第 5 章

组件化编程

组件(Component)是 Vue.js 最强大的功能之一。通过组件,可以扩展 HTML 元素,以及封装可重用的代码。有了组件后,开发人员就可以使用独立可复用的小组件,构建大型的前端应用。绝大多数应用的界面可以抽象成一个组件树。例如一个页面可能会有页头、侧边栏、内容区等部分,这些都可以封装成一个个独立的组件,然后按关系组合起来,形成一个完整的界面,如图 5.1 所示。

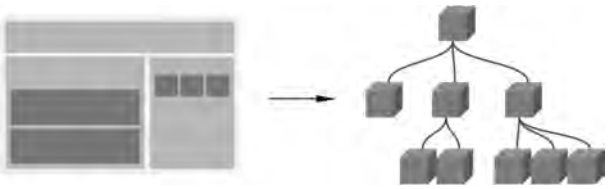

图 5.1　应用界面和组件树

本章将介绍怎样定义和使用组件,以及使用组件过程中的各种方式和技巧。

5.1　第 1 个组件

定义一个计算器组件,实现单击后自动累计按钮被单击的次数。在一个页面上就可以重复地使用这个计算器组件,各种累计自己被单击的次数,代码如下:

```
<div id = 'app'>
    <! -- 重复使用 ButtonCounter 组件 -- >
    <button - counter ></button - counter >
    <button - counter ></button - counter >
    <button - counter ></button - counter >
    <button - counter ></button - counter >
</div >
<script type = 'text/JavaScript'>
    //定义一个计算器组件 -- ButtonCounter
    Vue.component("ButtonCounter", {
```

```
        data(){
            return {
                count: 0
            }
        },
        template:`<button v-on:click="count++">单击了我{{count}}次</button`
    })
    const vm = new Vue({
        el: '#app',
        data: {
        },
        methods: {
        }
    })
</script>
```

5.2 使用自定义组件

自定义组件的步骤分为三步：定义并创建组件、注册组件和使用组件。接下来以计算器按钮组件为案例，介绍怎样自定义组件。

5.2.1 自定义组件

一个组件一般由视图、数据和业务逻辑三部分组成，所以在所定义的组件对象中，一般包含 template、data 和 methods。创建一个组件对象，通过 template、data 和 methods 属性定义组件包含的视图、数据和业务逻辑，代码如下：

```
//第 5 章/计算器按钮组件.html
…
<script type='text/JavaScript'>
    const buttonCounter = {
        template: `<button v-on:click="counter">我被单击了{{count}}次</button>`,
        data(){
            return {
                count: 0
            }
        },
        methods:{
            counter(){
                this.count++
            }
        }
    }
```

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

作为子组件(在一个页面中,有且只能有一个根 Vue.js 实例),data 属性必须是一个函数,返回包含子组件实例所有数据属性的对象。也就是说,一定要写成如上代码的形式,而不能写成如下的形式:

```
...
  data: {
     count: 0
  }
...
```

如果 templat 的视图内容比较复杂,直接写在组件对象的 template 数字后面会太复杂,不方便理解和维护,这时候可以使用< template >元素单独定义,再在组件对象的 template 属性中使用,代码如下:

```
...
<template id="buttonCounterTemplate">
    <button v-on:click="counter">我被单击了{{count}}次</button>
</template>
<script type='text/JavaScript'>
    const buttonCounter = {
        template: "#buttonCounterTemplate", //引用单独定义的模板
        ...
    }
    ...
</script>
...
```

使用< template >单独定义模板时需要注意,里面的内容有且只能有一个根元素。如果视图中有很多元素,则需要用一个大元素(例如 div)包含全部小元素。

5.2.2 全局注册组件

定义好组件对象后,在使用时需要先注册。注册组件有两种方式:全局注册和局部注册。这里先介绍全局注册的特点和方式。

全局注册的特点是，注册一个组件，在应用的其他组件中都可以使用。其优势是只要注册一次，其他要使用的地方不用再注册，可直接使用。

全局注册方式调用的是 Vue.js 的 component 方法，带两个参数，第 1 个参数是注册的组件名称，第 2 个参数是要注册的组件对象，代码如下：

```
Vue.component(组件名称, 组件对象)
```

使用 Vue.component()方法，全局注册前面定义的 buttonCounter 组件对象，组件的名称是 ButtonCounter，代码如下：

```
…
< script >
  …
  Vue.component("ButtonCounter", buttonCounter)
  …
</script
…
```

如果组件直接在 DOM 中使用，则组件名称的命名，除了遵循见名思意的原则外，强烈推荐遵循 W3C 规范中的自定义组件名（字母全小写且必须包含一个连字符）。这样可更好地避免在使用组件的时候，同当前及未来的 HTML 元素相冲突。

定义组件名的方式有两种。

1. 使用 kebab-case

```
Vue.component('my - component - name', { / * … * / })
```

当使用 kebab-case（短横线分隔命名）定义一个组件名时，开发人员必须在引用这个自定义元素时使用 kebab-case，例如 < my-component-name >。

2. 使用 PascalCase

```
Vue.component('MyComponentName', { / * … * / })
```

当使用 PascalCase（首字母大写命名）定义一个组件名时，开发人员在引用这个自定义元素时两种命名法都可以使用。也就是说< my-component-name >和< MyComponentName >都是可接受的。注意，尽管如此，直接在 DOM(非字符串的模板)中使用时只有 kebab-case 是有效的。

5.2.3 局部注册组件

全局注册是一次注册，在任何组件中都可以用，在方便使用的同时，也有劣势存在。例如，如果开发者使用一个像 webpack 这样的构建系统，则全局注册所有的组件意味着即便不再使用一个组件了，它仍然会被包含在最终的构建结果中。这造成了用户下载的

JavaScript 中包含了没必要的内容。

在这些情况下，开发人员可以通过一个普通的 JavaScript 对象来定义组件，代码如下：

```
var ComponentA = { /* ... */ }
var ComponentB = { /* ... */ }
var ComponentC = { /* ... */ }
```

然后在 components 选项中定义开发人员想要使用的组件，代码如下：

```
new Vue({
  el: '#app',
  components: {
    'component-a': ComponentA,
    'component-b': ComponentB
  }
})
```

对于 components 对象中的每个 property 来讲，其 property 名就是自定义元素的名字，其 property 值就是这个组件的选项对象。

注意局部注册的组件在其子组件中是不可用的。例如，如果希望 ComponentA 在 ComponentB 中可用，则需要写成如下代码的形式：

```
var ComponentA = { /* ... */ }

var ComponentB = {
  components: {
    'component-a': ComponentA
  },
  ...
}
```

或者通过 Babel 和 webpack 使用 ES2015 模块，这样代码看起来更科学，代码如下：

```
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  },
  ...
}
```

注意在 ES2015+中，在对象中放一个类似 ComponentA 的变量名其实是 ComponentA: ComponentA 的缩写，即这个变量名同时表示变量名和将对象赋给变量名。

在 Vue.js 根实例组件中，使用局部注册的方式注册 ButtonCounter 组件，代码如下：

```
//第 5 章/局部注册.html
…
<div id = 'app'>
    <button - counter></button - counter>
</div>
<template id = "buttonCounterTemplate">
    <button v - on:click = "counter">我被单击了{{count}}次</button>
</template>
<script type = 'text/JavaScript'>
    const ButtonCounter = {
        template: "#buttonCounterTemplate",
        data(){
            return {
                count: 0
            }
        },
        methods:{
            counter(){
                this.count++
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            ButtonCounter
        },
        data: {
        },
        methods: {
        }
    })
</script>
…
```

