重启

4.千万不要忘了重启

4.千万不要忘了重启

4.千万不要忘了重启

4.千万不要忘了重启

4.千万不要忘了重启

4.千万不要忘了重启

corc

mockjs模拟数据

mockjs就是为我们创建模拟数据的一个技术 因为在开发的时候 我们不可能写一个功能之前后端都给我们把对应的接口写好 所以有的时候我们需要有模拟数据 来给我们提供页面展示内容

1.在项目的src下新建一个mock的文件夹 用来容纳模拟数据

2.在mock文件夹下创建一个js文件用来容纳mockjs的代码 和你对应的json文件 用来容纳模拟数据

3.下载mockjs　npm install --save mockjs

4.在之前新建的js中写入如下代码

```
// 容纳模拟数据的代码
let Mock=require("mockjs")
// Mock.mock("地址随便写","请求方式get/post",require("你要读取的json文件路径"))
Mock.mock("/user_list/ceshidemo","get",require("./a.json"))
```

5.在main.js中关联mock

```
import Vue from 'vue'
import App from './App.vue'

require("./mock")//如果没有指定js　那么项目会自动进入mock文件夹去寻找index.js

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

路由

路由是什么?

就是可以让我们完成一个**SPA**单页面应用 传统的页面在跳转切换的时候会造成页面加载白屏 这样一来用户体验非常的差 但是 我们通过spa应用 可以达到类似于原生app的切换效果 切换没有白屏 丝滑切换 用户体验更高

路由的本质 就是根据**url**的不同来渲染不同的组件页面

路由基本创建

方式1 脚手架自动创建

在创建项目的时候 选中router项即可在项目中集成路由

拿到路由项目之后怎么办?

1.删除掉views文件夹中的内容

2.删除掉components下的helloword.vue

3.在app。vue中删除内容 但是router-view千万千万不要删

方式2 手工创建方式

稍后在说

一级路由创建

1.在views文件夹中创建对应的路由页面组件

2.配置路由 在router下index.js中进行配置

（2-1）先把你要使用的组件页面引用

```
import Vue from 'vue'
import VueRouter from 'vue-router'
// 1.把你要使用的路由页面引用
import Fenlei from '../views/fenlei.vue'
import Gouwuche from '../views/gouwuche.vue'
import Home from '../views/home.vue'
import Jingxi from '../views/jingxi.vue'
import Wode from '../views/wode.vue'
```

```
Vue.use(VueRouter)

const routes = [
  {
    path: '/',//url路径
    name: 'home',//给这个路由规则起个名字
    component: HomeView //引用组件
  },

  // 下面是后面要学的
  // {
  //   path: '/about',
  //   name: 'about',
  //   component: () => import('../views/AboutView.vue')
  // }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

(2-2)

配置路由规则

```
import Vue from 'vue'
import VueRouter from 'vue-router'
// 1.把你要使用的路由页面引用
import Fenlei from '../views/fenlei.vue'
import Gouwuche from '../views/gouwuche.vue'
import Home from '../views/home.vue'
import Jingxi from '../views/jingxi.vue'
import Wode from '../views/wode.vue'

Vue.use(VueRouter)

const routes = [
  {
    path: '/fenlei',//url路径
    name: 'fenlei',//给这个路由规则起个名字
    component: Fenlei //引用组件
  },
```

```
    {
      path: '/gouwuche',//url路径
      name: 'gouwuche',//给这个路由规则起个名字
      component: Gouwuche //引用组件
    },
    {
      path: '/home',//url路径
      name: 'home',//给这个路由规则起个名字
      component: Home //引用组件
    },
    {
      path: '/jingxi',//url路径
      name: 'Jingxi',//给这个路由规则起个名字
      component: Jingxi //引用组件
    },
    {
      path: '/wode',//url路径
      name: 'Wode',//给这个路由规则起个名字
      component: Wode //引用组件
    },









    // 下面是后面要学的
    // {
    //   path: '/about',
    //   name: 'about',
    //   component: () => import('../views/AboutView.vue')
    // }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

3.在app.vue中设置router-view路由出口

路由导航

路由导航就是在页面中的一些连接通过点击之后完成页面的跳转

标签的方式---声明式导航

不能使用a标签

router-link这个标签来完成页面之间的跳转 其中有一个to属性就是写你的路径

