第 3 章

Vue.js 基本语法

万丈高楼平地起。本章主要从如下几方面，介绍 Vue.js 的基本语法。

1. Vue.js 对象

介绍如何创建 Vue.js 对象，如何在 Vue.js 对象中定义数据属性和方法，以及 Vue.js 对象的生命周期。

2. 插值表达式

介绍如何在视图上绑定 Vue.js 对象的数据属性，包括简单文本的绑定、html 值的绑定、元素属性值的绑定、样式值的绑定和 JavaScript 表达式的值绑定等。

3. 表单输入绑定

介绍如何给复杂的表单元素绑定动态的数据。

4. 事件处理

介绍如何给视图元素的事件绑定事件监听，事件触发后将执行绑定的函数。

5. 指令

介绍 Vue.js 提供的常用指令，实现页面中事件的绑定、元素属性值的绑定和显示内容的逻辑控制指令等。

6. Vue.js 响应原理

介绍 Vue.js 框架内部的响应原理及对对象、数组的检测响应，还有异步更新问题的处理。

3.1 Vue.js 对象

虽然从严格意义上讲，Vue.js 没有完全遵循 MVVM 模型，但是 Vue.js 的整体设计还是受到了 MVVM 模型的启发，在 Vue.js 中有个非常重要的概念——Vue.js 实例。在后面的章节中，会经常使用 VM(ViewModel 的缩写)表示 Vue.js 实例。

每个 Vue.js 应用都是通过 Vue.js 函数创建一个新的 Vue.js 实例开始的，而且每个 Vue.js 应用都是由通过 Vue.js 函数创建的一个根 Vue.js 实例，以及多个可选的、嵌套的、可复用的组件树组成，如一个 Todo 应用的组件树结构如图 3.1 所示。

创建 Vue.js 实例的语法代码如下：

图 3.1　Todo 应用组件树

```
var vm = new Vue({
  //选项
})
```

注意语法中的 Vue.js,它是定义在 Vue.js 中的函数,所以在使用上面的语法创建 Vue.js 实例之前,需要引入 Vue.js,代码如下:

```
<script type="text/JavaScript" src="../static/js/Vue.js"></script>
```

3.1.1　Vue.js 实例的数据属性

创建 Vue.js 实例的时候,可以在创建的 JSON 对象中,使用可选属性 data 定义 Vue.js 实例的 property 属性,这些 property 属性将会被加入 Vue.js 的响应式系统中。当这些 property 属性的值发生改变的时候,视图将会产生“响应”,即匹配后更新为新值,代码如下:

```
//第 3 章/创建 Vue.js 实例和定义 data 属性.html

//数据对象
var data = { a: 1 }

//该对象被加入一个 Vue.js 实例中
var vm = new Vue({
  data: data
})

//获得这个实例上的 property
//返回源数据中对应的字段
vm.a == data.a //=> true

//设置 property 也会影响原始数据
vm.a = 2
data.a //=> 2
```

```
//反之亦然
data.a = 3
vm.a //=> 3
```

当这些数据改变时,视图会进行重新渲染。值得注意的是,只有当实例被创建时就已经存在于 data 中的 property 才是响应式的。也就是说,如果添加一个新的 property,例如 vm.b='hi',则对 b 的改动将不会触发任何视图的更新。如果开发人员知道在晚些时候需要一个或多个 property 属性,则需要在 data 属性中定义这些 property 属性,给它们赋予空值,后面需要的时候再操作或使用这些 property,代码如下:

```
data: {
 newTodoText: '',
 visitCount: 0,
 hideCompletedTodos: false,
 todos: [],
 error: null
}
```

这里唯一的例外是使用 Object.freeze()函数会阻止修改现有的 property 属性的"响应",也意味着响应系统无法再追踪变化。因为调用了 Object.freeze()函数,改变 property 属性值后,不会渲染到视图上,代码如下:

```
//第 3 章/freeze 方法.html

var data = {
  foo: 'bar'
}
//阻止 property 属性的响应
Object.freeze(data)

const vm = new Vue({
 el: '#app',
 data: data
})
<div id="app">
 <p>{{ foo }}</p>
 <!-- 这里的 `foo` 不会更新! -->
 <button v-on:click="foo = 'baz'">Change it</button>
</div>
```

除了数据 property 属性,Vue.js 实例还暴露了一些有用的 property 属性。开发人员可以在这些 property 属性名前面添加 $前缀,以便与开发人员定义的 property 区分开来,代码如下:

```
//第 3 章/freeze 方法.html
var data = {
  foo: 'bar'
}
var vm = new Vue({
  el: '#app',
  data: data
})

vm.$data === data // => true
vm.$el === document.getElementById("app") // => true
```

关于 Vue.js 实例自己暴露的其他 property 属性，读者可以参考官网（https://cn.vuejs.org/v2/api/）。

3.1.2 Vue.js 实例的方法

开发者可以在创建 Vue.js 实例对象的时候，在传入的 JSON 可选项中自定义若干方法封装业务逻辑，以便于在需要的时候重复调用，代码如下：

```
//第 3 章/自定义 Vue.js 实例方法.html
    <div id = "app">
        {{count}}
        <br/>
        <! -- 给单击事件绑定 increment 方法 -- >
        <input type = "button" v - on:click = "increment" value = "单击递增"></input>
    </div>
    <script type = "text/JavaScript">
         const data = {count:0}
         const vm = new Vue({
             el:'#app',
             data,
             methods:{
                 increment){
                     this.count++
                 }
             }
         })
         //调用两次 vm 实例中的 increment 方法
         vm.increment()
         vm.increment()
    </script>
```

如上面代码所示，开发人员可以在传给 Vue.js 函数的 JSON 对象参数中用 methods 属性定义多个自定义方法，在上面的样例代码中定义的是 increment 方法。定义好后，可以将它们绑定到页面元素的事件上，对应事件触发后自动执行，也可以在 JavaScript 代码中，同调用普通方法的语法一样调用。

同 Vue.js 实例的数据属性一样，开发人员可以自定义多种方法，也可以使用 Vue.js 实例暴露的其他方法，在使用的时候，也是用 $作为前缀，用以区分开发人员的自定义方法。如 vm.$mount 方法可以实现将 vm 对象挂载到指定的 DOM 元素上。在这里对这些方法不进行详细介绍，后面章节会陆续介绍常用的方法。读者可以参考官网(https://cn.vuejs.org/v2/api/)，也可以参考附录 API。

3.1.3 Vue.js 实例生命周期

每个 Vue.js 实例在被创建时都要经过一系列的初始化过程，例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫作生命周期钩子的函数，这给了开发人员在不同阶段添加自己代码的机会。

Vue.js 实例从创建到销毁的整个过程分为创建、绑定、更新和销毁 4 个阶段。对于每个阶段，Vue.js 里面都定义了两个对应的钩子函数，它们分别是 beforeCreate、created、beforeMount、mounted、beforeUpdate、updated、beforeDestroy 和 destroyed。

从调用 Vue.js 的构造函数开始，Vue.js 实例的生命周期的详细说明如下。

(1) 创建 Vue.js 实例。

(2) 初始化实例的事件和生命周期钩子函数。

(3) 调用 beforeCreate 钩子函数。

(4) 初始化 Vue.js 实例的注入属性和方法。

(5) 调用 created 钩子函数。

(6) 判断是否设置 el 选项。如果没有，就不将 vm 对象挂载到 DOM 元素上，否则进入下一步。

(7) 判断是否有 el 对应的模板。如果有对应的模板，就将模板转换成 render 函数，通过 render 函数完成渲染；如果没有对应的模板，就将 el 对应的外层 html 作为模板，进行渲染。

