

安装chrome助手 是一个小的翻墙软件 目的允许你可以访问谷歌的商店 安装infinity 印记中文 方便去访问我们保存的一些网站站点 安装vue devtools vue的调试工具 可以让我们方便的去观察vue程序当中的数据 安装vscode vue插件 vue代码的补全

加载vue.js

vue简介

- 渐进式：类似于迭代开发，vue.js只是一些核心代码，可以让你搭建基本页面，如果你的页面功能相对比较丰富，那么需要相关的一些插件去完成。（路由、图片加载、状态管理等）
- 插件：就是一些功能代码模块。它是为了给已经完成的功能代码，额外去添加功能用的。
 - 官方插件：vuex vue-router.....官方出品的 都是vue官方的插件
 - 第三方插件：也是为了给vue去添加功能用的，但是是别的人写的。比如axios
- 数据为尊，动态显示页面

原生、jquery、vue对比

- 三者对比，vue的性能是最好的
 - 原生的和jquery，他们都获取了dom，而且在修改dom元素内容的时候，都会重绘dom树，尤其是对多个元素。
 - vue和原生及jquery是不一样的，它采用了文档碎片把整个dom树的重绘从多次变成了1次。所以效率很高。

```
<div id="app">
  <p id="pp">{{str}}</p>
</div>

<script>
  //原生
  let str = '我爱你赵丽颖'
  let pp = document.getElementById('pp')
  pp.innerHTML = str

  //jQuery
  $('#pp').html(str) // 效率和原生一样的

  //vue版本
  const vm = new Vue({
    //配置对象
    el: '#app', //挂载点是上面的div
    data: {
      str: '我爱你赵丽颖'
    }
  })
</script>
```

复习函数

- 函数首先是函数，函数还是函数对象
- 函数是有两个角色：函数可以当函数用，()，也可以当对象去用，函数.属性
- 函数当函数用可以有多种用法：
 - 普通函数调用：直接() this 代表 window
 - 构造函数调用：new + () this 代表 实例化对象
 - 方法：a.b() this 代表 调用的方法的对象
 - 回调函数：自己不用系统调用 this 代表 事件的回调代表事件源
 - call apply：让一个对象借用另外一个对象的方法 this 代表 自己给定的对象
 - 箭头函数 this 代表 外层执行上下文的this
 - 组件 this 代表 当前组件
 - vm this 代表 当前vm

基本使用

- 使用步骤：
 - 引入vue.js
 - 在body当中必须写一个挂载点
 - 实例化一个Vue的实例化对象，和挂载点进行挂载
 - 挂载点一旦被vm挂载，那么内部就不是单纯的html，被称作模板
 - 模板是由两部分组成：html + js
 - 模板语法 指令和插值
 - 指令是用来修改模板当中标签的（属性 内容 样式）v-bind
 - 插值是专门用来修改模板当中标签的内容的 {{}}
 - 请求获取回来的数据，是配置对象当中的data
- 数据代理
 - vm对象和传递的配置对象不是同一个对象
 - 配置对象：属性名不变 属性值可变（new Vue括号里的数据）
 - 数据代理：使用vm代理了配置对象当中data的数据，vm身上也有和data当中同名的属性，模板当中访问的都是vm身上的属性（vm类似中介）
 - vm代理了data当中的数据，找vm获取数据其实最终还是拿的data当中的属性值；修改vm的数据其实本质是在修改data当中的数据

```
<div id="app">
  <p>{{msg}}</p>
  <p>{{msg.toUpperCase()}}</p>
</div>
<script>
  const vm = new Vue({
    el: '#app',          //被称作挂载点 本质上是一个css的选择器字符串，标识着vm要和谁去绑定挂载
    data: {              //data存放数据来源
      msg: 'i love you ~ 赵丽颖'
    }
  })
</script>
```

- 3个重要点
 - vue控制的所有回调函数的this都是vm
 - 所有data/computed/methods中的属性/方法都会成为vm的属性/方法
 - 模板中表达式读取数据, 是读取vm的属性; 模板中调用函数, 调用的是vm的方法

数据的绑定 v-bind和事件的添加 v-on

- v-bind: 单向数据绑定 简写:
- v-on: 绑定事件 简写@

```
<div id="app">
  <p>{{msg}}</p>
  <!-- 一旦用了指令, 那么这个被指令使用的属性, 属性值就是js代码了, 而原本的""不再代表字符串的边界 -->
  <a v-bind:href="url">点击去学习</a>
  <!-- 单向数据绑定指令的简写 -->
  <a :href="url">点击去学习</a>

  <button v-on:click="test">点我</button>
  <!-- 添加事件监听的简写 -->
  <button @click="test">点我</button>
</div>
<script src="./js/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',
    data: {
      //是专门用来写属性数据的
      msg: 'i love you~ zhaoliying',
      url: 'http://www.atguigu.com'
    },

    methods: {
      //这里是专门用来写函数的
      //无论是data里面的数据 还是methods当中方法 最终都会被vm代理 (变成vm的方法)
      test(){
        alert('嘿嘿')
      }
    },
  })
</script>
```

双向数据绑定 v-model

- v-model
- MVVM 说的就是双向数据绑定模型
 - M代表model 就是我们的数据
 - V代表的view 就是我们的页面
 - VM代表的就是Vue的实例化对象
- 双向数据绑定

- 数据从data流向页面
 - 页面的数据被更新，从页面流向data
 - 当data的数据被更改以后，又会重新流向页面
- 原理：显示数据 ——再绑定修改数据的事件——只不过在html和组件标签上绑定的@input事件不同（一个是原生的一个是自定义的事件）
- 用于html表单类元素
 - html input v-model的本质
 - :value = "data" //读取数据
 - @input = "data = \$event.target.value" //写数据

```
<div id="app">
  <!-- 单向数据绑定 -->
  <input type="text" :value="msg">
  <!-- 双向数据绑定 一般情况下只针对表单类元素才使用双向数据绑定-->
  <input type="text" v-model="msg">
  <p>我爱你{{msg}}</p>
</div>
<script src="./js/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',      //挂载第一种写法：一般情况挂载都是书写配置项
    //data中可以是对象也可以是函数 函数中return一个对象
    data(){
      return {
        msg: '赵丽颖'
      }
    }
  })

  const vm = new Vue({
    data(){
      return {
        msg: '赵丽颖'
      }
    }
  }).$mount('#app') //挂载另一种写法：使用一个函数$mount
</script>
```

复习循环

- Object.keys(obj): 返回的是参数对象的属性组成的数组，再用数组的方式forEach遍历对象
- for: 最基本的循环，用来专门遍历数组，可以使用break和continue
- for in: 专门用来遍历对象的属性（数组也是对象），这个属性能否遍历要看这个属性是不是可枚举的
 - 效率最低，因为处理遍历自身还要遍历原型
- for of: 专门遍历可迭代的数据，数组有迭代器可遍历，对象没有
- forEach: 数组的方法，效率极高，但是不可以使用break和continue

```
for(let key in obj){
  console.log(key,obj[key])
}

Object.keys(obj).forEach(item=>console.log(item,obj[item]))
```

Object.defineProperty

- 数据代理、计算属性、响应式数据的原理用到
- 这个方法在为对象添加或者修改 属性为响应式属性（自动变化）

```
let obj = {
  firstName:'zhao',
  lastName:'liying'
}

Object.defineProperty(obj,'fullName',{//第一个参数：给谁添加属性；第二个参数：添加的哪一个属性
  get(){
    //当访问对象的fullName属性时候
    return this.firstName + '-' + this.lastName
  },
  set(val){
    //当设置对象的fullName属性时候
    let arr = val.split('-')
    this.firstName = arr[0]
    this.lastName = arr[1]
  }
})

console.log(obj.fullName)
obj.fullName = 'liu-yifei'
console.log(obj)
```

- 模拟数据代理

```
let vm = {}
let data = {msg:'zhaoliying'} //接口请求到的数据

Object.defineProperty(vm,'msg',{
  get(){
    return data.msg //获取vm的msg属性时，获取的其实是data的msg属性
  },
  set(val){
    data.msg = val //给vm设置msg属性时，其实是在给data设置msg属性
  }
})

console.log(vm.msg)
vm.msg = 'heihei'
console.log(vm.msg)
console.log(data)
```

computed计算属性和watch监视

第一种：复杂的插值表达式写法

- 使用js的拼接 数据在模板当中 this全部指向的是vm 只不过模板当中的this可以省略

```
<div id="app">
  姓: <input type="text" v-model="firstName">
  名: <input type="text" v-model="lastName">
  <!-- 第一种不用 使用js的拼接 数据在模板当中 this全部指向的是vm 只不过模板当中的this可以省略-->
  <p>{{firstName + '-' + lastName}}</p>
</div>
<script>
  const vm = new Vue({
    el: '#app',
    data() {
      return {
        firstName: 'zhao',
        lastName: 'liying',
      }
    }
  })
</script>
```

第二种：methods方法实现（封装函数）

```
<div id="app">
  姓: <input type="text" v-model="firstName">
  名: <input type="text" v-model="lastName">
  <!-- 第二种方法不用，封装函数去写 -->
  <p>{{getFullName()}}</p>
</div>
<script>
  const vm = new Vue({
    el: '#app',
    data() {
      return {
        firstName: 'zhao',
        lastName: 'liying',
      }
    },
    methods: {
      getFullName() {
        //在vue中所有的函数内部的this都指向vm，因为这些方法或者函数都会被vm代理
        console.log('方法被调用了')
        return this.firstName + '-' + this.lastName
      }
    }
  })
</script>
```

