Vue.js

渐进式JavaScript 框架

官方文档同步版

ALEX 搬运

目　录

写在前面

这是什么？

这是cn.vuejs.org 官网文档的同步版本

鉴于官网那个难以忍受的访问速度及不是所有人都随时随地有环境，所以搬运一个看云的版本，可以适用于kindle等阅读设备的童鞋

更新日志

本文档最后更新时间：2018-10-30

对应vuejs最新稳定版本：2.5.16 (Vuejs每个版本的更新日志见 GitHub)。

搬运工

Alex [xux851@gmail.com]

https://www.ysido.com

感谢

官方翻译团队 https://github.com/vuejs/cn.vuejs.org/

基础

安装

兼容性

Vue 不支持 IE8 及以下版本，因为 Vue 使用了 IE8 无法模拟的 ECMAScript 5 特性。但它支持所有兼容 ECMAScript 5 的浏览器。

更新日志

最新稳定版本：2.5.16

每个版本的更新日志见 GitHub。

Vue Devtools

在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

直接用 `<script>` 引入

直接下载并用 `<script>` 标签引入，`Vue` 会被注册为一个全局变量。

在开发环境下不要使用压缩版本，不然你就失去了所有常见错误相关的警告!

开发版本包含完整的警告和调试模式

生产版本删除了警告，30.90KB min+gzip

CDN

我们推荐链接到一个你可以手动更新的指定版本号：

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.16/dist/vue.js"></script>
```

你可以在 cdn.jsdelivr.net/npm/vue 浏览 NPM 包的源代码。

Vue 也可以在 unpkg 和 cdnjs 上获取 (cdnjs 的版本更新可能略滞后)。

请确认了解不同构建版本并在你发布的站点中使用生产环境版本，把 `vue.js` 换成 `vue.min.js` 。这是一个更小的构建，可以带来比开发环境下更快的速度体验。

NPM

在用 Vue 构建大型应用时推荐使用 NPM 安装[1]。NPM 能很好地和诸如 webpack 或 Browserify 模块打包器配合使用。同时 Vue 也提供配套工具来开发单文件组件。

```
# 最新稳定版
$ npm install vue
```

命令行工具 (CLI)

Vue 提供了一个官方的 CLI，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。更多详情可查阅 Vue CLI 的文档。
CLI 工具假定用户对 Node.js 和相关构建工具有一定程度的了解。如果你是新手，我们强烈建议先在不用构建工具的情况下通读指南，在熟悉 Vue 本身之后再使用 CLI。

对不同构建版本的解释

在 NPM 包的 `dist/` 目录你将会找到很多不同的 Vue.js 构建版本。这里列出了它们之间的差别：

	UMD	CommonJS	ES Module
完整版	vue.js	vue.common.js	vue.esm.js
只包含运行时版	vue.runtime.js	vue.runtime.common.js	vue.runtime.esm.js
完整版 (生产环境)	vue.min.js	-	-
只包含运行时版 (生产环境)	vue.runtime.min.js	-	-

术语

- 完整版：同时包含编译器和运行时的版本。
- 编译器：用来将模板字符串编译成为 JavaScript 渲染函数的代码。
- 运行时：用来创建 Vue 实例、渲染并处理虚拟 DOM 等的代码。基本上就是除去编译器的其它一切。
- UMD：UMD 版本可以通过 `<script>` 标签直接用在浏览器中。jsDelivr CDN 的 https://cdn.jsdelivr.net/npm/vue 默认文件就是运行时 + 编译器的 UMD 版本 (`vue.js`)。
- CommonJS：CommonJS 版本用来配合老的打包工具比如 Browserify 或 webpack 1。这些打包工具的默认文件 (`pkg.main`) 是只包含运行时的 CommonJS 版本 (`vue.runtime.common.js`)。
- ES Module：ES module 版本用来配合现代打包工具比如 webpack 2 或 Rollup。这些打包工具的默认文件 (`pkg.module`) 是只包含运行时的 ES Module 版本 (`vue.runtime.esm.js`)。

运行时 + 编译器 vs. 只包含运行时

如果你需要在客户端编译模板 (比如传入一个字符串给 `template` 选项，或挂载到一个元素上并以其 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

```
// 需要编译器
new Vue({
  template: '<div>{{ hi }}</div>'
})
```

```
// 不需要编译器
new Vue({
  render (h) {
    return h('div', this.hi)
  }
})
```

当使用 `vue-loader` 或 `vueify` 的时候，`*.vue` 文件内部的模板会在构建时预编译成 JavaScript。你在最终打好的包里实际上是不需要编译器的，所以只用运行时版本即可。

因为运行时版本相比完整版体积要小大约 30%，所以应该尽可能使用这个版本。如果你仍然希望使用完整版，则需要在打包工具里配置一个别名：

webpack

```
module.exports = {
  // ...
  resolve: {
    alias: {
      'vue$': 'vue/dist/vue.esm.js' // 用 webpack 1 时需用 'vue/dist/vue.common.
js'
    }
  }
}
```

Rollup

```
const alias = require('rollup-plugin-alias')

rollup({
  // ...
  plugins: [
    alias({
      'vue': 'vue/dist/vue.esm.js'
    })
  ]
})
```

Browserify

添加到你项目的 `package.json`：

