# 第1章 概述

Vue 构造函数

Vue 的构造函数是 Vue.js 的核心。它允许你创建 Vue 实例。创建一个 Vue 实例非常简单：

1

var vm = new Vue({ /\* options \*/ })

当你初始化一个 Vue 实例时，你需要传递一个选项对象。这个对象可以包括目标 DOM 元素，初始 data 对象，mixin 方法，生命周期钩子函数等内容。完整的选项列表详见 组件选项。

每个 Vue 实例本质上就是一个 ViewModel (因此在本文档中你会看到好多变量名叫 vm)。每个实例都有一个对应的 DOM 节点 vm.$el，它大致相当于 MVVM 中的 V。每个实例也有一个 JavaScript 数据对象 vm.$data，相对应的就是 MVVM 中的 M。改变 M 会触发 V 的更新。对于双向绑定，用户在 V 当中触发的事件会导致 M 的状态变化。Vue 实例上暴露的全部属性详见 实例属性。

每个 Vue 实例也有不少 实例方法，包括数据监视，事件通讯和 DOM 操作等等。

Vue 构造函数本身也暴露了 全局 API，这些 API 可以让你扩展 Vue 的子类，配置全局选项，注册全局组件、指令、过滤器这些自定义资源等等。

实例初始化

如果你在实例化中提供了 el 选项，Vue 实例将会立即进入编译阶段。否则，编译工作直到其 vm.$mount() 被调用时才会开始。在编译阶段，Vue 会递归遍历模板中的 DOM 节点并收集其中的指令，将其数据和这些指令所对应的 DOM 节点 “链接” 起来。一旦链接完毕，这些 DOM 节点就算是被 Vue 实例正式接管了。一个 DOM 节点只能被一个 Vue 实例管理，并且不能被多次编译。

数据代理

每一个 Vue 实例都会对其 $data 对象上的所有属性进行代理。假如你的数据对象上有个属性叫 msg，你可以通过 vm.$data.msg 来访问这个属性，也可以通过 vm.msg 来访问它。看起来有点神奇，但这完全是可选的，你可以选择只使用 vm.$data.msg 这种方式来访问。但是，不论你的偏好如何，都需要注意 vm 和 vm.$data 之间的区别，因为前者是一个 Vue 实例，和纯粹的数据对象还是有区别的。

还有一点值得注意的是数据对象 (data objects) 并不一定只能被一个 Vue 实例观察。我们完全可以让多个 vm 实例可以观察同一个数据对象。当多个组件需要共享一些全局状态的时候，这个特性就非常有用。

# 第2章 组件选项

数据

data

类型： Object | Function

限制： 在 Vue.extend() 中使用时只接受 Function.

Vue实例的数据对象. 可以通过 vm.$data 访问:

1

2

3

4

5

var data = { a: 1 }

var vm = new Vue({

data: data

})

vm.$data === data // -> true

Vue 实例会代理其数据对象上的所有属性，因此你可以直接在 Vue 实例上进行数据操作，这些变化会同步到数据对象里：

1

2

3

4

5

vm.a // -> 1

vm.a = 2

data.a // -> 2

data.a = 3

vm.a // -> 3

注意以 \_ 或 $ 开头的属性将不会被代理在 Vue 实例上，因为它们会和 Vue 的内部属性和 API 方法产生冲突。你只能通过 vm.$data.\_property 的方式访问它们。

数据对象必须是合法的 JSON 对象 (不能有循环引用)。使用时和普通对象并没有区别，JSON.stringify 序列化的结果也完全一样。同时，同一个数据对象可以被多个 Vue 实例共享。

这里有一个特殊的情况，就是在 Vue.extend() 中使用 data 选项时，由于我们不想让同一个数据对象被所有同一个构造函数所创建实例的共享，所以必须提供一个工厂函数来返回一个全新的对象:

1

2

3

4

5

6

7

8

9

10

var MyComponent = Vue.extend({

data: function () {

return {

message: 'some default data.',

object: {

fresh: true

}

}

}

})

在内部, Vue.js 会在被观测的数据对象上创建一个隐藏属性 \_\_ob\_\_， 然后通过递归遍历，将所有可枚举的属性转化为 getters 和 setters，从而实现依赖收集。

props

Type: Array | Object

props 代表了当前组件预期从父组件处获得的外部数据。props 选项有一个简单的基于数组的写法，也有一个可选的基于对象的写法。对象写法让你对每个 prop 进行单独的高级配置，比如类型检查、自定义验证和默认值等等。

示例:

1

2

3

4

5

6

7

Vue.component('param-demo', {

props: ['size', 'myMessage'], // 数组写法

compiled: function () {

console.log(this.size) // -> 100

console.log(this.myMessage) // -> 'hello!'

