一 全局配置

Vue.config 是一个对象，包含Vue的全局对象。可以在启动应用前修改下列属性。

1. silent

|  |  |
| --- | --- |
| 类型 | boolean |
| 默认 | false |
| 用法 | Vue.config.silent = true |
| 职能 | 取消vue的所有日志和警告 |

2. optionMergeStrategies

|  |  |
| --- | --- |
| 类型 | {[key:string]:function} |
| 默认 | {} |
| 职能 | ● 自定义合并策略的选项。  ● 合并策略选项分别接受第一个参数作为父实例，第二个参数为子实例，Vue实例上下文被作为第三个参数传入。 |
| 用法 | Vue.config.optionMergeStrategies.\_my\_option = funciton(parent,child,vm){  return child + 1;  }  const Profile = Vue.extend({  \_my\_option = 1;  })  // Profile.options.\_my\_option = 2 |

1. devtools

|  |  |
| --- | --- |
| 类型 | boolean |
| 默认 | true (生产版为 false) |
| 职能 | 配置是否允许 vue-devtools 检查代码。开发版本默认为 true，生产版本默认为 false。生产版本设为 true 可以启用检查。 |
| 用法 | Vue.config.devtools = true; |

4. errorHandler

|  |  |
| --- | --- |
| 类型 | function |
| 默认 | 抛出错误 |
| 职能 | 指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue 实例。 |
| 用法 | Vue.config.errorHandler = function (err, vm) {  // handle error  } |

5. ignoredElements

|  |  |
| --- | --- |
| 类型 | Array[] |
| 默认 | [] |
| 职能 | 须使 Vue 忽略在 Vue 之外的自定义元素 (e.g., 使用了 Web Components APIs)。否则，它会假设你忘记注册全局组件或者拼错了组件名称，从而抛出一个关于 Unknown custom element 的警告。 |
| 用法 | Vue.config.ignoredElements = [  'my-custom-web-component', 'another-web-component'  ] |

6. keyCodes

|  |  |
| --- | --- |
| 类型 | {[key:string]:[number | Array[number]]} |
| 默认 | {} |
| 职能 | 给 v-on 自定义键位别名。 |
| 用法 | Vue.config.keyCodes = {  v: 86,  f1: 112,  mediaPlayPause: 179,  up: [38, 87]  } |

二 全局API

1. Vue.extend()

|  |  |
| --- | --- |
| 参数 | {object} options |
| 用法 | |
| 使用基础 Vue 构造器，创建一个“子类”。参数是一个包含组件选项的对象。  data 选项是特例，需要注意 - 在 Vue.extend() 中它必须是函数  <div id=”mount-point”></div>  //创建构造器  var Profile = Vue.extend({  template:”<p>{{firstName}} {{lastName}} aka {{alias}}</p>”,  data:function(){  return {  firstName: 'Walter',  lastName: 'White',  alias: 'Heisenberg'  }  }  });  new Profile().$mount(“#mount-point”); //创建 Profile 实例，并挂载到一个元素上。 | |
| 结果:  <p>Walter White aka Heisenberg</p> | |

2. Vue.nextTick([callback,context])

|  |  |
| --- | --- |
| 参数 | ◎ {function}[callback]  ◎ {object}[context] |
| 用法 | |
| 在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。  // 修改数据  vm.msg = 'Hello'  // DOM 还没有更新  Vue.nextTick(function () {  // DOM 更新了  }) | |

3. Vue.set(object,key,value)

|  |  |
| --- | --- |
| 参数 | ◎ {object} object  ◎ {string} key  ◎ {any} value |
| 返回值 | 设置的值 |
| 用法 | |
| 设置对象的属性。如果对象是响应式的，确保属性被创建后也是响应式的，同时触发视图更新。这个方法主要用于避开 Vue 不能检测属性被添加的限制。  注意对象不能是 Vue 实例，或者 Vue 实例的根数据对象 | |

4. Vue.delete(object,key)

|  |  |
| --- | --- |
| 参数 | ◎ {object} object  ◎ {string} key |
| 用法 | |
| 删除对象的属性。如果对象是响应式的，确保删除能触发更新视图。这个方法主要用于避开 Vue 不能检测到属性被删除的限制，但是你应该很少会使用它。  注意对象不能是 Vue 实例，或者 Vue 实例的根数据对象 | |

5. Vue.dirctive(id,[definition])

|  |  |
| --- | --- |
| 参数 | ◎ {string} id  ◎ {function | object} definition |
| 用法 | |
| 注册或获取全局指令。  // 注册  Vue.directive('my-directive', {  bind: function () {},  inserted: function () {},  update: function () {},  componentUpdated: function () {},  unbind: function () {}  })  // 注册（传入一个简单的指令函数）  Vue.directive('my-directive', function () {  // 这里将会被 `bind` 和 `update` 调用  })  // getter，返回已注册的指令  var myDirective = Vue.directive('my-directive') | |

6. Vue.filter(id,[definition])

|  |  |
| --- | --- |
| 参数 | ◎ {string} id  ◎ {function | object} definition |
| 用法 | |
| 注册或获取全局过滤器。  // 注册  Vue.filter('my-filter', function (value) {  // 返回处理后的值  })  // getter，返回已注册的过滤器  var myFilter = Vue.filter('my-filter') | |