第三种：computed计算属性实现

- 当我需要一个数据，但是这个数据我又没有，并且这个数据由目前我有的数据计算而来的。那就要用计算属性

```
<div id="app">
  姓: <input type="text" v-model="firstName">
  名: <input type="text" v-model="lastName">
  <!-- 第三种方法重要: 计算属性 -->
  <p>{{fullName}}</p>
  <!-- 使用计算属性的set方法 -->
  <input type="text" v-model="fullName">
</div>
<script>
const vm = new Vue({
  el: '#app',
  data(){
    return {
      firstName: 'zhao',
      lastName: 'liying',
    }
  },
  //当我需要一个数据，但是这个数据我又没有，并且这个数据由目前我有的数据计算而来的。那就要用计算属性
  computed:{
    //计算属性的完整写法
    fullName:{ //计算得来的属性 与data的属性没有什么区别
      get(){
        return this.firstName + '-' + this.lastName
      },
      // 当计算属性的数据能被修改时候使用（表单类元素在双向绑定计算属性值）
      set(val){
        //目前没用
        let arr = val.split('-')
        this.firstName = arr[0]
        this.lastName = arr[1]
      }
    },
    //如果计算属性当中只有get方法，那么可以简写为如下写法
    fullName(){
      console.log('computed被调用')
      return this.firstName + '-' + this.lastName
    },
    // computed内部只能是同步返回数据，不能异步返回数据 以下写法不能返回"嘿嘿"
    fullName(){
      let a = null
      setTimeout(() => {
        a = '嘿嘿'
      },1000)
      return a
    }
  }
})
```

```
}  
</script>
```

methods和computed的区别

- 使用方法去获取姓名（第二种）和使用计算属性去计算姓名（第三种）的区别（重要）
 - 对于方法调用：你使用了几次方法调用，那么这个方法被调用了几次
 - 对于计算属性：你使用了不管多少次计算属性，计算属性的get方法只调用了一次，计算属性一定存在缓存，这样有缓存使用多次的时候效率比使用方法高的多

第四种：watch监视属性

```
<div id="app">  
  姓: <input type="text" v-model="firstName">  
  名: <input type="text" v-model="lastName">  
  <!-- 第四种方法重要: watch监视 -->  
  <p>{{fullName}}</p>  
</div>  
<script>  
  const vm = new Vue({  
    el: '#app',  
    data() {  
      return {  
        firstName: 'zhao',  
        lastName: 'liying',  
        fullName: '' //监视的时候必须有这个属性, 属性值是什么不确定, 后面根据监视去给它赋值  
      }  
    },  
    //监视的数据一定是存在的, 而且是可以变化的  
    watch: {  
      firstName: {  
        //这个对象是一个配置对象  
        //当数据发生改变的时候会自动调用handler回调  
        handler(newVal, oldVal) {  
          this.fullName2 = newVal + '-' + this.lastName  
  
          setTimeout(() => {  
            // 异步修改数据 可行  
            this.fullName = '呵呵'  
          }, 1000);  
        },  
        immediate: true //配置这个配置项的作用是无论监视到属性发生不发生变化, 都要强制的执行一次回调  
      },  
    }  
  })  
  //watch监视另一种写法  
  vm.$watch('lastName', function(newVal, oldVal) {  
    //this决定了是否可以使用箭头函数 (这里不可以, 因为使用了箭头函数, this是window不是vm)  
    console.log(this)  
    this.fullName = this.firstName + '-' + newVal  
  })  
</script>
```



```

watch:{
  todos:{
    //代表深度监视
    //一般监视和深度监视
    //一般监视监视的是数组本身的数据，但是数组内部对象的数据监视不到
    //深度监视可以监视到数组本身的数据，也可以监视到数组内部对象的数据
    deep:true
  }
}

```

computed和watch的区别

- computed是计算属性：一般是没有这个值但是想要用这个值，那么根据已有的去做计算
- watch是监视属性：一般监视属性，监视的属性，以及后期要更改的属性都必须有（有但是发生改变，或者做其他操作，用watch）
- 通常能用computed的场合都可以使用watch去解决，但是能用watch解决的computed不一定能解决
- computed函数当中只能使用同步，而watch当中可以是同步也可以是异步（重要）
- 我们去比较computed和watch的时候起始比较的是计算属性的get方法
- 计算属性的set仅仅是对计算的属性添加了监视，如果数据后期修改，会发生响应式的变化（当计算属性的值修改之后，会调用set）

列表

列表渲染 v-for

- v-for="(person,index) in persons" :key="person.id"

```

<div id="app">
  <ul>
    <!-- 展示时单项绑定person.id -->
    <li v-for="(person,index) in persons" :key="person.id">
      {{person.id}} ---- {{person.name}} ---- {{person.age}}
    </li>
  </ul>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el:'#app',
    data:{
      persons:[
        {id:1,name:'zhaoliying',age:33},
        {id:2,name:'yangmi',age:34},
        {id:3,name:'qiwei',age:40},
        {id:4,name:'dilireba',age:20}
      ]
    }
  })
</script>

```

列表过滤和排序

- arr.filter()
- 功能：从原数组当中过滤出一个新的数组
- 参数：回调函数（同步的回调）
 - 功能：对遍历的每一项执行回调函数
 - 回调函数的参数：当前项、当前项的索引、当前遍历的数组
 - 返回值：返回的是一个布尔值，（布尔值，条件表达式），根据这个布尔值的真假来决定当前遍历的这项要不要收集到新数组当中
- 返回值：返回的是新的数组
- 函数和方法最主要的是三要素功能、参数、返回值

```
<div id="app">
  <!-- 双向绑定keyword -->
  <input type="text" placeholder="请输入过滤的条件" v-model="keyword">
  <ul>
    <!-- 展示时单项绑定person.id -->
    <li v-for="(person,index) in newPersons" :key="person.id">
      {{person.id}} ---- {{person.name}} ---- {{person.age}}
    </li>
  </ul>
  <!-- 绑定事件 -->
  <!-- <button @click="test(1)">按年龄升序</button>
  <button @click="test(2)">按年龄降序</button>
  <button @click="test(0)">按原样排序</button> -->

  <!-- 函数只有一行代码时，省略函数，直接把代码写在这里-->
  <button @click="sortType = 1">按年龄升序</button>
  <button @click="sortType = 2">按年龄降序</button>
  <button @click="sortType = 0">按原样排序</button>
</div>
<script src="./js/vue.js"></script>
<script>
  Vue.config.productionTip = false //消除启动浏览器的生产提示信息
  const vm = new Vue({
    el:'#app',
    data(){
      return {
        keyword:'',
        persons:[
          {id:1,name:'zhaoliying',age:33},
          {id:2,name:'yangmi',age:34},
          {id:3,name:'qiwei',age:40},
          {id:4,name:'dilireba',age:28}
        ],
        //排序首先要设计这个数据，标志用户点击的到底是什么排序类型
        sortType:0 //0代表原样 1代表升序 2代表降序
      }
    },
    methods: {
```

```

    // 添加一个方法，将num赋值给sortType
    // test(num){
    //    //函数当中如果只有一行代码，省略函数，直接把代码写在上面
    //    this.sortType = num
    // }
  },
  computed:{
    newPersons(){
      //解构赋值 this代表vm 从vm身上获取keyword和persons
      let {keyword,persons,sortType} = this

      //根据获取的这两个数据计算新的数据
      //非箭头函数
      let arr = persons.filter(function(item,index){
        return item.name.indexOf(keyword) !== -1 //indexOf()查询元素在字符串中的位置
        //每个字符串中 都包含空串 所以一开始keyword为空串时全都显示
      })
      //箭头函数
      let arr = persons.filter(item => item.name.indexOf(keyword) !== -1)

      //在返回数组之前，做排序
      if(sortType !== 0){
        if(sortType === 1){
          return a.age - b.age //a、b是数组里的对象
        }else{
          return b.age - a.age
        }
      }

      //将上面的if判断转换为三元运算符写法
      if(sortType !== 0){
        arr.sort((a,b) => sortType === 1? a.age - b.age : b.age - a.age)
      }

      return arr
    }
  }
})
</script>

```

响应式处理（针对对象和数组的区别）

- 在vue当中 一开始data中的属性数据都是响应式的
 - 数组的数据——每个数组当中元素整体
 - 对象的数据——对象的属性
- vue当中处理响应式数据对于数组和对象是不一样的
 - 修改数组中对象的属性——响应式
 - 因为Vue一开始就为data当中所有的属性通过Object.defineProperty添加了get和set
 - 修改数组中的整个对象
 - 使用特定的几个方法——响应式

- push()、pop()、shift()、unshift()、splice()、sort()、reverse()
 - Vue给数组以上方法添加了修改页面的功能（重写数组的方法，添加了展示到页面的功能）
- 直接通过下标操作数组的数据——非响应式
 - 因为数组的数据整体并没有添加get和set方法，数据更改，但是页面没有更新
- 总结：vue在对待数组和对象的时候处理响应式是不一样的
 - 对象是通过Object.defineProperty添加了get和set
 - 数组是重写数组的方法

```
<div id="app">
  <ul>
    <li v-for="(person,index) in persons" :key="person.id">
      {{person.id}} ---- {{person.name}} ---- {{person.age}}
    </li>
  </ul>
  <button @click="update">点击修改第一项的name</button>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el:'#app',
    data:{
      persons:[
        {id:1,name:'zhaoliying',age:33},
        {id:2,name:'yangmi',age:34},
        {id:3,name:'qiwei',age:40},
        {id:4,name:'dilireba',age:20}
      ]
    },
    methods: {
      update(){
        //1、更改persons数组里的对象的属性——响应式
        this.persons[0].name = 'yingbao'
        //2、通过下标更改数组中的数据，数据更改了，但是页面没有更新——非响应式
        this.persons[0] = {id:1,name:'yingbao',age:33}
        //3、使用特定方法更改数组中的额数据——响应式
        this.persons.splice(0,1,{id:1,name:'yingbao',age:33})
      }
    },
  })
</script>
```

条件渲染 v-if/v-show

- 区别
 - v-if：条件为真的被渲染，条件为假的不渲染，条件为假的元素根本不在dom上（内存中没有）
 - v-show：条件为真的被渲染，条件为假的不渲染，但是它是真实在dom上存在的，只是采用样式display:none去做的隐藏（内存中有）
- 切换频率很高，用v-show
- 切换频率不高，用v-if

```

<div id="app">
  <p v-if="isOk">表白成功</p>
  <p v-else>表白失败</p>

  <p v-show="isOk">求婚成功</p>
  <p v-show="!isOk">求婚失败</p>

  <button @click="isOk = !isOk">点击反转</button>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el: '#app',
    data() {
      return {
        isOk: false
      }
    }
  })
</script>