(8) 调用 beforeMount 钩子函数。

(9) 在渲染的视图上，绑定 vm 中定义的数据。

(10) 调用 mounted 钩子函数。

(11) 当 vm 中的数据发生变化时，循环地调用 beforeUpdate 钩子函数，更新页面显示，调用 updated 钩子函数。

(12) 在销毁 Vue.js 实例之前，调用 beforeDestroy 钩子函数。

(13) 清除 Vue.js 实例上的所有监听器子组件等。

(14) 调用 destroyed，完成 Vue.js 实例的销毁。

开发人员可以通过在代码中实现 beforeCreate、created、beforeMount、mounted、beforeUpdate、updated、beforeDestroy 和 destroyed 钩子函数，将合适的业务功能植入 Vue.js 实例的对应生命周期阶段执行。

Vue.js 实例的完整生命周期流程可以参考图 3.2。生命周期钩子方法的定义可以参考的代码如下：

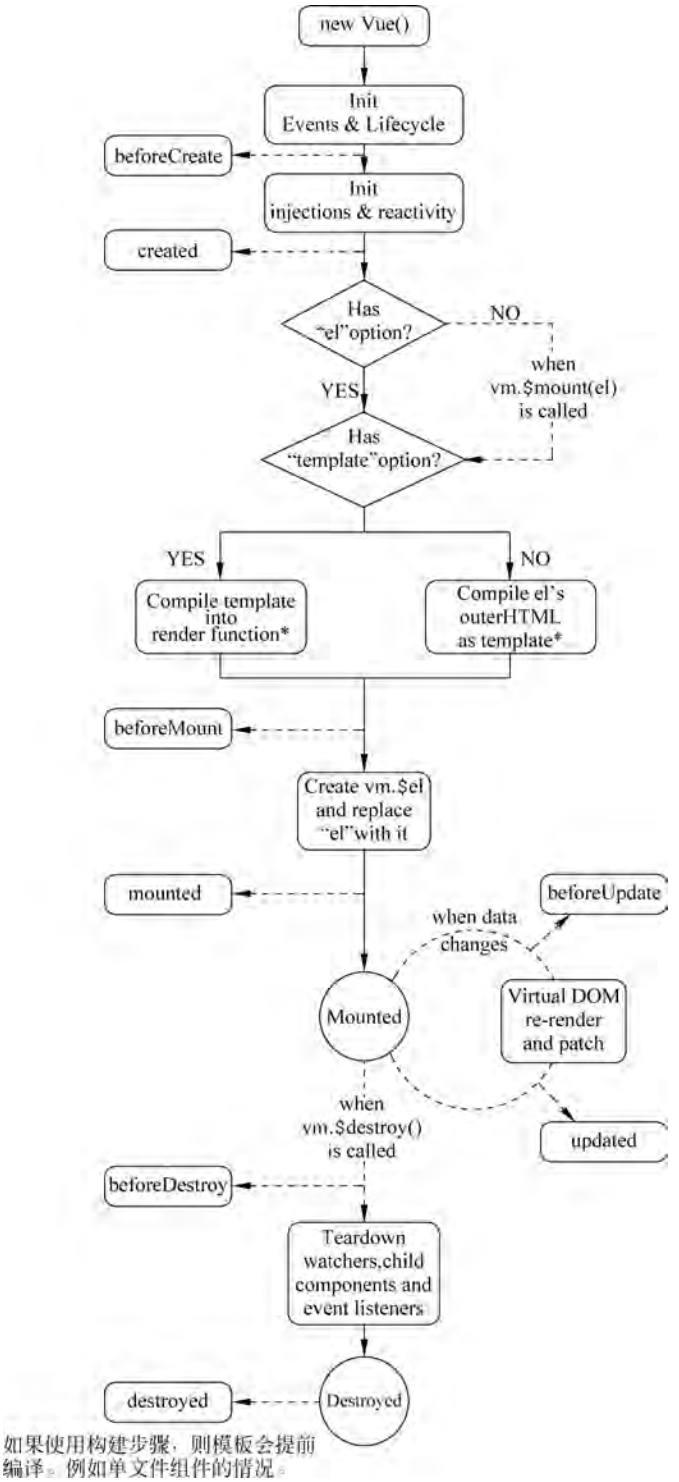

图 3.2　Vue.js 实例生命周期

```
//第 3 章/Vue.js 实例生命周期.html
<!DOCTYPE html>
<html lang="en">
    <head>
        ...
        <script type="text/JavaScript" src="../static/js/Vue.js"></script>
    </head>
    <style type="text/css"></style>
    <body>
        <div id='app'>
            {{message}}
        </div>
        <script type='text/JavaScript'>
            let vm = new Vue({
                el: '#app',
                data: {
                    message: 'old message'
                },
                methods: {
                },
                beforeCreate: function(){
                    console.log('beforeCreate')
                },
                created: function(){
                    console.log('created')
                },
                beforeMount: function(){
                    console.log('beforeMount')
                },
                mounted: function(){
                    console.log('mounted')
                },
                beforeUpdate: function(){
                    console.log('beforeUpdate')
                },
                updated: function(){
                    console.log('updated')
                },
                beforeDestroy: function(){
                    console.log('beforeDestroy')
                },
                destroyed: function(){
                    console.log('destroyed')
                }
            })
            vm.$data.message = 'new message'
```

```
            setTimeout(() =>{
                const appObject = document.getElementById("app";
                document.body.removeChild(appObject)
                vm = null
            }, 2000);

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

3.2 插值表达式

插值表达式的作用是在页面的对应位置插入动态的数据，以便动态地显示。当被插入的数据对象在业务逻辑中发生改变时，会实时地更新到页面上。根据插入动态值的不同位置，分为文本插值、原始 html 插值、JavaScript 插值、class 属性插值和内嵌样式插值 5 种插入表达式，它们的语法和使用时的注意事项如下。

1. 文本插值表达式

文本插值表达式的作用是在标签之间插入动态的文本值。

语法如下：

```
{{表达式}}
```

案例代码如下：

```
<span>hello, {{userName}}</span>
```

说明：在 span 元素中，动态显示 userName 数据对象的值。以文本的形式显示 userName 的值。如果 userName="zhangsan"，则显示“hello，张三”；如果 userName=zhangsan，则显示 hello，zhangsan。

2. 原始 html 插值表达式

原始 html 插值表达式的作用是在标签之间插入原始的 html 内容。如果里面包含 html 标签，则浏览器会进行解析。

语法如下：

```
<标签 v-html='表达式'></标签>
```

案例代码如下：

```
<p>显示 html 内容 <span v-html='userName'></span></p>
```

说明：如果 userName 的值是< font color＝'red'> zhangsan ，则显示的结果为显示红色的“zhangsan”。

3. 属性插值表达式

属性插值表达式的作用是给 html 元素的属性绑定动态值。特别需要注意的是，Mustache 语法(双大括号语法)是不能用在 html 元素属性上的。

语法如下：

```
<标签 v-bind:属性名称 = '表达式'>...<标签>
```

案例代码如下：

```
<div v-bind:id = "divId"> div demo </div>
```

说明：对于布尔属性(只要存在就意味着其值为 true)，v-bind 工作起来略有不同。示例代码如下：