7. Vue.component(id,[definition])

|  |  |
| --- | --- |
| 参数 | ◎ {string} id  ◎ {function | object} definition |
| 用法 | |
| 注册或获取全局组件。注册还会自动使用给定的id设置组件的名称  // 注册组件，传入一个扩展过的构造器  Vue.component('my-component', Vue.extend({ /\* ... \*/ }))  // 注册组件，传入一个选项对象（自动调用 Vue.extend）  Vue.component('my-component', { /\* ... \*/ })  // 获取注册的组件（始终返回构造器）  var MyComponent = Vue.component('my-component') | |

8. Vue.use(plugin)

|  |  |
| --- | --- |
| 参数 | ◎ {object | function} plugins |
| 用法 | |
| 安装 Vue.js 插件。如果插件是一个对象，必须提供 install 方法。如果插件是一个函数，它会被作为 install 方法。install 方法将被作为 Vue 的参数调用。  当 install 方法被同一个插件多次调用，插件将只会被安装一次。 | |

9. Vue.mixin(mixin)

|  |  |
| --- | --- |
| 参数 | ◎ {objects} plugins |
| 用法 | |
| 全局注册一个混合，影响注册之后所有创建的每个 Vue 实例。插件作者可以使用混合，向组件注入自定义的行为。不推荐在应用代码中使用。 | |

10. Vue.compile(template);

|  |  |
| --- | --- |
| 参数 | ◎ {string} template |
| 用法 | |
| 在render函数中编译模板字符串。只在独立构建时有效  var res = Vue.compile('<div><span>{{ msg }}</span></div>')  new Vue({  data: {  msg: 'hello'  },  render: res.render,  staticRenderFns: res.staticRenderFns  }) | |

三 选项/数据

1. data

|  |  |
| --- | --- |
| 类型 | object | function |
| 限制 | 组件的定义只接受function |
| 详细 | |
| Vue 实例的数据对象。Vue 将会递归将 data 的属性转换为 getter/setter，从而让 data 的属性能够响应数据变化。对象必须是纯粹的对象(含有零个或多个的key/value对)：浏览器 API 创建的原生对象，原型上的属性会被忽略。大概来说，data 应该只能是数据 - 不推荐观察拥有状态行为的对象。  一旦观察过，不需要再次在数据对象上添加响应式属性。因此推荐在创建实例之前，就声明所有的根级响应式属性。  实例创建之后，可以通过 vm.$data 访问原始数据对象。Vue 实例也代理了 data 对象上所有的属性，因此访问 vm.a 等价于访问 vm.$data.a。  以 \_ 或 $ 开头的属性 不会 被 Vue 实例代理，因为它们可能和 Vue 内置的属性、 API 方法冲突。你可以使用例如 vm.$data.\_property 的方式访问这些属性。  当一个组件被定义， data 必须声明为返回一个初始数据对象的函数，因为组件可能被用来创建多个实例。如果 data 仍然是一个纯粹的对象，则所有的实例将共享引用同一个数据对象！通过提供 data 函数，每次创建一个新实例后，我们能够调用 data 函数，从而返回初始数据的一个全新副本数据对象。  如果需要，可以通过将 vm.$data 传入 JSON.parse(JSON.stringify(...)) 得到深拷贝的原始数据对象。  //先用JSON.stringify(...)把对象解析成字符串  //在用JSON.parse()把字符串解析回对象 | |
| 示例 | |
| var data = { a: 1 }  // 直接创建一个实例  var vm = new Vue({  data: data  })  vm.a // -> 1  vm.$data === data // -> true  // Vue.extend() 中 data 必须是函数  var Component = Vue.extend({  data: function () {  return { a: 1 }  }  }) | |
| 注意，不应该对 data 属性使用箭头函数 (例如data: () => { return { a: this.myProp }})。理由是箭头函数绑定了父级作用域的上下文，所以 this 将不会按照期望指向 Vue 实例，this.myProp 将是 undefined | |

2. props

|  |  |
| --- | --- |
| 类型 | object | array |
| 详细 | |
| props 可以是数组或对象，用于接收来自父组件的数据。props 可以是简单的数组，或者使用对象作为替代，对象允许配置高级选项，如类型检测、自定义校验和设置默认值。 | |
| 示例 | |
| // 简单语法  Vue.component('props-demo-simple', {  props: ['size', 'myMessage']  })  // 对象语法，提供校验  Vue.component('props-demo-advanced', {  props: {  // 只检测类型  height: Number,  // 检测类型 + 其他验证  age: {  type: Number,  default: 0,  required: true,  validator: function (value) {  return value >= 0  }  }  }  }) | |

3.propsData

|  |  |
| --- | --- |
| 类型 | {[key:string] : any} |
| 限制 | 只用于 new 创建的实例中。 |
| 详细 | |
| 创建实例时传递 props。主要作用是方便测试。 | |
| 示例 | |
| var Comp = Vue.extend({  props: ['msg'],  template: '<div>{{ msg }}</div>'  })  var vm = new Comp({  propsData: {  msg: 'hello'  }  }) | |

4. computed

|  |  |
| --- | --- |
| 类型 | { [key: string]: Function | { get: Function, set: Function } } |
| 详细 | |
| 计算属性将被混入到 Vue 实例中。所有 getter 和 setter 的 this 上下文自动地绑定为 Vue 实例。  计算属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。 | |
| 示例 | |
| var vm = new Vue({  data: { a: 1 },  computed: {  // 仅读取，值只须为函数  aDouble: function () {  return this.a \* 2  },  // 读取和设置  aPlus: {  get: function () {  return this.a + 1  },  set: function (v) {  this.a = v - 1  }  }  }  })  vm.aPlus // -> 2  vm.aPlus = 3  vm.a // -> 2  vm.aDouble // -> 4 | |

