开篇：Vue.js 的精髓——组件

写在前面

Vue.js，无疑是当下最火热的前端框架 *Almost*，而 Vue.js 最精髓的，正是它的组件与组件化。写一个 Vue 工程，也就是在写一个个的组件。

业务场景是千变万化的，而不变的是 Vue.js 组件开发的核心思想和使用技巧，掌握了 Vue.js 组件的各种开发模式，再复杂的业务场景也可以轻松化解。本小册则着重介绍笔者在 3 年的 Vue.js 开发及两年的 iView 开源中积累和沉淀的对 Vue.js 组件的见解和经验。

本小册**不会**介绍 Vue.js 的基础用法，因为市面上已经沉淀了大量的相关技术资料，而且 Vue.js 的文档已经足够详细。如果您尚未接触 Vue.js 或正打算开始了解，推荐您先阅读笔者出版的《Vue.js 实战》（清华大学出版社）一书，它适合刚接触 Vue.js 的开发者。因此，本小册适合已经了解或使用过 Vue.js 的开发者。

这一节，我们先笼统地聊聊 Vue.js 组件和组件化以及本小册各章节的梳理。

组件的分类

一般来说，Vue.js 组件主要分成三类：

1. 由 `vue-router` 产生的每个页面，它本质上也是一个组件（.vue），主要承载当前页面的 HTML 结构，会包含数据获取、数据整理、数据可视化等常规业务。整个文件相对较大，但一般不会有 `props` 选项和 `自定义事件`，因为它作为路由的渲染，不会被复用，因此也不会对外提供接口。

   在项目开发中，我们写的大部分代码都是这类的组件（页面），协同开发时，每人维护自己的页面，很少有交集。这类组件相对是最好写的，因为主要是还原设计稿，完成需求，不需要太多模块和架构设计上的考虑。

2. 不包含业务，独立、具体功能的基础组件，比如**日期选择器**、**模态框**等。这类组件作为项目的基础控件，会被大量使用，因此组件的 API 进行过高强度的抽象，可以通过不同配置实现不同的功能。比如笔者开源的 iView，就是包含了 50 多个这样基础组件的 UI 组件库。

   每个公司都有自己的组件使用规范或组件库，但要开发和维护一套像 iView 这样的组件库，投入的人力和精力还是很重的，所以出于成本考虑，很多项目都会使用已有的开源组件库。

   独立组件的开发难度要高于第一类组件，因为它的侧重点是 API 的设计、兼容性、性能、以及复杂的功能。这类组件对 JavaScript 的编程能力有一定要求，也会包含非常多的技巧，比如在不依赖 Vuex 和 Bus（因为独立组件，无法依赖其它库）的情况下，各组件间的通信，还会涉及很多脑壳疼的逻辑，比如日期选择器要考虑不同时区、国家的日历习惯，支持多种日期格式。

   本小册也会重点介绍此类组件的各种开发模式和技巧，对应不同的模式，会带有具体的组件实战。

3. 业务组件。它不像第二类独立组件只包含某个功能，而是在业务中被多个页面复用的，它与独立组件的区别是，业务组件只在当前项目中会用到，不具有通用性，而且会包含一些业务，比如数据请求；而独立组件不含业务，在任何项目中都可以使用，功能单一，比如一个具有数据校验功能的输入框。

   业务组件更像是介于第一类和第二类之间，在开发上也与独立组件类似，但寄托于项目，你可以使用项目中的技术栈，比如 Vuex、axios、echarts 等，所以它的开发难度相对独立组件要容易点，但也有必要考虑组件的可维护性和复用性。

小册的内容

因为本小册是围绕 Vue.js 组件展开的，所以第二节会讲解 Vue.js 组件的三个 API：`prop`、`event`、`slot`，当然，如果你已经开发过一些独立组件，完全可以跳过这节内容。

3 - 7 小节会介绍组件间通信的一些方法和黑科技，一部分是 Vue.js 内置的，一部分是自行实现的，在实际开发中，会非常实用。同时也利用这些方法完成了两个具体的实战案例：

1. 具有数据校验功能的表单组件 —— Form；
2. 组合多选框组件 —— CheckboxGroup & Checkbox。

本小册都会以这种核心科技 + 对应实战的形式展开。

8 - 10 小节介绍 Vue 的构造器 extend 和手动挂载组件 $mount 的用法及案例。Vue.js 除了我们正常 `new Vue()` 外，还可以手动挂载的，这 3 节将介绍手动挂载一个 Vue 组件的使用场景。其中涉及到两个案例：

1. 动态渲染 .vue 文件的组件 —— Display；
2. 全局通知组件 —— $Alert。

Display 组件用于将 `.vue` 文件渲染出来，线上的案例是 iView Run，它不需要你重新编译项目，就可以渲染一个标准的 Vue.js 组件。

$Alert 是全局的通知组件，它的调用方法不同于常规组件。常规组件使用方法形如：

```
<template>
    <Alert content="通知内容" :duration="3"></Alert>
</template>
<script>
    import Alert from '../components/alert.vue';

    export default {
        components: { Alert }
    }
</script>
```

而 `$Alert` 的调用更接近 JS 语法：

```
export default {
    methods: {
        showMessage () {
            this.$Alert({
                content: '通知内容',
                duration: 3
            });
        }
    }
}
```

虽然与常规 Vue 组件调用方式不同，但底层仍然由 Vue 组件构成和维护。

11 - 12 小节介绍 Render 函数与 Functional Render，并完成一个能够渲染自定义列的 Table 组件。Render 函数也是 Vue.js 组件重要的一部分，只不过在大多数业务中不常使用。本小节会介绍它的使用场景。

13 小节介绍**作用域 slot（slot-scope）**，并基于这种方法同样实现 Table 组件。slot 用的很多，但 slot-scope 在业务中并不常用，但在一些特定场景下，比如组件内部有循环体时，会非常实用。

14 - 15 小节介绍递归组件，并完成树形控件 —— Tree。

16 - 19 小节是综合拓展，会着重讲解 Vue.js 容易忽略却很重要的 API，以及对 Vue.js 面试题的详细分析。除此之外，还会总结笔者在两年的 iView 开源经历中的经验，除了技术细节外，还包括开源项目的持续性发展、推广等。

结语

三年前，我开始接触 Vue.js 框架，当时就被它的轻量、组件化和友好的 API 所吸引。与此同时，我也开源了 iView 项目。三年的磨(cǎi)砺(kēng)，沉淀了不少关于 Vue.js 组件的经验。

本小册的内容也许不会让你的技术一夜间突飞猛进，但绝对使你醍醐灌顶。

那么，请准备好一台电脑和一杯咖啡，一起来探索 Vue.js 的精髓吧。

基础：Vue.js 组件的三个 API：prop、event、slot

> 如果您已经对 Vue.js 组件的基础用法了如指掌，可以跳过本小节，不过当做复习稍读一下也无妨。

组件的构成

一个再复杂的组件，都是由三部分组成的：prop、event、slot，它们构成了 Vue.js 组件的 API。如果你开发的是一个通用组件，那一定要事先设计好这三部分，因为组件一旦发布，后面再修改 API 就很困难了，使用者都是希望不断新增功能，修复 bug，而不是经常变更接口。如果你阅读别人写的组件，也可以从这三个部分展开，它们可以帮助你快速了解一个组件的所有功能。

属性 prop

`prop` 定义了这个组件有哪些可配置的属性，组件的核心功能也都是它来确定的。写通用组件时，props 最好用**对象**的写法，这样可以针对每个属性设置类型、默认值或自定义校验属性的值，这点在组件开发中很重要，然而很多人却忽视，直接使用 props 的数组用法，这样的组件往往是不严谨的。比如我们封装一个按钮组件 `<i-button>`：

```
<template>
  <button :class="'i-button-size' + size" :disabled="disabled"></button>
</template>
<script>
  // 判断参数是否是其中之一
  function oneOf (value, validList) {
    for (let i = 0; i < validList.length; i++) {
      if (value === validList[i]) {
        return true;
      }
    }
    return false;
  }

  export default {
    props: {
      size: {
        validator (value) {
          return oneOf(value, ['small', 'large', 'default']);
```

```
        },
        default: 'default'
      },
      disabled: {
        type: Boolean,
        default: false
      }
    }
  }
</script>
```

使用组件：

```
<i-button size="large"></i-button>
<i-button disabled></i-button>
```

组件中定义了两个属性：尺寸 size 和 是否禁用 disabled。其中 size 使用 `validator` 进行了值的自定义验证，也就是说，从父级传入的 size，它的值必须是指定的 **small、large、default** 中的一个，默认值是 default，如果传入这三个以外的值，都会抛出一条警告。

要注意的是，组件里定义的 props，都是**单向数据流**，也就是只能通过父级修改，组件自己不能修改 props 的值，只能修改定义在 data 里的数据，非要修改，也是通过后面介绍的自定义事件通知父级，由父级来修改。

在使用组件时，也可以传入一些标准的 html 特性，比如 **id**、**class**：

```
<i-button id="btn1" class="btn-submit"></i-button>
```

这样的 html 特性，在组件内的 `<button>` 元素上会继承，并不需要在 props 里再定义一遍。这个特性是默认支持的，如果不期望开启，在组件选项里配置 `inheritAttrs: false` 就可以禁用了。

插槽 slot

如果要给上面的按钮组件 `<i-button>` 添加一些文字内容，就要用到组件的第二个 API：插槽 slot，它可以分发组件的内容，比如在上面的按钮组件中定义一个插槽：

```
<template>
  <button :class="'i-button-size' + size" :disabled="disabled">
    <slot></slot>
  </button>
</template>
```

这里的 `<slot>` 节点就是指定的一个插槽的位置，这样在组件内部就可以扩展内容了：

```
<i-button>按钮 1</i-button>
<i-button>
  <strong>按钮 2</strong>
</i-button>
```

当需要多个插槽时，会用到具名 slot，比如上面的组件我们再增加一个 slot，用于设置另一个图标组件：

```
<template>
  <button :class="'i-button-size' + size" :disabled="disabled">
    <slot name="icon"></slot>
    <slot></slot>
  </button>
</template>
```

```
<i-button>
  <i-icon slot="icon" type="checkmark"></i-icon>
  按钮 1
</i-button>
```

这样，父级内定义的内容，就会出现在组件对应的 slot 里，没有写名字的，就是默认的 slot。

在组件的 `<slot>` 里也可以写一些默认的内容，这样在父级没有写任何 slot 时，它们就会出现，比如：

```
<slot>提交</slot>
```

自定义事件 event

现在我们给组件 `<i-button>` 加一个点击事件，目前有两种写法，我们先看自定义事件 event（部分代码省略）：

```
<template>
  <button @click="handleClick">
    <slot></slot>
  </button>
</template>
<script>
  export default {
    methods: {
      handleClick (event) {
        this.$emit('on-click', event);
      }
    }
  }
</script>
```

通过 `$emit`，就可以触发自定义的事件 `on-click` ，在父级通过 `@on-click` 来监听：

```
<i-button @on-click="handleClick"></i-button>
```

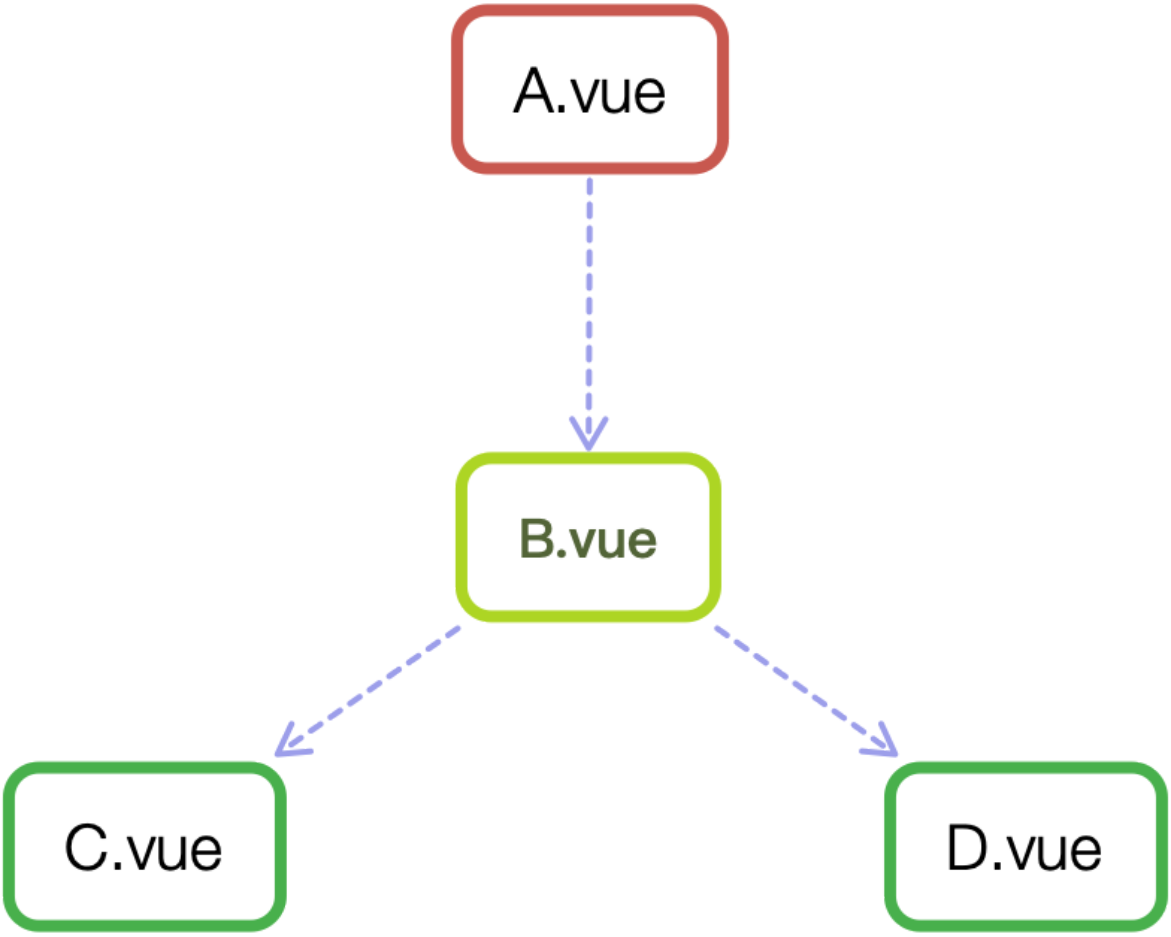
上面的 click 事件，是在组件内部的 `<button>` 元素上声明的，这里还有另一种方法，直接在父级声明，但为了区分原生事件和自定义事件，要用到事件修饰符 `.native`，所以上面的示例也可以这样写：

```
<i-button @click.native="handleClick"></i-button>
```

如果不写 `.native` 修饰符，那上面的 `@click` 就是**自定义事件** click，而非**原生事件** click，但我们在组件内只触发了 `on-click` 事件，而不是 `click`，所以直接写 `@click` 会监听不到。

组件的通信

一般来说，组件可以有以下几种关系：



A 和 B、B 和 C、B 和 D 都是父子关系，C 和 D 是兄弟关系，A 和 C 是隔代关系（可能隔多代）。组件间经常会通信，Vue.js 内置的通信手段一般有两种：

- `ref`：给元素或组件注册引用信息；
- `$parent` / `$children`：访问父 / 子实例。

这两种都是直接得到组件实例，使用后可以直接调用组件的方法或访问数据，比如下面的示例中，用 ref 来访问组件（部分代码省略）：

```
// component-a
export default {
  data () {
    return {
      title: 'Vue.js'
    }
```

```
  },
  methods: {
    sayHello () {
      window.alert('Hello');
    }
  }
}
```

```
<template>
  <component-a ref="comA"></component-a>
</template>
<script>
  export default {
    mounted () {
      const comA = this.$refs.comA;
      console.log(comA.title);  // Vue.js
      comA.sayHello();  // 弹窗
    }
  }
</script>
```

`$parent` 和 `$children` 类似，也是基于当前上下文访问父组件或全部子组件的。

这两种方法的弊端是，无法在**跨级**或**兄弟**间通信，比如下面的结构：

```
// parent.vue
<component-a></component-a>
<component-b></component-b>
<component-b></component-b>
```

我们想在 component-a 中，访问到引用它的页面中（这里就是 parent.vue）的两个 component-b 组件，那这种情况下，就得配置额外的插件或工具了，比如 Vuex 和 Bus 的解决方案，本小册不再做它们的介绍，读者可以自行阅读相关内容。不过，它们都是依赖第三方插件的存在，这在开发独立组件时是不可取的，而在小册的后续章节，会陆续介绍一些黑科技，它们完全不依赖任何三方插件，就可以轻松得到任意的组件实例，或在任意组件间进行通信，且适用于任意场景。

结语

本小节带您复习了 Vue.js 组件的核心知识点，虽然这并没有完全覆盖 Vue.js 的 API，但对于组件开发来说已经足够了，后续章节也会陆续扩展更多的用法。

基于 Vue.js 开发独立组件，并不是新奇的挑战，坦率地讲，它本质上还是 JavaScript。掌握了 Vue.js 组件的这三个 API 后，剩下的便是程序的设计。在组件开发中，最难的环节应当是解耦组件的交互逻辑，尽量把复杂的逻辑分发到不同的子组件中，然后彼此建立联系，在这其中，计算属性（computed）和混合（mixins）是两个重要的技术点，合理利用，就能发挥出 Vue.js 语言的最大特点：把状态（数据）的维护交给 Vue.js 处理，我们只专注在交互上。

当您最终读完本小册时，应该会总结出和笔者一样的感悟：Vue.js 组件开发，玩到最后还是在拼 JavaScript 功底。对于每一位使用 Vue.js 的开发者来说，阅读完本小册都可以尝试开发和维护一套属于自己的组件库，并乐在其中，而且你会越发觉得，一个组件或一套组件库，就是融合了前端精髓的产出。

扩展阅读

- Vue 组件通信之 Bus
- Vuex 通俗版教程

组件的通信 1：provide / inject

上一节中我们说到，`ref` 和 `$parent / $children` 在**跨级**通信时是有弊端的。当组件 A 和组件 B 中间隔了数代（甚至不确定具体级别）时，以往会借助 Vuex 或 Bus 这样的解决方案，不得不引入三方库来支持。本小节则介绍一种无依赖的组件通信方法：Vue.js 内置的 provide / inject 接口。

什么是 provide / inject

`provide / inject` 是 Vue.js 2.2.0 版本后新增的 API，在文档中这样介绍：

> https://cn.vuejs.org/v2/api/#provide-inject
>
> 这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效。如果你熟悉 React，这与 React 的上下文特性很相似。

并且文档中有如下提示：

> provide 和 inject 主要为高阶插件/组件库提供用例。并不推荐直接用于应用程序代码中。

看不懂上面的介绍没有关系，不过上面的这句提示应该明白，就是说 Vue.js 不建议在业务中使用这对 API，而是在插件 / 组件库（比如 iView，事实上 iView 的很多组件都在用）。**不过建议归建议，如果你用好了，这个 API 会非常有用。**

我们先来看一下这个 API 怎么用，假设有两个组件： **A.vue** 和 **B.vue**，B 是 A 的子组件。

```
// A.vue
export default {
  provide: {
    name: 'Aresn'
  }
}

// B.vue
export default {
  inject: ['name'],
  mounted () {
    console.log(this.name);  // Aresn
  }
}
```

可以看到，在 A.vue 里，我们设置了一个 **provide: name**，值为 Aresn，它的作用就是将 **name** 这个变量提供给它的所有子组件。而在 B.vue 中，通过 `inject` 注入了从 A 组件中提供的 **name** 变量，那么在组件 B 中，就可以直接通过 **this.name** 访问这个变量了，它的值也是 Aresn。这就是 provide / inject API 最核心的用法。

需要注意的是：

> provide 和 inject 绑定并**不是可响应**的。这是刻意为之的。然而，如果你传入了一个可监听的对象，那么其对象的属性还是可响应的。

所以，上面 A.vue 的 name 如果改变了，B.vue 的 this.name 是不会改变的，仍然是 Aresn。

替代 Vuex

我们知道，在做 Vue 大型项目时，可以使用 Vuex 做状态管理，它是一个专为 Vue.js 开发的**状态管理模式**，用于集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

那了解了 provide / inject 的用法，下面来看怎样替代 Vuex。当然，我们的目的并不是为了替代 Vuex，它还是有相当大的用处，这里只是介绍另一种可行性。

使用 Vuex，最主要的目的是跨组件通信、全局数据维护、多人协同开发。需求比如有：用户的登录信息维护、通知信息维护等全局的状态和数据。

一般在 webpack 中使用 Vue.js，都会有一个入口文件 **main.js**，里面通常导入了 Vue、VueRouter、iView 等库，通常也会导入一个入口组件 `app.vue` 作为根组件。一个简单的 app.vue 可能只有以下代码：

```
<template>
  <div>
    <router-view></router-view>
  </div>
</template>
<script>
  export default {

  }
</script>
```

使用 provide / inject 替代 Vuex，就是在这个 app.vue 文件上做文章。

我们把 app.vue 理解为一个最外层的根组件，用来存储所有需要的全局数据和状态，甚至是计算属性（computed）、方法（methods）等。因为你的项目中所有的组件（包含路由），它的父组件（或根组件）都是 app.vue，所以我们**把整个 app.vue 实例通过 `provide` 对外提供。**

app.vue：

```
<template>
  <div>
    <router-view></router-view>
  </div>
</template>
<script>
  export default {
    provide () {
      return {
        app: this
      }
    }
  }
</script>
```

上面，我们把整个 app.vue 的实例 `this` 对外提供，命名为 **app**（这个名字可以自定义，推荐使用 app，使用这个名字后，子组件不能再使用它作为局部属性）。接下来，任何组件（或路由）只要通过 `inject` 注入 app.vue 的 app 的话，都可以直接通过 **this.app.xxx** 来访问 app.vue 的 `data`、`computed`、`methods` 等内容。

app.vue 是整个项目第一个被渲染的组件，而且只会渲染一次（即使切换路由，app.vue 也不会被再次渲染），利用这个特性，很适合做一次性全局的状态数据管理，例如，我们将用户的登录信息保存起来：

app.vue，部分代码省略：

```
<script>
  export default {
    provide () {
      return {
        app: this
      }
    },
    data () {
      return {
        userInfo: null
      }
    },
    methods: {
      getUserInfo () {
        // 这里通过 ajax 获取用户信息后，赋值给 this.userInfo，以下为伪代码
        $.ajax('/user/info', (data) => {
          this.userInfo = data;
        });
      }
    },
    mounted () {
      this.getUserInfo();
    }
  }
</script>
```

这样，任何页面或组件，只要通过 `inject` 注入 `app` 后，就可以直接访问 `userInfo` 的数据了，比如：

```
<template>
  <div>
    {{ app.userInfo }}
  </div>
</template>
<script>
  export default {
    inject: ['app']
  }
</script>
```

是不是很简单呢。除了直接使用数据，还可以调用方法。比如在某个页面里，修改了个人资料，这时一开始在 `app.vue` 里获取的 `userInfo` 已经不是最新的了，需要重新获取。可以这样使用：

某个页面：

```
<template>
  <div>
    {{ app.userInfo }}
  </div>
</template>
<script>
  export default {
    inject: ['app'],
    methods: {
      changeUserInfo () {
        // 这里修改完用户数据后，通知 app.vue 更新，以下为伪代码
        $.ajax('/user/update', () => {
          // 直接通过 this.app 就可以调用 app.vue 里的方法
          this.app.getUserInfo();
        })
      }
    }
  }
</script>
```

同样非常简单。只要理解了 `this.app` 是直接获取整个 `app.vue` 的实例后，使用起来就得心应手了。想一想，配置复杂的 Vuex 的全部功能，现在是不是都可以通过 `provide / inject` 来实现了呢？

进阶技巧

如果你的项目足够复杂，或需要多人协同开发时，在 `app.vue` 里会写非常多的代码，多到结构复杂难以维护。这时可以使用 Vue.js 的混合 `mixins`，将不同的逻辑分开到不同的 js 文件里。

比如上面的用户信息，就可以放到混合里：

user.js：

```
export default {
  data () {
    return {
      userInfo: null
    }
  },
  methods: {
    getUserInfo () {
      // 这里通过 ajax 获取用户信息后，赋值给 this.userInfo，以下为伪代码
      $.ajax('/user/info', (data) => {
        this.userInfo = data;
      });
    }
  },
  mounted () {
    this.getUserInfo();
  }
}
```

然后在 `app.vue` 中混合：

app.vue：

```
<script>
  import mixins_user from '../mixins/user.js';

  export default {
    mixins: [mixins_user],
    data () {
      return {

      }
    }
  }
</script>
```

这样，跟用户信息相关的逻辑，都可以在 `user.js` 里维护，或者由某个人来维护，`app.vue` 也就很容易维护了。

独立组件中使用

如果你顾忌 Vue.js 文档中所说，provide / inject 不推荐直接在应用程序中使用，那没有关系，仍然使用你熟悉的 Vuex 或 Bus 来管理你的项目就好。我们介绍的这对 API，主要还是在独立组件中发挥作用的。

只要一个组件使用了 `provide` 向下提供数据，那其下所有的子组件都可以通过 `inject` 来注入，不管中间隔了多少代，而且可以注入多个来自不同父级提供的数据。需要注意的是，一旦注入了某个数据，比如上面示例中的 `app`，那这个组件中就不能再声明 `app` 这个数据了，因为它已经被父级占有。

独立组件使用 provide / inject 的场景，主要是具有联动关系的组件，比如接下来很快会介绍的第一个实战：具有数据校验功能的表单组件 Form。它其实是两个组件，一个是 Form，一个是 FormItem，FormItem 是 Form 的子组件，它会依赖 Form 组件上的一些特性（props），所以就需要得到父组件 Form，这在 Vue.js 2.2.0 版本以前，是没有 provide / inject 这对 API 的，而 Form 和 FormItem 不一定是父子关系，中间很可能间隔了其它组件，所以不能单纯使用 `$parent` 来向上获取实例。在 Vue.js 2.2.0 之前，一种比较可行的方案是用计算属性动态获取：

```
computed: {
  form () {
    let parent = this.$parent;
    while (parent.$options.name !== 'Form') {
      parent = parent.$parent;
    }
    return parent;
  }
}
```

每个组件都可以设置 `name` 选项，作为组件名的标识，利用这个特点，通过向上遍历，直到找到需要的组件。这个方法可行，但相比一个 `inject` 来说，太费劲了，而且不那么优雅和 native。如果用 inject，可能只需要一行代码：

```
export default {
  inject: ['form']
}
```

不过，这一切的前提是你使用 Vue.js 2.2.0 以上版本。

结语

如果这是你第一次听说 provide / inject 这对 API，一定觉得它太神奇了，至少笔者第一时间知晓时是这样的。它解决了独立组件间通信的问题，用好了还有出乎意料的效果。笔者在开发 iView Developer 时，全站就是使用这种方法来做全局数据的管理的，如果你有机会一个人做一个项目时，不妨试试这种方法。

下一节，将介绍另一种组件间通信的方法，不同于 provide / inject 的是，它们不是 Vue.js 内置的 API。

组件的通信 2：派发与广播——自行实现 dispatch 和 broadcast 方法

上一讲的 provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。然后有两种场景它不能很好的解决：

- 父组件向子组件（支持跨级）传递数据；
- 子组件向父组件（支持跨级）传递数据。

这种父子（含跨级）传递数据的通信方式，Vue.js 并没有提供原生的 API 来支持，而是推荐使用大型数据状态管理工具 Vuex，而我们之前已经介绍过 Vuex 的场景与在独立组件（或库）中使用的限制。本小节则介绍一种在父子组件间通信的方法 `dispatch` 和 `broadcast`。

*on*与 emit

如果您使用过较早的 Vue.js 1.x 版本，肯定对 *$dispatch* 和 *$broadcast* 这两个内置的方法很熟悉，不过它们都在 Vue.js 2.x 里废弃了。在正式介绍主角前，我们先看看 `$on` 与 `$emit` 这两个 API，因为它们是本节内容的基础。

`$emit` 会在**当前组件**实例上触发自定义事件，并传递一些参数给监听器的回调，一般来说，都是在父级调用这个组件时，使用 `@on` 的方式来监听自定义事件的，比如在子组件中触发事件：

```
// child.vue，部分代码省略
export default {
  methods: {
    handleEmitEvent () {
      this.$emit('test', 'Hello Vue.js');
    }
  }
}
```

在父组件中监听由 *child.vue* 触发的自定义事件 **test**：

```
```

```
<!-- parent.vue，部分代码省略-->
<template>
  <child-component @test="handleEvent">
</template>
<script>
  export default {
    methods: {
      handleEvent (text) {
        console.log(text);  // Hello Vue.js
      }
    }
  }
</script>
```

这里看似是在父组件 *parent.vue* 中绑定的自定义事件 **test** 的处理句柄，然而事件 test 并不是在父组件上触发的，而是在子组件 *child.vue* 里触发的，只是通过 `v-on` 在父组件中监听。既然是子组件自己触发的，那它自己也可以监听到，这就要使用 `$on` 来监听实例上的事件，换言之，组件使用 `$emit` 在自己实例上触发事件，并用 `$on` 监听它。

听起来这种神（sāo）操作有点多此一举，我们不妨先来看个示例：

（也可通过在线链接 https://run.iviewui.com/ggsomfHM 直接运行该示例）

```
<template>
  <div>
    <button @click="handleEmitEvent">触发自定义事件</button>
  </div>
</template>
<script>
  export default {
    methods: {
      handleEmitEvent () {
        // 在当前组件上触发自定义事件 test，并传值
        this.$emit('test', 'Hello Vue.js')
      }
    },
    mounted () {
      // 监听自定义事件 test
      this.$on('test', (text) => {
        window.alert(text);
      });
    }
  }
</script>
```

`$on` 监听了自己触发的自定义事件 test，因为有时不确定何时会触发事件，一般会在 `mounted` 或 `created` 钩子中来监听。

仅上面的示例，的确是多此一举的，因为大可在 handleEmitEvent 里直接写 window.alert(text)，没必要绕一圈。

之所以多此一举，是因为 handleEmitEvent 是当前组件内的 `<button>` 调用的，如果这个方法不是它自己调用，而是其它组件调用的，那这个用法就大有可为了。

了解了 `$on` 和 `$emit` 的用法后，我们再来看两个“过时的” API。

Vue.js 1.x 的 $dispatch$ 与 broadcast

虽然 Vue.js 1.x 已经成为过去时，但为了充分理解本节通信方法的使用场景，还是有必要来了解一点它的历史。

在 Vue.js 1.x 中，提供了两个方法：`$dispatch` 和 `$broadcast`，前者用于向上级派发事件，只要是它的父级（一级或多级以上），都可以在组件内通过 `$on`（或 events，2.x 已废弃）监听到，后者相反，是由上级向下级广播事件的。

来看一个简单的示例：

```
<!-- 注意：该示例为 Vue.js 1.x 版本 -->
<!-- 子组件 -->
<template>
  <button @click="handleDispatch">派发事件</button>
</template>
<script>
export default {
  methods: {
    handleDispatch () {
      this.$dispatch('test', 'Hello, Vue.js');
    }
  }
}
</script>
```

```
<!-- 父组件，部分代码省略 -->
<template>
  <child-component></child-component>
</template>
<script>
  export default {
    mounted () {
      this.$on('test', (text) => {
        console.log(text);  // Hello, Vue.js
      });
    }
  }
</script>
```