}

})

这里需要注意，因为 HTML 特性是大小写不敏感的，所以当一个 prop 作为特性出现在模板里的时候，你需要使用连字符格式：

1

2

<!-- myMessage 需要写成 my-message -->

<param-demo size="100" my-message="hello!"></param-demo>

基于对象的写法如下所示：

1

2

3

4

5

6

7

8

9

Vue.component('prop-validation-demo', {

props: {

size: Number,

name: {

type: String,

required: true

}

}

})

下面的组件用法会导致两个警告： “size” 的类型不匹配，以及缺少必需的 prop “name” 。

1

2

<prop-validation-demoo size="hello">

</prop-validation-demo>

更多关于数据传递的细节，请阅读教程中的以下章节：

传递回调 prop

prop 绑定类型

prop 验证规则

关于连字符格式

HTML 特性名大小写不敏感，所以我们在模板中通常使用连字符格式代替驼峰格式。在使用连字符格式的 props 特性的时候，需要注意一些特殊情况：

如果该特性是一个数据特性， data- 前缀会被自动去掉。比如 data-size 会被解析为 vm.size。

如果该特性仍然包含连字符 (-)，它会被驼峰化。比如 my-prop 会被解析为 vm.myProp。这是因为在模板里访问包含连字符的顶层属性不太方便：表达式 my-param 将会被解析为一个减法表达式，除非你使用尴尬的 this['my-param'] 写法。

methods

类型： Object

Methods 选项中包含的方法函数将会被混入到 Vue 实例中。你可以直接在 Vue 实例上访问这些方法，也可以在指令表达式里使用他们。 所有方法的 this 上下文都指向其所属的 Vue 实例。

示例：

1

2

3

4

5

6

7

8

9

10

var vm = new Vue({

data: { a: 1 },

methods: {

plus: function () {

this.a++

}

}

})

vm.plus()

vm.a // 2

computed

类型： Object

Computed 选项中所包含的内容将会被作为计算属性混入到 Vue 实例中。所有计算属性的 getter/setter 函数中 this 上下文都指向其所属的 Vue 实例。

示例：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

var vm = new Vue({

data: { a: 1 },

computed: {

// 只需要 getter 时，直接用一个函数即可

aDouble: function () {

return this.a \* 2

},

// 同时提供 getter 与 setter

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

DOM

el

类型： String | HTMLElement | Function

限制： 在 Vue.extend() 中使用时只接受 Function 类型。

为 Vue 实例提供一个作为挂载点的 DOM 元素。选项的值可以是一个 CSS 选择器，一个 HTML 元素，或是一个创建并返回 HTML 元素的函数。这个元素主要是起一个挂载点的作用：如果同时提供了模板选项，则该元素将被模板的内容替换 (除非 replace 选项为 false）。

在 Vue.extend() 中，此选项必须提供一个函数，以避免所有的实例共享同一个元素。

如果在初始化时提供了这个选项，实例将立即进入编译阶段。不然则需要手动调用 vm.$mount() 才能触发编译。

template

类型： String

此选项所提供的字符串模板会被用来生成 Vue 实例所管理的 HTML 内容。默认情况下，该模板会替代挂载的目标元素。当 replace 选项被设置为 false 时，模板则会被插入到挂载的元素内部。这两种情况下，挂载元素内部任何存在的原始内容都会被忽略，除非模板里包含了 内容插入点。

如果该字符串以 # 开头，它将会被当做 CSS 的 ID 选择器处理，并使用被选取元素的 innerHTML 作为字符串模板。利用这个功能，可以使用常见的 <script type="x-template"> 方式来引入模板。

注意，如果一个模板包含多个顶层节点，那么该实例将会成为一个 片段实例 - 也即没有单独根节点的实例。

Vue.js 使用基于 DOM 的模板编译机制。编译器遍历所有 DOM 元素，查找指令并创建数据绑定。这意味着所有的 Vue.js 模板都是可以解析的 HTML，HTML 被浏览器解析后生成 DOM。Vue.js 将字符串模板转为 DOM 片段并缓存，因此在创建新的 Vue 实例时，只需要直接复制这些片段即可。如果想让模板是完全符合标准的 HTML，你可以将指令前缀设置为 data-。

replace

类型： Boolean

默认值： true

限制： 需要和 template 选项协同使用

是否用模板替换正在被挂载的元素。如果设置为 false ，该模板将会覆盖元素内部的内容，而不是替代元素本身。

示例:

1

<div id="replace"></div>

1

2

3

4

new Vue({

el: '#replace',

template: '<p>replaced</p>'

})

编译结果：

1

<p>replaced</p>

相比之下, 当 replace 被设置成 false 时：

1

<div id="insert"></div>

1

2

3

4

5

new Vue({

el: '#insert',

replace: false,

template: '<p>inserted</p>'

})