```
{
  // ...
  "browser": {
    "vue": "vue/dist/vue.common.js"
```

```
  }
}
```

Parcel

在你项目的 `package.json` 中添加：

```
{
  // ...
  "alias": {
    "vue" : "./node_modules/vue/dist/vue.common.js"
  }
}
```

开发环境 vs. 生产环境模式

对于 UMD 版本来说，开发环境/生产环境模式是硬编码好的：开发环境下用未压缩的代码，生产环境下使用压缩后的代码。

CommonJS 和 ES Module 版本是用于打包工具的，因此我们不提供压缩后的版本。你需要自行将最终的包进行压缩。

CommonJS 和 ES Module 版本同时保留原始的 `process.env.NODE_ENV` 检测，以决定它们应该运行在什么模式下。你应该使用适当的打包工具配置来替换这些环境变量以便控制 Vue 所运行的模式。把 `process.env.NODE_ENV` 替换为字符串字面量同时可以让 UglifyJS 之类的压缩工具完全丢掉仅供开发环境的代码块，以减少最终的文件尺寸。

webpack

在 webpack 4+ 中，你可以使用 `mode` 选项：

```
module.exports = {
  mode: 'production'
}
```

但是在 webpack 3 及其更低版本中，你需要使用 DefinePlugin：

```
var webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: JSON.stringify('production')
```

```
      }
    })
  ]
}
```

Rollup

使用 rollup-plugin-replace：

```
const replace = require('rollup-plugin-replace')

rollup({
  // ...
  plugins: [
    replace({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
}).then(...)
```

Browserify

为你的包应用一次全局的 envify 转换。

```
NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js
```

也可以移步生产环境部署。

CSP 环境

有些环境，如 Google Chrome Apps，会强制应用内容安全策略 (CSP)，不能使用 `new Function()` 对表达式求值。这时可以用 CSP 兼容版本。完整版本依赖于该功能来编译模板，所以无法在这些环境下使用。

另一方面，运行时版本则是完全兼容 CSP 的。当通过 webpack + vue-loader 或者 Browserify + vueify 构建时，模板将被预编译为 `render` 函数，可以在 CSP 环境中完美运行。

开发版本

重要: GitHub 仓库的 `/dist` 文件夹只有在新版本发布时才会提交。如果想要使用 GitHub 上 Vue 最新的源码，你需要自己构建！

```
git clone https://github.com/vuejs/vue.git node_modules/vue
cd node_modules/vue
npm install
npm run build
```

Bower

Bower 只提供 UMD 版本。

```
# 最新稳定版本
$ bower install vue
```

AMD 模块加载器

所有 UMD 版本都可以直接用作 AMD 模块。

译者注 [1] 对于中国大陆用户，建议将 NPM 源设置为[国内的镜像](https://npm.taobao.org/)，可以大幅提升安装速度。

Vue实例

创建一个 Vue 实例

每个 Vue 应用都是通过用 `Vue` 函数创建一个新的 Vue 实例开始的：

```
var vm = new Vue({
  // 选项
})
```

虽然没有完全遵循 MVVM 模型，但是 Vue 的设计也受到了它的启发。因此在文档中经常会使用 `vm` (ViewModel 的缩写) 这个变量名表示 Vue 实例。

当创建一个 Vue 实例时，你可以传入一个选项对象。这篇教程主要描述的就是如何使用这些选项来创建你想要的行为。作为参考，你也可以在 API 文档 中浏览完整的选项列表。

一个 Vue 应用由一个通过 `new Vue` 创建的根 Vue 实例，以及可选的嵌套的、可复用的组件树组成。举个例子，一个 todo 应用的组件树可以是这样的：

```
根实例
└─ TodoList
   ├─ TodoItem
   │  ├─ DeleteTodoButton
   │  └─ EditTodoButton
   └─ TodoListFooter
      ├─ ClearTodosButton
      └─ TodoListStatistics
```

我们会在稍后的组件系统章节具体展开。不过现在，你只需要明白所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象 (一些根实例特有的选项除外)。

数据与方法

当一个 Vue 实例被创建时，它向 Vue 的响应式系统中加入了其 `data` 对象中能找到的所有的属性。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。

```
// 我们的数据对象
var data = { a: 1 }

// 该对象被加入到一个 Vue 实例中
var vm = new Vue({
  data: data
})
```

```
// 获得这个实例上的属性
// 返回源数据中对应的字段
vm.a == data.a // => true

// 设置属性也会影响到原始数据
vm.a = 2
data.a // => 2

// ……反之亦然
data.a = 3
vm.a // => 3
```

当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时 `data` 中存在的属性才是响应式的。也就是说如果你添加一个新的属性，比如：

```
vm.b = 'hi'
```

那么对 `b` 的改动将不会触发任何视图的更新。如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。比如：

```
data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos: false,
  todos: [],
  error: null
}
```

这里唯一的例外是使用 `Object.freeze()` ，这会阻止修改现有的属性，也意味着响应系统无法再追踪变化。

```
var obj = {
  foo: 'bar'
}

Object.freeze(obj)

new Vue({
  el: '#app',
  data: obj
})
```

```
<div id="app">
  <p>{{ foo }}</p>
```

```
  <!-- 这里的 `foo` 不会更新！ -->
  <button v-on:click="foo = 'baz'">Change it</button>
</div>
```

除了数据属性，Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 `$` ，以便与用户定义的属性区分开来。例如：

```
var data = { a: 1 }
var vm = new Vue({
  el: '#example',
  data: data
})

vm.$data === data // => true
vm.$el === document.getElementById('example') // => true

// $watch 是一个实例方法
vm.$watch('a', function (newValue, oldValue) {
  // 这个回调将在 `vm.a` 改变后调用
})
```

以后你可以在 API 参考中查阅到完整的实例属性和方法的列表。

实例生命周期钩子

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 `created` 钩子可以用来在一个实例被创建之后执行代码：

```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` 指向 vm 实例
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 `mounted` 、 `updated` 和 `destroyed` 。生命周期钩子的 `this` 上下文指向调用它的 Vue 实例。

不要在选项属性或回调上使用[箭头函数](https://developer.mozilla.org/zh-

CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions)，比如 `created: () => console.log(this.a)` 或 `vm.$watch('a', newValue => this.myMethod())`。因为箭头函数是和父级上下文绑定在一起的，`this` 不会是如你所预期的 Vue 实例，经常导致 `Uncaught TypeError: Cannot read property of undefined` 或 `Uncaught TypeError: this.myMethod is not a function` 之类的错误。