$broadcast 类似，只不过方向相反。这两种方法一旦发出事件后，任何组件都是可以接收到的，就近原则，而且会在第一次接收到后停止冒泡，除非返回 true。

这两个方法虽然看起来很好用，但是在 Vue.js 2.x 中都废弃了，官方给出的解释是：

> 因为基于组件树结构的事件流方式有时让人难以理解，并且在组件结构扩展的过程中会变得越来越脆弱。

虽然在业务开发中，它没有 Vuex 这样专门管理状态的插件清晰好用，但对独立组件（库）的开发，绝对是福音。因为独立组件一般层级并不会很复杂，并且剥离了业务，不会变的难以维护。

知道了 *$dispatch* 和 *$broadcast* 的前世今生，接下来我们就在 Vue.js 2.x 中自行实现这两个方法。

自行实现 dispatch 和 broadcast 方法

自行实现的 dispatch 和 broadcast 方法，不能保证跟 Vue.js 1.x 的 *$dispatch* 和 *$broadcast* 具有完全相同的体验，但基本功能是一样的，都是解决父子组件（含跨级）间的通信问题。

通过目前已知的信息，我们要实现的 dispatch 和 broadcast 方法，将具有以下功能：

- 在子组件调用 dispatch 方法，向上级指定的组件实例（最近的）上触发自定义事件，并传递数据，且该上级组件已预先通过 `$on` 监听了这个事件；
- 相反，在父组件调用 broadcast 方法，向下级指定的组件实例（最近的）上触发自定义事件，并传递数据，且该下级组件已预先通过 `$on` 监听了这个事件。

实现这对方法的关键点在于，如何正确地向上或向下找到对应的组件实例，并在它上面触发方法。在设计一个新功能（features）时，可以先确定这个功能的 API 是什么，也就是说方法名、参数、使用样例，确定好 API，再来写具体的代码。

因为 Vue.js 内置的方法，才是以 `$` 开头的，比如 `$nextTick`、`$emit` 等，为了避免不必要的冲突并遵循规范，这里的 dispatch 和 broadcast 方法名前不加 `$`。并且该方法可能在很多组件中都会使用，复用起见，我们封装在混合（mixins）里。那它的使用样例可能是这样的：

```
// 部分代码省略
import Emitter from '../mixins/emitter.js'

export default {
  mixins: [ Emitter ],
  methods: {
    handleDispatch () {
      this.dispatch();  // ①
    },
    handleBroadcast () {
      this.broadcast();  // ②
    }
  }
}
```

上例中行 ① 和行 ② 的两个方法就是在导入的混合 **emitter.js** 中定义的，这个稍后我们再讲，先来分析这两个方法应该传入什么参数。一般来说，为了跟 Vue.js 1.x 的方法一致，第一个参数应当是自定义事件名，比如 “test”，第二个参数是传递的数据，比如 “Hello, Vue.js”，但在这里，有什么问题呢？只通过这两个参数，我们没办法知道要在哪个组件上触发事件，因为自行实现的这对方法，与 Vue.js 1.x 的原生方法机理上是有区别的。上文说到，实现这对方法的关键点在于准确地**找到组件实例**。那在寻找组件实例上，我们的“惯用伎俩”就是通过遍历来匹配组件的 `name` 选项，在独立组件（库）里，每个组件的 `name` 值应当是唯一的，name 主要用于递归组件，在后面小节会单独介绍。

先来看下 **emitter.js** 的代码：

```
function broadcast(componentName, eventName, params) {
  this.$children.forEach(child => {
    const name = child.$options.name;

    if (name === componentName) {
      child.$emit.apply(child, [eventName].concat(params));
    } else {
      broadcast.apply(child, [componentName, eventName].concat([params]));
    }
  });
}
export default {
```

```
  methods: {
    dispatch(componentName, eventName, params) {
      let parent = this.$parent || this.$root;
      let name = parent.$options.name;

      while (parent && (!name || name !== componentName)) {
        parent = parent.$parent;

        if (parent) {
          name = parent.$options.name;
        }
      }
      if (parent) {
        parent.$emit.apply(parent, [eventName].concat(params));
      }
    },
    broadcast(componentName, eventName, params) {
      broadcast.call(this, componentName, eventName, params);
    }
  }
};
```

因为是用作 mixins 导入，所以在 methods 里定义的 dispatch 和 broadcast 方法会被混合到组件里，自然就可以用 `this.dispatch` 和 `this.broadcast` 来使用。

这两个方法都接收了三个参数，第一个是组件的 `name` 值，用于向上或向下递归遍历来寻找对应的组件，第二个和第三个就是上文分析的自定义事件名称和要传递的数据。

可以看到，在 dispatch 里，通过 *while* 语句，不断向上遍历更新当前组件（即上下文为当前调用该方法的组件）的父组件实例（变量 parent 即为父组件实例），直到匹配到定义的 `componentName` 与某个上级组件的 `name` 选项一致时，结束循环，并在找到的组件实例上，调用 `$emit` 方法来触发自定义事件 `eventName`。broadcast 方法与之类似，只不过是向下遍历寻找。

来看一下具体的使用方法。有 **A.vue** 和 **B.vue** 两个组件，其中 B 是 A 的子组件，中间可能跨多级，在 A 中向 B 通信：

```
<!-- A.vue -->
<template>
  <button @click="handleClick">触发事件</button>
</template>
<script>
  import Emitter from '../mixins/emitter.js';

  export default {
    name: 'componentA',
    mixins: [ Emitter ],
    methods: {
      handleClick () {
        this.broadcast('componentB', 'on-message', 'Hello Vue.js');
      }
    }
  }
</script>
```

```
// B.vue
export default {
  name: 'componentB',
  created () {
    this.$on('on-message', this.showMessage);
  },
  methods: {
    showMessage (text) {
      window.alert(text);
    }
  }
}
```

同理，如果是 B 向 A 通信，在 B 中调用 dispatch 方法，在 A 中使用 $on 监听事件即可。

以上就是自行实现的 dispatch 和 broadcast 方法，相比 Vue.js 1.x，有以下不同：

- 需要额外传入组件的 name 作为第一个参数；
- 无冒泡机制；
- 第三个参数传递的数据，只能是一个（较多时可以传入一个对象），而 Vue.js 1.x 可以传入多个参数，当然，你对 emitter.js 稍作修改，也能支持传入多个参数，只是一般场景传入一个对象足以。

结语

Vue.js 的组件通信到此还没完全结束，如果你想“趁热打铁”一口气看完，可以先阅读第 6 节组件的通信 3。亦或按顺序看下一节的实战，来进一步加深理解 provide / inject 和 dispatch / broadcast 这两对通信方法的使用场景。

注：本节部分代码参考 iView。

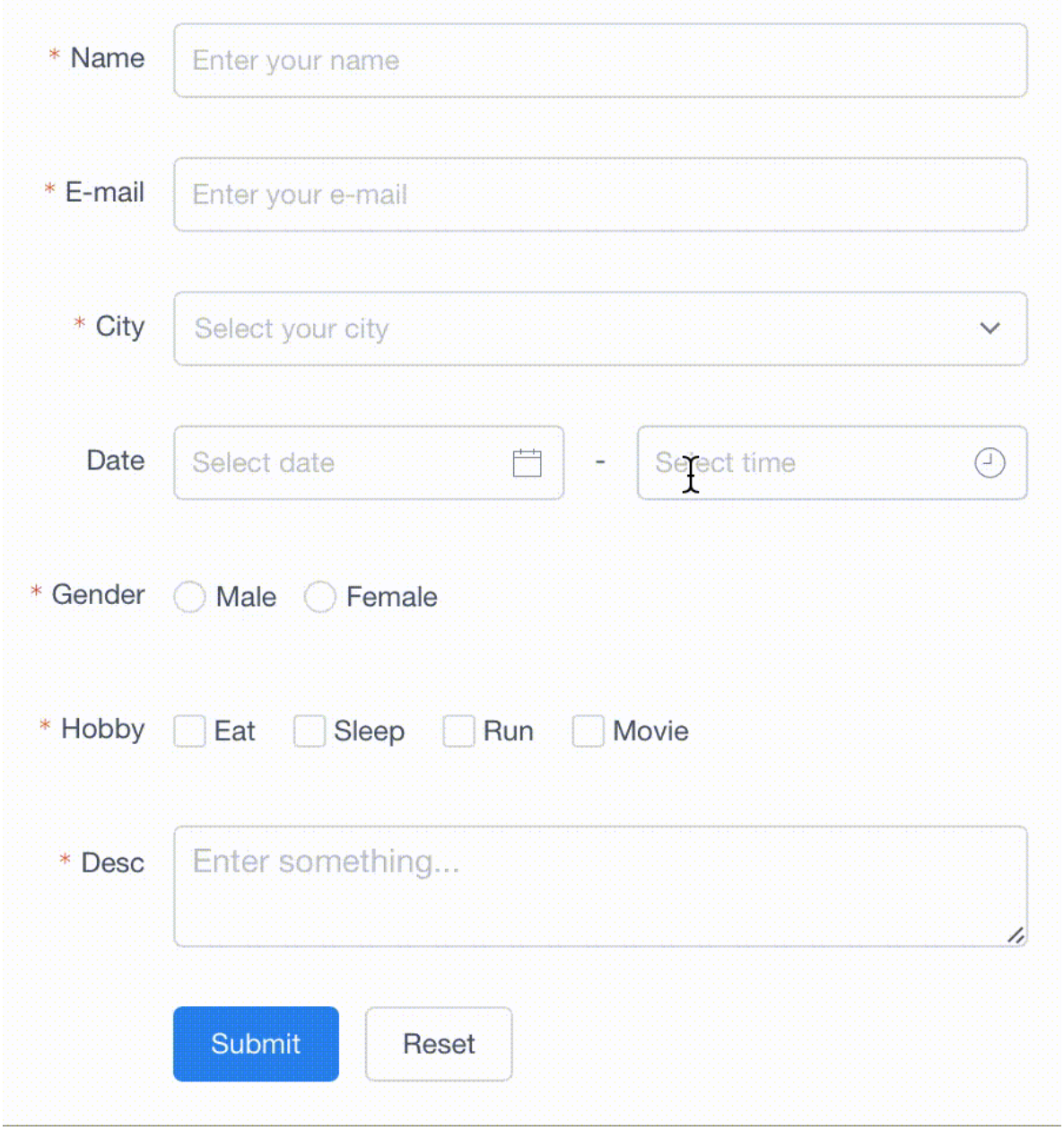
实战 1：具有数据校验功能的表单组件——Form

在第 3 节和第 4 节中，我们介绍了组件间的两种通信方法：provide / inject 和 dispatch / broadcast，前者是 Vue.js 内置的，主要用于子组件获取父组件（包括跨级）的状态；后者是自行实现的一种混合，用于父子组件（包括跨级）间通过自定义事件通信。本小节则基于这两种通信方法，来实现一个具有数据校验功能的表单组件——Form。

Form 组件概览

表单类组件在项目中会大量使用，比如输入框（Input）、单选（Radio）、多选（Checkbox）、下拉选择器（Select）等。在使用表单类组件时，也会经常用到数据校验，如果每次都写校验程序来对每一个表单控件校验，会很低效，因此需要一个能够校验基础表单控件的组件，也就是本节要完成的 Form 组件。一般的组件库都提供了这个组件，比如 iView，它能够校验内置的 15 种控件，且支持校验自定义组件，如下图所示：

（也可以在线访问本示例体验：https://run.iviewui.com/jwrqnFss）



Form 组件分为两个部分，一个是外层的 `Form` 表单域组件，一组表单控件只有一个 Form，而内部包含了多个 `FormItem` 组件，每一个表单控件都被一个 FormItem 包裹。基本的结构看起来像：

```
<i-form>
  <i-form-item>
    <i-input v-model="form.name"></i-input>
  </i-form-item>
  <i-form-item>
    <i-input v-model="form.mail"></i-input>
  </i-form-item>
</i-form>
```

Form 要用到数据校验，并在对应的 FormItem 中给出校验失败的提示，校验我们会用到一个开源库：async-validator，基本主流的组件库都是基于它做的校验。使用它很简单，只需按要求写好一个校验规则就好，比如：

```
[
  { required: true, message: '邮箱不能为空', trigger: 'blur' },
  { type: 'email', message: '邮箱格式不正确', trigger: 'blur' }
]
```

这个代表要校验的数据先判断是否为空（required: true），如果为空，则提示“邮箱不能为空”，触发校验的事件为失焦（trigger: 'blur'），如果第一条满足要求，再进行第二条的验证，判断是否为邮箱格式（type: 'email'）等等，还支持自定义校验规则。更详细的用法可以参看它的文档。

接口设计

我们先使用最新的 Vue CLI 3 创建一个空白的项目（如果你还不清楚 Vue CLI 3 的用法，需要先补习一下了，可以阅读文末的扩展阅读 1），并使用 `vue-router` 插件，同时安装好 `async-validator` 库。

在 `src/components` 下新建一个 `form` 文件夹，并初始化两个组件 `form.vue` 和 `form-item.vue`，然后初始化项目，配置路由，创建一个页面能够被访问到。

> 本节所有代码可以在 https://github.com/icarusion/vue-component-book 中查看，你可以一边看源码，一边阅读本节；也可以边阅读，边动手实现一遍，遇到问题再参考完整的源码。

第 2 节我们介绍到，编写一个 Vue.js 组件，最重要的是设计好它的接口，一个 Vue.js 组件的接口来自三个部分：props、slots、events。而 Form 和 FormItem 两个组件主要做数据校验，用不到 events。Form 的 slot 就是一系列的 FormItem，FormItem 的 slot 就是具体的表单控件，比如输入框 `<i-input>`。那主要设计的就是 props 了。

在 `Form` 组件中，定义两个 props：

- model：表单控件绑定的数据对象，在校验或重置时会访问该数据对象下对应的表单数据，类型为 Object。
- rules：表单验证规则，即上面介绍的 async-validator 所使用的校验规则，类型为 Object。

在 `FormItem` 组件中，也定义两个 props：

- label：单个表单组件的标签文本，类似原生的 `<label>` 元素，类型为 String。
- prop：对应表单域 Form 组件 model 里的字段，用于在校验或重置时访问表单组件绑定的数据，类型为 String。

定义好 props，就可以写出大概的用例了：

```
<template>
  <div>
    <i-form :model="formValidate" :rules="ruleValidate">
      <i-form-item label="用户名" prop="name">
        <i-input v-model="formValidate.name"></i-input>
      </i-form-item>
      <i-form-item label="邮箱" prop="mail">
        <i-input v-model="formValidate.mail"></i-input>
      </i-form-item>
    </i-form>
  </div>
</template>
<script>
  import iForm from '../components/form/form.vue';
  import iFormItem from '../components/form/form-item.vue';
  import iInput from '../components/input/input.vue';
```

```
  export default {
    components: { iForm, iFormItem, iInput },
    data () {
      return {
        formValidate: {
          name: '',
          mail: ''
        },
        ruleValidate: {
          name: [
            { required: true, message: '用户名不能为空', trigger: 'blur' }
          ],
          mail: [
            { required: true, message: '邮箱不能为空', trigger: 'blur' },
            { type: 'email', message: '邮箱格式不正确', trigger: 'blur' }
          ],
        }
      }
    }
  }
</script>
```

有两点需要注意的是：

1. 这里的 `<i-input>` 并不是原生的 `<input>` 输入框，而是一个特制的输入框组件，之后会介讲解的功能和代码；
2. `<i-form-item>` 的属性 `prop` 是字符串，所以它前面没有冒号（即不是 `:prop="name"`）。

当前的两个组件只是个框框，还没有实现任何功能，不过万事开头难，定义好接口，剩下的就是补全组件的逻辑，而对于使用者，知道了 props、events、slots，就已经能写出上例的使用代码了。

到此，Form 和 FormItem 的代码如下：

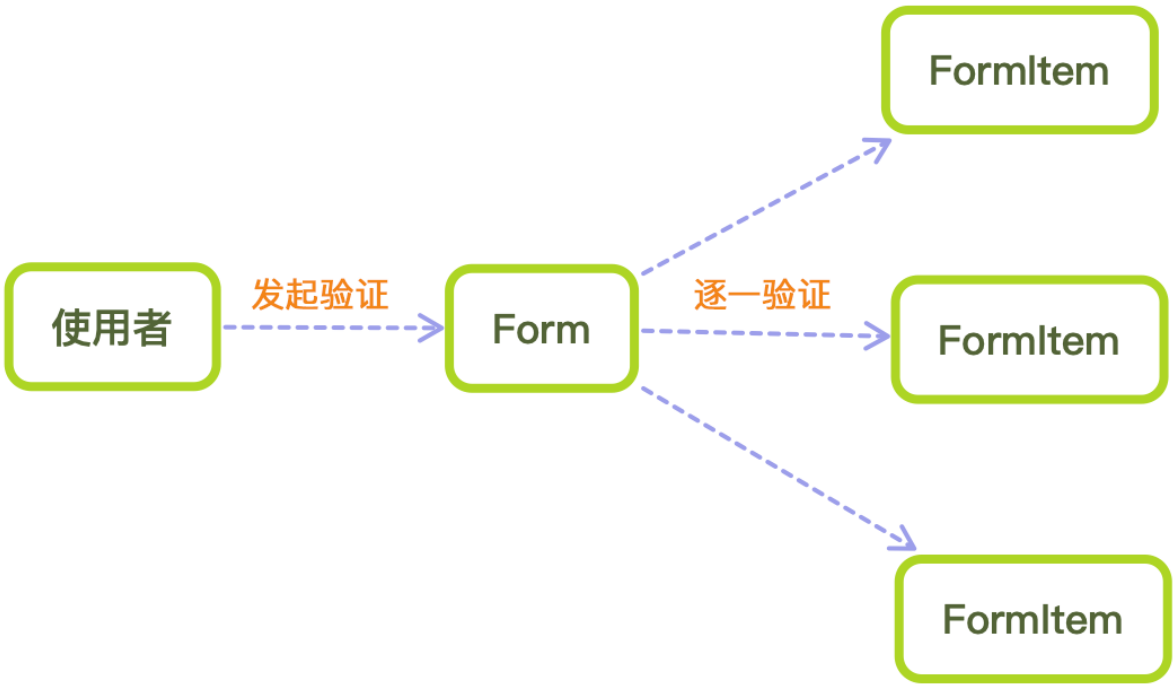
```
<!-- form.vue -->
<template>
  <form>
    <slot></slot>
  </form>
</template>
<script>
  export default {
    name: 'iForm',
    props: {
      model: {
        type: Object
      },
      rules: {
        type: Object
      }
    }
  }
</script>
```

```
<!-- form-item.vue -->
<template>
```

```
  <div>
    <label v-if="label">{{ label }}</label>
    <div>
      <slot></slot>
    </div>
  </div>
</template>
<script>
  export default {
    name: 'iFormItem',
    props: {
      label: {
        type: String,
        default: ''
      },
      prop: {
        type: String
      }
    }
  }
</script>
```

在 Form 中缓存 FormItem 实例

`Form` 组件的核心功能是数据校验，一个 Form 中包含了多个 FormItem，当点击提交按钮时，要逐一对每个 FormItem 内的表单组件校验，而校验是由使用者发起，并通过 `Form` 来调用每一个 `FormItem` 的验证方法，再将校验结果汇总后，通过 `Form` 返回出去。大致的流程如下图所示：



因为要在 Form 中逐一调用 FormItem 的验证方法，而 Form 和 FormItem 是独立的，需要预先将 FormItem 的每个实例缓存在 Form 中，这个操作就需要用到第 4 节的组件通信方法。当每个 FormItem 渲染时，将其自身（this）作为参数通过 `dispatch` 派发到 Form 组件中，然后通过一个数组缓存起来；同理当 FormItem 销毁时，将其从 Form 缓存的数组中移除。相关代码如下：

```
// form-item.vue，部分代码省略
```

```
import Emitter from '../../mixins/emitter.js';

export default {
  name: 'iFormItem',
  mixins: [ Emitter ],
  // 组件渲染时，将实例缓存在 Form 中
  mounted () {
    // 如果没有传入 prop，则无需校验，也就无需缓存
    if (this.prop) {
      this.dispatch('iForm', 'on-form-item-add', this);
    }
  },
  // 组件销毁前，将实例从 Form 的缓存中移除
  beforeDestroy () {
    this.dispatch('iForm', 'on-form-item-remove', this);
  }
}
```

注意，Vue.js 的组件渲染顺序是由内而外的，所以 FormItem 要先于 Form 渲染，在 FormItem 的 mounted 触发时，我们向 Form 派发了事件 `on-form-item-add`，并将当前 FormItem 的实例（this）传递给了 Form，而此时，Form 的 mounted 尚未触发，因为 Form 在最外层，如果在 Form 的 mounted 里监听事件，是不可以的，所以要在其 created 内监听自定义事件，Form 的 created 要先于 FormItem 的 mounted。所以 Form 的相关代码为：

```
// form.vue，部分代码省略
export default {
  name: 'iForm',
  data () {
    return {
      fields: []
    };
  },
  created () {
    this.$on('on-form-item-add', (field) => {
      if (field) this.fields.push(field);
    });
    this.$on('on-form-item-remove', (field) => {
      if (field.prop) this.fields.splice(this.fields.indexOf(field), 1);
    });
  }
}
```

定义的数据 `fields` 就是用来缓存所有 FormItem 实例的。

触发校验

Form 支持两种事件来触发校验：

- **blur**：失去焦点时触发，常见的有输入框失去焦点时触发校验；
- **change**：实时输入时触发或选择时触发，常见的有输入框实时输入时触发校验、下拉选择器选择项目时触发校验等。

以上两个事件，都是有具体的表单组件来触发的，我们先来编写一个简单的输入框组件 `i-input`。在 `components` 下新建目录 `input`，并创建文件 `input.vue`：

```
<!-- input.vue -->
<template>
  <input
         type="text"
         :value="currentValue"
         @input="handleInput"
         @blur="handleBlur"
         />
</template>
<script>
  import Emitter from '../../mixins/emitter.js';

  export default {
    name: 'iInput',
    mixins: [ Emitter ],
    props: {
      value: {
        type: String,
        default: ''
      },
    },
    data () {
      return {
        currentValue: this.value
      }
    },
    watch: {
      value (val) {
        this.currentValue = val;
      }
    },
    methods: {
      handleInput (event) {
        const value = event.target.value;
        this.currentValue = value;
        this.$emit('input', value);
        this.dispatch('iFormItem', 'on-form-change', value);
      },
      handleBlur () {
        this.dispatch('iFormItem', 'on-form-blur', this.currentValue);
      }
    }
  }
</script>
```

Input 组件中，绑定在 `<input>` 元素上的原生事件 `@input`，每当输入一个字符，都会调用句柄 `handleInput`，并通过 `dispatch` 方法向上级的 FormItem 组件派发自定义事件 `on-form-change`；同理，绑定的原生事件 `@blur` 会在 input 失焦时触发，并传递事件 `on-form-blur`。

基础组件有了，接下来要做的，是在 FormItem 中监听来自 Input 组件派发的自定义事件。这里可以在 mounted 中监听，因为你的手速远赶不上组件渲染的速度，不过在 created 中监听也是没任何问题的。相关代码如下：

```
// form-item.vue，部分代码省略
export default {
```

```
  methods: {
    setRules () {
      this.$on('on-form-blur', this.onFieldBlur);
      this.$on('on-form-change', this.onFieldChange);
    },
  },
  mounted () {
    if (this.prop) {
      this.dispatch('iForm', 'on-form-item-add', this);
      this.setRules();
    }
  }
}
```

通过调用 `setRules` 方法，监听表单组件的两个事件，并绑定了句柄函数 `onFieldBlur` 和 `onFieldChange`，分别对应 blur 和 change 两种事件类型。当 onFieldBlur 或 onFieldChange 函数触发时，就意味着 FormItem 要对**当前的数据**进行一次校验。当前的数据，指的就是通过表单域 Form 中定义的 props：model，结合当前 FormItem 定义的 props：prop 来确定的数据，可以回顾上文写过的用例。

因为 FormItem 中只定义了数据源的某个 key 名称（即属性 prop），要拿到 Form 中 model 里的数据，需要用到第 3 节的通信方法 provide / inject。所以在 Form 中，把整个实例（this）向下提供，并在 FormItem 中注入：

```
// form.vue，部分代码省略
export default {
  provide() {
    return {
      form : this
    };
  }
}
```

```
// form-item.vue，部分代码省略
export default {
  inject: ['form']
}
```

准备好这些，接着就是最核心的校验功能了。blur 和 change 事件都会触发校验，它们调用同一个方法，只是参数不同。相关代码如下：

```
// form-item.vue，部分代码省略
import AsyncValidator from 'async-validator';

export default {
  inject: ['form'],
  props: {
    prop: {
      type: String
    },
  },
  data () {
```

```
    return {
      validateState: '',  // 校验状态
      validateMessage: '',  // 校验不通过时的提示信息
    }
  },
  computed: {
    // 从 Form 的 model 中动态得到当前表单组件的数据
    fieldValue () {
      return this.form.model[this.prop];
    }
  },
  methods: {
    // 从 Form 的 rules 属性中，获取当前 FormItem 的校验规则
    getRules () {
      let formRules = this.form.rules;

      formRules = formRules ? formRules[this.prop] : [];

      return [].concat(formRules || []);
    },
    // 只支持 blur 和 change，所以过滤出符合要求的 rule 规则
    getFilteredRule (trigger) {
      const rules = this.getRules();
      return rules.filter(rule => !rule.trigger || rule.trigger.indexOf(trigger)
!== -1);
    },
    /**
     * 校验数据
     * @param trigger 校验类型
     * @param callback 回调函数
     */
    validate(trigger, callback = function () {}) {
      let rules = this.getFilteredRule(trigger);

      if (!rules || rules.length === 0) {
        return true;
      }

      // 设置状态为校验中
      this.validateState = 'validating';

      // 以下为 async-validator 库的调用方法
      let descriptor = {};
      descriptor[this.prop] = rules;

      const validator = new AsyncValidator(descriptor);
      let model = {};

      model[this.prop] = this.fieldValue;

      validator.validate(model, { firstFields: true }, errors => {
        this.validateState = !errors ? 'success' : 'error';
        this.validateMessage = errors ? errors[0].message : '';

        callback(this.validateMessage);
      });
    },
    onFieldBlur() {
```

```
      this.validate('blur');
    },
    onFieldChange() {
      this.validate('change');
    }
  }
}
```

在 FormItem 的 `validate()` 方法中，最终做了两件事：

1. 设置了当前的校验状态 `validateState` 和校验不通过提示信息 `validateMessage`（通过值为空）；
2. 将 validateMessage 通过回调 callback 传递给调用者，这里的调用者是 onFieldBlur 和 onFieldChange，它们只传入了第一个参数 `trigger`，callback 并未传入，因此也不会触发回调，而这个回调主要是给 Form 用的，因为 Form 中可以通过提交按钮一次性校验所有的 FormItem（后文会介绍）这里只是表单组件触发事件时，对当前 FormItem 做校验。

除了校验，还可以对当前数据进行重置。重置是指将表单组件的数据还原到最初绑定的值，而不是清空，因此需要预先缓存一份初始值。同时我们将校验信息也显示在模板中，并加一些样式。相关代码如下：

```
<!-- form-item.vue，部分代码省略 -->
<template>
  <div>
    <label v-if="label" :class="{ 'i-form-item-label-required': isRequired }">{{
label }}</label>
    <div>
      <slot></slot>
      <div v-if="validateState === 'error'" class="i-form-item-message">{{
validateMessage }}</div>
    </div>
  </div>
</template>
<script>
  export default {
    props: {
      label: {
        type: String,
        default: ''
      },
      prop: {
        type: String
      },
    },
    data () {
      return {
        isRequired: false,  // 是否为必填
        validateState: '',  // 校验状态
        validateMessage: '',  // 校验不通过时的提示信息
      }
    },
    mounted () {
      // 如果没有传入 prop，则无需校验，也就无需缓存
      if (this.prop) {
        this.dispatch('iForm', 'on-form-item-add', this);
```

```
        // 设置初始值，以便在重置时恢复默认值
        this.initialValue = this.fieldValue;

        this.setRules();
      }
    },
    methods: {
      setRules () {
        let rules = this.getRules();
        if (rules.length) {
          rules.every((rule) => {
            // 如果当前校验规则中有必填项，则标记出来
            this.isRequired = rule.required;
          });
        }

        this.$on('on-form-blur', this.onFieldBlur);
        this.$on('on-form-change', this.onFieldChange);
      },
      // 从 Form 的 rules 属性中，获取当前 FormItem 的校验规则
      getRules () {
        let formRules = this.form.rules;

        formRules = formRules ? formRules[this.prop] : [];

        return [].concat(formRules || []);
      },
      // 重置数据
      resetField () {
        this.validateState = '';
        this.validateMessage = '';

        this.form.model[this.prop] = this.initialValue;
      },
    }
  }
</script>
<style>
  .i-form-item-label-required:before {
    content: '*';
    color: red;
  }
  .i-form-item-message {
    color: red;
  }
</style>
```

至此，FormItem 代码已经完成，不过它只具有单独校验的功能，也就是说，只能对自己的一个表单组件验证，不能对一个表单域里的所有组件一次性全部校验。而实现全部校验和全部重置的功能，要在 Form 中完成。

上文已经介绍到，在 `Form` 组件中，预先缓存了全部的 FormItem 实例，自然也能在 Form 中调用它们。通过点击提交按钮全部校验，或点击重置按钮全部重置数据，只需要在 Form 中，逐一调用缓存的 FormItem 实例中的 `validate` 或 `resetField` 方法。相关代码如下：

```
// form.vue，部分代码省略
```

```
export default {
  data () {
    return {
      fields: []
    };
  },
  methods: {
    // 公开方法：全部重置数据
    resetFields() {
      this.fields.forEach(field => {
        field.resetField();
      });
    },
    // 公开方法：全部校验数据，支持 Promise
    validate(callback) {
      return new Promise(resolve => {
        let valid = true;
        let count = 0;
        this.fields.forEach(field => {
          field.validate('', errors => {
            if (errors) {
              valid = false;
            }
            if (++count === this.fields.length) {
              // 全部完成
              resolve(valid);
              if (typeof callback === 'function') {
                callback(valid);
              }
            }
          });
        });
      });
    }
  },
}
```

虽然说 Vue.js 的 API 只来自 prop、event、slot 这三个部分，但一些场景下，需要通过 `ref` 来访问这个组件，调用它的一些内置方法，比如上面的 `validate` 和 `resetFields` 方法，就需要使用者来主动调用。

resetFields 很简单，就是通过循环逐一调用 FormItem 的 resetField 方法来重置数据。validate 稍显复杂，它支持两种使用方法，一种是普通的回调，比如：

```
<template>
  <div>
    <i-form ref="form"></i-form>
    <button @click="handleSubmit">提交</button>
  </div>
</template>
<script>
  export default {
    methods: {
      handleSubmit () {
        this.$refs.form.validate((valid) => {
          if (valid) {
```

```
            window.alert('提交成功');
          } else {
            window.alert('表单校验失败');
          }
        })
      }
    }
  }
</script>
```

同时也支持 Promise，例如：

```
handleSubmit () {
  const validate = this.$refs.form.validate();

  validate.then((valid) => {
    if (valid) {
      window.alert('提交成功');
    } else {
      window.alert('表单校验失败');
    }
  })
}
```

