

全局配置

`Vue.config` 是一个对象，包含 `Vue` 的全局配置。可以在启动应用之前修改下面属性：

debug

- 类型： `Boolean`
- 默认值： `false`
- 用法：

JS

```
Vue.config.debug = true
```

在调试模式中，`Vue` 会：

1. 为所有的警告打印栈追踪。
2. 把所有的锚节点以注释节点显示在 `DOM` 中，更易于检查渲染结果的结构。

!

只有开发版本可以使用调试模式。

delimiters

- 类型： `Array<String>`
- 默认值： `["{{", "}}"]`
- 用法：

JS

```
// ES6 模板字符串  
Vue.config.delimiters = ['${', '}']
```

修改文本插值的定界符。

unsafeDelimiters

- 类型: `Array<String>`
- 默认值: `["{{", "}}"]`
- 用法:

JS

```
// make it look more dangerous  
Vue.config.unsafeDelimiters = ['{!!', '!!']
```

修改原生 HTML 插值的定界符。

silent

- 类型: `Boolean`
- 默认值: `false`
- 用法:

JS

```
Vue.config.silent = true
```

取消 Vue.js 所有的日志与警告。

async

- 类型: `Boolean`
- 默认值: `true`
- 用法:

JS

```
Vue.config.async = false
```

如果关闭了异步模式，Vue 在检测到数据变化时同步更新 DOM。在有些情况下这有助于调试，但是也可能导致性能下降，并且影响 watcher 回调的调用顺序。**async: false** 不推荐用在生产环境中。

devtools

- 类型: **Boolean**
- 默认值: **true** (生产版为 **false**)
- 用法:

JS

```
// 在加载 Vue 之后立即同步的设置  
Vue.config.devtools = true
```

配置是否允许 **vue-devtools** 检查代码。开发版默认为 **true**，生产版默认为 **false**。生产版设为 **true** 可以启用检查。

全局 API

Vue.extend(options)

- 参数:
 - **{Object} options**
- 用法:

创建基础 Vue 构造器的“子类”。参数是一个对象，包含组件选项。

这里要注意的特例是 **el** 和 **data** 选项——在 **Vue.extend()** 中它们必须是函数。

HTML

```
<div id="mount-point"></div>
```

```
// 创建可复用的构造器
var Profile = Vue.extend({
  template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>'
})
// 创建一个 Profile 实例
var profile = new Profile({
  data: {
    firstName: 'Walter',
    lastName: 'White',
    alias: 'Heisenberg'
  }
})
// 挂载到元素上
profile.$mount('#mount-point')
```

结果:

HTML

```
<p>Walter White aka Heisenberg</p>
```

- 另见: 组件

Vue.nextTick(callback)

- 参数:
 - {Function} callback
- 用法:

延迟回调在下次 DOM 更新循环之后执行。在修改数据之后立即使用这个方法，等待 DOM 更新。

JS

```
// 修改数据
vm.msg = 'Hello'
// DOM 没有更新
Vue.nextTick(function () {
  // DOM 更新了
})
```

- 另见: 异步更新队列

Vue.set(object, key, value)

- 参数：
 - {Object} object
 - {String} key
 - {*} value
- 返回值： 设置的值
- 用法：

设置对象的属性。如果对象是响应的，将触发视图更新。这个方法主要用于解决不能检测到属性添加的限制。
- 另见： 深入响应式原理

Vue.delete(object, key)

- 参数：
 - {Object} object
 - {String} key
- 用法：

删除对象的属性。如果对象是响应的，将触发视图更新。这个方法主要用于解决不能检测到属性删除的限制。
- 另见： 深入响应式原理

Vue.directive(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 () {},
  update: function () {},
  unbind: function () {}
})
```

```
    })

    // 注册, 传入一个函数
    Vue.directive('my-directive', function () {
      // this will be called as `update`
    })

    // getter, 返回已注册的指令
    var myDirective = Vue.directive('my-directive')
```

- 另见: 自定义指令

Vue.elementDirective(id, [definition])

- 参数:
 - {String} id
 - {Object} [definition]

- 用法:

注册或获取全局的元素指令。

```
    // 注册
    Vue.elementDirective('my-element', {
      bind: function () {},
      // 没有使用 `update`
      unbind: function () {}
    })

    // getter, 返回已注册的元素指令
    var myDirective = Vue.elementDirective('my-element')
```

JS

- 另见: 元素指令

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
})

// 双向过滤器
Vue.filter('my-filter', {
  read: function () {},
  write: function () {}
})

// getter, 返回已注册的指令
var myFilter = Vue.filter('my-filter')
```

- 另见：自定义过滤器

Vue.component(id, [definition])

- 参数：
 - {String} id
 - {Function | Object} [definition]
- 用法：

注册或获取全局组件。

JS

```
// 注册组件，传入一个扩展的构造器
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')
```

- 另见：组件

Vue.transition(id, [hooks])

- 参数：
 - {String} id

- `{Object} [hooks]`
- 用法：
注册或获取全局的过渡钩子对象。

JS

```
// 注册
Vue.transition('fade', {
  enter: function () {},
  leave: function () {}
})

// 获取注册的钩子
var fadeTransition = Vue.transition('fade')
```

- 另见： [过渡](#)

`Vue.partial(id, [partial])`

- 参数：
 - `{String} id`
 - `{String} [partial]`
- 用法：

注册或获取全局的 `partial`。

JS

```
// 注册
Vue.partial('my-partial', '<div>Hi</div>')

// 获取注册的 partial
var myPartial = Vue.partial('my-partial')
```