生命周期图示

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。

模板语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML ，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，直接写渲染 (render) 函数，使用可选的 JSX 语法。

插值

文本

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

```
<span>Message: {{ msg }}</span>
```

Mustache 标签将会被替代为对应数据对象上 `msg` 属性的值。无论何时，绑定的数据对象上 `msg` 属性发生了改变，插值处的内容都会更新。

通过使用 v-once 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

```
<span v-once>这个将不会改变: {{ msg }}</span>
```

原始 HTML

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 `v-html` 指令：

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

{% raw %}

Using mustaches: {{ rawHtml }}

Using v-html directive:

{% endraw %}

这个 `span` 的内容将会被替换成为属性值 `rawHtml` ，直接作为 HTML——会忽略解析属性值中的数据绑定。注意，你不能使用 `v-html` 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致 [XSS 攻击](https://en.wikipedia.org/wiki/Cross-site_scripting)。请只对可信内容使用 HTML 插值，**绝不要**对用户提供的内容使用插值。

特性

Mustache 语法不能作用在 HTML 特性上，遇到这种情况应该使用 v-bind 指令：

```
<div v-bind:id="dynamicId"></div>
```

在布尔特性的情况下，它们的存在即暗示为 `true` ， `v-bind` 工作起来略有不同，在这个例子中：

```
<button v-bind:disabled="isButtonDisabled">Button</button>
```

如果 `isButtonDisabled` 的值是 `null` 、 `undefined` 或 `false` ，则 `disabled` 特性甚至不会被包含在渲染出来的 `<button>` 元素中。

使用 JavaScript 表达式

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>
```

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

```
<!-- 这是语句，不是表达式 -->
{{ var a = 1 }}

<!-- 流控制也不会生效，请使用三元表达式 -->
{{ if (ok) { return message } }}
```

模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 `Math` 和 `Date` 。你不应该在模板表达式中试图访问用户定义的全局变量。

指令

指令 (Directives) 是带有 `v-` 前缀的特殊特性。指令特性的值预期是单个 JavaScript 表达式 (`v-for` 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。回顾我们在介绍中看到的例子：

```
<p v-if="seen">现在你看到我了</p>
```

这里， `v-if` 指令将根据表达式 `seen` 的值的真假来插入/移除 `<p>` 元素。

参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如， `v-bind` 指令可以用于响应式地更新 HTML 特性：

```
<a v-bind:href="url">...</a>
```

在这里 `href` 是参数，告知 `v-bind` 指令将该元素的 `href` 特性与表达式 `url` 的值绑定。
另一个例子是 `v-on` 指令，它用于监听 DOM 事件：

```
<a v-on:click="doSomething">...</a>
```

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

修饰符

修饰符 (Modifiers) 是以半角句号 `.` 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如， `.prevent` 修饰符告诉 `v-on` 指令对于触发的事件调用 `event.preventDefault()` ：

```
<form v-on:submit.prevent="onSubmit">...</form>
```

在接下来对 `v-on` 和 `v-for` 等功能的探索中，你会看到修饰符的其它例子。

缩写

`v-` 前缀作为一种视觉提示，用来识别模板中 Vue 特定的特性。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时， `v-` 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue.js 管理所有模板的单页面应用程序 (SPA - single page application) 时， `v-` 前缀也变得没那么重要了。因此，Vue.js 为 `v-bind` 和 `v-on` 这两个最常用的指令，提供了特定简写：

`v-bind` 缩写

```
<!-- 完整语法 -->
<a v-bind:href="url">...</a>

<!-- 缩写 -->
<a :href="url">...</a>
```

`v-on` 缩写

```
<!-- 完整语法 -->
<a v-on:click="doSomething">...</a>

<!-- 缩写 -->
<a @click="doSomething">...</a>
```

它们看起来可能与普通的 HTML 略有不同，但 `:` 与 `@` 对于特性名来说都是合法字符，在所有支持 Vue.js 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

计算属性和侦听器

计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。例如：

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>
```

在这个地方，模板不再是简单的声明式逻辑。你必须看一段时间才能意识到，这里是想要显示变量 `message` 的翻转字符串。当你想要在模板中多次引用此处的翻转字符串时，就会更加难以处理。

所以，对于任何复杂逻辑，你都应当使用计算属性。

基础例子

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})
```

结果：

{% raw %}

Original message: "{{ message }}"

Computed reversed message: "{{ reversedMessage }}"

{% endraw %}

这里我们声明了一个计算属性 `reversedMessage` 。我们提供的函数将用作属性 `vm.reversedMessage` 的 getter 函数：

```
console.log(vm.reversedMessage) // => 'olleH'
vm.message = 'Goodbye'
console.log(vm.reversedMessage) // => 'eybdooG'
```

你可以打开浏览器的控制台，自行修改例子中的 vm。 `vm.reversedMessage` 的值始终取决于 `vm.message` 的值。

你可以像绑定普通属性一样在模板中绑定计算属性。Vue 知道 `vm.reversedMessage` 依赖于 `vm.message` ，因此当 `vm.message` 发生改变时，所有依赖 `vm.reversedMessage` 的绑定也会更新。而且最妙的是我们已经以声明的方式创建了这种依赖关系：计算属性的 getter 函数是没有副作用 (side effect) 的，这使它更易于测试和理解。

计算属性缓存 vs 方法

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