在模块式开发前端应用的项目中,还可以使用 import/require 方式完成组件的局部注册。

5.2.4 使用组件

组件注册完后,就可以用标签的方式,在其他组件中使用被注册过的组件,注意全局注册的组件,可以用在任何组件的视图中,而局部注册的组件,则只能用在注册的当前组件中。定义 ComponentA、ComponentB 和 ComponentC 3 个组件,使用全局注册的方式注册 ComponentA 组件,在 vm 中局部注册 ComponentB 和 ComponentC 两个组件。这样 ComponentA 组件,既可以在 vm 中使用,也可以在 ComponentC 中使用,而 ComponentB

组件，只能在 vm 中使用，当在 ComponentC 组件中使用的时候，就会抛出异常，代码如下：

```
//第 5 章/使用子组件.html
…
<div id = 'app'>
    <component - a></component - a>
    <component - b></component - b><br/>
    <component - c></component - c>
</div>
<template id = "componentC">
    <div>
        <! -- 使用 ComponentA 按钮正常 -->
        <component - a></component - a><br/>
        <! -- 使用 ComponentB 按钮抛出异常 -->
        <component - b></component - b>
    </div>
</template>
<script type = 'text/JavaScript'>
    const ComponentA = {
        template:`<button>ComponentA 按钮</button>`
    }
    //全局注册 ComponentA,所以可以在 vm 和 ComponentC 中使用
    Vue.component('ComponentA', ComponentA)

    const ComponentB = {
        template:`<button>ComponentB 按钮</button>`
    }
    const ComponentC = {
        template:'#componentC'
    }

    const vm = new Vue({
        el: '#app',
        components:{
            ComponentB,
            ComponentC
        },
        data: {
        },
        methods: {
        }
    })
</script>
…
```

5.3 父组件将值传到子组件

在 Vue.js 中,可以在定义子组件中定义多个 prop 属性,用来接收父组件传过来的数据。也就是说,父组件可以通过子组件的 prop 属性,给子组件传递值。定义一个 ViewCount 组件,显示通过 propCount 传入的值。在 vm 根实例组件中,注册并且使用 ViewCount 组件,显示单击按钮累计的单击次数,代码如下:

```
//第 5 章/父组件通过 prop 属性给子组件传值.html
…
<div id = 'app'>
    <button v - on:click = "clickMe">单击我</button>
    <! -- 使用 ViewCount 组件显示单击次数 -- >
    <view - count :prop - count = 'count'></view - count>
</div>
<template id = "viewCountTemplate">
    <div>{{propCount}}</div>
</template>
<script type = 'text/JavaScript'>
    const ViewCount = {
        props:['propCount'],
        template:"#viewCountTemplate"
    }
    const vm = new Vue({
        el: '#app',
        components:{
            ViewCount
        },
        data: {
            count:0
        },
        methods: {
            clickMe(){
                this.count++
            }
        }
    })
</script>
…
```

传值的语法如下:

```
<子组件名称 :prop 属性名称 = "表达式"></子组件名称>
```

或

```
<子组件名称 v-bind:prop 属性名称 = "表达式"></子组件名称>
```

5.3.1 prop 的大小写

HTML 中的 attribute 名对大小写不敏感，所以浏览器会把所有大写字符解释为小写字符。这意味着当开发人员使用 DOM 中的模板时，用 camelCase（驼峰命名法）命名的 prop 名需要使用其等价的 kebab-case（短横线分隔命名法）命名，代码如下：

```
Vue.component('sub-component', {
  //在 JavaScript 中使用的是 camelCase
  props: ['postTitle'],
  template: '<h3>{{ postTitle }}</h3>'
})

<!-- 在 HTML 中使用的是 kebab-case -->
<sub-component post-title = "hello!"></sub-component>
```

5.3.2 prop 的数据类型

prop 除了支持数字和 string 类型外，还支持其他类型，代码如下：

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object,
  callback: Function,
  contactsPromise: Promise //or any other constructor
}
```

这不仅为组件提供了使用参考文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。使用方式和说明的样例代码如下。

1. 传入一个数字

```
<!-- 即便 '42' 是静态的，仍然需要 'v-bind' 来告诉 Vue.js -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串. -->
<sub-component v-bind:likes = "42"></sub-component>

<!-- 用一个变量进行动态赋值. -->
<sub-component v-bind:likes = "post.likes"></sub-component>
```

2. 传入一个布尔值

```
<!-- 包含该 prop 没有值的情况在内，都意味着 'true'. -->
<sub-component is-published></sub-component>
```

```
<!-- 即便 'false' 是静态的,仍然需要 'v-bind' 来告诉 Vue.js -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串. -->
<sub-component v-bind:is-published="false"></sub-component>

<!-- 用一个变量进行动态赋值. -->
<sub-component v-bind:is-published="post.isPublished"></sub-component>
```

3. 传入一个数组

```
<!-- 即便数组是静态的,我们仍然需要 'v-bind' 来告诉 Vue.js -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串. -->
<sub-component v-bind:comment-ids="[234, 266, 273]"></sub-component>

<!-- 用一个变量进行动态赋值. -->
<sub-component v-bind:comment-ids="post.commentIds"></sub-component>
```

4. 传入一个对象

```
<!-- 即便对象是静态的,仍然需要 'v-bind' 来告诉 Vue.js -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串. -->
<sub-component
  v-bind:author="{
    name: 'Veronica',
    company: 'Veridian Dynamics'
  }"
></sub-component>

<!-- 用一个变量进行动态赋值. -->
<sub-component v-bind:author="post.author"></sub-component>
```

5. 传入一个对象的所有 property

如果开发人员要将一个对象的所有 property 都作为 prop 传入,则可以使用不带参数的 v-bind(取代 v-bind:prop-name)。将 user 对象的所有属性传递到组件中,代码如下:

```
<template id="subUserId">
 <div>
 <p>name:{{userAttr.name}}</p>
 <p>age:{{userAttr.age}}</p>
 </div>
</template>
<div id="app">
 <!-- 传入用户对象 -->
 <user-component :user-attr="user"></user-component>
</div>
```

```
<script>
 Vue.component("user-component",{
   props:['userAttr'],
   template:"#subUserId"
 })
 const vm = new Vue({
   el:"#app",
   data:{
   user:{
      name:'张三',
      age:12
   }
  }
 })
</script>
```

5.3.3 prop 单向数据流

所有的 prop 都使其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外变更父级组件的状态，从而导致应用的数据流向难以理解。

另外，每次父级组件发生变更时，子组件中所有的 prop 都将被刷新为最新的值。这意味着开发人员不应该在一个子组件的内部改变 prop。如果这样做了，Vue.js 则会在浏览器的控制台中发出警告。

这里有两种常见的试图变更一个 prop 的情形：

(1) 使用 prop attribute 将一个初始值传递给子组件的本地属性，子组件直接操作本地属性。

(2) 在子组件中定义计算属性，基于 prop attribute 传入的值进行计算处理。

代码如下：

```
//第 5 章/prop 数据流向.html
…
<!-- 定义模板 -->
<template id="template1">
    <div>
        <!-- 初始值方式 -->
        <p>
            <span>传入的 counter:{{counter}}</span><br>
            subCounter:<input v-model="subCounter" /><br>
            <span>子组件更新 counter:{{subCounter}}</span>
        </p>
```

