

渐进式解释

渐进式意味着你可以将Vue作为你应用的一部分应用入其中，分阶段的将几部分应用vue，带来简更丰富的交互体验。（或者说由浅及深，一点点了解）

安装



Vue.js安装

- 使用一个框架，我们第一步要做什么呢？安装下载它
- 安装Vue的方式有很多：
- 方式一：直接CDN引入

□ 你可以选择引入开发环境版本还是生产环境版本

```
<!-- 开发环境版本，包含了有帮助的命令行警告 -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<!-- 生产环境版本，优化了尺寸和速度 -->
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

- 方式二：下载和引入

开发环境 <https://vuejs.org/js/vue.js>
生产环境 <https://vuejs.org/js/vue.min.js>

- 方式三：NPM安装

■ 后续通过webpack和CLI的使用，我们使用该方式。

杂乱简介

范式

vue编程范式：**声明式编程**

原来js变成范式：**命令式编程**

声明式编程会将数据和页面完全分离

el、data、methods

示例：

```
</head>
<body>

  <div id="app">
    <h2>{{message}}</h2>
    <h1>{{name}}</h1>
  </div>

  <div>{{message}}</div>

  <script src="../js/vue.js"></script>
  <script>
    // let(变量)/const(常量)
    // 编程范式: 声明式编程
    const app = new Vue({
      el: '#app', // 用于挂载要管理的元素
      data: { // 定义数据
        message: '你好啊, 李银河!',
        name: 'coderwhy'
      }
    })
  </script>
```

创建Vue类的一个实例app

el属性: 该属性决定了这个Vue对象挂载到哪一个元素上, 很明显, 这里挂载到了id为app的元素上

data属性: 该属性中通常会存储一些数据

这些数据可以是我们直接定义出来的, 比如像上面这样。

也可能是来自网络, 从服务器加载的。

methods属性: 将vue对象的一些方法定义到methods属性中去

可以直接写一个方法 (相当于**方法名:匿名函数**)

```
methods: {
  add:function(){
    this.counter++
  },
}
```

也可以直接写一个函数, 连function也不写, 之前没这么写过 (这是es6的函数增强写法)

```
},
methods: {
  getFullName() {
    return this.firstName + ' ' + this.lastName
  }
},
```

响应式的意思

vue的**响应式**的意思就是, 比如上面定义的Vue类的一个实例app, 修改data里面的属性值, 界面就会自动进行修改

v-for简单示例：

v-for后面的双引号中格式： "(数组项,数组下标) in 数组名"

或者"数组项 in 数组名"

后面在data中追加元素的时候，**追加的也可以在列表中渲染出来**

```
</head>
<body>

<div id="app">
  <ul>
    <li v-for="item in movies">{{item}}</li>
  </ul>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      movies: ['星际穿越', '大话西游', '少年派', '盗梦空间']
    }
  })
</script>
```

语法糖的意思

@click就相当于v-on:click，@click是v-on:click的语法糖，（语法糖就是一种语法的简写形式，因为简写，相当于尝到了甜头，取名为语法糖）

v-on:click=""，引号之间直接写methods里面方法的名字（也就相当于函数）

计数器例子

```
<body>
  <div id = "app">
    <h2>当前计数:{{counter}}</h2>
    <!-- <button type="button" v-on:click="counter++">+</button>
    <button type="button" v-on:click="counter--">-</button> -->
    <button type="button" v-on:click="add">+</button>
    <button type="button" @click="sub">-</button>
  </div>
  <script src=".js/vue.js"></script>
<script>
  const app = new Vue({
    el:'#app',
    data:{
      counter:0
    },
    methods:{
      add:function(){
        this.counter++
      },
      sub:function(){
        this.counter--
      }
    }
  })
</script>
```

vue中的MVVM

MVVM是Model-View-ViewModel的简写。它本质上就是MVC 的改进版

mvvm一般指MVC框架。经典MVC模式中， M是指业务模型， V是指用户界面， C则是控制器

■ View层:

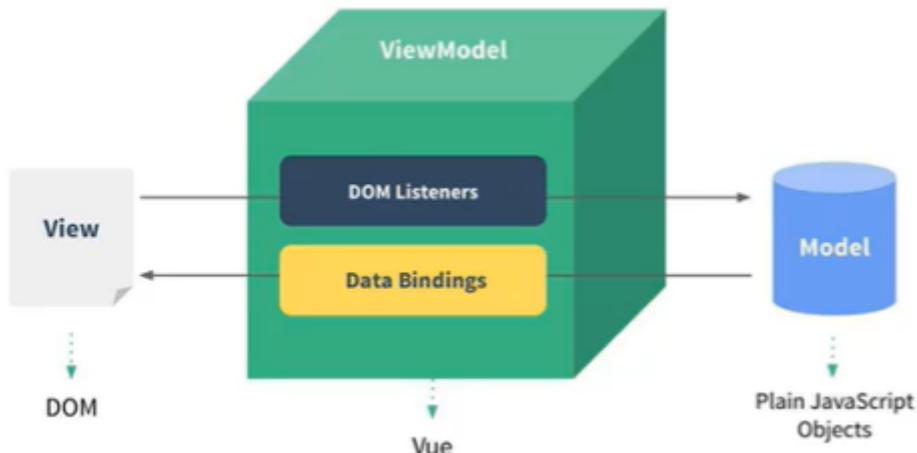
- 视图层
- 在我们前端开发中，通常就是DOM层。
- 主要的作用是给用户展示各种信息。

■ Model层:

- 数据层
- 数据可能是我们固有的死数据，更多的是来自我们服务器，从网络上请求下来的数据。
- 在我们计数器的案例中，就是后面抽取出来的obj，当然，里面的数据可能没有这么简单。

■ ViewModel层:

- 视图模型层
- 视图模型层是View和Model沟通的桥梁。
- 一方面它实现了Data Binding，也就是数据绑定，将Model的改变实时的反应到View中
- 另一方面它实现了DOM Listener，也就是DOM监听，当DOM发生一些事件(点击、滚动、touch等)时，可以监听到，并在需要的情况下改变对应的Data。



上面计数器的例子MVVM解释

小码哥教育 SEE MY GO 计数器的MVVM

■ 计数器的MVVM

- 我们的计数器中就有严格的MVVM思想
 - View依然是我们的DOM
 - Model就是我们抽离出来的obj
 - ViewModel就是我们创建的Vue对象实例

□ 它们之间如何工作呢？

- 首先ViewModel通过Data Binding让obj中的数据实时的在DOM中显示。
- 其次ViewModel通过DOM Listener来监听DOM事件，并且通过methods中的操作，来改变obj中的数据。

- 有了Vue帮助我们完成ViewModel层的任务，在后续的开发，我们就可以专注于数据的处理，以及DOM的编写工作了。

```
</div>

<script src="../js/vue.js"></script>
<script>
    // 语法糖：简写
    // proxy
    const obj = {
        counter: 0,
        message: 'abc'
    }

    const app = new Vue({
        el: '#app',
        data: obj,
        methods: {
            add: function () {
                console.log('add被执行');
                this.counter++
            },
            sub: function () {
                console.log('sub被执行');
                this.counter--
            }
        }
    })

```

这里的obj就相当于model
相当于一个代理，obj里面的数据在methods里面也可以使用

Vue中的options

目前来讲，Vue实例对象中有以下三种属性

el:

- ✓ 类型 : string | HTMLElement
- ✓ 作用 : 决定之后Vue实例会管理哪一个DOM。

data:

- ✓ 类型 : Object | Function (组件当中data必须是一个函数)
- ✓ 作用 : Vue实例对应的数据对象。

methods:

- ✓ 类型 : { [key: string]: Function }
- ✓ 作用 : 定义属于Vue的一些方法，可以在其他地方调用，也可以在指令中使用。

el属性中使用document.querySelector()方法是一样的，此方法相当于el挂载的底层实现

```
// el: '#app' | 1
el: document.querySelector(),
data: obj
```

组件中的data必须是一个函数

Vue的生命周期

一个vue示例对象创建的过程中，， vue会做很多操作，会有很多回调函数

插值操作

mustache语法

插值操作：将data中的文本数据插入到HTML中

可以通过Mustache语法：双大括号语法（胡子语法）

Mustache语法中，不仅仅可以直接写变量，也可以写一些简单的表达式

一个双括号中可以使用多个变量，用+加号加起来，也可以在两个变量之间添加空格''

也可以在双括号中进行简单地算数表达式

