--

[组件 1](#_Toc6890)

[基础概念 1](#_Toc10657)

[data 1](#_Toc6264)

[组件父子关系 1](#_Toc3955)

[prop 1](#_Toc25262)

[事件 2](#_Toc30821)

[杂项 3](#_Toc8505)

[单个组件 4](#_Toc24648)

[组件之间的传递 4](#_Toc20521)

[关于组件（父子间传值.sync v-model 组件的引用方式） 5](#_Toc11334)

[vuex状态 6](#_Toc15327)

[概况 6](#_Toc27574)

[关于组件的组合写法 8](#_Toc8447)

|  |  |
| --- | --- |
|  | <https://github.com/fengcms/vuedemo>  <http://blog.csdn.net/fungleo/article/details/54602753>  <http://blog.csdn.net/FungLeo/article/category/6258502>  vue实战系列文章 基础 vue-cli .vue router 代理 打包 |
|  | http://blog.csdn.net/column/details/15666.html  前端框架 Vue 初体验（15篇） 国际化 校验 css js统一 echarts |

组件

|  |  |
| --- | --- |
| 基础概念 | 组件（Component）是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素，封装可重用的代码。在较高层面上，组件是自定义元素， Vue.js 的编译器为它添加特殊功能。在有些情况下，组件也可以是原生 HTML 元素的形式，以 is 特性扩展。 |
|  | 之前说过，我们可以通过以下方式创建一个 Vue 实例：   |  | | --- | | new Vue({  el: '#some-element',  // 选项  }) |   要注册一个全局组件，你可以使用 Vue.component(tagName, options)。 例如：   |  | | --- | | Vue.component('my-component', {  // 选项  }) |   对于自定义标签名，Vue.js 不强制要求遵循 [W3C规则](https://www.w3.org/TR/custom-elements/" \l "concepts" \t "https://cn.vuejs.org/v2/guide/_blank) （小写，并且包含一个短杠），尽管遵循这个规则比较好。 |
|  | 全局注册  组件在注册之后，便可以在父实例的模块中以自定义元素 <my-component></my-component>的形式使用。要确保在初始化根实例 之前 注册了组件：   |  | | --- | | <div id="example">  <my-component></my-component>  </div> |  |  | | --- | | // 注册  Vue.component('my-component', {  template: '<div>A custom component!</div>'  })  // 创建根实例  new Vue({  el: '#example'  }) |   渲染为：   |  | | --- | | <div id="example">  <div>A custom component!</div>  </div> | |
|  | 局部注册  不必在全局注册每个组件。通过使用组件实例选项注册，可以使组件仅在另一个实例/组件的作用域中可用：   |  | | --- | | var Child = {  template: '<div>A custom component!</div>'  }  new Vue({  // ...  components: {  // <my-component> 将只在父模板可用  'my-component': Child  }  }) |   这种封装也适用于其它可注册的 Vue 功能，如指令。 |
|  | [DOM 模版解析说明](https://cn.vuejs.org/v2/guide/components.html" \l "DOM-模版解析说明)  当使用 DOM 作为模版时（例如，将 el 选项挂载到一个已存在的元素上）, 你会受到 HTML 的一些限制，因为 Vue 只有在浏览器解析和标准化 HTML 后才能获取模版内容。尤其像这些元素 <ul> ，<ol>，<table> ，<select> 限制了能被它包裹的元素， 而一些像<option> 这样的元素只能出现在某些其它元素内部。  在自定义组件中使用这些受限制的元素时会导致一些问题，例如：   |  | | --- | | <table>  <my-row>...</my-row>  </table> |   自定义组件 <my-row> 被认为是无效的内容，因此在渲染的时候会导致错误。变通的方案是使用特殊的 is 属性：   |  | | --- | | <table>  <tr is="my-row"></tr>  </table> |   应当注意，如果您使用来自以下来源之一的字符串模板，这些限制将不适用：  <script type="text/x-template">  JavaScript内联模版字符串  .vue 组件  因此，有必要的话请使用字符串模版。 |
| data | > [data 必须是函数](https://cn.vuejs.org/v2/guide/components.html" \l "data-必须是函数)  通过Vue构造器传入的各种选项大多数都可以在组件里用。 data 是一个例外，它必须是函数。 实际上，如果你这么做：   |  | | --- | | Vue.component('my-component', {  template: '<span>{{ message }}</span>',  data: {  message: 'hello'  }  }) |   那么 Vue 会停止，并在控制台发出警告，告诉你在组件中 data 必须是一个函数。  理解这种规则的存在意义很有帮助，让我们假设用如下方式来绕开Vue的警告：   |  | | --- | | <div id="example-2">  <simple-counter></simple-counter>  <simple-counter></simple-counter>  <simple-counter></simple-counter>  </div> |  |  | | --- | | var data = { counter: 0 }  Vue.component('simple-counter', {  template: '<button v-on:click="counter += 1">{{ counter }}</button>',   // 技术上 data 的确是一个函数了，因此 Vue 不会警告，   // 但是我们返回给每个组件的实例的却引用了同一个data对象   data: function () {  return data  }  })  new Vue({  el: '#example-2'  }) |   由于这三个组件共享了同一个 data ， 因此增加一个 counter 会影响所有组件！这不对。我们可以通过为每个组件返回全新的 data 对象来解决这个问题：   |  | | --- | | data: function () {  return {  counter: 0  }  } |   现在每个 counter 都有它自己内部的状态了； |