编译结果：

1

2

3

<div id="insert">

<p>inserted</p>

</div>

生命周期

所有生命周期钩子函数的 this 上下文都指向其所属的 Vue 实例。除了调用选项中的钩子函数之外，Vue 实例也会触发对应的事件，事件的形式为 "hook:<hookName>"。以 created 钩子为例，将会触发 hook:created 事件。

created

类型： Function

在实例被创建的时候同步调用。在这个阶段，实例已经完成了包含以下内容的准备工作：数据观察，计算属性，方法，以及事件回调。但 DOM 编译还没开始，vm.$el 此时尚不可用。

beforeCompile

类型： Function

在编译开始之前调用。

compiled

类型： Function

在编译完成后调用。在这个阶段，所有的指令都已经完成绑定，数据变化会触发DOM更新。但此时尚不能保证 $el 已经被插入到DOM中。

ready

类型： Function

在编译完成后并且 $el 第一次插入文档时调用，也就是刚好在第一次 attached 钩子之后调用。注意只有通过指令或 Vue 实例方法，比如 $appendTo() 插入才会触发 ready。

attached

类型： Function

当 vm.$el 被一个指令或是 vm 实例方法（例如$appendTo()）插入到DOM里的时候调用。注意直接操作 vm.$el 不会触发这个事件。

detached

类型： Function

当 vm.$el 被一个指令或是 vm 实例方法从 DOM 里移除的时候调用。注意直接操作 vm.$el 不会触发这个事件。

beforeDestroy

类型： Function

在一个 Vue 实例被销毁之前调用。这个时候，实例的绑定和指令仍工作正常。

destroyed

类型： Function

在一个 Vue 实例被销毁之后调用。当这个钩子被调用时，该 Vue 实例的所有指令都已经被解除绑定，所有子实例也已经被销毁。

注意如果有一个离开过渡效果，destroyed 会在过渡效果结束之后才被调用。

资源

以下选项用于注册一个组件的私有资源。这些资源只能被该 Vue 实例及其子实例访问。所有的资源选项都应该是一个对象，键名即是该资源的 id，值则是对应的资源本身。

directives

类型： Object

需要注册的指令。更多细节请参考 自定义指令.

elementDirectives

类型： Object

需要注册的元素指令。更多细节请参考 元素指令。

filters

类型： Object

需要注册的过滤器。更多细节请参考 自定义过滤器.

components

类型： Object

需要注册的组件。更多细节请参考 组件系统.

transitions

类型： Object

需要注册的过渡效果。更多细节请参考 过渡效果。

partials

Type: Object

需要注册的模板片段。更多细节请参考 Partial。

其他

inherit

类型： Boolean

默认值: false

是否继承父组件的数据作用域. 当设置为 true 的时候你可以：

在当前组件模板里绑定父组件的数据属性；

通过原型继承直接访问父组件的属性。

需要注意，当使用 inherit: true 的时候，子实例也可以改变父实例的属性值，因为所有 Vue 实例的数据属性都是 getter/setters。

示例：

1

2

3

4

5

6

7

8

9

10

11

12

13

var parent = new Vue({

data: { a: 1 }

})

var child = parent.$addChild({

inherit: true,

data: { b: 2 }

})

child.a // -> 1

child.b // -> 2

// 下面这行将会修改 parent.a 的值，

// 而不是在子实例上创建一个新的属性!

child.a = 2

parent.a // -> 2

events

注册事件回调。该选项对象的键值是要注册的事件名，值就是相应的回调函数值。注意这里监听的是 Vue 的事件而不是 DOM 事件。值也可以是一个存在于当前组件上的方法名字符串。Vue实例会在初始化的时候对每一个事件/回调调用 vm.$on()。

示例：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

var vm = new Vue({

events: {

'hook:created': function () {

console.log('created!')

},

greeting: function (msg) {

console.log(msg)

},

// 也可以用方法名字符串

bye: 'sayGoodbye'

},

methods: {

sayGoodbye: function () {

console.log('goodbye!')

}

}

}) // -> created!

vm.$emit('greeting', 'hi!') // -> hi!

vm.$emit('bye') // -> goodbye!

watch

类型: Object

注册数据观察回调。该选项对象的键名是要监听的表达式，而值是对应的回调函数。值也可以是一个方法名字符串，或者是一个包含额外选项的对象。 Vue 实例将会在初始化的时候对该对象的每个条目调用 $watch() 。

示例：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

var vm = new Vue({

data: {

a: 1

},

watch: {

'a': function (val, oldVal) {

console.log('new: %s, old: %s', val, oldVal)

},

// 方法名字符串

'b': 'someMethod',

// 深度观察

'c': {

handler: function (val, oldVal) { /\* ... \*/ },

deep: true

}

}

})

vm.a = 2 // -> new: 2, old: 1

mixins

类型: Array

mixins 接受一个 mixin 对象数组，并会将每个 mixin 所包含的内容混入到当前组件中。这些混入对象可以像一般的实例对象一样包含实例选项，选项将被合并，合并逻辑同 Vue.extend()。举例来说，如果混入对象有一个 created 钩子，组件本身也有一个，则两个钩子都会被调用。

混入的钩子函数会被按照它们的出现顺序调用，并且会在宿主组件本身的钩子函数之前被调用。

示例：

1

2

3

4

5

6

7

8

9

var mixin = {

created: function () { console.log(1) }

}

var vm = new Vue({

created: function () { console.log(2) },

mixins: [mixin]

})

// -> 1

// -> 2

name

类型: String

限制: 仅在使用 Vue.extend() 的时候有效。

当在控制台里观察一个扩展过的 Vue 组件的时候，默认的构造函数名是 VueComponent ，并不是很有用。但你可以在使用 Vue.extend() 的时候添加一个 name 选项来自定义在控制台里输出的构造函数名。这个字符串会被驼峰化，然后作为组件的构造函数的名字使用。