- 另见： [特殊元素 - `<partial>`](#)

`Vue.use(plugin, [options])`

- 参数：
 - `{Object | Function} plugin`
 - `{Object} [options]`

- 用法:

安装 Vue.js 插件。如果插件是一个对象，必须有一个 `install` 方法。如果它是一个函数，它会被作为安装方法。安装方法以 `Vue` 为参数。

- 另见: 插件

Vue.mixin(mixin)

- 参数:

- `{Object} mixin`

- 用法:

全局应用一个混合，将影响所有 `Vue` 实例。插件作者可以用它向组件注入自定义逻辑。不推荐用在应用代码中。

- 另见: 全局混合

选项 / 数据

data

- 类型: `Object | Function`
- 限制: 在组件定义中只能是函数。
- 详细:

Vue 实例的数据对象。Vue.js 会递归地将它全部属性转为 `getter/setter`，从而让它能响应数据变化。这个对象必须是普通对象：原生对象，`getter/setter` 及原型属性会被忽略。不推荐观察复杂对象。

在实例创建之后，可以用 `vm.$data` 访问原始数据对象。Vue 实例也代理了数据对象所有的属性。

在定义组件时，同一定义将创建多个实例，此时 `data` 必须是一个函数，返回原始数据对象。如果 `data` 仍然是一个普通对象，则所有的实例将指向同一个对象！换成函数后，每当创建一个实例时，会调用这个函数，返回一个新的原始数据对象的副本。

名字以 `_` 或 `$` 开始的属性不会被 Vue 实例代理，因为它们可能与 Vue 的内置属性与 API 方法冲突。用 `vm.$data._property` 访问它们。

可以通过将 `vm.$data` 传入 `JSON.parse(JSON.stringify(...))` 得到原始数据对象。

- 示例:

JS

```
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

// 在 Vue.extend() 中必须是函数
var Component = Vue.extend({
  data: function () {
    return { a: 1 }
  }
})
```

- 另见: [深入响应式原理](#)

props

- 类型: `Array | Object`
- 详细:

包含一些特性——期望使用的父组件数据的属性。可以是数组或对象。对象用于高级配置，如类型检查，自定义验证，默认值等。

- 示例:

JS

```
// 简单语法
Vue.component('props-demo-simple', {
  props: ['size', 'myMessage']
})

// 对象语法, 指定验证要求
Vue.component('props-demo-advanced', {
  props: {
    // 只检测类型
    size: Number,
    // 检测类型 + 其它验证
    name: {
```

```
    type: String,  
    required: true,  
    // 双向绑定  
    twoWay: true  
  }  
}  
})
```

- 另见: **Props**

propsData

1.0.22+

- 类型: **Object**
- 限制: 只用于 **new** 创建实例中。
- 详细:

在创建实例的过程传递 **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'  
  }  
})
```

JS

computed

- 类型: **Object**
- 详细:

实例计算属性。**getter** 和 **setter** 的 **this** 自动地绑定到实例。

- 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
```

- 另见：
 - 计算属性
 - 深入响应式原理：计算属性的奥秘

methods

- 类型： `Object`
- 详细：

实例方法。实例可以直接访问这些方法，也可以用在指令表达式内。方法的 `this` 自动绑定到实例。

- 示例：

```
var vm = new Vue({
  data: { a: 1 },
  methods: {
    plus: function () {
      this.a++
    }
  }
})
```

```
vm.plus()  
vm.a // 2
```

- 另见： [方法与事件处理器](#)

watch

- 类型： `Object`

- 详细：

一个对象，键是观察表达式，值是对应回调。值也可以是方法名，或者是对象，包含选项。在实例化时为每个键调用 `$watch()` 。

- 示例：

JS

```
var vm = new Vue({  
  data: {  
    a: 1  
  },  
  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
```

- 另见： [实例方法 - vm.\\$watch](#)

选项 / DOM

el

- 类型: `String | HTMLElement | Function`
- 限制: 在组件定义中只能是函数。
- 详细:

为实例提供挂载元素。值可以是 CSS 选择符，或实际 HTML 元素，或返回 HTML 元素的函数。注意元素只用作挂载点。如果提供了模板则元素被替换，除非 `replace` 为 `false`。元素可以用 `vm.$el` 访问。

用在 `Vue.extend` 中必须是函数值，这样所有实例不会共享元素。

如果在初始化时指定了这个选项，实例将立即进入编译过程。否则，需要调用 `vm.$mount()`，手动开始编译。

- 另见: 生命周期图示

template

- 类型: `String`
- 详细:

实例模板。模板默认替换挂载元素。如果 `replace` 选项为 `false`，模板将插入挂载元素内。两种情况下，挂载元素的内容都将被忽略，除非模板有内容分发 `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。

注意在一些情况下，例如如模板包含多个顶级元素，或只包含普通文本，实例将变成一个片断实例，管理多个节点而不是一个节点。片断实例的挂载元素上的非流程控制指令被忽略。

- 另见:
 - 生命周期图示
 - 使用 `slot` 分发内容
 - 片断实例

replace

- 类型: `Boolean`
- 默认值: `true`
- 限制: 只能与 `template` 选项一起用

- 详细:

决定是否用模板替换挂载元素。如果设为 `true` (这是默认值), 模板将覆盖挂载元素, 并合并挂载元素和模板根节点的 `attributes`。如果设为 `false` 模板将覆盖挂载元素的内容, 不会替换挂载元素自身。

- 示例:

HTML

```
<div id="replace" class="foo"></div>
```

JS

```
new Vue({
  el: '#replace',
  template: '<p class="bar">replaced</p>'
})
```