5. methods

|  |  |
| --- | --- |
| 类型 | { [key: string]: Function } |
| 详细 | |
| methods 将被混入到 Vue 实例中。可以直接通过 VM 实例访问这些方法，或者在指令表达式中使用。方法中的 this 自动绑定为 Vue 实例。 | |
| 示例 | |
| var vm = new Vue({  data: { a: 1 },  methods: {  plus: function () {  this.a++  }  }  })  vm.plus()  vm.a // 2 | |

6. watch

|  |  |
| --- | --- |
| 类型 | { [key: string]: string | Function | Object } |
| 详细 | |
| 一个对象，键是需要观察的表达式，值是对应回调函数。值也可以是方法名，或者包含选项的对象。Vue 实例将会在实例化时调用 $watch()，遍历 watch 对象的每一个属性。 | |
| 示例 | |
| var vm = new Vue({  data: {  a: 1,  b: 2,  c: 3  },  watch: {  a: function (val, oldVal) {  console.log('new: %s, old: %s', val, oldVal)  },  // 方法名  b: 'someMethod',  // 深度 watcher  c: {  handler: function (val, oldVal) { /\* ... \*/ },  deep: true  }  }  })  vm.a = 2 // -> new: 2, old: 1 | |

四 选项/Dom

1. el

|  |  |
| --- | --- |
| 类型 | string | HTMLElement |
| 限制 | 只在由 new 创建的实例中遵守。 |
| 详细 | |
| 提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 HTMLElement 实例。  在实例挂载之后， 元素可以用 vm.$el 访问。  如果这个选项在实例化时有作用，实例将立即进入编译过程，否则，需要显式调用 vm.$mount() 手动开启编译。  提供的元素只能作为挂载点。不同于 Vue 1.x，所有的挂载元素会被 Vue 生成的 DOM 替换。因此不推荐挂载root实例到 <html> 或者 <body> 上。  如果 render 函数和 template 属性都不存在，挂载 DOM 元素的 HTML 会被提取出来用作模板，此时，必须使用 Runtime + Compiler 构建的 Vue 库。 | |

2. template

|  |  |
| --- | --- |
| 类型 | string |
| 详细 | |
| 一个字符串模板作为 Vue 实例的标识使用。模板将会 替换 挂载的元素。挂载元素的内容都将被忽略，除非模板的内容有分发 slot。  如果值以 # 开始，则它用作选项符，将使用匹配元素的 innerHTML 作为模板。常用的技巧是用 <script type="x-template"> 包含模板。  如果 Vue 选项中包含 render 函数，template 选项将被忽略。 | |

3. render

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 字符串模板的代替方案，允许你发挥 JavaScript 最大的编程能力。render 函数接收一个 createElement 方法作为第一个参数用来创建 VNode。  如果组件是一个函数组件，Render 函数还会接收一个额外的 context 参数，为没有实例的函数组件提供上下文信息  Vue 选项中的 render 函数若存在，则 Vue 构造函数不会从 template 选项或通过 el 选项指定的挂载元素中提取出的 HTML 模板编译 render 函数。 | |

4. 选项/生命周期钩子

所有的生命周期钩子自动绑定 this 上下文到实例中，因此你可以访问数据，对属性和方法进行运算。这意味着 你不能使用箭头函数来定义一个生命周期方法 (例如 created: () => this.fetchTodos())。这是因为箭头函数绑定了父上下文，因此 this 与你期待的 Vue 实例不同， this.fetchTodos 的行为未定义。

1. beforeCreate

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 在实例初始化之后，数据观测(data observer) 和 event/watcher 事件配置之前被调用。 | |

2. created

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。 | |

3. beforeMount

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 在挂载开始之前被调用：相关的 render 函数首次被调用。  该钩子在服务器端渲染期间不被调用。 | |

4. mounted

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.$el 也在文档内。  该钩子在服务器端渲染期间不被调用。 | |

5.beforeUpdate

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。  你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。  该钩子在服务器端渲染期间不被调用。 | |

6. updated

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。  当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态，因为这可能会导致更新无限循环。  该钩子在服务器端渲染期间不被调用。 | |

7.activated

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| keep-alive 组件激活时调用。  该钩子在服务器端渲染期间不被调用。 | |

8.deactivated

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| keep-alive 组件停用时调用。  该钩子在服务器端渲染期间不被调用。 | |

9. beforeDestroy

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| 实例销毁之前调用。在这一步，实例仍然完全可用。  该钩子在服务器端渲染期间不被调用。 | |

10. destroyed

|  |  |
| --- | --- |
| 类型 | function |
| 详细 | |
| Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。  该钩子在服务器端渲染期间不被调用。 | |

五 选项/资源

1. directives

|  |  |
| --- | --- |
| 类型 | object |
| 详细 | |
| 包含 Vue 实例可用指令的哈希表。 | |

2. filters

|  |  |
| --- | --- |
| 类型 | object |
| 详细 | |
| 包含 Vue 实例可用过滤器的哈希表。 | |

3. components

|  |  |
| --- | --- |
| 类型 | object |
| 详细 | |
| 包含 Vue 实例可用组件的哈希表。 | |

六 选项/杂项

1. parent

|  |  |
| --- | --- |
| 类型 | vue instance(vue 实例) |
| 详细 | |
| 指定已创建的实例之父实例，在两者之间建立父子关系。子实例可以用 this.$parent 访问父实例，子实例被推入父实例的 $children 数组中。  同时使用 $parent 和 $children 有冲突 - 他们作为同一个入口 。更推荐用 props 和 events 实现父子组件通信 | |