```
<div id="app">
  <h2>{{message}}</h2>
  <h2>{{message}}, 李银河!</h2>
  <!--mustache语法中，不仅仅可以直接写变量，也可以写简单的表达式-->
  <h2>{{firstName + lastName}}</h2>
  <h2>{{firstName + ' ' + lastName}}</h2>
  <h2>{{firstName}} {{lastName}}</h2>
  <h2>{{counter * 2}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      firstName: 'kobe',
      lastName: 'bryant',
      counter: 100
    }
  })
</script>
```

你好啊

你好啊, 李银河!

kobebryant

kobe bryant

kobe bryant

200



v-once

v-once用来阻断vue对页面某个数据的响应式的改变

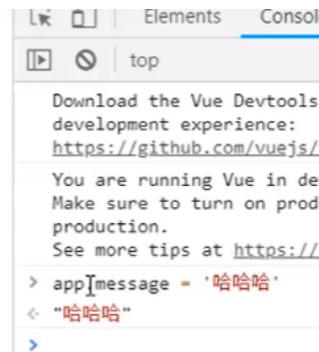
v-once只是一个指令，后面没有赋值

该指令表示元素和组件（组件后面才会学习）**只渲染一次，不会随着data中数据的改变而改变**

```
<div id="app">
  <h2>{{message}}</h2>
  <h2 v-once>{{message}}</h2>
</div>
```

哈哈哈

你好啊



v-html

当data中从服务器中获取的数据是HTML代码，用变量呈现在页面上，想解析这个HTML代码，获取最后的样子，就需要到v-html属性

v-html属性的值就是服务器返回的这个HTML代码变量

```
<div id = "app">
  <h2 v-html="url"></h2>
</div>
<script src=".js/vue.js"></script>
<script>
  const app = new Vue({
    el:'#app',
    data:{
      url:'<a href="http://www.baidu.com">百度一下</a>'
    }
  })
}
```

v-text

想在页面中展示data中的文本时，除了使用双括号语法，还可以使用v-text属性

用法像v-html，变量作为v-text属性的值

```
<h2 v-text="message"></h2>
</div>
```

但是这种写法一般不用，**因为不灵活，比如要做文本的拼接，v-text就做不了**，下图这种写法就直接将v-text中属性的内容覆盖了，不会进行文本拼接

```
<h2 v-text="message">, 李银河!</h2>
```

v-pre

v-pre属性用来将标签中的双括号语法及其内容原封不动的进行展示，而不进行解析，使用的时候直接用v-pre，没有属性值

你好啊

`{{message}}`

```
<div id="app">
  <h2>{{message}}</h2>
  <h2 v-pre>{{message}}</h2>
</div>
```

v-cloak

cloak: 斗篷

如果浏览器在加载页面的时候，vue实例的部分卡住了，页面中的数据会短暂的出现未渲染之前的变量值（就比如出现了双括号语法这种源码），这种情况叫做插值表达式闪烁问题

v-cloak就是用来解决这种情况

将v-cloak属性添加到标签中，当vue对HTML代码进行解析，就会自动删除这个v-cloak属性，在css中可以利用属性中有没有这个属性，来判断标签加载不加载

有v-cloak则不进行展示，没有则进行展示

```
③ <html>
③   <head>
④     <meta charset="utf-8">
④     <title></title>
③   <style>
④     [v-cloak]{
⑤       display: none;
④     }
③   </style>
③ </head>
③ <body>
③   <div id ="app">
④     <h2 v-cloak>{{ message}}</h2>
③   </div>
③   <script src="./js/vue.js"></script>
③   <script>
④     setTimeout(function(){
⑤       const app = new Vue({
⑥         el: '#app',
⑥         data: {
⑦           message: '你哈附件三'
⑧         }
⑨       })
④   
```

设置函数的延迟，强制出

现闪烁，但是使用v-cloak属性，只会出现1s空白，不会出现源码

这里v-cloak用的是属性选择器

但是在以后这个方法也用不上了

绑定属性

v-bind

- 前面我们学习的指令主要作用是将值插入到我们**模板的内容**当中。
- 但是，除了内容需要动态来决定外，某些属性我们也希望动态来绑定。
 - 比如动态绑定a元素的href属性
 - 比如动态绑定img元素的src属性
- 这个时候，我们可以使用v-bind指令：
 - **作用**：动态绑定属性
 - **缩写**：
 - **预期**：any (with argument) | Object (without argument)
 - **参数**：attrOrProp (optional)

mustache语法**不能用于a标签中href属性值的绑定、也不能用于img标签src属性值的绑定**，是没有用的

```
<!-- 错误的做法：这里不可以使用mustache语法-->
<!---->
```

这时使用v-bind属性

语法：v-bind:href/src/....="data中的变量"

使用了v-bind浏览器就知道href和src属性值的内容是一个变量

```

<a v-bind:href="aHref">百度一下</a>
```

像value、title、class这样需要赋值的属性都可以使用

v-bind的语法糖

直接使用**:**来代替**v-bind:**

v-bind动态绑定class

在使用v-bind:class=""时，可以向class传入一个对象，或者数组

对象方式绑定

对象的格式如下

{类名1:布尔值, 类名2:布尔值,.....}

布尔值为true，如果在这个对象内联样式表中没有类名1或者类名2，浏览器渲染页面后直接在该元素的内联样式表class中添加相应的类名

如果**布尔值为false**的话，在该元素的内联样式表class中删除相应类名

一般class传入的对象中的类名都是事先在css中定义好的，方便动态的添加或者删除相应类名，达到删除或者添加某种样式的效果

■ 绑定方式：对象语法

□ 对象语法的含义是：class后面跟的是一个对象。

■ 对象语法有下面这些用法：

用法一：直接通过{}绑定一个类

```
<h2 :class="{ 'active': isActive }">Hello World</h2>
```

用法二：也可以通过判断，传入多个值

```
<h2 :class="{ 'active': isActive, 'line': isLine }">Hello World</h2>
```

用法三：和普通的类同时存在，并不冲突

注：如果isActive和isLine都为true，那么会有title/active/line三个类

```
<h2 class="title" :class="{ 'active': isActive, 'line': isLine }">Hello World</h2>
```

用法四：如果过于复杂，可以放在一个methods或者computed中

注：classes是一个计算属性

```
<h2 class="title" :class="classes">Hello World</h2>
```

例子：

```
<meta charset="utf-8">
<title></title>
<style type="text/css">
    .active{
        color: red;
    }
</style>
</head>
<body>
    <div id ="app">
        <h2 v-bind:class="{active:isActive, line:isLine}">{{message}}</h2>
        <button v-on:click="btnClick">按钮</button>
    </div>
    <script src="./js/vue.js"></script>
    <script>
        const app = new Vue({
            el:'#app',
            data:{
                message:'你好啊',
                isActive:true,
                isLine:true
            },
            methods:{
                btnClick:function(){
                    this.isActive = !this.isActive
                }
            }
        })
    </script>
```

如果class属性值中的对象太长，可以将这种方法改写为把class属性值中的对象放到methods中

将此对象放到一个函数中去，返回值是这个对象就行，class的属性值就改为这个函数名()，记得要有括号，表示函数执行，才能有返回值

ps: 像之前讲的点击事件的响应函数（下面的btnClick函数）就没有写括号，是因为省略了，其实是有

```
<meta charset="utf-8">
<title></title>
<style type="text/css">
  .active{
    color: red;
  }
</style>
</head>
<body>
  <div id ="app">
    <!-- <h2 v-bind:class="{active:isActive, line:isLine}">{{message}}</h2> -->
    <h2 v-bind:class="getClasses()">{{message}}</h2>
    <button v-on:click="btnClick">按钮</button>
  </div>
  <script src="./js/vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data:{
        message:'你好啊',
        isActive:true,
        isLine:true
      },
      methods:{
        btnClick:function(){
          this.isActive = !this.isActive
        },
        getClasses:function(){
          return {active:this.isActive, line:this.isLine}
        }
      }
    })
  </script>
```

数组方式绑定

用的比较少

没有对象方式绑定那种可以动态增减类名的功能

数组里面可以使用带双引号的字符串，表示类名