```
<template>
  <div>
      <router-link to="/home">首页</router-link>
      <router-link to="/fenlei">分类</router-link>
      <router-link to="/jingxi">惊喜</router-link>
      <router-link to="/gouwuche">购物车</router-link>
      <router-link to="/wode">我的</router-link>
  </div>
</template>

<script>
export default {

}
</script>

<style>

</style>
```

扩展--动态类名

在当前路由页面下 vuerouter会给对应的声明式导航添加一个类名 通过这个类名可以设置当前的样式

```
<template>
  <div>
      <router-link to="/home">首页</router-link>
      <router-link to="/fenlei">分类</router-link>
      <router-link to="/jingxi">惊喜</router-link>
      <router-link to="/gouwuche">购物车</router-link>
      <router-link to="/wode">我的</router-link>
  </div>
</template>

<script>
export default {

}
```

```
</script>

<style scoped>
    .router-link-exact-active{
        background-color: red;
    }
</style>
```

js的方式---编程式导航

this.$router.push("/你要去的路径") push跳转的页面可以回退回来

```
<script>
export default {
    methods:{
        fun(){
            this.$router.push("/jingxi")
        }
    }
}
</script>
```

this.$router.replace('/替换路径') replace是替换 跳转之后不能回退

this.$router.go() 正数前进 负数后退

二级或者多级路由创建

二级路由或者多级路由在创建的时候 使用**children**关键字来进行规则的配置

1.路由页面创建 views文件夹

2.配置二级路由规则

（2-1）在router文件夹下的index.js中先引用二级路由页面

（2-2）配置路由规则 必须在对应的一级路由规则中使用**children**关键字来进行配置

```
{
    path: '/home',//url路径
    name: 'home',//给这个路由规则起个名字
    component: Home, //引用组件
    children:[//配置二级路由规则
      {
        path: '/era',
        name: 'era',
```

```
        component: Era
      },
      {
        path: '/erc',
        name: 'erc',
        component: Erc
      },
      {
        path: '/erd',
        name: 'erd',
        component: Erd
      },
    ]
  },
```

3.注意

3.注意

3.注意

3.注意

3.注意 必须设置二级路由的路由出口 **router-view** (写在对应一级路由的页面中)

扩展---二级路由path设置

在上面的笔记中 会发现 我们在配置二级路由的时候 path路径为 /二级 在路由导航的时候 我们在to中直接就写/二级

```
{
    path: '/home',//url路径
    name: 'home',//给这个路由规则起个名字
    component: Home, //引用组件
    children:[//配置二级路由规则
      {
        path: '/era',
        name: 'era',
        component: Era
      },
      {
        path: '/erc',  path路径 是直接/二级
        name: 'erc',
        component: Erc
      },
      {
        path: '/erd',
        name: 'erd',
        component: Erd
      },
```

```
    ]
  },
```

那么路由导航

```
<router-link to="/da">火锅</router-link>    这里直接/二级的path
```

注意

注意

注意

注意

注意

注意 我们在写二级路由的时候 path 也可以写成 不加/的方式

```
{
    path: '/meishi',
    name: 'meishi',
    component: Meishi,
    children:[
      {
        path: 'da',//这里二级路由的path没有加/
        name: 'da',
        component: Da
      },
      {
        path: 'db',
        name: 'db',
        component: Db
      },
      {
        path: 'dc',
        name: 'dc',
        component: Dc
      },