结果:

HTML

```
<p class="foo bar" id="replace">replaced</p>
```

`replace` 设为 `false` :

HTML

```
<div id="insert" class="foo"></div>
```

JS

```
new Vue({
  el: '#insert',
  replace: false,
  template: '<p class="bar">inserted</p>'
})
```

结果:

HTML

```
<div id="insert" class="foo">
  <p class="bar">inserted</p>
</div>
```

选项 / 生命周期钩子

init

- 类型: `Function`
- 详细:

在实例开始初始化时同步调用。此时数据观测、事件和 `watcher` 都尚未初始化。

- 另见: 生命周期图示

created

- 类型: `Function`
- 详细:

在实例创建之后同步调用。此时实例已经结束解析选项，这意味着已建立：数据绑定，计算属性，方法，`watcher`/事件回调。但是还没有开始 DOM 编译，`$el` 还不存在。

- 另见: 生命周期图示

beforeCompile

- 类型: `Function`
- 详细:

在编译开始前调用。

- 另见: 生命周期图示

compiled

- 类型: `Function`
- 详细:

在编译结束后调用。此时所有的指令已生效，因而数据的变化将触发 DOM 更新。但是不担保 `$el` 已插入文档。

- 另见：生命周期图示

ready

- 类型： `Function`
- 详细：

在编译结束和 `$el` 第一次插入文档之后调用，如在第一次 `attached` 钩子之后调用。注意必须是由 `Vue` 插入（如 `vm.$appendTo()` 等方法或指令更新）才触发 `ready` 钩子。

- 另见：生命周期图示

attached

- 类型： `Function`
- 详细：

在 `vm.$el` 插入 DOM 时调用。必须是由指令或实例方法（如 `$appendTo()`）插入，直接操作 `vm.$el` 不会触发这个钩子。

detached

- 类型： `Function`
- 详细：

在 `vm.$el` 从 DOM 中删除时调用。必须是由指令或实例方法删除，直接操作 `vm.$el` 不会触发这个钩子。

beforeDestroy

- 类型： `Function`
- 详细：

在开始销毁实例时调用。此时实例仍然有功能。

- 另见：生命周期图示

destroyed

- 类型: `Function`
- 详细:

在实例被销毁之后调用。此时所有的绑定和实例的指令已经解绑，所有的子实例也已经被销毁。

如果有离开过渡，`destroyed` 钩子在过渡完成之后调用。

- 另见: 生命周期图示

选项 / 资源

directives

- 类型: `Object`
- 详细:
一个对象，包含指令。
- 另见:
 - 自定义指令
 - 资源命名约定

elementDirectives

- 类型: `Object`
- 详细:
一个对象，包含元素指令。
- 另见:
 - 元素指令
 - 资源命名约定

filters

- 类型: `Object`

- 详细：
一个对象，包含过滤器。
- 另见：
 - 自定义过滤器
 - 资源命名约定

components

- 类型： **Object**
- 详细：
一个对象，包含组件。
- 另见：
 - 组件

transitions

- 类型： **Object**
- 详细：
一个对象，包含过渡。
- 另见：
 - 过渡

partials

- 类型： **Object**
- 详细：
一个对象，包含 partial。
- 另见：
 - 特殊元素 - **partial**

parent

- 类型: `Vue` 实例
- 详细:

指定实例的父实例，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。

- 另见: 父子组件通信

events

- 类型: `Object`
- 详细:

一个对象，键是监听的事件，值是相应的回调。注意这些事件是 `Vue` 事件而不是 `DOM` 事件。值也可以是方法的名字。在实例化的过程中，`Vue` 实例会调用对象的每个键。

- 示例:

JS

```
var vm = new Vue({
  events: {
    'hook:created': function () {
      console.log('created!')
    },
    greeting: function (msg) {
      console.log(msg)
    },
    // 也可以是方法的名字
    bye: 'sayGoodbye'
  },
  methods: {
    sayGoodbye: function () {
      console.log('goodbye!')
    }
  }
}) // -> created!
vm.$emit('greeting', 'hi!') // -> hi!
vm.$emit('bye') // -> goodbye!
```

- 另见:
 - 实例方法 - 事件
 - 父子组件通信

mixins

- 类型: `Array`
- 详细:

一个数组，包含混合对象。这些混合对象可以像普通实例对象一样包含实例选项，它们将合并成一个最终选项对象，合并策略同 `Vue.extend()`。比如，如果混合对象包含一个 `created` 钩子，组件自身也包含一个，两个钩子函数都会被调用。

混合后的钩子按它们出现顺序调用，并且是在调用组件自己的钩子之前调用。

- 示例:

JS

```
var mixin = {
  created: function () { console.log(1) }
}
var vm = new Vue({
  created: function () { console.log(2) },
  mixins: [mixin]
})
// -> 1
// -> 2
```

- 另见: 混合

name

- 类型: `String`
- 限制: 只能用在 `Vue.extend()` 中。
- 详细:

允许组件在它的模板内递归地调用它自己。注意如果组件是由 `Vue.component()` 全局注册，全局 ID 自动作为它的名字。

指定 `name` 选项的另一个好处是方便检查。当在控制台检查组件时，默认的构造器名字是 `VueComponent`，不大有用。在向 `Vue.extend()` 传入 `name` 选项后，可以知道正在检查哪个组件。值会被转换为驼峰形式，并用作组件构造器的名字。

- 示例:

JS

```
var Ctor = Vue.extend({
  name: 'stack-overflow',
  template:
    '<div>' +
      // 递归地调用自己
      '<stack-overflow></stack-overflow>' +
    '</div>'
})

// 将导致错误: Maximum call stack size exceeded
// 不过我们假定没问题...
var vm = new Ctor()

console.log(vm) // -> StackOverflow {$el: null, ...}
```

extends

1.0.22+

- 类型: `Object | Function`
- 详细:

声明式的扩展另一个组件（可以是选项对象或者构造器），而不必使用 `Vue.extend`。主要作用是更容易的扩展单文件组件。

这类似于 `mixins`，不同的是组件的选项比待扩展的源组件的选项优先。

- 示例: *

```
var CompA = { ... }

// 扩展 CompA, 不用调用 Vue.extend
var CompB = {
  extends: CompA,
  ...
}
```

JS

实例属性

vm.\$data

- 类型: `Object`
- 详细:

Vue 实例观察的数据对象。可以用一个新的对象替换。实例代理了它的数据对象的属性。

vm.\$el

- 类型: `HTMLElement`
- 只读
- 详细:

Vue 实例的挂载元素。注意对于片段实例, `vm.$el` 返回一个锚节点, 指示片断的开始位置。

vm.\$options

- 类型: `Object`
- 只读
- 详细:

当前实例的初始化选项。在选项中包含自定义属性时有用处:

JS

```
new Vue({
  customOption: 'foo',
  created: function () {
    console.log(this.$options.customOption) // -> 'foo'
  }
})
```

vm.\$parent

- 类型: `Vue 实例`
- 只读
- 详细:

父实例，如果当前实例有的话。

vm.\$root

- 类型： `Vue 实例`
- 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，值将是它自身。

vm.\$children

- 类型： `Array<Vue instance>`
- 只读
- 详细：

当前实例的直接子组件。

vm.\$refs

- 类型： `Object`
- 只读
- 详细：

一个对象，包含注册有 `v-ref` 的子组件。

- 另见：
 - 子组件索引
 - `v-ref`

vm.\$els

- 类型： `Object`
- 只读
- 详细：

一个对象，包含注册有 `v-el` 的 DOM 元素。

- 另见：`v-el`。

实例方法 / 数据

`vm.$watch(expOrFn, callback, [options])`

- 参数：
 - `{String | Function} expOrFn`
 - `{Function} callback`
 - `{Object} [options]`
 - `{Boolean} deep`
 - `{Boolean} immediate`

- 返回值：`{Function} unwatch`

- 用法：

观察 Vue 实例的一个表达式或计算函数。回调的参数为新值和旧值。表达式可以是某个键路径或任意合法绑定表达式。

! 注意：在修改（不是替换）对象或数组时，旧值将与新值相同，因为它们索引同一个对象/数组。Vue 不会保留修改之前值的副本。

- 示例：

```
// 键路径
vm.$watch('a.b.c', function (newVal, oldVal) {
  // 做点什么
})

// 表达式
vm.$watch('a + b', function (newVal, oldVal) {
  // 做点什么
})

// 函数
vm.$watch(
  function () {
    return this.a + this.b
```

JS

```
    },  
    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：

JS

```
var unwatch = vm.$watch('a', cb)  
// 之后取消观察  
unwatch()
```

- **Option: 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。

JS

```
vm.$watch('someObject', callback, {  
  deep: true  
})  
vm.someObject.nestedValue = 123  
// 触发回调
```

- **Option: 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：

JS

```
vm.$watch('a', callback, {  
  immediate: true  
})  
// 立即以 `a` 的当前值触发回调
```

`vm.$get(expression)`

- 参数：
 - `{String} expression`
- 用法：

从 `Vue` 实例获取指定表达式的值。如果表达式抛出错误，则取消错误并返回 `undefined`。

- 示例:

JS

```
var vm = new Vue({
  data: {
    a: {
      b: 1
    }
  }
})
vm.$get('a.b') // -> 1
vm.$get('a.b + 1') // -> 2
```

vm.\$set(keypath, value)

- 参数:
 - {String} keypath
 - {*} value

- 用法:

设置 Vue 实例的属性值。多数情况下应当使用普通对象语法，如 `vm.a.b = 123`。这个方法只用于下面情况:

1. 使用 keypath 动态地设置属性。
2. 设置不存在的属性。

如果 keypath 不存在，将递归地创建并建立追踪。如果用它创建一个顶级属性，实例将被强制进入“digest 循环”，在此过程中重新计算所有的 watcher。

- 示例:

JS

```
var vm = new Vue({
  data: {
    a: {
      b: 1
    }
  }
})

// keypath 存在
vm.$set('a.b', 2)
vm.a.b // -> 2

// keypath 不存在
```

```
vm.$set('c', 3)  
vm.c // -> 3
```

- 另见：深入响应式原理

vm.\$delete(key)

- 参数：

- `{String} key`

- 用法：

删除 Vue 实例（以及它的 `$data`）上的顶级属性。强制 `digest` 循环，不推荐使用。

vm.\$eval(expression)

- 参数：

- `{String} expression`

- 用法：

计算当前实例上的合法的绑定表达式。表达式也可以包含过滤器。

- 示例：

JS

```
// 假定 vm.msg = 'hello'  
vm.$eval('msg | uppercase') // -> 'HELLO'
```

vm.\$interpolate(templateString)

- 参数：

- `{String} templateString`

- 用法：

计算模板，模板包含 Mustache 标签。注意这个方法只是简单计算插值，模板内的指令将被忽略。

- 示例：