```

动态绑定样式

```

<style>
  .classA{
    background-color: hotpink;
  }
  .classB{
    color: yellowgreen;
  }
  .classC{
    font-size: 80px;
  }
</style>

```

绑定class

```

<div id="app">
  <!-- 第一种情况 我不知道这个元素用的是哪个类，用哪个类是后台数据告诉我的 -->
  <!-- 动态绑定class以后，静态class仍然可以使用 -->
  <p :class="myClass" class="classB">我爱你赵丽颖</p>

  <!-- 第二种情况 这个元素绑定的是哪些类确定了，但是生效不生效由数据决定，写成对象形式 -->
  <!-- 用的最多 -->
  <p :class="{classA: isA, classB: isB}">我爱你赵丽颖</p>

  <!-- 第三种情况 这个元素一下子有好多个确定的类 -->
  <!-- 静态数据绑定 直接写 -->
  <p class="classA classB classC">我爱你赵丽颖</p>
  <!-- 动态数据绑定 v-bind -->

```

```
<p :class="['classA','classB','classC']">我爱你赵丽颖</p>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el:'#app',
    data(){
      return {
        myClass:'classB',
        isA:true,
        isB:true
      }
    }
  })
</script>
```

绑定style

```
<div id="app">
  <!-- 静态数据绑定 直接写-->
  <p style="color:aqua;font-size: 80px;">我爱你赵丽颖</p>
  <!-- 动态数据绑定 写成对象形式-->
  <p :style="{color:myColor,fontSize: mySize}">我爱你赵丽颖</p>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el:'#app',
    data(){
      return {
        myColor:'red',
        mySize:'80px'
      }
    }
  })
</script>
```

事件相关

<https://cn.vuejs.org/>

学习——教程——事件处理——事件修饰符/按键修饰符

- v-on参数
 - 不传参不需要加（），默认传递事件对象参数\$event
 - 传递参数需要加（），但是默认的事件对象参数会被传递的参数覆盖
 - 如果既需要传递参数也需要使用事件对象，那么需要传递两个参数
 - 自己的参数
 - \$event

```

<div id="app">
  <!-- 第一种 事件监听的完整写法-->
  <button v-on:click="test1">test1</button>

  <!-- 第二种 事件监听的简写 -->
  <!-- vue的模板中, 事件回调函数在调用的时候可以传递$event事件对象 (可以不写), 名字不能随意改 -->
  <button @click="test2($event)">test2</button>

  <!-- 第三种 事件在添加的时候除了事件对象之外, 还要传递自己的参数, 必须写$event, 先后顺序无所谓 -->
  <button @click="test3($event,'ym')">test3</button>

  <!-- 第四种 vue事件中阻止冒泡 事件描述符-->
  <div style="width: 300px;height: 300px;background-color: hotpink;" @click="outer">
    <div style="width: 100px;height: 100px;background-color: skyblue;" @click.stop="inner">
  </div>
  </div>

  <!-- 第五种 vue事件中取消浏览器默认行为 prevent事件描述符 -->
  <a href="http://www.atguigu.com" @click.prevent="testPrevent">点我去学习</a>

  <!-- 第六种 键盘事件添加回车事件 键盘的事件描述符-->
  <!-- 键盘事件用在表单类元素或者document身上-->
  <input type="text" @keyup.enter="testKey">
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el:'#app',
    data:{
    },
    methods: {
      test1(event){
        console.log(event.target.innerHTML)
      },
      test2(event){
        console.log(event.target.innerHTML)
      },
      test3(event,str){
        console.log(event)
      },

      outer(){
        console.log('外部')
      },
      inner(event){
        console.log('内部')
        // event.stopPropagation(); //阻止冒泡
      },

      testPrevent(event){
        console.log('i love you~')
        // event.preventDefault();
      },
    }
  })

```

```

    testKey(event){
      console.log('i love you~ z l y')
      //区分是哪个键 回车
      // if(event.keyCode === 13){
        //内部就是回车事件的逻辑
        // console.log('i love you~ z l y')
      // }
      // if(event.keyCode === 37){
        // //内部就代表是方向键的逻辑
      // }
    }
  },
})
</script>

```

自动收集表单数据

```

<div id="app">
  <!-- v-model默认收集的其实是表单类元素当中的value值 -->
  <label for="in1">用户名: </label>
  <input type="text" id="in1" v-model="userInfo.username">
</label>
  密码: <input type="password" v-model="userInfo.password">
</label>
<br>
  性别:
<label>
    男: <input type="radio" name="sex" value="male" v-model="userInfo.gender">
</label>
<label>
    女: <input type="radio" name="sex" value="female" v-model="userInfo.gender">
</label>
<br>
  爱好:
<label>
    🏀 <input type="checkbox" value="basketball" v-model="userInfo.favs">
</label>
<label>
    ⚽ <input type="checkbox" value="football" v-model="userInfo.favs">
</label>
<label>
    🏓 <input type="checkbox" value="pingpang" v-model="userInfo.favs">
</label>
<br>
  城市:
  <!-- select 这个标签它的value值是 选中的option的value值 -->
  <select v-model="userInfo.cityId">
    <option :value="city.id" v-for="(city,index) in citys" :key="city.id">{{city.name}}
  </option>
</select>
<br>

```



```

<textarea cols="30" rows="10" v-model="userInfo.desc">
</textarea>
<br>
<button @click="submit">提交</button>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      citys: [
        {id: 1, name: '北京'},
        {id: 2, name: '上海'},
        {id: 3, name: '深圳'}
      ],
      userInfo: {
        username: '',
        password: '',
        gender: '',
        favs: [],
        cityId: 1,
        desc: ''
      }
    },
    methods: {
      submit() {
        axios({
          url: '后台给你的接口地址',
          method: 'post',
          data: this.userInfo
        })
      }
    }
  })
</script>

```

checkbox的两种用法

```

<!-- 1、多个复选框，绑定到同一个数组
      那么此时每个复选框需要自己填写value的属性值，最终收集数据到一个数组当中-->
<label>
  🏀 <input type="checkbox" value="basketball" v-model="userInfo.favs">
</label>
<label>
  ⚽ <input type="checkbox" value="football" v-model="userInfo.favs">
</label>
<label>
  🏓 <input type="checkbox" value="pingpang" v-model="userInfo.favs">
</label>
<!-- 2、单个复选框，绑定到布尔值
      不需要填写value值，v-model操作的数据，对应操作的就是复选框的checked属性值-->
<input type="checkbox" v-model="isChecked" />

```

内置指令 v-text/v-html/ref

```
<div id="app">
  <p>{{msg}}</p>
  <!-- v-text无法解析文本中的标签 -->
  <p v-text="message"></p>
  <!-- v-html可以解析文本中的html标签 -->
  <p v-html="message"></p>
  <!-- ref是vue当中给我们提供的一个标识数据，通过这个标识数据取到dom元素，可操作原生js -->
  <p ref="pp"></p>
  <p :ref="$index"></p>
<button @click="test">点击获取最后一个p</button>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      msg: '我爱你赵丽颖',
      message: '<h2>i love you~</h2>'
    },
    methods: {
      test(){
        console.log(this.$refs.pp)
        this.$refs.pp.innerText = '我爱你杨幂'
        this.$refs[index]    // index是变量 要采用这种写法
      }
    },
  })
</script>
```

生命周期

- new关键字的作用
 - 在堆中开辟空间 vm此时已经存在
 - this指向这个空间
 - 执行构造器（构造函数中的代码），丰满空间中的数据
 - 返回this指向的这个空间的地址给vm
- 初始化阶段：beforeCreate created
- 挂载阶段：beforeMount mounted（常用）
- 页面数据更新阶段：beforeUpdate updated
- 销毁阶段：beforeDestroy（常用） destroyed

```
<div id="app">
  <p ref="pp" v-show="isShow">{{isShow?'我爱你赵丽颖':'我爱你杨幂'}}</p>
  <button @click="destory">点击销毁实例</button>
</div>
<script>
  new Vue({
    el: '#app',
```

```

// template: '<p>{{isShow}}</p>', //模板的另一种写法
data(){
  return {
    isShow:true
  }
},
methods: {
  destory(){
    this.$destroy() //必须在某个特定的场合下 自己手动调用vm.$destory方法, 才能进入销毁的阶段
  }
},
//初始化阶段: 两个钩子 beforeCreate created
beforeCreate() {
  //初始化前: 数据还没代理好, 打印不到, 但是vm已经存在
  console.log(this,this.isShow)
},
created() {
  //初始化后: 数据能被访问
  console.log(this.isShow)
},

//挂载阶段: 两个钩子 beforeMount mounted (挂载---把虚拟dom变成真实dom)
//挂载前: 元素真正成为dom之前
beforeMount() {
  console.log(this.$refs.pp)
},
//挂载后: 元素成为真正dom
mounted() {
  //这个钩子用的是最多的
  //一般用于发ajax请求获取数据、开启定时器、添加一些事件
  console.log(this.$refs.pp)
  this.timer = setInterval(() => {
    this.isShow = !this.isShow
  },2000)
},
//前两个阶段完成 代表初始化展示页面就完成了

//页面数据更新阶段: 两个钩子 beforeUpdate updated (不是vm的数据更新)
beforeUpdate() {
  //页面数据更新前: vm的数据已经做了更新, 但是此时页面的数据还没有更新过来
  console.log(this.isShow,this.$refs.pp.innerHTML)
},
updated() {
  //页面数据更新后: vm的数据已经做了更新, 页面的数据也更新过来了
  console.log(this.isShow,this.$refs.pp.innerHTML)
},

//销毁阶段: 两个钩子 beforeDestroy destroyed
beforeDestroy() {
  //销毁之前
  //一般这个钩子用的也是比较频繁, 通常是在销毁前解绑事件监听、取消定时器等操作
  //clearTimeout也可行
  clearInterval(this.timer) //定时器管理模块把这个定时器停止, 但是并没有把这个编号从timer当中销毁