示例：

1

2

3

4

5

var Ctor = Vue.extend({

name: 'cool-stuff'

})

var vm = new Ctor()

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

# 第3章 实例属性

vm.$el

Type: HTMLElement

只读

返回当前 Vue 实例正在管理的 DOM 元素。注意对于片段实例，vm.$el 指向的是一个代表片段起始位置的锚节点。

vm.$data

Type: Object

返回当前 Vue 实例正在监视的数据对象 (data object)。你可以用新的对象去替换它。Vue 实例会代理其 $data 对象上的所有属性。

vm.$options

Type: Object

返回当前 Vue 实例所使用的实例化选项。如果你想要调用自定义选项，就会需要用到这个属性:

1

2

3

4

5

6

new Vue({

customOption: 'foo',

created: function () {

console.log(this.$options.customOption) // -> 'foo'

}

})

vm.$parent

Type: Vue

只读

返回当前 vm 的父实例（如果存在的话）。

vm.$root

Type: Vue

只读

返回当前组件树的根 Vue 实例。如果当前实例已经没有父实例的话将会返回它自己。

vm.$children

Type: Array<Vue>

只读

返回当前实例的直接子实例数组。

vm.$

Type: Object

只读

返回一个对象，这个对象包含通过 v-ref 指令注册的子组件。更多细节请查看 v-ref。

vm.$$

Type: Object

Read only

返回一个对象，这个对象包含通过 v-el 指令注册的 DOM 元素。更多细节请查看 v-el。

元属性

被 v-repeat 创建出来的实例也会拥有一些元属性 (Meta properties)。例如 vm.$index, vm.$key 和 vm.$value 等。更多细节请查看教程中的 列表渲染 章节。

# 第4章 实例方法

数据

你可以在 Vue 实例中监视数据的变化。需要注意的是，所有的 watch 回调都是异步发生的，而且这些回调和指令一样被进行了异步批量处理。这就意味着如果一份数据在同一个事件循环 (event loop) 中多次变化，只会以最后一个值触发一次回调。

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

expOrFn String|Function

callback( newValue, oldValue ) Function

options Object optional

deep Boolean

immediate Boolean

sync Boolean

在当前 Vue 实例上监听一个表达式或是一个计算函数的值。

1

2

3

4

5

6

7

8

9

10

11

12

vm.$watch('a + b', function (newVal, oldVal) {

// do something

})

// or

vm.$watch(

function () {

return this.a + this.b

},

function (newVal, oldVal) {

// do something

}

)

如果要监视在对象中深层嵌套的属性变化，需要在 options 参数里传递 deep: true 来启用深度数据观察。注意在监听数组本身结构变化 (mutations) 的时候不需要此参数。

1

2

3

4

5

vm.$watch('someObject', callback, {

deep: true

})

vm.someObject.nestedValue = 123

// 回调被触发

如果在 options 参数里加上 immediate: true，那么回调会带着现在表达式的结果被立即触发。

1

2

3

4

vm.$watch('a', callback, {

immediate: true

})

// 回调使用现在表达式的结果 `a` 被立刻触发

最后，vm.$watch 会返回一个停止数据观察的句柄函数。

1

2

3

var unwatch = vm.$watch('a', cb)

// 停止观察

unwatch()

vm.$get( expression )

expression String

为 Vue 实例传递一个表达式来获得结果，如果表达式抛错，该错误会被截获并返回 undefined。

vm.$set( keypath, value )

keypath String

value \*

通过给 Vue 实例传递一个可用的路径来设置结果，如果路径不存在那么会被递归创建。

vm.$add( key, value )

key String

value \*

为 Vue 实例及其 $data 对象添加一个顶层属性 (root level property)。由于 ES5 的限制，Vue 无法侦测到对象中属性的增加或者删除，所以当你需要动态添加/删除属性的时候请使用此方法和 vm.$delete，但请谨慎使用，因为此方法会使得当前 vm 对所有 watcher 进行一次脏检查。

vm.$delete( key )

key String

在 Vue 实例及其 $data 对象中删除一个顶层属性。

vm.$eval( expression )

expression String

对一个表达式求值，该表达式可以包含过滤器。

1

2

// 假定 vm.msg = 'hello'

vm.$eval('msg | uppercase') // -> 'HELLO'

vm.$interpolate( templateString )

templateString String

对一段包含 Mustache 插值的模板字符串进行处理。注意：此方法仅仅处理字符串插值，属性指令不会被编译。

1

2

// 假定 vm.msg = 'hello'