在 Form 组件定义的 Promise 中，只调用了 resolve(valid)，没有调用 reject()，因此不能直接使用 `.catch()`，不过聪明的你稍作修改，肯定能够支持到！

完整的用例如下：

```
<template>
  <div>
    <h3>具有数据校验功能的表单组件——Form</h3>
    <i-form ref="form" :model="formValidate" :rules="ruleValidate">
      <i-form-item label="用户名" prop="name">
        <i-input v-model="formValidate.name"></i-input>
      </i-form-item>
      <i-form-item label="邮箱" prop="mail">
        <i-input v-model="formValidate.mail"></i-input>
      </i-form-item>
    </i-form>
    <button @click="handleSubmit">提交</button>
    <button @click="handleReset">重置</button>
  </div>
</template>
<script>
  import iForm from '../components/form/form.vue';
  import iFormItem from '../components/form/form-item.vue';
  import iInput from '../components/input/input.vue';

  export default {
    components: { iForm, iFormItem, iInput },
    data () {
      return {
        formValidate: {
```

```
          name: '',
          mail: ''
        },
        ruleValidate: {
          name: [
            { required: true, message: '用户名不能为空', trigger: 'blur' }
          ],
          mail: [
            { required: true, message: '邮箱不能为空', trigger: 'blur' },
            { type: 'email', message: '邮箱格式不正确', trigger: 'blur' }
          ],
        }
      }
    },
    methods: {
      handleSubmit () {
        this.$refs.form.validate((valid) => {
          if (valid) {
            window.alert('提交成功');
          } else {
            window.alert('表单校验失败');
          }
        })
      },
      handleReset () {
        this.$refs.form.resetFields();
      }
    }
  }
</script>
```

运行效果：

完整的示例源码可通过 GitHub 查看：

https://github.com/icarusion/vue-component-book

> 项目基于 Vue CLI 3 构建，下载安装依赖后，通过 npm run serve 可访问。

结语

组件最终的效果看起来有点 “low”，但它实现的功能却不简单。通过这个实战，你或许已经感受到本小册一开始说的，组件写到最后，都是在拼 JavaScript 功底。的确，Vue.js 组件为我们提供了一种新的代码组织形式，但归根到底，是离不开 JS 的。

这个实战，你应该对独立组件间的通信用法有进一步的认知了吧，不过，这还不是组件通信的终极方案，下一节，我们就来看看适用于任何场景的组件通信方案。

注：本节部分代码参考 iView。

扩展阅读

- 一份超级详细的Vue-cli3.0使用教程

组件的通信 3：找到任意组件实例——findComponents 系列方法

概述

前面的小节我们已经介绍了两种组件间通信的方法：provide / inject 和 dispatch / broadcast。它们有各自的使用场景和局限，比如前者多用于子组件获取父组件的状态，后者常用于父子组件间通过自定义事件通信。

本节将介绍第 3 种组件通信方法，也就是 findComponents 系列方法，它并非 Vue.js 内置，而是需要自行实现，以工具函数的形式来使用，它是一系列的函数，可以说是组件通信的终极方案。
findComponents 系列方法最终都是返回组件的实例，进而可以读取或调用该组件的数据和方法。

它适用于以下场景：

- 由一个组件，向上找到最近的指定组件；
- 由一个组件，向上找到所有的指定组件；
- 由一个组件，向下找到最近的指定组件；
- 由一个组件，向下找到所有指定的组件；
- 由一个组件，找到指定组件的兄弟组件。

5 个不同的场景，对应 5 个不同的函数，实现原理也大同小异。

实现

5 个函数的原理，都是通过递归、遍历，找到指定组件的 `name` 选项匹配的组件实例并返回。

> 本节以及后续章节，都是基于上一节的工程来完成，后续不再重复说明。
>
> 完整源码地址：https://github.com/icarusion/vue-component-book

在目录 `src` 下新建文件夹 `utils` 用来放置工具函数，并新建文件 `assist.js`，本节所有函数都在这个文件里完成，每个函数都通过 `export` 对外提供（如果你不了解 export，请查看扩展阅读1）。

向上找到最近的指定组件——findComponentUpward

先看代码：

```
// assist.js
// 由一个组件，向上找到最近的指定组件
function findComponentUpward (context, componentName) {
  let parent = context.$parent;
```

```
  let name = parent.$options.name;

  while (parent && (!name || [componentName].indexOf(name) < 0)) {
    parent = parent.$parent;
    if (parent) name = parent.$options.name;
  }
  return parent;
}
export { findComponentUpward };
```

findComponentUpward 接收两个参数，第一个是当前上下文，比如你要基于哪个组件来向上寻找，一般都是基于当前的组件，也就是传入 `this`；第二个参数是要找的组件的 `name` 。

findComponentUpward 方法会在 while 语句里不断向上覆盖当前的 `parent` 对象，通过判断组件（即 parent）的 name 与传入的 componentName 是否一致，直到直到最近的一个组件为止。

与 dispatch 不同的是，findComponentUpward 是直接拿到组件的实例，而非通过事件通知组件。比如下面的示例，有组件 A 和组件 B，A 是 B 的父组件，在 B 中获取和调用 A 中的数据和方法：

```
<!-- component-a.vue -->
<template>
  <div>
    组件 A
    <component-b></component-b>
  </div>
</template>
<script>
  import componentB from './component-b.vue';

  export default {
    name: 'componentA',
    components: { componentB },
    data () {
      return {
        name: 'Aresn'
      }
    },
    methods: {
      sayHello () {
        console.log('Hello, Vue.js');
      }
    }
  }
</script>
```

```
<!-- component-b.vue -->
<template>
  <div>
    组件 B
  </div>
</template>
<script>
  import { findComponentUpward } from '../utils/assist.js';
```

```
  export default {
    name: 'componentB',
    mounted () {
      const comA = findComponentUpward(this, 'componentA');

      if (comA) {
        console.log(comA.name);  // Aresn
        comA.sayHello();  // Hello, Vue.js
      }
    }
  }
</script>
```

使用起来很简单，只要在需要的地方调用 findComponentUpward 方法就行，第一个参数一般都是传入 this，即当前组件的上下文（实例）。

上例的 comA，保险起见，加了一层 `if (comA)` 来判断是否找到了组件 A，如果没有指定的组件而调用的话，是会报错的。

findComponentUpward 只会找到最近的一个组件实例，如果要找到全部符合要求的组件，就需要用到下面的这个方法。

向上找到所有的指定组件——findComponentsUpward

代码如下：

```
// assist.js
// 由一个组件，向上找到所有的指定组件
function findComponentsUpward (context, componentName) {
  let parents = [];
  const parent = context.$parent;

  if (parent) {
    if (parent.$options.name === componentName) parents.push(parent);
    return parents.concat(findComponentsUpward(parent, componentName));
  } else {
    return [];
  }
}
export { findComponentsUpward };
```

与 findComponentUpward 不同的是，findComponentsUpward 返回的是一个数组，包含了所有找到的组件实例（注意函数名称中多了一个“s”）。

findComponentsUpward 的使用场景较少，一般只用在递归组件里面（后面小节会介绍），因为这个函数是一直向上寻找父级（parent）的，只有递归组件的父级才是自身。事实上，iView 在使用这个方法也都是用在递归组件的场景，比如菜单组件 Menu。由于递归组件在 Vue.js 组件里面并不常用，那自然 findComponentsUpward 也不常用了。

向下找到最近的指定组件——findComponentDownward

代码如下：

```
// assist.js
```

```
// 由一个组件，向下找到最近的指定组件
function findComponentDownward (context, componentName) {
  const childrens = context.$children;
  let children = null;

  if (childrens.length) {
    for (const child of childrens) {
      const name = child.$options.name;

      if (name === componentName) {
        children = child;
        break;
      } else {
        children = findComponentDownward(child, componentName);
        if (children) break;
      }
    }
  }
  return children;
}
export { findComponentDownward };
```

`context.$children` 得到的是当前组件的全部子组件，所以需要遍历一遍，找到有没有匹配到的组件 `name`，如果没找到，继续递归找每个 $children$的 children，直到找到最近的一个为止。

来看个示例，仍然是 A、B 两个组件，A 是 B 的父组件，在 A 中找到 B：

```
<!-- component-b.vue -->
<template>
  <div>
    组件 B
  </div>
</template>
<script>
  export default {
    name: 'componentB',
    data () {
      return {
        name: 'Aresn'
      }
    },
    methods: {
      sayHello () {
        console.log('Hello, Vue.js');
      }
    }
  }
</script>
```

```
<!-- component-a.vue -->
<template>
  <div>
    组件 A
    <component-b></component-b>
```

```
  </div>
</template>
<script>
  import componentB from './component-b.vue';
  import { findComponentDownward } from '../utils/assist.js';

  export default {
    name: 'componentA',
    components: { componentB },
    mounted () {
      const comB = findComponentDownward(this, 'componentB');
      if (comB) {
        console.log(comB.name);  // Aresn
        comB.sayHello();  // Hello, Vue.js
      }
    }
  }
</script>
```

示例中的 A 和 B 是父子关系，因此也可以直接用 `ref` 来访问，但如果不是父子关系，中间间隔多代，用它就很方便了。

向下找到所有指定的组件——findComponentsDownward

如果要向下找到所有的指定组件，要用到 findComponentsDownward 函数，代码如下：

```
// assist.js
// 由一个组件，向下找到所有指定的组件
function findComponentsDownward (context, componentName) {
  return context.$children.reduce((components, child) => {
    if (child.$options.name === componentName) components.push(child);
    const foundChilds = findComponentsDownward(child, componentName);
    return components.concat(foundChilds);
  }, []);
}
export { findComponentsDownward };
```

这个函数实现的方式有很多，这里巧妙使用 `reduce` 做累加器，并用递归将找到的组件合并为一个数组并返回，代码量较少，但理解起来稍困难。

用法与 findComponentDownward 大同小异，就不再写用例了。

找到指定组件的兄弟组件——findBrothersComponents

代码如下：

```
// assist.js
// 由一个组件，找到指定组件的兄弟组件
function findBrothersComponents (context, componentName, exceptMe = true) {
  let res = context.$parent.$children.filter(item => {
    return item.$options.name === componentName;
  });
  let index = res.findIndex(item => item._uid === context._uid);
  if (exceptMe) res.splice(index, 1);
  return res;
}
export { findBrothersComponents };
```

相比其它 4 个函数，findBrothersComponents 多了一个参数 `exceptMe`，是否把本身除外，默认是 true。寻找兄弟组件的方法，是先获取 `context.$parent.$children`，也就是父组件的全部子组件，这里面当前包含了本身，所有也会有第三个参数 exceptMe。Vue.js 在渲染组件时，都会给每个组件加一个内置的属性 `_uid`，这个 _uid 是不会重复的，借此我们可以从一系列兄弟组件中把自己排除掉。

举个例子，组件 A 是组件 B 的父级，在 B 中找到所有在 A 中的兄弟组件（也就是所有在 A 中的 B 组件）：

```
<!-- component-a.vue -->
<template>
  <div>
    组件 A
    <component-b></component-b>
  </div>
</template>
<script>
  import componentB from './component-b.vue';

  export default {
    name: 'componentA',
    components: { componentB }
  }
</script>
```

```
<!-- component-b.vue -->
<template>
  <div>
    组件 B
  </div>
</template>
<script>
  import { findBrothersComponents } from '../utils/assist.js';

  export default {
    name: 'componentB',
    mounted () {
      const comsB = findBrothersComponents(this, 'componentB');
      console.log(comsB);  // ① []，空数组
    }
  }
</script>
```

在 ① 的位置，打印出的内容为空数组，原因是当前 A 中只有一个 B，而 findBrothersComponents 的第三个参数默认是 true，也就是将自己除外。如果在 A 中再写一个 B：

```
<!-- component-a.vue -->
<template>
  <div>
    组件 A
    <component-b></component-b>
    <component-b></component-b>
  </div>
</template>
```

这时就会打印出 `[VueComponent]`，有一个组件了，但要注意在控制台会打印两遍，因为在 A 中写了两个 B，而 `console.log` 是在 B 中定义的，所以两个都会执行到。如果你看懂了这里，那应该明白打印的两遍 `[VueComponent]`，分别是另一个 `<component-b>`（如果没有搞懂，要仔细琢磨琢磨哦）。

如果将 B 中 findBrothersComponents 的第三个参数设置为 false：

```
// component-b.vue
export default {
  name: 'componentB',
  mounted () {
    const comsB = findBrothersComponents(this, 'componentB', false);
    console.log(comsB);
  }
}
```

此时就会打印出 `[VueComponent, VueComponent]`，也就是包含自身了。

以上就是 5 个函数的详细介绍，get 到这 5 个，以后就再也不用担心组件通信了。

结语

只有你认真开发过 Vue.js 独立组件，才会明白这 5 个函数的强大之处。

扩展阅读

- ES6 Module 的语法

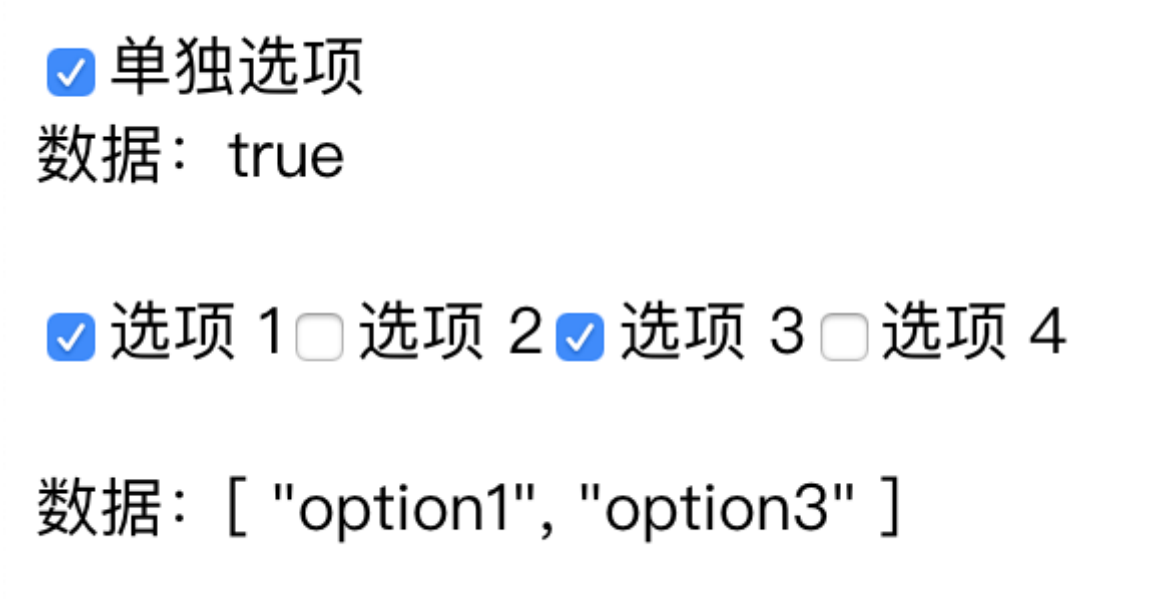
注：本节部分代码参考 iView。

实战 2：组合多选框组件——CheckboxGroup & Checkbox

在第 5 节，我们完成了具有数据校验功能的组件 Form，本小节继续开发一个新的组件——组合多选框 Checkbox。它作为基础组件，也能集成在 Form 内并应用其验证规则。

Checkbox 组件概览

多选框组件也是由两个组件组成：CheckboxGroup 和 Checkbox。单独使用时，只需要一个 Checkbox，组合使用时，两者都要用到。效果如下图所示：



单独使用，常见的场景有注册时勾选以同意注册条款，它只有一个独立的 Checkbox 组件，并且绑定一个布尔值，示例如下：

```
<template>
  <i-checkbox v-model="single">单独选项</i-checkbox>
</template>
<script>
  export default {
    data () {
      return {
        single: false
      }
    }
  }
</script>
```

而组合使用的场景就很多了，填写表单时会经常用到，它的结构如下：

```
<template>
  <i-checkbox-group v-model="multiple">
    <i-checkbox label="option1">选项 1</i-checkbox>
    <i-checkbox label="option2">选项 2</i-checkbox>
    <i-checkbox label="option3">选项 3</i-checkbox>
    <i-checkbox label="option4">选项 4</i-checkbox>
  </i-checkbox-group>
</template>
<script>
  export default {
    data () {
      return {
        multiple: ['option1', 'option3']
      }
    }
```

```
  }
</script>
```

`v-model` 用在了 CheckboxGroup 上，绑定的值为一个数组，数组的值就是内部 Checkbox 绑定的 label。

用法看起来比 Form 要简单多，不过也有两个个技术难点：

- Checkbox 要同时支持单独使用和组合使用的场景；
- CheckboxGroup 和 Checkbox 内可能嵌套其它的布局组件。

对于第一点，要在 Checkbox 初始化时判断是否父级有 CheckboxGroup，如果有就是组合使用的，否则就是单独使用。而第二点，正好可以用上一节的通信方法，很容易就能解决。

两个组件并行开发，会容易理不清逻辑，不妨我们先开发独立的 Checkbox 组件。

单独使用的 Checkbox

设计一个组件时，还是要从它的 3 个 API 入手：prop、event、slot。

因为要在 Checkbox 组件上直接使用 `v-model` 来双向绑定数据，那必不可少的一个 prop 就是 `value`，还有 event `input`，因为 v-model 本质上是一个语法糖（如果你还不清楚这种用法，可以阅读最后的扩展阅读 1）。

理论上，我们只需要给 `value` 设置为布尔值即可，也就是 true / false，不过为了扩展性，我们再定义两个 props：`trueValue` 和 `falseValue`，它们允许用户指定 `value` 用什么值来判断是否选中。因为实际开发中，数据库中并不直接保存 true / false，而是 1 / 0 或其它字符串，如果强制使用 Boolean，使用者就要再额外转换一次，这样的 API 设计不太友好。

除此之外，还需要一个 `disabled` 属性来表示是否禁用。

自定义事件 events 上文已经说了一个 `input`，用于实现 v-model 语法糖；另一个就是 `on-change`，当选中 / 取消选中时触发，用于通知父级状态发生了变化。

slot 使用默认的就好，显示辅助文本。

理清楚了 API，先来写一个基础的 `v-model` 功能，这在大部分组件中都类似。

在 `src/components` 下新建目录 `checkbox`，并新建两个文件 `checkbox.vue` 和 `checkbox-group.vue`。我们先来看 Checkbox：

```
<!-- checkbox.vue -->
<template>
  <label>
    <span>
      <input
             type="checkbox"
             :disabled="disabled"
             :checked="currentValue"
             @change="change">
    </span>
    <slot></slot>
  </label>
</template>
<script>
  export default {
    name: 'iCheckbox',
```

```
    props: {
      disabled: {
        type: Boolean,
        default: false
      },
      value: {
        type: [String, Number, Boolean],
        default: false
      },
      trueValue: {
        type: [String, Number, Boolean],
        default: true
      },
      falseValue: {
        type: [String, Number, Boolean],
        default: false
      }
    },
    data () {
      return {
        currentValue: this.value
      };
    },
    methods: {
      change (event) {
        if (this.disabled) {
          return false;
        }

        const checked = event.target.checked;
        this.currentValue = checked;

        const value = checked ? this.trueValue : this.falseValue;
        this.$emit('input', value);
        this.$emit('on-change', value);
      }
    }
  }
</script>
```

因为 `value` 被定义为 prop，它只能由父级修改，本身是不能修改的，在 `<input>` 触发 change 事件，也就是点击选择时，不能由 Checkbox 来修改这个 value，所以我们在 data 里定义了一个 `currentValue`，并把它绑定在 `<input :checked="currentValue">`，这样就可以在 Checkbox 内修改 `currentValue`。这是自定义组件使用 `v-model` 的“惯用伎俩”。

代码看起来都很简单，但有三个细节需要额外说明：

1. 选中的控件，直接使用了 `<input type="checkbox">`，而没有用 div + css 来自己实现选择的逻辑和样式，这样的好处是，使用 input 元素，你的自定义组件仍然为 html 内置的基础组件，可以使用浏览器默认的行为和快捷键，也就是说，浏览器知道这是一个选择框，而换成 div + css，浏览器可不知道这是个什么鬼。如果你觉得原生的 input 丑，没关系，是可以用 css 美化的，不过这不是本小册的重点，在此就不介绍了。
2. `<input>`、`<slot>` 都是包裹在一个 `<label>` 元素内的，这样做的好处是，当点击 `<slot>` 里的文字时，`<input>` 选框也会被触发，否则只有点击那个小框才会触发，那样不太容易选中，影响用户体验。

3. `currentValue` 仍然是布尔值（true / false），因为它是组件 Checkbox 自己使用的，对于使用者无需关心，而 value 可以是 String、Number 或 Boolean，这取决于 `trueValue` 和 `falseValue` 的定义。

现在实现的 `v-model`，只是由内而外的，也就是说，通过点击 `<input>` 选择，会通知到使用者，而使用者手动修改了 prop `value` ，Checkbox 是没有做响应的，那继续补充代码：

```
<!-- checkbox.vue，部分代码省略 -->
<script>
  export default {
    watch: {
      value (val) {
        if (val === this.trueValue || val === this.falseValue) {
          this.updateModel();
        } else {
          throw 'Value should be trueValue or falseValue.';
        }
      }
    },
    methods: {
      updateModel () {
        this.currentValue = this.value === this.trueValue;
      }
    }
  }
</script>
```

我们对 prop `value` 使用 watch 进行了监听，当父级修改它时，会调用 `updateModel` 方法，同步修改内部的 `currentValue` 。不过，不是所有的值父级都能修改的，所以用 if 条件判断了父级修改的值是否符合 trueValue / falseValue 所设置的，否则会抛错。

Checkbox 也是一个基础的表单类组件，它完全可以集成到 Form 里，所以，我们使用 Emitter 在 change 事件触发时，向 Form 派发一个事件，这样你就可以用第 5 节的 Form 组件来做数据校验了：

```
<!-- checkbox.vue，部分代码省略 -->
<script>
  import Emitter from '../../mixins/emitter.js';

  export default {
    mixins: [ Emitter ],
    methods: {
      change (event) {
        // ...
        this.$emit('input', value);
        this.$emit('on-change', value);
        this.dispatch('iFormItem', 'on-form-change', value);
      }
    },
  }
</script>
```

至此，Checkbox 已经可以单独使用了，并支持 Form 的数据校验。下面来看组合使用。

组合使用的 CheckboxGroup

> 友情提示：请先阅读 Vue.js 文档的 https://cn.vuejs.org/v2/guide/forms.html#复选框 内容。

CheckboxGroup 的 API 很简单：

- props：`value`，与 Checkbox 的类似，用于 v-model 双向绑定数据，格式为数组；
- events：`on-change`，同 Checkbox；
- slots：默认，用于放置 Checkbox。

如果写了 CheckboxGroup，那就代表你要组合使用多选框，而非单独使用，两种模式，只能用其一，而判断的依据，就是是否用了 CheckboxGroup 组件。所以在 Checkbox 组件内，我们用上一节的 `findComponentUpward` 方法判断父组件是否有 `CheckboxGroup`：

```
<!-- checkbox.vue，部分代码省略 -->
<template>
  <label>
    <span>
      <input
             v-if="group"
             type="checkbox"
             :disabled="disabled"
             :value="label"
             v-model="model"
             @change="change">
      <input
             v-else
             type="checkbox"
             :disabled="disabled"
             :checked="currentValue"
             @change="change">
    </span>
    <slot></slot>
  </label>
</template>
<script>
  import { findComponentUpward } from '../../utils/assist.js';

  export default {
    name: 'iCheckbox',
    props: {
      label: {
        type: [String, Number, Boolean]
      }
    },
    data () {
      return {
        model: [],
        group: false,
        parent: null
      };
    },
    mounted () {
      this.parent = findComponentUpward(this, 'iCheckboxGroup');

      if (this.parent) {
        this.group = true;
      }
```

```
      if (this.group) {
        this.parent.updateModel(true);
      } else {
        this.updateModel();
      }
    },
  }
</script>
```

在 mounted 时，通过 findComponentUpward 方法，来判断父级是否有 CheckboxGroup 组件，如果有，就将 `group` 置为 true，并触发 CheckboxGroup 的 `updateModel` 方法，下文会介绍它的作用。

在 template 里，我们又写了一个 `<input>` 来区分是否是 group 模式。Checkbox 的 data 里新增加的 `model` 数据，其实就是父级 CheckboxGroup 的 `value`，会在下文的 `updateModel` 方法里给 Checkbox 赋值。

Checkbox 新增的 prop： `label` 只会在组合使用时有效，结合 `model` 来使用，用法已在 Vue.js 文档中介绍了 https://cn.vuejs.org/v2/guide/forms.html#复选框。

在组合模式下，Checkbox 选中，就不用对 Form 派发事件了，应该在 CheckboxGroup 中派发，所以对 Checkbox 做最后的修改：

```
<!-- checkbox.vue，部分代码省略 -->
<script>
  export default {
    methods: {
      change (event) {
        if (this.disabled) {
          return false;
        }

        const checked = event.target.checked;
        this.currentValue = checked;

        const value = checked ? this.trueValue : this.falseValue;
        this.$emit('input', value);

        if (this.group) {
          this.parent.change(this.model);
        } else {
          this.$emit('on-change', value);
          this.dispatch('iFormItem', 'on-form-change', value);
        }
      },
      updateModel () {
        this.currentValue = this.value === this.trueValue;
      },
    },
  }
</script>
```

剩余的工作，就是完成 checkbox-gourp.vue 文件：

```
<!-- checkbox-group.vue -->
<template>
  <div>
    <slot></slot>
  </div>
</template>
<script>
  import { findComponentsDownward } from '../../utils/assist.js';
  import Emitter from '../../mixins/emitter.js';

  export default {
    name: 'iCheckboxGroup',
    mixins: [ Emitter ],
    props: {
      value: {
        type: Array,
        default () {
          return [];
        }
      }
    },
    data () {
      return {
        currentValue: this.value,
        childrens: []
      };
    },
    methods: {
      updateModel (update) {
        this.childrens = findComponentsDownward(this, 'iCheckbox');
        if (this.childrens) {
          const { value } = this;
          this.childrens.forEach(child => {
            child.model = value;

            if (update) {
              child.currentValue = value.indexOf(child.label) >= 0;
              child.group = true;
            }
          });
        }
      },
      change (data) {
        this.currentValue = data;
        this.$emit('input', data);
        this.$emit('on-change', data);
        this.dispatch('iFormItem', 'on-form-change', data);
      }
    },
    mounted () {
      this.updateModel(true);
    },
    watch: {
      value () {
        this.updateModel(true);
      }
    }
  };
```

```
</script>
```

代码很容易理解，需要介绍的就是 `updateModel` 方法。可以看到，一共有 3 个地方调用了 `updateModel`，其中两个是 CheckboxGroup 的 mounted 初始化和 watch 监听的 value 变化时调用；另一个是在 Checkbox 里的 mounted 初始化时调用。这个方法的作用就是在 CheckboxGroup 里通过 `findComponentsDownward` 方法找到所有的 Checkbox，然后把 CheckboxGroup 的 `value`，赋值给 Checkbox 的 `model`，并根据 Checkbox 的 `label`，设置一次当前 Checkbox 的选中状态。这样无论是由内而外选择，或由外向内修改数据，都是双向绑定的，而且支持动态增加 Checkbox 的数量。

以上就是组合多选组件——CheckboxGroup & Checkbox 的全部内容，不知道你是否 get 到了呢！

留两个小作业：

1. 将 CheckboxGroup 和 Checkbox 组件集成在 Form 里完成一个数据校验的示例；
2. 参考本节的代码，实现一个单选组件 Radio 和 RadioGroup。

结语

你看到的简单组件，其实都不简单。

扩展阅读

- v-model 指令在组件中怎么玩

注：本节部分代码参考 iView。

Vue 的构造器——extend 与手动挂载——$mount

本节介绍两个 Vue.js 内置但却不常用的 API——extend 和 $mount，它们经常一起使用。不常用，是因为在业务开发中，基本没有它们的用武之地，但在独立组件开发时，在一些特定的场景它们是至关重要的。

使用场景

我们在写 Vue.js 时，不论是用 CDN 的方式还是在 Webpack 里用 npm 引入的 Vue.js，都会有一个根节点，并且创建一个根实例，比如：

```
<body>
  <div id="app"></div>
</body>
<script>
  const app = new Vue({
    el: '#app'
  });
</script>
```

Webpack 也类似，一般在入口文件 main.js 里，最后会创建一个实例：

```
import Vue from 'vue';
import App from './app.vue';

new Vue({
  el: '#app',
  render: h => h(App)
});
```

因为用 Webpack 基本都是前端路由的，它的 html 里一般都只有一个根节点 `<div id="app"></div>`，其余都是通过 JavaScript 完成，也就是许多的 Vue.js 组件（每个页面也是一个组件）。

有了初始化的实例，之后所有的页面，都由 vue-router 帮我们管理，组件也都是用 `import` 导入后局部注册（也有在 main.js 全局注册的），不管哪种方式，组件（或页面）的创建过程我们是无需关心的，只是写好 `.vue` 文件并导入即可。这样的组件使用方式，有几个特点：

1. 所有的内容，都是在 `#app` 节点内渲染的；
2. 组件的模板，是事先定义好的；
3. 由于组件的特性，注册的组件只能在当前位置渲染。

比如你要使用一个组件 `<i-date-picker>`，渲染时，这个自定义标签就会被替换为组件的内容，而且在哪写的自定义标签，就在哪里被替换。换句话说，常规的组件使用方式，只能在规定的地方渲染组件，这在一些特殊场景下就比较局限了，例如：

1. 组件的模板是通过调用接口从服务端获取的，需要动态渲染组件；
2. 实现类似原生 `window.alert()` 的提示框组件，它的位置是在 `<body>` 下，而非 `<div id="app">`，并且不会通过常规的组件自定义标签的形式使用，而是像 JS 调用函数一样使用。

一般来说，在我们访问页面时，组件就已经渲染就位了，对于场景 1，组件的渲染是异步的，甚至预先不知道模板是什么。对于场景 2，其实并不陌生，在 jQuery 时代，通过操作 DOM，很容易就能实现，你可以沿用这种思路，只是这种做法不那么 Vue，既然使用 Vue.js 了，就应该用 Vue 的思路来解决问题。对于这两种场景，Vue.extend 和 vm.$mount 语法就派上用场了。

用法

上文我们说到，创建一个 Vue 实例时，都会有一个选项 `el`，来指定实例的根节点，如果不写 `el` 选项，那组件就处于未挂载状态。`Vue.extend` 的作用，就是基于 Vue 构造器，创建一个“子类”，它的参数跟 `new Vue` 的基本一样，但 `data` 要跟组件一样，是个函数，再配合 `$mount`，就可以让组件渲染，并且挂载到任意指定的节点上，比如 body。

比如上文的场景，就可以这样写：

```
import Vue from 'vue';

const AlertComponent = Vue.extend({
  template: '<div>{{ message }}</div>',
  data () {
    return {
      message: 'Hello, Aresn'
    };
  },
});
```

这一步，我们创建了一个构造器，这个过程就可以解决异步获取 template 模板的问题，下面要手动渲染组件，并把它挂载到 body 下：

```
const component = new AlertComponent().$mount();
```

这一步，我们调用了 `$mount` 方法对组件进行了手动渲染，但它仅仅是被渲染好了，并没有挂载到节点上，也就显示不了组件。此时的 `component` 已经是一个标准的 Vue 组件实例，因此它的 `$el` 属性也可以被访问：

```
document.body.appendChild(component.$el);
```