```

```

    this.timer = null
  },
  //销毁之后
  destroyed() {
    //没什么用
    console.log('vm销毁了')
  },
})
</script>

```

过渡和动画

过渡

 image-20220408103136011

```

<style>
  *{
    margin: 0;
    padding: 0;
  }
  .box{
    width: 300px;
    height: 500px;
    background-color: hotpink;
  }
  /* 在过渡中切换的类名中, 如果使用一个没有名字的<transition>, 则v-是这些类名的默认前缀。
     如果你使用了<transition name="slide">, 那么slide会替换为slide-enter。 */
  .slide-enter{
    height: 0;
    opacity: 0;
    background-color: black;
  }
  .slide-enter-to{
    height: 500px;
    opacity: 1;
    background-color: hotpink;
  }
  .slide-enter-active{
    transition: all 5s;
  }
</style>

```

```

<div id="app">
  <!--1、理解进入和离开, 一共6个类
       2、需要过度的元素, 用transition标签包裹 (vue定义的一个子组件标签)
       3、组件标签使用时, name值决定了类名前缀
       4、变化的元素添加过渡, 就是给对应的类添加样式-->
  <transition name="slide">
    <div class="box" v-show="isShow"></div>
  </transition>

```

```

<button @click="isShow = !isShow">点击切换</button>
</div>
<script src="./js/vue.js"></script>
<script>
  new Vue({
    el: '#app',
    data(){
      return{
        isShow:true
      }
    }
  })
</script>

```

动画

- 用法: <https://cn.vuejs.org/v2/guide/transitions.html#CSS-%E5%8A%A8%E7%94%BB>

自定义

自定义指令

- 定义全局指令
- 定义局部指令

```

<div id="app">
  <p v-text="msg"></p>
  <p v-upper="msg"></p>
</div>
<div id="app1">
  <p v-text="msg"></p>
  <p v-upper="msg"></p>
</div>
<script>
  //全局指令 所有的vm都可以使用
  //参数: 1、指令名称 (不包含v- 只能是全小写 用的时候加上v-)
  //      2、回调函数
  //回调函数参数: 1、使用这个指令的那个节点 2、这个指令使用的表达式的值以及表达式的集合
  vue.directive('upper',function(node,bindings){
    console.log(node,bindings)
    node.textContent = bindings.value.toUpperCase()
  })
  new Vue({
    el: '#app',
    data:{
      msg:'i love you~ zhao li ying~'
    },
    //directives是一个对象 可以定义多个指令
    directives:{
      //局部指令 只能在定义的模板里使用
      upper(node,bindings){
        node.textContent = bindings.value.toUpperCase()
      }
    }
  })

```

```

    }
  }
})

new Vue({
  el: '#app1',
  data: {
    msg: 'i love you~ zhao li ying~'
  },
})
</script>

```

自定义过滤器

- 为了让数据进一步的计算，得到最终的结果
 - 全局过滤器
 - 局部过滤器

```

<div id="app">
  <p>{{timeNow}}</p>
  <p>{{timeNow | timeFormat}}</p>
  <p>{{timeNow | timeFormat('YYYY-MM-DD')}}</p>
  <p>{{timeNow | timeFormat('hh:mm:ss')}}</p>
</div>
<div id="app1">
  <p>{{timeNow}}</p>
  <p>{{timeNow | timeFormat}}</p>
</div>
<script src="https://cdn.bootcdn.net/ajax/libs/moment.js/2.29.1/moment.js"></script>
<script>
  //定义全局过滤器和局部过滤器
  Vue.filter('timeFormat',function(value){      //这里的value就是timeNow的数据
    return moment(value).format('YYYY-MM-DD hh:mm:ss')
  })
  //另一种写法 在参数中定义格式，可以通过传递实参改变格式
  Vue.filter('timeFormat',function(value,format='YYYY-MM-DD hh:mm:ss'){
    return moment(value).format(format)
  })
  new Vue({
    el: '#app',
    data: {
      timeNow: Date.now() //当前时间 1970年1月1日0时分0秒到现在的毫秒数
    },
    //filters是一个对象 可以定义多个
    filters: {
      timeFormat(value) {
        return moment(value).format('YYYY-MM-DD hh:mm:ss')
      }
    }
  })
  new Vue({
    el: '#app1',

```

```

    data:{
      timeNow:Date.now()
    },
  })
</script>

```

自定义插件

- 插件myPlugin.js

```

(function(w){
  //一个插件最终是一个对象
  let MyPlugin = {}
  //一个插件必须要有一个install方法
  MyPlugin.install = function (Vue, options) {
    // 1. 添加全局方法或property
    Vue.myGlobalMethod = function () {
      console.log('全局方法被调用')
    }

    // 2. 添加全局资源
    Vue.directive('upper', function(el,bindings){
      el.textContent = bindings.value.toUpperCase()
    })

    // 4. 添加实例方法（1和4不能互用）
    Vue.prototype.$myMethod = function (methodOptions) {
      console.log('实例方法被调用')
    }
  }
  //将插件挂载window身上，暴露出去
  w.MyPlugin = MyPlugin
})(window)

```

- 调用插件

```

<div id="app">
  <p v-upper="msg"></p>
</div>
<script src="./js/vue.js"></script>
<!-- 一旦引用插件 window就多了一个MyPlugin对象 因为myPlugin.js中是匿名函数自调用 引用则立马调用 -->
<script src="./myPlugin.js"></script>
<script>
  Vue.use(MyPlugin) //这句话贼重要 声明使用插件 本质是在自动调用插件当中的install方法
  Vue.myGlobalMethod()
  const vm = new Vue({
    el:'#app',
    data(){
      return {
        msg:'i love you~'
      }
    }
  })

```

```
}  
vm.$myMethod()  
</script>
```

自定义组件

- 模块和模块化：
 - 一个js文件——模块
 - 一个项目由很多个模块组成——模块化
- 组件和组件化：
 - 一个文件（可以包含html/CSS/js），比模块大——组件
 - 所有组件相当于小的vm，vm相当于一个大的组件
 - 一个项目由很多组件组成——组件化

定义非单文件组件

- 一般情况下 我们定义的都是局部组件，因为这个组件只在一个地方用到，但是当一个组件被多个组件使用的时候，定义为全局组件比较方便。
- 全局组件——复杂写法

```
<div id="app">  
  <!-- 当每次写标签使用的时候，都会偷偷的实例化定义出来的函数VueComponent的实例化对象，被称作是组件对象  
  也就是说每一个组件标签背后都会对应自己的组件对象在支撑着，都有自己的内存 -->  
  <!-- 3、使用 -->  
  <mybutton></mybutton>  
  <mybutton></mybutton>  
</div>  
<script>  
  //1、定义组件  
  //本质上是根据一个配置对象定义返回一个函数，后期是当构造函数使用  
  const VueComponent = Vue.extend({  
    //组件配置对象和vue的配置对象很相似，除了el  
    data(){  
      // 组件的配置对象当中data只能写函数  
      return {  
        count:0  
      }  
    },  
    // data:{  
    // 组件当中不能写对象  
    //   count:0  
    // },  
    template: '<div><h2 style="color:hotpink">我爱你赵丽颖</h2><button @click="count++">你爱了  
{{count}}次</button></div>  
  }  
  })  
  //2、注册（全局注册和局部注册） 本质是把一个标签的名称和刚才定义出来的函数绑定在一起  
  vue.component('mybutton',VueComponent)  
  new vue({  
    el: '#app',  
  })  
</script>
```


- 全局组件——简单写法（定义带注册）

```
<div id="app">
  <!-- 2、使用 -->
  <mybutton></mybutton>
  <mybutton></mybutton>
</div>
<script>
  //定义组件（定义带注册）
  // 1、定义带注册
  //本质上 内部还是使用extend生成一个函数，然后再去和mybutton绑定
  Vue.component('mybutton',{
    data(){
      return {
        count:0
      }
    },
    template:'<div><h2 style="color:hotpink">我爱你赵丽颖</h2><button @click="count++">你爱了{{count}}次</button></div>'
  })
  new Vue({
    el: '#app',
  })
</script>
```

- 局部组件（常用）

```
<div id="app">
  <mybutton></mybutton>
  <mybutton></mybutton>
</div>
<script>
  new Vue({
    el: '#app',
    components: {           //配置组件
      mybutton: {
        data() {
          return {
            count: 0
          }
        },
        template: '<div><h2 style="color:hotpink">我爱你赵丽颖</h2><button @click="count++">你爱了{{count}}次</button></div>'
      }
    }
  })
</script>
```

定义单文件组件

- MyButton.vue

```

<template>
  <div>
    <h2>我爱你赵丽颖</h2>
    <button>爱了{{count}}</button>
  </div>
</template>
<script>
export default {
  name: '',
  data(){
    return {
      count:0
    }
  }
}
// 一个.vue文件通常我们说是一个组件，但是本质不是，本质其实是定义组件的配置对象
</script>
<style scoped>
  h2{
    color:hotpink
  }
</style>

```

- App.vue: 所有组件拼到app中

```

<template>
  <div>
    <!-- 名字一般大驼峰 单双标签都可以-->
    <MyButton></MyButton>
  </div>
</template>
<script>
import MyButton from 'day03/components/MyButton.vue' //拿到了一个定义组件的配置对象
export default {
  name: '',
  components:{
    // 在这里才使用导入进来的配置对象定义带注册了一个组件
    // MyButton:MyButton    //前一个是标签名 后一个是组件 参考局部组件写法
    MyButton                //可省略成一个
  }
}
</script>
<style scoped>
</style>

```

```
<div id="app"></div>
<script>
  import App from './App.vue'
  new Vue({
    el: '#app',
    components: {
      App //定义带注册App
    },
    template: '<App/>'
  })
</script>
```