```
<p>Reversed message: "{{ reversedMessage() }}"</p>
```

```
// 在组件中
methods: {
  reversedMessage: function () {
    return this.message.split('').reverse().join('')
  }
}
```

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是基于它们的依赖进行缓存的。只在相关依赖发生改变时它们才会重新求值。这就意味着只要 `message` 还没有发生改变，多次访问 `reversedMessage` 计算属性会立即返回之前的计算结果，而不必再次执行函数。

这也同样意味着下面的计算属性将不再更新，因为 `Date.now()` 不是响应式依赖：

```
computed: {
  now: function () {
    return Date.now()
  }
}
```

相比之下，每当触发重新渲染时，调用方法将总会再次执行函数。

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 A，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 A 。如果没有缓存，我们将不可避免的多次执行 A 的 getter！如果你不希望有缓存，请用方法来替代。

计算属性 vs 侦听属性

Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：侦听属性。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 `watch` ——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用计算属性而不是命令式的 `watch` 回调。细想一下这个例子：

```
<div id="demo">{{ fullName }}</div>
```

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  },
  watch: {
    firstName: function (val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName: function (val) {
      this.fullName = this.firstName + ' ' + val
    }
  }
})
```

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
```

好得多了，不是吗？

计算属性的 setter

计算属性默认只有 getter ，不过在需要时你也可以提供一个 setter ：

```
// ...
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
// ...
```

现在再运行 `vm.fullName = 'John Doe'` 时，setter 会被调用， `vm.firstName` 和 `vm.lastName` 也会相应地被更新。

侦听器

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 `watch` 选项提供了一个更通用的方法，来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的。

例如：

```
<div id="watch-example">
  <p>
    Ask a yes/no question:
    <input v-model="question">
  </p>
  <p>{{ answer }}</p>
</div>
```

```
<!-- 因为 AJAX 库和通用工具的生态已经相当丰富，Vue 核心代码没有重复 -->
<!-- 提供这些功能以保持精简。这也可以让你自由选择自己更熟悉的工具。 -->
<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></scr
ipt>
```

```
<script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js"></script>

<script>
var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'I cannot give you an answer until you ask a question!'
  },
  watch: {
    // 如果 `question` 发生改变，这个函数就会运行
    question: function (newQuestion, oldQuestion) {
      this.answer = 'Waiting for you to stop typing...'
      this.debouncedGetAnswer()
    }
  },
  created: function () {
    // `_.debounce` 是一个通过 Lodash 限制操作频率的函数。
    // 在这个例子中，我们希望限制访问 yesno.wtf/api 的频率
    // AJAX 请求直到用户输入完毕才会发出。想要了解更多关于
    // `_.debounce` 函数 (及其近亲 `_.throttle`) 的知识，
    // 请参考：https://lodash.com/docs#debounce
    this.debouncedGetAnswer = _.debounce(this.getAnswer, 500)
  },
  methods: {
    getAnswer: function () {
      if (this.question.indexOf('?') === -1) {
        this.answer = 'Questions usually contain a question mark. ;-)'
        return
      }
      this.answer = 'Thinking...'
      var vm = this
      axios.get('https://yesno.wtf/api')
        .then(function (response) {
          vm.answer = _.capitalize(response.data.answer)
        })
        .catch(function (error) {
          vm.answer = 'Error! Could not reach the API. ' + error
        })
    }
  }
})
</script>
```

结果：

{% raw %}

Ask a yes/no question:

{{ answer }}
{% endraw %}
在这个示例中，使用 `watch` 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

除了 `watch` 选项之外，您还可以使用命令式的 vm.$watch API。

Class 与 Style 绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 `v-bind` 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 `v-bind` 用于 `class` 和 `style` 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

绑定 HTML Class

对象语法

我们可以传给 `v-bind:class` 一个对象，以动态地切换 class：

```
<div v-bind:class="{ active: isActive }"></div>
```

上面的语法表示 `active` 这个 class 存在与否将取决于数据属性 `isActive` 的 truthiness。
你可以在对象中传入更多属性来动态切换多个 class。此外， `v-bind:class` 指令也可以与普通的 class 属性共存。当有如下模板:

```
<div class="static"
     v-bind:class="{ active: isActive, 'text-danger': hasError }">
</div>
```

和如下 data：

```
data: {
  isActive: true,
  hasError: false
}
```

结果渲染为：

```
<div class="static active"></div>
```

当 `isActive` 或者 `hasError` 变化时，class 列表将相应地更新。例如，如果 `hasError` 的值为 `true` ，class 列表将变为 `"static active text-danger"` 。
绑定的数据对象不必内联定义在模板里：

```
<div v-bind:class="classObject"></div>
```

```
data: {
  classObject: {
    active: true,
    'text-danger': false
  }
}
```

渲染的结果和上面一样。我们也可以在这里绑定一个返回对象的计算属性。这是一个常用且强大的模式：

```
<div v-bind:class="classObject"></div>
```

```
data: {
  isActive: true,
  error: null
},
computed: {
  classObject: function () {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal'
    }
  }
}
```

数组语法

我们可以把一个数组传给 `v-bind:class` ，以应用一个 class 列表：

```
<div v-bind:class="[activeClass, errorClass]"></div>
```

```
data: {
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

渲染为：

```
<div class="active text-danger"></div>
```

如果你也想根据条件切换列表中的 class，可以用三元表达式：

```
<div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>
```

这样写将始终添加 `errorClass` ，但是只有在 `isActive` 是 truthy[1] 时才添加 `activeClass` 。
不过，当有多个条件 class 时这样写有些繁琐。所以在数组语法中也可以使用对象语法：

```
<div v-bind:class="[{ active: isActive }, errorClass]"></div>
```

用在组件上

> 这个章节假设你已经对 Vue 组件有一定的了解。当然你也可以先跳过这里，稍后再回过头来看。

当在一个自定义组件上使用 `class` 属性时，这些类将被添加到该组件的根元素上面。这个元素上已经存在的类不会被覆盖。
例如，如果你声明了这个组件：

```
Vue.component('my-component', {
  template: '<p class="foo bar">Hi</p>'
})
```