    ]
  },
```

在路由导航的时候 必须写成 /一级/二级

```
      <router-link to="/meishi/da">火锅</router-link>
        <router-link to="/meishi/db">甜品</router-link>
        <router-link to="/meishi/dc">自助餐</router-link>
```

路由重定向---redirect

重（重新）定（定位）向（方向）

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Meishi from '../views/MeiShi.vue'
import Shouye from '../views/ShouYe.vue'


import Da from "@/views/MeiShichil/DemoA.vue"
import Db from "@/views/MeiShichil/DemoB.vue"
import Dc from "@/views/MeiShichil/DemoC.vue"

Vue.use(VueRouter)

const routes = [
  {
    path: '/meishi',
    name: 'meishi',
    component: Meishi,
    children:[
      {
        path: 'da',//这里二级路由的path没有加/
        name: 'da',
        component: Da
      },
      {
        path: 'db',
        name: 'db',
        component: Db
      },
      {
        path: 'dc',
        name: 'dc',
        component: Dc
      },

    ]
  },
  {
    path: '/shouye',
```

```
    name: 'shouye',
    component: Shouye
  },


  // 重定向
  {
    path:"/",
    redirect:"/shouye"
  }

  // {
  //   path: '/about',
  //   name: 'about',
  //   // route level code-splitting
  //   // this generates a separate chunk (about.[hash].js) for this
route
  //   // which is lazy-loaded when the route is visited.
  //   component: () => import(/* webpackChunkName: "about" */
'../views/AboutView.vue')
  // }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

404页面

就是给用户一个页面错误提示的作用

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Meishi from '../views/MeiShi.vue'
import Shouye from '../views/ShouYe.vue'
import No from '../views/NoView.vue'


import Da from "@/views/MeiShichil/DemoA.vue"
import Db from "@/views/MeiShichil/DemoB.vue"
import Dc from "@/views/MeiShichil/DemoC.vue"

Vue.use(VueRouter)
```

```
const routes = [
  {
    path: '/meishi',
    name: 'meishi',
    component: Meishi,
    children:[
      {
        path: 'da',//这里二级路由的path没有加/
        name: 'da',
        component: Da
      },
      {
        path: 'db',
        name: 'db',
        component: Db
      },
      {
        path: 'dc',
        name: 'dc',
        component: Dc
      },

    ]
  },
  {
    path: '/shouye',
    name: 'shouye',
    component: Shouye
  },


  // 重定向
  {
    path:"/",
    redirect:"/shouye"
  },

  // 404页面必须在所有规则的最下面
  {
    path: '*',
    name: 'no',
    component: No
  }

  // {
  //   path: '/about',
  //   name: 'about',
  //   // route level code-splitting
```

```
  //   // this generates a separate chunk (about.[hash].js) for this
route
  //   // which is lazy-loaded when the route is visited.
  //   component: () => import(/* webpackChunkName: "about" */
'../views/AboutView.vue')
  // }
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

路由传参 动态路由匹配

就是把数据从一个路由页面传递到另外一个路由页面中 （新闻列表页面 用户点击之后 会跳转到新闻详情页 但是新闻详情页展示的内容应该是用点击的那一条新闻 所以如何把数据从一个页面传递到另外一个页面）

params方式

1.在需要接受数据的路由页面规则上 设置接受参数

```
{
    path: '/all/:xiaoming', //设置接收参数
    name: 'All',
    component: All
  },
```

2.发送

声明式

```
 <router-link v-bind:to='{name:"All",params:{xiaoming:"111我是phone传递的
数据"}}'>点我使用声明式把数据传递给all页面</router-link>
```

编程式

```
<template>
    <div>
        <Link/>
```

```
        phone

        <h1 v-for="(v,i) in arr" :key="i" @click="fun()">{{v.title}}
</h1>
    </div>
</template>

<script>
export default {
    methods:{
        fun(){
            // 编程式导航
            // this.$router.push({name:"你要去的路由规则的名字",params:{你在
规则上配置的参数:你要传递的数据}});
            this.$router.push({name:"All",params:{xiaoming:"我是phone传递
的数据"}});
        }
    },

    data(){
        return {
            arr:[
                {
                    title:'我是辩题11',content:"我是内容11",
                },
                {
                    title:'我是辩题12',content:"我是内容12",
                },
                {
                    title:'我是辩题13',content:"我是内容14",
                },
                {
                    title:'我是辩题14',content:"我是内容15",
                },
            ]
        }
    }

}
</script>

<style>

</style>
```

3.接受

在接受的页面使用　　this.$route.params.xxx　注意单词

```
<template>
  <div>
      <h1 @click="fun()">&lt;</h1>
      <h1>新闻详情页---{{this.$route.params.xiaoming}}</h1>
  </div>
</template>

<script>
export default {
    methods:{
        fun(){
            // 路由回退
            this.$router.go(-1);
        }
    }
}
</script>

<style>

</style>
```

query方式

1.发送

声明式

```
<template>
  <div>
      <Link/>
      home

      <!-- <router-link :to="{name:'你要去的页面规则的name',query:{key:val}}">使用query方式把数据传递到购物页面</router-link> -->
      <router-link :to="{name:'Shop',query:{xiaohong:'我是query数据'}}">使用query方式把数据传递到购物页面</router-link>
      <!-- query方式发送参数的时候  不但可以写name   还可以写成path -->
       <router-link :to="{path:'/shop',query:{xiaohong:'1111我是query数据'}}">使用query方式把数据传递到购物页面</router-link>