2. mixin

|  |  |
| --- | --- |
| 类型 | Array<Object> |
| 详细 | |
| mixins 选项接受一个混合对象的数组。这些混合实例对象可以像正常的实例对象一样包含选项,他们将在 Vue.extend() 里最终选择使用相同的选项合并逻辑合并。举例：如果你混合包含一个钩子而创建组件本身也有一个,两个函数将被调用。  Mixin钩子按照传入顺序依次调用,并在调用组件自身的钩子之前被调用。 | |
| 示例 | |
| var mixin = {  created: function () { console.log(1) }  }  var vm = new Vue({  created: function () { console.log(2) },  mixins: [mixin]  })  // -> 1  // -> 2 | |

3. name

|  |  |
| --- | --- |
| 类型 | string |
| 限制 | 只有作为组件选项时起作用。 |
| 详细 | |
| 允许组件模板递归地调用自身。注意，组件在全局用 Vue.component() 注册时，全局 ID 自动作为组件的 name。  指定 name 选项的另一个好处是便于调试。有名字的组件有更友好的警告信息。另外，当在有 vue-devtools, 未命名组件将显示成 <AnonymousComponent>, 这很没有语义。通过提供 name 选项，可以获得更有语义信息的组件树。 | |

4. extends

|  |  |
| --- | --- |
| 类型 | Object | Function |
| 详细 | |
| 允许声明扩展另一个组件(可以是一个简单的选项对象或构造函数),而无需使用 Vue.extend。这主要是为了便于扩展单文件组件。  这和 mixins 类似，区别在于，组件自身的选项会比要扩展的源组件具有更高的优先级。 | |
| 示例 | |
| var CompA = { ... }  // 在没有调用 Vue.extend 时候继承 CompA  var CompB = {  extends: CompA,  ...  } | |

5. delimiters

|  |  |
| --- | --- |
| 类型 | Array<string> |
| 默认值 | ["{{", "}}"] |
| 详细 | |
| 改变纯文本插入分隔符。 这个选择只有在独立构建时才有用。 | |
| 示例 | |
| new Vue({  delimiters: ['${', '}']  })  // 分隔符变成了 ES6 模板字符串的风格 | |

6. functional

|  |  |
| --- | --- |
| 类型 | boolean |
| 详细 | |
| 使组件无状态（没有 data ）和无实例（没有 this 上下文）。他们用一个简单的 render 函数返回虚拟节点使他们更容易渲染。 | |

七 实例属性

1. vm.$data

|  |  |
| --- | --- |
| 类型 | object |
| 详细 | |
| Vue 实例观察的数据对象。Vue 实例代理了对其 data 对象属性的访问。 | |

2. vm.$el

|  |  |
| --- | --- |
| 类型 | HTMLElement |
| 详细 | |
| Vue 实例使用的根 DOM 元素。 | |

3. vm.$options

|  |  |
| --- | --- |
| 类型 | object |
| 详细 | |
| 用于当前 Vue 实例的初始化选项。需要在选项中包含自定义属性时会有用处： | |
| new Vue({  customOption: 'foo', //自定义属性  created: function () {  console.log(this.$options.customOption) // -> 'foo'  }  }) | |

4. vm.$parent

|  |  |
| --- | --- |
| 类型 | Vue instance 只读 |
| 详细 | |
| 父实例，如果当前实例有的话。 | |

5. vm.$root

|  |  |
| --- | --- |
| 类型 | Vue instance 只读 |
| 详细 | |
| 当前组件树的根 Vue 实例。如果当前实例没有父实例，此实例将会是其自已。 | |

6. vm.$children

|  |  |
| --- | --- |
| 类型 | Vue instance 只读 |
| 详细 | |
| 当前实例的直接子组件。需要注意 $children 并不保证顺序，也不是响应式的。如果你发现自己正在尝试使用 $children 来进行数据绑定，考虑使用一个数组配合 v-for 来生成子组件，并且使用 Array 作为真正的来源。 | |

7. vm.$slots

|  |  |
| --- | --- |
| 类型 | { [name: string]: ?Array<VNode> } 只读 |
| 详细 | |
| 用来访问被 slot 分发的内容。每个具名 slot 有其相应的属性（例如：slot="foo" 中的内容将会在 vm.$slots.foo 中被找到）。default 属性包括了所有没有被包含在具名 slot 中的节点 | |
| 示例 | |
| <blog-post>  <h1 slot="header">  About Me  </h1>  <p>Here's some page content, which will be included in vm.$slots.default, because it's not inside a named slot.</p>  <p slot="footer">  Copyright 2016 Evan You  </p>  <p>If I have some content down here, it will also be included in vm.$slots.default.</p>.  </blog-post>  Vue.component('blog-post', {  render: function (createElement) {  var header = this.$slots.header  var body = this.$slots.default  var footer = this.$slots.footer  return createElement('div', [  createElement('header', header),  createElement('main', body),  createElement('footer', footer)  ])  }  }) | |

8. vm.$scopedSlots——2.1.0新增

|  |  |
| --- | --- |
| 类型 | { [name: string]: props => VNode | Array<VNode> } 只读 |
| 详细 | |
| 用来访问 scoped slots。对于包括 默认 slot 在内的每一个 slot， 该对象都包含一个返回相应 VNode 的函数。  在使用 render 函数 书写一个组件时，访问 vm.$scopedSlots 最有帮助。 | |