```
        <!-- 计算属性方式 -->
        <p>
            <span>传入的 size:{{size}}</span><br>
            subSize:<input v-model="computeSize" /><br>
            <span>子组件更新 size:{{computeSize}}</span>
        </p>
    </div>
</template>

<div id="app">
    counter:<input v-model="counter"/><br>
    size:<input v-model="size" />
    <sub-test :counter="counter" :size="size"></sub-test>
</div>
<script>
    const subVue = {
        props:['counter', 'size'],          //定义 prop 属性接收父组件的值
        data:function(){
            return {
                subCounter:this.counter     //给本地属性赋值
            };
        },
        computed:{
            computeSize:{              //定义计算属性,处理 prop 属性传入的值后给子组件使用
                get:function(){
                    return 'result->' + this.size;
                },
                set:function(newValue){
                    //不能响应到父组件,会抛出异常
                    this.size = newValue.slice("result->".length)
                }
            }
        },
        template:"#template1"
    }
    const vm = new Vue({
        el:"#app",
        data:{
            counter:1,
            size:10
        },
        components:{subTest:subVue}
    })
</script>
...
```

5.3.4 prop 属性验证

开发人员还可以为组件的 prop 指定验证要求。如果有一个需求没有被满足，则 Vue.js 会在浏览器控制台中警告。这在开发一个会被别人用到的组件时尤其有帮助。

为了定制 prop 的验证方式，开发人员可以为 props 中的值提供一个带有验证需求的对象，而不是一个字符串数组，代码如下：

```
//第 5 章/验证 prop 属性.html
…
<template id="template1">
    <div>
        <span>{{attr1}}</span><br>
        <span>{{attr2}}</span><br>
        <span>{{attr3}}</span><br>
        <span>{{attr4}}</span><br>
        <span>{{attr5.message}}</span><br>
        <span>{{attr6}}</span><br>
    </div>
</template>
<div id="app">
<son-component
:attr1="a1"
:attr2="a2"
:attr3="a3"
:attr4="a4"
:attr5="a5"
:attr6="a6"></son-component>
</div>
<script>
    const SonComponent={
        props:{
            attr1:Number,
            attr2:[String,Number],
            attr3:{
                type:String,
                required:true
            },
            attr4:{
                type:Number,
                default:10
            },
            attr5:{
                type:Object,
                default:function(){
```

```
                    return {message:'hello'}
                }
            },
            attr6:{
                type:String,
                validator:function(value){
                    //这个值必须匹配下列字符串中的一个
                    return ['success', 'warning', 'danger'].indexOf(value) !== -1
                }
            }
        },
        template:"#template1"
    }
    const vm = new Vue({
        el:"#app",
            components:{
            SonComponent
        },
        data:{
            a1:1,
            a2:'hello',
            a3:'world',
            a4:11,
            a5: {message:'hai'},
            a6:'success'
        }
    })
</script>
…
```

当 prop 验证失败时，Vue.js 将会发出一个控制台警告。

实例的属性是在对象创建之前进行验证的，所以实例的属性（如 data 和 computed）在 default 和 validator 函数中不可用。

prop 支持的类型包括：String、Number、Boolean、Array、Object、Date、Function、Symbol，同时支持自定义的构造函数，能使用 instanceof 进行确认，代码如下：

```
//第 5 章/instanceof 的使用.html
…
<template id = "template1">
    <div>
        {{personAttr.firstName}} {{personAttr.lastName}}
    </div>
</template>
```

```
<div id = "app">
    <son - component :person - attr = "person"></son - component >
</div >
<script >
    function Person(first,last){
        this.firstName = first;
        this.lastName = last;
    }
    const SonComponent = {
        template:"#template1",
        props:{
            personAttr:{
                type: Person,
                validator:function(value){
                    return value instanceof Person;
                }
            }
        }
    }
    const vm = new Vue({
        el:"#app",
        components:{
            SonComponent
        },
        data:{
            person:new Person("san", "zhang")
        }
    })
</script >
…
```

5.3.5 非 prop 的 attribute

组件可以接受任意的 attribute，而这些 attribute 会被添加到这个组件的根元素上。

显式定义的 prop 适用于向一个子组件传入信息，这也是 Vue.js 中推荐的做法，即向子组件传值的方式，然而组件库的作者并不总能预见组件会被用于怎样的场景。这也是为什么组件可以接受任意的 attribute，而这些 attribute 会被添加到这个组件的根元素上。

如下例子中的 son-component 组件的 notprop 属性，没有在 son-component 的 props 属性中定义，但是会被直接渲染到子组件的根元素（div 元素）中，代码如下：

```
//第 5 章/非 prop 属性.html
…
<template id = "template1">
    <div class = "subClass" name = "subName">子组件</div >
```

```
</template>
<div id="app">
    <son-component
        :notprop="notPropValue"
        :class="clsValue"
        :name="nameValue"></son-component>
</div>
<script>
    const SonComponent = {
        template:"#template1"
    }
    const vm = new Vue({
        el:"#app",
        data:{
            notPropValue:'hello',
            clsValue:'parentClass',
            nameValue:'parentName'
        },
        components:{
            SonComponent
        }
    })
    vm.$data.notPropValue="hai"
</script>
...
```

渲染代码如下：

```
<div notprop="hai"
class="parentClass subClass"
name="parentName">子组件</div>
```

div 中的 notprop="hai"是从<son-component：notprop="notPropValue"></son-component>传递过去的。

1. 替换/合并已有的 attribute

如果在子组件中也定义了非 prop 的 attribute，同时在使用组件的时候也定义了该 attribute，这时候最后的值，存在替换/合并问题。class 和 style 的值会被合并，其他属性值会被替换。

如上面的代码中，在 son-component 组件中定义了 class="subClass" name="subName"，同时在组件使用的时候定义了：class="clsValue"和：name="nameValue"，最后渲染的结果是 class="parentClass subClass"和 name="parentName"。class 属性的值被合并了，而 name 属性的值只有一个：外面的值被替换了组件里面的 name 值。

2. 禁用 Attribute 继承

如果开发人员不希望组件的根元素继承 attribute，则可以在组件的选项中设置 inheritAttrs：false

因为继承的 attribute 只能作用到根元素上，如果需要将 attribute 继承到子组件的非根元素上，则可以使用 v-bind="$attrs"将 attribute 绑定到子元素的非根元素上，代码如下：

```
<div id="app">
  <son-component test='tValue' required
placeholder='请输入姓名'></son-component>
 </div>
 <script>
  Vue.component("SonComponent",{
      inheritAttrs:false,
      template:`<div><input type='text' v-bind="$attrs"/></div>`
  })
  const vm = new Vue({
      el:"#app"
  })
 </script>
```

注意：class 和 style 属性不在作用范围。

5.4 子组件将值传到父组件

组件的 prop 属性只能实现父组件向子组件传值，在实际的前端项目中，需要实现子组件将值传给父组件。Vue.js 提供了 3 种机制，实现子组件将值传给父组件。

5.4.1 使用 $emit 方法调用父组件方法传值

在 Vue.js 的父组件中，可以通过 v-on 指令，给子组件的指定事件绑定一个函数，在子组件中，用 $emit 方法触发自己的事件，从而执行被绑定的函数。$emit 方法的第 1 个参数是一个字符串，对应 v-on 指定的事件名称，父组件中使用 v-on 给 son-component 组件的 parent-method 事件绑定了定义在父组件中 parentMethod 函数，在 son-component 的 toTest 函数中，使用 this.$emit('parent-method')方式触发 parent-method 事件，执行 parentMethod 方法，实现父组件中的 count 自增，代码如下：