组件父子关系

|  |  |
| --- | --- |
|  | [构成组件](https://cn.vuejs.org/v2/guide/components.html" \l "构成组件)  组件意味着协同工作，通常父子组件会是这样的关系：组件 A 在它的模版中使用了组件 B 。它们之间必然需要相互通信：父组件要给子组件传递数据，子组件需要将它内部发生的事情告知给父组件。然而，在一个良好定义的接口中尽可能将父子组件解耦是很重要的。这保证了每个组件可以在相对隔离的环境中书写和理解，也大幅提高了组件的可维护性和可重用性。  **在 Vue.js 中，父子组件的关系可以总结为 props down, events up 。父组件通过 props 向下传递数据给子组件，子组件通过 events 给父组件发送消息。看看它们是怎么工作的。** |
| prop | [使用 Prop 传递数据](https://cn.vuejs.org/v2/guide/components.html" \l "使用-Prop-传递数据)  组件实例的作用域是孤立的。这意味着不能(也不应该)在子组件的模板内直接引用父组件的数据。要让子组件使用父组件的数据，我们需要通过子组件的props选项。  子组件要显式地用 **[props 选项](https://cn.vuejs.org/v2/api/" \l "props)**声明它期待获得的数据：   |  | | --- | | Vue.component('child', {  // 声明 props  props: ['message'],  // 就像 data 一样，prop 可以用在模板内  // 同样也可以在 vm 实例中像 “this.message” 这样使用  template: '<span>{{ message }}</span>'  }) |   然后我们可以这样向它传入一个普通字符串：   |  | | --- | | <child message="hello!"></child> |   结果：  hello!  [camelCase vs. kebab-case](https://cn.vuejs.org/v2/guide/components.html" \l "camelCase-vs-kebab-case)  HTML 特性是不区分大小写的。所以，当使用的不是字符串模版，camelCased (驼峰式) 命名的 prop 需要转换为相对应的 kebab-case (短横线隔开式) 命名：   |  | | --- | | Vue.component('child', {  // camelCase in JavaScript  props: ['myMessage'],  template: '<span>{{ myMessage }}</span>'  }) |  |  | | --- | | <!-- kebab-case in HTML -->  <child my-message="hello!"></child> |   如果你使用字符串模版，则没有这些限制。  [动态 Prop](https://cn.vuejs.org/v2/guide/components.html" \l "动态-Prop)  在模板中，要动态地绑定父组件的数据到子模板的props，与绑定到任何普通的HTML特性相类似，就是用 v-bind。每当父组件的数据变化时，该变化也会传导给子组件：   |  | | --- | | <div>  <input v-model="parentMsg">  <br>  <child v-bind:my-message="parentMsg"></child>  </div> |   使用 v-bind 的缩写语法通常更简单：   |  | | --- | | <child :my-message="parentMsg"></child> |   结果：  [字面量语法 vs 动态语法](https://cn.vuejs.org/v2/guide/components.html" \l "字面量语法-vs-动态语法)  初学者常犯的一个错误是使用字面量语法传递数值：   |  | | --- | | <!-- 传递了一个字符串 "1" -->  <comp some-prop="1"></comp> |   因为它是一个字面 prop ，它的值是字符串 "1" 而不是number。如果想传递一个实际的number，需要使用 v-bind ，从而让它的值被当作 JavaScript 表达式计算：   |  | | --- | | <!-- 传递实际的 number -->  <comp v-bind:some-prop="1"></comp> |   [单向数据流](https://cn.vuejs.org/v2/guide/components.html" \l "单向数据流)  prop 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来。这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。  另外，每次父组件更新时，子组件的所有 prop 都会更新为最新值。这意味着你不应该在子组件内部改变 prop 。如果你这么做了，Vue 会在控制台给出警告。  为什么我们会有修改prop中数据的冲动呢？通常是这两种原因：  prop 作为初始值传入后，子组件想把它当作局部数据来用；  prop 作为初始值传入，由子组件处理成其它数据输出。  对这两种原因，正确的应对方式是：  定义一个局部变量，并用 prop 的值初始化它：   |  | | --- | | props: ['initialCounter'],  data: function () {  return { counter: this.initialCounter }  } |   定义一个计算属性，处理 prop 的值并返回。   |  | | --- | | props: ['size'],  computed: {  normalizedSize: function () {  return this.size.trim().toLowerCase()  }  } |   注意在 JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 prop 是一个对象或数组，在子组件内部改变它会影响父组件的状态。  [Prop 验证](https://cn.vuejs.org/v2/guide/components.html" \l "Prop-验证)  我们可以为组件的 props 指定验证规格。如果传入的数据不符合规格，Vue 会发出警告。