  </div>
</template>

<script>
export default {
```

```
}
</script>

<style>

</style>
```

编程式

课下自行尝试

2.接受

在想使用数据的页面中 使用　　this.$route.query.xxxx

query方式与params方式传参区别

1.url展示形态

params方式 传递参数的时候相对安全一些 因为不显示传递的数据key

query方式 在传递的时候会显示数据key和val 相对来说没有parmas安全

2.刷新丢失

parmas方式 刷新会丢失 （上线之后）

query方式 刷新不会丢失

3.语法区别

$router与$route的区别

$router代表的是 路由对象 他所涉及的范围是全部项目页面

$route代表 当前路由页面对象

扩展----路由懒加载

在没有使用路由懒加载的时候　第一次加载会把所有的路由页面全部加载完　在显示页面　有的时候 用户的设备网络不好的时候 导致渲染时间过长　用户就会在第一次进入我们项目的时候有白屏问题（用户体验不好）

异步组件方式

```
    component: (resolve) => require(['你引用组件页面的路径'], resolve)
```

import导入方式

```
  component: () => import('../views/About.vue')
```

路由钩子/路由守卫/导航守卫/路由卫士

在路由跳转的特定时期自动触发的一些函数 （这些钩子函数通常是在页面跳转的时候 对项目中的用户权限进行设置的）

全局

全局前置守卫---beforeEach()

在所有路由跳转之前 触发的钩子函数（此时路由还没有跳转完成）

全局后置守卫--- afterEach(()

所有路由跳转之后 触发的钩子函数 （此时已经进入到了跳转之后的新页面）

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from '../views/Home.vue'

Vue.use(VueRouter)

const routes = [
  {
    path: '/home',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',

    component: () => import('../views/About.vue')
  },
  {
    path: '/phone',
    name: 'phone',

    component: () => import('../views/phone.vue')
  },
  {
    path: '/shop',
    name: 'shop',

    component: () => import('../views/shop.vue')
  },
```

```
]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

// 全局前置
// to  去哪里
// from 从哪个路由来的
// next 下一步（必须要写  不写的话  就会卡在这个钩子中不想下进行了）
router.beforeEach((to,from,next)=>{
  console.log(to);
  console.log(from);

  if(to.path=="/phone"||to.path=="/shop"){
        next()
  }else{
    alert("您没有登录请您登录后再访问")
    next("/phone")
  }


})
// 全局后置
router.afterEach((to,from)=>{
  console.log("全局后置守卫")
})

export default router
```

路由独享

路由独享---beforeEnter

只会对一个路由规则生效（路由独享写在那个规则之上 就对哪一个生效）

```
  {
    path: '/about',
    name: 'About',

    component: () => import('../views/About.vue'),
    // 路由独享守卫
    beforeEnter(to,from,next){
      console.log(to);
      console.log(from);
      next()
    }
  },
```

组件内

仅仅对某些组件在路由跳转的时候生效

进入组件时候---beforeRouteEnter

离开组件的时候---beforeRouteLeave

```
// 进入组件
 beforeRouteEnter(to,from,next){
    console.log("我进来了")
    console.log(to)
    console.log(from)
    next()
 },
 beforeRouteLeave(to,from,next){
    console.log(to)
    console.log(from)
    if(confirm("您确定离开吗？")){
        next()
    }else{
      next(false)
    }
 },
```

路由懒加载

因为使用传统的路由来进行开发的时候可能会出现首页加载白屏问题(因为如果我们有200个页面 那么在第一次加载的时候 会把这200个页面都先加载出来然后再把指定的哪一个显示到页面上) 白屏问题对于用户体验很不好

所以为了解决首页加载白屏 那么我们可以使用路由懒加载方式

懒（按需）加载

实现1 import方式

```
{
   path: '/about',
   name: 'about',
   component: () => import('你的路由页面路径')
 }
```