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

vm.$log( [keypath] )

keypath String optional

以一个纯对象的方式打印当前实例的数据，比直接打印数据本身更容易在控制台里观察。接受一个可选路径参数。

1

2

vm.$log() // 打印整个 ViewModel 数据

vm.$log('item') // 只打印 vm.item

事件系统

每个 vm 都是一个事件触发器。当你有多层嵌套的组件时，你可以用事件系统去在它们之间进行通信。

vm.$dispatch( event, [args…] )

event String

args… optional

从当前的 vm 向上传递一个事件，一直到它的 $root 实例为止。如果传递过程中任意一个回调返回了 false ，那么事件会在该回调所属的实例处停止传播。

vm.$broadcast( event, [args…] )

event String

args… optional

向当前 vm 的所有子 vm 向下广播该事件。广播会进行深度遍历。如果传递过程中一个回调返回了 false，那么该回调的所属实例就不会继续向下广播这个事件。

vm.$emit( event, [args…] )

event String

args… optional

只在当前 vm 上触发一个事件。

vm.$on( event, callback )

event String

callback Function

在当前 vm 上监听一个事件。

vm.$once( event, callback )

event String

callback Function

在当前 vm 上为此事件绑定一个一次性的监听器。

vm.$off( [event, callback] )

event String optional

callback Function optional

如果没有传递参数，那么停止监视所有事件；如果传递了一个事件，那么移除该事件的所有回调；如果事件和回调都被传递，则只移除该回调。

DOM

vm 的 DOM 操作方法和 jQuery 中的同名方法类似，不同点在于 vm 实例的 DOM 操作方法能够触发 Vue.js 的过渡效果。 关于过渡效果，请参考 过渡效果。

vm.$appendTo( element|selector, [callback] )

element HTMLElement | selector String

callback Function optional

将 vm 的 $el 插入到目标元素的最后。参数可以是一个元素，或者是一个 querySelector 的选择器。

vm.$before( element|selector, [callback] )

element HTMLElement | selector String

callback Function optional

在目标元素之前插入 vm 的 $el 。

vm.$after( element|selector, [callback] )

element HTMLElement | selector String

callback Function optional

在目标元素之后插入 vm 的 $el 。

vm.$remove( [callback] )

callback Function optional

从 DOM 中移除 vm 的 $el 。

vm.$nextTick( callback )

callback Function

延迟回调的执行直到下一次 DOM 更新循环结束。当你改变一些数据之后调用它，当回调触发时 DOM 已经更新完毕。这和全局的 Vue.nextTick 效果相同，只是回调的 this 上下文会自动指向调用此方法的实例。

生命周期

vm.$mount( [element|selector] )

element HTMLElement | selector String optional

如果 Vue 实例在实例化的时候没有被传递一个 el 的选项,你可以自己调用 $mount() 来启动编译阶段。默认情况下，被挂载的元素将被实例的模板替代。假如 replace 选项被设置成 false，则模板将会被插入到挂载的元素中并且覆盖其内部的原始内容。除非模板包含 <content> 出口，否则原始内容会被丢弃。

如果没有提供参数，模板将会被生成为一个文档之外的元素，并且你需要亲自使用 DOM 实例方法把它插入到文档里。如果 replace 选项被设置成 false ，那么就会自动生成一个空的 <div> 作为封装元素。在已经被挂载的实例上调用 $mount() 上是没有作用的。这个方法会返回实例本身，所以如果你可以配合其他实例方法进行链式调用。

vm.$destroy( [remove] )

remove Boolean optional (默认值: false)

完全销毁一个 vm。清除它与其他 vm 的关联，解除所有指令的绑定。并可选地从DOM中移除它的 $el。同时，所有的 $on 和 $watch 监听器也都会被自动移除。

vm.$compile( element )

element HTMLElement

局部编译一块 DOM (元素或者是文档片段)。此方法返回一个 decompile 函数，用来清理编译过程中生成的指令。注意 decompile 函数不会移除被编译的 DOM。这个方法主要是为高级插件的编写者提供的。

vm.$addChild( [options, constructor] )

options Object optional

constructor Function optional

为当前的实例增加一个子实例。选项对象和手动实例化时所用的选项是相同的。额外还可以传递一个用 Vue.extend() 创造的构造函数。

实例之间的父子关系有三个含义：

父实例和子实例可以通过事件系统来通信。

子实例可以访问父实例的所有资源（比如：自定义指令）。但在严格模式下这一条会被禁止。

如果是继承了父实例作用域的子实例，也可以访问父实例的数据属性。

# 第5章 全局 API

Vue.config

Vue.config 是包含 Vue 全局设置的对象。下面列出了所有可设置变量及其默认值：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

{

// 启动调试模式。详见下文

debug: true,

// 启用严格模式，详见下文

strict: false,

// 指令的特性前缀

prefix: 'v-',

// 插入分隔符

// 对于插入HTML，添加一个额外的最外层字符

delimiters: ['{&#123;', '&#125;}'],

// 抑制 warnings？

silent: false,

// 插入 mustache 绑定?

interpolate: true,

// (对 directives 和 watchers) 使用异步更新？

async: true,

// 允许改变所观察的Array的原型链？

proto: true

}

你可以直接对其修改，例如：

1

Vue.config.debug = true // 开启调试模式

调试模式

当 Vue.config.debug 设置为 true 时，Vue 会

打印所有警告的栈的追踪记录

让所有 DOM 里的锚结点都采用注释结点。这会让审查元素结构变得更容易。

调试模式在压缩过的生产版本中是不可用的。

严格模式

默认情况下，Vue 组件从继承链（通过 Vue.extend() 创建）和它在视图里的父组件那里继承所有的资源。在严格模式下，组件只能继承从类继承的资源，不能从父视图层次继承。当启用了严格的模式时，资源应该是全局性的，或者是依赖于需要它们的组件。使用严格模式，能更好的封装组件和增加大型的项目的可重用性。