当组件给其他人使用时，这很有用。  要指定验证规格，需要用对象的形式，而不能用字符串数组：   |  | | --- | | Vue.component('example', {  props: {  // 基础类型检测 （`null` 意思是任何类型都可以）  propA: Number,  // 多种类型  propB: [String, Number],  // 必传且是字符串  propC: {  type: String,  required: true  },  // 数字，有默认值  propD: {  type: Number,  default: 100  },  // 数组／对象的默认值应当由一个工厂函数返回  propE: {  type: Object,  default: function () {  return { message: 'hello' }  }  },  // 自定义验证函数  propF: {  validator: function (value) {  return value > 10  }  }  }  }) |   type 可以是下面原生构造器：  String  Number  Boolean  Function  Object  Array  Symbol  type 也可以是一个自定义构造器函数，使用 instanceof 检测。  当 prop 验证失败，Vue会在抛出警告 (如果使用的是开发版本)。 |
|  |  |
| 事件 | 我们知道，父组件是使用 props 传递数据给子组件，但如果子组件要把数据传递回去，应该怎样做？那就是自定义事件！  [使用 v-on 绑定自定义事件](https://cn.vuejs.org/v2/guide/components.html" \l "使用-v-on-绑定自定义事件)，每个 Vue 实例都实现了[事件接口(Events interface)](https://cn.vuejs.org/v2/api/" \l "实例方法-事件)，即：  使用 $on(eventName) 监听事件  使用 $emit(eventName) 触发事件  Vue的事件系统分离自浏览器的[EventTarget API](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget" \t "https://cn.vuejs.org/v2/guide/_blank)。尽管它们的运行类似，但是$on和 $emit 不是addEventListener 和 dispatchEvent 的别名。  另外，父组件可以在使用子组件的地方直接用 v-on 来监听子组件触发的事件。  不能用$on侦听子组件抛出的事件，而必须在模板里直接用v-on绑定，就像以下的例子：  下面是一个例子：   |  | | --- | | <div id="counter-event-example">  <p>{{ total }}</p>  <button-counter v-on:increment="incrementTotal"></button-counter>  <button-counter v-on:increment="incrementTotal"></button-counter>  </div> |  |  | | --- | | Vue.component('button-counter', {  template: '<button v-on:click="increment">{{ counter }}</button>',  data: function () {  return {  counter: 0  }  },  methods: {  increment: function () {  this.counter += 1  this.$emit('increment')  }  },  })  new Vue({  el: '#counter-event-example',  data: {  total: 0  },  methods: {  incrementTotal: function () {  this.total += 1  }  }  }) |   20  9 11  在本例中，子组件已经和它外部完全解耦了。它所做的只是报告自己的内部事件，至于父组件是否关心则与它无关。留意到这一点很重要。 |
| 给组件绑定原生事件  有时候，你可能想在某个组件的根元素上监听一个原生事件。可以使用 .native 修饰v-on 。例如：   |  | | --- | | <my-component v-on:click.native="doTheThing"></my-component> | |
| .sync 修饰符[2.3.0+]  在一些情况下，我们可能会需要对一个 prop 进行『双向绑定』。事实上，这正是 Vue 1.x 中的 .sync修饰符所提供的功能。当一个子组件改变了一个 prop 的值时，这个变化也会同步到父组件中所绑定的值。这很方便，但也会导致问题，因为它破坏了『单向数据流』的假设。由于子组件改变 prop 的代码和普通的状态改动代码毫无区别，当光看子组件的代码时，你完全不知道它何时悄悄地改变了父组件的状态。这在 debug 复杂结构的应用时会带来很高的维护成本。  上面所说的正是我们在 2.0 中移除 .sync 的理由。但是在 2.0 发布之后的实际应用中，我们发现 .sync 还是有其适用之处，比如在开发可复用的组件库时。我们需要做的只是让子组件改变父组件状态的代码更容易被区分。  在 2.3 我们重新引入了 .sync 修饰符，但是这次它只是作为一个编译时的语法糖存在。它会被扩展为一个自动更新父组件属性的 v-on 侦听器。  如下代码  <comp :foo.sync="bar"></comp>  会被扩展为：  <comp :foo="bar" @update:foo="val => bar = val"></comp>  当子组件需要更新 foo 的值时，它需要显式地触发一个更新事件：  this.$emit('update:foo', newValue) |
| v-model |
| 非父子组件通信  有时候两个组件也需要通信 (非父子关系)。在简单的场景下，可以使用一个空的 Vue 实例作为中央事件总线：  var bus = new Vue()  // 触发组件 A 中的事件  bus.$emit('id-selected', 1)  // 在组件 B 创建的钩子中监听事件  bus.$on('id-selected', function (id) {  // ...  })  在复杂的情况下，我们应该考虑使用专门的状态管理模式. |
|  |  |