然后在使用它的时候添加一些 class：

```
<my-component class="baz boo"></my-component>
```

HTML 将被渲染为:

```
<p class="foo bar baz boo">Hi</p>
```

对于带数据绑定 class 也同样适用：

```
<my-component v-bind:class="{ active: isActive }"></my-component>
```

当 `isActive` 为 truthy[1] 时，HTML 将被渲染成为：

```
<p class="foo bar active">Hi</p>
```

绑定内联样式

对象语法

`v-bind:style` 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用单引号括起来) 来命名：

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

```
data: {
  activeColor: 'red',
  fontSize: 30
}
```

直接绑定到一个样式对象通常更好，这会让模板更清晰：

```
<div v-bind:style="styleObject"></div>
```

```
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

同样的，对象语法常常结合返回对象的计算属性使用。

数组语法

`v-bind:style` 的数组语法可以将多个样式对象应用到同一个元素上：

```
<div v-bind:style="[baseStyles, overridingStyles]"></div>
```

自动添加前缀

当 `v-bind:style` 使用需要添加浏览器引擎前缀的 CSS 属性时，如 `transform` ，Vue.js 会自动侦测并添加相应的前缀。

多重值

> 2.3.0+

从 2.3.0 起你可以为 `style` 绑定中的属性提供一个包含多个值的数组，常用于提供多个带前缀的值，例如：

```
<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>
```

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 `display: flex` 。

译者注 [1] truthy 不是 `true`，详见 [MDN](https://developer.mozilla.org/zh-CN/docs/Glossary/Truthy) 的解释。

条件渲染

`v-if`

在字符串模板中，比如 Handlebars，我们得像这样写一个条件块：

```
<!-- Handlebars 模板 -->
{{#if ok}}
  <h1>Yes</h1>
{{/if}}
```

在 Vue 中，我们使用 `v-if` 指令实现同样的功能：

```
<h1 v-if="ok">Yes</h1>
```

也可以用 `v-else` 添加一个“else 块”：

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

在 `<template>` 元素上使用 `v-if` 条件渲染分组

因为 `v-if` 是一个指令，所以必须将它添加到一个元素上。但是如果想切换多个元素呢？此时可以把一个 `<template>` 元素当做不可见的包裹元素，并在上面使用 `v-if`。最终的渲染结果将不包含 `<template>` 元素。

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

`v-else`

你可以使用 `v-else` 指令来表示 `v-if` 的“else 块”：

```
<div v-if="Math.random() > 0.5">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

`v-else` 元素必须紧跟在带 `v-if` 或者 `v-else-if` 的元素的后面，否则它将不会被识别。

`v-else-if`

> 2.1.0 新增

`v-else-if` ，顾名思义，充当 `v-if` 的“else-if 块”，可以连续使用：

```
<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>
```

类似于 `v-else` ， `v-else-if` 也必须紧跟在带 `v-if` 或者 `v-else-if` 的元素之后。

用 `key` 管理可复用的元素

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做除了使 Vue 变得非常快之外，还有其它一些好处。例如，如果你允许用户在不同的登录方式之间切换：

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address">
</template>
```

那么在上面的代码中切换 `loginType` 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，`<input>` 不会被替换掉——仅仅是替换了它的 `placeholder` 。

自己动手试一试，在输入框中输入一些文本，然后按下切换按钮：

{% raw %}

{% endraw %}

这样也不总是符合实际需求，所以 Vue 为你提供了一种方式来表达“这两个元素是完全独立的，不要复用它

们”。只需添加一个具有唯一值的 `key` 属性即可：

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

现在，每次切换时，输入框都将被重新渲染。请看：

{% raw %}
{% endraw %}

注意，`<label>` 元素仍然会被高效地复用，因为它们没有添加 `key` 属性。

`v-show`

另一个用于根据条件展示元素的选项是 `v-show` 指令。用法大致一样：

```
<h1 v-show="ok">Hello!</h1>
```

不同的是带有 `v-show` 的元素始终会被渲染并保留在 DOM 中。`v-show` 只是简单地切换元素的 CSS 属性 `display`。

注意，`v-show` 不支持 `

列表渲染

用 `v-for` 把一个数组对应为一组元素

我们用 `v-for` 指令根据一组数组的选项列表进行渲染。 `v-for` 指令需要使用 `item in items` 形式的特殊语法， `items` 是源数据数组并且 `item` 是数组元素迭代的别名。

```
<ul id="example-1">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

结果：

{% raw %}

- {{item.message}}

{% endraw %}

在 `v-for` 块中，我们拥有对父作用域属性的完全访问权限。 `v-for` 还支持一个可选的第二个参数为当前项的索引。

```
<ul id="example-2">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>
```

```
var example2 = new Vue({
  el: '#example-2',
  data: {
    parentMessage: 'Parent',
```

```
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

结果：

{% raw%}

- {{ parentMessage }} - {{ index }} - {{ item.message }}

{% endraw %}
你也可以用 `of` 替代 `in` 作为分隔符，因为它是最接近 JavaScript 迭代器的语法：

```
<div v-for="item of items"></div>
```

一个对象的 `v-for`

你也可以用 `v-for` 通过一个对象的属性来迭代。

```
<ul id="v-for-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
```

```
new Vue({
  el: '#v-for-object',
  data: {
    object: {
      firstName: 'John',
      lastName: 'Doe',
      age: 30
    }
  }
})
```

结果：

{% raw %}

- {{ value }}

{% endraw %}

你也可以提供第二个的参数为键名：

```
<div v-for="(value, key) in object">
  {{ key }}: {{ value }}
</div>
```

{% raw %}
{{ key }}: {{ value }}
{% endraw %}
第三个参数为索引：