```
//第5章/子组件基于$emit调用父组件的方法.html
...
<div id='app'>
    <son-component v-on:parent-method="parentMethod"></son-component>
    <br/>
    <div>{{count}}</div>
```

```
</div>
<template id="sonComponent">
    <button v-on:click='toTest'>单击子组件</button>
</template>
<script type='text/JavaScript'>
    const SonComponent = {
        template:'#sonComponent',
        methods:{
            toTest(){
                this.$emit('parent-method')
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            SonComponent
        },
        data: {
            count:0
        },
        methods: {
            parentMethod(){
                this.count++
            }
        }
    })
</script>
…
```

$emit 方法必须有一个参数指定要触发的事件，同时支持更多的可选参数，通过这些参数，子组件可以将自己的数据传递给事件绑定的方法，而绑定的方法是定义在父组件中的，所以就可以间接地使用 $emit 方法，将子组件中的数据传递给父组件。在 son-component 子组件的 toTest 方法中，通过第 2 个参数给父组件中绑定的 parentMethod 方法传递 count 的递增幅度 step，代码如下：

```
//第5章/使用$emit方法传值.html
…
<div id='app'>
    <son-component v-on:parent-method="parentMethod"></son-component>
    <br/>
    <div>{{count}}</div>
</div>
<template id="sonComponent">
```

```
    <button v-on:click='toTest'>单击子组件</button>
</template>
<script type='text/JavaScript'>
    const SonComponent = {
        template:'#sonComponent',
        methods:{
            toTest(){
                this.$emit('parent-method', 2)
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            SonComponent
        },
        data: {
            count:0
        },
        methods: {
            parentMethod(step){
                this.count += step
            }
        }
    })
</script>
...
```

5.4.2 调用父组件的方法传值

prop 属性的数据类型支持 Function，利用这个特点，开发人员可以在父组件中定义一个 Function 类型的 prop 属性，给子组件传递一个函数对象，在子组件中调用这个函数，通过函数的参数，可以将子组件中的数据传递给父组件。父组件基于子组件的 funcData prop 属性，给子组件 son-component 传递 increment 函数对象，在子组件 son-component 的 toTest 方法中，调用传入的 increment 函数，并且传入参数 step 的值，代码如下：

```
//第5章/调用父组件的方法传值.html
...
<div id='app'>
    <son-component v-bind:func-data="increment"></son-component><br/>
    {{count}}
</div>
<template id="sonComponentTemplate">
    <button v-on:click="toTest">单击递增</button>
```

```
</template>
<script type='text/JavaScript'>
    const SonComponent = {
        template: '#sonComponentTemplate',
        props:{
            funcData:{
                type:Function
            }
        },
        methods:{
            toTest(){
                this.funcData(2)
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            SonComponent
        },
        data: {
            count: 0
        },
        methods: {
            increment(step){
                this.count += step
            }
        }
    })
</script>
...
```

5.4.3 使用 v-model 实现父子组件的数据同步

v-model 指令可以实现 input 输入框同组件数据属性双向同步，改变输入框的值，此值能自动被同步到 Vue.js 实例对象中。同样，改变 Vue.js 实例对象的数据属性，此数据属性也能自动被同步到 input 输入框，代码如下：

```
//第 5 章/使用 v-model 同步数据.html
...
<div id='app'>
    name:{{name}}<br/>
    <input v-model='name'/><br/>
</div>
```

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

实际上,v-model 是 v-bind:value 和 v-on:input 两个指令的组合。v-bind:value 指令将 Vue.js 实例对象的数据属性绑定到 input 元素的 value 属性。v-on:input 指令给 input 元素的 input 事件绑定一个函数,该函数将 input 输入框的 value 属性值赋给 Vue.js 实例对象的数据属性。<input>元素中的 b-bind:value='name',将 name 数据属性的值绑定到 input 的 value 属性上,这样 input 输入框就可以实时显示 name 数据属性的值了。<input>元素中的 v-on:input="demoInputChange($event)"将 demoInputChange 函数绑定到 input 元素的 input 事件上,并且传入了当前的事件对象,当 input 事件触发时自动执行 demoInputChange 函数,将 input 的 value 属性的值赋给 name 数据属性,从而实现了 input 元素中的 value 同 Vue.js 实例对象中的 name 数据属性的双向绑定,代码如下:

```
//第5章/使用v-model同步数据.html
…
<div id='app'>
    name:{{name}}<br/>
    <input v-bind:value='name' v-on:input="demoInputChange($event)"><br/>
</div>
<script type='text/JavaScript'>
    const vm = new Vue({
        el: '#app',
        data: {
            name: ''
        },
        methods: {
            demoInputChange(event){
                this.name = event.target.value
            }
        }
    })
</script>
…
```

既然 v-bind 和 v-on 的组合可以实现 input 元素的 value 属性同 Vue.js 实例对象的数据属性的双向绑定,同样可以用在子组件上,实现子组件的 value 和数据属性的双向绑定,代码如下:

```
//第 5 章/使用 v-model 同步数据.html
…
<div id='app'>
    <!--子组件使用 v-bind 和 v-on:input 的组合-->
    age:{{age}}<br/>
    <son-component v-bind:age="age" v-on:input="sonChange"></son-component>
</div>
<template id="sonComponentTemplate">
    <input type='text' v-bind:value="age" v-on:input="toChange($event)">
</template>
<script type='text/JavaScript'>
    const SonComponent = {
        template: '#sonComponentTemplate',
        props:['age'],
        methods:{
            toChange(event){
                this.$emit('input',event.target.value)
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            SonComponent
        },
        data: {
            age: 0
        },
        methods: {
            sonChange(age){
                this.age = age
            }
        }
    })
</script>
…
```

使用 v-model 合并子组件的 v-bind 和 v-on 指令，代码如下：

```
//第 5 章/使用 v-model 同步数据.html
…
<div id='app'>
    <!--子组件使用 v-bind 和 v-on:input 的组合-->
    age:{{age}}<br/>
    <son-component v-model="age"></son-component>
</div>
```

```
<template id="sonComponentTemplate">
    <input type='text' v-bind:value="age" v-on:input="toChange($event)">
</template>
<script type='text/JavaScript'>
    const SonComponent = {
        template: '#sonComponentTemplate',
        props:['age'],
        methods:{
            toChange(event){
                this.$emit('input',event.target.value)
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            SonComponent
        },
        data: {
            age: 0
        }
    })
</script>
...
```

5.5 Vue.js 组件对象的常用属性

在 Vue.js 中，给 Vue.js 组件对象定义了很多属性，比较常用的有 $data、$props、$parent、$root、$children 和 $refs 等属性。Vue.js 组件对象的属性都是以 $为前缀的，用来区分定义在组件里面的数据属性。这些属性的作用分别如下。

(1) $data：获取 Vue.js 组件的数据属性对象，包含自定义的所有数据属性。

(2) $props：获取 Vue.js 组件的 props 属性对象，包含所有的 prop 属性。

(3) $parent：获取 Vue.js 组件对象的父组件。

(4) $root：获取 Vue.js 组件对象的根对象，否则就是自己。

(5) $children：获取 Vue.js 组件对象的所有子组件数组。

(6) $refs：获取组件对象里面的所有 ref 组件数组，可以根据 ref 名称获取指定的子组件。

这些组件属性的使用，参见如下代码：