9. vm.$ref

|  |  |
| --- | --- |
| 类型 | Object 只读 |
| 详细 | |
| 一个对象，其中包含了所有拥有 ref 注册的子组件。 | |

10. vm.$isServer

|  |  |
| --- | --- |
| 类型 | boolean 只读 |
| 详细 | |
| 当前 Vue 实例是否运行于服务器。 | |

八 实例方法/数据

1. vm.$watch(expOrFn,callback,[option])

|  |  |
| --- | --- |
| 参数 | ◎ {string | Function} expOrFn  ◎ {Function} callback  ◎ {Object} [options]  ● {boolean} deep //深处  ● {boolean} immediate //直接的 |
| 返回值 | {Function} unwatch |
| 用法 | |
| 观察 Vue 实例变化的一个表达式或计算属性函数。回调函数得到的参数为新值和旧值。表达式只接受监督的键路径。对于更复杂的表达式，用一个函数取代。  注意：在变异（不是替换）对象或数组时，旧值将与新值相同，因为它们的引用指向同一个对象/数组。Vue 不会保留变异之前值的副本。 | |
| 示例 | |
| // 键路径  vm.$watch('a.b.c', function (newVal, oldVal) {  // 做点什么  })  // 函数  vm.$watch(  function () {  return this.a + this.b  },  function (newVal, oldVal) {  // 做点什么  }  )  vm.$watch 返回一个取消观察函数，用来停止触发回调：  var unwatch = vm.$watch('a', cb)  // 之后取消观察  unwatch() | |
| 选项：deep  为了发现对象内部值的变化，可以在选项参数中指定 deep: true 。注意监听数组的变动不需要这么做。  vm.$watch('someObject', callback, {  deep: true  })  vm.someObject.nestedValue = 123  // callback is fired | |
| 选项：immediate  在选项参数中指定 immediate: true 将立即以表达式的当前值触发回调：  vm.$watch('a', callback, {  immediate: true  })  // 立即以 `a` 的当前值触发回调 | |

2. vm.$set(object,key,value)

|  |  |
| --- | --- |
| 参数 | ◎ {Object} Object  ◎ {string} key  ◎ {any} value |
| 返回值 | 设置的值. |
| 用法 | |
| 这是全局 Vue.set 的别名。 | |

3. vm.$delete(object,key)

|  |  |
| --- | --- |
| 参数 | ◎ {Object} Object  ◎ {string} key |
| 用法 | |
| 这是全局 Vue.delete 的别名。 | |

九 实例方法/事件

1. vm.$on(event,callback)

|  |  |
| --- | --- |
| 参数 | ◎ {string} event  ◎ {function} callback |
| 用法 | |
| 监听当前实例上的自定义事件。事件可以由vm.$emit触发。回调函数会接收所有传入事件触发函数的额外参数。 | |
| vm.$on('test', function (msg1,msg2) {  console.log(msg1)  console.log(msg2)  })  vm.$emit('test', 'hi',”you”)  // -> "hi" | |

2. vm.$once(event,callback)

|  |  |
| --- | --- |
| 参数 | ◎ {string} event  ◎ {function} callback |
| 用法 | |
| 监听一个自定义事件，但是只触发一次，在第一次触发之后移除监听器。 | |

3.vm.$off([event,callback])

|  |  |
| --- | --- |
| 参数 | ◎ {string} event  ◎ {function} callback |
| 用法 | |
| 移除事件监听  ◎ 如果没有提供参数，则移除所有的事件监听器；  ◎ 如果只提供了事件，则移除该事件所有的监听器；  ◎ 如果同时提供了事件与回调，则只移除这个回调的监听器。 | |

4. vm.$emit(evebt[,...args])

|  |  |
| --- | --- |
| 参数 | ◎ {string} event  ◎ {any} args |
| 用法 | |
| 触发当前实例上的事件。附加参数都会传给监听器回调。 | |

十 实例方法/生命周期

1. vm.$mount([elementOrSelector])

|  |  |
| --- | --- |
| 参数 | ◎ {Element | string} [elementOrSelector]  ◎ {boolean} [hydrating] |
| 返回值 | vm-实例自身 |
| 用法 | |
| 如果没有提供 elementOrSelector 参数，模板将被渲染为文档之外的的元素，并且你必须使用原生DOM API把它插入文档中。  这个方法返回实例自身，因而可以链式调用其它实例方法。 | |
| 示例 | |
| var MyComponent = Vue.extend({  template: '<div>Hello!</div>'  })  // 创建并挂载到 #app (会替换 #app)  new MyComponent().$mount('#app')  // 同上  new MyComponent({ el: '#app' })  // 或者，在文档之外渲染并且随后挂载  var component = new MyComponent().$mount()  document.getElementById('app').appendChild(component.$el) | |

2. vm.$forceUpdate()

|  |  |
| --- | --- |
| 参数 | 无 |
| 用法 | |
| 迫使Vue实例重新渲染。注意它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。 | |

3. vm.$nextTick([callback])

|  |  |
| --- | --- |
| 参数 | {Function} [callback] |
| 用法 | |
| 将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新。它跟全局方法 Vue.nextTick 一样，不同的是回调的 this 自动绑定到调用它的实例上。 | |
| 示例 | |
| new Vue({  methods: { example: function () {  // 修改数据  this.message = 'changed'  // DOM 还没有更新  this.$nextTick(function () {  // DOM 现在更新了  // `this` 绑定到当前实例  this.doSomethingElse()  })  }  }  }) | |

