第 5 章

组　　件

组件(Component)是 Vue 最核心的功能部分,也是整个框架设计最精彩的地方,当然也是比较难掌握的。每个开发者都想在软件开发过程中使用之前写好的代码,但又担心引入这段代码会对现有的程序产生影响。Web Components 的出现提供了一种新的思路,可以自定义 tag 标签,并拥有自身的模板、样式和交互。

5.1　什么是组件

在正式介绍组件前,先看一个 Vue 组件的简单示例感受一下,代码如下:

```
// 定义一个名为 button-counter 的新组件
Vue.component('button-counter', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'
})
```

组件是可复用的 Vue 实例,且带有一个名字,在这个示例中组件名字是<button-counter>。可以在一个通过 new Vue 创建的 Vue 根实例中,把这个组件作为自定义标签来使用,代码如下:

```
<div id="components-demo">
<button-counter></button-counter>
<button-counter></button-counter>
</div>
new Vue({ el: '#components-demo' })
```

完整示例代码如下:

```
//第 5 章/自定义组件.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="components-demo">
    <button-counter></button-counter>
    <button-counter></button-counter>
</div>
<script>
    //定义一个名为 button-counter 的新组件
    Vue.component('button-counter', {
        data: function () {
            return {
                count: 0
            }
        },
        template: '<button v-on:click="count++">You clicked me {{ count }} times.
          </button>'
    })
    new Vue({ el: '#components-demo' })
</script>
</body>
</html>
```

这些类似于<button-counter>,平时工作中没见过的标签就是组件,每个标签代表一个组件,这样就可以将组件进行任意次数的复用。

Web 的组件其实就是页面组成的一部分,好比是计算机中的每一个硬件(如硬盘、键盘、鼠标等),它具有独立的逻辑和功能或界面,同时又能根据规定的接口规则进行相互融合,从而变成一个完整的应用。

Web 页面就是由一个个类似这样的部分组成的,例如导航、列表、弹窗、下拉菜单等。页面只不过是这些组件的容器,组件自由组合形成功能完整的界面,当不需要某个组件或者想要替换某个组件时,可以随时进行替换和删除,而不影响整个应用的运行。

前端组件化的核心思路就是将一个巨大且复杂的页面分成粒度合理的页面组成部分。

使用组件的好处:

(1) 提高开发效率。

(2) 方便重复使用。

(3) 简化调试步骤。

(4) 提升整个项目的可维护性。

(5) 便于协同开发。

组件是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素,封装可重用的代码。在较高层面上,组件是自定义元素,Vue.js 的编译器为它添加特殊功能。在有些情况下,组件也可以以原生 HTML 元素的形式存在,以 is 特性扩展。

组件系统让我们可以用独立可复用的小组件构建大型应用,几乎任意类型应用的界面可以抽象为一个组件树,如图 5-1 所示。

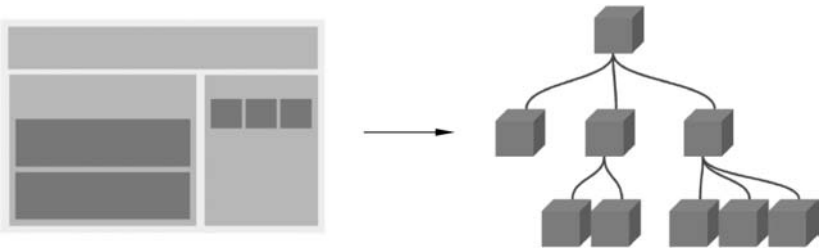

图 5-1 Vue 组件树

5.2 组件的基本使用

为了能在模板中使用,这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型:全局注册和局部注册。

5.2.1 全局注册

全局注册需要确保在根实例初始化之前注册,这样才能使组件在任意实例中被使用。全局注册有 3 种方式。

要注册一个全局组件,我们可以使用 Vue.component(tagName,options),代码如下:

```
Vue.component('my-component', {
  //选项
})
```

my-component 就是注册组件的自定义标签名称,推荐使用小写字母加“-”分隔符的形式命名。组件在注册之后,便可以在父实例的模块中以自定义元素的形式使用,示例代码如下:

```
<div id="app">
    <my-component></my-component>
</div>
<script>
```

```
    Vue.component('my-component', {
        template: '<h1>注册</h1>'
    });
    var vm = new Vue({
        el:'#app'
    })
</script>
```

注意：

(1) template 的 DOM 结构必须被一个而且是唯一根元素所包含，直接引用而不被“ ”包裹是无法被渲染的。

(2) 模板(template)声明了数据和最终展现给用户的 DOM 之间的映射关系。

除了 template 选项外，组件中还可以像 Vue 实例那样使用其他的选项(5.3 节将详细讲解)，例如 data、computed、methods 等，但是在使用 data 时和实例不同，data 必须是函数，然后将数据利用 return 返回，代码如下：

```
<div id="app">
    <my-component></my-component>
</div>
<script>
    Vue.component('my-component', {
        template: '<h1>{{message}}</h1>',
        data:function () {
            return{
                message:'注册'
            }
        }
    });
    var vm = new Vue({
        el:'#app'
    })
</script>
```

Vue 组件中 data 值不能为对象，因为对象是引用类型，组件可能会被多个实例同时引用。如果 data 值为对象，将导致多个实例共享一个对象，其中一个组件改变 data 属性值，其他实例也会受影响。

上面解释了 data 不能为对象的原因，这里简单地讲一下 data 为函数的原因。data 为函数，通过 return 返回对象的复制，致使每个实例都有自己独立的对象，实例之间可以互不影响地改变 data 属性值。

使用 Vue.extend 配合 Vue.component 方法，代码如下：

```
<div id = "app">
    <my - list></my - list>
</div>
<script>
    var list = Vue.extend({
        template:'<h1>this is a list</h1>',
    });
    Vue.component("my - list",list);
    //根实例
    new Vue({
            el:"#app",
        })
</script>
```

将模板字符串定义到 script 标签中,代码如下:

```
<script id = "tmpl" type = "text/x - template">
    <div><a href = "#">登录</a> | <a href = "#">注册</a></div>
</script>
```

同时,需要使用 Vue.component 定义组件,代码如下:

```
Vue.component('account', {
      template: '#tmpl'
    });
```

完整示例代码如下:

```
//第 5 章/全局注册.html
<div id = "app">
    <account></account>
</div>
<template id = "tmpl">
    <div><a href = "#">登录</a> | <a href = "#">注册</a></div>
</template>
<script>
    Vue.component('account', {
        template: '#tmpl'
    });
    new Vue({
        el:"#app",
    })
</script>
```