杂项

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **[编写可复用组件](https://cn.vuejs.org/v2/guide/components.html" \l "编写可复用组件)**  在编写组件时，记住是否要复用组件有好处。一次性组件跟其它组件紧密耦合没关系，但是可复用组件应当定义一个清晰的公开接口。  Vue 组件的 API 来自三部分 - props, events 和 slots ：  Props 允许外部环境传递数据给组件  Events 允许从外部环境在组件内触发副作用  Slots 允许外部环境将额外的内容组合在组件中。  简写  使用 v-bind 和 v-on 的简写语法，模板的缩进清楚且简洁：   |  | | --- | | <my-component  :foo="baz"  :bar="qux"  @event-a="doThis"  @event-b="doThat"  >  <img slot="icon" src="...">  <p slot="main-text">Hello!</p>  </my-component> |   [子组件索引](https://cn.vuejs.org/v2/guide/components.html" \l "子组件索引)  尽管有 props 和 events，但是有时仍然需要在 JavaScript 中直接访问子组件。为此可以使用ref 为子组件指定一个索引 ID。例如：   |  | | --- | | <div id="parent">  <user-profile ref="profile"></user-profile>  </div> |  |  | | --- | | var parent = new Vue({ el: '#parent' })  // 访问子组件  var child = parent.$refs.profile |   当 ref 和 v-for 一起使用时，ref 是一个数组，包含相应的子组件。  $refs 只在组件渲染完成后才填充，并且它是非响应式的。它仅仅作为一个直接访问子组件的应急方案——应当避免在模版或计算属性中使用 $refs。  [异步组件](https://cn.vuejs.org/v2/guide/components.html" \l "异步组件)  在大型应用中，我们可能需要将应用拆分为多个小模块，按需从服务器下载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：   |  | | --- | | Vue.component('async-example', function (resolve, reject) {  setTimeout(function () {  // Pass the component definition to the resolve callback  resolve({  template: '<div>I am async!</div>'  })  }, 1000)  }) |   工厂函数接收一个 resolve 回调，在收到从服务器下载的组件定义时调用。也可以调用reject(reason) 指示加载失败。这里 setTimeout 只是为了演示。怎么获取组件完全由你决定。推荐配合使用 ：**[Webpack 的代码分割功能](https://webpack.js.org/guides/code-splitting/" \t "https://cn.vuejs.org/v2/guide/_blank)**：   |  | | --- | | Vue.component('async-webpack-example', function (resolve) {  // 这个特殊的 require 语法告诉 webpack  // 自动将编译后的代码分割成不同的块，  // 这些块将通过 Ajax 请求自动下载。  require(['./my-async-component'], resolve)  }) |   你可以使用 Webpack 2 + ES2015 的语法返回一个 Promise resolve 函数：   |  | | --- | | Vue.component(  'async-webpack-example',  () => import('./my-async-component')  ) |   当使用**[局部注册](https://cn.vuejs.org/v2/guide/components.html" \l "局部注册)**时，你也可以直接提供一个返回 Promise 的函数：   |  | | --- | | new Vue({  // ...  components: {  'my-component': () => import('./my-async-component')  }  }) |   如果你是 Browserify 用户,可能就无法使用异步组件了,它的作者已经**[表明](https://github.com/substack/node-browserify/issues/58" \l "issuecomment-21978224" \t "https://cn.vuejs.org/v2/guide/_blank)**Browserify 是不支持异步加载的。Browserify 社区发现 **[一些解决方法](https://github.com/vuejs/vuejs.org/issues/620" \t "https://cn.vuejs.org/v2/guide/_blank)**，可能有助于已存在的复杂应用。对于其他场景，我们推荐简单实用 Webpack 构建，一流的异步支持  [高级异步组件](https://cn.vuejs.org/v2/guide/components.html" \l "高级异步组件)  2.3.0 新增  自 2.3 起，异步组件的工厂函数也可以返回一个如下的对象：   |  | | --- | | const AsyncComp = () => ({  // 需要加载的组件. 应当是一个 Promise  component: import('./MyComp.vue'),  // loading 时应当渲染的组件  loading: LoadingComp,  // 出错时渲染的组件  error: ErrorComp,  // 渲染 loading 组件前的等待时间。默认：200ms.  delay: 200,  // 最长等待时间。超出此时间则渲染 error 组件。默认：Infinity  timeout: 3000  }) |   注意，当一个异步组件被作为 vue-router 的路由组件使用时，这些高级选项都是无效的，因为在路由切换前就会提前加载所需要的异步组件。另外，如果你要在路由组件中上述写法，需要使用 vue-router 2.4.0+。  [组件命名约定](https://cn.vuejs.org/v2/guide/components.html" \l "组件命名约定)  当注册组件 (或者 props) 时，可以使用 kebab-case，camelCase，或 PascalCase。   |  | | --- | | // 在组件定义中  components: {  // 使用 kebab-case 形式注册  'kebab-cased-component': { /\* ... \*/ },  // register using camelCase  'camelCasedComponent': { /\* ... \*/ },  // register using PascalCase  'PascalCasedComponent': { /\* ... \*/ }  } |   在 HTML 模版中，请使用 kebab-case 形式：   |  | | --- | | <!-- 在HTML模版中始终使用 kebab-case -->  <kebab-cased-component></kebab-cased-component>  <camel-cased-component></camel-cased-component>  <pascal-cased-component></pascal-cased-component> |   当使用字符串模式时，可以不受 HTML 的 case-insensitive 限制。