当然，除了 body，你还可以挂载到其它节点上。

`$mount` 也有一些快捷的挂载方式，以下两种都是可以的：

```
// 在 $mount 里写参数来指定挂载的节点
new AlertComponent().$mount('#app');
// 不用 $mount，直接在创建实例时指定 el 选项
new AlertComponent({ el: '#app' });
```

实现同样的效果，除了用 extend 外，也可以直接创建 Vue 实例，并且用一个 Render 函数来渲染一个 .vue 文件：

```
import Vue from 'vue';
import Notification from './notification.vue';

const props = {};  // 这里可以传入一些组件的 props 选项

const Instance = new Vue({
  render (h) {
    return h(Notification, {
      props: props
    });
  }
});

const component = Instance.$mount();
document.body.appendChild(component.$el);
```

这样既可以使用 .vue 来写复杂的组件（毕竟在 template 里堆字符串很痛苦），还可以根据需要传入适当的 props。渲染后，如果想操作 Render 的 `Notification` 实例，也是很简单的：

```
const notification = Instance.$children[0];
```

因为 Instance 下只 Render 了 Notification 一个子组件，所以可以用 `$children[0]` 访问到。

> 如果你还不理解这样做的目的，没有关系，后面小节的两个实战你会感受到它的用武之地。

需要注意的是，我们是用 `$mount` 手动渲染的组件，如果要销毁，也要用 `$destroy` 来手动销毁实例，必要时，也可以用 `removeChild` 把节点从 DOM 中移除。

结语

这两个 API 并不难理解，只是不常使用罢了，因为多数情况下，我们只关注在业务层，并使用现成的组件库。

使用 Vue.js 也有二八原则，即 80% 的人看过 Vue.js 文档教程篇，20% 的人看过 Vue.js 文档 API。

下一节，我们来做点有趣的东西。

扩展阅读

- 聊聊 Vue.js 的 template 编译

实战 3：动态渲染 .vue 文件的组件——Display

你可能用过 jsfiddle 或 jsbin 之类的网站，在里面你可以用 CDN 的形式引入 Vue.js，然后在线写示例，实时运行，比如下面这个例子：

https://jsfiddle.net/c87yh92v/

不过，这类网站主要是一个 html，里面包含 js、css 部分，渲染侧是用 iframe 嵌入你编写的 html，并实时更新。在这些网站写示例，是不能直接写 `.vue` 文件的，因为没法进行编译。

再来看另一个网站 iView Run（之前小节也有提到），它是能够在线编写一个标准的 `.vue` 文件，并及时渲染的，它也预置了 iView 环境，你可以使用 iView 组件库全部的组件。本小节，我们就来实现这样一个能够动态渲染 .vue 文件的 `Display` 组件，当然，用到的核心技术就是上一节的 `extend` 和 `$mount`。

接口设计

一个常规的 `.vue` 文件一般都会包含 3 个部分：

- `<template>`：组件的模板；
- `<script>`：组件的选项，不包含 `el`；
- `<style>`：CSS 样式。

回忆一下用 `extend` 来构造一个组件实例，它的选项 `template` 其实就是上面 `<template>` 的内容，其余选项对应的是 `<script>`，样式 `<style>` 事实上与 Vue.js 无关，我们可以先不管。这样的话，当拿到一个 .vue 的文件（整体其实是字符串），只需要把 `<template>`、`<script>`、`<style>` 使用正则分割，把对应的部分传递给 extend 创建的实例就可以。

Display 是一个功能型的组件，没有交互和事件，只需要一个 prop：code 将 .vue 的内容传递过来，其余工作都是在组件内完成的，这对使用者很友好。当然，你也可以设计成三个 props，分别对应 html、js、css，那分割的工作就要使用者来完成。出于使用者优先原则，苦活累活当然是在组件内完成了，因此推荐第一个方案。

实现

在 `src/components` 目录下创建 `display` 目录，并新建 `display.vue` 文件，基本结构如下：

```
<!-- display.vue -->
<template>
  <div ref="display"></div>
</template>
<script>
  export default {
    props: {
      code: {
        type: String,
        default: ''
      }
    },
    data () {
      return {
        html: '',
        js: '',
        css: ''
      }
    },
  }
</script>
```

父级传递 `code` 后，将其分割，并保存在 data 的 html、js、css 中，后续使用。

我们使用正则，基于 `<>` 和 `</>` 的特性进行分割：

```
// display.vue，部分代码省略
export default {
  methods: {
    getSource (source, type) {
      const regex = new RegExp(`<${type}[^>]*>`);
      let openingTag = source.match(regex);

      if (!openingTag) return '';
      else openingTag = openingTag[0];

      return source.slice(source.indexOf(openingTag) + openingTag.length,
source.lastIndexOf(`</${type}>`));
    },
    splitCode () {
      const script = this.getSource(this.code, 'script').replace(/export
default/, 'return ');
      const style = this.getSource(this.code, 'style');
      const template = '<div id="app">' + this.getSource(this.code, 'template')
+ '</div>';

      this.js = script;
      this.css = style;
      this.html = template;
    },
  }
}
```

getSource 方法接收两个参数：

- source：.vue 文件代码，即 props: code；
- type：分割的部分，也就是 template、script、style。

分割后，返回的内容不再包含 `<template>` 等标签，直接是对应的内容，在 splitCode 方法中，把分割好的代码分别赋值给 data 中声明的 html、js、css。有两个细节需要注意：

1. .vue 的 `<script>` 部分一般都是以 `export default` 开始的，可以看到在 splitCode 方法中将它替换为了 `return`，这个在后文会做解释，当前只要注意，我们分割完的代码，仍然是字符串；
2. 在分割的 `<template>` 外层套了一个 `<div id="app">`，这是为了容错，有时使用者传递的 `code` 可能会忘记在外层包一个节点，没有根节点的组件，是会报错的。

准备好这些基础工作后，就可以用 `extend` 渲染组件了，在这之前，我们先思考一个问题：上文说到，当前的 `this.js` 是字符串，而 extend 接收的选项可不是字符串，而是一个对象类型，那就要先把 this.js 转为一个对象。

不卖关子，来介绍 `new Function` 用法，先看个示例：

```
const sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 6)); // 8
```

new Function 的语法：

```
new Function ([arg1[, arg2[, ...argN]],] functionBody)
```

arg1, arg2, ... argN 是被函数使用的参数名称，**functionBody** 是一个含有包括函数定义的 JavaScript 语句的**字符串**。也就是说，示例中的字符串 `return a + b` 被当做语句执行了。

上文说到，`this.js` 中是将 *export default* 替换为 *return* 的，如果将 this.js 传入 new Function 里，那么 this.js 就执行了，这时因为有 return，返回的就是一个对象类型的 this.js 了。

如果你还不是很理解 new Function，可以到文末的扩展阅读进一步了解。除了 new Function，你熟悉的 `eval` 函数也可以使用，它与 new Function 功能类似。

知道了这些，下面的内容就容易理解了：

```
<!-- display.vue，部分代码省略 -->
<template>
  <div ref="display"></div>
</template>
<script>
  import Vue from 'vue';

  export default {
    data () {
      return {
        component: null
      }
    },
    methods: {
      renderCode () {
        this.splitCode();

        if (this.html !== '' && this.js !== '') {
```

```
          const parseStrToFunc = new Function(this.js)();

          parseStrToFunc.template =  this.html;
          const Component = Vue.extend( parseStrToFunc );
          this.component = new Component().$mount();

          this.$refs.display.appendChild(this.component.$el);
        }
      }
    },
    mounted () {
      this.renderCode();
    }
  }
</script>
```

extend 构造的实例通过 $mount 渲染后，挂载到了组件唯一的一个节点 `<div ref="display">` 上。

现在 html 和 js 都有了，还剩下 css。加载 css 没有什么奇技淫巧，就是创建一个 `<style>` 标签，然后把 css 写进去，再插入到页面的 `<head>` 中，这样 css 就被浏览器解析了。为了便于后面在 `this.code` 变化或组件销毁时移除动态创建的 `<style>` 标签，我们给每个 style 标签加一个随机 id 用于标识。

在 `src/utils` 目录下新建 `random_str.js` 文件，并写入以下内容：

```
// 生成随机字符串
export default function (len = 32) {
  const $chars =
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890';
  const maxPos = $chars.length;
  let str = '';
  for (let i = 0; i < len; i++) {
    str += $chars.charAt(Math.floor(Math.random() * maxPos));
  }
  return str;
}
```

不难理解，这个方法是从指定的 a-zA-Z0-9 中随机生成 32 位的字符串。

补全 renderCode 方法：

```
// display.vue，部分代码省略
import randomStr from '../../utils/random_str.js';

export default {
  data () {
    return {
      id: randomStr()
    }
  },
  methods: {
    renderCode () {
      if (this.html !== '' && this.js !== '') {
        // ...
        if (this.css !== '') {
```

```
          const style = document.createElement('style');
          style.type = 'text/css';
          style.id = this.id;
          style.innerHTML = this.css;
          document.getElementsByTagName('head')[0].appendChild(style);
        }
      }
    }
  }
}
```

当 Display 组件销毁时，也要手动销毁 extend 创建的实例以及上面的 css：

```
// display.vue，部分代码省略
export default {
  methods: {
    destroyCode () {
      const $target = document.getElementById(this.id);
      if ($target) $target.parentNode.removeChild($target);

      if (this.component) {
        this.$refs.display.removeChild(this.component.$el);
        this.component.$destroy();
        this.component = null;
      }
    }
  },
  beforeDestroy () {
    this.destroyCode();
  }
}
```

当 `this.code` 更新时，整个过程要重新来一次，所以要对 `code` 进行 watch 监听：

```
// display.vue，部分代码省略
export default {
  watch: {
    code () {
      this.destroyCode();
      this.renderCode();
    }
  }
}
```

以上就是 Display 组件的所有内容。

使用

新建一条路由，并在 `src/views` 下新建页面 `display.vue` 来使用 Display 组件：

```
<!-- src/views/display.vue -->
<template>
```

```
  <div>
    <h3>动态渲染 .vue 文件的组件-- Display</h3>

    <i-display :code="code"></i-display>
  </div>
</template>
<script>
  import iDisplay from '../components/display/display.vue';
  import defaultCode from './default-code.js';

  export default {
    components: { iDisplay },
    data () {
      return {
        code: defaultCode
      }
    }
  }
</script>
```

```
// src/views/default-code.js
const code =
`<template>
    <div>
        <input v-model="message">
        {{ message }}
    </div>
</template>
<script>
    export default {
        data () {
            return {
                message: ''
            }
        }
    }
</script>`;

export default code;
```

如果使用的是 Vue CLI 3 默认的配置，直接运行时，会抛出下面的错误：

> [Vue warn]: You are using the runtime-only build of Vue where the template compiler is not available. Either pre-compile the templates into render functions, or use the compiler-included build.

这涉及到另一个知识点，就是 Vue.js 的版本。在使用 Vue.js 2 时，有独立构建（standalone）和运行时构建（runtime-only）两种版本可供选择，详细的介绍请阅读文末扩展阅读 2。

Vue CLI 3 默认使用了 vue.runtime.js，它不允许编译 template 模板，因为我们在 Vue.extend 构造实例时，用了 `template` 选项，所以会报错。解决方案有两种，一是手动将 template 改写为 Render 函数，但这成本太高；另一种是对 Vue CLI 3 创建的工程做简单的配置。我们使用后者。

在项目根目录，新建文件 `vue.config.js`：

```
module.exports = {
  runtimeCompiler: true
};
```

它的作用是，是否使用包含运行时编译器的 Vue 构建版本。设置为 `true` 后就可以在 Vue 组件中使用 `template` 选项了，但是应用额外增加 10kb 左右（还好吧）。

加了这个配置，报错就消失了，组件也能正常显示。

以上就是 Display 组件所有的内容，如果你感兴趣，可以把它进一步封装，做成 iView Run 这样的产品。

结语

这个小小的 Display 组件，能做的事还有很多，比如要写一套 Vue 组件库的文档，传统方法是在开发环境写一个个的 .vue 文件，然后编译打包、上传资源、上线，如果要修改，哪怕一个标点符号，都要重新编译打包、上传资源、上线。有了 Display 组件，只需要提供一个服务来在线修改文档的 .vue，就能实时更新，不用打包、上传、上线。

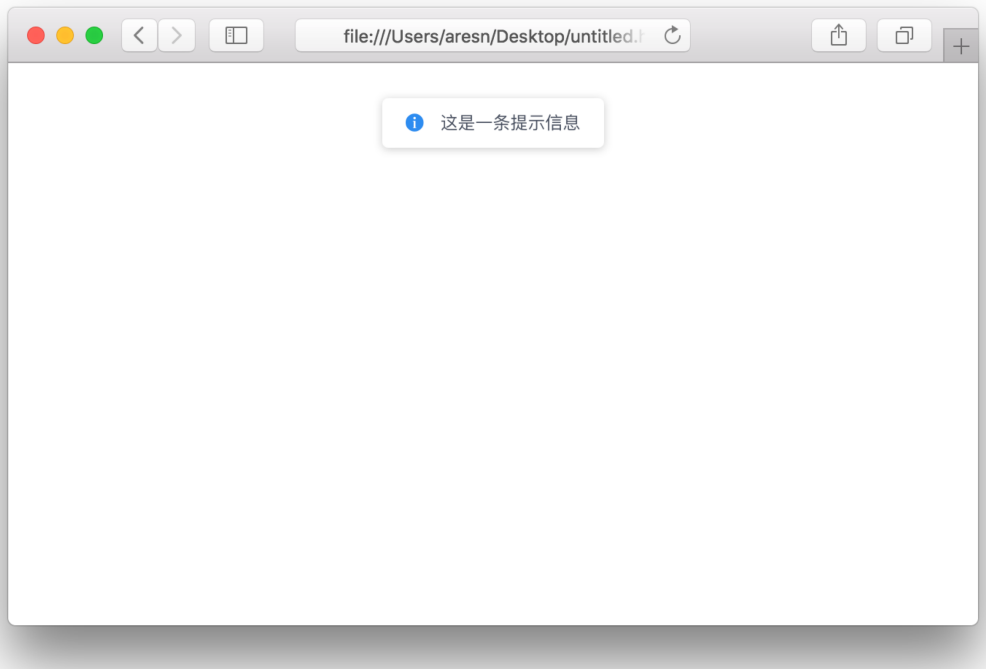
还有一点很重要的是，可以看到，在 iView Run 里，默认是直接可以写 iView 组件库的全部组件，并没有额外引入，这是因为 Display 所在的工程，已经将 iView 安装在了全局，Vue.extend 在构造实例时，已经可以使用全局安装的插件了，如果你还全局安装了其它插件，比如 axios，都是可以直接使用的。

扩展阅读

- new Function
- Vue.js 2.0 独立构建和运行时构建的区别

实战 4：全局提示组件——$Alert

有一种 Vue.js 组件，它不同于常规的组件，但组件结构本身很简单，比如下面的全局提示组件：



实现这样一个组件并不难，只需要简单的几行 div 和 css，但使用者可能要这样来显示组件：

```
<template>
  <div>
    <Alert v-if="show">这是一条提示信息</Alert>
    <button @click="show = true">显示</button>
  </div>
</template>
<script>
  import Alert from '../component/alert.vue';

  export default {
    components: { Alert },
    data () {
      return {
        show: false
      }
    }
  }
</script>
```

这样的用法，有以下缺点：

- 每个使用的地方，都得注册组件；
- 需要预先将 `<Alert>` 放置在模板中；
- 需要额外的 data 来控制 Alert 的显示状态；
- Alert 的位置，是在当前组件位置，并非在 body 下，有可能会被其它组件遮挡。

总之对使用者来说是很不友好的，那怎样才能优雅地实现这样一个组件呢？事实上，原生的 JavaScript 早已给出了答案：

```
// 全局提示
window.alert('这是一条提示信息');
// 二次确认
const confirm = window.confirm('确认删除吗？');
if (confirm) {
  // ok
} else {
  // cancel
}
```

所以，结论是：我们需要一个能用 JavaScript 调用组件的 API。

如果你使用过 iView 之类的组件库，一定对它内置的 $Message$、Notice、$Modal$等组件很熟悉，本节就来开发一个全局通知组件——Alert。

1/3 先把组件写好

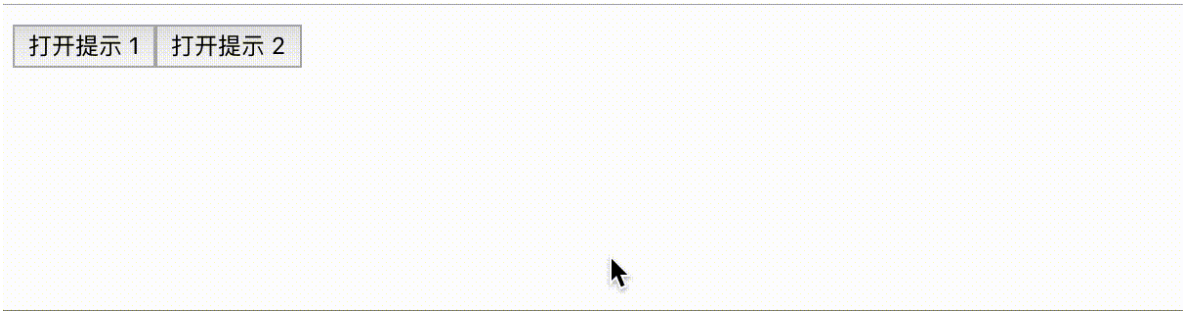
我们期望最终的 API 是这样的：

```
methods: {
  handleShow () {
    this.$Alert({
      content: '这是一条提示信息',
      duration: 3
    })
  }
}
```

`this.$Alert` 可以在任何位置调用，无需单独引入。该方法接收两个参数：

- content：提示内容；
- duration：持续时间，单位秒，默认 1.5 秒，到时间自动消失。

最终效果如下：



我们从最简单的入手，不考虑其它，先写一个基本的 Alert 组件。

在 `src/component` 下新建 `alert` 目录，并创建文件 `alert.vue`：

通知可以是多个，我们用一个数组 `notices` 来管理每条通知：

```
<!-- alert.vue -->
<template>
  <div class="alert">
    <div class="alert-main" v-for="item in notices" :key="item.name">
      <div class="alert-content">{{ item.content }}</div>
    </div>
  </div>
</template>
<script>
  export default {
    data () {
      return {
        notices: []
      }
    }
  }
</script>
<style>
  .alert{
    position: fixed;
    width: 100%;
    top: 16px;
    left: 0;
    text-align: center;
    pointer-events: none;
  }
```

```
  .alert-content{
    display: inline-block;
    padding: 8px 16px;
    background: #fff;
    border-radius: 3px;
    box-shadow: 0 1px 6px rgba(0, 0, 0, .2);
    margin-bottom: 8px;
  }
</style>
```

Alert 组件不同于常规的组件使用方式，它最终是通过 JS 来调用的，因此组件不用预留 props 和 events 接口。

接下来，只要给数组 `notices` 增加数据，这个提示组件就能显示内容了，我们先假设，最终会通过 JS 调用 Alert 的一个方法 `add`，并将 `content` 和 `duration` 传入进来：

```
<!-- alert.vue，部分代码省略 -->
<script>
  let seed = 0;

  function getUuid() {
    return 'alert_' + (seed++);
  }

  export default {
    data () {
      return {
        notices: []
      }
    },
    methods: {
      add (notice) {
        const name = getUuid();

        let _notice = Object.assign({
          name: name
        }, notice);

        this.notices.push(_notice);

        // 定时移除，单位：秒
        const duration = notice.duration;
        setTimeout(() => {
          this.remove(name);
        }, duration * 1000);
      },
      remove (name) {
        const notices = this.notices;

        for (let i = 0; i < notices.length; i++) {
          if (notices[i].name === name) {
            this.notices.splice(i, 1);
            break;
          }
        }
      }
```

```
    }
  }
</script>
```

在 `add` 方法中，给每一条传进来的提示数据，加了一个不重复的 `name` 字段来标识，并通过 `setTimeout` 创建了一个计时器，当到达指定的 `duration` 持续时间后，调用 `remove` 方法，将对应 `name` 的那条提示信息找到，并从数组中移除。

由这个思路，Alert 组件就可以无限扩展，只要在 add 方法中传递更多的参数，就能支持更复杂的组件，比如是否显示手动关闭按钮、确定 / 取消按钮，甚至传入一个 Render 函数都可以，完成本例后，读者可以尝试"改造"。

2/3 实例化封装

这一步，我们对 Alert 组件进一步封装，让它能够实例化，而不是常规的组件使用方法。实例化组件我们在第 8 节中介绍过，可以使用 Vue.extend 或 new Vue，然后用 $mount 挂载到 body 节点下。

在 `src/components/alert` 目录下新建 `notification.js` 文件：

```
// notification.js
import Alert from './alert.vue';
import Vue from 'vue';

Alert.newInstance = properties => {
  const props = properties || {};

  const Instance = new Vue({
    data: props,
    render (h) {
      return h(Alert, {
        props: props
      });
    }
  });

  const component = Instance.$mount();
  document.body.appendChild(component.$el);

  const alert = Instance.$children[0];

  return {
    add (noticeProps) {
      alert.add(noticeProps);
    },
    remove (name) {
      alert.remove(name);
    }
  }
};

export default Alert;
```

notification.js 并不是最终的文件，它只是对 alert.vue 添加了一个方法 `newInstance`。虽然 alert.vue 包含了 template、script、style 三个标签，并不是一个 JS 对象，那怎么能够给它扩展一个方法 `newInstance` 呢？事实上，alert.vue 会被 Webpack 的 vue-loader 编译，把 template 编译为 Render 函数，最终就会成为一个 JS 对象，自然可以对它进行扩展。

Alert 组件没有任何 props，这里在 Render Alert 组件时，还是给它加了 props，当然，这里的 props 是空对象 {}，而且即使传了内容，也不起作用。这样做的目的还是为了扩展性，如果要在 Alert 上添加 props 来支持更多特性，是要在这里传入的。不过话说回来，因为能拿到 Alert 实例，用 data 或 props 都是可以的。

在第 8 节已经解释过，`const alert = Instance.$children[0];`，这里的 alert 就是 Render 的 Alert 组件实例。在 `newInstance` 里，使用闭包暴露了两个方法 `add` 和 `remove`。这里的 add 和 remove 可不是 alert.vue 里的 add 和 remove，它们只是名字一样。

3/3 入口

最后要做的，就是调用 notification.js 创建实例，并通过 `add` 把数据传递过去，这是组件开发的最后一步，也是最终的入口。在 `src/component/alert` 下创建文件 `alert.js`：

```
// alert.js
import Notification from './notification.js';

let messageInstance;

function getMessageInstance () {
  messageInstance = messageInstance || Notification.newInstance();
  return messageInstance;
}

function notice({ duration = 1.5, content = '' }) {
  let instance = getMessageInstance();

  instance.add({
    content: content,
    duration: duration
  });
}

export default {
  info (options) {
    return notice(options);
  }
}
```

`getMessageInstance` 函数用来获取实例，它不会重复创建，如果 messageInstance 已经存在，就直接返回了，只在第一次调用 Notification 的 `newInstance` 时来创建实例。

alert.js 对外提供了一个方法 `info`，如果需要各种显示效果，比如成功的、失败的、警告的，可以在 info 下面提供更多的方法，比如 success、fail、warning 等，并传递不同参数让 Alert.vue 知道显示哪种状态的图标。本例因为只有一个 info，事实上也可以省略掉，直接导出一个默认的函数，这样在调用时，就不用 `this.$Alert.info()` 了，直接 `this.$Alert()`。

来看一下显示一个信息提示组件的流程：

最后把 alert.js 作为插件注册到 Vue 里就行，在入口文件 `src/main.js` 中，通过 `prototype` 给 Vue 添加一个实例方法：

```
// src/main.js
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import Alert from '../src/components/alert/alert.js'

Vue.config.productionTip = false

Vue.prototype.$Alert = Alert

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

这样在项目任何地方，都可以通过 `this.$Alert` 来调用 Alert 组件了，我们创建一个 alert 的路由，并在 `src/views` 下创建页面 `alert.vue`：

```
<!-- src/views/alert.vue -->
<template>
  <div>
    <button @click="handleOpen1">打开提示 1</button>
    <button @click="handleOpen2">打开提示 2</button>
  </div>
</template>
<script>
  export default {
    methods: {
      handleOpen1 () {
        this.$Alert.info({
          content: '我是提示信息 1'
        });
      },
      handleOpen2 () {
        this.$Alert.info({
          content: '我是提示信息 2',
          duration: 3
        });
      }
    }
  }
</script>
```

`duration` 如果不传入，默认是 1.5 秒。

以上就是全局通知组件的全部内容。

友情提示

本示例算是一个 MVP（最小化可行方案），要开发一个完善的全局通知组件，还需要更多可维护性和功能性的设计，但离不开本例的设计思路。以下几点是同类组件中值得注意的：

1. Alert.vue 的最外层是有一个 .alert 节点的，它会在第一次调用 `$Alert` 时，在 body 下创建，因为不在 `<router-view>` 内，它不受路由的影响，也就是说一经创建，除非刷新页面，这个节点是不会消失的，所以在 alert.vue 的设计中，并没有主动销毁这个组件，而是维护了一个子节点数组 `notices`。
2. .alert 节点是 `position: fixed` 固定的，因此要合理设计它的 `z-index`，否则可能被其它节点遮挡。
3. notification.js 和 alert.vue 是可以复用的，如果还要开发其它同类的组件，比如二次确认组件 `$Confirm`，只需要再写一个入口 `confirm.js`，并将 `alert.vue` 进一步封装，将 `notices` 数组的循环体写为一个新的组件，通过配置来决定是渲染 Alert 还是 Confirm，这在可维护性上是友好的。
4. 在 notification.js 的 new Vue 时，使用了 Render 函数来渲染 alert.vue，这是因为使用 template 在 runtime 的 Vue.js 版本下是会报错的。
5. 本例的 content 只能是字符串，如果要显示自定义的内容，除了用 `v-html` 指令，也能用 Functional Render（之后章节会介绍）。

结语

Vue.js 的精髓是组件，组件的精髓是 JavaScript。将 JavaScript 开发中的技巧结合 Vue.js 组件，就能玩出不一样的东西。

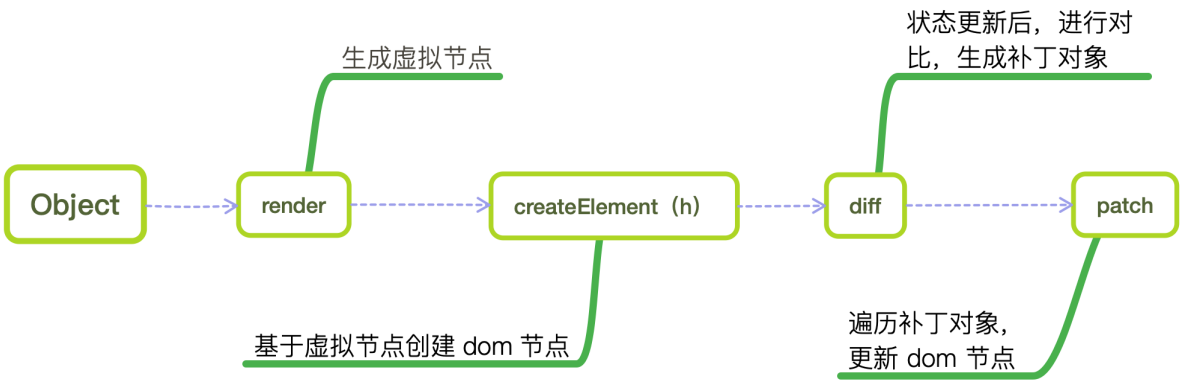
注：本节部分代码参考 iView。

更灵活的组件：Render 函数与 Functional Render

Vue.js 2.x 与 Vue.js 1.x 最大的区别就在于 2.x 使用了 Virtual DOM（虚拟 DOM）来更新 DOM 节点，提升渲染性能。

一般来说，我们写 Vue.js 组件，模板都是写在 `<template>` 内的，但它并不是最终呈现的内容，template 只是一种对开发者友好的语法，能够一眼看到 DOM 节点，容易维护，在 Vue.js 编译阶段，会解析为 Virtual DOM。

与 DOM 操作相比，Virtual DOM 是基于 JavaScript 计算的，所以开销会小很多。下图演示了 Virtual DOM 运行的过程：



正常的 DOM 节点在 HTML 中是这样的：

```
<div id="main">
  <p>文本内容</p>
  <p>文本内容</p>
</div>
```

用 Virtual DOM 创建的 JavaScript 对象一般会是这样的：

```
const vNode = {
  tag: 'div',
  attributes: {
    id: 'main'
  },
  children: [
    // p 节点
  ]
}
```

vNode 对象通过一些特定的选项描述了真实的 DOM 结构。

在 Vue.js 中，对于大部分场景，使用 template 足以应付，但如果想完全发挥 JavaScript 的编程能力，或在一些特定场景下（后文介绍），需要使用 Vue.js 的 Render 函数。

Render 函数

正如上文介绍的 Virtual DOM 示例一样，Vue.js 的 Render 函数也是类似的语法，需要使用一些特定的选项，将 template 的内容改写成一个 JavaScript 对象。

对于初级前端工程师，或想快速建站的需求，直接使用 Render 函数开发 Vue.js 组件是要比 template 困难的，原因在于 Render 函数返回的是一个 JS 对象，没有传统 DOM 的层级关系，配合上 if、else、for 等语句，将节点拆分成不同 JS 对象再组装，如果模板复杂，那一个 Render 函数是难读且难维护的。所以，绝大部分组件开发和业务开发，我们直接使用 template 语法就可以了，并不需要特意使用 Render 函数，那样只会增加负担，同时也放弃了 Vue.js 最大的优势（React 无 template 语法）。

很多学习 Vue.js 的开发者在遇到 Render 函数时都有点"躲避"，或直接放弃这部分，这并没有问题，因为不用 Render 函数，照样可以写出优秀的 Vue.js 程序。不过，Render 函数并没有想象中的那么复杂，只是配置项特别多，一时难以记住，但归根到底，Render 函数只有 3 个参数。

来看一组 template 和 Render 写法的对照：

```
<template>
  <div id="main" class="container" style="color: red">
    <p v-if="show">内容 1</p>
    <p v-else>内容 2</p>
  </div>
</template>
<script>
  export default {
    data () {
      return {
        show: false
      }
    }
  }
</script>
```

```
export default {
  data () {
    return {
      show: false
    }
  },
  render: (h) => {
    let childNode;
    if (this.show) {
      childNode = h('p', '内容 1');
    } else {
      childNode = h('p', '内容 2');
    }

    return h('div', {
      attrs: {
        id: 'main'
      },
      class: {
        container: true
      },
      style: {
        color: 'red'
      }
    }, [childNode]);
  }
}
```

这里的 `h`，即 `createElement`，是 Render 函数的核心。可以看到，template 中的 **v-if / v-else** 等指令，都被 JS 的 **if / else** 替代了，那 **v-for** 自然也会被 **for** 语句替代。

h 有 3 个参数，分别是：

1. 要渲染的元素或组件，可以是一个 html 标签、组件选项或一个函数（不常用），该参数为必填项。示例：

   ```
   // 1. html 标签
   h('div');
   // 2. 组件选项
   import DatePicker from '../component/date-picker.vue';
   h(DatePicker);
   ```

2. 对应属性的数据对象，比如组件的 props、元素的 class、绑定的事件、slot、自定义指令等，该参数是可选的，上文所说的 Render 配置项多，指的就是这个参数。该参数的完整配置和示例，可以到 Vue.js 的文档查看，没必要全部记住，用到时查阅就好：[createElement 参数]。

3. 子节点，可选，String 或 Array，它同样是一个 h。示例：