```
<div id="app">
  <h2 class="title" :class="['active', 'line']">{{message}}</h2>
</div>
```

也可以使用不带双引号的字符串，表示的是data中的变量，这时候就可以从服务器中读取实时的变量

```
<div id="app">
  <h2 class="title" :class="[active, line]">{{message}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      active: 'aaaaaa',
      line: 'bbbbbbb'
    }
  })
</script>

</body>
</html>
```

同样也可以写到**methods**中去

```
<div id="app">
  <h2 class="title" :class="[active, line]">{{message}}</h2>
  <h2 class="title" :class="getClasses()">{{message}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      active: 'aaaaaa',
      line: 'bbbbbbb'
    },
    methods: {
      getClasses: function () {
        return [this.active, this.line]
      }
    }
  })
</script>
```

v-bind绑定style

绑定dass有两种方式: **对象语法**、**数组语法**

■ 绑定方式一：对象语法

```
:style="{color: currentColor, fontSize: fontSize + 'px' }"
```

■ style后面跟的是一个对象类型

➢ 对象的key是CSS属性名称

➢ 对象的value是具体赋的值，值可以来自于data中的属性

■ 绑定方式二：数组语法

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

■ style后面跟的是一个数组类型

➢ 多个值以，分割即可

css的属性名或者class属性的值中如果有-作为连接符，比如：font-size，top-bar...，要么用**驼峰式命名**修改一下，要么用**单引号引起来**

对象方式绑定

跟绑定class相似，可以直接绑定样式名与属性值，作为一个对象传给style，此时样式的**属性值必须要有双引号，样式名要么有单引号，要么不加单引号用驼峰命名法。**

```
<!--<h2 :style="{key(属性名): value(属性值)}">{{message}}</h2>-->
<!-- '50px' 必须加上单引号，否则是当做一个变量去解析-->
<!--<h2 :style="{fontSize: '50px'}"> {{message}}</h2>-->
```

也可以绑定样式名与变量，变量从data中读取，**变量不要加双引号**

语法**{样式名1:变量, 样式名2:变量,.....}**

```
<!--finalSize当成一个变量使用-->
<!--<h2 :style="{fontSize: finalSize}">{{message}}</h2>-->
<h2 :style="{fontSize: finalSize + 'px', color: finalColor}">{{message}}</h2>
</div>
```



```
<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      finalSize: 100,
      finalColor: 'red',
    }
  })
</script>
```

(像font-size这样的有单位的属性值，data中的变量值可以是纯数字，单位放到绑定时再加)

同样，对象太长的话，可以放到methods中的函数中去，作为函数返回值调用

```
<!--<h2 :style="{fontSize: finalSize}">{{message}}</h2>-->
<h2 :style="fontSize + 'px', backgroundColor: finalColor">{{message}}</h2>
<h2 :style="getStyles()">{{message}}</h2>

```

</div>

```

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      finalSize: 100,
      finalColor: 'red',
    },
    methods: {
      getStyles: function () {
        return {fontSize: this.finalSize + 'px', backgroundColor: this.finalColor}
      }
    }
  })

```

数组方式绑定

将每一个样式名:属性值 作为一个对象存到data中，一个样式，一个对象，把这些对象的名字放到数组中去

但这种方法很少用

```
<div id="app">
  <h2 :style="[baseStyle, baseStyle1]">{{message}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      baseStyle: {backgroundColor: 'red'},
      baseStyle1: {fontSize: '100px'},
    }
  })
</script>
```

计算属性的基本使用

什么是计算属性？

- 我们知道，在模板中可以直接通过插值语法显示一些data中的数据。
- 但是在某些情况，我们可能需要对数据进行一些转化后再显示，或者需要将多个数据结合起来进行显示
 - 比如我们有firstName和lastName两个变量，我们需要显示完整的名称。
 - 但是如果多个地方都需要显示完整的名称，我们就需要写多个{{firstName}} {{lastName}}
- 我们可以将上面的代码换成计算属性：

- OK，我们发现计算属性是写在实例的computed选项中的。

```
<div id="app">
  <h2>{{firstName}} {{lastName}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',
    data: {
      firstName: 'Kobe',
      lastName: 'Bryant'
    },
    computed: {
      fullName() {
        return this.firstName + ' ' + this.lastName
      }
    }
  })
</script>
```

在模板中通过差值法在页面中展示一些属性时，有时候需要将几个数值进行结合、组合在进行展示，这种转换的操作就用到了**计算属性**

就比如想要拼接字符串时

可以在HTML中使用的时候进行拼接

```
<div id="app">
  <h2>{{firstName + ' ' + lastName}}</h2>
  <h2>{{firstName}} {{lastName}}</h2>
</div>
```

也可以在methods中写一个方法，到时候在**双括号**中调用这个方法

```
<h2>{{getFullName()}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      firstName: 'Lebron',
      lastName: 'James'
    },
    methods: {
      getFullName() {
        return this.firstName + ' ' + this.lastName
      }
    }
  })
</script>
```

再或者就用**计算属性**来实现

vue对象中还有一个属性，**computed属性**

这个属性中保存着一些**函数对象**，专门用来进行计算，并且返回结果值的

本质是函数，为什么叫计算属性，因为vue是把它当做属性来用的，调用的时候**不能使用括号**，就跟属性一样

(所以命名习惯也跟函数不太一样，一般函数命名时习惯上开头用动词，比如getFullName，而计算属性习惯上就直接用一个名词就行了，比如fullName，因为属性也是名词)

```
<h2>{{getFullName()}}</h2>
<h2>{{fullName}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      firstName: 'Lebron',
      lastName: 'James'
    },
    // computed: 计算属性()
    computed: {
      fullName: function () {
        return this.firstName + ' ' + this.lastName
      }
    },
    methods: {
      getFullName() {
        return this.firstName + ' ' + this.lastName
      }
    }
  })
</script>
```

methods每调用一次就执行一次，而计算属性调用多次，只执行一次，有缓存，效率更高

```
<div id="app">
  <h2>总价格: {{totalPrice}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      books: [
        {id: 110, name: 'Unix编程艺术', price: 119},
        {id: 111, name: '代码大全', price: 105},
        {id: 112, name: '深入理解计算机原理', price: 98},
        {id: 113, name: '现代操作系统', price: 87},
      ]
    },
    computed: {
      totalPrice: function () {
        let result = 0
        for (let i=0; i < this.books.length; i++) {
          result += this.books[i].price
        }
        return result
      }
    }
  })
</script>
```

计算属性详解

setter和getter

承接上面讲的fullName计算属性

```
computed: {
    fullName: function () {
        return this.firstName + ' ' + this.lastName
    }
}
```

上面这种写法其实是计算属性封装简写之后的样子，其内部实现比这复杂

每一个computed的属性都是一个对象，对象当中都会有两个属性：**set和get**，这两个属性里都是函数，（也就相当于两个方法）

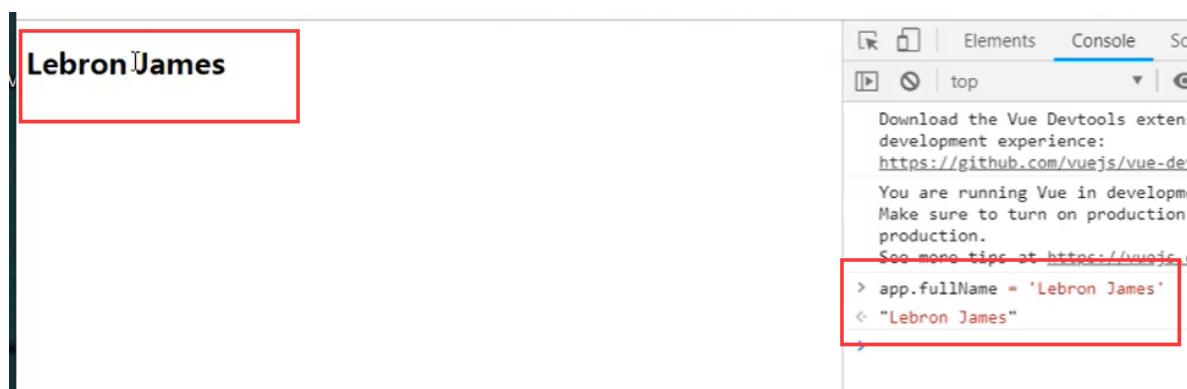
其中**get属性**中函数中保存着我们在computed中自定义的函数，所以返回值是通过get属性里面的函数来实现的

绝大多数set属性中的函数是空着的，不写也行，所以每次都调用get方法，为了简便，就改成了我们常用的样子