这意味实际上在模版中，你可以使用下面的方式来引用你的组件：  kebab-case  camelCase 或 kebab-case 如果组件已经被定义为 camelCase  kebab-case，camelCase 或 PascalCase 如果组件已经被定义为 PascalCase   |  | | --- | | components: {  'kebab-cased-component': { /\* ... \*/ },  camelCasedComponent: { /\* ... \*/ },  PascalCasedComponent: { /\* ... \*/ }  } |  |  | | --- | | <kebab-cased-component></kebab-cased-component>  <camel-cased-component></camel-cased-component>  <camelCasedComponent></camelCasedComponent>  <pascal-cased-component></pascal-cased-component>  <pascalCasedComponent></pascalCasedComponent>  <PascalCasedComponent></PascalCasedComponent> |   这意味着 PascalCase 是最通用的 声明约定 而 kebab-case 是最通用的 使用约定。  如果组件未经 slot 元素传递内容，你甚至可以在组件名后使用 / 使其自闭合：   |  | | --- | | <my-component/> |   当然，这只在字符串模版中有效。因为自闭的自定义元素是无效的 HTML，浏览器原生的解析器也无法识别它。  [递归组件](https://cn.vuejs.org/v2/guide/components.html" \l "递归组件)  组件在它的模板内可以递归地调用自己，不过，只有当它有 name 选项时才可以：   |  | | --- | | name: 'unique-name-of-my-component' |   当你利用Vue.component全局注册了一个组件, 全局的ID作为组件的 name 选项，被自动设置.   |  | | --- | | Vue.component('unique-name-of-my-component', {  // ...  }) |   如果你不谨慎, 递归组件可能导致死循环:   |  | | --- | | name: 'stack-overflow',  template: '<div><stack-overflow></stack-overflow></div>' |   上面组件会导致一个错误“max stack size exceeded”，所以要确保递归调用有终止条件 (比如递归调用时使用 v-if 并让他最终返回 false )。  [组件间的循环引用](https://cn.vuejs.org/v2/guide/components.html" \l "组件间的循环引用)  假设你正在构建一个文件目录树，像在Finder或文件资源管理器中。你可能有一个tree-folder组件:   |  | | --- | | <p>  <span>{{ folder.name }}</span>  <tree-folder-contents :children="folder.children"/>  </p> |   然后 一个tree-folder-contents组件：   |  | | --- | | <ul>  <li v-for="child in children">  <tree-folder v-if="child.children" :folder="child"/>  <span v-else>{{ child.name }}</span>  </li>  </ul> |   当你仔细看时，会发现在渲染树上这两个组件同时为对方的父节点和子节点–这点是矛盾的。当使用Vue.component将这两个组件注册为全局组件的时候，框架会自动为你解决这个矛盾，如果你是这样做的，就不用继续往下看了。 然而，如果你使用诸如Webpack或者Browserify之类的模块化管理工具来requiring/importing组件的话，就会报错了：   |  | | --- | | Failed to mount component: template or render function not defined. |   为了解释为什么会报错，简单的将上面两个组件称为 A 和 B，模块系统看到它需要 A，但是首先 A 需要 B，但是 B 需要 A，而 A 需要 B，陷入了一个无限循环，因此不知道到底应该先解决哪个。要解决这个问题，我们需要在其中一个组件中 (比如 A) 告诉模块化管理系统，“A 虽然需要 B，但是不需要优先导入 B”  在我们的例子中，我们选择在tree-folder 组件中来告诉模块化管理系统循环引用的组件间的处理优先级，我们知道引起矛盾的子组件是tree-folder-contents，所以我们在beforeCreate 生命周期钩子中去注册它：   |  | | --- | | beforeCreate: function () {  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue')  } |   问题解决了。  [内联模版](https://cn.vuejs.org/v2/guide/components.html" \l "内联模版)  如果子组件有 inline-template 特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板更灵活。   |  | | --- | | <my-component inline-template>  <div>  <p>These are compiled as the component's own template.</p>  <p>Not parent's transclusion content.</p>  </div>  </my-component> |   但是 inline-template 让模板的作用域难以理解。最佳实践是使用 template 选项在组件内定义模板或者在 .vue 文件中使用 template 元素。  [X-Templates](https://cn.vuejs.org/v2/guide/components.html" \l "X-Templates)  另一种定义模版的方式是在 JavaScript 标签里使用 text/x-template 类型，并且指定一个 id。例如：   |  | | --- | | <script type="text/x-template" id="hello-world-template">  <p>Hello hello hello</p>  </script> |  |  | | --- | | Vue.component('hello-world', {  template: '#hello-world-template'  }) |   这在有很多模版或者小的应用中有用，否则应该避免使用，因为它将模版和组件的其他定义隔离了。  [对低开销的静态组件使用 v-once](https://cn.vuejs.org/v2/guide/components.html" \l "对低开销的静态组件使用-v-once)  尽管在 Vue 中渲染 HTML 很快，不过当组件中包含大量静态内容时，可以考虑使用 v-once将渲染结果缓存起来，就像这样：   |  | | --- | | Vue.component('terms-of-service', {  template: '\  <div v-once>\  <h1>Terms of Service</h1>\  ... a lot of static content ...\  </div>\  '  }) | |