5.2.2 局部注册

如果不需要全局注册或者只让组件使用在其他组件内，可以采用选项对象的components属性实现局部注册，示例代码如下：

```
//第5章/局部注册.html
<div id="app">
    <account></account>
</div>
<script>
    //创建 Vue 实例,得到 ViewModel
    var vm = new Vue({
        el: '#app',
        data: {},
        methods: {},
        components: {              //定义子组件
            account: {             //account 组件
                template: '<div><h1>这是 Account 组
                    </h1><login></login></div>', //在这里使用定义的子组件
                components: {      //定义子组件的子组件
                    login: {       //login 组件
                        template: "<h3>这是登录组件</h3>"
                    }
                }
            }
        }
    });
</script>
```

可以使用 flag 标识符结合 v-if 和 v-else 切换组件，如例 5-1 所示。

【例 5-1】 使用标识符切换组件

```
//第5章/使用标识符切换组件.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <input type="button" value="toggle" @click="flag=!flag">
    <account v-if="flag"></account>
```

```
    <login v-else="flag"></login>
</div>
<script>
    //创建 Vue 实例,得到 ViewModel
    var vm = new Vue({
        el: '#app',
        data: {
            flag: true
        },
        methods: {},
        components: {      //定义子组件
            account: {     //account 组件
                template: '<div><h1>这是 Account 组件</h1></div>',
                                                        // 在这里使用定义的子组件
            },
            login: {       //login 组件
                template: "<h3>这是登录组件</h3>"
            }
        }
    });
</script>
</body>
</html>
```

5.2.3 DOM 模板解析说明

当使用 DOM 作为模版时(例如,将 el 选项挂载到一个已存在的元素上),会受到 HTML 的一些限制,因为 Vue 只有在浏览器解析和标准化 HTML 后才能获取模板内容。尤其像这些元素 、<ol>、<table>、<select> 限制了能被它包裹的元素,而一些像 <option> 这样的元素只能出现在某些其他元素内部。

在自定义组件中使用这些受限制的元素时会导致一些问题,例如:

```
<table>
    <my-row>...</my-row>
</table>
```

自定义组件有时被认为是无效的内容,因此在渲染的时候会导致错误。这是因为使用特殊的 is 属性来挂载组件,代码如下:

```
<table>
    <tr is="my-row"></tr>
</table>
```

也就是说，在标准 HTML 中，一些元素只能放置特定的子元素，而另一些元素只能存在于特定的父元素中。例如 table 中不能放置 div，tr 的父元素不能为 div 等。所以，当使用自定义标签时，标签名还是那些标签的名字，但是可以在标签的 is 属性中填写自定义组件的名字，如例 5-2 所示。

【例 5-2】 DOM 模板解析示例

```
//第 5 章/DOM 模板解析.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <table border="1" cellpadding="5" cellspacing="0">
        <my-row></my-row>
        <tr is="my-row"></tr>
    </table>
</div>
<script type="text/javascript">
    new Vue({
        el:'#app',
        components:{
            myRow:{
                template:'<tr><td>123456</td></tr>'
            }
        }
    });
</script>
</body>
</html>
```

示例执行效果如图 5-2 所示。

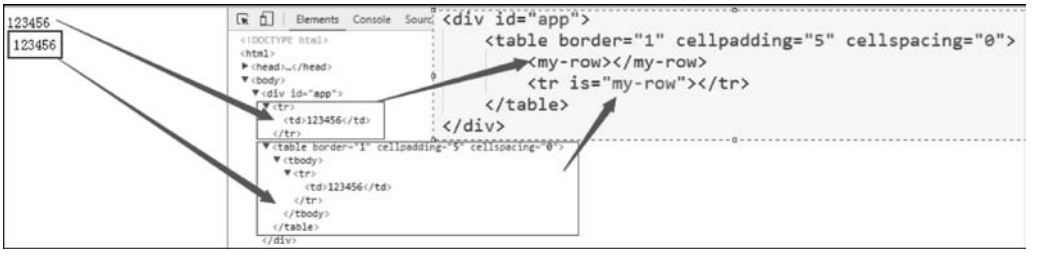

图 5-2 DOM 模板解析效果图

从图中不难发现直接引用< my-row >组件标签并没有被< table >标签包裹，而用 is 特殊属性挂载的组件可以达到所需效果。

注意：如果使用的是字符串模板，则不受限制。

5.3 组件选项

Vue 的组件可以理解为预先定义好行为的 ViewModel 类。一个组件可以预先定义很多选项。但是最核心的有以下几个。

(1) 模板(template)：模板声明了数据和最终展现给用户的 DOM 之间的映射关系。

(2) 初始数据(data)：一个组件的初始数据状态。对于可复用的组件来说，通常是私有的状态。

(3) 接收的外部参数(props)：组件之间通过参数来进行数据的传递和共享。参数默认是单向绑定(由上至下)，但也可以显式声明为双向绑定。

(4) 方法(methods)：对数据的改动操作一般都在组件的方法内进行。可以通过 v-on 指令将用户输入事件和组件方法进行绑定。

(5) 生命周期钩子函数(lifecycle hooks)：一个组件会触发多个生命周期钩子函数，例如 created、attached、destroyed 等。在这些钩子函数中，我们可以封装一些自定义的逻辑。和传统的 MVC 相比，这可以理解为 Controller 的逻辑被分散到了这些钩子函数中。

组件接收的选项大部分与 Vue 实例一样，相同的部分就不再赘述了。我们重点讲解一下二者不同的选项 data 和 props，data 在 5.2.1 节中已经讲解过了，所以本节主要讲解 props，它用于接收父组件传递的参数。

5.3.1 组件 props

上述我们在使用组件时主要是把组件模板的内容进行复用，代码如下：

```
//父组件
<div id = "app">
    <my-component></my-component>
</div>
<script>
    //子组件
    Vue.component('my-component', {
        template: '<h1>注册</h1>'
    });
    var vm = new Vue({
        el:'#app'
    })
</script>
```

但是组件中更重要的功能是组件间进行通信，选项 props 是组件中非常重要的一个选项，起到父子组件间桥梁的作用。

1. 静态 props

组件实例的作用域是孤立的。这意味着不能（也不应该）在子组件的模板内直接引用父组件的数据，代码如下：

```
<div id = "app">
    <my - componet message = "来自父组件的数据!"></my - componet >
</div >
<script type = "text/javascript">
    Vue.component('my - componet', {
        template: '<span >{{ message }}</span >'
    })
    new Vue({
        el:'#app'
    });
</script >
```

子组件接收不到父组件 message 的数据，而且会报错，如图 5-3 所示。

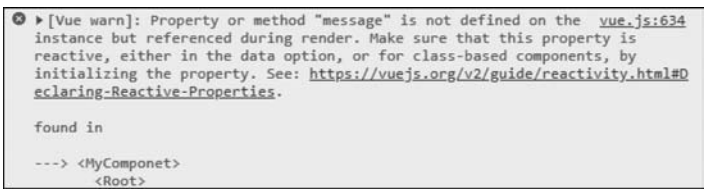

图 5-3　直接引用父组件 message 后的错误信息

要想让子组件使用父组件的数据，需要通过子组件的 props 选项实现。子组件要显式地用 props 选项声明它期望获得的数据，如例 5-3 所示。

【例 5-3】 子组件使用父组件的数据

```
//第 5 章/子组件使用父组件的数据.html
//部分代码省略
<div id = "app">
    <my - componet message = "来自父组件的数据!"></my - componet >
</div >
<script type = "text/javascript">
    Vue.component('my - componet', {
        //声明 props
        props: ['message'],
        //就像 data 一样，props 可以用在模板内
        //同样也可以在 vm 实例中像 "this.message" 这样使用
```

```
        template: '<span>{{ message }}</span>'
    })
    new Vue({
        el:'#app'
    });
</script>
```