改变分隔符 (Delimiters)

如果设置了文本插值的分隔符，HTML 插值的分隔符也将改变。只需要用文本分隔符符最外面的字符再包裹一层：

1

2

3

Vue.config.delimiters = ['(%', '%)']

// (% %) 是文本插值的标签

// 则 ((% %)) 是 HTML 插值的标签

Vue.extend( options )

options Object

创建一个 Vue 的子类。所有的实例化选项都可以在此通用。其中需要特殊提及的是 el 和 data，在这里它们必须以函数形式出现。

直接用对象形式提供的组件在实例化之前 Vue 会在内部对其隐式调用 Vue.extend()。关于组件更多的细节，参见组件系统。

例子

1

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

// 创建可复用的 Profile 组件构造函数

var Profile = Vue.extend({

template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>'

})

// 创建一个 Profile 组件的实例

var profile = new Profile({

data: {

firstName : 'Walter',

lastName : 'White',

alias : 'Heisenberg'

}

})

// 挂载到元素上

profile.$mount('#mount-point')

结果：

1

<p>Walter White aka Heisenberg</p>

Vue.nextTick( callback )

callback Function

延迟执行回调到下一次 DOM 更新循环。在修改数据后调用它，等待 DOM 更新。详见 理解异步更新。

Vue.directive( id, [definition] )

id String

definition Function or Object optional

注册或取得一个全局自定义指令。详情见编写自定义指令。

Vue.elementDirective( id, [definition] )

id String

definition Function or Object optional

注册或取得一个全局自定义元素指令。详情见元素指令。

Vue.filter( id, [definition] )

id String

definition Function optional

注册或取得一个全局自定义过滤器。详情见编写自定义过滤器。

Vue.component( id, [definition] )

id String

definition Function Constructor or Object optional

注册或取得一个全局自定义组件。详情见组件系统。

Vue.transition( id, [definition] )

id String

definition Object optional

注册或取得一个全局 JavaScript 过渡效果定义。详见教程中JavaScript 过渡效果的相关章节。

Vue.partial( id, [definition] )

id String

definition String | Node optional

注册或获取全局的分模板（partial）。详见 Partial。

Vue.use( plugin, [args…] )

plugin Object or Function

args… optional

加载一个 Vue.js 的插件。如果该插件是一个对象，它必须有一个可调用的 install 方法。如果它本身就是一个函数，它会直接被作为安装函数来调用。Vue 会将其本身作为一个参数传递到安装函数里。详见插件。

Vue.util

公开内部的 util 模块。该模块包含一些常用的工具函数。此模块只是为了方便编写高级插件/指令，因此你必须查看源码来了解其中细节。

# 第6章 指令

响应式指令

指令可以将自己与一个 Vue 实例上的属性绑定，也可以与一个实例作用域中的表达式绑定。当属性或表达式的值发生改变时，指令的 update() 方法会在下一个事件循环中被异步调用。

v-text

更新元素的 textContent。

{{ Mustache }} 的插值也会被当做文字节点上的 v-text 指令进行编译。

v-html

更新元素的 innerHTML。

将 v-html 绑定到用户提供的数据上会有 XSS 的风险，因此使用 v-html 时应确保数据的安全性，或通过自定义过滤器将不被信任的 HTML 内容进行预处理。

v-show

可触发过渡效果。

根据绑定值将元素的 display CSS 属性设置为 none 或它的原始值。

v-class

接受一个可选的参数

如果没有提供参数，则将绑定值作为 CSS 类命字符串添加到元素的 classList 中。

如果提供了参数，则会以参数为 CSS 类名，并根据绑定值的真伪进行切换。可以配合多重从句使用：

1

2

3

4

5

<span v-class="

red : hasError,

bold : isImportant,

hidden : isHidden

"></span>

或者，你可以直接将该指令绑定到一个对象。对象的键值会作为类名被添加到元素的 classList，并根据对应值的真伪切换。

v-attr

需要一个参数

更新元素的指定特性（由参数指定）

示例:

1

<canvas v-attr="width:w, height:h"></canvas>

除 0 外的其他伪值将移除该特性。

或者，你可以直接将该指令绑定到一个对象。对象的键会被作为特性名称，根据对应的值的真伪进行切换。

在内部，普通特性中的 {{ Mustache }} 插值会被转换为 v-attr 指令进行编译。

从 0.12.9 开始，当 v-attr 直接用在 <input> 元素的 value 特性中时，实际设置的会是其 value 属性 (property) 而不是特性 (attribute)。例如，<input value="{{val}}"> 不会更新其特性值，而是会直接设置其属性值。

在为 <img> 元素设置 src 特性时，应该使用 v-attr 绑定而不是 mustache 模板绑定。浏览器会先于 Vue.js 对你的模板进行解析。所以当浏览器试图获取图片 URL 时，使用 mustache 模板绑定的数据会导致 404 错误。

v-style

接受一个可选参数

为元素添加内联 CSS 样式。

如果没有参数名，绑定值可以是一个字符串或是一个对象。

如果绑定值为字符串，则会将该元素的 style.cssText 属性设置为参数的值。

如果绑定值为对象，对象里的每一对键值都将被赋值到元素的 style 对象上。

示例:

1

<div v-style="myStyles"></div>

1

2

3

4

5

6

7

8

9

// myStyles 可以是字符串:

"color:red; font-weight:bold;"

// 也可以是对象:

{

color: 'red',

// CSS 规则用驼峰格式和连字符格式都可以

fontWeight: 'bold',

'font-size': '2em'

}

如果有参数的话，参数会被当作 CSS 规则名来用。可以配合多重从句添加多条规则：

示例:

1

2

3

4

5

<div v-style="

top: top + 'px',

left: left + 'px',

background-color: 'rgb(0,0,' + bg + ')'

"></div>

v-style 会智能地检测 CSS 规则是否需要浏览器前缀，所以你可以放心的使用无前缀版本的 CSS 规则。

1

2

<!-- 如果需要的话会自动使用 -webkit-transform -->

<div v-style="transform: 'scale(' + scale + ')'"></div>

由于 IE 的所有版本都会在解析 HTML 时将无法解析的内联样式删除，因此推荐统一使用 v-style 指令来代替 style 特性中的 {{mustache}} 插值。

v-on

需要一个参数。

绑定值应该是一个函数或者声明。

为元素添加一个 DOM 事件监听器。事件的类型由参数指定。这也是唯一可以和 key 过滤器一起使用的指令。详细请见事件监听。

v-model

只能用于 <input>, <select> 或 <textarea> 元素。

指令特性参数: lazy, number, options, debounce

在表单 input 元素上创建一个双向绑定。默认在 input 事件时同步数据。详情请见处理表单.

v-if

可触发过渡效果。

基于绑定值的真伪，插入或移除元素。如果元素为 <template> 元素，则它的内容将会被提取出来作为被插入或移除的片段。

示例:

1

2

3

4

<template v-if="test">

<p>hello</p>

<p>world</p>

</template>

渲染结果:

1

2

3

4

<!--v-if-start-->

<p>hello</p>

<p>world</p>

<!--v-if-end-->

v-repeat

会创建子 Vue 实例。

绑定值应为一个数组、对象或数字。

可触发过渡效果。

接受一个可选参数。

指令特性参数：track-by, stagger, enter-stagger, leave-stagger

为绑定的数组或对象中的每一项创建一个子 ViewModel 实例。如果值是一个数字，则会创建对应数量的子实例。当数组或对象的可变方法 (mutating method) 比如 push() 被调用，或是绑定的数字变化时，相应的子实例都会自动被创建或删除。

如果没有提供参数，子实例会直接使用数组内分配的元素作为其 $data。如果值不是一个对象，则会将其作为 $value 直接存放在实例上。

示例:

1

2

3

4

5

<ul>

<li v-repeat="users">

{{name}} {{email}}

</li>

</ul>

如果提供了参数，则总是会创建一个数据包装对象，用参数字符串作为键名来存放原本的数据对象。别名参数可以使得模板中的数据访问更加清晰：

1

2

3

4

5

<ul>

<li v-repeat="user : users">

{{user.name}} {{user.email}}

</li>

</ul>

0.12.8 引入了一种更自然的别名语法：

1

2

3

4

5

<ul>

<li v-repeat="user in users">

{{user.name}} {{user.email}}

</li>

</ul>

更多详细示例，请参看列表渲染.

字面指令

字面指令将它们的特性值视为纯字符串，不会视图建立数据绑定。它们只负责调用 bind()函数一次。字面指令的值也可以包含 mustache 表达式插值，具体的处理方式请参考动态字面指令。

v-transition

可以响应 Mustaches 插值

通知 Vue.js 为元素应用过渡效果。当一个可以触发过渡效果的指令将元素插入或移除时，或是当 Vue 实例中操作 DOM 的方法被调用时，对应的过渡效果就会被应用到该元素上。

详情请见过渡系统。

v-ref

在父组件上注册一个引用到子组件，便于访问只能和组件或是 v-repeat 协同使用。注册之后，父级的 $ 对象上可以访问注册的子实例。示例见子组件引用.

当该指令与 v-repeat 一起使用时，v-ref 注册的值将会是一个包含了所有子实例的数组，这个数组与 v-repeat 绑定的数组是相对应的。

如果 v-repeat 的源数据是一个对象，则 v-ref 注册的值会是一个对象，按照键值包含对应的子实例。

v-el

在 Vue 实例的 $$ 对象里注册一个 DOM 元素的引用。比如 <div v-el="hi"> 可以使用vm.$$.hi 获取。

空指令

空指令不需要参数，并且会忽略它的特性值。

v-pre

跳过编译此元素和此元素所有的子元素。跳过大量没有指令的节点可以加快编译速度。

v-cloak

直到关联的 ViewModel 结束编译之前本属性都会留在元素上。与 [v-cloak] { display: none } 类似的样式结合，这个指令可以用来在 ViewModel 准备好之前隐藏没有被编译的 {{ Mustache }} 模板。