```
[
  '内容',
  h('p', '内容'),
  h(Component, {
    props: {
      someProp: 'foo'
    }
  })
]
```

约束

所有的组件树中，如果 vNode 是组件或含有组件的 slot，那么 vNode 必须唯一。以下两个示例都是**错误**的：

```
// 局部声明组件
const Child = {
  render: (h) => {
    return h('p', 'text');
  }
}

export default {
  render: (h) => {
    // 创建一个子节点，使用组件 Child
    const ChildNode = h(Child);

    return h('div', [
      ChildNode,
      ChildNode
    ]);
  }
}
```

```
{
  render: (h) => {
    return h('div', [
      this.$slots.default,
      this.$slots.default
    ])
  }
}
```

重复渲染多个组件或元素，可以通过一个循环和工厂函数来解决：

```
const Child = {
  render: (h) => {
    return h('p', 'text');
  }
}

export default {
```

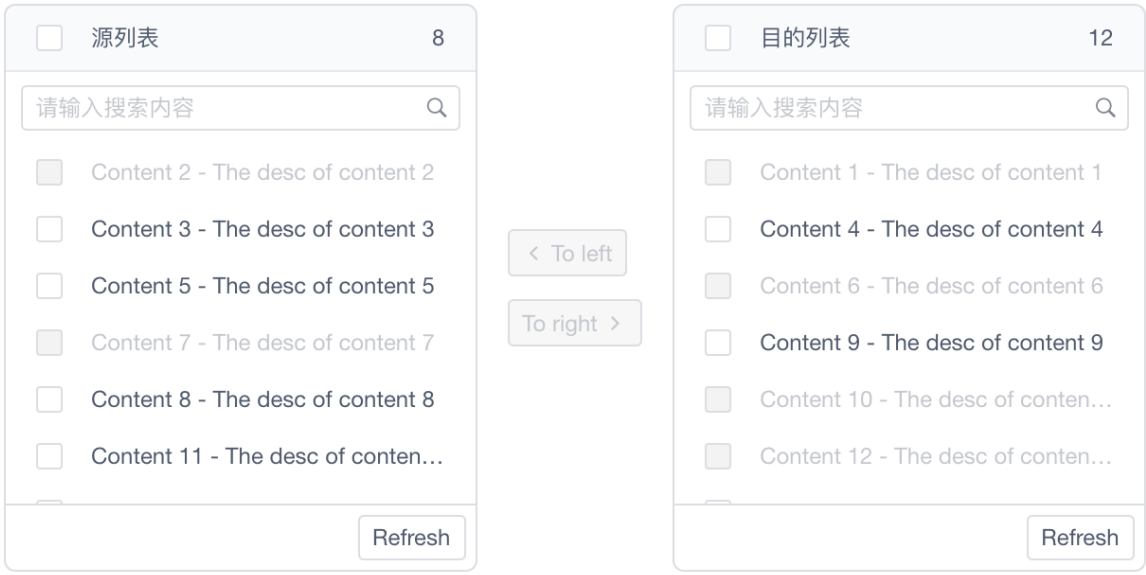
```
  render: (h) => {
    const children = Array.apply(null, {
      length: 5
    }).map(() => {
      return h(Child);
    });
    return h('div', children);
  }
}
```

对于含有组件的 slot，复用比较复杂，需要将 slot 的每个子节点都克隆一份，例如：

```
{
  render: (h) => {
    function cloneVNode (vnode) {
      // 递归遍历所有子节点，并克隆
      const clonedChildren = vnode.children && vnode.children.map(vnode =>
cloneVNode(vnode));
      const cloned = h(vnode.tag, vnode.data, clonedChildren);
      cloned.text = vnode.text;
      cloned.isComment = vnode.isComment;
      cloned.componentOptions = vnode.componentOptions;
      cloned.elm = vnode.elm;
      cloned.context = vnode.context;
      cloned.ns = vnode.ns;
      cloned.isStatic = vnode.isStatic;
      cloned.key = vnode.key;

      return cloned;
    }

    const vNodes = this.$slots.default === undefined ? [] : this.$slots.default;
    const clonedVNodes = this.$slots.default === undefined ? [] :
vNodes.map(vnode => cloneVNode(vnode));

    return h('div', [
      vNodes,
      clonedVNodes
    ])
  }
}
```

在 Render 函数里创建了一个 cloneVNode 的工厂函数，通过递归将 slot 所有子节点都克隆了一份，并对 VNode 的关键属性也进行了复制。

深度克隆 slot 并非 Vue.js 内置方法，也没有得到推荐，属于黑科技，在一些特殊的场景才会使用到，正常业务几乎是用不到的。比如 iView 组件库的穿梭框组件 Transfer，就用到了这种方法：



它的使用方法是：

```
<Transfer
    :data="data"
    :target-keys="targetKeys"
    :render-format="renderFormat">
  <div :style="{float: 'right', margin: '5px'}">
    <Button size="small" @click="reloadMockData">Refresh</Button>
  </div>
</Transfer>
```

示例中的默认 slot 是一个 Refresh 按钮，使用者只写了一遍，但在 Transfer 组件中，是通过克隆 VNode 的方法，显示了两遍。如果不这样做，就要声明两个具名 slot，但是左右两个的逻辑可能是完全一样的，使用者就要写两个一模一样的 slot，这是不友好的。

> Render 函数的基本用法还有很多，比如 v-model 的用法、事件和修饰符、slot 等，读者可以到 Vue.js 文档阅读。Vue.js 渲染函数

Render 函数使用场景

上文说到，一般情况下是不推荐直接使用 Render 函数的，使用 template 足以，在 Vue.js 中，使用 Render 函数的场景，主要有以下 4 点：

1. 使用两个相同 slot。在 template 中，Vue.js 不允许使用两个相同的 slot，比如下面的示例是错误的：

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

   解决方案就是上文中讲到的**约束**，使用一个深度克隆 VNode 节点的方法。

2. 在 SSR 环境（服务端渲染），如果不是常规的 template 写法，比如通过 Vue.extend 和 new Vue 构造来生成的组件实例，是编译不过的，在前面小节也有所介绍。回顾上一节的 `$Alert` 组件的 notification.js 文件，当时是使用 Render 函数来渲染 Alert 组件，如果改成另一种写法，在 SSR 中会报错，对比两种写法：

```
// 正确写法
import Alert from './alert.vue';
import Vue from 'vue';

Alert.newInstance = properties => {
  const props = properties || {};

  const Instance = new Vue({
    data: props,
    render (h) {
      return h(Alert, {
        props: props
      });
    }
  });

  const component = Instance.$mount();
  document.body.appendChild(component.$el);

  const alert = Instance.$children[0];

  return {
    add (noticeProps) {
      alert.add(noticeProps);
    },
    remove (name) {
      alert.remove(name);
    }
  }
};

export default Alert;
```

```
// 在 SSR 下报错的写法
import Alert from './alert.vue';
import Vue from 'vue';

Alert.newInstance = properties => {
  const props = properties || {};

  const div = document.createElement('div');
  div.innerHTML = `<Alert ${props}></Alert>`;
  document.body.appendChild(div);

  const Instance = new Vue({
    el: div,
    data: props,
    components: { Alert }
  });
```

```
  const alert = Instance.$children[0];

  return {
    add (noticeProps) {
      alert.add(noticeProps);
    },
    remove (name) {
      alert.remove(name);
    }
  }
};

export default Alert;
```

3. 在 runtime 版本的 Vue.js 中，如果使用 Vue.extend 手动构造一个实例，使用 template 选项是会报错的，在第 9 节中也有所介绍。解决方案也很简单，把 template 改写为 Render 就可以了。需要注意的是，在开发独立组件时，可以通过配置 Vue.js 版本来使 template 选项可用，但这是在自己的环境，无法保证使用者的 Vue.js 版本，所以对于提供给他人用的组件，是需要考虑兼容 runtime 版本和 SSR 环境的。
4. 这可能是使用 Render 函数最重要的一点。一个 Vue.js 组件，有一部分内容需要从父级传递来显示，如果是文本之类的，直接通过 `props` 就可以，如果这个内容带有样式或复杂一点的 html 结构，可以使用 `v-html` 指令来渲染，父级传递的仍然是一个 HTML Element 字符串，不过它仅仅是能解析正常的 html 节点且有 XSS 风险。当需要最大化程度自定义显示内容时，就需要 `Render` 函数，它可以渲染一个完整的 Vue.js 组件。你可能会说，用 slot 不就好了？的确，slot 的作用就是做内容分发的，但在一些特殊组件中，可能 slot 也不行。比如一个表格组件 `Table`，它只接收两个 props：列配置 columns 和行数据 data，不过某一列的单元格，不是只将数据显示出来那么简单，可能带有一些复杂的操作，这种场景只用 slot 是不行的，没办法确定是那一列的 slot。这种场景有两种解决方案，其一就是 Render 函数，下一节的实战就是开发这样一个 Table 组件；另一种是用作用域 slot（slot-scope），后面小节也会详细介绍。

Functional Render

Vue.js 提供了一个 `functional` 的布尔值选项，设置为 true 可以使组件无状态和无实例，也就是没有 data 和 this 上下文。这样用 Render 函数返回虚拟节点可以更容易渲染，因为函数化组件（Functional Render）只是一个函数，渲染开销要小很多。

使用函数化组件，Render 函数提供了第二个参数 context 来提供临时上下文。组件需要的 data、props、slots、children、parent 都是通过这个上下文来传递的，比如 this.level 要改写为 context.props.level，this.$slots.default 改写为 context.children。

> 您可以阅读 Vue.js 文档—函数式组件 来查看示例。

函数化组件在业务中并不是很常用，而且也有类似的方法来实现，比如某些场景可以用 is 特性来动态挂载组件。函数化组件主要适用于以下两个场景：

- 程序化地在多个组件中选择一个；
- 在将 children、props、data 传递给子组件之前操作它们。

比如上文说过的，某个组件需要使用 Render 函数来自定义，而不是通过传递普通文本或 v-html 指令，这时就可以用 Functional Render，来看下面的示例：

1. 首先创建一个函数化组件 **render.js**：

```
// render.js
export default {
  functional: true,
  props: {
    render: Function
  },
  render: (h, ctx) => {
    return ctx.props.render(h);
  }
};
```

它只定义了一个 props：render，格式为 Function，因为是 functional，所以在 render 里使用了第二个参数 `ctx` 来获取 props。这是一个中间文件，并且可以复用，其它组件需要这个功能时，都可以引入它。

2. 创建组件：

```
<!-- my-component.vue -->
<template>
  <div>
    <Render :render="render"></Render>
  </div>
</template>
<script>
  import Render form './render.js';

  export default {
    components: { Render },
    props: {
      render: Function
    }
  }
</script>
```

3. 使用上面的 my-compoennt 组件：

```
<!-- demo.vue -->
<template>
  <div>
    <my-component :render="render"></my-component>
  </div>
</template>
<script>
  import myComponent from '../components/my-component.vue';

  export default {
    components: { myComponent },
    data () {
      return {
        render: (h) => {
          return h('div', {
            style: {
              color: 'red'
            }
```

```
          }, '自定义内容');
        }
      }
    }
  }
</script>
```

这里的 render.js 因为只是把 demo.vue 中的 Render 内容过继，并无其它用处，所以用了 Functional Render。

就此例来说，完全可以用 slot 取代 Functional Render，那是因为只有 `render` 这一个 prop。如果示例中的 `<Render>` 是用 `v-for` 生成的，也就是多个时，用 一个 slot 是实现不了的，那时用 Render 函数就很方便了，后面章节会专门介绍。

结语

如果想换一种思路写 Vue.js，就试试 Render 函数吧，它会让你“又爱又恨”！

注：本节部分内容参考了《Vue.js 实战》（清华大学出版社），部分代码参考 iView。

实战 5：可用 Render 自定义列的表格组件——Table

表格组件 Table 是中后台产品中最常用的组件之一，用于展示大量结构化的数据。大多数组件库都提供了表格组件，比如 iView，功能也是非常强大。正规的表格，是由 `<table>`、`<thead>`、`<tbody>`、`<tr>`、`<th>`、`<td>` 这些标签组成，一般分为表头 **columns** 和数据 **data**。本小节就来开发一个最基本的表格组件 Table，它支持使用 Render 函数来自定义某一列。

分析

如果表格只是呈现数据，是比较简单的，比如下图：

Name	Age	Address
John Brown	18	New York No. 1 Lake Park
Jim Green	24	London No. 1 Lake Park
Joe Black	30	Sydney No. 1 Lake Park
Jon Snow	26	Ottawa No. 2 Lake Park

因为结构简单，我们甚至不需要组件，直接使用标准的 table 系列标签就可以。但有的时候，除了呈现数据，也会带有一些交互，比如有一列操作栏，可以编辑整行的数据：

姓名	年龄	出生日期	地址	操作
王小明	18	1999-2-21	北京市朝阳区芍药居	修改
张小刚	25	1992-1-23	北京市海淀区西二旗	修改
李小红	30	1987-11-10	上海市浦东新区世纪大道	修改
周小伟	26	1991-10-10	深圳市南山区深南大道	修改

写一个个的 table 系列标签是很麻烦并且重复的，而组件的好处就是省去这些基础的工作，我们直接给 Table 组件传递列的配置 **columns** 和行数据 **data**，其余的都交给 Table 组件做了。

开发 Table 组件前，有必要先了解上文说到的一系列 table 标签。一般的 table 结构是这样的：

```
<table>
  <thead>
    <tr>
      <th>姓名</th>
      <th>年龄</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>王小明</td>
      <td>18</td>
    </tr>
    <tr>
      <td>张小刚</td>
      <td>25</td>
    </tr>
  </tbody>
</table>
```

- table：定义 HTML 表格；
- thead：定义表头；
- tbody：定义表格主体；
- tr：定义表格行；
- th：定义表头单元格；
- td：定义表格单元。

标准的表格系列标签，跟 div+css 实现是有很大区别的，比如表格在做单元格合并时，有提供原生属性，用 div 就很麻烦了；再比如渲染原理上也有一定的区别，table 会在内容全部下载完后加载。详细的介绍可以阅读文末的扩展阅读 1。

知道了表格的结构，再来分析如何定制 API。可以看到，表格分为了两部分，表头 thead 和数据 tbody，那 props 也定义两个：

- columns：列配置，格式为数组，其中每一列 column 是一个对象，用来描述这一列的信息，它的具体说明如下：
  - title：列头显示文字；
  - key：对应列内容的字段名；
  - render：自定义渲染列，使用 Vue 的 Render 函数，不定义则直接显示为文本。

  比如：

```
[
  {
    title: '姓名',
    key: 'name'
  },
  {
    title: '年龄',
    key: 'age'
  }
]
```

- data：显示的结构化数据，格式为数组，其中每一个对象，就是一行的数据，比如：

```
[
  {
    name: '王小明',
    age: 18
  },
  {
    name: '张小刚',
    age: 25
  }
]
```

column 定义的 `key` 值，与 data 是一一对应的，这是一种常见的数据接口定义规则，也是 Vue.js 组件中，用数据驱动而不是 slot 驱动的经典案例。

为什么 Table 组件要用数据驱动，而不是 slot 驱动呢？slot 在很多组件中的确很好用，不过 Table 组件包含了大量的基础表格标签，如果都交给使用者由 slot 承载的话，开发成本不亚于自己实现一个 table 了，而数据驱动就简单的多，数据一般从服务端获取后就可以直接使用（或简单处理），使用者主要来定义每列的配置 **columns** 就可以了。

因为不确定使用者要对某一列做什么交互，所以不能在 Table 内来实现自定义列。使用 Render 函数可以将复杂的自定义列模板的工作交给使用者来配置，Table 内只用一个 Functional Render 做中转。

完成基础表格

我们先来完成一个基础的表格组件，之后再接入 Render 来配置自定义列。

在 `src/components` 目录下新建 `table-render` 目录，并创建 `table.vue` 文件：

```
<!-- src/components/table-render/table.vue -->
<template>
  <table>
    <thead>
      <tr>
        <th v-for="col in columns">{{ col.title }}</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="row in data">
        <td v-for="col in columns">{{ row[col.key] }}</td>
      </tr>
    </tbody>
```

```
  </table>
</template>
<script>
  export default {
    props: {
      columns: {
        type: Array,
        default () {
          return [];
        }
      },
      data: {
        type: Array,
        default () {
          return [];
        }
      }
    }
  }
</script>
<style>
  table{
    width: 100%;
    border-collapse: collapse;
    border-spacing: 0;
    empty-cells: show;
    border: 1px solid #e9e9e9;
  }
  table th{
    background: #f7f7f7;
    color: #5c6b77;
    font-weight: 600;
    white-space: nowrap;
  }
  table td, table th{
    padding: 8px 16px;
    border: 1px solid #e9e9e9;
    text-align: left;
  }
</style>
```

> props 中的 columns 和 data 的格式都是数组，这里要注意的是，如果 props 的类型是**对象**或**数组**，它的默认值必须从一个工厂函数获取。

tbody 内嵌套使用了两次 `v-for`，外层循环数据 data，内层循环列 columns，这样就填充了每个单元格。

新建路由 `table-render`，并在 `src/views/` 目录下新建页面 `table-render.vue`：

```
<!-- src/views/table-render.vue -->
<template>
  <div>
    <table-render :columns="columns" :data="data"></table-render>
  </div>
</template>
<script>
```

```
  import TableRender from '../components/table-render/table.vue';

  export default {
    components: { TableRender },
    data () {
      return {
        columns: [
          {
            title: '姓名',
            key: 'name'
          },
          {
            title: '年龄',
            key: 'age'
          },
          {
            title: '出生日期',
            key: 'birthday'
          },
          {
            title: '地址',
            key: 'address'
          },
          {
            title: '操作'
          }
        ],
        data: [
          {
            name: '王小明',
            age: 18,
            birthday: '919526400000',
            address: '北京市朝阳区芍药居'
          },
          {
            name: '张小刚',
            age: 25,
            birthday: '696096000000',
            address: '北京市海淀区西二旗'
          },
          {
            name: '李小红',
            age: 30,
            birthday: '563472000000',
            address: '上海市浦东新区世纪大道'
          },
          {
            name: '周小伟',
            age: 26,
            birthday: '687024000000',
            address: '深圳市南山区深南大道'
          }
        ]
      }
    }
  }
</script>
```

运行后的效果如下图：

姓名	年龄	出生日期	地址	操作
王小明	18	919526400000	北京市朝阳区芍药居	
张小刚	25	696096000000	北京市海淀区西二旗	
李小红	30	563472000000	上海市浦东新区世纪大道	
周小伟	26	687024000000	深圳市南山区深南大道	

表格已经能渲染出来了，但现在的单元格只是将 data 当作纯文本来显示，所以出生日期列显示为时间戳，因为服务端对日期有时会保存为时间戳格式。如果要显示正常的日期（如1991-5-14），目前可以另写一个计算属性（computed），手动将时间戳换算为标准日期格式后，来动态修改 data 里的 `birthday` 字段。这样做对于出生日期这样的数据还好，但对于**操作**这一列就不可取了，因为它带有业务逻辑，点击**编辑**按钮，是可以对当前行数据进行修改的。这时就要用到 Render 函数。

使用 Render 自定义列模板

上一节我们已经介绍过函数式组件 Functional Render 的用法，它没有状态和上下文，主要用于中转一个组件，用在本节的 Table 组件非常合适。

先在 `src/components/table-render` 目录下新建 `render.js` 文件：

```
// src/components/table-render/render.js
export default {
  functional: true,
  props: {
    row: Object,
    column: Object,
    index: Number,
    render: Function
  },
  render: (h, ctx) => {
    const params = {
      row: ctx.props.row,
      column: ctx.props.column,
      index: ctx.props.index
    };

    return ctx.props.render(h, params);
  }
};
```

render.js 定义了 4 个 props：

- **row**：当前行的数据；
- **column**：当前列的数据；
- **index**：当前是第几行；
- **render**：具体的 render 函数内容。

这里的 `render` 选项并没有渲染任何节点，而是直接返回 props 中定义的 render，并将 h 和当前的行、列、序号作为参数传递出去。然后在 table.vue 里就可以使用 render.js 组件：

```
<!-- table.vue，部分代码省略 -->
<template>
```

```
  <table>
    <thead>
      <tr>
        <th v-for="col in columns">{{ col.title }}</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="(row, rowIndex) in data">
        <td v-for="col in columns">
          <template v-if="'render' in col">
            <Render :row="row" :column="col" :index="rowIndex"
:render="col.render"></Render>
          </template>
          <template v-else>{{ row[col.key] }}</template>
        </td>
      </tr>
    </tbody>
  </table>
</template>
<script>
  import Render from './render.js';

  export default {
    components: { Render },
    props: {
      columns: {
        type: Array,
        default () {
          return [];
        }
      },
      data: {
        type: Array,
        default () {
          return [];
        }
      }
    }
  }
</script>
```

如果 columns 中的某一列配置了 render 字段，那就通过 render.js 完成自定义模板，否则以字符串形式渲染。比如对出生日期这列显示为标准的日期格式，可以这样定义 column：

```
// src/views/table-render.vie，部分代码省略
export default {
  data () {
    return {
      columns: [
        // ...
        {
          title: '出生日期',
          render: (h, { row, column, index }) => {
            const date = new Date(parseInt(row.birthday));
            const year = date.getFullYear();
            const month = date.getMonth() + 1;
```

```
            const day = date.getDate();

            const birthday = `${year}-${month}-${day}`;

            return h('span', birthday);
          }
        }
      ]
    }
  }
}
```

效果如下图：

姓名	年龄	出生日期	地址	操作
王小明	18	1999-2-21	北京市朝阳区芍药居	
张小刚	25	1992-1-23	北京市海淀区西二旗	
李小红	30	1987-11-10	上海市浦东新区世纪大道	
周小伟	26	1991-10-10	深圳市南山区深南大道	

需要注意的是，columns 里定义的 render，是有两个参数的，第一个是 createElement（即 h），第二个是从 render.js 传过来的对象，它包含了当前行数据（`row`）、当前列配置（`column`）、当前是第几行（`index`），使用者可以基于这 3 个参数得到任意想要的结果。由于是自定义列了，显示什么都是使用者决定的，因此在使用了 render 的 column 里可以不用写字段 `key` 。

如果你真正理解了，应该知道 columns 里定义的 render 字段，它仅仅是名字叫 render 的一个普通函数，并非 Vue.js 实例的 *render* 选项，只是我们恰巧把它叫做 render 而已，如果愿意，也可以改为其它名字，比如 renderRow。真正的 Render 函数只有一个地方，那就是 render.js 中的 render 选项，只是它代理了 column 中的 render。这里有点绕，理清这个关系，就对 Functional Render 彻底理解了。

修改当前行

有了 render，Table 组件就已经完成了，剩余工作都是使用者来配置 columns 完成各种复杂的业务逻辑。本例来介绍最常见的表格中对整行数据编辑的功能。

操作这一列，默认是一个**修改**按钮，点击后，变为**保存**和**取消**两个按钮，同时本行其它各列都变为了输入框，并且初始值就是刚才单元格的数据。变为输入框后，可以任意修改单元格数据，点击保存按钮保存整行数据，点击取消按钮，还原至修改前的数据。

当进入编辑状态时，每一列的输入框都要有一个临时的数据使用 `v-model` 双向绑定来响应修改，所以在 data 里再声明四个数据：

```
// table-render.vue，部分代码省略
{
  data () {
    return {
      // ...
      editName: '',  // 第一列输入框
      editAge: '',  // 第二列输入框
      editBirthday: '',  // 第三列输入框
      editAddress: '',  // 第四列输入框
    }
  }
}
```

同时还要知道是在修改第几行的数据，所以再加一个数据标识当前正在修改的行序号（从 0 开始）：

```
// table-render.vue，部分代码省略
{
  data () {
    return {
      // ...
      editIndex: -1,  // 当前聚焦的输入框的行数
    }
  }
}
```

`editIndex` 默认给了 -1，也就是一个不存在的行号，当点击修改按钮时，再将它置为正确的行号。我们先定义操作列的 render：

```
// table-render.vue，部分代码省略
{
  data () {
    columns: [
      // ...
      {
        title: '操作',
        render: (h, { row, index }) => {
          // 如果当前行是编辑状态，则渲染两个按钮
          if (this.editIndex === index) {
            return [
              h('button', {
                on: {
                  click: () => {
                    this.data[index].name = this.editName;
                    this.data[index].age = this.editAge;
                    this.data[index].birthday = this.editBirthday;
                    this.data[index].address = this.editAddress;
                    this.editIndex = -1;
                  }
                }
              }, '保存'),
              h('button', {
                style: {
                  marginLeft: '6px'
                },
```

```
                on: {
                  click: () => {
                    this.editIndex = -1;
                  }
                }
              }, '取消')
            ];
          } else {  // 当前行是默认状态，渲染为一个按钮
            return h('button', {
              on: {
                click: () => {
                  this.editName = row.name;
                  this.editAge = row.age;
                  this.editAddress = row.address;
                  this.editBirthday = row.birthday;
                  this.editIndex = index;
                }
              }
            }, '修改');
          }
        }
      }
    ]
  }
}
```

render 里的 `if / else` 可以先看 else，因为默认是非编辑状态，也就是说 `editIndex` 还是 -1。当点击**修改**按钮时，把 render 中第二个参数 `{ row }` 中的各列数据赋值给了之前在 data 中声明的 4 个数据，这样做是因为之后点击**取消**按钮时，editName 等值已经修改了，还没有还原，所以在开启编辑状态的同时，初始化各输入框的值（当然也可以在取消时重置）。最后再把 `editIndex` 置为了对应的行序号 `{ index }`，此时 render 的 `if` 条件 `this.editIndex === index` 为真，编辑列变成了两个按钮：保存和取消。点击保存，直接修改表格源数据 data 中对应的各字段值，并将 editIndex 置为 -1，退出编辑状态；点击取消，不保存源数据，直接退出编辑状态。

除编辑列，其它各数据列都有两种状态：

1. 当 editIndex 等于当前行号 index 时，呈现输入框状态；
2. 当 editIndex 不等于当前行号 index 时，呈现默认数据。

以姓名为例：

```
// table-render.vue，部分代码省略
{
  data () {
    columns: [
      // ...
      {
        title: '姓名',
        key: 'name',
        render: (h, { row, index }) => {
          let edit;

          // 当前行为聚焦行时
          if (this.editIndex === index) {
            edit = [h('input', {
              domProps: {
```

```
                value: row.name
              },
              on: {
                input: (event) => {
                  this.editName = event.target.value;
                }
              }
            })];
          } else {
            edit = row.name;
          }

          return h('div', [
            edit
          ]);
        }
      }
    ]
  }
}
```

变量 `edit` 根据 editIndex 呈现不同的节点，还是先看 else，直接显示了对应字段的数据。在聚焦时（this.editIndex === index），渲染一个 `input` 输入框，初始值 `value` 通过 render 的 `domProps` 绑定了 `row.name`（这里也可绑定 editName），并监听了 input 事件，将输入的内容，实时缓存在数据 `editName` 中，供保存时使用。事实上，这里绑定的 value 和事件 input 就是语法糖 `v-model` 在 Render 函数中的写法，在 template 中，经常写作 `<input v-model="editName">`。

其它列与姓名类似，只是对于的字段不同：

```
// table-render.vue，部分代码省略
{
  data () {
    return {
      columns: [
        // ...
        {
          title: '年龄',
          key: 'age',
          render: (h, { row, index }) => {
            let edit;

            // 当前行为聚焦行时
            if (this.editIndex === index) {
              edit = [h('input', {
                domProps: {
                  value: row.age
                },
                on: {
                  input: (event) => {
                    this.editAge = event.target.value;
                  }
                }
              })];
            } else {
              edit = row.age;
            }
```

```
          return h('div', [
            edit
          ]);
        }
      },
      {
        title: '出生日期',
        render: (h, { row, index }) => {
          let edit;

          // 当前行为聚焦行时
          if (this.editIndex === index) {
            edit = [h('input', {
              domProps: {
                value: row.birthday
              },
              on: {
                input: (event) => {
                  this.editBirthday = event.target.value;
                }
              }
            })];
          } else {
            const date = new Date(parseInt(row.birthday));
            const year = date.getFullYear();
            const month = date.getMonth() + 1;
            const day = date.getDate();

            edit = `${year}-${month}-${day}`;
          }

          return h('div', [
            edit
          ]);
        }
      },
      {
        title: '地址',
        key: 'address',
        render: (h, { row, index }) => {
          let edit;

          // 当前行为聚焦行时
          if (this.editIndex === index) {
            edit = [h('input', {
              domProps: {
                value: row.address
              },
              on: {
                input: (event) => {
                  this.editAddress = event.target.value;
                }
              }
            })];
          } else {
            edit = row.address;
          }
```

```
            return h('div', [
              edit
            ]);
          }
        },
      ]
    }
  }
}
```

> 完整的代码见：https://github.com/icarusion/vue-component-book/blob/master/src/views/table-render.vue

这样，可编辑行的表格示例就完成了：

姓名	年龄	出生日期	地址	操作
王小明	18	1999-2-21	北京市朝阳区芍药居	修改
张小刚	25	1992-1-23	北京市海淀区西二旗	修改
李小红	30	1987-11-10	上海市浦东新区世纪大道	修改
周小伟	26	1991-10-10	深圳市南山区深南大道	修改

New rec

结语

本示例的 Table 组件，只展现了表格最核心的功能——自定义列模板，一个完整的 Table 组件功能要复杂的多，比如排序、筛选、列固定、表头固定、表头嵌套等。万事开头难，打好了 Table 的地基，后面的功能可以持续开发。

事实上，很多 Vue.js 的开发难题，都可以用 Render 函数来解决，它比 template 模板更灵活，可以完全发挥 JavaScript 的编程能力，因此很多 JS 的开发思想都可以借鉴。如果你习惯 JSX，那完全可以抛弃传统的 template 写法。

Render 函数虽好，但也是有弊端的，通过上面的示例可以发现，写出来的 VNode 对象是很难读的，维护性也比 template 差。下一节，我们将改写 Table 组件，用另一种思想来实现同样的功能。

扩展阅读

- Div 和 Table 的区别

实战 6：可用 slot-scope 自定义列的表格组件——Table

上一节，我们基于 Render 函数实现了在表格中自定义列模板的组件 Table，虽说 Render 函数能够完全发挥 JavaScript 的编程能力，实现几乎所有的自定义工作，但本质上，使用者写的是一个庞大的 JS 对象，它不具备 DOM 结构，可读性和可维护性都比较差。对于大部分写 Vue.js 的开发者来说，更倾向于使用 template 的语法，毕竟它是 Vue.js 独有的特性。本小节则在上一节的 Table 组件基础上修改，实现一种达到同样渲染效果，但对使用者更友好的 slot-scope 写法。

什么是 slot-scope

slot（插槽）我们都很熟悉，它是 Vue.js 组件的 3 个 API 之一，用于分发内容。那 slot-scope 是什么呢？先来看一个场景，比如某组件拥有下面的模板：