4. vm.$destory()

|  |  |
| --- | --- |
| 参数 | 无 |
| 用法 | |
| 完全销毁一个实例。清理它与其它实例的连接，解绑它的全部指令及事件监听器。  触发 beforeDestroy 和 destroyed 的钩子。  在大多数场景中你不应该调用这个方法。最好使用 v-if 和 v-for 指令以数据驱动的方式控制子组件的生命周期。 | |

十一 指令

1. v-text

|  |  |
| --- | --- |
| 类型 | string |
| 详细 | |
| 更新元素的 textContent。如果要更新部分的 textContent ，需要使用 {{ Mustache }} 插值。 | |
| 示例 | |
| <span v-text="msg"></span>  <!-- 和下面的一样 -->  <span>{{msg}}</span> | |

2. v-html

|  |  |
| --- | --- |
| 类型 | string |
| 详细 | |
| 更新元素的 innerHTML 。注意：内容按普通 HTML 插入 - 不会作为 Vue 模板进行编译 。如果试图使用 v-html 组合模板,可以重新考虑是否通过使用组件来替代。  在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。只在可信内容上使用 v-html，永不用在用户提交的内容上。 | |
| 示例 | |
| <div v-html="html"></div> | |

3. v-show

|  |  |
| --- | --- |
| 类型 | any |
| 详细 | |
| 根据表达式之真假值，切换元素的 display CSS 属性。  当条件变化时该指令触发过渡效果。 | |

1. v-if

|  |  |
| --- | --- |
| 类型 | any |
| 详细 | |
| 根据表达式的值的真假条件渲染元素。在切换时元素及它的数据绑定 / 组件被销毁并重建。如果元素是 <template> ，将提出它的内容作为条件块。  当条件变化时该指令触发过渡效果。 | |

5. v-else

|  |  |
| --- | --- |
| 类型 | 不需要表达式 |
| 限制 | 前一兄弟元素必须有 v-if 或 v-else-if。 |
| 详细 | |
| 为 v-if 或者 v-else-if 添加 “else 块”。 | |
| <div v-if="Math.random() > 0.5">  Now you see me  </div>  <div v-else>  Now you don't  </div> | |

6. v-else-if

|  |  |
| --- | --- |
| 类型 | any |
| 限制 | 前一兄弟元素必须有 v-if 或 v-else-if。 |
| 用法 | |
| 表示 v-if 的 “else if 块”。可以链式调用。 | |
| <div v-if="type === 'A'">  A  </div>  <div v-else-if="type === 'B'">  B  </div>  <div v-else-if="type === 'C'">  C  </div>  <div v-else>  Not A/B/C  </div> | |

7. v-for

|  |  |
| --- | --- |
| 类型 | Array | Object | number | string |
| 用法 | |
| 基于源数据多次渲染元素或模板块。此指令之值，必须使用特定语法 alias in expression ，为当前遍历的元素提供别名： | |
| <div v-for="item in items">  {{ item.text }}  </div> | |
| 另外也可以为数组索引指定别名（或者用于对象的键）：  <div v-for="(item, index) in items"></div>  <div v-for="(val, key) in object"></div>  <div v-for="(val, key, index) in object"></div> | |
| v-for 默认行为试着不改变整体，而是替换元素。迫使其重新排序的元素,您需要提供一个 key 的特殊属性:  <div v-for="item in items" :key="item.id">  {{ item.text }}  </div> | |

8. v-on

|  |  |
| --- | --- |
| 类型 | Array | Object | number | string |
| 缩写 | @ |
| 修饰符 | ◎ .stop  ◎ .prevent //阻止事件默认行为  ◎ .capture //添加事件捕获模式  ◎ .self //仅元素本身才可触发事件，子元素无效  ◎ .once //仅执行一次 |
| 用法 | |
| 绑定事件监听器。事件类型由参数指定。表达式可以是一个方法的名字或一个内联语句，如果没有修饰符也可以省略。  用在普通元素上时，只能监听 原生 DOM 事件。用在自定义元素组件上时，也可以监听子组件触发的自定义事件。  在监听原生 DOM 事件时，方法以事件为唯一的参数。如果使用内联语句，语句可以访问一个 $event 属性： v-on:click="handle('ok', $event)"。 | |
| 示例 | |
| <!-- 方法处理器 -->  <button v-on:click="doThis"></button>  <!-- 内联语句 -->  <button v-on:click="doThat('hello', $event)"></button>  <!-- 缩写 -->  <button @click="doThis"></button>  <!-- 停止冒泡 -->  <button @click.stop="doThis"></button>  <!-- 阻止默认行为 -->  <button @click.prevent="doThis"></button>  <!-- 阻止默认行为，没有表达式 -->  <form @submit.prevent></form>  <!-- 串联修饰符 -->  <button @click.stop.prevent="doThis"></button>  <!-- 键修饰符，键别名 -->  <input @keyup.enter="onEnter">  <!-- 键修饰符，键代码 -->  <input @keyup.13="onEnter"> | |
| 在子组件上监听自定义事件（当子组件触发 “my-event” 时将调用事件处理器）：  <my-component @my-event="handleThis"></my-component>  <!-- 内联语句 -->  <my-component @my-event="handleThis(123, $event)"></my-component>  <!-- 组件中的原生事件 必须使用native修饰符-->  <my-component @click.native="onClick"></my-component> | |