脚手架

安装和目录结构（参考全家桶）

- <http://nodejs.cn/> 安装node
 - 安装cnpm: `npm install cnpm -g --registry=https://registry.npm.taobao.org`
- 安装npm淘宝镜像
 - `npm config set registry http://registry.npm.taobao.org/` 安装好node后即可开始操作
 - `npm config get registry` 检查是否更换成功
- 创建一个文件夹——在文件夹路径中输入cmd
- 创建脚手架4/3的vue项目, 并运行
 - `npm install -g @vue/cli` 安装脚手架4/3的版本
 - `npm uninstall -g @vue/cli` 卸载
 - `npm install -g @vue/cli@~4.5.0` 安装指定版本
 - `vue -V` 检测是否安装成功
 - `vue create vue-demo` 使用安装的手脚手架创建一个新的vue项目（项目名称）
 - 选择manually 可以手动选择要安装的东西（按空格键选择/取消选择）
 - 选择vue 2版本 vuex route
 - history mode for router——yes
 - eslint + standard / eslint + prettier
 - lint on save
 - in dedicated config files
 - save this as a preset for future projects——no
 - 选择default vue 2版本 什么都不用管
 - `cd 项目名称`
 - `npm run serve` 运行创建的项目命令
 - 浏览器: <http://localhost:8080/>
- 如果node_modules删掉的话 `npm i` 重新安装
- 查看文件夹:
 - git仓库
 - node_modules依赖

- public公共资源
- src写代码的 (assets所有组件公用的静态资源)
- .gitignore上传的忽略文件
- package.json包文件
 - name
 - version版本号
 - script脚本 (serve启动项目的命令、build打包到本地)
 - dependencies运行依赖
 - devDependencies开发/打包依赖

脚手架3与脚手架2对比

- webpack配置
 - 2: 配置是暴露的, 我们可以直接在里面修改配置
 - 3: 配置是包装隐藏了, 需要通过脚手架扩展的vue.config.js来配置
- 运行启动命令
 - 2: npm run dev
 - 3: npm run serve

eslint禁用

- 局部禁用当前这个类型的错误 (只禁用当前文件的这个类型的报错)

```
/* eslint-disable no-unused-vars */
```

- package.json当中找到eslintConfig项, 全局配置禁用某些错误提示

```
"rules": {
    "no-unused-vars": "off"
}
```

- 开发阶段直接关闭eslint的提示功能, 手动创建vue.config.js (和package.json同级)

```
module.exports = {
  //写自己想要配置的东西去覆盖系统自带的
  //关闭ESLint的规则
  lintOnSave: false
}
```

组件模板解析 (渲染)

- Vue渲染两种方式:
 - render: h => h(App)
 - components注册组件, template解析, 但是vue导入需要导入带解析器的版本

```

import Vue from 'vue' //默认引入的就是runtime-only版本的，不带解析器
// import Vue from 'vue/dist/vue.esm.js' //是我们自己找到的带解析器的版本
import App from '@/App'
new Vue({
  el: '#app',
  components: {
    App,
  },
  template: '<App />'
})

// You are using the runtime-only build of vue where the template compiler is not available.
// Either pre-compile the templates into render functions, or use the compiler-included build.
// 我们现在默认导入的Vue是一个 runtime-only版本的Vue，这个版本的Vue不带解析器

// 这个使用render函数的 为什么就可以使用runtime-only版本的Vue
new Vue({
  el: '#app',
  render: h => h(App) //这个函数和上面我们自己写的 功能差不多
                        // 1、定义并注册了App
                        // 2、使用了App组件
                        // 3、比上面的写法多干了一件事，就是寻找解析器的loader
})

//以后我们要使用的是下面这种
//下面的打包出来的项目体积小

```

GIT

- 先有本地库再有远程库
 - 创建本地库 git init
 - 创建远程库
 - 让本地库和远程库建立联系 git remote add origin https://.....
 - 本地库有修改，推送到远程库 git add .——git commit -m"first"——git push origin master
 - 远程库有修改，拉取到本地库 git pull origin master
- 先有远程库再有本地库
 - clone git clone url地址

comments组件化开发

- 静态页面实现：
 - 拆组件：拆分页面 定义组件 最大化重用（html,css,图片拆到最细）按功能拆分
 - 组装组件：各个组件组装起来放在App.vue里面（import——components——）
 - 渲染组件：组件拼装完后，不考虑数据，先展示出来（main.js）

- 自己写的css拆分到组件中（components文件夹，再拼到app中），别人的库的css/初始化css在public的index.html中直接引入
- 动态组件界面
 - 初始化数据动态显示（和后台交互 取到数据 渲染在页面上）
 - 初始化数据分析：
 - 数据类型：一般我们的数据最终都是放在一个数组内部，数组内部放对象——[{},{},{}]
 - 数据名称：comments:[{},{},{}]
 - 定义在哪个组件（看哪个组件还是哪些个组件使用到）
 - 数据用到不是说展示就代表用，而是说数据的增删改查都叫用到数据
 - 如果这个数据只是某一个组件用的，那么数据就在这一个组件当中定义
 - 如果这个数据在某些个组件当中用的，那么就找这些个共同的祖先组件去定义
 - 组件标签名和属性名大小写问题：
 - 基本规则：要么原样去写，要么转小写中间用-连接
 - AddComment 或者
 - 交互（与用户的交互）
 - 对于数据的操作：
 - 数据在哪，操作数据的方法就要在哪定义，而不是随便的在某一个组件当中去操作数据
 - 想要操作数据的组件，可以通过调用操作数据的方法，间接去操作数据（props通信）
 - 添加和删除：子组件添加事件和事件回调，事件回调当中去调用外部操作数据的方法，数据所在的组件去添加操作数据的方法

localStorage

- localStorage 是前端本地存储的方案，是一个小型的数据库，存储到localStorage当中的东西都会自动转化为字符串
- localStorage当中有4个Api

```
//给localStorage存储数据
localStorage.setItem('键',值)
//获取localStorage中某个数据 能获取到就获取 获取不到返回null 不会影响其它
localStorage.getItem('键')
//删除localStorage当中某个数据
localStorage.removeItem('键')
//清空localStorage所有的数据
localStorage.clear()
```

通信方式

props组件通信(父传子)

- 是组件通信最常用最简单的一种方式，最基础的通信，常用
- 适用场合：适用于父子之间
- 父可以给子传递函数数据和非函数数据
 - 传递非函数数据，本质就是父组件给子组件传数据

- 传递函数数据，本质是父组件想要子组件的数据，通过函数调用传参的方式把数据传递给父组件
- 不足：不是父子就很麻烦，兄弟关系，就必须先把一个数据给了父亲，然后通过父亲再给另一个

```
<Add :addTodo="addTodo"></Add>
```

```
props: ['addTodo'] //第一种写法
props: {
  //第二种写法 限定了传递过来属性值的类型
  addTodo: Function,
  //第三种写法 也是对象写法，比第二种对象写法更加严格
  addTodo: {
    //第三种，这是一个配置对象，它可以限定属性值的更多
    type: Function,
    required: true, //代表必须传
    default: 10 //默认为10 不传的话默认值就是10
  }
}
```

- 特殊用法——路由props（具体见vue-router）
 - 路由配置 props（三种）路由组件之间没有标签，但是可以把参数通过路由映射为属性
 - 在路由器中（router文件夹的index.js）使用props
 - 布尔值：props:true——只能传递params参数
 - 对象：props:{name:'xxx',age:'20'}——传递额外的参数
 - 函数形式：props:(route)=>{ adress:route.query.adress, skuld:route.params.skuld }
 - 如果不用props，组件内要用数据必须写成this.\$route.params.xxx this.\$route.query.xxx

自定义事件组件通信(子传父)

- 和props父给子传递函数数据的时候类似
- 自定义事件和原生dom事件对比
 - 自己定义的事件
 - 事件类型：自己定义无数个
 - 回调函数：
 - 自己定义
 - 自己调用，使用\$emit触发调用的
 - 默认传参是自己给的参数，不传就是undefined

```
this.$emit('haha',10)
```

- 系统定义的事件（原生dom事件）
 - 事件类型：固定几个
 - 回调函数：
 - 自己定义
 - 触发了，系统调用浏览器去调用
 - 默认参数是事件对象event

```
box.onclick = function(event){}
```

- 适用：专门子向父通信
- 做法：
 - 在父组件当中，可以看到子组件对象（组件标签），给子组件对象绑定自定义事件\$on，回调函数的定义在父组件中
 - 在子组件当中，我们需要传递数据的地方，通过\$emit去触发自己身上父组件给绑定的事件，调用事件回调函数中传参给父组件
- 为什么父不能通过自定义事件给子传递数据？
 - 因为父组件内部可以看到子组件对象，可以给子组件对象绑定事件，回调函数在父组件定义，而子组件内部看不到父组件对象，没法给父组件对象绑定事件，子组件没法定义回调函数，但是可以看到自己身上的事件，可以触发
- 接受数据的组件必须能看到预绑定事件的组件对象，才能绑定
- 发送数据的组件必须能看到绑定了事件的组件对象（自己），才能触发事件


```
<!-- 自定义事件 父-->
<Add @addComment="addComment"></Add>
```

```
// 自定义事件 父
methods: {
  addComment(comment) {
    this.comments.unshift(comment);
  },
}
// 自定义事件 子
this.$emit('addComment',obj)
```

- 原生dom事件在html标签和组件标签上的区别（Event1组件测试）
 - 在html标签上添加就是原生的dom事件
 - 在组件标签上添加就是自定义事件，想成为原生的事件得添加修饰符.native，就是把原生dom事件添加到组件根元素上（事件委派）
- vue自定义的事件在html标签和组件标签上的区别（Event2组件测试）
 - 在html标签上添加自定义事件无意义，所以自定义事件是给组件标签添加的
 - 事件名可以任意，也可以和原生的dom事件名相同，但是在组件标签身上即使添加原生dom事件也是自定义的

vm和组件对象的关系

- vm实例化对象的原型是组件对象原型
- \$on \$emit \$off \$once等方法是在vm的隐式原型身上的（Vue的显式原型身上）
 - \$off //解绑事件
 - \$once //绑定只能触发一次的事件
- VueComponent.prototype=Object.create(Vue.prototype) ==>VueComponent的原型是Object以vue的原型为原型创建出来的

 image-20220410231524321

全局事件总线(万能)