```
// 假定 vm.msg = 'hello'  
vm.$interpolate('{{msg}} world!') // -> 'hello world!'
```

vm.\$log([keypath])

- 参数:
 - {String} [keypath]
- 用法:

打印当前实例的数据，比起一堆 getter/setter 要友好。keypath 可选。

JS

```
vm.$log() // 打印整个 ViewModel 的数据  
vm.$log('item') // 打印 vm.item
```

实例方法 / 事件

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.$emit` , `vm.$dispatch` 或 `vm.$broadcast` 触发。传入这些方法的附加参数都会传入这个方法的回调。

- 示例:

JS

```
vm.$on('test', function (msg) {  
  console.log(msg)  
})  
vm.$emit('test', 'hi')  
// -> "hi"
```

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，在第一次触发之后删除监听器。

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

- 参数：
 - {String} [event]
 - {Function} [callback]
- 用法：

删除事件监听器。

 - 如果没有参数，则删除所有的事件监听器；
 - 如果只提供了事件，则删除这个事件所有的监听器；
 - 如果同时提供了事件与回调，则只删除这个回调。

vm.\$emit(event, [...args])

- 参数：
 - {String} event
 - [...args]

触发当前实例上的事件。附加参数都会传给监听器回调。

vm.\$dispatch(event, [...args])

- 参数：
 - {String} event
 - [...args]
- 用法：

派发事件，首先在实例上触发它，然后沿着父链向上冒泡在触发一个监听器后停

止，除非它返回 `true` 。附加参数都会传给监听器回调。

- 示例:

JS

```
// 创建父链
var parent = new Vue()
var child1 = new Vue({ parent: parent })
var child2 = new Vue({ parent: child1 })

parent.$on('test', function () {
  console.log('parent notified')
})
child1.$on('test', function () {
  console.log('child1 notified')
})
child2.$on('test', function () {
  console.log('child2 notified')
})

child2.$dispatch('test')
// -> "child2 notified"
// -> "child1 notified"
// 没有通知 parent，因为 child1 的回调没有返回 true
```

- 另见：父子组件通信

vm.\$broadcast(event, [...args])

- 参数:
 - `{String} event`
 - `[...args]`

- 用法:

广播事件，通知给当前实例的全部后代。因为后代有多个枝杈，事件将沿着各“路径”通知。每条路径上的通知在触发一个监听器后停止，除非它返回 `true` 。

- 示例:

JS

```
var parent = new Vue()
// child1 和 child2 是兄弟
var child1 = new Vue({ parent: parent })
var child2 = new Vue({ parent: parent })
// child3 在 child2 内
var child3 = new Vue({ parent: child2 })
```

```
child1.$on('test', function () {
  console.log('child1 notified')
})
child2.$on('test', function () {
  console.log('child2 notified')
})
child3.$on('test', function () {
  console.log('child3 notified')
})

parent.$broadcast('test')
// -> "child1 notified"
// -> "child2 notified"
// 没有通知 child3, 因为 child2 的回调没有返回 true
```

实例方法 / DOM

vm.\$appendTo(elementOrSelector, [callback])

- 参数:
 - `{Element | String} elementOrSelector`
 - `{Function} [callback]`

• 返回值: `vm` ——实例自身

• 用法:

将实例的 DOM 元素或片断插入目标元素内。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.\$before(elementOrSelector, [callback])

- 参数:
 - `{Element | String} elementOrSelector`
 - `{Function} [callback]`

• 返回值: `vm` ——实例自身

• 用法:

将实例的 DOM 元素或片断插到目标元素的前面。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过

渡则立即执行。

vm.\$after(elementOrSelector, [callback])

- 参数：
 - `{Element | String} elementOrSelector`
 - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

将实例的 DOM 元素或片断插到目标元素的后面。第一个参数可以是一个元素或选择器字符串。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

vm.\$remove([callback])

- 参数：
 - `{Function} [callback]`
- 返回值： `vm` ——实例自身
- 用法：

从 DOM 中删除实例的 DOM 元素或片断。如果有过渡则触发过渡。回调在过渡完成后执行，如果没有触发过渡则立即执行。

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()
    })
  }
}
})
```

- 另见：
 - **Vue.nextTick**
 - 异步更新队列

实例方法 / 生命周期

vm.\$mount([elementOrSelector])

- 参数：
 - `{Element | String} [elementOrSelector]`
- 返回值： `vm` ——实例自身
- 用法：

如果 Vue 实例在实例化时没有收到 `e1` 选项，则它处于“未挂载”状态，没有关联的 DOM 元素或片断。可以使用 `vm.$mount()` 手动地开始挂载/编译未挂载的实例。

如果没有参数，模板将被创建为文档之外的片断，需要手工用其它的 DOM 实例方法把它插入文档中。如果 `replace` 选项为 `false`，则自动创建一个空 `<div>`，作为包装元素。

在已经挂载的实例上调用 `$mount()` 没有效果。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' })

// 手动挂载
new MyComponent().$mount().$appendTo('#container')
```

- 另见：生命周期图示

vm.\$destroy([remove])

- 参数：
 - `{Boolean} [remove] - default: false`

- 用法：

完全销毁实例。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，如果 `remove` 参数是 `true`，则从 DOM 中删除它关联的 DOM 元素或片断。

触发 `beforeDestroy` 和 `destroyed` 钩子。

- 另见：生命周期图示

指令

v-text

- 类型： `String`
- 详细：

更新元素的 `textContent`。

在内部，`{{ Mustache }}` 插值也被编译为 `TextNode` 的一个 `v-text` 指令。这个指令需要一个包装元素，不过性能稍好并且避免 FOUC (Flash of Uncompiled Content)。

- 示例:

```
<span v-text="msg"></span>
<!-- same as -->
<span>{{msg}}</span>
```

HTML

v-html

- 类型: `String`
- 详细:

更新元素的 `innerHTML` 。内容按普通 HTML 插入——数据绑定被忽略。如果想复用模板片断，应当使用 `partials` 。

在内部，`{{{ Mustache }}}` 插值也会被编译为锚节点上的一个 `v-html` 指令。这个指令需要一个包装元素，不过性能稍好并且避免 FOUC (Flash of Uncompiled Content) 。

! 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>
<!-- 相同 -->
<div>{{{html}}}</div>
```

HTML

v-if

- 类型: `*`
- 用法:

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。

- 另见: 条件渲染

v-show

- 类型: *
- 用法:

根据表达式的值的真假切换元素的 `display` CSS 属性，如果有过渡将触发它。

- 另见: [条件渲染 - v-show](#)

v-else

- 不需要表达式
- 限制: 前一兄弟元素必须有 `v-if` 或 `v-show` 。
- 用法:

为 `v-if` 和 `v-show` 添加“else 块”。

HTML

```
<div v-if="Math.random() > 0.5">
  Sorry
</div>
<div v-else>
  Not sorry
</div>
```

- 另见: [条件渲染 - v-else](#)
- 另见: [条件渲染 - 组件警告](#)

v-for

- 类型: `Array | Object | Number | String`
- **Param Attributes:**
 - `track-by`
 - `stagger`
 - `enter-stagger`
 - `leave-stagger`

- 用法:

基于源数据将元素或模板块重复数次。指令的值必须使用特定语法

`alias (in|of) expression`，为当前遍历的元素提供别名:

```
<div v-for="item in items">
  {{ item.text }}
</div>
```

1.0.17+ 支持 `of` 分隔符。

另外也可以为数组索引指定别名（如果值是对象可以为键指定别名）：

```
<div v-for="(index, item) in items"></div>
<div v-for="(key, val) in object"></div>
```

HTML

- 另见：列表渲染

v-on

- 缩写： `@`
- 类型： `Function | Inline Statement`
- 参数： `event (required)`
- 修饰符：
 - `.stop` - 调用 `event.stopPropagation()` 。
 - `.prevent` - 调用 `event.preventDefault()` 。
 - `.capture` - 添加事件侦听器时使用 `capture` 模式。
 - `.self` - 只当事件是从侦听器绑定的元素本身触发时才触发回调。
 - `.{keyCode | keyAlias}` - 只在指定按键上触发回调。

- 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)"` 。

1.0.11+ 在监听自定义事件时，内联语句可以访问一个 `$arguments` 属性，它是一个数组，包含传给子组件的 `$emit` 回调的参数。

- 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）：

```
HTML

<my-component @my-event="handleThis"></my-component>

<!-- 内联语句 -->
<my-component @my-event="handleThis(123, $arguments)"></my-component>
```

- 另见：方法与事件处理器

v-bind

- 缩写： `:`
- 类型： `*` (with argument) | `Object` (without argument)
- 参数： `attrOrProp` (optional)
- 修饰符：

- `.sync` - 双向绑定，只能用于 `prop` 绑定。
 - `.once` - 单次绑定，只能用于 `prop` 绑定。
 - `.camel` - 将绑定的特性名字转回驼峰命名。只能用于普通 HTML 特性的绑定，通常用于绑定用驼峰命名的 SVG 特性，比如 `viewBox`。
- 用法：

动态地绑定一个或多个 `attribute`，或一个组件 `prop` 到表达式。

在绑定 `class` 或 `style` 时，支持其它类型的值，如数组或对象。

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，可以绑定到一个对象。注意此时 `class` 和 `style` 绑定不支持数组和对象。

- 示例：

HTML

```
<!-- 绑定 attribute -->


<!-- 缩写 -->


<!-- 绑定 class -->
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>
<div :class="[classA, { classB: isB, classC: isC }]"></div>

<!-- 绑定 style -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- 绑定到一个对象 -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- prop 绑定, "prop" 必须在 my-component 组件内声明 -->
<my-component :prop="someThing"></my-component>

<!-- 双向 prop 绑定 -->
<my-component :prop.sync="someThing"></my-component>

<!-- 单次 prop 绑定 -->
<my-component :prop.once="someThing"></my-component>
```

- 另见：
 - **Class** 和 **Style** 绑定
 - 组件 **Props**

v-model

- 类型： 随表单控件类型不同而不同。
- 限制：
 - `<input>`
 - `<select>`
 - `<textarea>`
- **Param Attributes:**
 - **lazy**
 - **number**
 - **debounce**
- 用法：

在表单控件上创建双向绑定。
- 另见： 表单控件绑定

v-ref

- 不需要表达式
- 限制： 子组件
- 参数： **id (required)**
- 用法：

在父组件上注册一个子组件的索引，便于直接访问。不需要表达式。必须提供参数 `id`。可以通过父组件的 `$refs` 对象访问子组件。

在和 `v-for` 一起用时，注册的值将是一个数组，包含所有的子组件，对应于绑定数组。如果 `v-for` 用在一个对象上，注册的值将是一个对象，包含所有的子组件，对应于绑定对象。
- 注意：

因为 HTML 不区分大小写，camelCase 名字比如 `v-ref:someRef` 将全转为小写。可以用 `v-ref:some-ref` 设置 `this.$els.someRef` 。
- 示例：

```
<comp v-ref:child></comp>  
<comp v-ref:some-child></comp>
```