```
<ul>
  <li v-for="book in books" :key="book.id">
    {{ book.name }}
  </li>
</ul>
```

使用者传递一个数组 `books`，由组件内的 `v-for` 循环显示，这里的 `{{ book.name }}` 是纯文本输出，如果想自定义它的模板（即内容分发），就要用到 slot，但 slot 只能是固定的模板，没法自定义循环体中的一个具体的项，事实上这跟上一节的 Table 场景是类似的。

常规的 slot 无法实现对组件循环体的每一项进行不同的内容分发，这就要用到 slot-scope，它本质上跟 slot 一样，只不过可以传递参数。比如上面的示例，使用 slot-scope 封装：

```
<ul>
  <li v-for="book in books" :key="book.id">
    <slot :book="book">
      <!-- 默认内容 -->
      {{ book.name }}
    </slot>
  </li>
</ul>
```

在 slot 上，传递了一个自定义的参数 `book`，它的值绑定的是当前循环项的数据 book，这样在父级使用时，就可以在 slot 中访问它了：

```
<book-list :books="books">
  <template slot-scope="slotProps">
    <span v-if="slotProps.book.sale">限时优惠</span>
    {{ slotProps.book.name }}
  </template>
</book-list>
```

使用 slot-scope 指定的参数 `slotProps` 就是这个 slot 的全部参数，它是一个对象，在 slot-scope 中是可以传递多个参数的，上例我们只写了一个参数 `book`，所以访问它就是 `slotProps.book`。这里推荐使用 ES6 的解构，能让参数使用起来更方便：

```
<book-list :books="books">
  <template slot-scope="{ book }">
    <span v-if="book.sale">限时优惠</span>
    {{ book.name }}
  </template>
</book-list>
```

除了可以传递参数，其它用法跟 slot 是一样的，比如也可以“具名”：

```
<slot :book="book" name="book">
  {{ book.name }}
</slot>
```

```
<template slot-scope="{ book }" slot="book">
  <span v-if="book.sale">限时优惠</span>
  {{ book.name }}
</template>
```

这就是作用域 slot（slot-scope），能够在组件的循环体中做内容分发，有了它，Table 组件的自定义列模板就不用写一长串的 Render 函数了。

为了把 Render 函数和 slot-scope 理解透彻，下面我们用 3 种方法来改写 Table，实现 slot-scope 自定义列模板。

方案一

第一种方案，用最简单的 slot-scope 实现，同时也兼容 Render 函数的旧用法。拷贝上一节的 Table 组件目录，更名为 `table-slot`，同时也拷贝路由，更名为 `table-slot.vue`。为了兼容旧的 Render 函数用法，在 columns 的列配置 column 中，新增一个字段 `slot` 来指定 slot-scope 的名称：

```
<!-- src/components/table-slot/table.vue -->
<template>
  <table>
    <thead>
      <tr>
        <th v-for="col in columns">{{ col.title }}</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="(row, rowIndex) in data">
        <td v-for="col in columns">
          <template v-if="'render' in col">
            <Render :row="row" :column="col" :index="rowIndex"
:render="col.render"></Render>
          </template>
          <template v-else-if="'slot' in col">
            <slot :row="row" :column="col" :index="rowIndex" :name="col.slot">
</slot>
          </template>
          <template v-else>{{ row[col.key] }}</template>
        </td>
      </tr>
    </tbody>
  </table>
</template>
```

相比原先的文件，只在 `'render' in col` 的条件下新加了一个 `template` 的标签，如果使用者的 column 配置了 render 字段，就优先以 Render 函数渲染，然后再判断是否用 slot-scope 渲染。在定义的作用域 slot 中，将行数据 `row`、列数据 `column` 和第几行 `index` 作为 slot 的参数，并根据 column 中指定的 slot 字段值，动态设置了具名 `name`。使用者在配置 columns 时，只要指定了某一

列的 slot，那就可以在 Table 组件中使用 slot-scope。我们以上一节的可编辑整行数据为例，用 slot-scope 的写法实现完全一样的效果：

```
<!-- src/views/table-slot.vue -->
<template>
  <div>
    <table-slot :columns="columns" :data="data">
      <template slot-scope="{ row, index }" slot="name">
        <input type="text" v-model="editName" v-if="editIndex === index" />
        <span v-else>{{ row.name }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="age">
        <input type="text" v-model="editAge" v-if="editIndex === index" />
        <span v-else>{{ row.age }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="birthday">
        <input type="text" v-model="editBirthday" v-if="editIndex === index" />
        <span v-else>{{ getBirthday(row.birthday) }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="address">
        <input type="text" v-model="editAddress" v-if="editIndex === index" />
        <span v-else>{{ row.address }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="action">
        <div v-if="editIndex === index">
          <button @click="handleSave(index)">保存</button>
          <button @click="editIndex = -1">取消</button>
        </div>
        <div v-else>
          <button @click="handleEdit(row, index)">操作</button>
        </div>
      </template>
    </table-slot>
  </div>
</template>
<script>
  import TableSlot from '../components/table-slot/table.vue';

  export default {
    components: { TableSlot },
    data () {
      return {
        columns: [
          {
            title: '姓名',
            slot: 'name'
          },
          {
            title: '年龄',
            slot: 'age'
          },
          {
            title: '出生日期',
```

```
            slot: 'birthday'
          },
          {
            title: '地址',
            slot: 'address'
          },
          {
            title: '操作',
            slot: 'action'
          }
        ],
        data: [
          {
            name: '王小明',
            age: 18,
            birthday: '919526400000',
            address: '北京市朝阳区芍药居'
          },
          {
            name: '张小刚',
            age: 25,
            birthday: '696096000000',
            address: '北京市海淀区西二旗'
          },
          {
            name: '李小红',
            age: 30,
            birthday: '563472000000',
            address: '上海市浦东新区世纪大道'
          },
          {
            name: '周小伟',
            age: 26,
            birthday: '687024000000',
            address: '深圳市南山区深南大道'
          }
        ],
        editIndex: -1,  // 当前聚焦的输入框的行数
        editName: '',  // 第一列输入框，当然聚焦的输入框的输入内容，与 data 分离避免重构的
闪烁
        editAge: '',  // 第二列输入框
        editBirthday: '',  // 第三列输入框
        editAddress: '',  // 第四列输入框
      }
    },
    methods: {
      handleEdit (row, index) {
        this.editName = row.name;
        this.editAge = row.age;
        this.editAddress = row.address;
        this.editBirthday = row.birthday;
        this.editIndex = index;
      },
      handleSave (index) {
        this.data[index].name = this.editName;
        this.data[index].age = this.editAge;
        this.data[index].birthday = this.editBirthday;
        this.data[index].address = this.editAddress;
```

```
        this.editIndex = -1;
      },
      getBirthday (birthday) {
        const date = new Date(parseInt(birthday));
        const year = date.getFullYear();
        const month = date.getMonth() + 1;
        const day = date.getDate();

        return `${year}-${month}-${day}`;
      }
    }
  }
</script>
```

示例中在 `<table-slot>` 内的每一个 `<template>` 就对应某一列的 slot-scope 模板，通过配置的 `slot` 字段，指定具名的 slot-scope。可以看到，基本是把 Render 函数还原成了 html 的写法，这样看起来直接多了，渲染效果是完全一样的。在 slot-scope 中，平时怎么写组件，这里就怎么写，Vue.js 所有的 API 都是可以直接使用的。

方案一是最优解，一般情况下，使用这种方案就可以了，其余两种方案是基于 Render 的。

方案二

第二种方案，不需要修改原先的 Table 组件代码，只是在使用层面修改即可。先来看具体的使用代码，然后再做分析。注意，这里使用的 Table 组件，仍然是上一节 `src/components/table-render` 的组件，它只有 Render 函数，没有定义 slot-scope：

```
<!-- src/views/table-render.vue 的改写 -->
<template>
  <div>
    <table-render ref="table" :columns="columns" :data="data">
      <template slot-scope="{ row, index }" slot="name">
        <input type="text" v-model="editName" v-if="editIndex === index" />
        <span v-else>{{ row.name }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="age">
        <input type="text" v-model="editAge" v-if="editIndex === index" />
        <span v-else>{{ row.age }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="birthday">
        <input type="text" v-model="editBirthday" v-if="editIndex === index" />
        <span v-else>{{ getBirthday(row.birthday) }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="address">
        <input type="text" v-model="editAddress" v-if="editIndex === index" />
        <span v-else>{{ row.address }}</span>
      </template>

      <template slot-scope="{ row, index }" slot="action">
        <div v-if="editIndex === index">
          <button @click="handleSave(index)">保存</button>
          <button @click="editIndex = -1">取消</button>
```

```
        </div>
        <div v-else>
          <button @click="handleEdit(row, index)">操作</button>
        </div>
      </template>
    </table-render>
  </div>
</template>
<script>
  import TableRender from '../components/table-render/table.vue';

  export default {
    components: { TableRender },
    data () {
      return {
        columns: [
          {
            title: '姓名',
            render: (h, { row, column, index }) => {
              return h(
                'div',
                this.$refs.table.$scopedSlots.name({
                  row: row,
                  column: column,
                  index: index
                })
              )
            }
          },
          {
            title: '年龄',
            render: (h, { row, column, index }) => {
              return h(
                'div',
                this.$refs.table.$scopedSlots.age({
                  row: row,
                  column: column,
                  index: index
                })
              )
            }
          },
          {
            title: '出生日期',
            render: (h, { row, column, index }) => {
              return h(
                'div',
                this.$refs.table.$scopedSlots.birthday({
                  row: row,
                  column: column,
                  index: index
                })
              )
            }
          },
          {
            title: '地址',
            render: (h, { row, column, index }) => {
```

```
              return h(
                'div',
                this.$refs.table.$scopedSlots.address({
                  row: row,
                  column: column,
                  index: index
                })
              )
            }
          },
          {
            title: '操作',
            render: (h, { row, column, index }) => {
              return h(
                'div',
                this.$refs.table.$scopedSlots.action({
                  row: row,
                  column: column,
                  index: index
                })
              )
            }
          }
        ],
        data: [],
        editIndex: -1,  // 当前聚焦的输入框的行数
        editName: '',  // 第一列输入框，当然聚焦的输入框的输入内容，与 data 分离避免重构的
闪烁
        editAge: '',  // 第二列输入框
        editBirthday: '',  // 第三列输入框
        editAddress: '',  // 第四列输入框
      }
    },
    methods: {
      handleEdit (row, index) {
        this.editName = row.name;
        this.editAge = row.age;
        this.editAddress = row.address;
        this.editBirthday = row.birthday;
        this.editIndex = index;
      },
      handleSave (index) {
        this.data[index].name = this.editName;
        this.data[index].age = this.editAge;
        this.data[index].birthday = this.editBirthday;
        this.data[index].address = this.editAddress;
        this.editIndex = -1;
      },
      getBirthday (birthday) {
        const date = new Date(parseInt(birthday));
        const year = date.getFullYear();
        const month = date.getMonth() + 1;
        const day = date.getDate();

        return `${year}-${month}-${day}`;
      }
    },
    mounted () {
```

```
      this.data = [
        {
          name: '王小明',
          age: 18,
          birthday: '919526400000',
          address: '北京市朝阳区芍药居'
        },
        {
          name: '张小刚',
          age: 25,
          birthday: '696096000000',
          address: '北京市海淀区西二旗'
        },
        {
          name: '李小红',
          age: 30,
          birthday: '563472000000',
          address: '上海市浦东新区世纪大道'
        },
        {
          name: '周小伟',
          age: 26,
          birthday: '687024000000',
          address: '深圳市南山区深南大道'
        }
      ];
    }
  }
</script>
```

在 slot-scope 的使用上（即 template 的内容），与方案一是完全一致的，可以看到，在 column 的定义上，仍然使用了 render 字段，只不过每个 render 都渲染了一个 div 节点，而这个 div 的内容，是指定来在 `<table-render>` 中定义的 slot-scope：

```
render: (h, { row, column, index }) => {
  return h(
    'div',
    this.$refs.table.$scopedSlots.name({
      row: row,
      column: column,
      index: index
    })
  )
}
```

这正是 Render 函数灵活的一个体现，使用 `$scopedSlots` 可以访问 slot-scope，所以上面这段代码的意思是，name 这一列仍然是使用 Functional Render，只不过 Render 的是一个预先定义好的 slot-scope 模板。

有一点需要注意的是，示例中的 `data` 默认是空数组，而在 mounted 里才赋值的，是因为这样定义的 slot-scope，初始时读取 `this.$refs.table.$scopedSlots` 是读不到的，会报错，当没有数据时，也就不会去渲染，也就避免了报错。

这种方案虽然可行，但归根到底是一种 hack，不是非常推荐，之所以列出来，是为了对 Render 和 slot-scope 有进一步的认识。

方案三

第 3 中方案的思路和第 2 种是一样的，它介于方案 1 与方案 2 之间。这种方案要修改 Table 组件代码，但是用例与方案 1 完全一致。

在方案 2 中，我们是通过修改用例使用 slot-scope 的，也就是说 Table 组件本身没有支持 slot-scope，是我们“强加”上去的，如果把强加的部分，集成到 Table 内，那对使用者就很友好了，同时也避免了初始化报错，不得不把 data 写在 mounted 的问题。

保持方案 1 的用例不变，修改 `src/components/table-render` 中的代码。为了同时兼容 Render 与 slot-scope，我们在 `table-render` 下新建一个 slot.js 的文件：

```
// src/components/table-render/slot.js
export default {
  functional: true,
  inject: ['tableRoot'],
  props: {
    row: Object,
    column: Object,
    index: Number
  },
  render: (h, ctx) => {
    return h('div', ctx.injections.tableRoot.$scopedSlots[ctx.props.column.slot]
({
      row: ctx.props.row,
      column: ctx.props.column,
      index: ctx.props.index
    }));
  }
};
```

它仍然是一个 Functional Render，使用 `inject` 注入了父级组件 table.vue（下文改写） 中提供的实例 `tableRoot`。在 render 里，也是通过方案 2 中使用 `$scopedSlots` 定义的 slot，不过这是在组件级别定义，对用户来说是透明的，只要按方案 1 的用例来写就可以了。

table.vue 也要做一点修改：

```
<!-- src/components/table-slot/table.vue 的改写，部分代码省略 -->
<template>
  <table>
    <thead>
      <tr>
        <th v-for="col in columns">{{ col.title }}</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="(row, rowIndex) in data">
        <td v-for="col in columns">
          <template v-if="'render' in col">
            <Render :row="row" :column="col" :index="rowIndex"
:render="col.render"></Render>
          </template>
```

```
          <template v-else-if="'slot' in col">
            <slot-scope :row="row" :column="col" :index="rowIndex"></slot-scope>
          </template>
          <template v-else>{{ row[col.key] }}</template>
        </td>
      </tr>
    </tbody>
  </table>
</template>
<script>
  import Render from './render.js';
  import SlotScope from './slot.js';

  export default {
    components: { Render, SlotScope },
    provide () {
      return {
        tableRoot: this
      };
    },
    props: {
      columns: {
        type: Array,
        default () {
          return [];
        }
      },
      data: {
        type: Array,
        default () {
          return [];
        }
      }
    }
  }
</script>
```

因为 slot-scope 模板是写在 table.vue 中的（对使用者来说，相当于写在组件 `<table-slot></table-slot>` 之间），所以在 table.vue 中使用 provide 向下提供了 Table 的实例，这样在 slot.js 中就可以通过 inject 访问到它，继而通过 `$scopedSlots` 获取到 slot。需要注意的是，在 Functional Render 是没有 this 上下文的，都是通过 h 的第二个参数临时上下文 ctx 来访问 prop、inject 等的。

方案 3 也是推荐使用的，当 Table 的功能足够复杂，层级会嵌套的比较深，那时方案 1 的 slot 就不会定义在第一级组件中，中间可能会隔许多组件，slot 就要一层层中转，相比在任何地方都能直接使用的 Render 就要麻烦了。所以，如果你的组件层级简单，推荐用第一种方案；如果你的组件已经成型（某 API 基于 Render 函数），但一时间不方便支持 slot-scope，而使用者又想用，那就选方案 2；如果你的组件已经成型（某 API 基于 Render 函数），但组件层级复杂，要按方案 1 那样支持 slot-scope 可能改动较大，还有可能带来新的 bug，那就用方案 3，它不会破坏原有的任何内容，但会额外支持 slot-scope 用法，关键是改动简单。

结语

理论上，绝大多数能用 Render 的地方，都可以用 slot-scope。对于极客来说，喜欢挑战各种新奇的写法，所以会在 Vue.js 中大量使用 Render 函数、JSX 甚至 TS；而对于求稳的开发者来说，常规的 template、slot、slot-scope 写法会是好的选择。如果非要选一种，那要从你团队的整体情况来定，如果团队大部分是写后端为主的，那可能更倾向于 TS；如果写过 React，或许 JSX 是不错的选择；如果实在不知道选什么，那就求稳吧！

递归组件与动态组件

递归组件

递归组件就是指组件在模板中调用自己，开启递归组件的必要条件，就是在组件中设置一个 `name` 选项。比如下面的示例：

```
<template>
  <div>
    <my-component></my-component>
  </div>
</template>
<script>
  export default {
    name: 'my-component'
  }
</script>
```

在 Webpack 中导入一个 Vue.js 组件，一般是通过 `import myComponent from 'xxx'` 这样的语法，然后在当前组件（页面）的 `components: { myComponent }` 里注册组件。这种组件是不强制设置 `name` 字段的，组件的名字都是使用者在 import 进来后自定义的，但递归组件的使用者是组件自身，它得知道这个组件叫什么，因为没有用 `components` 注册，所以 `name` 字段就是必须的了。除了递归组件用 `name`，我们在之前的小节也介绍过，用一些特殊的方法，通过遍历匹配组件的 name 选项来寻找组件实例。

不过呢，上面的示例是有问题的，如果直接运行，会抛出 `max stack size exceeded` 的错误，因为组件会无限递归下去，死循环。解决这个问题，就要给递归组件一个限制条件，一般会在递归组件上用 `v-if` 在某个地方设置为 `false` 来终结。比如我们给上面的示例加一个属性 count，当大于 5 时就不再递归：

```
<template>
  <div>
    <my-component :count="count + 1" v-if="count <= 5"></my-component>
  </div>
</template>
<script>
  export default {
    name: 'my-component',
    props: {
      count: {
        type: Number,
        default: 1
      }
    }
  }
</script>
```

所以，总结下来，实现一个递归组件的必要条件是：

- 要给组件设置 **name**；
- 要有一个明确的结束条件

递归组件常用来开发具有未知层级关系的独立组件，在业务开发中很少使用。比如常见的有级联选择器和树形控件：

级联选择器

树形控件

这类组件一般都是数据驱动型的，父级有一个字段 children，然后递归。下一节的实战，会开发一个树形控件 Tree。

动态组件

有的时候，我们希望根据一些条件，动态地切换某个组件，或动态地选择渲染某个组件。在之前小节介绍函数式组件 Functional Render 时，已经说过，它是一个没有上下文的函数，常用于程序化地在多个组件中选择一个。使用 Render 或 Functional Render 可以解决动态切换组件的需求，不过那是基于一个 JS 对象（Render 函数），而 Vue.js 提供了另外一个内置的组件 `<component>` 和 `is` 特性，可以更好地实现动态组件。

先来看一个 `<component>` 和 `is` 的基本示例，首先定义三个普通组件：

```
<!-- a.vue -->
<template>
  <div>
    组件 A
  </div>
</template>
<script>
  export default {

  }
</script>
```

```
<!-- b.vue -->
<template>
  <div>
    组件 B
  </div>
</template>
<script>
  export default {

  }
</script>
```

```
<!-- c.vue -->
<template>
  <div>
    组件 C
  </div>
</template>
<script>
  export default {

  }
</script>
```

然后在父组件中导入这 3 个组件，并动态切换：

```
<template>
  <div>
    <button @click="handleChange('A')">显示 A 组件</button>
    <button @click="handleChange('B')">显示 B 组件</button>
    <button @click="handleChange('C')">显示 C 组件</button>

    <component :is="component"></component>
  </div>
</template>
<script>
  import componentA from '../components/a.vue';
  import componentB from '../components/b.vue';
  import componentC from '../components/c.vue';

  export default {
    data () {
      return {
        component: componentA
      }
    },
    methods: {
      handleChange (component) {
        if (component === 'A') {
          this.component = componentA;
        } else if (component === 'B') {
          this.component = componentB;
        } else if (component === 'C') {
          this.component = componentC;
```

```
        }
      }
    }
  }
</script>
```

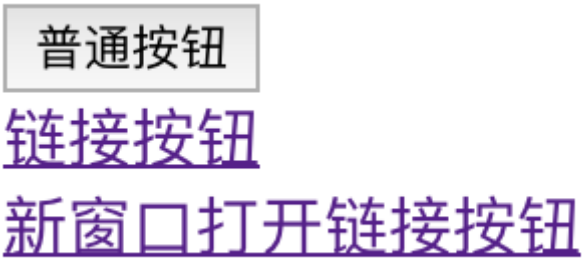
这里的 `is` 动态绑定的是一个组件对象（Object），它直接指向 a / b / c 三个组件中的一个。除了直接绑定一个 Object，还可以是一个 String，比如标签名、组件名。下面的这个组件，将原生的按钮 button 进行了封装，如果传入了 `prop: to`，那它会渲染为一个 `<a>` 标签，用于打开这个链接地址，如果没有传入 `to`，就当作普通 button 使用。来看下面的示例：

```
<!-- button.vue -->
<template>
  <component :is="tagName" v-bind="tagProps">
    <slot></slot>
  </component>
</template>
<script>
  export default {
    props: {
      // 链接地址
      to: {
        type: String,
        default: ''
      },
      // 链接打开方式，如 _blank
      target: {
        type: String,
        default: '_self'
      }
    },
    computed: {
      // 动态渲染不同的标签
      tagName () {
        return this.to === '' ? 'button' : 'a';
      },
      // 如果是链接，把这些属性都绑定在 component 上
      tagProps () {
        let props = {};

        if (this.to) {
          props = {
            target: this.target,
            href: this.to
          }
        }

        return props;
      }
    }
  }
</script>
```

使用组件：

```
<template>
  <div>
    <i-button>普通按钮</i-button>
    <br>
    <i-button to="https://juejin.im">链接按钮</i-button>
    <br>
    <i-button to="https://juejin.im" target="_blank">新窗口打开链接按钮</i-button>
  </div>
</template>
<script>
  import iButton from '../components/a.vue';

  export default {
    components: { iButton }
  }
</script>
```

最终会渲染出一个原生的 `<button>` 按钮和两个原生的链接 `<a>`，且第二个点击会在新窗口中打开链接，如图：



i-button 组件中的 `<component>` `is` 绑定的就是一个标签名称 button / a，并且通过 `v-bind` 将一些额外的属性全部绑定到了 `<component>` 上。

再回到第一个 a / b / c 组件切换的示例，如果这类的组件，频繁切换，事实上组件是会重新渲染的，比如我们在组件 A 里加两个生命周期：

```
<!-- a.vue -->
<template>
  <div>
    组件 A
  </div>
</template>
<script>
  export default {
    mounted () {
      console.log('组件创建了');
    },
    beforeDestroy () {
      console.log('组件销毁了');
    }
  }
</script>
```

只要切换到 A 组件，`mounted` 就会触发一次，切换到其它组件，`beforeDestroy` 也会触发一次，说明组件再重新渲染，这样有可能导致性能问题。为了避免组件的重复渲染，可以在 `<component>` 外层套一个 Vue.js 内置的 `<keep-alive>` 组件，这样，组件就会被缓存起来：

```
<keep-alive>
  <component :is="component"></component>
</keep-alive>
```

这时，只有 `mounted` 触发了，如果不离开当前页面，切换到其它组件，`beforeDestroy` 不会被触发，说明组件已经被缓存了。

keep-alive 还有一些额外的 props 可以配置：

- `include`：字符串或正则表达式。只有名称匹配的组件会被缓存。
- `exclude`：字符串或正则表达式。任何名称匹配的组件都不会被缓存。
- `max`：数字。最多可以缓存多少组件实例。

结语

还有一类是异步组件，Vue.js 文档已经介绍的很清楚了，可以阅读文末的扩展阅读 1。事实上异步组件我们用的很多，比如 router 的配置列表，一般都是用的异步组件形式：

```
{
  path: '/form',
  component: () => import('./views/form.vue')
}
```

这样每个页面才会在路由到时才加载对应的 JS 文件，否则入口文件会非常庞大。

递归组件、动态组件和异步组件是 Vue.js 中相对冷门的 3 种组件模式，不过在封装复杂的独立组件时，前两者会经常使用。

扩展阅读

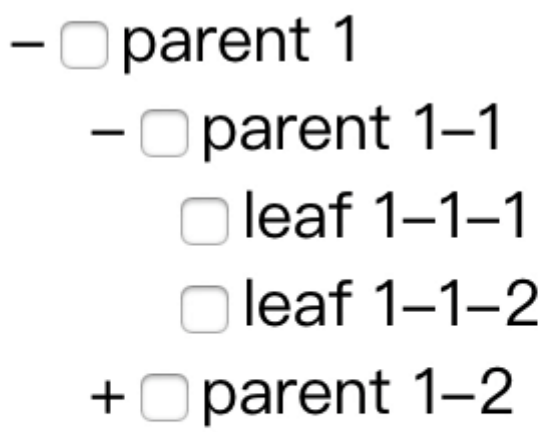
- 异步组件

实战 7：树形控件——Tree

本小节基于 Vue.js 的递归组件知识，来开发一个常见的树形控件—Tree。

Tree 组件是递归类组件的典型代表，它常用于文件夹、组织架构、生物分类、国家地区等等，世间万物的大多数结构都是树形结构。使用树控件可以完整展现其中的层级关系，并具有展开收起选择等交互功能。

本节要实现的 Tree 组件具有以下功能：



- 节点可以无限延伸（递归）；
- 可以展开 / 收起子节点；
- 节点可以选中，选中父节点，它的所有子节点也全部被选中，同样，反选父节点，其所有子节点也取消选择；
- 同一级所有子节点选中时，它的父级也自动选中，一直递归判断到根节点。

API

Tree 是典型的数据驱动型组件，所以节点的配置就是一个 data，里面描述了所有节点的信息，比如图片中的示例数据为：

```
data: [
  {
    title: 'parent 1',
    expand: true,
    children: [
      {
        title: 'parent 1-1',
        expand: true,
        children: [
          {
            title: 'leaf 1-1-1'
          },
          {
            title: 'leaf 1-1-2'
          }
        ]
      },
      {
        title: 'parent 1-2',
        children: [
          {
            title: 'leaf 1-2-1'
          },
          {
```

```
            title: 'leaf 1-2-1'
          }
        ]
      }
    ]
  }
]
```

每个节点的配置（props：data）描述如下：

- **title**：节点标题（本例为纯文本输出，可参考 Table 的 Render 或 slot-scope 将其扩展）；
- **expand**：是否展开直子节点。开启后，其直属子节点将展开；
- **checked**：是否选中该节点。开启后，该节点的 Checkbox 将选中；
- **children**：子节点属性数组。

如果一个节点没有 children 字段，那它就是最后一个节点，这也是递归组件终结的判断依据。

同时再提供一个是否显示多选框的 props：showCheckbox，以及两个 events：

- **on-toggle-expand**：展开和收起子列表时触发；
- **on-check-change**：点击复选框时触发。

因为是数据驱动，组件的 API 都比较简单，这一点跟 Table 组件是一样的，它们复杂的逻辑都在组件本身。

入口 tree.vue

在 `src/components` 中新建目录 `tree`，并在 tree 下创建两个组件 `tree.vue` 和 `node.vue`。tree.vue 是组件的入口，用于接收和处理数据，并将数据传递给 node.vue；node.vue 就是一个递归组件，它构成了每一个**节点**，即一个可展开 / 关闭的按钮（+或-）、一个多选框（使用第 7 节的 Checkbox 组件）、节点标题以及递归的下一级节点。可能现在听起来比较困惑，不要慌，下面逐一分解。

tree.vue 主要负责两件事：

1. 定义了组件的入口，即组件的 API；
2. 对接收的数据 props：data 初步处理，为了在 tree.vue 中不破坏使用者传递的源数据 data，所以会克隆一份数据（cloneData）。

因为传递的数据 data 是一个复杂的数组结构，克隆它要使用深拷贝，因为浅拷贝数据仍然是引用关系，会破坏源数据。所以在工具集 `src/utils/assist.js` 中新加一个深拷贝的工具函数 `deepCopy`：

```
// assist.js，部分代码省略
function typeOf(obj) {
  const toString = Object.prototype.toString;
  const map = {
    '[object Boolean]'  : 'boolean',
    '[object Number]'   : 'number',
    '[object String]'   : 'string',
    '[object Function]' : 'function',
    '[object Array]'    : 'array',
    '[object Date]'     : 'date',
    '[object RegExp]'   : 'regExp',
    '[object Undefined]': 'undefined',
    '[object Null]'     : 'null',
```

```
    '[object Object]'   : 'object'
  };
  return map[toString.call(obj)];
}
// deepCopy
function deepCopy(data) {
  const t = typeOf(data);
  let o;

  if (t === 'array') {
    o = [];
  } else if ( t === 'object') {
    o = {};
  } else {
    return data;
  }

  if (t === 'array') {
    for (let i = 0; i < data.length; i++) {
      o.push(deepCopy(data[i]));
    }
  } else if ( t === 'object') {
    for (let i in data) {
      o[i] = deepCopy(data[i]);
    }
  }
  return o;
}

export {deepCopy};
```

deepCopy 函数会递归地对数组或对象进行逐一判断，如果某项是数组或对象，再拆分继续判断，而其它类型就直接赋值了，所以深拷贝的数据不会破坏原有的数据（更多深拷贝与浅拷贝的内容，可阅读扩展阅读 1）。

先来看 tree.vue 的代码：

```
<!-- src/components/tree/tree.vue -->
<template>
  <div>
    <tree-node
      v-for="(item, index) in cloneData"
      :key="index"
      :data="item"
      :show-checkbox="showCheckbox"
    ></tree-node>
  </div>
</template>
<script>
  import TreeNode from './node.vue';
  import { deepCopy } from '../../utils/assist.js';

  export default {
    name: 'Tree',
    components: { TreeNode },
    props: {
```

```
      data: {
        type: Array,
        default () {
          return [];
        }
      },
      showCheckbox: {
        type: Boolean,
        default: false
      }
    },
    data () {
      return {
        cloneData: []
      }
    },
    created () {
      this.rebuildData();
    },
    watch: {
      data () {
        this.rebuildData();
      }
    },
    methods: {
      rebuildData () {
        this.cloneData = deepCopy(this.data);
      }
    }
  }
</script>
```

在组件 created 时（以及 watch 监听 data 改变时），调用了 `rebuildData` 方法克隆源数据，并赋值给了 `cloneData`。

在 template 中，先是渲染了一个 node.vue 组件（`<tree-node>`），这一级是 Tree 的根节点，因为 cloneDate 是一个数组，所以这个根节点不一定只有一项，有可能是并列的多项。不过这里使用的 node.vue 还没有用到 Vue.js 的递归组件，它只处理第一级根节点。

`<tree-node>` 组件（node.vue）接收两个 props：

1. showCheckbox：与 tree.vue 的 showCheckbox 相同，只是进行传递；
2. data：node.vue 接收的 data 是一个 Object 而非 Array，因为它只负责渲染当前的一个节点，并递归渲染下一个子节点（即 children），所以这里对 cloneData 进行循环，将每一项节点数据赋给了 tree-node。

递归组件 node.vue

node.vue 是树组件 Tree 的核心，而一个 tree-node 节点包含 4 个部分：

1. 展开与关闭的按钮（+或-）；
2. 多选框；
3. 节点标题；
4. 递归子节点。

先来看 node.vue 的基本结构：