实现2 异步组件方式

```
   component: (resolve) => require(['你的路由页面路径'], resolve),
```

路由模式

1.hash模式　默认模式　带#号　hash模式上线正常　hash兼容性好

2.history模式 必须手工指定 不带#号　history上线之后刷新会丢失页面（让后端给你们配置服务器的重定向） 是H5新增 所以兼容性比较差

修改 使用mode属性来进行修改

ui库--vantui

mixin混入

就是vue组件中的一个封装技巧 他的作用就是对组件中重复出现的数据 方法 生命周期等内容进行封装方便其他组件复用

全局混入---mixin

全局混入 能不用尽量不要用 因为会造成全局的污染

1.新建一个文件夹 mixin 用来存储混入的代码

```
let demo={
```

```
    methods:{
        fun(){
            console.log("你好")
        }
    }


}


export default demo
```

2.main.JS中进行配置

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'


// 全局混入
import demo from "./mixin/demo.js"
// 全局混入
Vue.mixin(demo)

// 全局过滤器
// Vue.filter("xiaoming",(val)=>{
//   if(val.length>6){
//    return val.slice(0,5)+"..."
//   }else{
//     return val
//   }
// })
// 全局过滤器

Vue.config.productionTip = false

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

局部混入---mixins

1.新建一个文件夹 mixin 用来存储混入的代码

```
let demo={


    methods:{
        fun(){
            console.log("你好")
        }
    }


}


export default demo
```

2.在你想使用的组件中 引用 使用

```
<template>
  <div>
      <!-- 3.你就可以在组件内容任意的使用混入的内容 -->
      aaaaaaaa-- <button @click="fun()">点我--{{newnum}}</button>
  </div>
</template>

<script>
// 1.引用
import demo from "@/mixin/demo.js"
export default {
    // 2.使用mixins调用
    // 与data  methods同级使用
    mixins:[demo]
}
</script>

<style>

</style>
```

ref

ref就是对页面的dom进行操作

使用场景1--绑定到DOM元素身上

进行基本的页面dom操作。

1.在标签的dom之上使用ref=“随便起个名字”

2.this.$refs.你的那个名字即可找到指定元素

```
<template>
  <div>
    <!-- 1.绑定 -->
    <h1 ref="wangcai">找到我</h1>
  <button @click="fun()">点我修改上面的内容</button>

  </div>
</template>

<script>
export default {
  methods:{
    fun(){
      // 2.找到他
      this.$refs.wangcai.style.color="red";
    }
  }
}
</script>

<style>

</style>
```

使用场景2--绑定到组件身上

因为组件的核心就是自定义标签　上面的ref可以绑定到标签之上　我们的自定义标签能不能使用ref 如果可以使用那么得到的是什么?

可以得到绑定的那个组件所有的信息（得到了vue组件对象）

```
<template>
  <div>
    <h1>我是home</h1>
    <!-- 1.把ref绑定到组件身上 -->
    <Rc ref="com"/>
    <button @click="fun()">点我</button>
  </div>
</template>
```

```
<script>
import Rc from "@/components/refcom.vue"
export default {
  components:{
    Rc
  },
  methods:{
    fun(){
      // 2.把ref绑定到组件身上父组件就可以得到子组件的所有属性和方法
      console.log(this.$refs.com)
    }
  }
}
</script>

<style>

</style>
```

1.ref可以完成逆向传值

2.父组件如何触发子组件的方法?

使用**ref**绑定到组件即可访问

$set

在vue中 数据改变试图不变怎么解决?

在vue数据改变试图有的时候是不会更新的如下代码

```
<template>
  <div>
      <h2>数据变试图不会改变</h2>
      <h3>{{obj.age}}</h3>
      <button @click="funb()">点我</button>
  </div>
</template>

<script>
export default {
    methods:{
        funb(){
            this.obj.age=18

            console.log(this.obj.age)
        }
```

```
    },
    data(){
        return {
            obj:{
                name:"xixi"
            }
        }
    }
}
</script>

<style>

</style>
```

在**vue2.0**中 数据的双向绑定 是基于数据劫持与发布者订阅者模式的

其中数据劫持是通过**Object.defineProperty()**这个方法来拦截劫持**data**中的数据的 因为有了这个方法

所以数据改变试图也会更新

但是 **Object.defineProperty()**有个问题 他是会监听初始化的数据 如果中途给数组或者对象

添加新属性的时候 **Object.defineProperty()** 就不会监听到 不会监听到就没有数据劫持 没有数据劫持就没有双向绑定 没有双向绑定就没有数据变试图变

如果我就是想在程序运行的时候 给vue的data中对象或者数组添加新属性 并且让试图改变怎么办?