```
//第 5 章/Vue.js 实例对象的常用属性.html
...
<div id='app'>
```

```
<son1></son1>

    <hr>
    <son2 ref="second"></son2>
    <br/>
    <button v-on:click="toTest">parent test</button>
</div>
<template id="sonTemplate">
    <div>
        name:{{name}} --- value:{{value}}<br/>
        <button v-on:click="toTest">测试</button>
    </div>
</template>
<script type='text/JavaScript'>
    const Son1 = {
        name: 'Son1',
        template:'#sonTemplate',
        data(){
            return {
                name:'son1',
                value:'value1'
            }
        },
        methods:{
            toTest(){
                console.log(this.$parent.$data.value)
            }
        }
    }

    const Son2 = {
        name: 'Son2',
        template:'#sonTemplate',
        data(){
            return {
                name:'son2',
                value:'value2'
            }
        },
        methods:{
            toTest(){
                console.log(this.$parent.$data.value)
            }
        }
    }
```

```
    const vm = new Vue({
        el: '#app',
        components:{
            Son1,
            Son2
        },
        data: {
            value: 'parentValue'
        },
        methods: {
            toTest(){
                this.$children.forEach(element => {
                    console.log(element.$data.name + ',,,,' + element.$data.value)
                });
                console.log(this.$refs.second.$data.value)
            }
        }
    })
</script>
...
```

5.6 事件总线

在 Vue.js 中实现组件之间数据的传递，大概有以下几种方式：

(1) 通过 props 将父组件的数据传递给子组件。

(2) 通过 $emit 方法将子组件中的数据，通过绑定的函数传递给父组件。

(3) 在父组件中定义方法，通过@(v-bind)自定义方法='自定义函数名称'传递给子组件，在子组件中可以直接调用。

(4) 通过 Vue.js 组件实例的 $parent、$root、$children、$refs 等属性，获取相关的组件对象。

还可以使用事件总线的方式和 Vuex 的 state 的属性实现任意组件之间的数据共享。接下来介绍 Vue.js 中的事件总线，并用它实现组件之间的数据共享传递。

事件总线又称为 EventBus。在 Vue.js 中可以使用 EventBus 来作为沟通的桥梁，就像是所有组件共用相同的事件中心，可以向该中心注册发送事件或接收事件，所以组件都可以上下平行地通知其他组件，但由于太方便，所以若使用不慎，就会造成难以维护的灾难(所以由 Vuex 作为管理状态中心，将通知的概念提升为状态共享)。

使用事件总线，可以分三步实现。

1. 注册事件总线

创建一个全局的 Vue.js 对象，用来充当事件总线中心，以及用来传递事件。该事件总

线只能用在当前页面的组件中，也可以将这个 Vue.js 对象保存到 Vue.js 的原型属性中，从而注册一个全局的事件总线中心，代码如下：

```
//创建事件总线
const eventBus = new Vue()
//或者注册全局事件中心
//Vue.prototype.$bus = eventBus
```

2. 接收事件总线

定义一个组件，使用 $on(事件名称，回调函数)方法，给指定的总线事件绑定一个函数，监听事件总线中的事件，代码如下：

```
//第 5 章/事件总线.html
…
<!-- 第 1 个组件模板，显示 age -->
<template id="view1">
    <div>view1 age:{{age}}</div>
</template>

<script type="text/JavaScript">
    //创建 MyView1 对象
    const MyView1 = {
        data() {
            return {
                age: 1
            }
        },

        mounted() {
            //接收 bus 总线中的 bindEvent 事件
            //this.$bus.$on('bindEvent', user => {
            eventBus.$on('bindEvent', user => {
                this.age = user.age + this.age
            })
        },
        template: '#view1'
}
…
</script>
…
```

3. 发送事件总线

在组件代码中，用 $emit(事件名称，参数)方法，向事件总线中心发送一个事件，并传递参数内容，代码如下：

```
//第 5 章/事件总线.html
…
< script type = "text/JavaScript">
    …
    const vm = new Vue({
        …
        methods: {
            publishEvent() {
                this.value++
                //往 bus 总线发布 bindEvent 事件(全局)
                //this. $bus. $emit('bindEvent', {name:'zhangsan', age:this.value})
                //局部
                eventBus. $emit('bindEvent', {name:'zhangsan', age:this.value})
            }
        }
    })
</script >
…
```

整合后的代码如下：

```
//第 5 章/事件总线.html
…
< div id = "app">
    < my - view1 ></my - view1 >
    < my - view2 ></my - view2 >

    < button v - on:click = 'publishEvent'>测试</button >
</div >

<! -- 第 1 个组件模板,显示 age -- >
< template id = "view1">
    < div > view1 age:{{age}}</div >
</template >

<! -- 第 2 个组件目标,显示 age -- >
< template id = "view2">
    < div > view2 age:{{age}}</div >
</template >

< script type = "text/JavaScript">
    //创建 MyView1 对象
    const MyView1 = {
        data() {
            return {
                age: 1
```

```
        }
    },
    mounted() {
        //接收 bus 总线中的 bindEvent 事件
        //this.$bus.$on('bindEvent', user => {
        eventBus.$on('bindEvent', user => {
            this.age = user.age + this.age
        })
    },
    template: '#view1'
}

//创建 MyView2 对象
const MyView2 = {
    data() {
        return {
            age: 10
        }
    },
    mounted() {
        //接收 bus 总线中的 bindEvent 事件
        //this.$bus.$on('bindEvent', user => {
        // this.age = this.age + user.age
        //})
        //this.$bus.$once('bindEvent', user => {
        // this.age = this.age + user.age
        //})
        eventBus.$on('bindEvent', user => {
            this.age = this.age + user.age
        })
    },
    template: '#view2'
}

//事件总线
//Vue.prototype.$bus = new Vue()
const eventBus = new Vue()

const vm = new Vue({
    el:'#app',
    data: {
        value: 1
    },
    components: {
        MyView1,
        MyView2
```

```
        },
        methods: {
            publishEvent() {
                this.value++
                //往 bus 总线发布 bindEvent 事件
                //this. $bus. $emit('bindEvent', {name:'zhangsan', age:this.value})
                eventBus. $emit('bindEvent', {name:'zhangsan', age:this.value})
            }
        }
    })
</script>
…
```

5.7 插槽

项目中有很多组件显示的内容,在不同的地方使用的时候,有些内容需要变化,这时候可以用插槽内容分发功能实现。例如开发人员可以定义一个组件,专门用来显示提示信息。只是显示的信息包含在插槽 slot 元素中,满足在不同的地方,提示的内容不一样。MessageAlert 组件中包含了 slot 插槽,运行的时候,就会将组件元素包含的内容动态地渲染到视图中,代码如下:

```
//第 5 章/简单插槽.html
…
<div id = "app">
    <message - alert >提醒</message - alert >
    <message - alert >警告</message - alert >
    <message - alert >错误</message - alert >
</div>
<script>
    const MessageAlert = {
            template:` <div>
                <strong> msg:</strong>
                <slot></slot>
                </div>`
    }
    Vue.component("MessageAlert",MessageAlert;
    const vm = new Vue({
            el:"#app"
    })
</script>
…
```

5.7.1 插槽的缺省内容和编译作用域

在实战中，slot 中往往希望有缺省值，实现起来很简单，直接在子元素的 slot 之间包含缺省值即可，调用的时候，如果有新值，则自动会将缺省值覆盖，如< slot >缺省值</slot >。

当然，分发到 slot 中的内容，可以包含动态内容，此时动态的变量只能使用父组件里面的 property，代码如下：