页面显示结果为：来自父组件的数据！

由于 HTML 的特性不区分大小写，所以，当使用的不是字符串模板时，camelCased（驼峰式）命名的 props 需要转换为相对应的 kebab-case（短横线隔开式）命名，代码如下：

```
<div id="app">
    <my-componet my-message="来自父组件的数据!"></my-componet>
</div>
<script type="text/javascript">
    Vue.component('my-componet',{
        props: ['myMessage'],
        template: '<span>{{ myMessage }}</span>'
    })
    new Vue({
        el:'#app'
    });
</script>
```

2. 动态 props

在模板中，有时候传递的数据并不一定是固定不变的，而是要动态地绑定父组件的数据到子模板的 props，与绑定到任何普通的 HTML 特性相类似，采用 v-bind 进行绑定。当父组件的数据变化时，该变化也会传递给子组件，如例 5-4 所示。

【例 5-4】 动态传递父组件数据到子组件

```
//第5章/动态 props.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <input type="text" v-model="parentMessage">
    <my-componet :message="parentMessage"></my-componet>
</div>
```

```
<script type="text/javascript">
    Vue.component('my-componet', {
        props: ['message'],
        template: '<span>{{ message }}</span>'
    })
    new Vue({
        el:'#app',
        data:{
            parentMessage:''
        }
    });
</script>
</body>
</html>
```

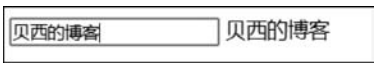

图 5-4　动态接收父组件数据效果图

页面显示效果如图 5-4 所示。

这里使用 v-model 绑定了父组件数据 parentMessage，当在输入框中输入数据时，子组件接收到的 props 'message' 也会实时响应，并更新组件模板。

对于初学者常犯的一个错误来讲，如果在父组件中直接传递数字、布尔值、数组、对象，则它所传递的值均为字符串。如果想传递一个实际的值，则需要使用 v-bind，从而让它的值被当作 JavaScript 表达式进行计算，代码如下：

```
<div id="app">
    <my-componet message="1 + 1"></my-componet><br>
    <my-componet :message="1 + 1"></my-componet>
</div>
<script type="text/javascript">
    Vue.component('my-componet', {
        props: ['message'],
        template: '<span>{{ message }}</span>'
    })
    new Vue({
        el:'#app'
    });
</script>
```

如果不使用 v-bind 指令，则页面显示结果是字符串“1＋1”。如果使用 v-bind 指令，则会当作 JavaScript 表达式进行计算，其计算结果是数字 2。

5.3.2　props 验证

前面我们介绍的 props 选项的值都是数组，除了数组，还可以是对象。可以为组件的

props 指定验证规则，如果传入的数据不符合规则，则 Vue 会发出警告。当 props 需要验证时，需要采用对象写法。

当组件给其他人使用时，推荐进行数据验证。以下是一些 props 的示例代码：

```
Vue.component('example', {
        props: {
            //基础类型检测,null 的意思是任何类型都可以
            propA: Number,
            //多种类型
            propB: [String, Number],
            //必传且是字符串
            propC: {
                type: String,
              required: true
            },
            //数字,有默认值
            propD: {
                type: Number,
                default: 100
            },
            //数组/对象的默认值应当由一个工厂函数返回
            propE: {
                type: Object,
                default: function () {
                    return { message: 'hello' }
                }
            },
            //自定义验证函数
            propF: {
                validator: function (value) {
                    return value > 10
                }
            }
    }
    })
```

验证的 type 类型可以是 String、Number、Boolean、Function、Object、Array 等，type 也可以是一个自定义构造器函数，使用 instanceof 检测。

当 props 验证失败时，Vue 会抛出警告（如果使用的是开发版本）。props 会在组件实例创建之前进行校验，所以在 default 或 validator 函数里，诸如 data、computed 或 methods 等实例属性还无法使用。

【例 5-5】 如果传入子组件的 message 不是数字，则抛出警告

```
//第 5 章/props 验证.html
```

```
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="example">
    <parent></parent>
</div>
<script>
    var childNode = {
        template: '<div>{{message}}</div>',
        props:{
            'message':Number
        }
    }
    var parentNode = {
            template:'<div class="parent"><child :message="msg"></child></div>',
            components: {
                'child': childNode
            },
            data(){
            return{
                msg: '123'
            }
    }
    };
    new Vue({     //创建根实例
        el: '#example',
        components: {
            'parent': parentNode
        }
    })
</script>
</body>
</html>
```

传入数字 123 时，无警告提示。但当传入字符串“123”时，控制台会发出警告，如图 5-5 所示。

5.3.3 单向数据流

所有的 props 都使得其父子 props 之间形成了一个单向下行绑定：父级 props 的更新

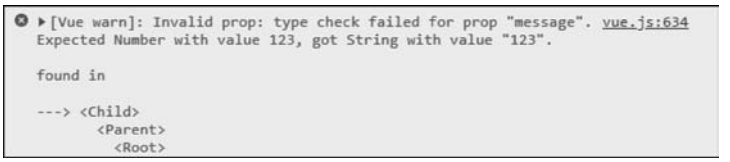

图 5-5 props 验证失败时的警告信息

会向下流动到子组件中,但是反过来则不行。之所以这样设计,是尽可能将父子组件解耦,避免子组件无意中修改了父组件的状态。

额外地,每次父级组件发生变更时,子组件中所有的 props 都将刷新为最新的值。这意味着不应该在一个子组件内部改变 props。如果这样做了,Vue 会在浏览器的控制台中发出警告,如例 5-6 所示。

【例 5-6】 单向数据传递

```
//第 5 章/单向数据传递.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="example">
    <parent></parent>
</div>
<script>
    var childNode = {
            template:
        '<div class="child"><div><span>子组件数据</span>' +
              '<input v-model="childMsg"></div><p>{{childMsg}}</p></div>',
        props:['childMsg']
    }
    var parentNode = {
            template:
        '<div class="parent"><div><span>父组件数据</span>' +
            '<input v-model="msg"></div><p>{{msg}}</p><child :child-msg="msg">
        </child></div>',
        components: {
            'child': childNode
        },
        data(){
        return {
```

```
            'msg':'match'
        }
    }
    };
    new Vue({     //创建根实例
        el: '#example',
        components: {
            'parent': parentNode
        }
    })
</script>
</body>
</html>
```