HTML

```
// 从父组件访问  
this.$refs.child  
this.$refs.someChild
```

使用 `v-for` :

```
<comp v-ref:list v-for="item in list"></comp>
```

HTML

```
// 值是一个数组  
this.$refs.list
```

JS

- 另见：子组件索引

v-el

- 不需要表达式
- 参数： `id` （必需）
- 用法：

为 DOM 元素注册一个索引，方便通过所属实例的 `$els` 访问这个元素。

- 注意：

因为 HTML 不区分大小写，camelCase 名字比如 `v-el:someEl` 将转为全小写。可以用 `v-el:some-el` 设置 `this.$els.someEl` 。

- 示例：

```
<span v-el:msg>hello</span>  
<span v-el:other-msg>world</span>
```

HTML

```
this.$els.msg.textContent // -> "hello"  
this.$els.otherMsg.textContent // -> "world"
```

JS

v-pre

- 不需要表达式
- 用法:

跳过编译这个元素和它的子元素。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:

HTML

```
<span v-pre>{{ this will not be compiled }}</span>
```

v-cloak

- 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>
```

`<div>` 不会显示，直到编译结束。

特殊元素

component

- 特性：
 - `is`
 - `keep-alive`
 - `transition-mode`

- 用法：

另一种调用组件的语法。主要是和 `is` 特性一起用于动态组件。

HTML

```
<!-- 动态组件 -->  
<!-- 由实例的 `componentId` 属性控制 -->  
<component :is="componentId"></component>
```

- 另见： 动态组件

slot

- 特性：
 - `name`

- 用法：

`<slot>` 元素作为组件模板之中的内容分发插槽。这个元素自身将被替换。

有 `name` 特性的 `slot` 称为具名 `slot`。有 `slot` 特性的内容将分发到名字相匹配的具名 `slot`。

- 另见： 使用 `slot` 分发内容

partial

- 特性：
 - `name`

- 用法：

`<partial>` 元素是已注册的 `partial` 的插槽，`partial` 在插入时被 Vue 编译。

`<partial>` 元素本身会被替换。`<partial>` 元素需要指定 `name` 特性。

- 示例：

```
// 注册 partial
Vue.partial('my-partial', '<p>This is a partial! {{msg}}</p>')
```

HTML

```
<!-- 静态 partial -->
<partial name="my-partial"></partial>

<!-- 动态 partial -->
<!-- 渲染 partial, id === vm.partialId -->
<partial v-bind:name="partialId"></partial>

<!-- 动态 partial, 使用 v-bind 缩写语法 -->
<partial :name="partialId"></partial>
```

过滤器

capitalize

- 示例:

HTML

```
{{ msg | capitalize }}
```

'abc' => *'Abc'*

uppercase

- 示例:

HTML

```
{{ msg | uppercase }}
```

'abc' => *'ABC'*

lowercase

- 示例:

```
{{ msg | lowercase }}
```

'ABC' => 'abc'

HTML

currency

- 参数:
 - `{String}` [货币符号] - 默认值: '\$'
 - **1.0.22+** `{Number}` [小数位] - 默认值: 2
- 示例:

```
{{ amount | currency }}
```

12345 => \$12,345.00

使用其它符号:

```
{{ amount | currency '£' }}
```

12345 => £12,345.00

一些货币使用 3 或 4 个小数位, 而一些货币不会, 例如日元 (¥)、越南盾 (₫):

```
{{ amount | currency '₫' 0 }}
```

12345 => ₫12,345

HTML

HTML

HTML

pluralize

- 参数:
 - `{String}` `single`, [`double`, `triple`, ...]

- 用法:

如果只有一个参数，复数形式只是简单地在末尾添加一个“s”。如果有多个参数，参数被当作一个字符串数组，对应一个、两个、三个...复数词。如果值的个数多于参数的个数，多出的使用最后一个参数。

- 示例:

HTML

```
{{count}} {{count | pluralize 'item'}}
```

1 => '1 item'

2 => '2 items'

HTML

```
{{date}} {{date | pluralize 'st' 'nd' 'rd' 'th'}}
```

结果:

1 => '1st'

2 => '2nd'

3 => '3rd'

4 => '4th'

5 => '5th'

json

- 参数:

- {Number} [indent] - 默认值: 2

- 用法:

输出经 `JSON.stringify()` 处理后的结果，而不是输出 `toString()` 的结果（如 `[object Object]`）。

- 示例:

以四个空格的缩进打印一个对象:

HTML

```
<pre>{{ nestedObject | json 4 }}</pre>
```

debounce

- 限制：指令的值须是函数，如 `v-on`
- 参数：
 - `{Number} [wait]` - 默认值： `300`
- 用法：

包装处理器，让它延迟执行 `x ms`，默认延迟 `300ms`。包装后的处理器在调用之后至少将延迟 `x ms`，如果在延迟结束前再次调用，延迟时长重置为 `x ms`。

- 示例：

HTML

```
<input @keyup="onKeyUp | debounce 500">
```

limitBy

- 限制：指令的值须是数组，如 `v-for`
- 参数：
 - `{Number} limit`
 - `{Number} [offset]`
- 用法：

限制数组为开始 `N` 个元素，`N` 由第一个参数指定。第二个参数是可选的，指定开始的偏移量。

HTML