```
//第 5 章/插槽缺省值和编译作用域.html
…
<div id = "app">
    <! -- 使用缺省值 -- >
    < slot - dft - content ></slot - dft - content >
    <! -- 使用静态内容 -- >

    < slot - dft - content >提交</slot - dft - content >
    <! -- 编译作用域 -- >

    < slot - dft - content >{{parentValue}}</slot - dft - content >
    <! -- 放开会报异常，sonValue 未定义

    < slot - dft - content >{{sonValue}}</slot - dft - content >
     -- >

</div >
< script >
    Vue.component("SlotDftContent",{
        template:` < button type = 'submit'>
                        < slot > submit </slot >
                </button >`,
        data:function){
                return {
                                sonValue:'sonContent'
                }
        }
    })
    const vm = new Vue({
        el:"#app",
        data:{
            parentValue:'parent'
        }
    })
</script >
…
```

5.7.2 具名插槽

有时候在一个子元素中需要定义多个插槽，使用子元素的时候，将不同内容分发到不同插槽中去。这时候，开发人员可以使用 slot 的 name 属性，给每个 slot 命名(没有命名的为 default)，使用子组件时，用 template 元素包含要分发到每个 slot 的内容，使用“v-slot：目标插槽的名称”来指定对应的分发插槽，代码如下：

```
//第 5 章/具名插槽.html
…
<div id = "app">
    <base - layout >
        <template v - slot:header >头</template >
        <template v - slot:footer >尾</template >
        <p >缺省</p >
        <p >内容</p >
    </base - layout >
</div >
<script >
    Vue.component("BaseLayout",{
        template:`<div class = "container">
            <header >
                <slot name = "header"></slot >
            </header >
            <main >
                <slot ></slot >
            </main >
            <footer >
                <slot name = "footer"></slot >
            </footer >
            </div >`
    })
    const vm = new Vue({
        el:"#app"
    })
</script >
…
```

5.7.3 作用域插槽

有时候，需要获取子元素的数据，进行整理后再分发给插槽，这时候可以使用作用域插槽。开发人员可以在 slot 元素中，使用 v-bind 指令将对象绑定后，传递到组件外面，如：<slot v-bind：userAttr="user" >...</slot >就是将子元素中的 user 数据，通过 v-bind 指令绑定到 userAttr 属性上。userAttr 属性，我们称它为 slot property。在父组件中，可以在

template 元素中使用指定的插槽属性变量操作 userAttr 绑定的 user 数据，代码如下：

```
<div id = "app">
    <! -- v - slot:default 指定分发给 default slot -- >
    <range - slot v - slot:default = "slotProps">
        {{slotProps.userAttr.userName + "整理好了"}}
    </range - slot >
</div >
<script >
    //作用域插槽
    Vue.component("RangeSlot",{
            template:`<div >
                        <slot v - bind:userAttr = "user">
                            {{user.name}}
                        </slot >
                      </div >`,
                        data:function){
                            return {
                                    user:{
                                        name:'李四',
                                        userName:'lisi'
                                    }
                            }
                        }
    })
    const vm = new Vue({
        el:"#app"
    })
</script >
```

5.7.4 动态插槽名

动态指令参数也可以用在 v-slot 上，用来定义动态的插槽名。特别要注意的是，这里不支持驼峰命名的变量名，语法如下：

```
<range - slot >
   <template v - slot:[动态的插槽名称]>
      ...
   </template >
</range - slot >
```

5.7.5 具名插槽的缩写

跟 v-on 和 v-bind 一样，v-slot 也有缩写，即把参数之前的所有内容（v-slot：）替换为字符 # 。v-slot：header 可以被重写为 #header，代码如下：

```
<base-layout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

然而，和其他指令一样，该缩写只在有参数的时候才可用，这意味着以下语法是无效的：

```
<!-- 这样会触发一个警告 -->
<current-user #="{ user }">
  {{ user.firstName }}
</current-user>
```

如果开发人员希望使用缩写，则必须始终以明确的插槽名取而代之，代码如下：

```
<current-user #default="{ user }">
  {{ user.firstName }}
</current-user>
```

5.8 动态组件和异步组件

Vue.js 提供了两种特殊组件：动态组件和异步组件。

5.8.1 动态组件

有时候，一个组件内部需要根据不同的选择，显示不同的组件。如在标签卡布局里面，单击不同的标签，内容块需要显示该标签对应的内容。Vue.js 中提供了 component 元素组件，可以动态地显示指定的组件。

在 component 元素中有个 is 属性，通过给 is 属性绑定一个组件名称，component 就可以显示该组件名称对应的组件，语法如下：

```
<component v-bind:is="currTabComponent"></component>
```

如果 currTabComponent 的值是 test-comp1，就表示显示名称为 test-comp1 的组件。样例代码如下：

```
//第 5 章/动态组件.html
…
<div id = "app">
    <! -- 显示标签 Tabs -- >
    <button
        v - for = "tab in tabs"
        v - bind:key = "tab"
        v - bind:class = "['tab - button', { active: currentTab === tab }]"
        v - on:click = "currentTab = tab"
        >
        {{ tab }}
        </button>
    <! -- 显示 tab 对应的组件 -- >
        <! -- <keep - alive> -- >
        <component v - bind:is = "currentTabComponent" class = "tab"></component>
        <! -- </keep - alive> -- >
</div>
<script>
    //<! -- 定义 Home 组件 -- >
    Vue.component("dync - comp0",{
        template:`<div>Component Home</div>`
    };
    //<! -- 定义组件 1 -- >
    Vue.component("dync - comp1",{
        template:`<div>Component 111</div>`
    };
    //<! -- 定义组件 2 -- >
    Vue.component("dync - comp2",{
        template:`<div>Component 222</div>`
    })
    const vm = new Vue({
        el:"#app",
        data:{
            currentTab:'Comp1',
            tabs:["Comp0","Comp1","Comp2"]
        },
        computed:{
            currentTabComponent:function(){
                //返回 tab 对应组件的名称,以便在 component 中动态显示
                return 'dync - ' + this.currentTab.toLocaleLowerCase;
            }
        }
    })
</script>
…
```

5.8.2 异步组件

在大型应用中，开发人员可能需要将应用分割成小的代码块，并且只在需要的时候才从服务器加载所需的模块。为了简化，Vue.js 允许开发人员以一个工厂函数的方式定义组件，这个工厂函数会异步解析组件的定义。Vue.js 只有在这个组件需要被渲染的时候才会被触发，并且会把结果缓存起来供未来重渲染，代码如下：

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    //在此定义组件
    resolve({
      template: `
        <div>
          我是异步加载的哦
        </div>
      `
    })
  }, 1000);
});
```

这个工厂函数会收到一个 resolve 回调，这个回调函数会在从服务器得到组件定义的时候被调用。开发人员也可以调用 reject(reason)来表示加载失败。这里的 setTimeout 是为了演示用的，如何获取组件取决于开发人员自己。

5.8.3 keep-alive

在前面的动态组件案例中，用户单击 tab 切换组件后，回到以前 tab 对应的组件上时，该组件会被重新渲染。有时候希望回到上次单击的状态，这时候，开发人员可以在 component 的外面包含 keep-alive 元素，这样就可以保持这些组件的状态，以避免反复重渲染导致的性能问题，代码如下：