页面显示效果如图 5-6 所示。

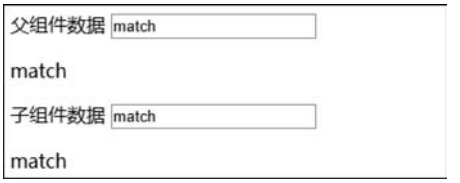

图 5-6　单向数据传递页面效果

父组件数据变化时，子组件数据会相应变化，而子组件数据变化时，父组件数据保持不变，并在控制台显示警告，如图 5-7 所示。

业务中经常会遇到需要修改 props 中的数据的情况，通常有以下两种原因：

(1) props 作为初始值传入后，子组件想把它当作局部数据来使用。

(2) props 作为初始值传入，由子组件处理成其他数据并输出。

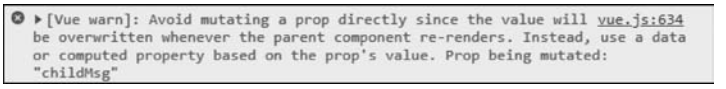

图 5-7　修改子组件时的警告信息

注意：JS 中对象和数组是引用类型，指向同一个内存空间，如果 props 是一个对象或数组，在子组件内部改变它会影响父组件的状态。

对于这两种情况，正确的应对方式是：

(1) 定义一个局部变量，并用 props 的值初始化它，代码如下：

```
props: ['initialCounter'],
data: function () {
  return { counter: this.initialCounter }
}
```

但是，定义的局部变量 counter 只能接收 initialCounter 的初始值，当父组件要传递的值发生变化时，counter 无法接收到最新值，示例代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<div id="example">
    <parent></parent>
</div>
<script>
    var childNode = {
            template:
        '<div class="child"><div><span>子组件数据</span>' +
                '<input v-model="temp"></div><p>{{temp}}</p></div>',
        props:['childMsg'],
          data(){
                return{
                    temp:this.childMsg
                }
            },
            };
    var parentNode = {
            template:
        '<div class="parent"><div><span>父组件数据</span><input v-model="msg"> ' +
              '</div><p>{{msg}}</p><child :child-msg="msg"></child></div>',
          components: {
              'child': childNode
          },
          data(){
                return {
                    'msg':'match'
                }
            }
            };
    new Vue({    //创建根实例
            el: '#example',
          components: {
            'parent': parentNode
           }
     })
</script>
</body>
</html>
```

在示例中，除初始值外，父组件的值无法更新到子组件中。

（2）定义一个计算属性，处理 props 的值并返回，代码如下：

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

但是，由于是计算属性，因此只能显示值，不能设置值，示例代码如下：

```
<script>
    var childNode = {
            template:
        '<div class="child"><div><span>子组件数据</span>' +
                '<input v-model="temp"></div><p>{{temp}}</p></div>',
          props:['childMsg'],
          computed:{
              temp(){
                  return this.childMsg
              }
          },
    };
    var parentNode = {
            template:'<div class="parent"><div><span>父组件数据</span> ' +
                '<input v-model="msg"></div><p>{{msg}}</p>' +
                '<child :child-msg="msg"></child></div>',
        components: {
            'child': childNode
        },
  data(){
        return {
            'msg':'match'
        }
    }
    };
   new Vue({     //创建根实例
        el: '#example',
        components: {
           'parent': parentNode
          }
    })
</script>
```

示例中，由于子组件使用的是计算属性，所以子组件的数据无法手动修改。

（3）更加妥帖的方案是，使用变量储存 props 的初始值，并使用 watch 观察 props 的值的变化。当 props 的值发生变化时，更新变量的值，代码如下：

```
<div id = "example">
    <parent></parent>
</div>
<script>
    var childNode = {
            template:
        '<div class = "child"><div><span>子组件数据</span>' +
                '<input v-model = "temp"></div><p>{{temp}}</p></div>',
          props:['childMsg'],
          data(){
                return{
                    temp:this.childMsg
                }
            },
          watch:{
              childMsg(){
                  this.temp = this.childMsg
              }
          }
    };
    var parentNode = {
            template:
        '<div class = "parent"><div><span>父组件数据</span>' +
                '<input v-model = "msg"></div><p>{{msg}}</p>' +
                '<child :child-msg = "msg"></child></div>',
          components: {
              'child': childNode
          },
          data(){
                return {
                    'msg':'match'
                }
            }
    };
    new Vue({    //创建根实例
        el: '#example',
          components: {
              'parent': parentNode
          }
    })
</script>
```

5.4 组件通信

在 Vue 组件通信中其中最常见的通信方式就是父子组件之中的通信，而父子组件的设定方式在不同情况下又各不相同，归纳起来，组件之间通信如图 5-8 所示。最常见的就是父组件为控制组件而子组件为视图组件。父组件传递数据给子组件使用，遇到业务逻辑操作时子组件触发父组件的自定义事件。

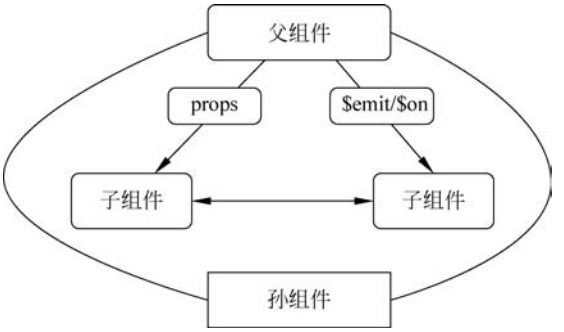

图 5-8 组件通信

作为一个 Vue 初学者不得不了解的就是组件间的数据通信(暂且不谈 Vuex，后面章节会讲到)。通信方式根据组件之间的关系有不同之处。组件关系有下面 3 种：父→子，子→父，非父子。

我们已经知道，父组件向子组件通信是通过 props 传递数据的，就好像方法的传参一样，父组件调用子组件并传入数据，子组件接收父组件传递的数据进行验证后使用。

props 可以是数组也可以是对象，用于接收来自父组件的数据。props 可以是简单的数组或者使用对象作为替代，对象允许配置高级选项，如类型检测、自定义校验和设置默认值。

5.4.1 自定义事件

当子组件需要向父组件传递数据时，就要用到自定义事件。v-on 指令除了监听 DOM 事件外，还可以用于组件之间的自定义事件。

在子组件中用 $emit()来触发事件以便将内部的数据传递给父组件，如例 5-7 所示。

【例 5-7】 子组件向父组件传递数据

```
//第 5 章/子组件向父组件传递数据.html
<!DOCTYPE html>
<html>
<head>
<title></title>
```

```
<meta charset = "utf - 8"/>
<script src = "https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id = "app">
    <my - component v - on:myclick = "onClick"></my - component>
</div>
<script>
    Vue.component('my - component', {
        template:'<div>' +
            '<button type = "button" @click = "childClick">单击触发自定义事件</button>
            </div>',
        methods: {
            childClick () {
                this.$emit('myclick', '这是我传递出去的数据', '这是我传递出去的数据 2')
            }
        }
    })
    new Vue({
        el: '#app',
        methods: {
            onClick () {
                console.log(arguments)
            }
        }
    })
</script>
</body>
</html>
```

解析上面示例中代码的执行步骤：

(1) 子组件在自己的方法中将自定义事件及需要传递的数据通过以下代码传递出去：