```
<script>
data: {
    firstName: 'Kobe',
    lastName: 'Bryant'
},
computed: {
    // fullName: function () {
    //     return this.firstName + ' ' + this.lastName
    // }
    // name: 'coderwhy'
    fullName: {
        set: function () {
            ...
        },
        get: function () {
            return this.firstName + ' ' + this.lastName
        }
    }
}
</script>
```

set方法当中可以接受一个参数，当在控制台为computed中的fullName属性赋值时，就会调用set方法

在set中编写函数，根据参数值修改data中的值，data中的数据就会发生改变，get方法输出值也会发生改变



```
<script>
  const app = new Vue({
    el: '#app',
    data: {
      firstName: 'Kobe',
      lastName: 'Bryant'
    },
    computed: {
      // fullName: function () {
      //   return this.firstName + ' ' + this.lastName
      // }
      // name: 'coderwhy'
      // 计算属性一般是没有set方法，只读属性.
      fullName: {
        set: function(newValue) {
          // console.log('-----', newValue);
          const names = newValue.split(' ');
          this.firstName = names[0];
          this.lastName = names[1];
        },
        get: function () {
          return this.firstName + ' ' + this.lastName
        }
      },
    }
  });

```

The screenshot shows a browser's developer tools console. A red box highlights the assignment of 'Lebron James' to the `fullName` computed property. Another red box highlights the resulting output 'Lebron James' in the browser's DOM.

计算属性与methods对比

methods每调用一次就执行一次，而计算属性如果输入参数一样，浏览器会有computed的缓存，调用多次，只执行一次，效率更高

事件的监听

在前端开发中，我们需要经常和用户交互。

这个时候，我们就必须监听用户发生的时间，比如点击、拖拽、键盘事件等等

在Vue中使用**v-on指令**如何监听事件

■ v-on介绍

口作用：绑定事件监听器

口缩写：@

口预期：Function | Inline Statement | Object

口参数：event

为事件绑定的内容可以是函数、表达式、对象

表达式：

```
<button v-on:click="counter++">+</button>
<button v-on:click="counter--">-</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      counter: 0
    }
  })
</script>
```

函数：

```
<button v-on:click='increment'>+</button>
<button v-on:click='decrement'>-</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      counter: 0
    },
    methods: {
      increment() {
        // ...
      },
      decrement() {
        // ...
      }
    }
  })
</script>
```

语法糖：@

v-on参数的传递

① 在事件监听的时候，**如果不需要传递参数**，事件绑定的方法**可以不写括号**（写与不写没影响。注意，只能是事件监听的时候，如果方法用在别的地方，可能就不能省略括号）

② 如果函数需要参数，但是没有传入，**但写了小括号**，那么函数的形参为 **undefined**

③ 如果函数需要参数，但是没有传入，**也没有写小括号**，这时浏览器会默认将浏览器产生的**event事件对象作为参数传进函数**

④ 在调用方法时，在传入参数的时候如果想传入到浏览器产生的**event事件对象**，使用**\$event**

```
<button @click="btn3Click(abc, $event)">按钮3</button>
<button>按钮4</button>
```

```
  btn3Click(abc, event) {
    console.log('+++++++', abc, event);
  }
}   methods
}
```

v-on修饰符

```
<!-- 停止冒泡 -->
<button @click.stop="doThis"></button>

<!-- 阻止默认行为 -->
<button @click.prevent="doThis"></button>

<!-- 阻止默认行为，没有表达式 -->
<form @submit.prevent></form>

<!-- 串联修饰符 -->
<button @click.stop.prevent="doThis"></button>

<!-- 键修饰符，键别名 -->
<input @keyup.enter="onEnter">

<!-- 键修饰符，键代码 -->
<input @keyup.13="onEnter">

<!-- 点击回调只会触发一次 -->
<button @click.once="doThis"></button>
```

.stop

阻止事件冒泡

在之前的js中使用: **事件.cancelBubble=true**, 来阻止事件冒泡

```
//为s1绑定一个单击响应函数
var s1 = document.getElementById("s1");
s1.onclick = function(event){
    event = event || window.event;
    alert("我是span的单击响应函数");

    //取消冒泡
    //可以将事件对象的cancelBubble设置为true, 即可取消冒泡
    event.cancelBubble = true;
};
```

在vue中使用**.stop**修饰符来阻止冒泡

```
<div id="app">
  <div @click="divClick">
    aaaaaaa
    <button @click.stop="btnClick">按钮</button>
  </div>
</div>
```

.prevent

阻止事件的默认行为

```
<form action="baidu">
  <input type="submit" value="提交" @click.prevent="submitClick">
</form>

methods {
  submitClick() {
    console.log('submitClick');
  }
}
```

.{keyCode|keyAlias}

在绑定的事件后面加上.**{}**, 中括号里面写键盘按键的**Unicode编码**或者**按键缩写**, 表明事件只是从特定按键触发时才回调, 一般绑定在**键盘事件上**

```
<!--3. 监听某个键盘的键帽-->
<input type="text" @keyup.enter="keyUp">
```

.native

监听组件根元素的原生事件，直接用一般的事件绑定对于自定义组件是没有用的，需要加上`.native`

```
<cpn @click.native="cpnClick"></cpn>
```

.once

只触发一次回调

```
<!--4. .once修饰符的使用-->
<button @click.once="btn2Click">按钮2</button>
/div>
```

只在第一次点击时触发回调函数

条件判断

v-if

在元素中添加`v-if`元素，为`v-if`属性赋值为布尔值

`true`，此元素及其子元素都会正常显示

`false`，此元素及其子元素都不会显示

```
<div id="app">
  <h2 v-if="false">
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    {{message}}
  </h2>
</div>
```

布尔值也可以用变量来代替，变量写不写在双引号中都可以

```
<div id="app">
  <h2 v-if="isShow">
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    <div>abc</div>
    {{message}}
  </h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      isShow: true
    }
  })
</script>
```

v-if和v-else结合使用

有v-if和v-else属性的两个标签必须相邻

```
<h2 v-if="isShow">
  <div>abc</div>
  <div>abc</div>
  <div>abc</div>
  <div>abc</div>
  <div>abc</div>
  {{message}}
</h2>
<h1 v-else>isShow为false时，显示我</h1>
</div>
```

v-if和v-else-if和v-else结合使用

```
<div id="app">
  <h2 v-if="score>=90">优秀</h2>
  <h2 v-else-if="score>=80">良好</h2>
  <h2 v-else-if="score>=60">及格</h2>
  <h2 v-else>不及格</h2>
</div>
```

但一般这样写太复杂，阅读性不强，一般还是写在计算方法中

```
        computed: {
          result() {
            let showMessage = '';
            if (this.score >= 90) {
              showMessage = '优秀';
            } else if (this.score >= 80) {
              showMessage = '良好';
            }
            // ...
            return showMessage
          }
        }
      }
    }
  }
}
```

```
<h1>{{result}}</h1>
```

登录案例

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id = "app">
      <span v-if="isUser">
        <label for="username">用户账号</label>
        <input type="text" id="username" placeholder="用户账号">
      </span>

      <span v-else>
        <label for="email">用户邮箱</label>
        <input type="text" id="email" placeholder="用户邮箱">
      </span>
      <button @click="isUser=!isUser">切换类型</button>
    </div>
    <script src="./js/vue.js"></script>
    <script>
      const app = new Vue({
        el:'#app',
        data:{
          isUser:true
        }
      })
    </script>
  </body>
</html>
```

■ 小问题：

- 如果我们在有输入内容的情况下，切换了类型，我们会发现文字依然显示之前的输入的内容。
- 但是按道理讲，我们应该切换到另外一个input元素中了。
- 在另一个input元素中，我们并没有输入内容。
- 为什么会出现这个问题呢？

■ 问题解答：

- 这是因为Vue在进行DOM渲染时，出于性能考虑，会尽可能的复用已经存在的元素，而不是重新创建新的元素。
- 在上面的案例中，Vue内部会发现原来的input元素不再使用，直接作为else中的input来使用了。