9. v-bind

|  |  |
| --- | --- |
| 类型 | any (with argument) | Object (without argument) |
| 参数 | attrOrProp (optional) |
| 缩写 | ： |
| 修饰符 | ◎ .prop //被用于绑定 DOM 属性。(what’s the difference?)  ◎.camel //transform the kebab-case attribute name into camelCase. (supported since 2.1.0) |
| 用法 | |
| 动态地绑定一个或多个特性，或一个组件 prop 到表达式。  在绑定 class 或 style 特性时，支持其它类型的值，如数组或对象。可以通过下面的教程链接查看详情。  在绑定 prop 时，prop 必须在子组件中声明。可以用修饰符指定不同的绑定类型。  没有参数时，可以绑定到一个包含键值对的对象。注意此时 class 和 style 绑定不支持数组和对象。 | |
| 示例 | |
| <!-- 绑定一个属性 -->  <img v-bind:src="imageSrc">  <!-- 缩写 -->  <img :src="imageSrc">  <!-- with inline string concatenation -->  <img :src="'/path/to/images/' + fileName">  <!-- class 绑定 -->  <div :class="{ red: isRed }"></div>  <div :class="[classA, classB]"></div>  <div :class="[classA, { classB: isB, classC: isC }]">  <!-- style 绑定 -->  <div :style="{ fontSize: size + 'px' }"></div>  <div :style="[styleObjectA, styleObjectB]"></div>  <!-- 绑定一个有属性的对象 -->  <div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>  <!-- 通过 prop 修饰符绑定 DOM 属性 -->  <div v-bind:text-content.prop="text"></div>  <!-- prop 绑定. “prop” 必须在 my-component 中声明。 -->  <my-component :prop="someThing"></my-component>  <!-- XLink -->  <svg><a :xlink:special="foo"></a></svg> | |
| .camel 修饰符允许在使用 DOM 模板时将 v-bind 属性名称驼峰化，例如 SVG 的 viewBox 属性：  <svg :view-box.camel="viewBox"></svg>  在使用字符串模板或通过 vue-loader/vueify 编译时，无需使用 .camel。 | |

10. v-model

|  |  |
| --- | --- |
| 类型 | 随表单类型不同而不同 |
| 限制 | ◎ input  ◎ select  ◎ textarea  ◎ components |
| 修饰符 | ◎ .lazy //取代input监听change事件  ◎.number //输入字符串转换为数字类型  ◎ .trim //去除首尾空格 |
| 用法 | |
| 表单控件或者组件上创建双向绑定。细节请看下面链接的教程。 | |

11. v-pre

|  |  |
| --- | --- |
| 类型 | 不需要表达式 |
| 用法 | |
| 跳过这个元素和它的子元素的编译过程。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译。 | |
| 示例 | |
| <span v-pre>{{ this will not be compiled }}</span> | |

12. v-cloak

|  |  |
| --- | --- |
| 类型 | 不需要表达式 |
| 用法 | |
| 这个指令保持在元素上直到关联实例结束编译。和 CSS 规则如 [v-cloak] { display: none } 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。 | |
| 示例 | |
| //css  [v-cloak] {  display: none;  }  //html  <div v-cloak>  {{ message }}  </div>  不会显示，直到编译结束。 | |

13. v-once

|  |  |
| --- | --- |
| 类型 | 不需要表达式 |
| 用法 | |
| 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。 | |
| 示例 | |
| <!-- 单个元素 -->  <span v-once>This will never change: {{msg}}</span>  <!-- 有子元素 -->  <div v-once>  <h1>comment</h1>  <p>{{msg}}</p>  </div>  <!-- 组件 -->  <my-component v-once :comment="msg"></my-component>  <!-- v-for 指令-->  <ul>  <li v-for="i in list" v-once>{{i}}</li>  </ul> | |

十二 特殊属性

1. key

|  |  |
| --- | --- |
| 类型 | string |
| 用法 | |
| key 的特殊属性主要用在 Vue的虚拟DOM算法，在新旧nodes对比时辨识VNodes。如果不使用key，Vue会使用一种最大限度减少动态元素并且尽可能的尝试修复/再利用相同类型元素的算法。使用key，它会基于key的变化重新排列元素顺序，并且会移除key不存在的元素。  有相同父元素的子元素必须有独特的key。重复的key会造成渲染错误。  最常见的用例是结合 v-for:  <ul>  <li v-for="item in items" :key="item.id">...</li>  </ul>  它也可以用于强制替换元素/组件而不是重复使用它。当你遇到如下场景时它可能会很有用:  ◎ 完整触发组件的生命周期钩子  ◎ 触发过渡 | |
| <transition>  <span :key="text">{{ text }}</span>  </transition>  当 text 发生改变时，<span> 会随时被更新，因此会触发过渡。 | |

2. ref

|  |  |
| --- | --- |
| 类型 | string |
| 用法 | |
| ref 被用来给元素或子组件注册引用信息。引用信息将会注册在父组件的 $refs 对象上。如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素; 如果用在子组件上，引用就指向组件实例:  <!-- vm.$refs.p will be the DOM node -->  <p ref="p">hello</p>  <!-- vm.$refs.child will be the child comp instance -->  <child-comp ref="child"></child-comp>  当 v-for 用于元素或组件的时候，引用信息将是包含 DOM 节点或组件实例的数组。  关于ref注册时间的重要说明: 因为ref本身是作为渲染结果被创建的，在初始渲染的时候你不能访问它们 - 它们还不存在！$refs 也不是响应式的，因此你不应该试图用它在模版中做数据绑定。 | |