单个组件

|  |  |
| --- | --- |
| 1 | 三分钟教你写个Vue组件  <http://www.jianshu.com/p/e731d9ad4ed1> |
| 定义 | [btn.vue] |
| 使用 | <ict-button :disabled="true" title="a button with click"></ict-button>  <script>  import IctButton from './btn.vue'  export default {  data() {  return {  message: 'Hello Vue.js!'  }  },  components: { IctButton }  }  </script> |

组件之间的传递

|  |  |
| --- | --- |
| 2 | <http://www.cnblogs.com/SamWeb/p/6391373.html>  [vuejs 单文件组件.vue 文件](http://www.cnblogs.com/SamWeb/p/6391373.html)【包含父子组件之间的传值】 |
|  | 在.vue 文件中，template 中都是html 代码，它定义了在页面中显示的内容，由于里面还有变量，也可以说定义了一个模版；script中都是js 代码，它定义这个组件中所需要的数据和及其操作，style 里面是css 样式，定义这个组件的样式，scoped 表明这里写的css 样式只适用于该组件，可以限定样式的作用域。  script 标签中 export defalut 后面的对象的理解：在不使用.vue 单文件时，我们是通过 Vue 构造函数创建一个 Vue 根实例来启动vuejs 项目，Vue 构造函数接受一个对象，这个对象有一些配置属性 el, data, component, template 等，从而对整个应用提供支持。  new Vue({  el: '#app',  data: {  msg: "hello Vue"  }  })  在.vue文件中，export default 后面的对象 就相当于 new Vue() 构造函数中的接受的对象，它们都是定义组件所需要的数据（data）, 以及操作数 据的方法等， 更为全面的一个 export default 对象，有methods, data, computed, 这时可以看到, 这个对象和new Vue() 构造函数中接受的对象是一模一样的。但要注意data 的书写方式不同。在 .vue 组件, data 必须是一个函数，它return（返回一个对象），这个返回的对象的数据，供组件实现。 |
|  | 父子组件之间的通信  每一个.vue 文件就是一个 组件，组件和组件相互组合，就成了一个应用，这就涉及到的组件和组件之间的通信，最常用的就是父子之间的通信。在vue 中， 在一个组件中通过 import 引入另一个组件，这个组件就是父组件，被引入的组件就是子组件。  在我们这个vue-cli 项目中，src 文件夹下有一个App.vue 文件，它的script标签中，import Hello from './components/Hello'，那么 App.vue 就是父组件，components 文件夹下的Hello.vue 就是子组件。父组件通过props 向子组件传递数据，子组件通过自定义事件向父组件传递数据。  父组件向子组件传值, 它主要是通过元素的属性进行的. 在App.vue 的template中,有一个 <hello></hello>, 这就是我们引入的子组件.  给其添加属性如 mes-father="message from father";  父组件将数据传递进去,子组件需要接收才能使用. 怎样接收呢?  在Hello.vue 中, export default 后面的对象中,添加一个字段props, 它是一个数组, 专门用来接收父组件传递过来的数据. props: ["mesFather"], 这里定义了mesFather 字符串, 和父组件中定义的元素的属性一一对应. 但是我们在父组件,就是在 <hello /> 元素中定义的属性是mes-father， 没有一一对应啊?  这主要是因为，在html 元素中大小写是不敏感的。 如果我们写成<hello mesFather="message from father"></hello>， 里面的mesFather  就会转化成mesfather, 相当于我们向子组件传递了一个mesfather数据， 如果在js 文件中，我们定义 props: ["mesFather"]，我们是接受不到数据的，因为js 是区分大小写的， 只能写成props: ["mesfather"].  但是在js 文件中，像这种两个单词拼成的数据，我们习惯用驼峰命名法，所以vue 做了一个转化，如果在组件中属性是 - 表示，它 自动会转化成驼峰式。  传进来的数据是mes-father, 转化成mesFather, 我们在js 里面写mesFather, 一一对应，子组件可以接受到组件。 props 属性是和data， methods 属性并列的，属同一级别。 props 属性里面定义的变量，在 子组件中的template 中可以直接使用。  App.vue 的template 更改如下：  <template>  <div id="app">  <img src="./assets/logo.png">  <hello mes-father="message from father"></hello>  </div>  </template>  Hello.vue组件，这里还是把项目中自带的Hello.vue 清空，自己写，变成如下内容  <template>  <div class="hello">  <p>{{mesFather}}</p>  </div>  </template>  <script>  export default {  props:['mesFather']  }  </script>  这时，在页面中看到 message from father 字样，父元素向子元素传递数据成功。  子组件向父组件传递数据，需要用到自定义事件。 例如，我们在Hello.vue ，写入一个input, 接收用户输入，我们想把用户输入的数据传给父组件。这时，input 需要先绑定一个keypress 事件，获取用户的输入，同时还要发射自定义事件，如valueUp, 父组件只要监听这个自定义事件，就可以知道子组件要向他传递数据了。子组件在发射自定义事件时，还可以携带参数，父组件在监听该事件时，还可以接受参数，参数，就是要传递的数据。  在 Hello.vue template中，添加一个input输入框，给它一个v-model 获取用户的输入，再添加keypress的事件，用于发射事件，传输数据。script 中添加data，定义变量来获取用户的输入，添加methods 来处理keypress事件的处理函数enter, 整个Hello.vue 文件如下  <template>  <div class="hello">  <input type="txet" placeholder="请输入文字" v-model="msg" **@keypress.enter="enter"**>  <p>{{upper}}</p>  <!--  mesFather值来自父组件(.vue);  使用方式：  在父组件的子组件使用中添加相应的属性值：<input-com mes-father="message from father"></input-com>  在子组件中添加props属性  -->  <p>{{mesFather}}</p>  </div>  </template>  <script>  export default {  props: ['mesFather'],  data() {  return {  msg: 'hello'  }  },  methods: {  enter() {  **this.$emit("sonvaluechange", this.msg);**  // 子组件发射自定义事件 sonvaluechange, 并携带要传递给父组件的值；  // 如果要传递给父组件很多值，这些值要作为参数依次列出 如 this.$emit('sonvaluechange', this.inputValue, this.mesFather);  // 事件的名称需使用小写（？）  }  },  computed: {  upper() {  return this.msg.toUpperCase();  }  }  }  </script>  <style scoped>  input {  width: 200px;  height: 20px;  }  p {  color: red;  }  </style> |
|  | 在App.vue 中， template中hello 组件绑定一个自定义事件，@valueUp =“receive”, 用于监听子组件发射的事件，再写一个 p 元素，用于展示子组件传递过来的数据，<p>子组件传递过来的数据 {{ childMes }}</p>  相应地，在scrpit中，data 中，定义一个变量childMes, 并在 methods 中，定义一个事件处理函数reciever。整个App.vue修改如下：  <template>  <div>  <p>{{ message }}</p>  <input v-model="message">  <ict-button :disabled="true" title="a button with click"></ict-button>  <input-com mes-father="message from father" **@sonvaluechange="receive"**></input-com>  <p>{{sonValue}}</p>  </div>  </template>  <script>  import IctButton from './btn.vue'  import InputCom from './inputCom.vue'  export default {  data() {  return {  message: 'Hello Vue.js!',  sonValue: "default value"  }  },  methods: {  **receive(mes) { // recieve 事件需要设置参数，这些参数就是子组件传递过来的数据，因此，参数的个数，也要和子元素传递过来的一致。**  **this.sonValue = mes;**  **}**  },  components: { IctButton, InputCom }  }  </script> |
|  | 这时在input中输入内容，然后按enter键，就以看到子组件传递过来的数据，子组件向父组件传递数据成功。  当在input输入框中输入数据，并按enter键时，它会触发keypress.enter事件，从而调用事件处理函数enter， 在enter 中， 我们发射了一个事件valueUp， 并携带了一个参数，由于在<hello @valueUp=”recieve”></hello> 组件中， 我们绑定valueUp 事件，所以父组件在时刻监听valueUp 事件， 当子组件发射value 事件时，父组件立刻捕获到，并立即调用它的回调函数receive, 在receive 中，我们获取到子组件传递过来的数据，并赋值了data 中的变量childMes, 由于data 数据发生变化，从而触发dom更新，页面中就显示子组件传递过来的内容。  其实在子组件中， props 最好的写法是props 验证，我们在子组件Hello.vue中写 props:['mesFather'], 只是表达出，它接受一个参数mesFather, 如果写成props 验证，不仅能表达出它需要什么参数，还能表达参数类型，并且如有错误，vue 会做出警告。现在把props 改成props 验证的写法, Hello.vue 中的js中的props修改如下：  props： {  'mesFather': {  type: String,  default: 'from father',  required:true  }  }  如果是组件与组件之间的通信非常复杂，不光是父子组件，还有兄弟组件，那就需要用到状态管理，vuex |