■ 解决方案：

- 如果我们不希望Vue出现类似重复利用的问题，可以给对应的input添加key
- 并且我们需要保证key的不同

上面这个问题在原生的js中就不会存在

当将vue应用到某个元素时，vue会自动将**创建虚拟dom**，作用是将创建的这些子元素通过虚拟dom放到**内存**中，然后再将虚拟dom渲染到浏览器上面，这是出于性能的考虑，**会尽可能复用已经存在的元素**

也就是说上面中if和else中的两个table和input标签**都是在用一个**，只不过else使用时将if中的标签进行修改就使用了，**并没有创建新的**。input中的value值也保存下来了

解决方法

可以在input标签中添加**key属性**，两个input标签key属性如果相同，表示可以复用，key不相同表示不可以复用

```
<div id="app">
  <span v-if="isUser">
    <label for="username">用户账号</label>
    <input type="text" id="username" placeholder="用户账号" key="username">
  </span>
  <span v-else>
    <label for="email">用户邮箱</label>
    <input type="text" id="email" placeholder="用户邮箱" key="email">
  </span>
  <button @click="isUser = !isUser">切换类型</button>
</div>
```

v-show

v-show决定一个元素是否进行渲染

和v-if用法非常相似，为v-show属性赋予布尔值或者变量

但是**v-if**的false是将整个元素从HTML标签中删去

v-show是在元素标签中添加一个行内元素display:none, HTML代码中还是有这个元素的

```
<div id="app">
  <h2 v-if="isShow" id="aaa">{{message}}</h2>
  <h2 v-show="isShow" id="bbb">{{message}}</h2>
</div>
```

```
<div id="app">
  <!--v-if: 当条件为false时, 包含v-if指令的元素, 根本就不会存在dom中-->
  <h2 v-if="isShow" id="aaa">{{message}}</h2>

  <!--v-show: 当条件为false时, v-show只是给我们的元素添加一个行内样式: display: none-->
  <h2 v-show="isShow" id="bbb">{{message}}</h2>
</div>
```

当需要在显示与隐藏之间切换**很频繁**, 使用show

当只有**一次切换**时, 使用v-if (用的比较多)

v-for

遍历数组

v-for是元素中的一个属性, 遍历的时候, 这个元素就会进行多次遍历

但是**标签写一次就行**

v-for后面的双引号中格式: **"(数组项,数组下标) in 数组名"**

或者**"数组项 in 数组名"**

后面在data中追加元素的时候, **追加的也可以在列表中渲染出来**

```
<div id="app">
  <!--1. 在遍历的过程中, 没有使用索引值(下标值)-->
  <ul>
    <li v-for="item in names">{{item}}</li>
  </ul>

  <!--2. 在遍历的过程中, 获取索引值-->
  <ul>
    <li v-for="(item, index) in names">
      {{index+1}}.{{item}}
    </li>
  </ul>
</div>
```

name是data中的一个数组数据

遍历对象

①在遍历对象的过程中, 如果只是获取一个值, 那么获取到的是**value**

格式: **value in 对象名**

```
<ul>
  <li v-for="item in info">{{item}}</li>
</ul>
```

② 获取key和value的值, 格式: **(value, key) in 对象名**

value在前面

```
<!--2. 获取key和value 格式: (value, key) -->
<ul>
  <li v-for="(value, key) in info">{{value}}-{{key}}</li>
</ul>
</div>
```

③ 获取key、value、index的值, 格式: **(value, key, index) in 对象名**

```
<!--3. 获取key和value和index 格式: (value, key, index) -->
<ul>
  <li v-for="(value, key, index) in info">{{value}}-{{key}}-{{index}}</li>
</ul>
</div>
```

v-for绑定key和非绑定key的区别

官方推荐我们在使用v-for时, 给对应的元素或组件添加上一个: **key属性**。, key属性内容是每次遍历时的**数组项**

key属性的作用在于, 当要往数组中插入新的内容时 (使用的是**diff的算法**)

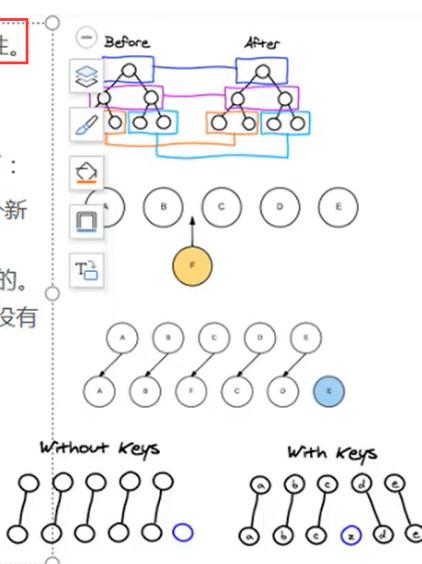
默认情况下是从插入的地方, 将索引和数组项的对象关系往后移一位, 效率很低

加上了key属性时, Diff算法就可以正确的识别此节点, **直接找到正确的位置区插入新的节点。**



组件的key属性

- 官方推荐我们在使用v-for时, 给对应的元素或组件添加上一个**key属性**。
- 为什么需要这个key属性呢 (了解)?
 - 这个其实和Vue的虚拟DOM的**Diff算法**有关系。
 - 这里我们借用React's diff algorithm中的一张图来简单说明一下:
- 当某一层有很多相同的节点时, 也就是列表节点时, 我们希望插入一个新的节点
 - 我们希望可以在B和C之间加一个F, **Diff算法默认执行起来是这样的**。
即把C更新成F, D更新成C, E更新成D, 最后再插入E, 是不是很没有效率?
 - 所以我们需要使用key来给每个节点做一个唯一标识
 - **Diff算法就可以正确的识别此节点**
找到正确的位置区插入新的节点。
- 所以一句话, **key的作用主要是为了高效的更新虚拟DOM**。



```

<div id="app">
  <ul>
    <li v-for="item in letters" :key="item">{{item}}</li>
  </ul>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      letters: ['A', 'B', 'C', 'D', 'E']
    }
  })
</script>

</body>
</html>

```

有key内存里实际就是链表了，所以插入操作效率高了

但是这样的话，数组中的数据如果有重复，key属性的值也会重复，浏览器就会报错

数组中哪些方法是响应式的

像平时我们修改了data中的数据时，页面会自动进行实时的刷新

这是因为数据是响应式的，vue的内部会监听数据的变化，vue会根据新的数据，重新渲染出一个虚拟的dom

虚拟的dom把我们的真实dom进行修改

数组中有的方法和操作是可以将页面进行响应式改变，但有的却不能

比如，**通过索引值修改数组中的元素，此操作并不是响应式的。** data中的数据是被修改了，但是页面中渲染的结果不能被修改，还是之前的值

```

// 注意：通过索引值修改数组中的元素
this.letters[0] = 'bbbbbb';

```

如果想修改数组中的元素，可以使用① **splice()方法**，此方法是响应式的

②或者使用**vue中**有一个方法：**set()**

三个参数：

要修改的对象

索引值

修改后的值

```
Vue.set(this.letters, 0, 'bbbbbb')
```

以下的数组的方法是响应式的：

push()：在数组最后添加一个或多个元素

pop(): 删除数组中的最后一个元素

shift(): 删除数组中的第一个元素

unshift(): 在数组最前面添加一个或多个元素

splice():删除元素、插入元素、替换元素

sort(): 排序

reverse(): 翻转数组

vue中的过滤器

简单介绍一下过滤器，顾名思义，**过滤就是一个数据经过了这个过滤之后出来另一样东西，可以是从中取得你想要的，或者给那个数据添加点什么装饰，那么过滤器则是过滤的工具。**例如，从['abc','abd','ade']数组中取得包含'ab'的值，那么可通过过滤器筛选出来'abc'和'abd'；把'Hello'变成'Hello World'，那么可用过滤器给值'Hello'后面添加上' World'；或者把时间节点改为时间戳等等都可以使用过滤器。

过滤器定义

首先，过滤器可在new Vue实例前注册全局的，也可以在组件上写局部。

全局过滤器：

```
Vue.filter('globalFilter', function (value) {  
  return value + "!!!!"  
})
```

组件过滤器（局部）：

```
filters:{  
    componentFilter:function(value){  
        return value + "!!!!"  
    },  
},
```

上面有种写法有个需要注意的问题：全局注册时是filter，没有s的。而组件过滤器是filters，是有s的，这要注意了，虽然你写的时候没有s不报错，但是过滤器是没有效果的

过滤器调用的时候不用写括号

过滤器的使用方法

用法有二：

一，在双花括号插值

```
{{ 'ok' | globalFilter }}
```

二，在v-bind表达式中使用

上面简单介绍了一下过滤器的调用，那么接下来我们讲解一下过滤器的参数写法

一、{{ message | filterA | filterB }}

上述代码中，message是作为参数传给filterA函数，而filterA函数的返回值作为参数传给filterB函数，最终结果显示是由filterB返回的。

```
<div> {{'2018' | filterA | filterB}} </div>  
  
filters:{  
    filterA:function(value){  
        return value + '年 '  
    },  
    filterB:function(value){  
        return value + 'Hello World!'  
    }  
},
```

2018年 Hello World!