```
<div v-for="(value, key, index) in object">
  {{ index }}. {{ key }}: {{ value }}
</div>
```

{% raw %}
{{ index }}. {{ key }}: {{ value }}
{% endraw %}
在遍历对象时，是按 `Object.keys()` 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的。

key

当 Vue.js 用 `v-for` 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。这个类似 Vue 1.x 的 `track-by="$index"` 。
这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出。
为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 `key` 属性。理想的 `key` 值是每项都有的唯一 id。这个特殊的属性相当于 Vue 1.x 的 `track-by` ，但它的工作方式类似于一个属性，所以你需要用 `v-bind` 来绑定动态值 (在这里使用简写)：

```
<div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

建议尽可能在使用 `v-for` 时提供 `key` ，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。
因为它是 Vue 识别节点的一个通用机制， `key` 并不与 `v-for` 特别关联，key 还具有其他用途，我们将在后面的指南中看到其他用途。

数组更新检测

变异方法

Vue 包含一组观察数组的变异方法，所以它们也将会触发视图更新。这些方法如下：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

你打开控制台，然后用前面例子的 `items` 数组调用变异方法：`example1.items.push({ message: 'Baz' })` 。

替换数组

变异方法 (mutation method)，顾名思义，会改变被这些方法调用的原始数组。相比之下，也有非变异 (non-mutating method) 方法，例如： `filter()` ， `concat()` 和 `slice()` 。这些不会改变原始数组，但总是返回一个新数组。当使用非变异方法时，可以用新数组替换旧数组：

```
example1.items = example1.items.filter(function (item) {
  return item.message.match(/Foo/)
})
```

你可能认为这将导致 Vue 丢弃现有 DOM 并重新渲染整个列表。幸运的是，事实并非如此。Vue 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的、启发式的方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

注意事项

由于 JavaScript 的限制，Vue 不能检测以下变动的数组：

1. 当你利用索引直接设置一个项时，例如： `vm.items[indexOfItem] = newValue`
2. 当你修改数组的长度时，例如： `vm.items.length = newLength`

举个例子：

```
var vm = new Vue({
  data: {
```

```
    items: ['a', 'b', 'c']
  }
})
vm.items[1] = 'x' // 不是响应性的
vm.items.length = 2 // 不是响应性的
```

为了解决第一类问题，以下两种方式都可以实现和 `vm.items[indexOfItem] = newValue` 相同的效果，同时也将触发状态更新：

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
```

```
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

你也可以使用 `vm.$set` 实例方法，该方法是全局方法 `Vue.set` 的一个别名：

```
vm.$set(vm.items, indexOfItem, newValue)
```

为了解决第二类问题，你可以使用 `splice` ：

```
vm.items.splice(newLength)
```

对象更改检测注意事项

还是由于 JavaScript 的限制，Vue 不能检测对象属性的添加或删除：

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` 现在是响应式的

vm.b = 2
// `vm.b` 不是响应式的
```

对于已经创建的实例，Vue 不能动态添加根级别的响应式属性。但是，可以使用 `Vue.set(object, key, value)` 方法向嵌套对象添加响应式属性。例如，对于：

```
var vm = new Vue({
```

```
  data: {
    userProfile: {
      name: 'Anika'
    }
  }
})
```

你可以添加一个新的 `age` 属性到嵌套的 `userProfile` 对象：

```
Vue.set(vm.userProfile, 'age', 27)
```

你还可以使用 `vm.$set` 实例方法，它只是全局 `Vue.set` 的别名：

```
vm.$set(vm.userProfile, 'age', 27)
```

有时你可能需要为已有对象赋予多个新属性，比如使用 `Object.assign()` 或 `_.extend()` 。在这种情况下，你应该用两个对象的属性创建一个新的对象。所以，如果你想添加新的响应式属性，不要像这样：

```
Object.assign(vm.userProfile, {
  age: 27,
  favoriteColor: 'Vue Green'
})
```

你应该这样做：

```
vm.userProfile = Object.assign({}, vm.userProfile, {
  age: 27,
  favoriteColor: 'Vue Green'
})
```

显示过滤/排序结果

有时，我们想要显示一个数组的过滤或排序副本，而不实际改变或重置原始数据。在这种情况下，可以创建返回过滤或排序数组的计算属性。

例如：

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
```

```
computed: {
  evenNumbers: function () {
    return this.numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}
```

在计算属性不适用的情况下 (例如，在嵌套 `v-for` 循环中) 你可以使用一个 method 方法：

```
<li v-for="n in even(numbers)">{{ n }}</li>
```

```
data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
methods: {
  even: function (numbers) {
    return numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}
```

一段取值范围的 `v-for`

`v-for` 也可以取整数。在这种情况下，它将重复多次模板。

```
<div>
  <span v-for="n in 10">{{ n }} </span>
</div>
```

结果：

{% raw %}

{{ n }}
{% endraw %}

`v-for` on a `<template>`

类似于 `v-if`，你也可以利用带有 `v-for` 的 `<template>` 渲染多个元素。比如：

```
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
```

```
  </template>
</ul>
```

v-for with v-if

当它们处于同一节点，`v-for` 的优先级比 `v-if` 更高，这意味着 `v-if` 将分别重复运行于每个 `v-for` 循环中。当你想为仅有的一些项渲染节点时，这种优先级的机制会十分有用，如下：

```
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo }}
</li>
```

上面的代码只传递了未完成的 todos。

而如果你的目的是有条件地跳过循环的执行，那么可以将 `v-if` 置于外层元素 (或 `<template>`)上。如：

```
<ul v-if="todos.length">
  <li v-for="todo in todos">
    {{ todo }}
  </li>
</ul>
<p v-else>No todos left!</p>
```