```
<!-- src/components/tree/node.vue -->
<template>
  <ul class="tree-ul">
    <li class="tree-li">
      <span class="tree-expand" @click="handleExpand">
        <span v-if="data.children && data.children.length && !data.expand">+
</span>
        <span v-if="data.children && data.children.length && data.expand">-
</span>
      </span>
      <i-checkbox
        v-if="showCheckbox"
        :value="data.checked"
        @input="handleCheck"
      ></i-checkbox>
      <span>{{ data.title }}</span>
      <tree-node
        v-if="data.expand"
        v-for="(item, index) in data.children"
        :key="index"
        :data="item"
        :show-checkbox="showCheckbox"
      ></tree-node>
    </li>
  </ul>
</template>
<script>
  import iCheckbox from '../checkbox/checkbox.vue';

  export default {
    name: 'TreeNode',
    components: { iCheckbox },
    props: {
      data: {
        type: Object,
        default () {
          return {};
        }
      },
      showCheckbox: {
        type: Boolean,
        default: false
      }
    }
  }
</script>
<style>
  .tree-ul, .tree-li{
    list-style: none;
    padding-left: 10px;
  }
  .tree-expand{
    cursor: pointer;
  }
</style>
```

`props: data` 包含了当前节点的所有信息，比如是否展开子节点（expand）、是否选中（checked）、子节点数据（children）等。

第一部分 expand，如果当前节点不含有子节点，也就是没有 children 字段或 children 的长度是 0，那就说明当前节点已经是最后一级节点，所以不含有展开 / 收起的按钮。

多选框直接使用了第 7 节的 Checkbox 组件（单用模式），这里将 prop: value 和事件 @input 分开绑定，没有使用 `v-model` 语法糖。value 绑定的数据 `data.checked` 表示当前节点是否选中，在点击多选框时，handleCheck 方法会修改 data.checked 数据，下文会分析。这里之所以不使用 `v-model` 而是分开绑定，是因为 @input 里要额外做一些处理，不是单纯的修改数据。

上一节我们说到，一个 Vue.js 递归组件有两个必要条件：name 特性和终结条件。name 已经指定为 `TreeNode`，而这个终结递归的条件，就是 `v-for="(item, index) in data.children"`，当 `data.children` 不存在或为空数组时，自然就不会继续渲染子节点，递归也就停止了。

注意，这里的 `v-if="data.expand"` 并不是递归组件的终结条件，虽然它看起来像是一个可以为 `false` 的判断语句，但它的用处是判断当前节点**的子节点**是否展开（渲染），如果当前节点不展开，那它所有的子节点也就不会展开（渲染）。

上面的代码保留了两个方法 `handleExpand` 与 `handleCheck`，先来看前者。

点击 + 号时，会展开直属子节点，点击 - 号关闭，这一步只需在 `handleExpand` 中修改 data 的 expand 数据即可，同时，我们通过 Tree 的根组件（tree.vue）触发一个自定义事件 `@on-toggle-expand`（上文已介绍）：

```
// node.vue，部分代码省略
import { findComponentUpward } from '../../utils/assist.js';

export default {
  data () {
    return {
      tree: findComponentUpward(this, 'Tree')
    }
  },
  methods: {
    handleExpand () {
      this.$set(this.data, 'expand', !this.data.expand);

      if (this.tree) {
        this.tree.emitEvent('on-toggle-expand', this.data);
      }
    },
  }
}
```

```
// tree.vue，部分代码省略
export default {
  methods: {
    emitEvent (eventName, data) {
      this.$emit(eventName, data, this.cloneData);
    }
  }
}
```

在 node.vue 中，通过 `findComponentUpward` 向上找到了 Tree 的实例，并调用它的 `emitEvent` 方法来触发自定义事件 `@on-toggle-expand`。之所以使用 `findComponentUpward` 寻找组件，而不是用 `$parent`，是因为当前的 node.vue，它的父级不一定就是 tree.vue，因为它是递归组件，父级有可能还是自己。

这里有一点需要注意，修改 data.expand，我们是通过 Vue 的 `$set` 方法来修改，并没有像下面这样修改：

```
this.data.expand = !this.data.expand;
```

这样有什么区别呢？如果直接用上面这行代码修改，发现数据虽然被修改了，但是视图并没有更新（原来是 + 号，点击后还是 + 号）。要理解这里，我们先看下，到底修改了什么。这里的 `this.data`，是一个 props，它是通过上一级传递的，这个上一级有两种可能，一种是递归的 node.vue，一种是根组件 tree.vue，但是递归的 node.vue，最终也是由 tree.vue 传递的，追根溯源，要修改的 `this.data` 事实上是 tree.vue 的 `cloneData`。cloneData 里的节点数据，是不一定含有 `expand` 或 `checked` 字段的，如果不含有，直接通过 `this.data.expand` 修改，这个 expand 就不是可响应的数据，Vue.js 是无法追踪到它的变化，视图自然不会更新，而 `$set` 的用法就是对可响应对象中添加一个属性，并确保这个新属性同样是响应式的，且触发视图更新。总结来说，如果 expand 字段一开始是存在的（不管 true 或 false），不管用哪种方式修改都是可以的，否则必须用 `$set` 修改，结合起来，干脆直接用 $set 了。同理，后文的 `checked` 也是一样。

接下来是整个 Tree 组件最复杂的一部分，就是处理节点的响应状态。你可能会问，不就是选中或取消选中吗，跟 expand 一样，修改数据就行了？如果只是考虑一个节点，的确这样就可以了，但树组件是有上下级关系的，它们分为两种逻辑，当选中（或取消选中）一个节点时：

1. 它下面的所有子节点都会被选中；
2. 如果同一级所有子节点选中时，它的父级也自动选中，一直递归判断到根节点。

第 1 个逻辑相对简单，当选中一个节点时，只要递归地遍历它下面所属的所有子节点数据，修改所有的 `checked` 字段即可：

```
// node.vue，部分代码省略
export default {
  methods: {
    handleCheck (checked) {
      this.updateTreeDown(this.data, checked);

      if (this.tree) {
        this.tree.emitEvent('on-check-change', this.data);
      }
    },
    updateTreeDown (data, checked) {
      this.$set(data, 'checked', checked);

      if (data.children && data.children.length) {
        data.children.forEach(item => {
          this.updateTreeDown(item, checked);
        });
      }
    }
  }
}
```

updateTreeDown 只是向下修改了所有的数据，因为当前节点的数据里，是包含其所有子节点数据的，通过递归遍历可以轻松修改，这也是第 1 种逻辑简单的原因。

再来看第 2 个逻辑，它的难点在于，无法通过当前节点数据，修改到它的父节点，因为拿不到。写到这里，正常的思路应该是在 `this.updateTreeDown(this.data, checked);` 的下面再写一个 `updateTreeUp` 的方法，向上遍历，问题就是，怎样向上遍历，一种常规的思路是，把 `updateTreeUp` 方法写在 tree.vue 里，并且在 node.vue 的 `handleCheck` 方法里，通过 this.tree 调用根组件的 `updateTreeUp`，并且传递当前节点的数据，在 tree.vue 里，要找到当前节点的位置，那还需要一开始在 cloneData 里预先给每个节点设置一个唯一的 key，后面的逻辑读者应该能想到了，就是通过 key 找到节点位置，并向上递归判断……但是，这个方法想着就麻烦。

正常的思路不太好解决，我们就换个思路。一个节点，除了手动选中（或反选），还有就是第 2 种逻辑的被动选中（或反选），也就是说，如果这个节点的所有直属子节点（就是它的第一级子节点）都选中（或反选）时，这个节点就自动被选中（或反选），递归地，可以一级一级响应上去。有了这个思路，我们就可以通过 `watch` 来监听当前节点的子节点是否都选中，进而修改当前的 `checked` 字段：

```
// node.vue，部分代码省略
export default {
  watch: {
    'data.children': {
      handler (data) {
        if (data) {
          const checkedAll = !data.some(item => !item.checked);
          this.$set(this.data, 'checked', checkedAll);
        }
      },
      deep: true
    }
  }
}
```

在 watch 中，监听了 data.children 的改变，并且是深度监听的。这段代码的意思是，当 `data.children` 中的数据的某个字段发生变化时（这里当然是指 checked 字段）,也就是说它的某个子节点被选中（或反选）了，这时执行绑定的句柄 handler 中的逻辑。`const checkedAll = !data.some(item => !item.checked);` 也是一个巧妙的缩写，checkedAll 最终返回结果就是当前子节点是否都被选中了。

这里非常巧妙地利用了递归的特性，因为 node.vue 是一个递归组件，那每一个组件里都会有 watch 监听 `data.children`，要知道，当前的节点有两个"身份"，它既是下属节点的父节点，同时也是上级节点的子节点，它作为下属节点的父节点被修改的同时，也会触发上级节点中的 watch 监听函数。**这就是递归。**

以上就是 Tree 组件的所有内容，完整的代码见： https://github.com/icarusion/vue-component-book/tree/master/src/components/tree

用例：https://github.com/icarusion/vue-component-book/blob/master/src/views/tree.vue

结语

递归就像人类繁衍一样，蕴藏了无限可能，充满着神奇与智慧。

扩展阅读

- 浅拷贝与深拷贝

注：本节部分代码参考 iView。

拓展：Vue.js 容易忽略的 API 详解

前面的 15 小节已经覆盖了 Vue.js 组件的绝大部分内容，但还是有一些 API 容易忽略。本节则对 Vue.js 的一些重要且易忽略的 API 进行详细介绍。

nextTick

nextTick 是 Vue.js 提供的一个函数，并非浏览器内置。nextTick 函数接收一个回调函数 `cb`，在下一个 DOM 更新循环之后执行。比如下面的示例：

```
<template>
  <div>
    <p v-if="show" ref="node">内容</p>
    <button @click="handleShow">显示</button>
  </div>
</template>
<script>
  export default {
    data () {
      return {
        show: false
      }
    },
    methods: {
      handleShow () {
        this.show = true;
        console.log(this.$refs.node);  // undefined
        this.$nextTick(() => {
          console.log(this.$refs.node);  // <p>内容</p>
        });
      }
    }
  }
</script>
```

当 `show` 被置为 true 时，这时 p 节点还未被渲染，因此打印出的是 undefined，而在 nextTick 的回调里，p 已经渲染好了，这时能正确打印出节点。

nextTick 的源码在 https://github.com/vuejs/vue/blob/dev/src/core/util/next-tick.js，可以看到，Vue.js 使用了 `Promise`、`setTimeout` 和 `setImmediate` 三种方法来实现 nextTick，在不同环境会使用不同的方法。

v-model 语法糖

`v-model` 常用于表单元素上进行数据的双向绑定，比如 `<input>`。除了原生的元素，它还能在自定义组件中使用。

v-model 是一个语法糖，可以拆解为 props: value 和 events: input。就是说组件必须提供一个名为 value 的 prop，以及名为 input 的自定义事件，满足这两个条件，使用者就能在自定义组件上使用 `v-model`。比如下面的示例，实现了一个数字选择器：

```
<template>
  <div>
    <button @click="increase(-1)">减 1</button>
    <span style="color: red;padding: 6px">{{ currentValue }}</span>
    <button @click="increase(1)">加 1</button>
  </div>
</template>
<script>
  export default {
    name: 'InputNumber',
    props: {
      value: {
        type: Number
      }
    },
    data () {
      return {
        currentValue: this.value
      }
    },
    watch: {
      value (val) {
        this.currentValue = val;
      }
    },
    methods: {
      increase (val) {
        this.currentValue += val;
        this.$emit('input', this.currentValue);
      }
    }
  }
</script>
```

props 一般不能在组件内修改，它是通过父级修改的，因此实现 v-model 一般都会有一个 `currentValue` 的内部 data，初始时从 value 获取一次值，当 value 修改时，也通过 watch 监听到及时更新；组件不会修改 value 的值，而是修改 currentValue，同时将修改的值通过自定义事件 `input` 派发给父组件，父组件接收到后，由父组件修改 value。所以，上面的数字选择器组件可以有下面两种使用方式：

```
<template>
  <InputNumber v-model="value" />
</template>
<script>
  import InputNumber from '../components/input-number/input-number.vue';

  export default {
    components: { InputNumber },
    data () {
      return {
        value: 1
      }
    }
  }
</script>
```

或：

```
<template>
  <InputNumber :value="value" @input="handleChange" />
</template>
<script>
  import InputNumber from '../components/input-number/input-number.vue';

  export default {
    components: { InputNumber },
    data () {
      return {
        value: 1
      }
    },
    methods: {
      handleChange (val) {
        this.value = val;
      }
    }
  }
</script>
```

如果你不想用 `value` 和 `input` 这两个名字，从 Vue.js 2.2.0 版本开始，提供了一个 `model` 的选项，可以指定它们的名字，所以数字选择器组件也可以这样写：

```
<template>
  <div>
    <button @click="increase(-1)">减 1</button>
    <span style="color: red;padding: 6px">{{ currentValue }}</span>
    <button @click="increase(1)">加 1</button>
  </div>
</template>
<script>
  export default {
    name: 'InputNumber',
    props: {
      number: {
        type: Number
      }
    },
    model: {
      prop: 'number',
      event: 'change'
    },
    data () {
      return {
        currentValue: this.number
      }
    },
    watch: {
      value (val) {
        this.currentValue = val;
```

```
      }
    },
    methods: {
      increase (val) {
        this.currentValue += val;
        this.$emit('number', this.currentValue);
      }
    }
  }
</script>
```

在 model 选项里，就可以指定 prop 和 event 的名字了，而不一定非要用 value 和 input，因为这两个名字在一些原生表单元素里，有其它用处。

.sync 修饰符

如果你使用过 Vue.js 1.x，一定对 `.sync` 不陌生。在 1.x 里，可以使用 .sync 双向绑定数据，也就是父组件或子组件都能修改这个数据，是双向响应的。在 Vue.js 2.x 里废弃了这种用法，目的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。

不过在 Vue.js 2.3.0 版本，又增加了 `.sync` 修饰符，但它的用法与 1.x 的不完全相同。2.x 的 .sync 不是真正的双向绑定，而是一个语法糖，修改数据还是在父组件完成的，并非在子组件。

仍然是数字选择器的示例，这次不用 v-model，而是用 .sync，可以这样改写：

```
<template>
  <div>
    <button @click="increase(-1)">减 1</button>
    <span style="color: red;padding: 6px">{{ value }}</span>
    <button @click="increase(1)">加 1</button>
  </div>
</template>
<script>
  export default {
    name: 'InputNumber',
    props: {
      value: {
        type: Number
      }
    },
    methods: {
      increase (val) {
        this.$emit('update:value', this.value + val);
      }
    }
  }
</script>
```

用例：

```
<template>
  <InputNumber :value.sync="value" />
</template>
<script>
```

```
  import InputNumber from '../components/input-number/input-number.vue';

  export default {
    components: { InputNumber },
    data () {
      return {
        value: 1
      }
    }
  }
</script>
```

看起来要比 v-model 的实现简单多，实现的效果是一样的。v-model 在一个组件中只能有一个，但 .sync 可以设置很多个。.sync 虽好，但也有限制，比如：

- **不能**和表达式一起使用（如 `v-bind:title.sync="doc.title + '!'"` 是无效的）；
- **不能**用在字面量对象上（如 `v-bind.sync="{ title: doc.title }"` 是无法正常工作的）。

$set

在上一节已经介绍过 `$set`，有两种情况会用到它：

1. 由于 JavaScript 的限制，Vue 不能检测以下变动的数组：
    1. 当利用索引直接设置一个项时，例如：`this.items[index] = value`；
    2. 当修改数组的长度时，例如：`vm.items.length = newLength`。
2. 由于 JavaScript 的限制，**Vue 不能检测对象属性的添加或删除。**

举例来看，就是：

```
// 数组
export default {
  data () {
    return {
      items: ['a', 'b', 'c']
    }
  },
  methods: {
    handler () {
      this.items[1] = 'x';  // 不是响应性的
    }
  }
}
```

使用 `$set`：

```
// 数组
export default {
  data () {
    return {
      items: ['a', 'b', 'c']
    }
  },
  methods: {
```

```
    handler () {
      this.$set(this.items, 1, 'x');  // 是响应性的
    }
  }
}
```

以对象为例：

```
// 对象
export default {
  data () {
    return {
      item: {
        a: 1
      }
    }
  },
  methods: {
    handler () {
      this.item.b = 2;  // 不是响应性的
    }
  }
}
```

使用 `$set`：

```
// 对象
export default {
  data () {
    return {
      item: {
        a: 1
      }
    }
  },
  methods: {
    handler () {
      this.$set(this.item, 'b', 2);  // 是响应性的
    }
  }
}
```

另外，数组的以下方法，都是**可以**触发视图更新的，也就是响应性的：

`push()`、`pop()`、`shift()`、`unshift()`、`splice()`、`sort()`、`reverse()`。

还有一种小技巧，就是先 copy 一个数组，然后通过 index 修改后，再把原数组整个替换，比如：

```
handler () {
  const data = [...this.items];
  data[1] = 'x';
  this.items = data;
}
```

计算属性的 set

计算属性（computed）很简单，而且也会大量使用，但大多数时候，我们只是用它默认的 get 方法，也就是平时的常规写法，通过 computed 获取一个依赖其它状态的数据。比如：

```
computed: {
  fullName () {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

这里的 fullName 事实上可以写为一个 Object，而非 Function，只是 Function 形式是我们默认使用它的 get 方法，当写为 Object 时，还能使用它的 set 方法：

```
computed: {
  fullName: {
    get () {
      return `${this.firstName} ${this.lastName}`;
    },
    set (val) {
      const names = val.split(' ');
      this.firstName = names[0];
      this.lastName = names[names.length - 1];
    }
  }
}
```

计算属性大多时候只是读取用，使用了 set 后，就可以写入了，比如上面的示例，如果执行 `this.fullName = 'Aresn Liang'`，computed 的 set 就会调用，firstName 和 lastName 会被赋值为 Aresn 和 Liang。

剩余值得注意的 API

还有一些 API，可能不常用，也比较简单，只需知道就好，本册不详细展开介绍，可以通过指引到 Vue.js 文档查看。

delimiters

改变纯文本插入分隔符，Vue.js 默认的是 `{{ }}`，如果你使用其它一些后端模板，比如 Python 的 Tornado 框架，那 Vue.js 和 Tornado 的 `{{ }}` 就冲突了，这时用它可以修改为指定的分隔符。

v-once

只渲染元素和组件**一次**。随后的重新渲染，元素/组件及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能。

vm.$isServer

当前 Vue 实例是否运行于服务器，如果你的组件要兼容 SSR，它会很有用。

inheritAttrs

一些原生的 html 特性，比如 `id`，即使没有定义 props，也会被集成到组件根节点上，设置 inheritAttrs 为 false 可以关闭此特性。

errorHandler

使用 `errorHandler` 可以进行异常信息的获取。

watch

监听状态的变化，用的也很多了，但它和 computed 一样，也有 Object 的写法，这样能配置更多的选项，比如：

- handler 执行的函数
- deep 是否深度
- immediate 是否立即执行

完整的配置可以阅读文档。

comments

开启会保留 html 注释。

transition

内置的组件，可做过渡效果，比如 CSS 的高度从 0 到 auto（使用纯 CSS 是无法实现动画的）。

结语

彻底掌握一门语言（框架），不需要阅读它所有的源码，但至少要阅读它所有的 API。

拓展：Vue.js 面试、常见问题答疑

在过去的很多面试中，我会经常问候选人一些关于 Vue.js 的问题。这些问题从题面来看很简单，但仔细想又不是那么简单，不同的人，会答出不同的层次，从而更好地了解一个人对 Vue.js 的理解程度。

题目

v-show 与 v-if 区别

第一题应该是最简单的，提这个问题，也是想让候选人不那么紧张，因为但凡用过 Vue.js，多少知道 `v-show` 和 `v-if` 的区别，否则就没得聊了。不过这最简单的一道题，有三个层次，我会逐一追问。首先，基本所有人都会说到：

`v-show` 只是 CSS 级别的 `display: none;` 和 `display: block;` 之间的切换，而 `v-if` 决定是否会选择代码块的内容（或组件）。

回答这些，已经可以得到 50 分了，紧接着我会追问，什么时候用 v-show，什么时候用 v-if？到这里一部分人会比较吞吐，可能是知道，但表达不出来。我比较倾向的回答是：

频繁操作时，使用 `v-show`，一次性渲染完的，使用 `v-if`，只要意思对就好。

第二问可以得到 80 分了，最后一问很少有人能答上：**那使用 `v-if` 在性能优化上有什么经验？**这是一个加分项，要对 Vue.js 的组件编译有一定的理解。说一下期望的答案：

因为当 `v-if="false"` 时，内部组件是不会渲染的，所以在特定条件才渲染部分组件（或内容）时，可以先将条件设置为 `false`，需要时（或异步，比如 $nextTick）再设置为 `true`，这样可以优先渲染重要的其它内容，合理利用，可以进行性能优化。

绑定 class 的数组用法

动态绑定 class 应该不陌生吧，这也是最基本的，但是这个问题却有点绕，什么叫**绑定 class 的数组用法？**我们看一下，最常用的绑定 class 怎么写：

```
<template>
  <div :class="{show: isShow}">内容</div>
</template>
<script>
  export default {
    data () {
      return {
        isShow: true
      }
    }
  }
</script>
```

绑定 class 的对象用法能满足大部分业务需求，不过，在复杂的场景下，会用到**数组**，来看示例：

```
<template>
  <div :class="classes"></div>
</template>
<script>
  export default {
    computed: {
      classes () {
        return [
          `${prefixCls}`,
          `${prefixCls}-${this.type}`,
          {
            [`${prefixCls}-long`]: this.long,
            [`${prefixCls}-${this.shape}`]: !!this.shape,
            [`${prefixCls}-${this.size}`]: this.size !== 'default',
            [`${prefixCls}-loading`]: this.loading != null && this.loading,
            [`${prefixCls}-icon-only`]: !this.showSlot && (!!this.icon ||
!!this.customIcon || this.loading),
            [`${prefixCls}-ghost`]: this.ghost
          }
        ];
      }
    }
  }
</script>
```

示例来自 iView 的 Button 组件，可以看到，数组里，可以是固定的值，还有动态值（对象）的混合。

计算属性和 watch 的区别

回答该题前，一般都会思考一下。很多人会偏题，直接去答计算属性和 watch 怎么用，这是不得分的，因为题目是问**区别**，并不是用法。

计算属性是自动监听依赖值的变化，从而动态返回内容，监听是一个过程，在监听的值变化时，可以触发一个回调，并做一些事情。

所以区别来源于用法，只是需要动态值，那就用计算属性；需要知道值的改变后执行业务逻辑，才用 watch，用反或混用虽然可行，但都是不正确的用法。

这个问题会延伸出几个问题：

1. computed 是一个对象时，它有哪些选项？
2. computed 和 methods 有什么区别？
3. computed 是否能依赖其它组件的数据？
4. watch 是一个对象时，它有哪些选项？

问题 1，已经在 16 小节介绍过，有 get 和 set 两个选项。

问题 2，methods 是一个方法，它可以接受参数，而 computed 不能；computed 是可以缓存的，methods 不会；一般在 `v-for` 里，需要根据当前项动态绑定值时，只能用 methods 而不能用 computed，因为 computed 不能传参。

问题 3，computed 可以依赖其它 computed，甚至是其它组件的 data。

问题 4，第 16 小节也有提到，有以下常用的配置：

- handler 执行的函数
- deep 是否深度
- immediate 是否立即执行

事件修饰符

这个问题我会先写一段代码：

```
<custom-component>内容</custom-component>
```

然后问：**怎样给这个自定义组件 custom-component 绑定一个 `原生` 的 click 事件？**

我一开始并不会问什么是事件修饰符，但是如果候选人说 `<custom-component @click="xxx">`，就已经错了，说明它对这个没有概念。这里的 `@click` 是自定义事件 click，并不是原生事件 click。绑定原生的 click 是这样的：

```
<custom-component @click.native="xxx">内容</custom-component>
```

该问题会引申很多，比如常见的事件修饰符有哪些？如果你能说上 `.exact`，说明你是个很爱探索的人，会大大加分哦。

> .exact 是 Vue.js 2.5.0 新加的，它允许你控制由精确的系统修饰符组合触发的事件，比如：

```
<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->
<button @click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 被按下的时候才触发 -->
<button @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 @click.exact="onClick">A</button>
```

你可能还需要了解常用的几个事件修饰符：

- `.stop`
- `.prevent`
- `.capture`
- `.self`

而且，事件修饰符在连用时，是有先后顺序的。

组件中 data 为什么是函数

为什么组件中的 data 必须是一个函数，然后 return 一个对象，而 new Vue 实例里，data 可以直接是一个对象？

因为组件是用来复用的，JS 里对象是引用关系，这样作用域没有隔离，而 new Vue 的实例，是不会被复用的，因此不存在引用对象的问题。

keep-alive 的理解

这是个概念题，主要考察候选人是否知道这个用法。简单说，就是把一个组件的编译缓存起来。在第 14 节有过详细介绍，也可以看看 Vue.js 的文档。

递归组件的要求

回答这道题，首先你得知道什么是**递归组件**。而不到 10% 的人知道递归组件。其实在实际业务中用的确实不多，在独立组件中会经常使用，第 14 节和 15 节专门讲过递归组件。那回到问题，递归组件的要求是什么？主要有两个：

- 要给组件设置 **name**；
- 要有一个明确的结束条件。

自定义组件的语法糖 v-model 是怎样实现的

在第 16 节已经详细介绍过，这里的 v-model，并不是给普通输入框 `<input />` 用的那种 v-model，而是在自定义组件上使用。既然是语法糖，就能够还原，我们先还原一下：

```
<template>
  <div>
    {{ currentValue }}
    <button @click="handleClick">Click</button>
  </div>
</template>
<script>
  export default {
    props: {
      value: {
```

```
        type: Number,
        default: 0
      }
    },
    data () {
      return {
        currentValue: this.value
      }
    },
    methods: {
      handleClick () {
        this.currentValue += 1;
        this.$emit('input', this.currentValue);
      }
    },
    watch: {
      value (val) {
        this.currentValue = val;
      }
    }
  }
</script>
```

这个组件中，只有一个 props，但是名字叫 `value`，内部还有一个 `currentValue`，当改变 currentValue 时，会触发一个自定义事件 `@input`，并把 currentValue 的值返回。这就是一个 `v-model` 的语法糖，它要求 props 有一个叫 `value` 的项，同时触发的自定义事件必须叫 `input`。这样就可以在自定义组件上用 `v-model` 了：

```
<custom-component v-model="value"></custom-component>
```

如果你能说到 `model` 选项，绝对是加分的。

Vuex 中 mutations 和 actions 的区别

主要的区别是，actions 可以执行异步。actions 是调用 mutations，而 mutations 来修改 store。

Render 函数

这是比较难的一题了，因为很少有人会去了解 Vue.js 的 Render 函数，因为基本用不到。Render 函数的内容本小册已经很深入的讲解过了，遇到这个问题，一般可以从这几个方面来回答：

- 什么是 Render 函数，它的使用场景是什么。
- createElement 是什么？
- Render 函数有哪些常用的参数？

说到 Render 函数，就要说到虚拟 DOM（Virtual DOM）,Virtual DOM 并不是真正意义上的 DOM，而是一个轻量级的 JavaScript 对象，在状态发生变化时，Virtual DOM 会进行 Diff 运算，来更新只需要被替换的 DOM，而不是全部重绘。

它的使用场景，就是完全发挥 JavaScript 的编程能力，有时需要结合 JSX 来使用。

createElement 是 Render 函数的核心，它构成了 Vue Virtual DOM 的模板，它有 3 个参数：

```
createElement () {
```

```
  // {String | Object | Function}
  // 一个 HTML 标签，组件选项，或一个函数
  // 必须 return 上述其中一个
  'div',
    // {Object}
    // 一个对应属性的数据对象，可选
    // 您可以在 template 中使用
    {
    // 详细的属性
  },
    // {String | Array}
    // 子节点（VNodes），可选
    [
    createElement('h1', 'hello world'),
    createElement(MyComponent, {
      props: {
        someProps: 'foo'
      }
    }),
    'bar'
  ]
}
```

常用的参数，主要是指上面第二个参数里的值了，这个比较多，得去看 Vue.js 的文档。

怎样理解单向数据流

这个概念出现在组件通信。父组件是通过 prop 把数据传递到子组件的，但是这个 prop 只能由父组件修改，子组件不能修改，否则会报错。子组件想修改时，只能通过 `$emit` 派发一个自定义事件，父组件接收到后，由父组件修改。

一般来说，对于子组件想要更改父组件状态的场景，可以有两种方案：

1. 在子组件的 data 中拷贝一份 prop，data 是可以修改的，但 prop 不能：

```
export default {
  props: {
    value: String
  },
  data () {
    return {
      currentValue: this.value
    }
  }
}
```

2. 如果是对 prop 值的转换，可以使用计算属性：

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

如果你能提到 v-model 实现数据的双向绑定、.sync 用法，会大大加分的，这些在第 16 节已经详细介绍过。

生命周期

Vue.js 生命周期 主要有 8 个阶段：

- 创建前 / 后（beforeCreate / created）：在 beforeCreate 阶段，Vue 实例的挂载元素 el 和数据对象 data 都为 undefined，还未初始化。在 created 阶段，Vue 实例的数据对象 data 有了，el 还没有。
- 载入前 / 后（beforeMount / mounted）：在 beforeMount 阶段，Vue 实例的 $el 和 data 都初始化了，但还是挂载之前为虚拟的 DOM 节点，data 尚未替换。在 mounted 阶段，Vue 实例挂载完成，data 成功渲染。
- 更新前 / 后（beforeUpdate / updated）：当 data 变化时，会触发 beforeUpdate 和 updated 方法。这两个不常用，且不推荐使用。
- 销毁前 / 后（beforeDestroy / destroyed）：beforeDestroy 是在 Vue 实例销毁前触发，一般在这里要通过 removeEventListener 解除手动绑定的事件。实例销毁后，触发 destroyed。

组件间通信

本小册一半的篇幅都在讲组件的通信，如果能把这些都吃透，基本上 Vue.js 的面试就稳了。

这个问题看似简单，却比较大，回答时，可以拆分为几种场景：

1. 父子通信：

   父向子传递数据是通过 props，子向父是通过 events（$emit$）；通过父链/子链也可以通信（parent / $children）；`ref` 也可以访问组件实例；provide / inject API。

2. 兄弟通信：

   Bus；Vuex；

3. 跨级通信：

   Bus；Vuex；provide / inject API。

除了常规的通信方法，本册介绍的 dispatch / broadcast 和 findComponents 系列方法也可以说的，如果能说到这些，说明你对 Vue.js 组件已经有较深入的研究。

路由的跳转方式

一般有两种：

1. 通过 `<router-link to="home">`，router-link 标签会渲染为 `<a>` 标签，在 template 中的跳转都是用这种；
2. 另一种是编程式导航，也就是通过 JS 跳转，比如 `router.push('/home')`。

Vue.js 2.x 双向绑定原理

这个问题几乎是面试必问的，回答也是有深有浅。基本上要知道核心的 API 是通过 `Object.defineProperty()` 来劫持各个属性的 setter / getter，在数据变动时发布消息给订阅者，触发相应的监听回调，这也是为什么 Vue.js 2.x 不支持 IE8 的原因（IE 8 不支持此 API，且无法通过 polyfill 实现）。