关于组件（父子间传值.sync v-model 组件的引用方式）

|  |  |
| --- | --- |
|  | <https://github.com/linzx1993/linzx1993.github.io/issues>  <https://github.com/linzx1993/linzx1993.github.io/issues/6>  手写Vue组件踩坑与心得 #6  记得以前看过一句话，说市面上任何的UI库都无法满足一个产品的所有需求。  事实上的确如此，产品需求总是千奇百怪。正如我公司现在的产品，引用的是elemen-ui的库，但是无法级联多选，下拉多选的展现形式不对，穿梭框无法上下移动等各种需求逼迫我们只能自己去写组件实现了。  自己手写实现了两个组件之后，先写一篇记录一些坑和学会的新东西  1、子组件不允许修改父组件数据  之前在写angular的父子组件传递数据，子组件可以修改父组件传递进来的数据。不过在Vue中子组件不允许修改父组件穿进来的值，以vue举例来说  <!--父组件HTML内容-->  <children-component :value="data"></children-component>  <!--父组件的js内容-->  export default{  data(){  return {  data : [1,2,3,4]  }  }  }  <!--子组件的js内容-->  export default{  props : ["value"],  created () {  this.value = [5,6,7,8];  }  }  此举会引发一个非常常见的报错，  IMG_256  那么解决方法是什么呢，那就要看你需求了。比如传进来的是一个渲染列表，我需要的只是修改渲染的数据，那么可以emit出去，然后在父组件重新赋值，通过双向绑定，触发子组件的再次渲染。  <!--父组件HTML内容-->  <children-component :value="data" @changeProp="changeData"></children-component>  <!--父组件的js内容-->  export default({  data(){  return {  data : [1,2,3,4]  }  },  methods : {  changeData(value){  this.data = value;  }  }  })  <!--子组件的js内容-->  export default({  props : ["value"],  created : {  this.$emit("changeProp",[5,6,7,8])  }  })  例子写的比较简单，其实原理就是emit一个数据到父组件上去，然后在父组件中接受到这个传递上来的新值，将data赋予新值，然后重新传递到了子组件，起到一个变向修改子组件的效果。  被人提醒到一个方法是用.sync，这个方法也是可以的。之前一直以为是被废弃就没用，才发现它其实是在的。看了一下.sync文档，发现这其实是一个语法糖的形式展现出来。如文档所示  <!--日常使用语法糖形态-->  <comp :foo.sync="bar"></comp>  <!--↓↓↓↓↓真实形态↓↓↓↓↓↓-->  <comp :foo="bar" @update:foo="val => bar = val"></comp>  <!--子组件js代码-->  this.$emit('update:foo', newValue)  个人感觉Vue框架其实还是不赞同直接修改数据，但是它帮你定义了一个update事件，让你在子组件可以直接显式调用，不需要自己去定义事件这么麻烦了。  2.Vue给每个组件提供了一个默认的父子传递事件v-model。  但是你每次父子组件传递数据时，都要父子处定义一个事件是很麻烦的，vue则为每个组件提供了一个默认v-model的语法糖。  <!--父组件HTML内容-->  <children-component v-mode="data" ></children-component>  <!--父组件的js内容-->  export default({  data(){  return {  data : [1,2,3,4]  }  },  watch : {  //可以在此处监听子组件传递上来的数据  data(n,o){  console.log(n,o); //[5,6,7,8],[1,2,3,4]  }  }  })  <!--子组件的js内容-->  export default({  <!--大家注意我下面这行代码是打了注释的，说明我在子组件没有定义任何属性-->  // props : ["value"]  <!--并且我接下去直接在代码中使用了this.value(一个完全没有定义过的value)。-->  created : {  console.log(this.value);  this.$emit("input",[5,6,7,8]); //当我想改变传进来值的时候  }  })  因为之前一直使用ng-model来用于表单组件的传递，所以开始对于v-mode也是这个印象，不过后面看了element-ui的源码才发现我想简单了，然后网上搜了一下对于这个语法糖的解释。  //注意，该组件不是表单组件  <children-component :value="data" @input="data = arguments[0]"></children-component>  看了这行代码大家心里估计也能明白的差不多了，其实Vue只是帮我们把父组件上的两段声明合二为一了，同时再帮我们在子组件处直接省略了定义。语法糖说到底就是帮我们省力的嘛。  3.全局一次性引用写好的组件  假如我们写好了一些组件，接下去肯定还要引入和使用吧。但是你写了这么多组件，在每个地方一个个引用想要的是一件很麻烦的事情。我们最好是在一个初始的地方一次性全部引入，然后在用的地方直接使用  (当然全部引入无可避免的会引入和打包不需要的东西，不过这是公共组件库的烦恼，我们自己写的肯定会全部用到)。  // 在文件开头初始引入所有的组件文件  import b from "./components/common/b.vue"  import c from "./components/common/c.vue"  const components = [b,c];  const install = function (Vue, opts = {}) {  components.map(component => {  Vue.component(component.name, component);  });  };  export default install  然后直接在启动的main.js文件里面引入就好  import ui from "install.js"  Vue.use(ui);  然后你就可以随意在任何一个组件里面直接调用了，比自己之前在每个组件里重复调用要方便一点。  <b v-model="data1"></b>  <c v-model="data2"></c>  好了，暂时到这里，都是自己的踩得坑和心得，希望对大家有所帮助吧，接下去还有的话再补充好了。干货不多，烂笔头为主。 |