一个组件的 v-for

> 了解组件相关知识，查看 组件。完全可以先跳过它，以后再回来查看。

在自定义组件里，你可以像任何普通元素一样用 `v-for` 。

```
<my-component v-for="item in items" :key="item.id"></my-component>
```

> 2.2.0+ 的版本里，当在组件中使用 `v-for` 时，`key` 现在是必须的。

然而，任何数据都不会被自动传递到组件里，因为组件有自己独立的作用域。为了把迭代数据传递到组件里，我们要用 `props` ：

```
<my-component
  v-for="(item, index) in items"
  v-bind:item="item"
  v-bind:index="index"
  v-bind:key="item.id"
></my-component>
```

不自动将 `item` 注入到组件里的原因是，这会使得组件与 `v-for` 的运作紧密耦合。明确组件数据的来源能够使组件在其他场合重复使用。

下面是一个简单的 todo list 的完整例子：

```
<div id="todo-list-example">
  <form v-on:submit.prevent="addNewTodo">
    <label for="new-todo">Add a todo</label>
    <input
      v-model="newTodoText"
      id="new-todo"
      placeholder="E.g. Feed the cat"
    >
    <button>Add</button>
  </form>
  <ul>
    <li
      is="todo-item"
      v-for="(todo, index) in todos"
      v-bind:key="todo.id"
      v-bind:title="todo.title"
      v-on:remove="todos.splice(index, 1)"
    ></li>
  </ul>
</div>
```

注意这里的 `is="todo-item"` 属性。这种做法在使用 DOM 模板时是十分必要的，因为在 `

`元素内只有 `

- `元素会被看作有效内容。这样做实现的效果与 `` 相同，但是可以避开一些潜在的浏览器解析错误。查看 [DOM 模板解析说明](components.html#解析-DOM-模板时的注意事项) 来了解更多信息。

```
Vue.component('todo-item', {
  template: '\
    <li>\
      {{ title }}\
      <button v-on:click="$emit(\'remove\')">Remove</button>\
    </li>\
  ',
  props: ['title']
})

new Vue({
  el: '#todo-list-example',
  data: {
    newTodoText: '',
    todos: [
      {
```

```
          id: 1,
          title: 'Do the dishes',
        },
        {
          id: 2,
          title: 'Take out the trash',
        },
        {
          id: 3,
          title: 'Mow the lawn'
        }
      ],
      nextTodoId: 4
    },
    methods: {
      addNewTodo: function () {
        this.todos.push({
          id: this.nextTodoId++,
          title: this.newTodoText
        })
        this.newTodoText = ''
      }
    }
  })
```

{% raw %}

-

{% endraw %}

事件处理

监听事件

可以用 `v-on` 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

示例：

```
<div id="example-1">
  <button v-on:click="counter += 1">Add 1</button>
  <p>The button above has been clicked {{ counter }} times.</p>
</div>
```

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    counter: 0
  }
})
```

结果：

{% raw %}

The button above has been clicked {{ counter }} times.

{% endraw %}

事件处理方法

然而许多事件处理逻辑会更为复杂，所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。因此 `v-on` 还可以接收一个需要调用的方法名称。

示例：

```
<div id="example-2">
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>
```

```
var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // 在 `methods` 对象中定义方法
```

```
  methods: {
    greet: function (event) {
      // `this` 在方法里指向当前 Vue 实例
      alert('Hello ' + this.name + '!')
      // `event` 是原生 DOM 事件
      if (event) {
        alert(event.target.tagName)
      }
    }
  }
})

// 也可以用 JavaScript 直接调用方法
example2.greet() // => 'Hello Vue.js!'
```

结果：

{% raw %}
{% endraw %}

内联处理器中的方法

除了直接绑定到一个方法，也可以在内联 JavaScript 语句中调用方法：

```
<div id="example-3">
  <button v-on:click="say('hi')">Say hi</button>
  <button v-on:click="say('what')">Say what</button>
</div>
```

```
new Vue({
  el: '#example-3',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
})
```

结果：

{% raw %}
{% endraw %}

有时也需要在内联语句处理器中访问原始的 DOM 事件。可以用特殊变量 `$event` 把它传入方法：

```
<button v-on:click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>
```

```
// ...
methods: {
  warn: function (message, event) {
    // 现在我们可以访问原生事件对象
    if (event) event.preventDefault()
    alert(message)
  }
}
```

事件修饰符

在事件处理程序中调用 `event.preventDefault()` 或 `event.stopPropagation()` 是非常常见的需求。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题，Vue.js 为 `v-on` 提供了事件修饰符。之前提过，修饰符是由点开头的指令后缀来表示的。

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`
- `.passive`

```
<!-- 阻止单击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联 -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>

<!-- 添加事件监听器时使用事件捕获模式 -->
<!-- 即元素自身触发的事件先在此处理，然后才交由内部元素进行处理 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
<div v-on:click.self="doThat">...</div>
```

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止**所有的点击**，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击。

> 2.1.4 新增

```
<!-- 点击事件将只会触发一次 -->
<a v-on:click.once="doThis"></a>
```

不像其它只能对原生的 DOM 事件起作用的修饰符，`.once` 修饰符还能被用到自定义的组件事件上。如果你还没有阅读关于组件的文档，现在大可不必担心。

> 2.3.0 新增

Vue 还对应 `addEventListener` 中的 `passive` 选项提供了 `.passive` 修饰符。

```
<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->
<!-- 而不会等待 `onScroll` 完成  -->
<!-- 这其中包含 `event.preventDefault()` 的情况 -->
<div v-on:scroll.passive="onScroll">...</div>
```

这个 `.passive` 修饰符尤其能够提升移动端的性能。

不要把 `.passive` 和 `.prevent` 一起使用，因为 `.prevent` 将会被忽略，同时浏览器可能会向你展示一个警告。请记住，`.passive` 会告诉浏览器你*不*想阻止事件的默认行为。

按键修饰符

在监听键盘事件时，我们经常需要检查常见的键值。Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符：