```
<button v-bind:disabled = "isButtonDisabled"> Button </button>
```

如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled 属性不会被包含在渲染出来的< button >元素中，只有当 isButtonDisabled 是 true 或在 JavaScript 中能转换成 true 时，才会被渲染出来。

4. class 属性插值表达式

class 是 html 标签中的一个特殊属性，给 class 属性绑定动态值也采用 v-bind 的方式，同普通属性一样。将 active 数据属性的值绑定到 div 的 class 属性上，代码如下：

```
<div v-bind:class = "active">..</div>
```

对于 class 属性，还支持根据条件绑定数据，将多个类样式和新旧 class 值一起绑定，代码如下：

```
<div v-bind:class = "isActive:active">..</div>
```

其中 isActive 和 active 都是定义在 Vue model 中的属性名称，只有当 isActive 的值(或转换后的值)为 true 时，才将 active 属性的值绑定到 class 属性上，否则不绑定，从而实现根据条件给 class 属性绑定值，代码如下：

```
<div v-bind:class = "{clsName1, clsName2 ...}">..</div>
```

其中 clsName1 和 clsName2 都是 Vue model 的属性名称，Vue.js 的渲染结果是同时给 div 的 class 属性绑定 clsName1 和 clsName2 对应的两个类样式。当然，也可以给每个 clsName 添加条件控制，代码如下：

```
<div v-bind:class="{isClsName1:clsName1, clsName2...}">..</div>
```

最后，div 自己有个 class 属性，其值是 static，同时用 v-bind：class 给 class 属性绑定了 clsName 动态属性值，代码如下：

```
<div class="static" v-bind:class="clsName">..</div>
```

渲染结果是最后的 html 的 class 属性值既包含 static，也包含 clsName 对应的动态值，代码如下：

```
<div class="static clsValue">..</div>
```

5. 内嵌样式插值表达式

v-bind：style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS property 名可以用驼峰式(camelCase)或短横线分隔(kebab-case，记得用引号括起来)来命名，代码如下：

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

data: {
  activeColor: 'red',
  fontSize: 30
}
```

直接绑定到一个样式对象通常更好，这会让模板更清晰，代码如下：

```
<div v-bind:style="styleObject"></div>

data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

v-bind：style 的数组语法可以将多个样式对象应用到同一个元素上，代码如下：

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

6. JavaScript 插值表达式

在页面中除了可以插入一个简单的属性值外，也可以绑定 JavaScript 表达式，代码如下：

```
{{userName + 'demo'}}
{{ok?'yes':'no'}}
{{message.split('').reverse().join('')}}
<div v-html = "'hello' + userName"></div>
```

这些表达式会在所属 Vue.js 实例的数据作用域下作为 JavaScript 被解析，但有个限制，即每个绑定都只能包含单个表达式，所以下面的例子都不会生效，代码如下：

```
<!-- 这是语句,不是表达式 -->
{{ var a = 1 }} `
<!-- 流控制也不会生效,请使用三元表达式 -->
{{ if (ok) { return message } }}
```

注意，模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 Math 和 Date。不能在模板表达式中试图访问用户定义的全局变量，全局变量的白名单如下：

```
const allowedGlobals = makeMap(
   'Infinity,undefined,NaN,isFinite,isNaN,' +
   'parseFloat,parseInt,decodeURI,decodeURIComponent,encodeURI, ' +
   'encodeURIComponent,Math,Number,Date,Array,Object,Boolean,' +
   'String,RegExp,Map,Set,JSON,Intl,require' //for Webpack/Browserify
  )
```

3.3 表单输入绑定

开发人员可以用 v-model 指令在表单元素< input >、< textarea >及< select > 上创建双向数据绑定。v-model 会根据控件类型自动选取正确的方法来更新元素。虽然看起来有些神奇，但 v-model 本质上不过是语法糖(Syntactic Sugar)。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

v-model 会忽略所有表单元素的 value、checked、selected attribute 的初始值而总是将 Vue.js 实例的数据作为数据来源。应该通过 JavaScript 在组件的 data 选项中声明初始值。

v-model 在内部为不同的输入元素使用不同的 property 并抛出不同的事件。

(1) txt 和 textarea 元素使用 value property 和 input 事件。

(2) checkbox 和 radio 元素使用 checked property 和 change 事件。

(3) select 元素将 value 作为 prop 并将 change 作为事件。

下面从 v-model 的基本使用、值绑定和修饰符 3 方面，对使用 v-model 实现表单输入绑定进行介绍。

3.3.1 基本用法

在项目开发过程中，经常希望在表单元素中输入值后，能自动同步到对象属性中。

v-model 很好地解决了这个问题。下面通过案例，分别演示使用 v-model 改变表单元素的值后，如何自动同步到对象属性中。

1. 单行文本输入框

在单行文本输入框 input type＝"text"元素中，用 v-model 绑定了 Vue.js 实例中的 message 数据属性。当用户在单行文本输入框中输入值时，会自动被同步到 message 属性中，从而同步 div 中 message 的显示内容，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
…

        <div id = 'app'>
            <! -- 单行文本输入框 -- >
            < input type = "text" v - model = "message" placeholder = "输入值"/>< br/>

            <div >
                message is: {{message}}
            </div >
        </div >
        < script type = 'text/JavaScript'>
            const vm = new Vue({
                el: '#app',
                data: {
                    message: ''
                },
                methods: {
                }
            })
        </script >
…
```

2. 多行文本输入框

在多行文本输入框 textarea 元素中，用 v-model 绑定了 Vue.js 实例中的 message 数据属性。当用户在多行文本输入框中输入值时，会自动被同步到 message 属性中，从而同步 div 中 message 的显示内容，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
…

        <div id = 'app'>
            <!—多行文本输入框 -- >
            < textarea v - model = "message" placeholder = "输入值"></textarea >< br/>

            <div >
                message is: {{message}}
```

```
            </div>
        </div>
        <script type='text/JavaScript'>
            const vm = new Vue({
                el: '#app',
                data: {
                    message: ''
                },
                methods: {
                }
            })
        </script>
...
```

使用 textarea 元素时需要注意，在文本区域插值(<textarea>{{text}}</textarea>)不会生效，需要用 v-model 来代替。

3. 复选框

复选框绑定的对象属性值可以是 boolean 类型的值，也可以是其他值。绑定的对象属性是 boolean 类型的值，当 checkbox 被选中时，会被同步到 checked 数据属性，这个属性是 boolean 类型的，同时会在 label 部分显示 true，否则显示 false，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
...
            <!-- 复选框绑定 boolean -->
            <input type="checkbox" v-model='checked' id="checkbox"/>
            <label for="checkbox">{{checked}}</label>
        </div>
        <script type='text/JavaScript'>
            const vm = new Vue({
                el: '#app',
                data: {
                    checked: false
                },
                methods: {
                }
            })
        </script>
...
```

选中的那些 checkbox 会自动被同步到 checkedNames 数据属性，这个属性是字符串数组，同时会在 span 中显示所有被选中的 checkbox 的值，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
…
<div id = 'app'>
  <! -- 复选框绑定数组 -- >
  <input type = "checkbox" id = "jack" value = "Jack" v - model = "checkedNames">
  <label for = "jack"> Jack </label >
  <input type = "checkbox" id = "john" value = "John" v - model = "checkedNames">
  <label for = "john"> John </label >
  <input type = "checkbox" id = "mike" value = "Mike" v - model = "checkedNames">
  <label for = "mike"> Mike </label ><br >
  <span > Checked names: {{ checkedNames }}</span >
</div >
<script type = 'text/JavaScript'>
  const vm = new Vue({
    el: '#app',
    data: {
      checkedNames: []
    },
    methods: {}
  })
</script >
…
```

4. 单选按钮

选中某个单选按钮后,会自动被同步到 picked 数据属性,在 span 元素中同步显示,代码如下:

```
//第 3 章/表单输入绑定之基本用法.html
…
<div id = 'app'>
   <! -- 单选按钮 -- >

   <input type = "radio" id = "one" value = "One" v - model = "picked">
   <label for = "one"> One </label >
   <input type = "radio" id = "two" value = "Two" v - model = "picked">
   <label for = "two"> Two </label >
   <br >
   <span > Picked: {{ picked }}</span >
</div >
<script type = 'text/JavaScript'>
   const vm = new Vue({
      el: '#app',
      data: {
         picked: ''
      },
```

```
        methods: {}
    })
</script>
…
```

5. 选择框

当选中选择框的某个选项时，会自动将选项的值同步到 selected 数据属性中，同时会被显示到 span 元素中，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
…
        <div id = 'app'>
            <! -- 选择框 -- >
            <select v - model = "selected">
                <option disabled value = "">请选择</option>
                <option > A </option >
                <option > B </option >
                <option > C </option >
              </select >
              <span > Selected: {{ selected }}</span >
        </div >
        <script type = 'text/JavaScript'>
            const vm = new Vue({
                el: '#app',
                data: {
                    selected: ''
                },
                methods: {
                }
            })
        </script >
…
```

如果 v-model 表达式的初始值未能匹配任何选项，则< select >元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第 1 个选项。因为在这样的情况下，iOS 不会触发 change 事件，因此，更推荐像上面那样提供一个值为空的禁用选项。

v-model 同时还支持多个选项的选择框，这时候可以绑定一个数组类型的数据属性。当选中多个选项时，会被自动同步到 selectedArray 数据属性中，同时对应的 span 元素会显示被选中选项的值，代码如下：

```
//第 3 章/表单输入绑定之基本用法.html
…
<div id = 'app'>
   <! -- 选择框 -- >
```

```
    <select v-model="selectedArray" multiple style="width: 50px;">
      <option>A</option>
      <option>B</option>
      <option>C</option>
    </select>
    <br>
    <span>selectedArray: {{ selectedArray }}</span>
</div>
<script type='text/JavaScript'>
    const vm = new Vue({
      el: '#app',
      data: {
        selectedArray: []
      },
      methods: {}
    })
</script>
...
```

3.3.2 值绑定

对于单选按钮、复选框及选择框的选项，v-model 绑定的值通常是静态字符串（对于复选框也可以是布尔值），代码如下：

```
<!-- 当选中时，`picked` 为字符串 "a" -->
<input type="radio" v-model="picked" value="a">

<!-- `toggle` 为 true 或 false -->
<input type="checkbox" v-model="toggle">

<!-- 当选中第 1 个选项时，`selected` 为字符串 "abc" -->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

但是有时开发人员可能想把值绑定到 Vue.js 实例的一个动态 property 上，这时可以用 v-bind 实现，并且这个 property 的值可以不是字符串。

1. 复选框

在 checkbox 元素中，添加 true-value 和 false-value 属性，分别表示选中和没选中绑定的值。当选中的时候，v-model 绑定的 toggle 数据属性的值是 yes，否则是 no，代码如下：

```
<input type="checkbox" v-model="toggle" true-value="yes" false-value="no" />

//当选中时
```

```
vm.toggle === 'yes'
//当没有选中时
vm.toggle === 'no'
```

这里的 true-value 和 false-value 属性并不会影响输入控件的 value 属性，因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交(yes 或 no)，则应换用单选按钮。

2. 单选框

当选中 radio 时，pick 数据属性的值就是 v-bind 绑定的值 a，代码如下：

```
<input type="radio" v-model="pick" v-bind:value="a">

//当选中时
vm.pick === vm.a
```

3. 选择框

当选中选项时，通过 v-model 绑定的 selected 数据属性的值会被自动同步为选项中用 v-bind 绑定的对象，代码如下：

```
<select v-model="selected">
    <!-- 内联对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
//当选中时
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

3.3.3 修饰符

在使用 v-model 绑定 Vue.js 实例的数据属性时，还可以添加相关的修饰符，在将数据同步到 Vue.js 实例的数据属性的时候，对数据进行必要预处理。例如转换成数字类型、去掉空格等。

1. lazy 修饰符

在默认情况下，v-model 在每次 input 事件触发后会将输入框的值与数据进行同步(除了上述输入法组合文字时)。开发人员可以添加 lazy 修饰符，从而转换为在 change 事件之后进行同步，代码如下：

```
<!-- 在"change"时而非"input"时更新 -->
<input v-model.lazy="msg">
```

2. number 修饰符

如果想自动将用户的输入值转换为数值类型，则可以给 v-model 添加 number 修饰符，

代码如下：

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

这种方法通常很有用，因为即使在 type="number" 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 parseFloat()函数解析，则会返回原始的值。

3. trim 修饰符

如果要自动过滤用户输入的首尾空白字符，则可以给 v-model 添加 trim 修饰符，代码如下：

```
<input v-model.trim="msg">
```

3.4 事件处理

在开发前端页面的时候，经常需要给 html 元素绑定事件代码，当对应的事件被触发时，能自动执行指定的业务代码。本小节将介绍 Vue.js 中对事件的处理。

3.4.1 监听事件

在 Vue.js 中，可以使用 v-on 指令监听 DOM 事件，并在触发时运行 JavaScript 代码。使用 v-on 给 button 元素绑定一段 JavaScript 代码，每单击一次按钮，就给 counter 数据属性加 1，代码如下：

```
//第3章/事件监听.html
…
<div id='app'>
   <button v-on:click="counter += 1">Add 1</button>
   <p>按钮被单击了 {{ counter }} 次.</p>
</div>
<script type='text/JavaScript'>
   const vm = new Vue({
      el: '#app',
      data: {
         counter: 0
      }
   })
</script>
…
```

3.4.2 事件处理方法

在实际项目中，许多事件处理逻辑很复杂，所以直接把 JavaScript 代码写在 v-on 指令

中是不可行的，因此 v-on 应可以接收一个需要调用的方法名称。使用 v-on 给 button 元素绑定一个定义在 Vue.js 实例中的方法，当单击按钮的时候，自动执行 Vue.js 实例中绑定的方法，代码如下：

```
//第 3 章/事件处理方法.html
…
<div id = 'app'>
   <! -- `greet` 是在下面定义的方法名 -- >
   <button v - on:click = "greet"> Greet </button >
</div >
<script type = 'text/JavaScript'>
   const vm = new Vue({
      el: '#app',
      data: {
         name: 'Vue.js'
      },
      methods: {
         greet: function (event) {
            //`this` 在方法里指向当前 Vue.js 实例
            alert('Hello ' + this.name + '!')
            //`event` 是原生 DOM 事件
            if (event) {
                    alert(event.target.tagName)
                }
            }
       }
   })

//也可以用 JavaScript 直接调用方法
   vm.greet() // = > 'Hello Vue.js!'
</script >
…
```

3.4.3 内联处理器中的方法

除了可以使用 v-on 给元素绑定一种方法外，也可以在内联 JavaScript 中调用方法。在 Say hi 和 Say what 两个按钮的内联 JavaScript 中调用 say 方法，代码如下：

```
//第 3 章/内联处理器中调用方法.html
…
<div id = 'app'>
   <button v - on:click = "say('hi')"> Say hi </button >
   <button v - on:click = "say('what')"> Say what </button >
</div >
<script type = 'text/JavaScript'>
```

```
    const vm = new Vue({
        el: '#app',
        methods: {
            say: function(msg){
                alert(msg)
            }
        }
    })
</script>
...
```