```
<!-- 失活的组件将会被缓存! -->
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
```

5.9 处理组件边界问题

接下来介绍 Vue.js 中处理与边界情况有关的功能，即需要对 Vue.js 的规则做一些小调整的特殊情况。不过应注意这些功能都是有好有坏的，开发人员需要根据实际情况斟酌选择。

5.9.1 访问元素的 & 组件

在绝大多数情况下，开发人员最好不要触达另一个组件实例内部或手动操作 DOM 元素。不过在有些情况下，需要直接操作组件实例的内部数据和手动操作 DOM 元素，Vue.js 为 Vue.js 实例对象提供了很多 & 组件，开发人员用这些方式可以直接操作组件内部的其他组件。

1. 访问根实例

在每个 Vue.js 实例中都定义了 $root 属性，开发人员可以使用 $root 获取整个组件树的根组件，从而操作根组件实例的内容。创建一个根组件对象，代码如下：

```
//Vue.js 根实例
new Vue({
  data: {
    foo:1
  },
  computed: {
    bar:function () { /* ... */ }
  },
  methods: {
    baz:function () { /* ... */ }
  }
})
```

所有的子组件都可以将这个实例作为一个全局 store 访问或使用，代码如下：

```
//获取根组件的数据
this.$root.foo

//写入根组件的数据
this.$root.foo = 2

//访问根组件的计算属性
this.$root.bar

//调用根组件的方法
this.$root.baz()
```

注意：对于 demo 或非常小型的有少量组件的应用来讲这个模式是很方便的，不过将这个模式扩展到中大型应用就不然了。因此在绝大多数情况下，我们强烈推荐使用 Vuex 来管理应用的状态。

2. 访问父级组件实例

和 $root 类似，$parent property 可以用来从一个子组件访问父组件的实例。它提供了一种机制，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。

在绝大多数情况下，触达父级组件会使应用更难调试和理解，尤其是隔了一段时间后，很难找出哪个变更是从哪里发起的。

<google-map>组件可以定义一个 map property，所有的子组件都需要访问它。在这种情况下<google-map-markers>可以通过类似 this.$parent.getMap 的方式访问那个地图，以便为其添加一组标记，代码如下：

```
<google-map>
  <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
</google-map>
```

但是，通过这种模式构建出来的组件的内部仍然容易出现问题。例如，在<google-map>和<google-map-markers>之间添加一个新的 <google-map-region> 组件时，代码如下：

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

在 <google-map-markers> 内部就需要编写一些类似的代码：

```
var map = this.$parent.map || this.$parent.$parent.map
```

这样它很快就会失控，变得难以理解和追踪。

3. 访问子组件实例或子元素

尽管存在 prop 和事件，有的时候开发人员仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，可以通过 ref 的 attribute 为子组件赋予一个 ID 引用，代码如下：

```
<base-input ref="usernameInput"></base-input>
```

这样在 JavaScript 代码中，就可以用以下方式访问 <base-input> 实例，代码如下：

```
this.$refs.usernameInput
```

当然，也可以使用一个类似的 ref 提供对内部这个指定元素的访问，示例代码如下：

```
<input ref="input">
```

甚至可以通过其父级组件定义方法，代码如下：

```
methods: {
  //用来从父级组件聚焦输入框
  focus:function() {
```

```
      this.$refs.input.focus()
    }
  }
```

这样就允许父级组件通过下面的代码聚焦 < base-input > 里的输入框，代码如下：

```
this.$refs.usernameInput.focus()
```

当 ref 和 v-for 一起使用的时候，开发人员得到的 ref 将会是一个包含了对应数据源的子组件的数组。

注意：$refs 只会在组件渲染完成之后生效，并且不是响应式的。它仅作为一个用于直接操作子组件的“逃生舱”——开发人员应该避免在模板或计算属性中访问 $refs。

4. 依赖注入

前面介绍的 $parent 属性，可以为开发人员提供一种直接引用父组件的方式，但是在更深层级的嵌套组件上使用的时候，很难跟踪，特别是在中间插入新的组件的时候。Vue.js 提供了依赖注入的机制，可以很好地解决这个问题。

所谓依赖注入，就是在父组件中，可以使用 provide 选项指定想要提供给后代子组件使用的数据和方法，然后在任何后代子组件中都可以使用 inject 选项来接收想要添加到当前实例上的属性。

在< google-map >组件中使用 provide 选项，暴露 getMap 方法，代码如下：

```
provide:function () {
  return {
    getMap: this.getMap                    //暴露 getMap 方法
  }
}
```

在< google-map >的任何后代组件中，使用 inject 选项添加到自己的属性中，代码如下：

```
inject: ['getMap']
```

本质上，依赖注入是一部分大范围有效的 prop 属性，以前说的 props 属性只能将数据传递给直接子组件，而依赖注入则可以将数据传递给任何后代子组件。满足以下两个条件的数据传递，可以使用依赖注入方式实现。

（1）祖先组件不需要知道哪些后代组件使用它提供的 property。

（2）后代组件不需要知道被注入的 property 来自哪里。

然而，依赖注入还是有负面影响的。它将应用程序中的组件与它们当前的组织方式耦合起来，使重构变得更加困难。同时所提供的 property 是非响应式的。这是出于设计的考虑，因为使用它们来创建一个中心化规模化的数据和使用 $root 做这件事都是不够好的。如果开发人员想要共享的这个 property 是应用特有的，而不是通用化的，或者如果想在祖

先组件中更新所提供的数据，开发人员则可以使用像 Vuex 这样真正的状态管理方案实现。

5.9.2 程序化的事件侦听

在 Vue.js 中，可以使用 $emit 触发一个事件，v-on 指令可以在侦听到事件后，执行绑定在事件上的函数。另外，Vue.js 还提供了以下几种方式侦听 $emit 发送的事件。

(1) 通过 $on(eventName,eventHandler)侦听一个事件。

(2) 通过 $once(eventName, eventHandler)一次性侦听一个事件。

(3) 通过 $off(eventName,eventHandler)停止侦听一个事件。

项目中一般不会用到这些，但是当项目需要在一个组件实例上手动侦听事件时，这些就可以派上用场。在 EventComponent 组件的 mounted 回调函数中，使用 $on 的方式给自己添加了一个侦听 myclick 事件的函数，当单击 EventComponent 组件的 button 按钮时，在执行的事件代码中使用 $emit 方式触发 myclick 事件，从而实现 count 属性的递增，代码如下：

```
//第 5 章/程序化侦听.html
…
<div id = 'app'>
    <event - component ></event - component >
</div >
<template id = "eventComponent">
    <div >
    count: {{count}}< br/>
    <button v - on:click = "toClick">单击</button >
    </div >
</template >
<script type = 'text/JavaScript'>
    const EventComponent = {
        template:'#eventComponent',
        data(){
            return {
                count: 0
            }
        },
        mounted:function(){
            this.$on('myclick', function(step){
                this.count += step
            })
        },
        methods:{
            toClick(){
                this.$emit('myclick', 2)
            }
```

```
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            EventComponent
        },
    })
</script>
…
```

5.9.3 循环引用组件

Vue.js 中的组件支持自己引用自己的循环引用和自己同其他组件之间的相互循环引用。

1. 递归循环引用

Vue.js 中的组件可以自己引用自己。因为组件在使用之前，需要进行注册。递归引用自己的组件，一样支持局部注册和全局注册。全局注册和使用一个递归组件的样例，代码如下：

```
//第 5 章/递归组件.html
…
<div id = 'app'>
    <recursion - component :count = "count"></recursion - component>
</div>
<template id = "recursionComponent">
    <span> &gt;
        <span v - if = "count > 1">
            <!—递归引用自己 -->
            <recursion - component :count = "count - 1"></recursion - component>
        </span>
    </span>
</template>
<script type = 'text/JavaScript'>
    const RecursionComponent = {
        name:'RecursionComponent',
        template:'#recursionComponent',
        props:{
            count:{
                type:Number
            }
        },
        mounted:function(){
```