- 适用于所有场合
- 全局事件总线本质是一个对象
- 角色标准
 - 所有的组件对象都可以看到
 - 可以使用\$on和\$emit方法
- 添加事件总线
 - 安装总线 在main.js中 new vue内部 在创建之前 beforeCreate(){Vue.prototype.\$bus = this },
 - 在接收数据的组件对象当中 获取总线给总线绑定自定义事件 this.\$bus.\$on
 - 在发送数据的组件对象当中 获取总线触发总线身上绑定的自定义事件 this.\$bus.\$emit

image-20220412152352659

- main.js

```
new Vue({
  beforeCreate() {
    //配置总线 就是把vm挂到vue的原型上，让所有的组件对象都能找到它，进而调用$on和$emit
    Vue.prototype.$bus = this
  },
  el: '#app',
  render: h => h(App)
})
```

- App.vue

```
mounted() {
  // 在App当中找到总线，给总线绑定一个事件'deleteOne'调用deleteOne方法 接数据
  // 参数: 事件名、方法
  this.$bus.$on('deleteOne', this.deleteOne)
}
methods: {
  deleteOne(index) {
    this.todos.splice(index, 1)
  }
}
```

- Item.vue

```
methods: {
  deleteOne() {
    //全局事件总线 发数据
    //参数: 事件名、数据
    this.$bus.$emit('deleteOne', this.index)
  }
}
```

消息的订阅和发布pubsubjs(万能)

- react框架中使用的较多
- install pubsub-js -S
- App.vue

```
import PubSub from 'pubsub-js'
mounted(){
  //消息订阅处理修改单个 订阅一个消息:接收数据
  PubSub.subscribe('heihei',this.updateOne)
}
methods:{
  // msg是消息订阅和发布 传递过来的消息名称,即使不用 也要占位
  updateOne(msg,index){
    this.todos[index].isOver = !this.todos[index].isOver
  }
}
```

- Item.vue

```
import PubSub from 'pubsub-js'
methods:{
  updateO(){
    //消息的发布 发布消息组件中传递数据
    PubSub.publish('heihei',this.index)
  }
}
```

slot插槽(父子)

- 通信的一般是结构 (标签)
- Child.vue

```
<template>
  <div>
    <h2>我爱你赵丽颖</h2>
    <!-- vue当中内置的组件标签 -->
    <slot>
      <!-- 默认插槽 默认插槽只能有一个-->
      <!-- slot内部的东西 是等待着父组件使用的时候给传过来 -->
      <span>嘿嘿</span>
    </slot>

    <slot name="xxx">
      <!-- 具名插槽 -->
    </slot>
    <slot name="yyy">
      <!-- 具名插槽 -->
      <ul>
        <li>我爱你赵丽颖1</li>
        <li>我爱你赵丽颖2</li>
      </ul>
    </slot>
  </div>
</template>
```

```

</slot>

<!-- 无论是默认插槽还是具名插槽，里面都可以写东西，也可以不写东西
如果写了东西，就看你用的时候有没有给slot传递新的东西，如果传递了，slot当中写的东西就被覆盖了
如果你没有给slot传递新的东西，那么默认显示的就是slot当中的东西
-->
</div>
</template>

```

- ScopedChild.vue

```

<template>
  <div>
    <ul>
      <li v-for="(todo,index) in todos" :key="todo.id">
        <slot :todo="todo" :index="index">
          {{todo.content}}
        </slot>
      </li>
    </ul>
  </div>
</template>
<script>
export default {
  name: '',
  props:['todos']
}
</script>

```

- App.vue

```

<template>
  <div>
    <Child>
      <template>
        <button>点我</button>
      </template>
      <template slot="xxx">
        <a href="https://www.baidu.com">点我去百度</a>
      </template>
    </Child>
    <Child>
      <template>
        <p>pp</p>
      </template>
      <template slot="xxx">
        <span>我爱你</span>
      </template>
    </Child>
  <!-- 作用域插槽 -->
  <!-- 数据是由父组件传给子组件去展示的
子组件展示数据的过程当中，数据的结构是由父组件说了算的

```

```

-->
<!-- 如果是isOver为true的前面有√ 并且颜色是hotpink
如果没有为true的, 原样显示
-->
<ScopedChild :todos="todos">
  <!-- <template slot-scope="scopeProps"> -->
  <template slot-scope="{todo,index}">
    <!-- <span v-if="scopeProps.todo.isOver" style="color:hotpink">√
  {{scopeProps.todo.content}}</span> -->
    <span v-if="todo.isover" style="color:hotpink">√ {{todo.content}}</span>
  </template>
</ScopedChild>
</div>
</template>
<script>
import Child from '@components/Child'
import ScopedChild from '@components/ScopedChild'
export default {
  name: '',
  components:{
    Child,
    ScopedChild
  },
  data(){
    return {
      todos:[
        {id:1,content:'抽烟',isover:false},
        {id:2,content:'喝酒',isover:true},
        {id:3,content:'烫头',isover:false},
      ]
    }
  }
}
</script>

```

- 作用域插槽

- 数据在父组件当中, 数据是要给子组件去展示 (v-for) , 展示的过程中, 数据的结构由父组件决定
- 步骤
 - 父组件传递数据给到子组件, 子组件props接收数据, 然后通过遍历展示数据
 - 数据的结构或样式由父组件决定, 此时需要将遍历过的数据重新传递给父组件, 将要改变的数据部分用slot包裹, 在slot标签中将数据传递给父组件
 - 父组件接收到子组件传递过来的数据, 在template标签中, 使用slot-scope='{解构出来的数据}', 然后进行样式或结构的改变

```

<!--父组件中:isComplete:true或false
1、先将数据传递给子组件 :todos='todos'
4、父组件接收slot-scope="{todo}"子组件传递的数据, 然后改变结构-->
<List :todos='todos'>
  <!--slot-scope后面的值是把传过来的数据包裹成一个对象, 用{}解构-->
  <template slot-scope="{todo}">
    <li :style="{color:todo.isComplete?'red':'yellow'}">{{todo.text}}</li>
  </template>
</List>

```

```

<!--子组件中:首先props:['todos']
2、在子组件接收后 遍历展示
3、将遍历后的数据需要发生结构变化的数据传递给父组件-->
<li v-for="(todo, index) in todos" :key="index">
  <slot :todo='todo' :index='index'>
    {{todo.text}}
  </slot>
</li>

```

Vuex(万能)

 image-20220415100604719

- Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式，是一个官方插件。它采用集中式存储管理应用的所有组件的状态（数据），并以相应的规则保证状态以一种可预测的方式发生变化
 - 我们也可以认为它也是一种组件间通信的方式，并且适用于任意组件
- 理解：对vue应用中多个组件的共享状态（数据）进行集中式的管理(读/写)
- 为什么要用vuex
 - 多个视图依赖于同一状态（多个组件共同读一个数据）
 - 来自不同视图的行为需要变更同一状态（多个组件共同改一个数据）
 - 以前的解决办法
 - 将数据以及操作数据的行为都定义在父组件（App）
 - 将数据以及操作数据的行为传递给需要的各个子组件（有可能需要多级传递）
- 什么时候用
 - Vuex 可以帮助我们管理共享状态，并附带了更多的概念和框架。这需要对短期和长期效益进行权衡。
 - 也就是说应用简单（组件比较少）就不需要使用（但是可以），如果应用复杂，使用就会带来很大的便捷
- 核心：把所有的共享状态数据拿出来放在Vuex中进行集中式管理
- 使用
 - 安装vuex：npm i vuex -S
 - 查看可安装版本：npm view vuex versions
 - 指定版本安装：npm install vuex@3.6.2
 - 建一个和vuex相关的文件夹——store——index.js
 - 在index.js中配置vuex相关，引入vuex并声明使用vuex插件
 - 向外暴露一个store的实例化对象
 - 将暴露出去的store实例化对象引入到实例化Vue的配置对象（main.js）当中使用

- 书写store对象当中包含的4个核心概念
- 简单代码
 - store——index.js

```
//2、引入并声明使用vuex插件
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
// state是一个包含多个属性（不是方法）的对象，用来存储数据
const state = {
  count: 0,
};
// mutations是一个包含了多个方法的对象，用这个里面的方法去直接操作数据
// 这个方法不能包含 if、for、异步，是直接操作的
const mutations = {
  INCREMENT(state) { //一般大写
    state.count++;
  },
  DECREMENT(state) {
    state.count--;
  },
};
// actions是一个包含了多个方法的对象。这个对象内部的方法用来和vue当中用户的操作去关联（比如点击等）
// 这个方法可以包含 if、for、异步。
// 如果是actions当中的方法，第一个参数一定是context，就算不用也要占位
// 第二个参数才是dispatch的时候传递过来的参数
const actions = {
  // context 上下文对象 本质其实就是store对象
  increment(context) {
    context.commit("INCREMENT"); //store对象.commit方法
  },
  decrement({ commit }) { // {commit}从store对象中把commit解构出来
    commit("DECREMENT");
  },
  incrementOfOdd({ commit }) {
    if (state.count % 2 === 1) {
      commit("INCREMENT");
    }
  },
  incrementAsync({ commit }) {
    setTimeout(() => {
      commit("INCREMENT");
    }, 1000);
  },
};
// getters是一个包含了多个方法的对象。这个对象内部的每个方法对应了一个计算属性的get
// 通过state当中的已有数据 计算出来的一个新的想要使用的属性数据
const getters = {};
//3、向外暴露一个store的实例化对象
export default new Vuex.Store({
  state,
  mutations,
```

```

    actions,
    getters,
  });

```

o main.js

```

import Vue from "vue";
import App from "@/App";
import store from "@/store" //store是个文件夹 默认会找index
Vue.config.productionTip = false;
new Vue({
  ...
  //4、将暴露出去的store实例化对象引入到实例化Vue的配置对象main.js当中使用
  store //store:store简写 store对象在配置对象当中配置使用，每个组件对象当中都可以通过
  this.$store获取到这个对象
});