```
this.$emit('myclick','这是我传递出去的数据', '这是我传递出去的数据 2')
```

- 第一个参数是自定义事件的名字。
- 后面的参数是依次想要传递出去的数据。

(2) 父组件利用 v-on 为事件绑定处理器，代码如下：

```
<my - component v - on:myclick = "onClick"></my - component>
```

这样，在 Vue 实例的 methods 方法中就可以调用传进来的参数了。

5.4.2 $emit/$on

这种方法通过一个空的 Vue 实例作为中央事件中心，用它来触发事件和监听事件，巧妙而轻量地实现了任何组件间的通信，包括父子、兄弟、跨级。当我们的项目比较大时，可以选择更好的状态管理解决方案 Vuex。

实现方式如下：

```
var Event = new Vue();
    Event.$emit(事件名,数据);
    Event.$on(事件名,data =>{});
```

假设兄弟组件有 3 个，分别是 A、B、C 组件，C 组件如何获取 A 或者 B 组件的数据，如例 5-8 所示。

【例 5-8】 组件之间的通信

```
//第5章/$emit-$on.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <my-a></my-a>
    <my-b></my-b>
    <my-c></my-c>
</div>
<template id="a">
    <div>
        <h3>A组件:{{name}}</h3>
        <button @click="send">将数据发送给C组件</button>
    </div>
</template>
<template id="b">
    <div>
        <h3>B组件:{{age}}</h3>
        <button @click="send">将数组发送给C组件</button>
    </div>
</template>
<template id="c">
    <div>
```

```
        <h3>C 组件: {{name}},{{age}}</h3>
    </div>
</template>
<script>
    var Event = new Vue();        //定义一个空的 Vue 实例
    var A = {
        template: '#a',
        data() {
        return {
            name: 'beixi'
        }
    },
        methods: {
            send() {
                Event.$emit('data-a', this.name);
            }
        }
    }
    var B = {
        template: '#b',
        data() {
        return {
            age: 18
        }
    },
        methods: {
            send() {
                Event.$emit('data-b', this.age);
            }
        }
    }
    var C = {
        template: '#c',
        data() {
        return {
            name: '',
                age: ""
    }
    },
        mounted() {               //在模板编译完成后执行
        Event.$on('data-a',name => {
            this.name = name;     //箭头函数内部不会产生新的 this,此处如果不用=>,
                                  //则 this 指代 Event
        })
        Event.$on('data-b',age => {
```

```
                this.age = age;
            })
        }
        }
        var vm = new Vue({
            el: '#app',
            components: {
                'my-a': A,
                'my-b': B,
                'my-c': C
            }
        });
    </script>
    </body>
    </html>
```

页面显示效果如图 5-9 所示。

图 5-9　组件间通信效果图

$on 监听了自定义事件 data-a 和 data-b，因为有时不确定何时会触发事件，一般会在 mounted 或 created 钩子中进行监听。

5.5　内容分发

在实际项目开发中，时常会把父组件的内容与子组件自己的模板混合起来使用。而这样的一个过程在 Vue 中被称为内容分发，也常常被称为 slot（插槽）。其主要参照了当前 Web Components 规范草案，使用特殊的< slot >元素作为原始内容的插槽。

5.5.1　基础用法

由于 slot 是一块模板，因此对于任何一个组件，从模板种类的角度来分，其实都可分为非插槽模板和插槽模板。其中非插槽模板指的是 HTML 模板（也就是 HTML 的一些元素，例如 div、span 等构成的元素），其显示与否及怎样显示完全由插件自身控制，但插槽模

板(也就是 slot)是一个空壳子,它显示与否及怎样显示完全由父组件来控制。不过,插槽显示的位置由子组件自身决定,slot 写在组件 template 的哪部分,父组件传过来的模板将来就显示在哪部分。

一般定义子组件的代码如下:

```
<div id = "app">
    <child>
        <span>123456</span>
    </child>
</div>
  <script>
    new Vue({
        el:'#app',
        components:{
            child:{
                template:"<div>这是子组件内容</div>"
                }
         }
        });
</script>
```

页面显示结果:这是子组件内容。123456内容并不会显示。

注意:虽然标签被子组件的 child 标签所包含,但由于它不在子组件的 template 属性中,因此不属于子组件。

在 template 中添加<slot></slot>标签,代码如下:

```
<div id = "app">
      <child>
          <span>123456</span>
      </child>
</div>
<script>
    new Vue({
        el:'#app',
        components:{
            child:{
                template:"<div><slot></slot>这是子组件内容</div>"
            }
        }
    });
</script>
```

页面显示结果:123456 这是子组件内容。

我们分步解析一下内容分发,现在我们看一个架空的例子,帮助理解刚刚讲过的严谨而

难懂的定义。假设有一个组件名为 my-component，其使用上下文代码如下：

```
<my-component>
    <p>hi,slots</p>
</my-component>
```

再假设此组件的模板为：

```
<div>
    <slot></slot>
<div>
```

那么注入后的组件 HTML 相当于：

```
<div>
    <p>hi,slots</p>
<div>
```

标签<slot>会把组件使用上下文的内容注入此标签所占据的位置。组件分发的概念简单而强大，因为它意味着对一个隔离的组件除了通过属性、事件交互之外，还可以注入内容。

将此案例变成可以执行的代码，代码如下：

```
//部分代码省略
<div class="" id="app">
    <my-component>
        <p>hi,slots</p>
    </my-component>
</div>
<script>
    Vue.component('my-component', {
            template:'<div><slot></slot><div>'
    });
    new Vue({
        el: "#app"
});
</script>
```

一个组件如果需要外部传入简单数据，如数字、字符串等，可以使用 property；如果需要传入 js 表达式或者对象，可以使用事件；如果希望传入的是 HTML 标签，那么使用内容分发就再好不过了。所以，尽管内容分发这个概念听起来极为复杂，而实际上可以简化为把 HTML 标签传入组件的一种方法。所以归根结底，内容分发是一种为组件传递参数的方法。

5.5.2 编译作用域

在深入了解分发 API 之前，先明确内容在哪个作用域里编译。假定模板为：

```
<child-component>
  {{ message }}
</child-component>
```

这里的 message 应该绑定父组件的数据，还是绑定子组件的数据呢？答案是 message 就是一个 slot，但它绑定的是父组件的数据，而不是组件<child-component>的数据。

组件作用域简单地来说就是：父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译，如例 5-9 所示。

【例 5-9】 组件作用域