$set() 就是在程序运行的时候给对象或者数组添加新属性并且让试图改变

语法:

This.$set("你要给谁添加","你要添加的key","你要添加的val")

```
<template>
  <div>
      <h2>数据变试图不会改变</h2>
      <h3>{{obj.age}}</h3>
      <button @click="funb()">点我</button>
  </div>
</template>
```

```
<script>
// 在vue2.0中  数据的双向绑定 是基于数据劫持与发布者订阅者模式的
// 其中数据劫持是通过Object.defineProperty()这个方法来拦截劫持data中的数据的  因为有了这个方法
// 所以数据改变试图也会更新

// 但是 Object.defineProperty()有个问题   他是会监听初始化的数据  如果中途给数组或者对象
// 添加新属性的时候  Object.defineProperty() 就不会监听到  不会监听到就没有数据劫持  没有
// 数据劫持就没有双向绑定  没有双向绑定就没有数据变试图变


export default {
    methods:{
        funb(){
            // this.obj.age=18

            // console.log(this.obj.age)

            this.$set(this.obj,"age",18)
        }
    },
    data(){
        return {
            obj:{
                name:"xixi"
            }
        }
    }
}
</script>

<style>

</style>
```

动态组件

多个组件 使用同一个挂载点 并且动态切换

1.需要有多个组件

2.设置挂载点

```
<component :is="你要显示组件的变量名"></component>


   <h1>动态组件</h1>
  <component :is="Com"></component>
```

3.动态切换 修改挂载点上绑定的变量即可

```
<template>
  <div>


      <h1>动态组件</h1>
      <button @click="fun('Da')">显示a</button>
      <button @click="fun('Db')">显示b</button>
      <button @click="fun('Dc')">显示c</button>
    <component :is="Com"></component>

  </div>
</template>

<script>
import Da from "@/components/demoa.vue"
import Db from "@/components/demob.vue"
import Dc from "@/components/democ.vue"
export default {
    methods:{
        fun(val){
            this.Com=val
        }
    },
    data(){
        return {
            Com:"Dc"
        }
    },
    components:{
        Da,Db,Dc
    }
}
</script>

<style>

</style>
```

keep-alive--保存组件的状态

在路由或者动态组件中　页面用户填写的数据可能会随着页面的切换 而丢失（原因是因为 在我们每次切换的时候　**vue**都会创建一个新的组件实例 ）

如果我就是想在切换的时候保存之前的页面内容呢?

keep-alive 用来保存组件切换的时候页面状态内容　使用keep-alive包裹的组件 不会随着页面的切换 而数据丢失 因为当前组件在切换的时候会被缓存起来 那么这样一来 在组件切换的时候能减低性能上的损耗

使用

动态组件

包裹挂载点即可

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

路由

包裹路由出口即可

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

keep-alive 属性与钩子

属性

incloud 你要缓存谁

excloud 你不想缓存谁

如果 我把两个都写了 excloud的优先级大于incloude

```
    <keep-alive exclude="Db">
       <!-- 设置动态组件的挂载点 -->
       <component :is="com"></component>
     </keep-alive>
```

钩子函数

activated　　在进入被kepp-alive管理的组件时候触发

deactivated　在离开被kepp-alive管理的组件时候触发

这两个钩子应该写在被keep-alive管理的组件中 与data等属性同级

```
<template>
  <div>
    aaaaaaaa
    <input type="text" />
  </div>
</template>

<script>
export default {
  activated() {
    console.log("进入到了被keep-alive管理的组件中了");
  },
  deactivated() {
    console.log("离开到了被keep-alive管理的组件中了");
  },
};
</script>

<style>
</style>
```

nexttick

他的作用就是等待dom加载完毕执行的一个方法

```
<template>
  <div>
    <h1>nexttick</h1>
    <h1 ref="demoh">{{ text }}</h1>
  </div>
</template>

<script>
export default {
  created() {
    //    在这个实例创建完毕的钩子函数中   我如果想获取到h1里面的显示内容可以吗？
    // 但是我就是想获取怎么办？

    this.$nextTick(()=>{
        console.log(this.$refs.demoh.innerText)
    })
```

```
  },
  data() {
    return {
      text: "你好",
    };
  },
};
</script>

<style>
</style>
```