Vue.js 文档已经对 深入响应式原理 解释的很透彻了。

什么是 MVVM，与 MVC 有什么区别

MVVM 模式是由经典的软件架构 MVC 衍生来的。当 View（视图层）变化时，会自动更新到 ViewModel（视图模型），反之亦然。View 和 ViewModel 之间通过双向绑定（data-binding）建立联系。与 MVC 不同的是，它没有 Controller 层，而是演变为 ViewModel。

ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View 和 Model 之间的同步工作是由 Vue.js 完成的，我们不需要手动操作 DOM，只需要维护好数据状态。

结语

一个人的简历，是由简单到复杂再到简单，技术是无止尽的，接触的越多，越能感到自己的渺小。

拓展：如何做好一个开源项目（上篇）

iView 的故事

毕业四年以来，我一直觉得自己是一个很幸运的人，幸运参与过创业，幸运一路有大牛带，幸运开源了 iView 项目。

2016 年初，我还是一名普通的前端工程师，那时候还是 Vue.js 1.x 的时代，知名度也远不如现在，在大部分人眼中，Vue.js 就是一个轻量级的 Angular。

我所在的公司是做 to B 业务的，与大部分公司一样，那时主导 jQuery，把 Vue.js 引入团队乃至整个公司，是一件很不易的事，因为不是你自己在用，你要说服所有人用，现在看来，当初的决定是很正确的。如果你有兴趣，欢迎阅读扩展阅读 1，那篇文章详细介绍了我的 Vue.js “推广史”。

到了 16 年 7 月，我们团队已经完全认可了 Vue.js + Webpack 的技术栈，一直践行至今。一个偶然的机会，公司举办了一个创新大赛，作为可视化团队，我提议做一套自己的组件库。当初也只是一个想法，抱着试试看的态度就报名了。比赛的目的只是呼吁大家创新，后续就没有动作了，但是我没有就此放下，从那时起，几乎全职开发 iView，两年半来，我每天的工作基本就是维护 iView。不得不说，一个成功的开源项目，必须有 leader 的认可和公司的支持，缺一不可。

那时不比现在，优秀的 Vue.js 组件库非常少，文献更是少得可怜。一开始做 iView 没什么头绪，只是规划了一张 MindNode 脑图，罗列了一期要做的所有组件。这里有一个插曲，因为这张 todo 的脑图有贴在 GitHub 的 Readme，以至于一开始大部分 issues 都是问这个脑图是用什么软件做的，因为做的很好看，索性写了个提问须知，注明了脑图是用 MindNode 制作的。

起初对开源工作不是很了解，连一个编译工程都搭不起来，想了数日也搞不定。与大部分人一样，一开始也是要参考其它人的开源项目（就像现在很多人参考 iView 是一样的），那是我参考 Vux（移动端的 Vue.js 组件库，当时有 6k star），向作者捐了杯咖啡钱，顺便加了好友（因为是支付宝捐赠，能加好友），请教了工程的问题，这才一步步搭起来，现在想想真是幸运。

万事开头难，把地基搭好，剩下的都是添砖加瓦的事。差不多 16 年底，iView 一期完成了（43 个组件），也发布了第一个正式版，有了第一批用户（当然，主要还是自己公司），然而在那个时候，又做了一个在今天看来稍晚但正确的决定——支持 Vue.js 2。

虽然是开源项目，但主要还是先满足自己团队和公司的需求，那个时候所有的项目都是 Vue.js 1.x 的，还没有用 Vue.js 2，但 2.x 已经发布有一段时间了，以至于在 2017 年初，GitHub 有很多 issues 问什么时候支持 2.x。很长一段时间没有升级到 2.x，主要因为要支持公司的项目，而我当时也觉得升级到 2.x 是一个很耗时的工作，就没有支持。后来，有一个人提交了一个非常大的 PR，将所有组件都支持了 2.x，而且只用了一个周末的时间。这件事让我意识到支持 Vue.js 2.x 的重要性了，于是花了 2 周时间研究，并升级到了 2.x（那个 pr 没有直接合并，而是参考，因为很多地方有 bug）。接下来的几个月，都在不断维护新的 iView，使用者也不断增多，社区又重新活跃起来。

如果当初没有及时转型（现在看来仍然是晚了），还在一味地维护 Vue.js 1.x 版本，那 iView 也许早就完了。有了好的开头，就要不放弃，不抛弃，坚持做下去，认真处理每一个 PR，因为质量是一个开源组件库最重要的，但并不是每一个 PR 都适合 merge，即使作者付出了很多时间提交这个 PR，也会有质量问题。

有了不错的口碑，在 2018 年继续完善，目前已经是同类产品里功能最丰富的组件库，而且在 18 年的 7 月 28 日（iView 两周年）成功举行了 3.0 发布会，一切都在朝着更好的一面发展着……

这就是我的 iView 开源的故事，一个成功的开源项目，或许带有一点小幸运，因为如今的同类开源产品已经数不胜数了，不过确实有很多值得分享的地方，如果你打算做一个开源项目，希望下面的些许经验能够帮助到你。

不要盲目造轮子

每一个做开源项目的开发者，都是有目的的，如果你做一个开源项目没有任何目的，那你的目的多半是要造个轮子来提升下技术。

中国当前的开源环境和氛围虽不是很好，“拿来主义”者居多，但相比几年前，已是不错了，更多的人喜欢把自己杰出的代码开源出来。目前多数“正经”的开源项目，都是 KPI 导向的，不能说项目不好，只是一定时间后，可能就不维护了，而开源项目不仅仅要解决问题，持续维护也是观望着最在意的。

相比 React，Vue.js 的组件库开源项目要多的多，每隔一段时间就能看到几个新的，可能是因为 Vue.js 更容易上手，开发者很活跃。既然已经有这么的同类项目了，那为什么还要持续造轮子呢？因为中国的开发者就是喜欢自己折腾一个，而不是去维护一个现成的。

就以 Vue.js 组件库的开源现象来说，源源不断的轮子，主要还是市面上已有的组件库不能完全满足自己的业务，主要还是 UI 层面的，单论功能，市面上成熟的组件库足够完善，甚至超出你的业务需求。不好说这种现象是好是坏，好的是从头开发一个组件库，对技术的提升还是有的，它会督促你学习别人的代码，来改良自己的代码，否则造一个还不如别人的轮子也就没意义了。对于公司来说，也有了自己的 UI 组件库和规范（虽然很多只是 fork 后改的），感觉用自己的东西，对内对外都是一件很自豪的事。不好的就是，这是一种程序员向产品或 UI 或 leader 的妥协，因为没办法说服他们使用市面上成熟的组件库，否则，完全可以把造轮子的精力用于维护一个成熟的项目。

造轮子这个词，似乎成了前端圈的代名词，的确，前端的造轮子能力是极强的，但不能盲目造轮子。如果你只是仿照其它开源项目练习，那就不说了，如果是想认真做一个开源项目，而不是造个轮子玩玩，那一定要构思好你要做的东西。

一般来说，在决定做一个开源项目前，都是要做市场调研的，你要很清楚的知道，自己做的项目弥补了同类产品的哪些不足，或者有哪些新的特性，因为它们是用户选择你的开源项目的主要依据，否则内容都一样，为什么不选一个成熟的呢。

如果某个开源项目已经很不错，但你希望基于它进行改造，但不是以 PR 的形式，那你可以在开源协议允许的前提下，fork 后，基于某个 release 独立维护。

每个开源项目都有一个核心的功能，或是解决用户的核心痛点，比如 iView 就是解决用户建站的问题，相比同类产品，它的特点就是组件丰富程度最高，功能也是最全面的，生态完善，有技术支持渠道。如果你的开源项目解决的问题是其它任何开源项目没有的，那用户很有可能会使用。但对于相同功能的，用户更倾向于选择还在维护的、star 数多的（star 多，说明这个项目的关注度高，更活跃）。所以，在

做开源时，你不仅要知道自己产品的核心技术和特性，还要了解市面同类产品，去其糟粕，取其精华，不断更新和修复 bug，逐渐就会获得第一批用户。

做了东西要用

虽然我这两年的工作基本全是在维护 iView 开源项目，但偶尔也会穿插做几个项目，因为使用了，才能更好地了解自己的开源项目。

你可能会问，我每天都在维护 iView，还有我不了解的吗？还真是，维护和使用完全不一样，在开发组件库时，往往只聚焦在某个组件上，我们定义了 API，然后通过文档告诉使用者怎么用，有些功能是实现了，然而维护者只知道提供了这个功能，却不知在实际项目中好不好用，能用和好用是两个概念。

很多 feature，是自己用了才提炼出来的。比如 iView 的 单元格组件 Cell 和 相对时间组件 Time，都是我在做项目时发现并增加的，没有项目的支持，可能觉得这个组件没什么意义，完全应该由使用者在业务里自己写，作为基础组件后，使用体验确实好很多。

如果你足够重视你的开源项目，应该亲自使用并且安利别人使用来收集反馈，对于组件库来说，细节是很重要的，而很多细节与我们的用户习惯有关，比如 iView 在 3.0 版本开始，对按钮 Button 组件提供了一个新的 props： `to`，用于指定跳转路径，之前版本如果点击按钮跳转，必须监听它的 `@click` 事件，然后通过 vue-router 的编程式导航（也就是通过 JS）跳转，如果不亲自使用，绝对不知道这种原先的跳转模式写起来有多麻烦。如果你足够注重细节，这个 Button 组件在跳转时，还应该支持键盘的 Command 键（Windows 为 Ctrl，要做兼容）在新窗口打开链接。这些细节都与使用习惯息息相关，所以，一定要多用用自己的开源项目，在一个个小细节都处理好后，你会的开源项目自然会蒸蒸日上。

第一批用户

开源项目做好后，要获得第一批使用者。现在的环境，大多是公司或团队主导做开源，个人的很少，所以你的公司或团队自然就是第一批用户，做开源的主要目的，也是服务他们。

第一批用户也可以算是开源项目的小白鼠，一开始说服全公司使用，还是比较困难的，可以先小范围推广使用。公司都希望自己的产品稳定，而新的开源项目前期必定会有不少 bug，在经历几个小项目试水后，再尝试向更多的团队推广。因为有了成功案例，又是“自家”的开源项目，给自己人推广还是比较容易的。

在第一批用户的推广中，你的 leader 可能会起到决定性作用。你的开源项目，八成市场上已经有同类的了，不得不承认，即使让“自己人”做技术选型，也更期望选市面上成熟的，这种情况，就需要你的 leader 出来“拍板”了，否则，你的开源项目也许永远不会有第一批用户来试错。记得当时我在团队推广 Vue.js 时，起初也是很多质疑，有一部分人坚持使用 jQuery + 前端模板，得到 leader 的认可后，试水了几个项目，大家都能感受到 Vue.js 和 webpack 带来的高效开发体验，从此就没人再用 jQuery 了。不过呢，你最好确保你的开源项目质量还不错，否则这锅就得 leader 替你背了。

与市面上同类成熟的开源项目相比，你的开源项目最大的优势就是你能为第一批用户提供优质的“售后”服务。如果是同事的问题，你可以很快直接解答；如果是通过 GitHub 提交的 issues（前期不会很多了），尽量在半小时内回复，这样使用者会觉得你确实在用心维护这个开源项目。通过前期的口碑积累，你的第一批用户也成为了最忠实的用户，这时可以组建微信或 QQ 群，更直接地提供“售后”，而这第一批忠实用户，会成为后期推广的重要人脉。

生态

生态（ecosystem）不是与生俱来的，当你的开源项目有了一定的规模后，可以考虑发展生态体系。比如 iView，起初只是一个组件库，后续逐渐提供了 iview-project 工程、具有 GUI 的 iView CLI（这个概念可比 Vue CLI 3 早了一年）、后台模板 iview-admin、支持 Vue CLI 3 的 vue-cli-plugin-iview，以及业务组件 iview-area 和 markdown 编辑器 iview-editor，再到后来支持 SSR、TS。

完善的生态体系对于新用户来说，可以最快速搭建产品，减少学习和开发成本；对于观望者（正在决定是否使用的人）来说，更愿意选择生态完善的开源项目。所以，你的开源项目生态越完善，使用者也会越多。

生态的另一个好处，就是让用户产生依赖。最典型的例子就是 Vue 全家桶，一般刚接触的人，只会用个 Vue.js，再后来 vue-router、vuex 都是必须的了，再后来搭配一个三方的组件库，比如 iView，各种业务都能轻松应对。一旦用户对你的生态足够依赖，就很难更换技术栈了，因为生态的深入，更换成本很大，这也是很多企业的老项目所谓的“历史原因”。

生态的建设，不一定都是官方的行为，但是最核心的还是要自己维护，用户既然选择你的开源项目，也就意味着信任你的技术实力，放心用你的生态。项目到了一定规模，自然有不少第三方的开发者一起建设生态，这些 contributors 都是最有价值的开发者，尽量联系他们，一起来贡献更多的代码。

对于 Vue.js 组件库来说，生态一般分为脚手架、后台模板和业务组件。最新的 Vue CLI 3 提供了插件机制，现在的主流做法都是提供一个类似 vue-cli-plugin-iview 的插件，很少有单独提供自己的工程了，在文档里，要推荐使用者用 Vue CLI 3 来管理项目，享受 Vue 的生态。后台模板是开箱即用的，默认配置好了路由、权限管理、多语言、登录等常规的后台系统功能，使用者 down 下来后，稍作修改就能很快开发自己的后台管理系统，主流的 UI 组件库，都会提供自家的后台模板，当然也有第三方专注在做后台模板的。最后一类业务组件，比如城市级联选择器 iview-area，它基于 iView 的基础组件开发，但又不是基础组件，所以不能归到 iView 里，只能作为独立组件单独维护，业务组件理论上使用者可以自己封装，但是重复性的工作，还是交给社区做吧，这就是开源。

下一篇，将介绍：

- **持续运营**
- **国际化**
- **让更多的人参与**
- **让 Robot 来做“坏人”**
- **赞助与商业化**

扩展阅读

- 2016我的心路历程：从 Vue 到 Webpack 到 iView

拓展：如何做好一个开源项目（下篇）

持续运营

项目有了一定的规模和进展后，需要持续运营，让更多的人知道和使用。运营并不是个技术活，对于程序员来说，还是或缺的技能。最简单的运营手段，就是在一些技术社区分享“软文”，iView 在早期就是这样做的，还总结出了一个 “500 star 定律”，也就是说，每一次分享文章，差不多能在 GitHub 带来 500 个 star。star 对于一个开源项目来说，还是蛮重要的，它直接决定了用户是否会选择你的项目，但用户都是程序员，又不傻，如果项目质量低，star 就变的一文不值了，还会坏了口碑。切记，不要刷 star，前端圈堪比娱乐圈，会被针对的很惨。

当然，不是什么内容都能发，比如更新日志，最好就不要发了，除非像 2.0、3.0 这种大版本。即使是发大版本的更新内容，也不是说把更新日志一贴就完事，如果你足够重视你的开源项目，就应该重视每一篇文章，把更新的核心思路说清楚。典型的案例可以参考 iView：

- iView 发布 3.0 版本，以及开发者社区等 5 款新产品
- iView 近期的更新，以及那些“不为人知”的故事
- iView 发布后台管理系统 iview-admin，没错，它就是你想要的

标题的重要性就不必说了，在信息爆炸的时代，你的标题不够吸引人，根本没人看。

目前，有几个社区是值得关注和积累粉丝的：

- 掘金：比较活跃的程序员社区，前端属性较浓，社区运营做的很好，对开源项目有扶持，相关的文章首次亮相，官方都会给予一定的资源支持。
- 知乎：流量最大的社区，大 V 属性，如果你是初入知乎，可以把文章投稿到热门的专栏，比如前端评论外刊、前端之巅等。因为自己起初是没有粉丝订阅的，发表了也不会有人看到，投稿就不一样了。而且，被某个大 V 赞一下，那效果就像中奖。
- v2ex：不用解释，就是很火的社区。

开源项目，一般都会在 GitHub 托管，不过也可以在开源中国（Gitee）同步一份，每个版本的更新日志，可以以新闻的形式，向开源中国投稿。开源中国在国内还是有一定的影响力的。

除了发表文章，一些技术分享大会也可以关注，可以以公司的名义申请成为嘉宾做分享。如果有机会，还可以到其它公司做技术分享，尤其是大厂商，这些都是难得推广开源项目的好机会。你的开源项目，如果有几个 BAT 这类的大厂使用，那会成为维护者、社区用户和观望者的信心来源。

还有发布会。在国内，开源项目搞发布会的，据我所知只有 iView。没错，18 年 7 月，iView 搞了一场新品发布会，线下进行，线上同步直播，当时有超过 2 万的在线用户观看，推广效果还是不错的。一场“合格”的活动，要分**活动前**、**活动中**、**活动后**。活动开始前一个月，就要散布消息，让用户有个初步印象，之间还可以爆料一些活动热点；活动进行中，要有专人负责现场，还要与观众互动；活动结束后，要加个班，把核心内容整理为文字，在第一时间通过官方渠道发表出来。这种大规模的活动，没有公司支持，个人很难完成的，因为这不是一两个人的事，需要很多工作人员一起完成，幕前幕后、直播的网络、现场 wifi，还要应对各种突破状况，不过，最重要的还是活动内容的策划准备了，否则一切都是纸上谈兵。

讲到这，你可能会说，老老实实做技术不好吗，非要弄这些花里胡哨的东西。的确，推广这件事，并不是做开源必须的，老实做技术没有错，推广只是让你的开源项目更快传达给目标用户。做这些事的目的就一个，让更多的人使用你的开源项目，让更多的开发者参与贡献代码。

最后一点，如果你的公司或团队有经费，适当投放一点广告也是不错的。

国际化

是时候与世界接轨了。一般来说，国际化（Internationalization，简称 i18n）分 3 个部分，首先是你的开源项目支持多国语言，对于 UI 组件库来说，这个还是很好支持的，只需要提供一个多语言的配置文件就行，每种语言一个文件，然后由社区贡献更多的语言。以 Vue.js 为例，社区也提供了 vue-i18n 插件，那你的组件库还要兼容 vue-i18n，可能还要考虑兼容多个主流的版本。

另一部分是文档的国际化，除了中文，至少应该提供一个英文版本，毕竟英文算是通用的语言。如果文档内容不多，可以让社区来提供更多的翻译版本。维护多语言的文档是一件很辛苦的事，这意味着每一个版本更新都是中英双语的，并不是说文档翻译一遍就不管了。好在翻译文档是个一次性的技术活，前期多找一些英文好的热心用户一起翻译，后面只要确保每次更新都保持中英双语就好了。

做国际化，意味着要服务国际友人，那就不能强求他们用微信或 QQ。在开源界，比较通用的是 Gitter，只需要关联一次 GitHub 的 repo 就行。除此之外，官方可以在 Twitter 开通一个账户，来更新一些动态，与其它 Twitter 互动。Discord 也是技术圈比较热门的一个 App，以 Vue.js 来说，你可以加入一个名为 **Vue Land** 的服务器，在里面找到 **#ui-libraries** 的频道，就可以和全世界的开发者讨论组件库的话题了。

支持国际化，短期来看，是一件付出回报比很低的事，但从长远利益出发，对国际化的支持，有助于更多的国外开发者成为核心 maintainers，让全世界能够参与进来，才是开源的意义所在。

让更多的人参与

开源项目从来就不是一个人的事，一个健壮的开源项目，需要不断有人贡献代码。在项目有了一定知名度和使用人群后，自然会有不少 PR 进来，知名的开源项目 contributors 都有几百人，哪怕修改一行代码，只要被 merge，就算一个 contributors。最核心的维护者一般不会超过 5 人，而且除了作者本人，很多都是阶段性的，毕竟是开源，大多数人还是兼职做的，能贡献一点是一点，业务忙了就没顾不上了。

为开源项目贡献代码，主要以 PR 的形式进行，作为一个开源项目的 owner，即使 organization 的其他成员有直接 commit 的权限，也应该建议他们提交 PR，而不是直接 commit，owner 需要认真 review 每一个 PR 以确保代码质量。修复一个问题最怕的，是引起新的问题，或导致以前已修复的问题又复现，有时候，contributor 可能只为了 fix 某一个 issue，但它对整个项目是不了解的，而且对以前"发生的事情"都不了解，会导致一些他看不到的问题，这种情况作为 owner 就要认真审查了。

参与一个开源项目的方式有很多，除了最直接的 PR，还可以 review issues。项目活跃时，每天都会有不少 issues 进来，owner 可能没时间及时处理，但可以**打标签**（labels）。一个 issue 被标记了 label，说明已经审核过此 issue，常见的 label 有以下几种：

- **bug**：已确定为 bug；
- **feature request**：已确定为请求新功能；
- **invalid**：无效的 issue，一般可以直接关闭；
- **contribution welcome**：owner 可能暂时没有精力处理，期望社区来贡献代码；
- **provide example**：issue 需要提供复现示例；
- **discussion**：暂时无法断定，需要进一步讨论；
- **may be supported in the future**：先标记一下，也许未来会支持。

管理 issues 的另一个方式是用好**里程碑**（milestones）功能。milestones 可以按照版本号创建，把期望在这个版本解决的 issues 添加进去，发版前对当前 milestone 的所有 issues 集中查看，是否都处理完成了。

有一些**有价值**的 issues 可能会耗费不少精力处理，而且社区很多用户都希望能够解决，owner 当然也希望处理，只是没有时间。这种情况不妨**有偿悬赏**，推荐一个新起的国外社区 IssueHunt，用户可以为某一个 repo 的 issue 众筹，谁处理了，就可以得到全部赏金。

每一个版本发布后，记得创建一条 release，这样做一是有一个版本更新日志记录的地方，二是 watch 你项目的人都可以及时收到邮件通知提醒升级，三是 release 会打一个 tag，其它贡献者可以切换到此 tag。release 最好不要在发版前再创建，不然整理起来很费劲，建议每个 release 发布后，就新建下一个版本的 release 作为草稿（draft），处理一个 issue，就记录一条，避免遗漏。

版本号也是有讲究的，比如 3.2.1，这里的最后一位，代表只有 bug fixed，中间一位代表有 new features，第一位代表有 break changes。一般来说，除了第一位，剩下的版本都是兼容式的，就是说用户升级后不会影响当前项目，如果有 API 的变更，应该发布第一位版本号。

代码贡献越活跃，贡献者越多，开源项目也越健壮，作为 owner，应该及时联络有价值的贡献者，一个人的能力毕竟是有限的。当你与世界各地讲着不同语言的的人，一起完成一个开源项目，会觉得开源真是一件了不起的事情。

让 Robot 来做"坏人"

开源项目有一定的规模后，社区就会很活跃，每天都会有大量的 issues，这些 issues 越积越多，不及时处理掉，对 owner 来说就是精神压力。在项目初期，由于使用者不多，是鼓励提 issues 的，建议、新功能请求、bug 反馈、问题咨询等各种内容都可以提交，而且作者有足够的时间和精力来认真回答。到了一定规模后，可能什么 issues 都会出现，不乏一些带有恶意的、言语攻击的，如果直接关闭 issue，可能还会继续"纠缠"，说 owner 态度不好之类的，这些都是笔者亲身经历过的。

除了恶意的 issues，还有很多 issues 不符合格式要求，连代码格式化都没有，甚至连问题都说不清楚，也没有描述，就一个标题，这些无效的 issues 一个个回复都会占据大量的精力，直接 close 还会被说没处理怎么就关闭了，实属无奈。

这时你需要一个 GitHub 机器人来充当“坏人”的角色，也就是注册另一个 GitHub 账户，用它来处理一些不符合要求的 issues，这是一个很聪明的做法，关闭 issues 这些活都让 robot 来操作。比如 iView 的“坏人”就是 iview-bot，不过它是一个智能的 robot，不需要 owner 控制，会自动关闭不合格的 issues 并回复提问者。GitHub 提供了 API 来接收每一个 issues 并通过 API 来操作 issues，包括关闭、打标签、回复等，只要给 robot 设置足够的权限就行。比如 iView 的 issues 机器人代码是 https://github.com/iview/iview-bot。用户如果直接通过 GitHub 提交 issues，会被 robot 立即关闭，并回复：

> Hello, this issue has been closed because it does not conform to our issue requirements. Please use the Issue Helper to create an issue - thank you!

就是说，用户必须通过 Issue Helper 这个页面提交 issues 才可以，不是通过它提交的，会被检测出来立即关闭。在这个页面中，用户需要提供详细的描述才能通过表单验证。issues 只接受 bug 报告或是新功能请求 (feature requests)，对于使用咨询等其它问题，都不能提交，而是鼓励到 Stackoverflow 之类的社区讨论。如果是 Bug，还必须提供能够最小化复现问题的在线链接，以及详细的复现步骤。

robot 还有一个作用：翻译。它会把中文的 issues 自动翻译为英文并把翻译内容自动创建一条回复，同时标题也会修改为英文。开源项目到这个规模，使用者和贡献者不仅仅是中国人了，世界各地的开发者都有，使用英文会让所有人都看懂 issues。

虽然 robot 能自动过滤 80% 不合格的 issues，但仍有浑水摸鱼的用户跳过这些验证，这时可以给 robot 设置一些快捷回复，人为来 comment & close：

- Hello, this issue has been closed because similar problems exist or have been explained in the documentation, please check carefully. *已有相同 issues，或文档有说明*
- Hello, this issue has been closed because it is not required to submit or describe is not clear. *描述不清楚*
- Hello, this issue has been closed because it has nothing to do with the **bug report** or **feature request**. Maybe you can ask normal question through SegmentFault or stackoverflow. *不是 bug 反馈或 新功能请求，请到社区讨论*
- Hello, this issue has been closed because it is not a bug, but a usage problem, please consult other communities. *用法不对*
- Please provide online code. You can quickly create an example using the following online link：https://run.iviewui.com/. *没有提供在线示例*

其实呢，这个“坏人”也没那么坏，还是挺可爱的。

赞助与商业化

开源项目的发展离不开资金的支持，向社区寻求赞助并不是一件“羞耻”的事情，而是理所当然的。

最简单的赞助方式就是通过二维码打赏，不过这种方式在国内几乎没有什么用，中国的开发者大多比较“囊中羞涩”，而且由于打赏的匿名性（微信），时不时收到个 1 分钱，也就呵呵了。

这里推荐几种比较好的“募资”方式：patreon 和 opencollective 是开源项目最常用的，可以一次性支持，或周期性，以美元结算，可转至 PayPal。不过这两种都是美元，而且转到 owner 这里，扣除手续费可能少很多，不过对赞助者（往往是企业）来说，好处就是有发票。另一种方式是通过开源中国来赞助，开源中国的用户还是比较慷慨的。

另一种是投放广告，这里推荐 Carbon，不同于 Google Ads 的是，它的广告都是与互联网相关的，而且样式可以完全自定义，很美观，不会让用户产生反感，广告根据展示和点击转化付费。Carbon 的中国市场负责人中文很溜哦，作为中国开发者，不用担心谈不来。

不过呢，最值得推荐的还是接入**品牌广告**，但前提是你的文档要有一定的流量。开源项目的文档有着最大的特点：访问者几乎都是程序员，所以你要是挂个某多多的广告，几乎会被喷死。在线教育、云主机服务商都是不错的选择。一般不会有人主动联系 owner 投放的，除非像 Vue.js 这种级别的，但你可以尝试发一封友好的邮件来询问。不知道发给谁？告诉你个好办法，去其它社区（比如 v2ex）看看都有

哪些金主投放就知道了，既然已经投放，说明有投放广告的需求，都是潜在的目标“客户”。

再来说说商业化。

开源并不是意味着免费，根据开源协议的不同，有的开源软件在用于商业时，可能要购买授权，源码是开放的，但不一定可以免费使用。不过能够收取授权费，也说明你的软件确实无可替代。企业为了避免不必要的纠纷，肯定是愿意购买你的软件的。但是对于大多数 MIT 的开源项目，可以商业化吗？答案是肯定的。

首先要知道，能够付费的，都是企业，而非个人，个人也没有付费的必要。一种比较常见的模式就是软件免费，然后可以向企业提供额外的付费咨询服务或顾问。最懂开源项目的人，绝对是这个项目的 owner，如果企业是深度用户，还是很愿意支付一些费用来咨询问题的。我是做 to B 业务的，我们公司也是做 to B 的，公司高管大多也来自 Oracle（算是比较大的 to B 企业了），所以我对企业服务也有一定的理解，一款好的产品，绝对是技术加咨询服务。

商业化还是有很多方式的，具体要看开源项目的类型。以组件库为例，它本身是免费的，也可以无限制免费使用，但可能提供付费的高级组件或模板系统，以及其它生态产品，比如基于组件库的 IM 系统。

当然了，并不是所有的开源项目都要商业化，大部分还是完全免费的，商业化也有利弊，如果没有一定的实力，很有可能搞砸哦！

以上，就是我从事开源工作两年多的一些浅薄经验，希望能给聪明的你带来帮助。

结语

每个开发者，都应该尝试维护一个开源项目。

每个开发者，都应该抱着一颗敬畏之心使用他人的开源项目，而不是“用你的是看得起你”。

每个开发者，都应该适当地赞助一个帮助过你的开源项目。

写在最后

亲爱的读者，到这里本小册就要结束了，你是否从中学习到了属于你的知识呢？我们来回顾一下小册的内容吧。

Vue.js 在开发独立组件时，由于它的特殊性，无法使用 Vuex、Bus 这样的第三方插件来做组件通信，因此小册提到了 3 种组件间的通信方法，都是支持跨多级的：

1. provide / inject：由父组件通过 `provide` 向下提供一个数据，子组件在需要时，通过 `inject` 注入这个依赖，就可以直接访问父级的数据了。
2. dispatch / broadcast：组件向上派发或向下广播一个自定义事件，组件通过 `$on` 来监听。
3. findComponents 系列：共包含 5 个方法，通过组件的 `name` 选项，遍历找到对应的实例，这是组件通信的终极方案，适用于所有场景。

本册总共讲解了 7 个组件的实例：

1. 具有数据校验功能的 Form 组件，它用到了第 1 种组件通信；
2. 组合多选框 Checkbox 组件，它用到了第 2 种和第 3 种组件通信；
3. Display 组件，它利用 Vue.js 的构造器 extend 和手动挂载 $mount API；
4. 全局通知 $Alert$组件，也是利用了 mount API，与传统组件不同的是，它基于 Vue.js 组件开发，但却是以 JavaScript 的形式调用的，并且组件会在 body 节点下渲染；
5. 表格组件 Table，典型的数据驱动型组件，使用了函数式组件（Functional Render）来自定义列模板；
6. 表格组件 Table，与上例不同的是，它的自定义列模板使用了 `slot-scope`；
7. 树形控件 Tree，典型的数据驱动型组件，也是典型的递归组件，其中利用 computed 做父子节点联动是精髓。

最后的拓展部分，对 Vue.js 组件的常见 API 做了详细介绍，以及常见的 Vue.js 面试题分析和对开源的一些见解。

Vue.js 组件开发，归根到底拼的是 JavaScript 的功底，Vue.js 在其中只是决定了开发模式，所以，打好 JavaScript 基础才是最重要的。

最后，祝愿亲爱的读者能在编程的道路上越走越远。

另外，如果您有关于 Vue.js 或其它前端相关的问题，可以通过 iView 社区 付费向笔者提问，希望能对一部分人有所帮助。