```
//第5章/组件作用域.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
        <child-component v-show="someChildProperty"></child-component>
</div>
<script>
    Vue.component('child-component', {
        template: '<div>这是子组件内容</div>',
        data: function () {
            return {
                someChildProperty: true
            }
        }
    })
    new Vue({
        el:'#app'
    })
</script>
</body>
</html>
```

这里 someChildProperty 绑定的是父组件的数据，所以是无效的，因此获取不到数据。如果想在子组件上绑定，可以采用如下代码：

```
<div id = "app">
    <child - component ></child - component >
</div >
<script >
    Vue.component('child - component', {
        // 有效,因为是在正确的作用域内
        template: '<div v - show = "someChildProperty">这是子组件内容</div >',
        data: function () {
            return {
                someChildProperty: true
            }
        }
    })
    new Vue({
        el:'#app'
    })
</script >
```

因此,slot 分发的内容是在父作用域内进行编译的。

5.5.3 默认 slot

如果要使父组件在子组件中插入内容,必须在子组件中声明 slot 标签,如果子组件模板不包含< slot >插口,父组件的内容将会被丢弃,如例 5-10 所示。

【例 5-10】 默认 slot

```
//第 5 章/默认 slot.html
<!DOCTYPE html >
<html >
<head >
<title ></title >
<meta charset = "utf - 8"/>
<script src = "https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script >
</head >
<body >
<div id = "app">
    <!-- 1.2 组件 innerHTML 位置以后不管有什么代码,都会被放进插槽里面去 -->
    <index >
        <span >首页</span >
        <span >首页</span >
        <span >首页</span >
        <h1 >手机</h1 >
    </index >
</div >
```

```
<script>
    //插槽的作用就是将组件外部获取的代码片段放到组件内部
    /*定义默认插槽,通过 slot 组件定义,定义好了之后,就相当于一个插槽,你可以把它理解为
计算机的 UBS 插口*/
    Vue.component('index', {
        template:'<div>index</div>'
    })
    var vm = new Vue({
        el: '#app',
    })
</script>
</body>
</html>
```

页面显示结果为：index。所有子组件中的内容都不会被显示,而是被丢弃。要使父组件在子组件中插入内容,必须在子组件中声明 slot 标签,示例代码如下：

```
<script>
    Vue.component('index', {
        template:'<div><slot></slot>index</div>'
    })
    var vm = new Vue({
        el: '#app',
    })
</script>
```

5.5.4 具名 slot

slot 元素可以用一个特殊的属性 name 来配置如何分发内容。多个 slot 标签可以有不同的名字,如例 5-11 所示。

使用方法如下：

(1) 父组件要在分发的标签中添加属性"slot=name 名"。

(2) 子组件在对应分发位置上的 slot 标签中添加属性"name=name 名"。

【例 5-11】 多个 slot 应用

```
//第 5 章/多个 slot 应用.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

```
</head>
<body>
    <div id="app">
            <child>
                <span slot="one">123456</span>
                <span slot="two">abcdef</span>
            </child>
        </div>
    <script>
    new Vue({
        el:'#app',
        components:{
            child:{
                template:"<div><slot name='two'></slot>我是子组件<slot name='one'>
                </slot></div>"
            }
        }
    });
</script>
</body>
</html>
```

页面显示结果为：abcdef 我是子组件 123456。

总结：slot 分发其实就是父组件在子组件内放一些 DOM，它负责这些 DOM 是否显示，以及在哪个地方显示。

5.5.5 作用域插槽

插槽分为单个插槽、具名插槽和作用域插槽，前两种比较简单并且在前面已将讲解过，本节的重点是讨论作用域插槽。

简单来说，前两种插槽的内容和样式皆由父组件决定，也就是说显示什么内容和怎样显示都由父组件决定，而作用域插槽的样式仍由父组件决定，但内容由子组件控制。即前两种插槽不能绑定数据，而作用域插槽是一个带绑定数据的插槽。

作用域插槽更具代表性的应用是列表组件，允许组件自定义如何渲染列表的每一项，代码如下：

```
<div id="app">
    <child></child>
</div>
<script>
    Vue.component('child', {
        data(){
```

```
            return {
                list:[1,2,3,4]
            }
            },
            template: '<div><ul>' +
                '<li v-for="item of list">{{item}}</li></ul></div>',
        })
        var vm = new Vue({
            el: '#app'
        })
    </script>
```

在上面示例代码中，如果需要 child 组件在很多地方被调用，那么我们希望在不同的地方调用 child 的组件时，明确这个列表到底怎么循环。列表的样式不是由 child 组件控制的，而是由外部 child 模板占位符告诉我们组件的每一项该如何渲染，也就是说这里不应采用 li 标签，而是应该采用 slot 标签，示例代码如下：

```
    <div id="app">
        <child>
            <template slot-scope="props"><!-- 固定写法,属性值可以自定义 -->
                <li>{{props.item}}</li><!-- 插值表达式可以直接使用 -->
            </template>
        </child>
    </div>
    <script>
        Vue.component('child', {
            data(){
                return {
                    list:[1,2,3,4]
                }
            },
            template: '<div><ul>' +
                '<slot v-for="item of list" :item=item></slot></ul></div>',
        })
        var vm = new Vue({
            el: '#app'
        })
    </script>
```

<slot v-for="item of list" :item=item></slot>这段代码的意思是 child 组件去实现一个列表的循环，但是列表项中的每一项怎样显示，并不用关心，具体怎样显示，由外部来决定。

<template slot-scope="props"></template>是一个固定写法，属性值可以自定义。

它的意思是当子组件用 slot 时，会向子组件传递一个 item，子组件接收的数据都放在 props 上。

什么时候使用作用域插槽呢？当子组件循环或某一部分的 DOM 结构应该由外部传递进来时，我们要用作用域插槽。使用作用域插槽，子组件可以向父组件的作用域插槽传递数据，父组件如果想接收这个数据，必须在外层使用 template 模版占位符，同时通过 slot-scope 对应的属性名字，接收传递过来的数据。如上面代码，传递一个 item 过来，在父组件的作用域插槽里就可以接收到这个 item，然后就可以使用它了。

5.6 动态组件

让多个组件使用同一个挂载点，并动态切换，这就是动态组件。通过使用保留的 <component>元素，动态地绑定它的 is 特性，可以实现动态组件。它的应用场景往往应用在路由控制或者 tab 切换中。

5.6.1 基本用法

我们通过一个切换页面的例子来说明动态组件的基本用法，如例 5-12 所示。

【例 5-12】 切换页面

```
//第 5 章/切换页面.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <button @click="change">切换页面</button>
    <component :is="currentView"></component>
</div>
<script>
    new Vue({
        el: '#app',
        data:{
            index:0,
            arr:[
                {template:'<div>我是主页</div>'},
                {template:'<div>我是提交页</div>'},
                {template:'<div>我是存档页</div>'}
            ],
```

```
        },
            computed:{
                currentView(){
                    return this.arr[this.index];
                }
            },
            methods:{
                change(){
                    this.index = (++this.index) % 3;
                }
            }
        })
    </script>
    </body>
    </html>
```

component 标签中 is 属性决定了当前采用的子组件,;is 是 v-bind 的简写,绑定了父组件中 data 的 currentView 属性。单击按钮时,会更改数组 arr 的索引值,同时也修改了子组件的内容。