有时也需要在内联语句处理器中访问原始的 DOM 事件，可以用特殊变量 $event 把它传入方法。使用 $event 把事件对象传入方法中，代码如下：

```
//第 3 章/内联处理器中调用方法.html
...
        <div id = 'app'>
            <button v-on:click = "warn('表单不能被提交')"> submit </button>
        </div>
        <script type = 'text/JavaScript'>
            const vm = new Vue({
                el: '#app',
                methods: {
                    warn: function (message, event) {
                        //现在可以访问原生事件对象了
                        if (event) {
                            //阻止事件提交
                            event.preventDefault()
                        }
                        alert(message)
                    }
                }
            })
        </script>
...
```

3.4.4 事件修饰符

在事件处理程序中调用 event.preventDefault()或 event.stopPropagation()方法是非常常见的需求。尽管开发人员可以在方法中轻松实现这一点，但更好的方式是方法只有纯粹的数据逻辑，而不是去处理 DOM 事件的细节。

为了解决这个问题，Vue.js 为 v-on 提供了事件修饰符。同前文提过的 v-model 修饰符一样，是由点开头的指令后缀来表示的。

(1).stop：阻止单击事件继续传播。

(2).prevent：提交事件不再重载页面。

(3).capture：使用事件捕获模式添加事件。也就是说，如果内联元素中有其他事件，则先触发.capture修饰的事件。

(4).self：只触发发生在当前元素身上的事件，也就是说，事件不会由内部触发。

(5).once：只触发一次。

(6).passive：执行默认方法。浏览器只有等内核线程执行到事件监听器对应的JavaScript代码时，才能知道内部是否会调用preventDefault()方法来阻止事件的默认行为，所以浏览器本身是没有办法对这种场景进行优化的。这种场景下，用户的手势事件无法快速产生，会导致页面无法快速执行滑动逻辑，从而让用户感觉到页面卡顿。通俗地说就是每次事件产生时，浏览器都会去查询是否应由preventDefault阻止该次事件的默认动作。加上passive就是为了明确地告诉浏览器，不用查询了，这里没用preventDefault阻止默认动作。.passive修饰符一般用于滚动监听，如@scroll和@touchmove。因为在滚动监听过程中，移动每像素都会产生一次事件，每次都使用内核线程查询prevent会使滑动卡顿，通过添加passive将内核线程查询跳过，可以大大提升滑动的流畅度。

各种事件修饰符的使用和说明如下：

```
<!-- 阻止单击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联 -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>

<!-- 添加事件监听器时使用事件捕获模式 -->
<!-- 即内部元素触发的事件先在此处理,然后才交由内部元素进行处理 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只在 event.target 是当前元素时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
<div v-on:click.self="doThat">...</div>

<!-- 单击事件将只会触发一次 -->
<a v-on:click.once="doThis"></a>

<!-- 滚动事件的默认行为 (滚动行为) 将会立即触发 -->
<!-- 而不会等待 `onScroll` 完成 -->
<!-- 这其中包含 `event.preventDefault()` 的情况 -->
<div v-on:scroll.passive="onScroll">...</div>
```

使用事件修饰符时的注意事项如下。

(1) .passive 和.prevent 冲突,不能同时绑定在同一个监听器上。

(2) 使用修饰符时,顺序很重要,相应的代码会以同样的顺序产生。

用 v-on: click.prevent.self 会阻止所有的单击,而 v-on: click.self.prevent 只会阻止对元素自身的单击。

(3) .once 修饰符还能被用到自定义的组件事件上。

3.4.5 按键修饰符

在监听键盘事件时,经常需要检查详细的按键。Vue.js 允许为 v-on 在监听键盘事件时添加按键修饰符,对按键进行详细检查后再执行绑定的方法。当所在的键是 Enter 键的时候,调用 vm.submit()方法,代码如下:

```
<!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->
<input v-on:keyup.enter="submit">
```

Vue.js 提供了绝大多数常用的按键的别名,它们是:

(1) .enter。

(2) .tab。

(3) .delete(捕获“删除”和“退格”键)。

(4) .esc。

(5) .space。

(6) .up。

(7) .down。

(8) .left。

(9) .right。

除了这些 Vue.js 提供的按键别名外,Vue.js 还提供了一个机制,允许开发人员通过全局 config.keyCodes 对象自定义按键修饰符别名。通过 config.keyCodes 对象,定义 v、f1、"media-play-pause"和 up4 个按键别名,代码如下:

```
Vue.config.keyCodes = {
  v: 86,
  f1: 112,
  //camelCase 不可用
  mediaPlayPause: 179,
  //取而代之的是 kebab-case 且用双引号括起来
  "media-play-pause": 179,
  up: [38, 87]
}
```

自定义按键别名的代码如下:

```
<input type="text" v-on:keyup.media-play-pause|v|f1|up="method">
```

3.4.6 系统修饰符

除了前面的事件、按键修饰符外，Vue.js还提供了系统修饰符，实现按特殊键配合鼠标或键盘事件才能触发事件。比较常用的键是Ctrl、Alt、Shift和Meta键，它们对应的修饰符是.ctrl、.alt、.shift和.meta。注意：在Mac系统的键盘上，Meta键对应command键(⌘)。在Windows系统的键盘上Meta键对应Windows徽标键(⊞)。在Sun操作系统的键盘上，Meta键对应实心宝石键(◆)。它们同其他键盘和鼠标的联合使用代码如下：

```
<!-- Alt + C -->
<input v-on:keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div v-on:click.Ctrl="doSomething">Do something</div>
```

需要注意的是，修饰键与常规按键不同，在和keyup事件一起使用时，当事件触发时修饰键必须处于按下状态。换句话说，只有在按住Ctrl键的情况下释放其他按键，才能触发keyup.ctrl，而单单释放Ctrl键不会触发事件。如果想要这样的行为，应将Ctrl键换作keyCode：keyup.17。

Vue 2.5以后，新增了.exact修饰符，能精准地控制系统修饰符组合事件，代码如下：

```
<!-- 即使 Alt 或 Shift 键被一同按下时也会触发 -->
<button v-on:click.Ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 键被按下的时候才触发 -->
<button v-on:click.Ctrl.exact="onCtrlClick">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->
<button v-on:click.exact="onClick">A</button>
```

最后，Vue.js还提供了3个仅响应特定的鼠标按钮的修饰符，它们分别是.left、.right和.middle，代码如下：

```
<span v-on:mousedown.left='mouseLeft'>鼠标左击</span>
<span v-on:mousedown.right='mouseRight'>鼠标右击</span>
<span v-on:mousedown.middle='mouseMiddle'>鼠标中键</span>
```

3.5 指令

指令(Directives)是带有v-前缀的特殊attribute。指令的职责是，当表达式的值改变时，将其产生的连带影响响应式地作用于DOM。熟悉使用指令，能方便开发人员快速地实

现页面逻辑。

3.5.1 v-text 和 v-html 指令

v-text 和 v-html 的作用是在元素之间插入动态值，不同的是，v-html 会识别动态内容中的网页信息，而 v-text 是将网页信息当纯文本处理。

语法如下：

```
<标签 v-text/v-html="表达式"></标签>
```

案例代码如下：

```
<div id="app">
    <p v-text="htmlName"></p>
    <p v-html="htmlName"></p>
</div>
<script>
    const vm = new Vue({
        el:"#app",
        data:{
            htmlName:'<font color="red">张三</font>'
        }
    })

</script>
```