```
<!-- 只显示开始 10 个元素 -->  
<div v-for="item in items | limitBy 10"></div>  
  
<!-- 显示第 5 到 15 元素-->  
<div v-for="item in items | limitBy 10 5"></div>
```

filterBy

- 限制：指令的值须是数组，如 `v-for`
- 参数：
 - `{String | Function} targetStringOrFunction`

- "in" (optional delimiter)
- {String} [...searchKeys]

- 用法:

返回过滤后的数组。第一个参数可以是字符串或函数。

如果第一个参数是字符串，则在每个数组元素中搜索它:

HTML

```
<div v-for="item in items | filterBy 'hello'">
```

在上例中，只显示包含字符串 "hello" 的元素。

如果 item 是一个对象，过滤器将递归地在它所有属性中搜索。为了缩小搜索范围，可以指定一个搜索字段:

HTML

```
<div v-for="user in users | filterBy 'Jack' in 'name'">
```

在上例中，过滤器只在用户对象的 name 属性中搜索 "Jack"。为了更好的性能，最好始终限制搜索范围。

上例使用静态参数，当然可以使用动态参数作为搜索目标或搜索字段。配合 v-model 我们可以轻松实现输入提示效果:

HTML

```
<div id="filter-by-example">
  <input v-model="name">
  <ul>
    <li v-for="user in users | filterBy name in 'name'">
      {{ user.name }}
    </li>
  </ul>
</div>
```

JS

```
new Vue({
  el: '#filter-by-example',
  data: {
    name: '',
    users: [
      { name: 'Bruce' },
      { name: 'Chuck' },
      { name: 'Jackie' }
    ]
  }
})
```

})

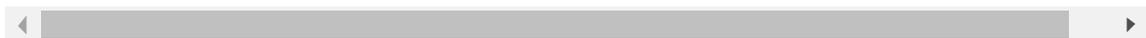
- Bruce
- Chuck
- Jackie

- 另一个示例:

多搜索字段:

HTML

```
<li v-for="user in users | filterBy searchText in 'name' 'phone'"></li>
```



多搜索字段为一个动态数组:

HTML

```
<!-- fields = ['fieldA', 'fieldB'] -->
<div v-for="user in users | filterBy searchText in fields">
```

使用自定义过滤函数:

HTML

```
<div v-for="user in users | filterBy myCustomFilterFunction">
```

orderBy

- 限制: 指令的值须是数组, 如 `v-for`
- 参数:
 - `{String | Array<String> | Function} ...sortKeys`
 - `{String} [order]` - 默认值: 1
- 用法:

返回排序后的数组。你可以传入多个键名。你也可以传入一个数组, 此数组包含

排序的键名。如果你想使用自己的排序策略，可以传入一个函数。可选参数 `order` 决定结果升序（ `order >= 0` ）或降序（ `order < 0` ）。

对于原始类型数组，可以忽略 `sortBy` ，只提供排序，例如 `orderBy 1` 。

- 示例：

按名字排序用户：

HTML

```
<ul>
  <li v-for="user in users | orderBy 'name'">
    {{ user.name }}
  </li>
</ul>
```

降序：

HTML

```
<ul>
  <li v-for="user in users | orderBy 'name' -1">
    {{ user.name }}
  </li>
</ul>
```

原始类型数组：

HTML

```
<ul>
  <li v-for="n in numbers | orderBy true">
    {{ n }}
  </li>
</ul>
```

动态排序：

HTML

```
<div id="orderby-example">
  <button @click="order = order * -1">Reverse Sort Order</button>
  <ul>
    <li v-for="user in users | orderBy 'name' order">
      {{ user.name }}
    </li>
  </ul>
</div>
```

```
new Vue({
  el: '#orderby-example',
  data: {
    order: 1,
    users: [{ name: 'Bruce' }, { name: 'Chuck' }, { name: 'Jackie' }]
  }
})
```

使用两个键名排序:

HTML

```
<ul>
  <li v-for="user in users | orderBy 'lastName' 'firstName'">
    {{ user.lastName }} {{ user.firstName }}
  </li>
</ul>
```

Reverse Sort Order

- Bruce
- Chuck
- Jackie

使用一个函数排序:

HTML

```
<div id="orderby-compare-example" class="demo">
  <button @click="order = order * -1">Reverse Sort Order</button>
  <ul>
    <li v-for="user in users | orderBy ageByTen order">
      {{ user.name }} - {{ user.age }}
    </li>
  </ul>
</div>
```

JS

```
new Vue({
  el: '#orderby-compare-example',
  data: {
    order: 1,
    users: [
```

```
{
  name: 'Jackie',
  age: 62
},
{
  name: 'Chuck',
  age: 76
},
{
  name: 'Bruce',
  age: 61
}
]
},
methods: {
  ageByTen: function (a, b) {
    return Math.floor(a.age / 10) - Math.floor(b.age / 10)
  }
}
})
```

Reverse Sort Order

- Jackie - 62
- Bruce - 61
- Chuck - 76

数组扩展方法

Vue.js 在 `Array.prototype` 上添加了两个方法，以方便常见的数组操作，并且能触发视图更新。

array.\$set(index, value)

- 参数：
 - `{Number} index`
 - `{*} value`

- 用法：

通过索引设置数组元素并触发视图更新。

```
vm.animals.$set(0, { name: 'Aardvark' })
```

- 另见： [数组检测问题](#)

array.\$remove(reference)

- 参数：
 - `{Reference} reference`

- 用法：

通过索引删除数组元素并触发视图更新。这个方法先在数组中搜索这个元素，如果找到了则调用 `array.splice(index, 1)`。

JS

```
var aardvark = vm.animals[0]  
vm.animals.$remove(aardvark)
```

- 另见： [变异方法](#)

发现错误？想参与编辑？在 [Github](#) 上编辑此页！