二、{{ message | filterA('arg1', arg2) }}

上述代码中，filterA的第一个参数是message，依次是'arg1',arg2

```
<div>{{ '2018' | filterA('07','17') }}</div>

filters:{
  filterA:function(value,arg1,arg2){
    return value + '-' + arg1 + '-' + arg2
  }
},
```

结果: 2018-07-17

三、 {{ 'a','b' | filterB }}

上述代码表示'a'和'b'分别作为参数传给filterB

```
<div>结果: {{ 'Hello','World' | filterB }}</div>

filters:{
  filterB:function(value,value2){
    return value + ' ' + value2
  }
},
```

结果: Hello World

js中的高阶函数

filter、map、reduce这三个函数，其实可以看成三个方法

为了理解这三个方法的高级性，引入下面这个例子，这个例子可以用这三个函数进行优化

```
// 编程范式：面向对象编程(第一公民：对象)/函数式编程(第一公民：函数)
// filter/map/reduce
const nums = [10, 20, 111, 222, 444, 40, 50]

// 1.需求：取出所有小于100的数字
let newNums = []
for (let n of nums) {
  if (n < 100) {
    newNums.push(n)
  }
}

// 2.需求：将所有小于100的数字进行转化：全部*2
let new2Nums = []
for (let n of newNums) {
  new2Nums.push(n * 2)
}

console.log(new2Nums);

// 3.需求：将所有new2Nums数字相加，得到最终的记过
let total = 0
for (let n of new2Nums) {
  total += n
}
```

filter

filter方法，是**数组**的一个方法（**注意区别上面的过滤器**）

语法：`数组.filter(function(变量){....., return})`

参数：一个回调函数

回调函数的形参是遍历的数组项

回调函数的返回值是布尔值

作用：将数组的各个值进行遍历，并且每次遍历都执行一次**回调函数**，此回调函数作用是进行某种判断，得出一个**布尔值**，**通过函数来返回**，通过返回的布尔值来决定是否将遍历的数组项添加到一个新的数组中去，作为**方法的返回值用一个新的变量来接受**。从而达到**过滤数组的目的**

返回值：一个新的数组

true：当返回true时，函数内部会自动将这次回调的**数组项加入到新的数组中**

false：当返回 false时，函数内部会**过滤掉这次的数组项**

```
const nums = [10, 20, 111, 222, 444, 40, 50]

let newNums = nums.filter(function (n) {
  return n < 100
})
console.log(newNums);
```

map

map方法 (本质上是一种映射)

语法: 变量=数组.map(function(形参){.....return 返回值})

作用: 跟filter方法类似，也会**遍历数组的数组项**，将数组的每一项传入操作后添加到**新数组**

参数: 参数是一个**回调函数**

回调函数的形参是遍历的数组项

回调函数的返回值是将数组项进行某种操作后的值

返回值: 函数的返回值会添加到**新数组**

新数组作为方法的返回值，可以用一个**变量来接收**

```
// 2.map函数的使用
let new2Nums = newNums.map(function (n) {
  return n * 2
})
```

reduce

reduce方法 (相当于一个递归操作) 也是**数组**的一个方法

作用: 对数组中所有数组项进行汇总计算，每次遍历都会使用**上次返回函数的返回值和此次遍历的数组项**，相当于**递归操作** (比如逐项累加...等等一些操作)

语法:

变量=数组.reduce(function(preValue,数组项)

{

.....,return 返回值

},preValue的初始值)

参数: 两个

①preValue: **上一个函数的返回值**，当第一次遍历时，preValue就自定义的值，也就是此方法的第二个参数

②preValue的初始值

返回值: 递归操作后的一个值

```
// 3.reduce函数的使用
// reduce作用对数组中所有的内容进行汇总
let total = new2Nums.reduce(function (preValue, n) {
    return preValue + n
}, 0)
```

至此，上面例子的优化写法如下：

```
let total = nums.filter(function (n) {
    return n < 100
}).map(function (n) {
    return n * 2
}).reduce(function (preValue, n) {
    return preValue + n
}, 0)
```

使用箭头函数进一步优化：

```
let total = nums.filter(n => n < 100).map(n => n * 2).reduce((pre, n) => pre + n);
```

表单绑定

v-model基本使用

v-model实现表单的**双向绑定**，比如在标签、标签等

在表单中使用了v-model属性后，如下图

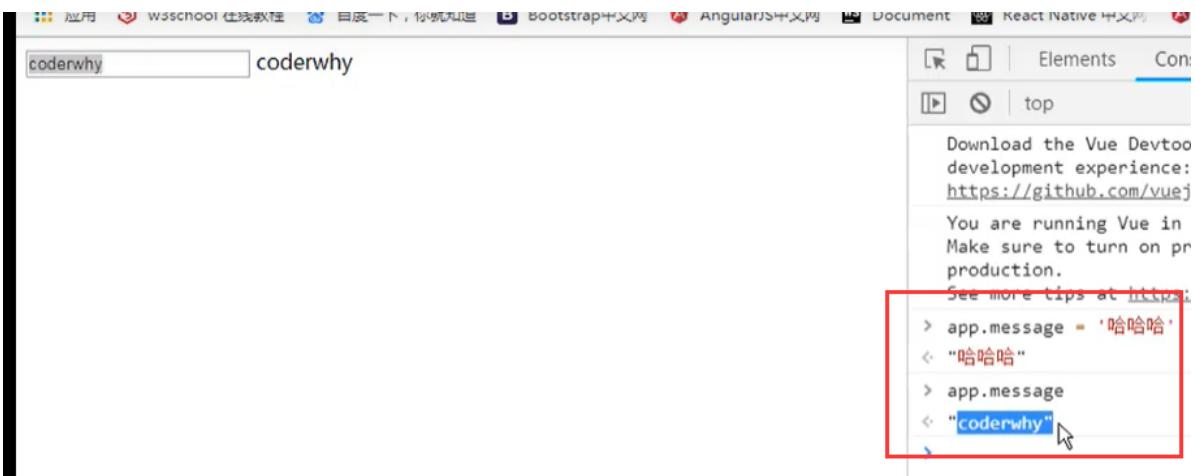
双向绑定体现在：

data中的值可以作为文本框的默认值，**data中的值改变，文本框中的值也会改变**

如果直接在文本框中输入内容（value）也修改data中的数据值，**value中的值改变，会影响到data中的数据改变**

```
<div id="app">
  <input type="text" v-model="message">
  {{message}}
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    }
  })
</script>
```



v-model的原理

v-model具体背后的原理如下：

■ v-model其实是一个语法糖，它的背后本质上是包含两个操作：

- 1.v-bind绑定一个value属性
- 2.v-on指令给当前元素绑定input事件

■ 也就是说下面的代码：等同于下面的代码：

```
<input type="text" v-model="message">
等同于
<input type="text" v-bind:value="message" v-on:input="message = $event.target.value">
```

原理实现例子：

```

<div id="app">
  <!--<input type="text" v-model="message"-->
  <input type="text" :value="message" @input="valueChange">
  <h2>{{message}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    },
    methods: {
      valueChange(event) {
        this.message = event.target.value;
      }
    }
  })
</script>

```

↑
监听事件的回调函数如果不传参数的话，默认传的是事件对象

文本框中的内容通过:value="message"来动态绑定data中的数据

文本框通过设置对input事件的监听，通过监听事件的返回函数来改变data中的数据

其中input事件监听的改进

```

<!--<input type="text" :value="message" @input="valueChange"-->
<input type="text" :value="message" @input="message = $event.target.value">
<h2>{{message}}</h2>