5.6.2 keep-alive

动态切换掉的组件(非当前显示的组件)被移除掉了,如果把切换出去的组件保留在内存中,可以保留它的状态或避免重新渲染。< keep-alive >包裹动态组件时,会缓存不活动的组件实例,而不是销毁它们,以便提高提取效率。和< transition >相似,< keep-alive >是一个抽象组件,它自身不会渲染一个 DOM 元素,也不会出现在父组件链中,如例 5-13 所示。

【例 5-13】 keep-alive 基础用法

```
//第 5 章/ keep - alive 基础用法.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset = "utf - 8"/>
<script src = "https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id = "app">
    <button @click = "change">切换页面</button>
    <keep - alive>
        <component :is = "currentView"></component>
    </keep - alive>
</div>
```

```
<script>
    new Vue({
        el: '#app',
        data:{
            index:0,
            arr:[
                {template:'<div>我是主页</div>'},
                {template:'<div>我是提交页</div>'},
                {template:'<div>我是存档页</div>'}
            ],
        },
        computed:{
            currentView(){
                return this.arr[this.index];
            }
        },
        methods:{
            change(){
                /*ES6 新增了 let 命令,用来声明变量。它的用法类似于 var,但是所声明的变量只在 let 命令所在的代码块内有效。*/
                let len = this.arr.length;
                this.index = (++this.index) % len;
            }
        }
        })
</script>
</body>
</html>
```

如果有多个条件性的子元素,<keep-alive>要求同时只有一个子元素被渲染时可以使用条件语句进行判断,如例 5-14 所示。

【例 5-14】 利用条件判断缓冲子元素

```
//第 5 章/利用条件判断缓冲子元素.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <button @click="change">切换页面</button>
```

```
        <keep-alive>
            <home v-if="index===0"></home>
            <posts v-else-if="index===1"></posts>
            <archive v-else></archive>
        </keep-alive>
    </div>
    <script>
        new Vue({
            el: '#app',
            components:{
                home:{template:'<div>我是主页</div>'},
                    posts:{template:'<div>我是提交页</div>'},
                        archive:{template:'<div>我是存档页</div>'},
                        },
            data:{
                index:0,
            },
            methods:{
                change(){
                    //在 data 外面定义的属性和方法通过 $options 可以获取和调用
                    let len = Object.keys(this.$options.components).length;
                    this.index = (++this.index) % len;
                }
            }
        })
    </script>
    </body>
    </html>
```

5.6.3 activated 钩子函数

Vue 给组件提供了 activated 钩子函数，作用于动态组件切换或者静态组件初始化的过程中。activated 是和 template、data 等属性平级的一个属性，其形式是一个函数，函数里默认有一个参数，而这个参数是一个函数，执行这个函数时，才会切换组件，即可延迟执行当前的组件，如例 5-15 所示。

【例 5-15】 activated 钩子函数

```
//第5章/activated钩子函数.html
<!DOCTYPE html>
<html>
<head>
<title></title>
<meta charset="utf-8"/>
```

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <button @click='toShow'>单击显示子组件</button>
    <!---- 或者采用<component v-bind:is="which_to_show" keep-alive></component>也行 ----->
    <keep-alive>
        <component v-bind:is="which_to_show"></component>
    </keep-alive>
</div>
<script>
    var vm = new Vue({                    //创建根实例
        el: '#app',
        data: {
            which_to_show: "first"
        },
        methods: {
            toShow: function () {    //切换组件显示
                var arr = ["first", "second", "third", ""];
                var index = arr.indexOf(this.which_to_show);
                if (index < 2) {
                    this.which_to_show = arr[index + 1];
                } else {
                    this.which_to_show = arr[0];
                }
                console.log(this.$children);
            }
        },
        components: {
            first: {                      //第1个子组件
                template: "<div>这里是子组件1</div>"
            },
            second: {                     //第2个子组件
                template: "<div>这里是子组件2,这里是延迟后的内容:{{hello}}</div>",
                data: function () {
                    return {
                        hello: ""
                    }
                },
                activated: function (done) {                    //执行这个参数时,才会切换组件
                    console.log('beixi')
                    var self = this;
                    var startTime = new Date().getTime(); //获得当前时间
                    //2s后执行
                    while (new Date().getTime() < startTime + 2000){
```

```
                        self.hello = '我是延迟后的内容';
                    }
                }
            },
            third: { //第 3 个子组件
                template: "<div>这里是子组件 3</div>"
            }
        }
    });
</script>
</body>
</html>
```

当切换到第 2 个组件的时候，会先执行 activated 钩子函数，会在 2s 后显示组件 2，起到了延迟加载的作用。

5.6.4 异步组件

在大型应用中，我们可能需要将应用分割成小一些的代码块，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 允许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所定义的组件。Vue 只有在这个组件需要被渲染的时候才会触发该工厂函数，且会把结果缓存起来供未来重新渲染，代码如下：

```
<div id = "app">
    <async - example></async - example>
</div>
<script>
    Vue.component('async - example', function (resolve, reject) {
        setTimeout(function () {
            //向 resolve 回调传递组件定义
            resolve({
                template: '<div>这是异步渲染的内容!</div>'
            })
        }, 1000)
    })
    new Vue({
        el:'#app'
    })
</script>
```

由此所见，这个工厂函数会收到一个 resolve 回调，这个回调函数会在服务器得到组件定义的时候被调用。也可以调用 reject(reason) 来表示加载失败。这里的 setTimeout 是为了演示异步，如何获取组件取决于自己。例如把组件配置成一个对象，通过 Ajax 进行请

求，然后调用 reslove 传入配置选项。

5.6.5 ref 和 $ refs

在 Vue 中一般很少直接操作 DOM，但不可避免有时候确定需要用到，这时我们可以通过 ref 和 $ refs 来实现：

（1）ref：ref 被用来给元素或子组件注册引用信息，引用信息将会注册在父组件的 $ refs 对象上，如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素，如果在子组件上，引用指向的就是组件的实例。

（2）$ refs：$ refs 是一个对象，持有已注册过 ref 的所有子组件。

1. 普通获取 DOM 的方式

我们先通过 getElementById 方法获取，代码如下：

```
<div id = "app">
    <input type = "button" value = "获取 h3 的值" @click = "getElement">
    <h3 id = "myh3">我是 h3 </h3 >
</div >
<script >
    var vm = new Vue({
        el:"#app",
        data:{},
        methods:{
            getElement(){
                //通过 getElementById 方式获取 DOM 对象
                console.log(document.getElementById("myh3").innerHTML);
            }
        }
    })
</script >
```

2. ref 使用

接下来我们通过 ref 属性来获取，代码如下：