3.5.2 v-bind 指令

v-bind 指令的作用是动态地绑定一个或多个 attribute，或将一个组件 prop 绑定到表达式。

在绑定 class 或 style attribute 时，支持其他类型的值，如数组或对象。

在绑定 prop 时，prop 必须在子组件中声明。可以用修饰符指定不同的绑定类型。

当没有参数时，可以绑定到一个包含键值对的对象。注意此时 class 和 style 绑定不支持数组和对象。

语法如下：

```
<标签 v-bind:属性='表达式'>...</标签>
```

案例代码如下：

```
<!-- 绑定一个 attribute -->
<img v-bind:src="imageSrc">

<!-- 动态 attribute 名 (2.6.0+) -->
```

```
<button v-bind:[key]="value"></button>

<!-- 缩写 -->
<img :src="imageSrc">

<!-- 动态 attribute 名缩写 (2.6.0+) -->
<button :[key]="value"></button>

<!-- 内联字符串拼接 -->
<img :src="'/path/to/images/' + fileName">

<!-- class 绑定 -->
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>
<div :class="[classA, { classB: isB, classC: isC }]">

<!-- style 绑定 -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- 绑定一个全是 attribute 的对象 -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- 通过 prop 修饰符绑定 DOM attribute -->
<div v-bind:text-content.prop="text"></div>

<!-- prop 绑定."prop"必须在 my-component 中声明. -->
<my-component :prop="someThing"></my-component>

<!-- 通过 $props 将父组件的 props 一起传给子组件 -->
<child-component v-bind="$props"></child-component>

<!-- XLink -->
<svg><a :xlink:special="foo"></a></svg>


<div id="app">
    <div v-bind:id="divId"> div demo </div>
    <div v-bind:disabled="isdisabled"> disabled </div>
</div>
<script>
    const vm = new Vue({
            el:"#app",
            data:{
                    divId:'demoDiv',
                    isdisabled:true
```

```
        }
    })

    console.log(document.getElementById('demoDiv'))
</script>
```

说明：如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled attribute 甚至不会被包含在渲染出来的 <button> 元素中。

v-bind 可以缩写成冒号，代码如下：

```
<标签 :属性 = '表达式'>...</标签>
```

3.5.3 v-once 指令

v-once 指令的作用是控制插值只会在 UI 上渲染一次，当再改变属性的值时此值不会同步到 UI 上。

语法如下：

```
<标签 v-once ...>...</标签>
```

案例代码如下：

```
<div id = "app">
    <p v-once>{{userName}}</p>
    <p>{{userName}}</p>
    <p v-once v-bind:id = "'id' + age"> age{{age}}</p>
    <p v-bind:id = "'id' + age"> age{{age}}</p>
</div>
<script>
    const vm = new Vue({
        el:"#app",
        data:{
            userName:'zhangsan',
            age:12
        }
    })
    //修改值后，不再同步到 UI
    vm.$data.age = 14;
    vm.$data.userName = 'lisi';
</script>
```

3.5.4 v-model 指令

v-text 和 v-html 只能实现单向绑定：对象属性能同步到 UI，但是在 UI 上改变值后，不

会同步到对象属性。v-model 指令可以实现双向绑定，经常用在表单元素上，也可以用在 Vue.js 的自定义 Component 上。

语法如下：

```
<表单元素|VueComponent v-mode="表达式" ...>...</表单元素>
```

案例代码如下：

```
<div id="app">
    <input v-model="message" type="text" name="msg" placeholder="请输入消息" /><br/>
    <textarea v-model="message"></textarea><br/>
    <input v-model="edu" type="text" placeholder="输入学历:1,2,3">
    <select v-model="edu">
        <option value="1">大专</option>
        <option value="2">本科</option>
        <option value="3">博士</option>
    </select><br/>
    <input v-model="sex" placeholder="输入性别:0,1" />
    <input v-model="sex" type="radio" value="0" name="sex"/>男
    <input v-model="sex" type="radio" value="1" name="sex"/>女<br/>
    <input v-model="likes" placeholder="输入值" />
    <input v-model="likes" type="checkbox" value="0" name="likes0"/>阅读<br/>
</div>
<script>
    const vm = new Vue({
        el:"#app",
        data:{
            message:'demo',
            edu:2,
            sex:1,
            likes:false
        }
    })
</script>
```

3.5.5 v-if、v-else-if 和 v-else 指令

v-if、v-else-if 和 v-else 指令的作用是条件性地渲染一部分内容。

语法如下：

```
<div v-if="type === 'A'">
  A
```

```
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

案例代码如下：

```
<div id="app">
    <div v-if="type > 0.8">
     A
    </div>
    <div v-else-if="type > 0.6">
     B
    </div>
    <div v-else-if="type > 0.4">
     C
    </div>
    <div v-else>
     Not A/B/C
    </div>
    <button v-on:click="changeType">changeType</button>
</div>
<script>
    const vm = new Vue({
        el:"#app",
        data:{
            type:Math.random()
        },
        methods:{
                    changeType:function(){
                        this.type = Math.random;
            }
        }
    })
</script>
```

说明：在分支语句和 v-for 语句中，可能会存在相同元素需要渲染。Vue.js 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做除了使 Vue.js 运行时变得非常快之外，还有其他一些好处。例如，应用允许用户在不同的登录方式之间切换，代码如下：

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username">
</template>
<template v-else>
  <label>E-mail</label>
  <input placeholder="Enter your email address">
</template>
```

在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，<input>不会被替换掉——仅仅替换了它的 placeholder。这样不总是符合实际需求，所以 Vue.js 为开发人员提供了一种方式来表达“这两个元素是完全独立的，不要复用它们”。只需添加一个具有唯一值的 key attribute，代码如下：

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>E-mail</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

每次切换时，输入框都将被重新渲染。

3.5.6 v-show 指令

v-show 指令的作用是根据条件展示元素。带有 v-show 的元素始终会被渲染并保留在 DOM 中。v-show 只是简单地切换元素的 CSS property display。

语法如下：

```
<标签 v-show="表达式">... </标签>
```

案例代码如下：

```
<div id="app">
    <div v-show="ok">ok</div>
    <button v-on:click="toChange">Change</button>
</div>
<script>
    const vm = new Vue({
        el:"#app",
        data:{
            ok:true
```

```
                },
                methods:{
                    toChange:function(){
                        this.ok = !this.ok;
                    }
                }
            })
</script>
```

注意：v-show 不支持< template >元素，也不支持 v-else。

v-if 同 v-show 的对比：

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

v-if 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时才会开始渲染条件块。v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来讲，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销，因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

3.5.7 v-for 指令

v-for 指令的作用是遍历数组元素，并且渲染到 ui。

语法如下：

```
<标签 v-for="item in|of items" :key="item">...</标签>
or
<标签 v-for="(item, index) in|of items" :key="index">...</标签>
```

1）遍历数组

案例代码如下：

```
<div id="app">
    <div v-for="item in items" :key="item.msg">
            <span v-text="item.msg"></span>
    </div>
    <div v-for="(item, index) in items" :key="index">
            <p v-text="item.msg + '->' + index"></p>
    </div>

    <div v-for="item of items" :key="item.msg">
            <span v-text="item.msg"></span>
    </div>
    <div v-for="(item, index) of items" :key="index">
```

```
        <p v-text="item.msg + '->' + index"></p>
    </div>
</div>
<script>
    const vm = new Vue({
            el:"#app",
            data:{
                items:[
                  {msg:'m1'},
                  {msg:'m2'},
                  {msg:'m3'},
                  {msg:'m4'},
                ]
            }
    })
</script>
```