```

v-model结合radio类型使用

radio可选按钮

```

<div id="app">
  <label for="male">
    <input type="radio" id="male" name="sex" value="男" v-model="sex">男
  </label>
  <label for="female">
    <input type="radio" id="female" name="sex" value="女" v-model="sex">女
  </label>
  <h2>您选择的性别是: {{sex}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      sex: ''
    }
  })

```

男 女

您选择的性别是: 男

如果使用v-model绑定的是data中同一个变量

之前用于将几个可选按钮相互互斥分为一组的name属性就可以省略不写

由于v-model属性相同，这几个可选按钮仍是互斥的

```
<div id="app">
  <label for="male">
    <input type="radio" id="male" value="男" v-model="sex">男
  </label>
  <label for="female">
    <input type="radio" id="female" value="女" v-model="sex">女
  </label>
  <h2>您选择的性别是: {{sex}}</h2>
</div>
```

此时定义可选按钮的默认按钮时，之前使用value，现在就直接指定data中绑定的变量值就行就设置了默认按钮

v-model结合checkbox类型使用

checkbox复选框

分两种情况：单个勾选框和多个勾选框

单个勾选框：

```
<div id="app">
  <label for="agree">
    <input type="checkbox" id="agree" v-model="isAgree">同意协议
  </label>
  <h2>您选择的是: {{isAgree}}</h2>
  <button :disabled="!isAgree">下一步</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      isAgree: false
    }
  })
</script>
```

同意协议

您选择的是: false

下一步

单选框时v-model绑定的变量是一个**布尔值**

多个勾选框:

```
<!--2.checkbox多选框-->
<input type="checkbox" value="篮球" v-model="hobbies">篮球
<input type="checkbox" value="足球" v-model="hobbies">足球
<input type="checkbox" value="乒乓球" v-model="hobbies">乒乓球
<input type="checkbox" value="羽毛球" v-model="hobbies">羽毛球
<h2>您的爱好是: {{hobbies}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      isAgree: false,
      hobbies: []
    }
  })
</script>
```

篮球 足球 乒乓球 羽毛球

您的爱好是: ["篮球", "羽毛球", "足球"]

复选框的话, v-model绑定的变量就是一个**数组**, 多选的value会被添加到数组中去

v-model结合select类型的使用

select下拉列表

select标签中还有**option子标签**, 用来表示下拉选项

```
<!-- 下拉列表 -->
<select name="haha">
  <option value="i">选项一</option>
  <option value="ii">选项二</option>
  <option value="iii">选项三</option>
</select>
```



注意: select使用v-model时绑定在**select标签中**

和checkbox一样， select也分单选和多选两种情况。

单选: 只能选中一个值。

v- model:绑定的是一个**值**。

当我们选中 option中的一个时，会将它对应的 value赋值到 data的变量中

```
<div id="app">
    <!--1.选择一个-->
    <select name="abc" id="" v-model="fruit">
        <option value="苹果">苹果</option>
        <option value="香蕉">香蕉</option>
        <option value="榴莲">榴莲</option>
        <option value="葡萄">葡萄</option>
    </select>
    <h2>您选择的水果是: {{fruit}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
    const app = new Vue({
        el: '#app',
        data: {
            message: '你好啊',
            fruit: '香蕉'
        }
    })
</script>
```

多选: 可以选中多个值。

v-mode绑定的是一个**数组**

按住ctrl键，选中多个值时，就会将选中的 option对应的 value添加到data的变量的数组中

```

<!--2.选择多个-->
<select name="abc" v-model="fruits" multiple>
  <option value="苹果">苹果</option>
  <option value="香蕉">香蕉</option>
  <option value="榴莲">榴莲</option>
  <option value="葡萄">葡萄</option>
</select>
<h2>您选择的水果是: {{fruits}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      fruit: '香蕉',
      fruits: []
    }
  })
</script>

```



您选择的水果是["苹果", "香蕉", "榴莲"]

v-model修饰符

.lazy修饰符

默认情况下，**v-model**默认是在input事件中根据输入框的数据**实时改变**data的数据。

lazy修饰符可以让数据在**失去焦点**或者**回车**时才会更新

.number修饰符

默认情况下，v-model双向绑定时，通过v-model修改data中的数据时，传入的数据默认是**字符串**

哪怕使用的是**number输入框**，传入的也是字符串

即使data中的数据初始值是数字，不进行修改确实一直是number值，一旦修改又成了**string类型**

```
<!--2.修饰符: number-->
<input type="number" v-model="age">
<h2>{{age}}-{{typeof age}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      age: 0
    }
  })
</script>
```

01

01-string

此时使用.number修饰符进行类型的指定

```
<!--2.修饰符: number-->
<input type="number" v-model.number="age">
<h2>{{age}}-{{typeof age}}</h2>
```

011221233

11221233-number

.trim修饰符

去除字符串两边的空格

当在文本框中输入的字符串有很多空格时，绑定的data中的数据也会有很多空格

```
<!--3.修饰符: trim-->
<input type="text" v-model="name">
<h2>您输入的名字:{{name}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      age: 0,
      name: ''
    }
  })
</script>
```

The screenshot shows a code editor with a Vue.js application. The template part contains an input field with the binding `v-model="name"`. The script part defines a Vue instance with a data object containing `name`. A tooltip below the input field shows the current value of `name` as "aaaaaa". A text input field at the bottom has the value "aaa" entered.

通过`.trim`来去除字符串两边的空格

```
<!--3.修饰符: trim-->
<input type="text" v-model.trim="name">
<h2>您输入的名字:{{name}}</h2>
</div>
```

vue中watch的详细用法

在vue中，使用watch来响应数据的变化。watch的用法大致有三种。

1. 常用用法

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

```
new Vue({
  el: '#app',
  data: {
    name: '咸鱼'
  },
  watch: {
    name(newVal,oldVal) {
      // ...
    }
  }
})
```

直接写一个监听处理函数，当每次监听到 name 值发生改变时，执行函数。函数有两个参数，**第一个：新的值，第二个：旧的值**，两个参数可选，只写一个代表新的值

也可以在所监听的数据后面直接加字符串形式的方法名：

```
watch: {
  name: 'nameChange'
}
```

2. 立即执行(immediate和handler)

第一种用法watch有一个特点，就是当值**第一次绑定的时候，不会执行监听函数，只有值发生改变才会执行。**

如果我们需要在**最初绑定值的时候也执行函数**，则就需要用到**immediate属性**。

比如当父组件向子组件动态传值时，子组件props首次获取到父组件传来的默认值时，也需要执行函数，此时就需要将immediate设为true。



```
new Vue({
  el: '#app',
  data: {
    name: ''
  },
  watch: {
    name: {
      handler(newVal,oldVal) {
        // ...
      },
      immediate: true
    }
  }
})
```

监听的数据后面写成对象形式，包含**handler方法**和**immediate**，之前我们写的函数其实就是在写这个**handler方法**；

immediate表示在watch中首次绑定的时候，是否执行handler，值为true则表示在watch中声明的时候，就立即执行handler方法，值为false，则和一般使用watch一样，在数据发生变化的时候才执行handler。

3. 深度监听

当需要监听复杂数据类型(对象)的改变时，**普通的watch方法无法监听到对象内部属性的改变**，只有data中的数据才能够监听到变化，此时就需要deep属性对对象进行深度监听。

```
<input type="text" v-model="person.name"/>
```

```
new Vue({
  el: '#app',
  data: {
    person: {id: 1, name: '咸鱼'}
  },
  watch: {
    person: {
      handler(newVal, oldVal) {
        // ...
      },
      deep: true,
      immediate: true
    }
  }
})
```

设置deep: true 则可以监听到person.name的变化，此时会给person的所有属性都加上这个监听器，当对象属性较多时，每个属性值的变化都会执行handler。**如果只需要监听对象中的一个属性值**，则可以做以下优化：**使用字符串的形式监听对象属性**：

```
watch: {
  'person.name': {
    handler(newVal, oldVal) {
      // ...
    },
    deep: true,
    immediate: true
  }
}
```