|  |  |
| --- | --- |
|  |  |

vuex状态

|  |  |
| --- | --- |
| 概况 | 今天试了一下Vuex,感觉跟Redux的实现思想类似。再此，简单地总结一下。  什么是Vuex  在Vue中，多组件的开发给我们带来了很多的方便，但同时当项目规模变大的时候，多个组件间的数据通信和状态管理就显得难以维护。而Vuex就此应运而生。将状态管理单独拎出来，应用统一的方式进行处理，在后期维护的过程中数据的修改和维护就变得简单而清晰了。Vuex采用和Redux类似的单向数据流的方式来管理数据。用户界面负责触发动作（Action)进而改变对应状态（State），从而反映到视图（View）上。如下图所示：  IMG_256  vuex-flow.png  使用Vuex  安装:  npm install --save vuex  引入：  import Vuex from 'vuex'import Vue from 'vue'  Vue.use(Vuex)  Vuex的组成部分  完整的应用Vuex开发的应用结构应该是这样的：  IMG_257  vuex-complete.png  下面针对比较核心的部分进行说明。  **最简单的** Store  每一个 Vuex 应用的核心就是 store（仓库）。"store" 基本上就是一个容器，它包含着你的应用中大部分的状态(state)。Vuex 和单纯的全局对象有以下两点不同：  > Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。  > 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交(commit) mutations。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。  提示：我们将在后续的文档示例代码中使用 ES2015 语法。如果你还没能掌握 ES2015，**[你得抓紧了](https://babeljs.io/docs/learn-es2015/" \t "https://vuex.vuejs.org/zh-cn/_blank)**！  **[安装](https://vuex.vuejs.org/zh-cn/installation.html)** Vuex 之后，让我们来创建一个 store。创建过程直截了当——仅需要提供一个初始 state 对象和一些 mutations：  // 如果在模块化构建系统中，请确保在开头调用了 Vue.use(Vuex)  const store = new Vuex.Store({  state: {  count: 0  },  mutations: {  increment (state) {  state.count++  }  }})  现在，你可以通过 store.state 来获取状态对象，以及通过 store.commit 方法触发状态变更：  store.commit('increment')  console.log(store.state.count) // -> 1  再次强调，我们通过提交 mutation 的方式，而非直接改变 store.state.count，是因为我们想要更明确地追踪到状态的变化。这个简单的约定能够让你的意图更加明显，这样你在阅读代码的时候能更容易地解读应用内部的状态改变。此外，这样也让我们有机会去实现一些能记录每次状态改变，保存状态快照的调试工具。有了它，我们甚至可以实现如时间穿梭般的调试体验。  由于 store 中的状态是响应式的，在组件中调用 store 中的状态简单到仅需要在计算属性中返回即可。触发变化也仅仅是在组件的 methods 中提交 mutations。  这是一个**[最基本的 Vuex 记数应用](https://jsfiddle.net/n9jmu5v7/1269/" \t "https://vuex.vuejs.org/zh-cn/_blank)**示例。  接下来，我们将会更深入地探讨一些核心概念。让我们先从 **[State](https://vuex.vuejs.org/zh-cn/state.html)** 概念开始。  State  State负责存储整个应用的状态数据，一般需要在使用的时候在跟节点注入store对象，后期就可以使用this.$store.state直接获取状态  //store为实例化生成的import store from './store'  new Vue({  el: '#app',  store,  render: h => h(App)  })  这个store可以理解为一个容器，包含着应用中的state等。实例化生成store的过程是：  const mutations = {...};  const actions = {...};  const state = {...};  Vuex.Store({  state,  actions,  mutation  });  后续在组件中使用的过程中，如果想要获取对应的状态你就可以直接使用this.$store.state获取，当然，也可以利用vuex提供的mapState辅助函数将state映射到计算属性中去，如  //我是组件  import {mapState} from 'vuex'  export default {  computed: mapState({  count: state => state.count  })  }  这样直接就可以在组件中直接使用了。  Mutations  Mutations的中文意思是“变化”，利用它可以更改状态，本质就是用来处理数据的函数，其接收唯一参数值state。store.commit(mutationName)是用来触发一个mutation的方法。需要记住的是，定义的mutation必须是同步函数，否则devtool中的数据将可能出现问题，使状态改变变得难以跟踪。  const mutations = {  mutationName(state) {  //在这里改变state中的数据  }  }  在组件中触发：  //我是一个组件  export default {  methods: {  handleClick() {  this.$store.commit('mutationName')  }  }  }  或者使用辅助函数mapMutations直接将触发函数映射到methods上，这样就能在元素事件绑定上直接使用了。如：  import {mapMutations} from 'vuex'  //我是一个组件export default {  methods: mapMutations([  'mutationName'  ])  }  Actions  Actions也可以用于改变状态，不过是通过触发mutation实现的，重要的是可以包含异步操作。其辅助函数是mapActions与mapMutations类似，也是绑定在组件的methods上的。如果选择直接触发的话，使用this.$store.dispatch(actionName)方法。  //定义Actionsconst actions = {  actionName({ commit }) {  //dosomething  commit('mutationName')  }  }  在组件中使用  import {mapActions} from 'vuex'  //我是一个组件export default {  methods: mapActions([  'actionName',  ])  }  Getters  有些状态需要做二次处理，就可以使用getters。通过this.$store.getters.valueName对派生出来的状态进行访问。或者直接使用辅助函数mapGetters将其映射到本地计算属性中去。  const getters = {  strLength: state => state.aString.length  }  //上面的代码根据aString状态派生出了一个strLength状态  在组件中使用  import {mapGetters} from 'vuex'  //我是一个组件export default {  computed: mapGetters([  'strLength'  ])  }  Plugins  插件就是一个钩子函数，在初始化store的时候引入即可。比较常用的是内置的logger插件，用于作为调试使用。  import createLogger from 'vuex/dist/logger'  const store = Vuex.Store({  ...  plugins: [createLogger()]  })  最后，还有一些高级用法，如严格模式，**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)**等可能使用频率不会特别高。有需要的时候查官方文档就可以了。总的来说，Vuex还是相对比较简单的，特别是如果之前有学过Flux,Redux之类的话，上手起来更加容易。  参考文档  [http://vuex.vuejs.org/zh-cn/index.html](http://vuex.vuejs.org/zh-cn/index.html" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank) |

关于组件的组合写法

|  |  |
| --- | --- |
| 3个组件 | ./transition/t0.vue  ./transition/t1.vue  ./transition/t2.vue |
| 包含所有 | ./transition/transitionAll.vue  <**template**>  <**div**>  <**t0**></**t0**>  <**t1**></**t1**>  <**t2**></**t2**>  </**div**> </**template**>  <**script**>  **import** t0 **from** "./t0.vue";  **import** t1 **from** "./t1.vue";  **import** t2 **from** "./t2.vue";   **export default** {  data() {  **return** {  show: **true** };  },  components: {t0, t1, t2}  }; </**script**> <**style**></**style**> |
| 路由嵌套 | ./transition/transitionRouter.vue  <**template**>  <**div**>  <**router-link** to="/transition-router/t0">t0 显影</**router-link**>  <**router-link** to="/transition-router/t1">t1 复杂</**router-link**>  <**router-link** to="/transition-router/t2">t2 大小</**router-link**>  <**router-view**></**router-view**>  </**div**> </**template**>  <**script**></**script**> <**style**></**style**> |
|  | **/\*transition\*/ import** t0 **from** "./transition/t0.vue"; **import** t1 **from** "./transition/t1.vue"; **import** t2 **from** "./transition/t2.vue";  **import** transitionAll **from** "./transition/transitionAll.vue"; // 包含三个动效  **import** transitionRouter **from** "./transition/transitionRouter.vue"; // 包含子路由 |
| 使用方式1 | ==dom  <**div** id="transition">  <**t0**></**t0**>  <**t1**></**t1**>  <**t2**></**t2**> </**div**>  ==js  **/\* 直接展示3个组件\*/ new** Vue({  el: "#transition",  components: { t0, t1, t2 } }); |
| 路由展示1 | ==dom  <**router-view**></**router-view**>  <**router-link** to="/transition-all">渐变1</**router-link**>  ==js  **const** routes = [   {path: '/transition-all', component: transitionAll},  ];  **const** router = **new** VueRouter({routes});  **new** Vue({  el: "#main",  router }); |
| 路由展示2 | **const** routes = [  {  path: '/transition-router',  name: "transition",  component: transitionRouter,  children: [  {path: "", redirect: "t0"},  {path: 't0', component: t0},  {path: 't1', component: t1},  {path: 't2', component: t2}  ]  } ];  **const** router = **new** VueRouter({routes});  **new** Vue({  el: "#main",  router }); |