3. slot

|  |  |
| --- | --- |
| 类型 | string |
| 用法 | |
| 用于标记往哪个slot中插入子组件内容。  详细用法，请参考下面指南部分的链接。 | |

十三 内置组件

1. component

|  |  |
| --- | --- |
| props(属性) | ◎ is //指定组件名  ◎ inline-template //boolean |
| 用法 | |
| 渲染一个“元组件”为动态组件。依 is 的值，来决定哪个组件被渲染。 | |
| <!-- 动态组件由 vm 实例的属性值 `componentId` 控制 -->  <component :is="componentId"></component>  <!-- 也能够渲染注册过的组件或 prop 传入的组件 -->  <component :is="$options.components.child"></component> | |

2. transition

|  |  |
| --- | --- |
| props(属性) | ◎ name - string，用于自动生成 CSS 过渡类名。  ◎ appear - boolean，默认false；是否在初始渲染时使用过渡。  ◎ css - boolean，默认true；是否使用css过渡类。  ◎ type - string，有效值为 "transition" 和 "animation"。  ◎ mode - string， 控制离开/进入的过渡时间序列。有效的模式有 "out-in" 和 "in-out"；默认同时生效。  ◎ enter-class - string  ◎ leave-class - string  ◎ enter-active-class - string  ◎ leave-active-class - string  ◎ appear-class - string  ◎ appear-active-class - string |
| 事件 | ◎ before-enter  ◎ enter  ◎ after-enter  ◎ before-leaver  ◎ leaver  ◎ after-leaver  ◎ before-appear  ◎ appear  ◎ after-appear |
| 用法 | |
| <transition> 元素作为单个元素/组件的过渡效果。<transition> 不会渲染额外的 DOM 元素，也不会出现在检测过的组件层级中。它只是将内容包裹在其中，简单的运用过渡行为。 | |
| <!-- 简单元素 -->  <transition>  <div v-if="ok">toggled content</div>  </transition>  <!-- 动态组件 -->  <transition name="fade" mode="out-in" appear>  <component :is="view"></component>  </transition>  <!-- 事件钩子 -->  <div id="transition-demo">  <transition @after-enter="transitionComplete">  <div v-show="ok">toggled content</div>  </transition>  </div>  new Vue({  methods: {  transitionComplete: function (el) {  // 传入 'el' 这个 DOM 元素作为参数。  }  }  }).$mount('#transition-demo') | |

3. transition-group

|  |  |
| --- | --- |
| props(属性) | ◎ tag - string，默认<span>。  ◎ move-class，覆盖移动过渡期间应用的 CSS 类。。  ◎ 除了 mode，其他特性和 <transition> 相同。 |
| 事件 | ◎ 事件和 <transition> 相同. |
| 用法 | |
| <transition-group> 元素作为多个元素/组件的过渡效果。<transition-group> 渲染一个真实的 DOM 元素。默认渲染 <span>，可以通过 tag 属性配置哪个元素应该被渲染。  注意，每个 <transition-group> 的子节点必须有 独立的key ，动画才能正常工作  <transition-group> 支持通过 CSS transform 过渡移动。当一个子节点被更新，从屏幕上的位置发生变化，它将会获取应用 CSS 移动类（通过 name 属性或配置 move-class 属性自动生成）。如果 CSS transform 属性是“可过渡”属性，当应用移动类时，将会使用 FLIP 技术 使元素流畅地到达动画终点。 | |
| <transition-group tag="ul" name="slide">  <li v-for="item in items" :key="item.id">  {{ item.text }}  </li>  </transition-group> | |

4. keep-alive

|  |  |
| --- | --- |
| props(属性) | ◎ include - 字符串或正则表达式，只有匹配的组件会被缓存  ◎ exclude - 字符串或正则表达式，只有匹配的组件不会被缓存 |
| 用法 | |
| <keep-alive> 包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。和 <transition> 相似，<keep-alive> 是一个抽象组件：它自身不会渲染一个 DOM 元素，也不会出现在父组件链中。  当组件在 <keep-alive> 内被切换，它的 activated 和 deactivated 这两个生命周期钩子函数将会被对应执行。  主要用于保留组件状态或避免重新渲染。 | |
| <!-- 基本 -->  <keep-alive>  <component :is="view"></component>  </keep-alive>  <!-- 多个条件判断的子组件 -->  <keep-alive>  <comp-a v-if="a > 1"></comp-a>  <comp-b v-else></comp-b>  </keep-alive>  <!-- 和 <transition> 一起使用 -->  <transition>  <keep-alive>  <component :is="view"></component>  </keep-alive>  </transition> | |
| include 和 exclude 属性允许组件有条件地缓存。二者都可以用逗号分隔字符串或正则表达式来表示:  <!-- 逗号分隔字符串 -->  <keep-alive include="a,b">  <component :is="view"></component>  </keep-alive>  <!-- 正则表达式 (使用 v-bind) -->  <keep-alive :include="/a|b/">  <component :is="view"></component>  </keep-alive>  匹配首先检查组件自身的 name 选项，如果 name 选项不可用，则匹配它的局部注册名称（父组件 components 选项的键值）。匿名组件不能被匹配。  <keep-alive> 不会在函数式组件中正常工作，因为它们没有缓存实例。 | |

5. slot

|  |  |
| --- | --- |
| props(属性) | ◎ name - string, 用于命名插槽。 |
| 用法 | |
| <slot> 元素作为组件模板之中的内容分发插槽。 <slot> 元素自身将被替换。 | |