这样只会给对象的某个特定的属性加监听器。

数组（一维、多维）的变化不需要通过深度监听，对象数组中对象的属性变化则需要deep深度监听。

组件

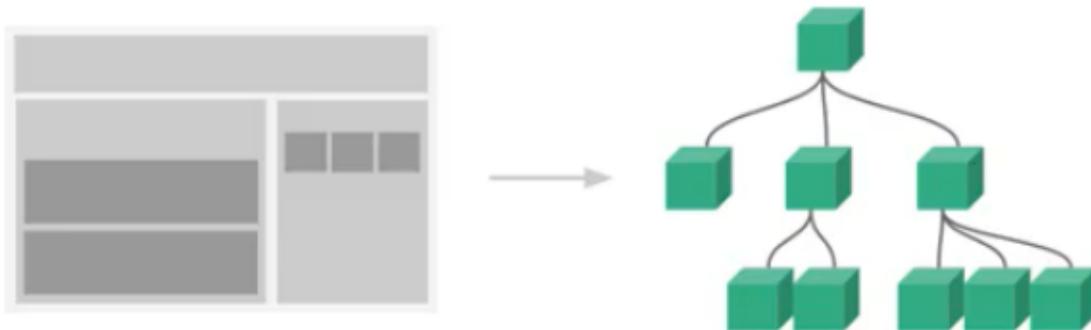
实现和使用步骤

我们将一个完整的页面分成很多个组件。

每个组件都用于实现页面的一个功能块。

而每一个组件又可以进行细分。

□ 任何的应用都会被抽象成一颗组件树。

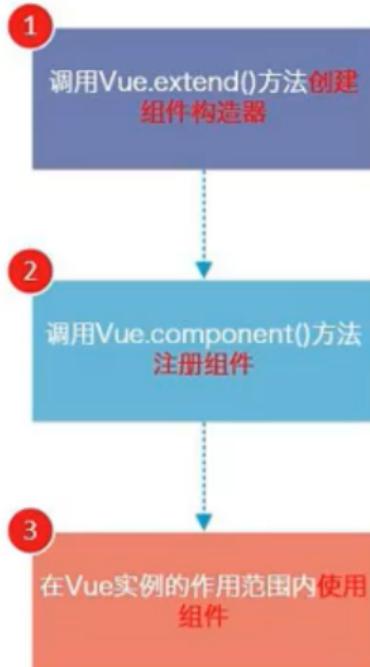


组件的使用分成三个步骤

1. 创建组件构造器

2. 注册组件

3. 使用组件。



```

<div id="app">
  <!--3. 使用组件-->
  <my-cpn></my-cpn>
</div>

<script src="../js/vue.js"></script>
<script>
  // 1. 创建组件构造器
  const myComponent = Vue.extend({
    template: `
      <div>
        <h2>组件标题</h2>
        <p>我是组件中的一个段落内容</p>
      </div>
    `});
  // 2. 注册组件，并且定义组件标签的名称
  Vue.component('my-cpn', myComponent)
</script>

```

知乎 @coderwhy

组件标题

我是组件中的一个段落内容

vue组件化的基本使用

```

<div id="app">
  <!-- 组件的使用 -->
  <my-cpn></my-cpn>
</div>
<script src="./js/vue.js"></script>
<script>
  /* 创建组件构造器对象 */
  const cpnC = Vue.extend({
    /* 模板 */
    template: `
      <div>
        <h2>标题</h2>
        <p>内容</p>
        <p>内容</p>
      </div>
    `})
  /* 注册组件 */
  Vue.component('my-cpn', cpnC)

  const app = new Vue({
    el: '#app',
    data: {
    },
    methods: {
    }
  })
</script>

```

1. Vue.extend():

调用Vue.extend()创建的是一个**组件构造器**。

通常在创建组件构造器时，传入**templated()**代表我们自定义组件的**模板**。

该模板就是在使用到组件的地方，要显示的HTML代码。

事实上，这种写法在**Vue2x的文档中几乎已经看不到了**，会直接使用下面讲到的**语法糖**，但是在很多资料还是会提到这种方式

2. Vue.component():

调用Vue.component()是将刚才的组件构造器**注册为一个组件**，并且给它起一个组件的标签名称

所以需要传递**两个参数**：

1、注册组件的**标签名**

2、**组件构造器**

3. 组件的使用

组件使用时必须在vue实例中使用，通过引入自己自定义的标签名使用

全局组件和局部组件

全局组件

上面在**vue构造函数外面**，通过Vue.component()注册的组件是全局组件

全局组件意味着可以在多个Vue的实例中使用

局部组件

一个网页中可以创建多个vue实例

不同vue构造函数挂载的的**el属性值不同**

当id不同时，就属于不同的vue实例

只能在同一个vue实例（id相同）中的组件叫**局部组件**

在**vue构造函数当中注册组件**

构造器还是在构造函数的外面

通过**components属性**

components属性中可以注册多个组件

属性内容是一个对象，对象的**属性值是自定义的标签名，属性值是构造器**

```
const app = new Vue({
  el: '#app',
  data: {
    message: '你好啊'
  },
  components: {
    // cpn使用组件时的标签名
    cpn: cpnC
  }
})
```

父组件和子组件

```
// 1. 创建第一个组件构造器
const cpnC1 = Vue.extend({
  template:
    <div>
      <h2>我是标题1</h2>
      <p>我是内容， 哈哈哈哈哈</p>
    </div>
})

// 2. 创建第二个组件构造器
const cpnC2 = Vue.extend({
  template:
    <div>
      <h2>我是标题2</h2>
      <p>我是内容， 呵呵呵呵</p>
      <cpn1></cpn1>
    </div>
  components: {
    cpn1: cpnC1
  }
})
```

```
const app = new Vue({  
  el: '#app',  
  data: {  
    message: '你好啊'  
  },  
  components: {  
    cpn2: cpnC2  
  }  
})  
</script>
```

父组件和子组件的构造器没有包含关系，都放在vue构造函数外面就行。

子组件的注册(components属性)放在了**父组件的构造器**里面

父组件的注册放在了**vue构造函数**中

(这样看来，也可以将vue构造函数看成一个特殊的组件构造器，最顶层的组件，是所有组件的父组件，只不过目前template模板没用到)

上面这个例子中的子组件**并不能直接应用到vue实例中去**，如果想用，需要**再次在vue构造函数中注册一遍**

注册组件的语法糖

全局组件的语法糖

将组件的**构造器具体内容**当做参数，替换掉原来**注册时构造器变量的位置**

将**注册**和**构造器**写在一起

这样其实**构造器本质上还是使用的vue的extend方法**，只不过简写了，就不写extend了

```
Vue.component('cpn1', {  
  template:  
    <div>  
      <h2>我是标题1</h2>  
      <p>我是内容，哈哈哈哈</p>  
    </div>  
)
```

局部组件的语法糖

局部组件同样可以将**构造器的具体内容**替换掉原来**构造器代表的变量的位置**

extend方法也可以省略不写

```
const app = new Vue({
  el: '#app',
  data: {
    message: '你好啊'
  },
  components: {
    'cpn2': {
      template: `
        <div>
          <h2>我是标题1</h2>
          <p>我是内容，哈哈哈哈</p>
        </div>
      `
    }
  }
})
```

主要是省去了调用Vue.extend的步骤，而是可以直接使用一个对象来代替。

组件模板抽离的写法

之前在构造、注册组件时，在编写组件的模板时，每次都在js代码中使用字符串编写

这样格式混乱，且书写效率不高

抽离方法：

①通过script标签

可以将组件的模板标签抽离到另一个script标签中，全局、局部组件都可以用

script标签中嵌套HTML代码

script标签的type属性值为：text/x-template

并为script标签指定一个id属性，构造注册组件时，template属性的值为：#id属性值

注意：

组件的模板script标签应该只能包含一个根元素，也就是说作为元素script的直系儿子的元素只能有一个

解决办法是将script的子元素用一个div包裹起来

```
<script type="text/x-template" id="cpn">
<div>
  <h2>我是标题</h2>
  <p>我是内容,哈哈哈</p>
</div>
</script>

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

  // 1.注册一个全局组件
  Vue.component('cpn', {
    template: '#cpn'
  })

```

②使用template标签

直接新建一个template标签，不用写在script标签中

同样绑定一个**id属性**，将构造注册时的**template属性值**等于**#id属性值**

全局、局部组件都可以用

注意：（同script方法）

组件的模板**template标签应该只能包含一个根元素**，也就是说作为元素