```
<div id = "app">
    <input type = "button" value = "获取 h3 的值" @click = "getElement">
    <h3 id = "myh3" ref = "myh3">我是 h3 </h3 >
</div >
```

然后在控制台查看 vm 实例对象，如图 5-10 所示。

通过上面的示例我们发现，在 vm 实例上有一个 $ refs 属性，而且该属性拥有我们通过 ref 注册的 DOM 对象，于是我们可以这样获取 DOM 对象，代码如下：

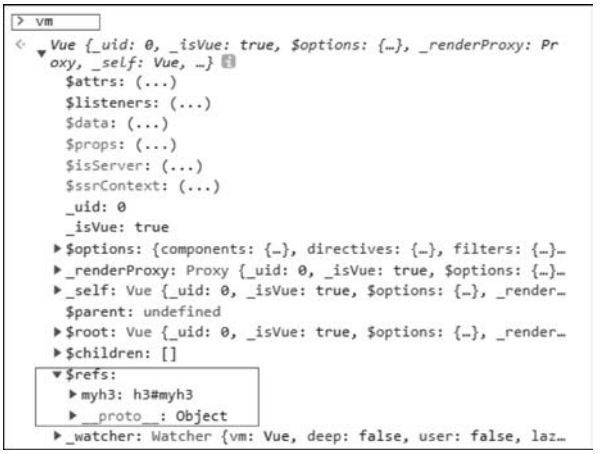

图 5-10 vm 实例对象信息

```
<script>
    var vm = new Vue({
        el:"#app",
        data:{},
        methods:{
            getElement(){
                console.log(this.$refs.myh3.innerHTML);
            }
        }
    })
</script>
```

3. ref 在组件中使用

在子组件中使用 ref 属性，会将子组件添加到父组件的 $refs 对象中，代码如下：

```
<div id="app">
    <input type="button" value="获取 h3 的值" @click="getElement">
    <h3 id="myh3" ref="myh3">我是 h3</h3>
    <hr>
    <login ref="mylogin"></login>
</div>
```

在控制台输入 vm，查看 vm 对象，如图 5-11 所示。

通过 vm 实例查看，发现 $refs 中已绑定 mylogin 组件，而且还看到了对应组件中的 msg 属性和 show 方法，这样我们便可以调用它们了，代码如下：

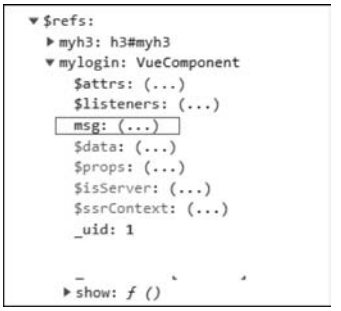

图 5-11　vm 实例对象信息

```
var vm = new Vue({
    el:"#app",
    data:{},
    methods:{
        getElement(){
            //通过 getElementById 方式获取 DOM 对象
            //console.log(this.$refs.myh3.innerHTML);
            console.log(this.$refs.mylogin.msg);
            this.$refs.mylogin.show();
        }
    },
        components:{
            login
        }
})
```

完整实例代码如例 5-16 所示。

【例 5-16】 ref 在组件中的应用

```
//第 5 章/ref 在组件中的应用.html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
<!-- 引入 vue -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
    <input type="button" value="获取 h3 的值" @click="getElement">
    <h3 id="myh3" ref="myh3">我是 h3</h3>
```

```
    <hr>
    <login ref="mylogin"></login>
</div>
<script>
    var login = {
        template: "<h3>我是 login 子组件</h3>",
        data(){
        return {
            msg: "ok"
        }
    },
        methods:{
            show(){
                console.log("show 方法执行了...")
            }
        }
    }
    var vm = new Vue({
        el:"#app",
        data:{},
        methods:{
            getElement(){
                //通过 getElementById 方式获取 DOM 对象
                //console.log(this.$refs.myh3.innerHTML);
                console.log(this.$refs.mylogin.msg);
                this.$refs.mylogin.show();
            }
        },
     components:{
         login
     }
    })
</script>
</body>
</html>
```

5.7 综合案例

添加组件后，我们通过单击“发表评论”按钮来添加内容并传递到评论列表中，如例 5-17 所示。实现的逻辑是：

(1) 通过单击“发表评论”按钮触发单击事件并调用组件中 methods 所定义的方法。

(2) 在 methods 所定义的方法中加载并保存 localStorage 的列表数据到 list 中。

(3) 将录入的信息添加到 list 中，然后将数据保存到 localStorage 中。

（4）调用父组件中的方法来刷新列表数据。

【例 5-17】 综合案例

```
//第 5 章/综合案例.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <!-- 引入 vue -->
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <!-- 引入 bootstrap -->
    <link rel="stylesheet" href="./lib/bootstrap-3.3.7.css">
</head>
<body>
<div id="app">
    <cmt-box @func="loadComments"></cmt-box>
    <ul class="list-group">
      <li class="list-group-item" v-for="item in list" :key="item.id">
        <span class="badge">评论人：{{ item.user }}</span>
            {{ item.content }}
    </li>
    </ul>
</div>
<template id="tmpl">
    <div>
        <div class="form-group">
            <label>评论人：</label>
            <input type="text" class="form-control" v-model="user">
        </div>
        <div class="form-group">
            <label>评论内容：</label>
            <textarea class="form-control" v-model="content"></textarea>
        </div>
        <div class="form-group">
            <input type="button" value="发表评论" class="btn btn-primary" @click=
"postComment">
        </div>
    </div>
</template>
<script>
    var commentBox = {
        data() {
            return {
```

```
                user: '',
                content: ''
            }
        },
        template: '#tmpl',
        methods: {
            postComment() {     //发表评论的方法
                var comment = { id: Date.now(), user: this.user, content: this.content }
                // 从 localStorage 中获取所有的评论
                var list = JSON.parse(localStorage.getItem('cmts') || '[]')
                list.unshift(comment)
                // 重新保存最新的评论数据
                localStorage.setItem('cmts', JSON.stringify(list))
                this.user = this.content = ''
                this.$emit('func')
            }
        }
    }
    //创建 Vue 实例,得到 ViewModel
    var vm = new Vue({
        el: '#app',
        data: {
            list: [
                { id: Date.now(), user: 'beixi', content: '这是我的网名' },
                { id: Date.now(), user: 'jzj', content: '这是我的真名' },
                { id: Date.now(), user: '贝西奇谈', content: '有任何问题可以关注公众号' }
            ]
        },
        beforeCreate(){ /* 注意: 这里不能调用 loadComments 方法,因为在执行这个钩子函数
的时候,data 和 methods 都还没有被初始化 */
    },
        created(){
        this.loadComments()
    },
        methods: {
            loadComments() {    //从本地的 localStorage 中加载评论列表
                var list = JSON.parse(localStorage.getItem('cmts') || '[]')
                this.list = list
            }
        },
        components: {
            'cmt-box': commentBox
        }
    });
</script>
</body>
</html>
```

页面显示效果如图 5-12 所示。

图 5-12　综合案例实现效果