2）遍历对象

案例代码如下：

```
<div id="app">
    <div v-for="value in user" :key='value'>
            {{value}}
    </div>

    <div v-for="(value, name) in user" :key='name'>
            {{name}}->{{value}}
    </div>

    <div v-for="(value, name, index) in user" :key='name+index'>
            {{index}}:{{name}}->{{value}}
    </div>
</div>
<script>
    const vm = new Vue({
            el:"#app",
            data:{
                user:{
                    name:'zhangsan',
                    age:12,
                    sex:'男',
                }
            }
    })
</script>
```

3.5.8 v-on 指令

v-on 指令的作用是绑定事件监听器。事件类型由参数指定。表达式可以是一种方法的名字或一个内联语句,如果没有修饰符,则可以省略。

当用在普通元素上时,只能监听原生 DOM 事件。当用在自定义元素组件上时,也可以监听子组件触发的自定义事件。

在监听原生 DOM 事件时,方法以事件为唯一的参数。如果使用内联语句,则语句可以访问一个 $event property: v-on: click="handle('ok', $event)"

语法如下:

```
<标签 v-on:事件名称[.修饰符]="表达式"> ... </标签>
```

v-on 指令可以缩写为 @。

v-on 支持的修饰符如下。

.stop: 调用 event.stopPropagation()。

.prevent: 调用 event.preventDefault()。

.capture: 添加事件侦听器时使用 capture 模式。

.self: 只当事件是从侦听器绑定的元素本身触发时才触发回调。

.{keyCode |keyAlias}: 只当事件是从特定键触发时才触发回调。

.native: 监听组件根元素的原生事件。

.once: 只触发一次回调。

.left: (2.2.0)只当单击鼠标左键时触发。

.right: (2.2.0)只当右击时触发。

.middle: (2.2.0)只当单击鼠标中键时触发。

.passive: (2.3.0)以{passive: true}模式添加侦听器。

案例代码如下:

```
<!-- 方法处理器 -->
<button v-on:click="doThis"></button>

<!-- 动态事件 (2.6.0+) -->
<button v-on:[event]="doThis"></button>

<!-- 内联语句 -->
<button v-on:click="doThat('hello', $event)"></button>

<!-- 缩写 -->
<button @click="doThis"></button>

<!-- 动态事件缩写 (2.6.0+) -->
```

```
<button @[event] = "doThis"></button>

<!-- 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">

<!-- 单击回调只会触发一次 -->
<button v-on:click.once = "doThis"></button>

<!-- 对象语法 (2.4.0+) -->
<button v-on = "{ mousedown: doThis, mouseup: doThat }"></button>
```

3.6 Vue.js 响应原理

Vue.js 的一个重要特色是实现了 View 和 Model 的自动同步响应,接下来介绍一下 Vue.js 的响应原理和特殊数据的检测响应。

3.6.1 响应式原理

当项目代码把一个普通的 JavaScript 对象传入 Vue.js 实例作为 data 选项时,Vue.js 将遍历此对象所有的 property,并使用 Object.defineProperty 把这些 property 全部转换为 getter/setter。Object.defineProperty 是 ES5 中一个无法 shim(低版本不兼容)的特性,这也就是 Vue.js 不支持 IE 8 及更低版本浏览器的原因。

这些 getter/setter 对用户来讲是不可见的,但是在内部它们让 Vue.js 能够追踪依赖,在 property 被访问和修改时通知变更。这里需要注意的是不同浏览器在控制台打印数据对象时对 getter/setter 的格式化并不同。

每个组件实例都对应一个 watcher 实例,它会在组件渲染的过程中把“接触”过的数据 property 记录为依赖。之后当依赖项的 setter 被触发时,会通知 watcher,从而使与它关联

的组件重新被渲染，如图 3.3 所示。

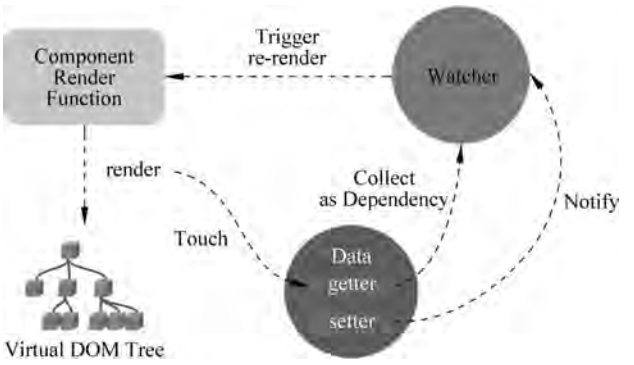

图 3.3　响应式原理图

由于 JavaScript 的限制，Vue.js 不能检测数组和对象的变化。尽管如此，状态管理工具 Vuet 提供了一些办法来回避这些限制并保证它们的响应性。

3.6.2　对象的检测响应

Vue.js 无法检测对象 property 的添加或移除。由于 Vue.js 会在初始化实例时对 property 执行 getter/setter 转换，所以 property 必须在 data 对象上存在才能让 Vue.js 将它转换为响应式的。

同时 Vue.js 不允许动态添加根级响应式 property，所以开发人员必须在初始化实例前声明所有根级响应式 property，哪怕只是一个空值。如果未在 data 选项中声明 message，则 Vue.js 将警告渲染函数正在试图访问不存在的 property，代码如下：

```
var vm = new Vue({
  data: {
    //将 message 声明为一个空值字符串
    message: ''
  },
  template: '<div>{{ message }}</div>'
})
//之后设置 `message`
vm.message = 'Hello!'
```

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue.js 实例能更好地配合类型检查系统工作，但与此同时在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的结构(schema)。提前声明所有的响应式 property，可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

对于已经创建的实例，Vue.js 不允许动态添加根级别的响应式 property，但是，可以使用 Vue.set(object，propertyName，value)方法向嵌套对象添加响应式 property，代码如下：

```
Vue.set(vm.someObject, '新属性名称', 属性值)
```

还可以使用 vm. $set 实例方法,这也是全局 Vue. set()方法的别名,代码如下:

```
this. $set(this.someObject,'b',2)
```

如果需要添加多个属性,则可以使用 Object. assign(),将新对象的属性(可以是多个属性)与原对象的属性一起混合进一个新对象,代码如下:

```
//第 3 章/检测对象.html
…
<div id = "app">
    <p>{{a}}</p>
    <p>{{user.userName}}</p>
    <p>{{user.age}}</p>
    <p>{{user.sex}}</p>
    <p>{{user.code}}</p>
    <p>{{user.name}}</p>
</div>
<script>
    const _address = {code:'123'}
    const vm = new Vue({
            el:"#app",
            data:{
                a:1,
                user:{
                    userName:'zhangsan',
                }
            }
    })
    vm.user.age = 12;
    //能输出 age 的值,但是不能响应到 View
    console.log(vm.user.age + "," + vm.user.sex)
    Vue.set(vm.user, 'sex', '男')
    //输出用户对象,并且能响应到 View
    console.log(vm.user)
    //全部合并成 vm.user 对象,响应到 View
    vm.user = Object.assign({},vm.user, {code:'001',name:'张三'})
</script>
…
```