```
            console.log(this.count)
        }
}
//全局注册
    Vue.component("RecursionComponent", RecursionComponent)
    const vm = new Vue({
        el: '#app',
        data: {
            count: 5
        },
        methods: {
        }
    })
</script>
…
```

因为是全局注册，所以在任何组件里面，包括自己都可以使用。同样，递归组件支持局部注册，只是需要注意的是，在组件中注册自己是通过 name 选项完成的。RecursionComponent 组件在内部注册了自己，代码如下：

```
const RecursionComponent = {
        name:'RecursionComponent',
        …
}
```

在递归引用组件的时候，一定要用 v-if 等手段，避免无限循环，否则会抛出无限循环错误。

2. 组件之间循环引用

以下代码，定义了两个组件，即 TreeNode 和 TreeNodeChild。TreeNode 用来显示当前节点的名称，如果包含子节点，就循环 TreeNodeChild 显示子节点。TreeNodeChild 组件用来显示一个新的 TreeNode，这样就形成了组件之间的循环调用。

需要注意的是，组件之间的循环引用，要注意条件判断，避免出现死循环。另外，因为组件之间要相互引用，引用前要相互注册，而目前注册的前提是被注册的组件要先被初始化才能被注册，这样就会造成 TreeNode 和 TreeNodeChild 组件注册的悖论，所以暂时循环引用的组件，只适用于全局注册，代码如下：

```
//第 5 章/循环组件 tree.html
…
<style type="text/css">
    *{
        margin: 0;
```

```
            padding: 0;
        }
        li{
            list-style: none;
        }
        ul li{
            margin-left: 20px;
        }
    </style>
    <div id='app'>
        <tree-node :node="treeData"></tree-node>
    </div>
    <!-- 显示树节点内容 -->
    <template id="treeNode">
        <!-- 如果有子节点 -->
        <ul v-if="node.children">
            <!-- 显示自己的名称 -->
            <span>{{node.name}}</span>
            <!-- 便利列出子节点 -->
            <li v-for="(child,index) in node.children" :key="index">
                <tree-node-child :node="child"></tree-node-child>
            </li>
        </ul>
        <!-- 如果没有子节点 -->
        <span v-else>{{node.name}}</span>
    </template>
    <!-- 显示子节点 -->
    <template id="treeNodeChild">
        <tree-node :node="node"></tree-node>
    </template>

    <script type='text/JavaScript'>
        const TreeNode = {
            name: 'TreeNode',
            template: '#treeNode',
            props:{
                node: {
                    type: Object
                }
            }
        }
        const TreeNodeChild = {
            name: 'TreeNodeChild',
            template: '#treeNodeChild',
            props: {
                node: {
```

```
            type: Object
        }
    }
}

Vue.component('TreeNode', TreeNode)
Vue.component('TreeNodeChild', TreeNodeChild)

const vm = new Vue({
    el: '#app',
    data: {
        treeData:{
            name: 'TP2012 班',
            children: [
                {
                    name: '组 1',
                    children: [
                        {
                            name: '1 组员 1'
                        },
                        {
                            name: '1 组员 2'
                        },
                        {
                            name: '1 组员 3'
                        },
                        {
                            name: '1 组员 4'
                        }
                    ]
                },
                {
                    name: '组 2',
                    children: [
                        {
                            name: '2 组员 1'
                        },
                        {
                            name: '2 组员 2'
                        },
                        {
                            name: '2 组员 3'
                        },
                        {
                            name: '2 组员 4'
                        }
```

```
                        ]
                    }
                    ,{
                        name: '组 3',
                        children: [
                            {
                                name: '3 组员 1'
                            },
                            {
                                name: '3 组员 2'
                            },
                            {
                                name: '3 组员 3'
                            },
                            {
                                name: '3 组员 4'
                            }
                        ]
                    }
                ]
            }
        },
        methods: {
        }
    })
</script>
…
```

5.9.4 其他模板

在 Vue.js 中,除了可以使用 template 定义组件模板外,还支持内联模板和 X-Template 模板。

1. 内联模板

在使用组件的时候,在组件中添加 inline-template 特殊属性,就是告知 Vue.js,使用组件包含的内容作为模板。使用 inline-template 标记,并且使用< inner-template-component > 元素中包含的内容作为模板,代码如下:

```
//第 5 章/其他模板.html
…
<div id = 'app'>
    <inner-template-component inline-template>
        <div>
            <span v-html = "desc"></span>:{{count}}
```

```
        </div>
    </inner-template-component>
</div>
<script type='text/JavaScript'>

    const InnerTemplateComponent = {
        name: 'InnerTemplateComponent',
        data(){
            return {
                count: 1,
                desc: '内联模板组件案例'
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            InnerTemplateComponent
        },
        data: {
        },
        methods: {
        }
    })
</script>
...
```

在 Vue.js 所示的 DOM 中定义内联模板，使模板的撰写工作更加灵活，但是 inline-template 会让模板的作用域变得更加难以理解，所以作为最佳实践，应在组件内优先选择 template 选项或.vue 文件里的一个 template 元素来定义模板。

2. X-Template 模板

X-Template 模板定义的方式是在一个 script 元素中，为其带上 text/x-template 的类型，然后通过一个 id 将模板引用过去，代码如下：

```
//第5章/其他模板.html
...
<div id='app'>
    <xtemplate-component></xtemplate-component>
</div>
<script type='text/JavaScript'>
    const XtemplateComponent = {
        name: 'XTemplateComponent',
        template: '#XTemplateComponent',
        data(){
```

```
            return {
                count: 100,
                desc: 'XTemplate 模板组件案例'
            }
        }
    }
    const vm = new Vue({
        el: '#app',
        components:{
            XtemplateComponent
        },
        data: {
        },
        methods: {
        }
    })
</script>
...
```

X-Template 模板需要定义在 Vue.js 所属的 DOM 元素外，可以用于模板特别大的 demo 或极小型的应用，但是其他情况下应避免使用，因为这会将模板和该组件的其他定义分离开。

5.9.5 控制组件的更新

Vue.js 作为一个响应式系统，它自动实现更新，不过还有特殊情况，如开发人员希望能强制更新，又或者如开发人员希望阻止不必要的更新。Vue.js 也提供了对应的支持方式。

1. 强制更新

开发人员可以通过调用 Vue.js 实例的 $forceUpdate 方法，迫使 Vue.js 实例重新渲染。注意它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

需要强制更新的情况比较少，绝大部分是开发人员的编码失误，例如没有留意数组或对象的变更检测事项，或者依赖了未被 Vue.js 响应式系统跟踪的状态。

2. 降低更新

渲染普通的 HTML 元素在 Vue.js 中是非常快速的，但有的时候可能有一个组件，这个组件包含了大量静态内容。在这种情况下，可以在根元素上添加 v-once attribute 以确保这些内容只计算一次，然后缓存起来，代码如下：

```
Vue.component('terms-of-service', {
  template: `
    <div v-once>
      <h1>Terms of Service</h1>
      ... a lot of static content ...
```

```
        </div>
    `
})
```

注意：不要过度使用这个模式。当需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非当前功能非常明显渲染变慢了，不然它完全是没有必要的——再加上它在后期会带来很多困惑。例如，设想另一个开发者并不熟悉 v-once 或漏看了它在模板中，他们可能会花很多个小时去找出模板为什么无法正确更新。