```

o App.vue

```

<template>
  <div>
    <button @click="increment">+1</button>
    <button @click="decrement">-1</button>
    <button @click="incrementOfOdd">是奇数就+1</button>
    <button @click="incrementAsync">一秒后+1</button>
    <!-- this可省略 -->
    <h1>{{ this.$store.state.count }}</h1>
  </div>
</template>

<script>
export default {
  name: "",
  methods: {
    increment() {
      //dispatch 和 emit 都是触发分发的意思 分发触发actions当中对应的方法
      //dispatch在触发的时候本质就是在调用的actions当中的方法，可以传递参数，但是只能传递一个参
      数
      //如果要传递多个，请把多个参数封装为对象再传：{keyword,name:'zly'}
      this.$store.dispatch("increment");
    },
    decrement() {
      this.$store.dispatch("decrement");
    },
    incrementOfOdd() {
      this.$store.dispatch("incrementOfOdd");
    },
    incrementAsync() {
      this.$store.dispatch("incrementAsync");
    },
  },
};

```

```
</script>

<style scoped>
</style>
```

- 映射代码

- App.vue

```
<template>
  <div>
    <button @click="increment">+1</button>
    <button @click="decrement">-1</button>
    <button @click="incrementOfOdd">是奇数就+1</button>
    <button @click="incrementAsync">一秒后+1</button>
    <h1>{{ count }}</h1>
  </div>
</template>

<script>
import { mapActions, mapState } from "vuex"; //导入
export default {
  name: "",
  //映射属性数据，无论是state的数据还是getters当中的方法都映射到computed中
  computed: {
    ...mapState(['count'])
  },

  //映射方法，无论是actions还是mutations的方法都映射到methods中
  //下方methods内容的简写（几乎不用）
  methods: mapActions(['increment', 'decrement', 'incrementIfOdd', 'incrementAsync'])

  methods: {
    // 当回调函数名字和actions当中方法名字一样的时候，这个函数允许传递一个数组——自动映射
    // 不加...时返回的是一个对象，加上...进行拆包
    ...mapActions(["increment", "decrement", "incrementOfOdd", "incrementAsync"])

    // increment() {
    //   this.$store.dispatch("increment");
    // },
    // decrement() {
    //   this.$store.dispatch("decrement");
    // },
    // incrementOfOdd() {
    //   this.$store.dispatch("incrementOfOdd");
    // },
    // incrementAsync() {
    //   this.$store.dispatch("incrementAsync");
    // },

    //当回调函数名字和actions当中方法名字不一样的时候——使用对象
    ...mapActions(["decrement", "incrementIfOdd", "incrementAsync"]), //一样的传数组
    ...mapActions({ //不一样的传对象
```



```

        increment() {
            this.$store.dispatch("iincrement");
        },
    },
};
</script>

<style scoped>
</style>

```

\$attrs和\$listeners

- 本质是父组件中给子组件传递的所有属性组成的对象及自定义事件方法组成的对象
 - 父向子传递属性数据和事件绑定监听的数据
- \$attrs、\$listener用来自己封装组件使用
 - \$attrs取的是除了class、style属性以及props接收过的属性之外，所有的属性组成的一个对象
 - \$listener取的是父组件中给子组件标签绑定的所有自定义事件监听组成的对象
- 可以通过v-bind 一次性把父组件传递过来的属性添加给子组件
- 可以通过v-on 一次性把父组件传递过来的事件监听添加给子组件

```

<template>
  <a href="javascript:;" :title="title">
    <el-button v-bind="$attrs" v-on="$listeners"></el-button>
  </a>
</template>

```

v-model深入

- 用于组件标签（实现父子组件双向数据同步问题）
- 本质上还是自定义事件和props组合
- 组件标签上 v-model本质
 - :value = "data" //父组件传递属性给子组件，子组件需要接受
 - @input = "data = \$event" //父组件当中给子组件添加的自定义事件 数据在父组件当中
 - 子组件当中必须这样写
 - 先接受props:['value']
 - 子组件表单类元素
 - :value = "value"
 - @input = "\$emit('input',\$event.target.value)"

属性修饰符sync

- 实现父子组件双向数据同步问题，和 v-model 实现效果几乎一样，本质上还是自定义事件和props组合
 - v-model一般用于带表单项的组件
 - sync属性修饰符一般用于不带表单项的组件
- 父组件给子组件属性传递数据后面添加.sync

- 子组件修改数据，需要分发事件@click = \$emit("update:属性名",要更新的数据)
 - <button @click="\$emit('update:moeny',moeny-100)">花钱

\$parent 和 \$children以及\$ref

- \$ref
 - 给子组件标签使用ref \$refs可以直接操作子组件内部的数据及方法
 - 通过this.\$refs.son可以拿到组件对象本身，可以直接操作子组件对象的数据和方法
 - 如果需要修改data数据，可以直接修改this.\$refs.son.msg=XXX
 - 给html标签使用ref

拿到的是html标签本身的dom元素this.\$refs.pp

```
<SpuForm v-show="scene == 1" @changeScene="changeScene" ref="spu" />
// 通过$refs获取子组件spu中的方法
this.$refs.spu.initSpuData(row);
```

- 以下方法慎用
 - \$children：所有子组件对象的数组
 - 父组件当中可以通过\$children找到所有的子组件去操作子组件的数据（当然可以找孙子组件）
 - 找子组件时\$children 是将子组件对象放入数组中 不能通过索引操作 因为子组件对象顺序不固定
 - \$parent：代表父组件对象
 - 子组件当中可以通过\$parent找到父组件（当然可以继续找爷爷组件）操作父组件的数据
 - 找父组件时\$parent 存在组件共用 此时可能不是一个父组件 会存在多个父组件

复习axios

- axios是根据xhr和promise封装而来的一个http库（插件）
- http请求包括普通http请求 和 ajax请求
- 用来发送ajax请求最多的工具 就是axios
- 引入axios之后暴露给我们的是一个函数
 - 安装：npm install axios -S
- axios是一个函数 也是函数对象 有两种用法 函数用法和对象用法
 - axios() 函数用法
 - axios.get() 函数对象的用法
- 请求方式：
 - 一般普通的http请求就是get和post
 - ajax请求：get post put delete
- 前后端分离
 - 前端做前端的：搭页面、做交互（请求获取数据展示渲染）
 - 后端做后端的：数据库的存储操作，接口的书写，服务器的部署
- restful Api 接口规范 是现在目前最流行的书写后端接口的一个规范
- 资源化

- 每个数据都会被看作是一个资源（商品） 对于每一个资源来说操作数据库有4个情况 增 删 改 查
- 后端人员对数据库的每个操作 对应 前端人员ajax的4个请求
 - get: 调用接口 查找数据资源
 - post: 调用接口 添加数据资源
 - put: 调用接口 修改数据资源
 - delete: 调用接口 删除数据资源
- axios三种参数
 - params: 参数是路径的一部分，并且这个参数只能在url路径当中出现
 - query: 查询参数，这个参数可以出现在url当中，也可以在配置项当中配置.query参数不属于路径的一部分
 - 在url当中 是? 后面的 key = value & key = value,
 - 在配置项当中 配置项的名称叫params
 - body请求体参数
 - 通常用在跑post和put当中，只能在配置对象当中配置
 - data 这个配置项就是body请求体参数 这个数据必须是一个对象
- url: 协议 ip 端口 路径（可以包含params参数） ?query参数

```
<body>
<script src="https://cdn.bootcdn.net/ajax/libs/axios/0.21.0/axios.js"></script>
<script>
  //函数写法 返回的是一个promise
  axios({
    //配置对象
    // url:'https://api.github.com:80/search/repositories?q=v&sort=stars',
    url:'https://api.github.com/search/repositories',
    method:"get",
    params:{
      q:'v',
      sort:'stars'
    }
  }).then(response => {
    console.log(response.data)
  }).catch(error => {
    console.log(error)
  })
  // 如果是对象写法，请求方式是对象的方法名，第一个参数都代表url（params和query都直接写在url当中）
  // 如果有body请求体参数，post和put 第二个参数代表的就是请求体参数。
  // get delete 第二个参数不是请求体参数，代表是一个配置对象
  axios.get('https://api.github.com/search/repositories?q=v&sort=stars')
  axios.post('https://api.github.com/search/repositories?q=v&sort=stars',{请求体参数})
</script>
</body>
```

复习async与await

- async await: 使用同步代码实现异步效果
- async函数代表这是一个异步函数，async函数返回的是promise

- async函数返回值不看return 必然返回promise
- async函数返回的promise是成功还是失败 看return
- return的结果代表promise是成功还是失败
 - 如果return的是一个非promise的值 代表async函数返回的promise是成功的
 - 成功的结果是return的结果
 - 如果返回的是成功的promise 代表async函数返回的promise也是成功的（他们不是同一个promise）
 - 成功的结果是return的promise的成功结果
 - 如果返回的是失败的promise 代表async函数返回的promise是失败的
 - 失败的原因是return的promise失败的原因
 - 如果throw出错误，代表代表async函数返回的promise是失败的
 - 失败的原因是抛出的错误原因

```
async function add(a,b) {
  throw new Error('嘿嘿')
  return a + b
}
console.log(add(10,20))
```

响应状态码

image-20220423171459893跨域

- 浏览器上的同源策略验证规则
- 特点：
 - 跨域只存在于浏览器，不在浏览器发请求不存在跨域问题
 - http请求分为两大类：普通http请求和ajax请求（跨域是出现在ajax请求）
 - 普通请求和ajax请求区别
 - 普通请求：一般只有get（a标签和地址栏输入回车）和 post(form表单) 页面会刷新 不会跨域
 - ajax请求：一般 get post delete put 一般都是异步发送的 页面不刷新 局部更新
- 在什么地方会出现跨域
 - 浏览器给服务器发ajax请求会跨域 因为跨域（同源策略）只存在于浏览器
 - 服务器给服务器发ajax请求不会跨域
- 什么条件会跨域
 - 同源（协议 ip 端口一致）不跨域
 - 不同源（三个中间有一个不一样）就跨域
- 解决跨域：前端可以解决、后端解决。一般后端解决比前端解决容易