# 第7章 过滤器

capitalize

‘abc’ => ‘Abc’

uppercase

‘abc’ => ‘ABC’

lowercase

‘ABC’ => ‘abc’

currency

接受一个可选参数

12345 => $12,345.00

你可以传递一个可选参数来表示货币符号（默认为 $）。

pluralize

要求至少一个参数

将值转为复数。如果只有一个参数，复数形式只是在末尾添加一个 “s”。如果有多个参数，参数列表对应于 1 个，2 个，3 个 … 。如果参数的个数不够，将使用最后一项。

示例：

1

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

1 => ‘1 item’

2 => ‘2 items’

1

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

结果为:

1 => ‘1st’

2 => ‘2nd’

3 => ‘3rd’

4 => ‘4th’

5 => ‘5th’

json

接受一个可选参数

使用 JSON.stringify() 来处理传入的值，而不是直接将其字符串化（如 [object Object])。接受一个可选参数来指定缩进级别（默认是 2）。

1

<pre>{{$data | json 4}}</pre>

key

只能和 v-on 指令协同使用

要求至少一个参数

包装事件处理器，只有当 keyCode 等于参数时才调用它。对于一些常用的键也可以用字符串形式的别名：

enter

tab

delete

esc

up

down

left

right

space

示例：

1

<input v-on="keyup:doSomething | key 'enter'">

只有按回车键 (enter) 时才会调用 doSomething。

debounce (防反跳)

只能和 v-on 指令协同使用

可以附带一个可选参数

将响应函数包装为 X 毫秒之后防反跳，X 就是这个参数，默认值是 300ms。一个防反跳的响应函数会延迟到这个调用之后 X 毫秒执行，如果响应函数在延迟执行之前再次被调用，则延迟时间会被重置为 X 毫秒。

filterBy

语法： filterBy searchKey [in dataKey...].

只能和数组协同使用

返回原数组过滤后的结果。searchKey 参数是当前 ViewModel 的一个属性名。这个属性的值会被用作查找的目标：

1

2

3

4

<input v-model="searchText">

<ul>

<li v-repeat="users | filterBy searchText">{{name}}</li>

</ul>

当使用此过滤器时，将递归遍历 users 数组的每一个元素来寻找 searchText 的当前值从而过滤该数组。举例来说，如果一个条目是 { name: 'Jack', phone: '555-123-4567' }，而 searchText 的值是 '555'，这个条目就被认为是符合条件。

你也可以通过可选的 in dataKey 参数来指定具体要在哪个属性中进行查找：

1

2

3

4

<input v-model="searchText">

<ul>

<li v-repeat="user in users | filterBy searchText in 'name'">{{name}}</li>

</ul>

现在只有当 name 属性包含 searchText 时这个条目才符合条件。所以当 searchText 的值是 '555' 时这个条目将不符合条件，而当值是 'Jack' 时则符合。

0.12.11 更新

从 0.12.11 开始你可以传入多个数据的键名：

1

<li v-repeat="user in uers | filterBy searchText in 'name' 'phone'"></li>

或者以数组形式传入一个动态的参数：

1

2

<!-- fields = ['fieldA', 'fieldB'] -->

<div v-repeat="user in users | filterBy searchText in fields">

或者直接传入一个自定义过滤函数：

1

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

orderBy

语法： orderBy sortKey [reverseKey].

只能和数组协同使用

返回原数组排序后的结果。sortKey 参数是当前 ViewModel 的一个属性名。这个属性的值表示用来排序的键名。可选的 reverseKey 参数也是当前 ViewModel 的一个属性名，如果这个属性值为真则数组会被倒序排列。

1

2

3

<ul>

<li v-repeat="user in users | orderBy field reverse">{{name}}</li>

</ul>

1

2

3

4

5

6

7

new Vue({

/\* ... \*/

data: {

field: 'name',

reverse: false

}

})

你可以使用引号来表示字面量的排序键名。使用 -1 来表示字面量的 reverse 参数。

1

2

3

<ul>

<li v-repeat="user in users | orderBy 'name' -1">{{name}}</li>

</ul>

# 第8章 特殊元素

component

使用组件的另一种语法。主要配合 is 特性用于动态化的组件：

1

2

<!-- 动态组件，由 vm 的 `componentId` 属性控制 -->

<component is="{{componentId}}"></component>

content

<content> 标签用于作为组件模板的内容插入点，而 <content> 元素本身会被真正的内容所替换。它接受一个 select 的可选特性，该特性应该是一个合法的 CSS 选择器，这样只有匹配这一选择器的被插入内容的子集才会被展示：

1

2

<!-- 只显示插入内容中的 <li> -->

<content select="li"></content>

其实 select 特性还可以包含 mustache 表达式。更多细节参见内容插入。

partial

<partial> 用作注册的模板片段的插入点。片段内容在插入时也会被 Vue 编译，同时 <partial> 元素本身会被替换掉。它需要提供一个 name 特性。比如：

1

2

// 注册模板片段

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

1

2

3

4

5

<!-- 静态模板片段 -->

<partial name="my-partial"></partial>

<!-- 动态模板片段 -->

<partial name="{{partialId}}"></partial>