3.6.3 数组的检测响应

Vue.js 不能检测以下数组的变动:

(1) 当利用索引直接设置一个数组项时,Vue.js 检测不到数组的变动。

例如：vm.items[indexOfItem]=newValue。

(2) 当修改数组的长度时，Vue.js 检测不到数组的变动。

例如：vm.items.length=newLength。

Vue.js 提供了两种方式都可以实现和 vm.items[indexOfItem]=newValue 相同的效果，同时也将在响应式系统内触发状态更新。一种方式是调用 Vue.js 的 set()方法，替代数组指定下标的值；另一种是调用数组的 splice()方法，替换数组指定下标的值。可以修改数组指定索引对应元素的值，Vue.js 检测不到变动的问题，代码如下：

```
//Vue.set
Vue.set(vm.items, indexOfItem, newValue)
//Array.prototype.splice
//splice(index,howmany,item1…itemn) 删除从 index 开始的 howmany 个元素
//用 item1..n 替代这些删除的元素，并且返回删除的元素
vm.items.splice(indexOfItem, 1, newValue)
```

也可以使用 vm.$set()实例方法，该方法是全局方法 Vue.set()的一个别名，代码如下：

```
vm.$set(vm.items, indexOfItem, newValue)
```

为了修改数组长度，Vue.js 检测不到变动的问题可以使用 splice，代码如下：

```
vm.items.splice(newLength)
```

综合示例代码如下：

```
//第 3 章/检测数组.html
<div id="app">
    <p>{{ages}}</p>
</div>
<script>
    const vm = new Vue({
            el:"#app",
            data:{
                ages:[1,2,3]
            }
    })
    vm.ages[1] = 20;
    //输出修改，但不响应到 View
    console.log(vm.ages)
    //响应到 View
    Vue.set(vm.ages, 1, 21)
    vm.$set(vm.ages, 1, 22)
    vm.$set(vm.ages, vm.ages.length,4)
    console.log(vm.ages)
```

```
    //修改数组长度,响应到 View
    vm.ages.splice(vm.ages.length - 1)
</script>
```

除了前面介绍的 set()方法和 splice()方法外,Vue.js 还提供了一些其他方法,实现变更数组和替换数组,同时还能检测到数组的变化。

1. 变更数组方法

变更方法会改变以前的数组(原数组)。Vue.js 将被侦听的数组的变更方法进行了包裹,所以它们也将会触发视图更新。被包裹的变更数组的方法如下。

1) push()

在数组中添加一个新元素,并返回数组的新长度值,代码如下:

```
arrayObj.push([item1 [item2 [. . . [itemN ]]]])
```

2) pop()

移除数组中的最后一个元素并返回该元素,代码如下:

```
arrayObj.pop()
```

3) shift()

移除数组中的第 1 个元素并返回该元素,代码如下:

```
arrayObj.shift()
```

4) unshift()

将指定的元素插入数组的开始位置并返回新数组的长度,代码如下:

```
arrayObj.unshift([item1[, item2 [, . . . [, itemN]]]])
```

5) splice()

从一个数组中移除一个或多个元素,如果必要,则在所移除元素的位置上插入新元素,并且返回所移除的元素,代码如下:

```
arrayObj.splice(start, deleteCount,[item1[, item2[, . . . [,itemN]]]])
```

6) sort()

返回一个对元素完成排序的数组,代码如下:

```
arrayobj.sort(sortfunction)
```

如果为 sortfunction 参数提供了一个函数,则表示在执行 sort 时,基于 sortfunction 对元素进行大小比较。该函数必须返回下列值之一:

(1) 负值,如果所传递的第 1 个参数比第 2 个参数小。

(2) 零,如果两个参数相等。

(3) 正值,如果第 1 个参数比第 2 个参数大。

7) reverse()

返回一个数组元素被反转的数组,代码如下:

```
arrayObj.reverse()
```

2. 替换数组方法

替换返回,不会改变以前的数组,但是会返回一个新的数组。

1) filter()

根据 filter 条件返回新数组,代码如下:

```
array.filter(function(currentValue, indedx, arr), thisValue)
```

currentValue:必须,当前元素的值;index:可选,当前元素的索引值;arr:可选,当前元素属于的数组对象;thisValue:可选,对象作为该执行回调时使用,传递给函数,用作"this"的值。如果省略了 thisValue,则"this"的值为"undefined"。以下代码演示了 filter 方法的使用:

```
var arr = [3,9,4,3,6,0,9];
//返回大于 5 的值
function max5(arr){
     return arr.filter(x) = (> x > 5);
}
//移除数组中与 item 相同的值并返回
function remove(arr, item) {
    return arr.filter(val => val != item);
}
//移除重复值
var r = arr.filter(function (element, index, self) {
        return self.indexOf(element) == index;
});
//查找所有同 item 值相同的元素的位置
function findAllOccurrences(arr, target) {
    var res = [];
    arr.filter(function(ele,index){
        return (ele === target)&&res.push(index);
    })
    return res;
}
```

2) concat()

返回一个新数组,这个新数组是由两个或更多个数组组合而成的,代码如下:

```
array1.concat([item1[, item2[, . . . [, itemN]]]])
```

3) slice()

截取一段子数组并返回,代码如下:

```
arrayObject.slice(start,end)              //包含开始,不包含结尾
```

3.6.4 异步更新问题

Vue.js 在更新 DOM 时是异步执行的。只要侦听到数据变化,Vue.js 将开启一个队列,并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发,则只会被推入队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的,然后,在下一个事件的循环 tick 中,Vue.js 将刷新队列并执行实际(已去重的)工作。Vue.js 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate,如果执行环境不支持,则会采用 setTimeout(fn,0)代替。

例如,如果设置 vm.someData='new value',则该组件不会立即被重新渲染。当刷新队列时,组件会在下一个事件循环 tick 中更新。多数情况下开发人员不需要关心这个过程,但是如果开发人员想基于更新后的 DOM 状态来做点什么,这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员使用"数据驱动"的方式思考,避免直接接触 DOM,但是有时开发人员必须这么做。为了在数据变化之后等待 Vue.js 完成更新 DOM 操作,可以在数据变化之后立即使用 Vue.nextTick(callback)。这样回调函数将在 DOM 更新完成后被调用,代码如下:

```
<div id="app">
    <div>{{message}}</div>
</div>
<script>
    const vm = new Vue({
            el:"#app",
            data:{
                        message:"123"
        }
    })
    vm.message = 'new message' //更改数据
    let bool = vm.$el.textContent === 'new message' //false
    console.log(bool)
    Vue.nextTick(function () {
```

```
        let bool = vm.$el.textContent === 'new message' //true
        console.log(bool)
      })
</script>
```

在组件内使用 vm.$nextTick() 实例方法特别方便，因为它不需要全局 Vue.js，并且回调函数中的 this 将自动被绑定到当前的 Vue.js 实例上，代码如下：

```
Vue.component('example', {
  template: '<span>{{ message }}</span>',
  data: function ) {
    return {
      message: '未更新'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = '已更新'
      console.log(this.$el.textContent) // => '未更新'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => '已更新'
      })
    }
  }
})
```

因为 $nextTick() 会返回一个 Promise 对象，所以开发人员可以使用新的 ES2017 async/await 语法完成相同的事情，代码如下：

```
methods: {
  updateMessage: async function () {
    this.message = '已更新'
    console.log(this.$el.textContent) // => '未更新'
    await this.$nextTick()
    console.log(this.$el.textContent) // => '已更新'
  }
}
```