代理

- 正向代理：给前台解决问题，用户目标明确（例如翻墙）
- 反向代理：给后台解决问题

配置代理解决跨域

- 本身我们现在就跑到开发服务器 webpack-dev-server,而这个服务器带了一个模块, 这个模块可以支持我们使用代理
- 原理: 在浏览器发请求的时候, 把这个请求发给服务器上的这个代理模块, 再由这个代理模块转发给我们真正的服务器。这样的话, 我们原来由浏览器直接发送请求到服务器就转化为服务器到服务器之间的请求
- 要让代理转发, 那么得告诉代理这个请求什么情况需要转发, 配置以固定什么开头的路径需要代理转发, 代理看到这个路径是以它开头就会帮你转发
- 代理转发的时候会把路径交给真正的请求服务器, 作为请求路径, 需要把固定的开头去除

 image-20220413212509894

 image-20220413213246807

- vue.config.js 配置完需要重启

```
devServer:{
  proxy: {
    "/api": {
      // 本身: 8080/api/users/info
      // target 代表转发的目标服务器 4000/api/users/info
      // pathRewrite 4000/api/users/info 和真正的地方多了个/api 需要把/api剥掉
      target: "http://localhost:4000",
      pathRewrite: {"^/api" : ""}, //4000/users/info
      changeOrigin:true //不管改变哪个跨域的条件都会转发
    }
  }
}
```

```
axios({
  url: "http://localhost:8080/api/users/info",
  method: "get",
});
```

element-ui

- 安装: npm i element-ui -S
- 完整引入

```
//全局引入并注册组件库当中定义好的各个组件
import ElementUI from 'element-ui';
import 'element-ui/lib/theme-chalk/index.css';
Vue.use(ElementUI)
```

- 按需引入
 - 首先安装 npm install babel-plugin-component -D
 - 脚手架3: babel.config.js / 脚手架2: .babelrc

```
{
  "presets": [["es2015", { "modules": false }]], //一般不用改
  "plugins": [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "theme-chalk"
      }
    ]
  ]
}
```

◦ 按需引入

```
// 全局引入部分的element-ui组件并且进行全局注册
// element-ui当中的组件分为两大类：按使用方式区分  组件标签形式的组件 和  js代码形式的组件
// 组件标签形式的组件
import { Button, Select } from 'element-ui';
Vue.component(Button.name, Button);
Vue.component(Select.name, Select);
// 或
Vue.use(Button)
Vue.use(Select)
// js代码形式的组件
Vue.prototype.$loading = Loading.service;
```

- element-ui的button添加click事件会触发，添加dblclick就不会触发
 - element-ui的button 子组件内部触发了这个单击事件
 - element-ui的button 子组件内部没有触发这个双击事件 必须在@dblclick.native转化为原生dom事件 就可以触发双击事件

```
<el-button type="primary" icon="el-icon-edit" @click="test1">点击</el-button>
<el-button type="primary" icon="el-icon-delete" @dblclick.native="test1">点击</el-button>
```

vue-router

- 是什么
 - 是vue官方的一個插件
 - 专门用来实现一个SPA应用
 - 单页Web应用（single page web application, SPA）
 - 整个应用只有一个完整的页面（这个完整的页面，由多个组件组成）
 - 点击页面中的链接不会刷新页面, 本身也不会向服务器发普通请求
 - 当点击路由链接时, 只会做页面的局部更新（组件切换）
 - 数据都需要通过ajax请求获取, 并在前端异步展现
 - 基于vue的项目基本都会用到此库
 - vuex vue-router 这两个插件应用比较广泛

- 路由：key:value的映射关系，路径与函数/组件的映射关系

- 前台路由：与组件的映射关系

- 当点击链接的时候，路径会发生变化，但是不会向服务器发普通请求，而是去显示对应的组件，显示组件过程当中发ajax请求，把数据渲染在组件当中
 - 简单理解前台路由：路由可以实现组件的切换和跳转：点击链接——匹配路由——显示对应的组件

```
//路径 和 要显示的组件
{
  path: '/home',
  component: Home
}
```

- 后台路由：与函数的映射关系

- 当点击链接（a标签）的时候，路径会发生变化，而且会向服务器发普通请求，然后匹配到后端的一个函数处理这个路由的请求，返回需要的数据

```
//路径 和 匹配的函数
app.get('/users/info',function(){})
```

- 路由组件和非路由组件

- 组件：一个组件包含 html css js img的结合体 定义 注册 使用

- 路由组件：定义 注册（不是在另外一个组件当中注册，是在路由当中注册的） 使用

```
<router-link></router-link>用户点击的链接
<router-view></router-view>组件切换的地方
```

- 非路由组件：定义 注册（一定是在另外一个组件当中去注册的） 使用

```
<Header />
```

- 最大区别

- 除了在注册和使用的时候这两种组件有区别，生命周期有很大区别的
 - 路由组件的生命周期是点击链接的时候，才开始的，路由组件才会创建，mounted才能执行，路由组件在切换的时候，会被销毁，显示的时候重新创建
 - 同一个路由组件传参显示不同数据，mounted回调只会执行一次，因为是同一个组件

- 使用

- 拆分页面定义组件：路由组件（components中）和非路由组件（views中）

- 路由器当中定义路由：

- 安装：npm i vue-router -S
 - 查看可安装版本：npm view vue-router versions
 - 指定版本安装：npm install vue-router@3.6.2
 - 建一个和vue-router 相关的文件夹——router——index.js
 - 引入VueRouter并声明使用VueRouter

- 实例化一个路由器对象并暴露
- 将暴露出去的路由器实例化对象引入到new Vue的配置对象（main.js）当中使用
- new路由器时候配置对象当中的代码
- 使用路由实现组件切换（在路由链接的文件中 App.vue）
 - router-link 路由连接，就是点哪，可以让你的路径变为你指定的 to
 - router-view 路由组件显示区域，就是组件需要在哪显示
- 路由传参
 - 把参数写在路径当中（三种）
 - 最原始的传参
 - 参数：params参数，是属于路径的一部分 /message/10
 - query参数路径后使用?去拼接起来的 /xxx/?aa=bb&&xx=yy
 - 路由链接组件传递数据给命名路由
 - 路由链接组件中给路由传参可以写成对象形式，前提需要给路由起名字name，也叫命名路由
 - 点击路由链接的时候，路径会去路由器当中的路由当中匹配，匹配同时会把参数解析添加到路由对象当中
 - 匹配成功，显示对应的路由组件同时把当前的路由对象传递到路由组件当中，我们就可以从路由对象当中获取参数（props）
 - 使用props简化路由传参给子组件操作（路由当中传参的三种操作）
 - 布尔值：路由当中需要配置 props:true，只能接收params参数，它会把路由当中接收的参数，置为子组件的属性
 - 对象：很少用，只能额外的给子组件传递默认静态值
 - 函数：用的比较多，比较灵活，可以把params和query的参数都映射为子组件的属性

```

props(route){ //route就是当前我这个路由对象
  //把路由对象当中的参数，不管什么参数全部拿到作为子组件的属性去使用
  return {
    msgId:route.params.msgId,
    msgContent:route.query.msgContent
  }
}

```

- 缓存路由组件
 - 使用这个东西可以保证我们在切换组件的时候，原来显示的组件不被销毁
 - 找到互相切换两个组件的父文件

```

<!-- include="Message" 包含这个名字（组件对象当中的name值）的组件，会被缓存-->
<keep-alive include="Message">
  <router-view></router-view>
</keep-alive>

```

- 程式化导航和声明式导航
 - 声明式导航：借助router-link自动生成的跳转方式去跳转的
 - 程式化导航：自己手写代码，去跳转
 - 方法

- `this.$router.push(path)`: 相当于点击路由链接(可以返回到当前路由界面)

```
this.$router.push('/home/news/newsdetail/'+news.id + '?content=' +
news.content)
this.$router.push(`/home/news/newsdetail/${news.id}?content=${news.content}`)
this.$router.push({name: 'newsdetail', params: {newsid: news.id}, query:
{content: news.content}})
```

- `this.$router.replace(path)`: 用新路由替换当前路由(不可以返回到当前路由界面)
- `this.$router.back()`: 请求(返回)上一个记录路由
- `this.$router.go(-1)`: 请求(返回)上一个记录路由
- `this.$router.go(1)`: 请求下一个记录路由
- `$router.push()`和`$router.replace()`的区别, 返回有区别
 - `$router.push()`是往历史记录里面追加
 - `$router.replace()`每一次都是覆盖添加
- 路由模式
 - hash模式
 - 路径中带#: <http://localhost:8080/#/home/news>
 - 发请求的路径: <http://localhost:8080> 项目根路径
 - 响应: 返回的总是index页面 ==> path部分(/home/news)被解析为前台路由路径
 - history模式
 - 路径中不带#: <http://localhost:8080/home/news>
 - 发请求的路径: <http://localhost:8080/home/news>
 - 响应: 404错误
 - 希望: 如果没有对应的资源, 返回index页面, path部分(/home/news)被解析为前台路由路径
 - 解决: 添加配置
 - devServer添加: `historyApiFallback: true`, // 任意的 404 响应都被替代为 index.html
 - output添加: `publicPath: '/'`, // 引入打包的文件时路径以/开头

mixin

- html、js、css 相同: 封装组件
- 单个组件js代码重复: 封装函数
- 不同的组件js代码重复: 封装混合
 - 步骤
 - 新建一个mymixin.js文件, 在js文件中暴露一个对象, 对象内部可以有data、methods、computed... 会将js文件中暴露出的数据、方法等混入到组件内部

```
export const xxxMixin = {
  methods: {
    //重复的代码写在这
  }
}
```

- 在组件内部引入 import myminxi from './myminxi.js'
- 使用mixins:[mymixin] 例如:

```
import mymixin from './mymixin'
export default {
  name: 'Daughter',
  mixins: [mymixin],
}
}
```