```
<!-- 只有在 `keyCode` 是 13 时调用 `vm.submit()` -->
<input v-on:keyup.13="submit">
```

记住所有的 `keyCode` 比较困难，所以 Vue 为最常用的按键提供了别名：

```
<!-- 同上 -->
<input v-on:keyup.enter="submit">

<!-- 缩写语法 -->
<input @keyup.enter="submit">
```

全部的按键别名：

- `.enter`
- `.tab`
- `.delete` (捕获“删除”和“退格”键)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

可以通过全局 `config.keyCodes` 对象自定义按键修饰符别名：

```
// 可以使用 `v-on:keyup.f1`
Vue.config.keyCodes.f1 = 112
```

自动匹配按键修饰符

> 2.5.0 新增

你也可直接将 `KeyboardEvent.key` 暴露的任意有效按键名转换为 kebab-case 来作为修饰符：

```
<input @keyup.page-down="onPageDown">
```

在上面的例子中，处理函数仅在 `$event.key === 'PageDown'` 时被调用。

有一些按键 (`.esc` 以及所有的方向键) 在 IE9 中有不同的 `key` 值, 如果你想支持 IE9，它们的内置别名应该是首选。

系统修饰键

> 2.1.0 新增

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

> 注意：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽

> 标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”。

例如：

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

请注意修饰键与常规按键不同，在和 `keyup` 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 `ctrl` 的情况下释放其它按键，才能触发 `keyup.ctrl`。而单单释放 `ctrl` 也不会触发事件。如果你想要这样的行为，请为 `ctrl` 换用 `keyCode`：`keyup.17`。

`.exact` 修饰符

> 2.5.0 新增

`.exact` 修饰符允许你控制由精确的系统修饰符组合触发的事件。

```
<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->
<button @click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 被按下的时候才触发 -->
<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->
<button @click.exact="onClick">A</button>
```

鼠标按钮修饰符

> 2.2.0 新增

- `.left`
- `.right`
- `.middle`

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

为什么在 HTML 中监听事件?

你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护上的困难。实际上，使用 `v-on` 有几个好处：

1. 扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。
2. 因为你无须在 JavaScript 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。
3. 当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

表单输入绑定

基础用法

你可以用 `v-model` 指令在表单 `<input>` 、 `<textarea>` 及 `<select>` 元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇，但 `v-model` 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

`v-model` 会忽略所有表单元素的 `value`、`checked`、`selected` 特性的初始值而总是将 Vue 实例的数据作为数据来源。你应该通过 JavaScript 在组件的 `data` 选项中声明初始值。

对于需要使用[输入法](https://zh.wikipedia.org/wiki/%E8%BE%93%E5%85%A5%E6%B3%95) (如中文、日文、韩文等) 的语言，你会发现 `v-model` 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 `input` 事件。

文本

```
<input v-model="message" placeholder="edit me">
<p>Message is: {{ message }}</p>
```

{% raw %}

Message is: {{ message }}

{% endraw %}

多行文本

```
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p>
<br>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

{% raw %}

Multiline message is:

{{ message }}

{% endraw %}

在文本区域插值 (``) 并不会生效，应用 `v-model` 来代替。

复选框

单个复选框，绑定到布尔值：

```
<input type="checkbox" id="checkbox" v-model="checked">
```

```
  <label for="checkbox">{{ checked }}</label>
```

{% raw %}
{{ checked }}
{% endraw %}
多个复选框，绑定到同一个数组：

```
<div id='example-3'>
  <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
  <label for="jack">Jack</label>
  <input type="checkbox" id="john" value="John" v-model="checkedNames">
  <label for="john">John</label>
  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
  <label for="mike">Mike</label>
  <br>
  <span>Checked names: {{ checkedNames }}</span>
</div>
```

```
new Vue({
  el: '#example-3',
  data: {
    checkedNames: []
  }
})
```

{% raw %}
JackJohnMike
Checked names: {{ checkedNames }}
{% endraw %}

单选按钮

```
<div id="example-4">
  <input type="radio" id="one" value="One" v-model="picked">
  <label for="one">One</label>
  <br>
  <input type="radio" id="two" value="Two" v-model="picked">
  <label for="two">Two</label>
  <br>
  <span>Picked: {{ picked }}</span>
</div>
```

```
new Vue({
  el: '#example-4',
  data: {
```

```
    picked: ''
  }
})
```

{% raw %}

One
Two
Picked: {{ picked }}
{% endraw %}

选择框

单选时：

```
<div id="example-5">
  <select v-model="selected">
    <option disabled value="">请选择</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
```

```
new Vue({
  el: '...',
  data: {
    selected: ''
  }
})
```

{% raw %}

请选择 Selected: {{ selected }}
{% endraw %}
如果 `v-model` 表达式的初始值未能匹配任何选项，`

Selected: {{ selected }} {% endraw %}

用 `v-for` 渲染的动态选项：

```
<select v-model="selected">
  <option v-for="option in options" v-bind:value="option.value">
    {{ option.text }}
  </option>
</select>
<span>Selected: {{ selected }}</span>
```

```
new Vue({
  el: '...',
  data: {
    selected: 'A',
    options: [
      { text: 'One', value: 'A' },
      { text: 'Two', value: 'B' },
      { text: 'Three', value: 'C' }
    ]
  }
})
```

{% raw %}
{{ option.text }} Selected: {{ selected }}
{% endraw %}

值绑定

对于单选按钮，复选框及选择框的选项， `v-model` 绑定的值通常是静态字符串 (对于复选框也可以是布尔值)：

```
<!-- 当选中时，`picked` 为字符串 "a" -->
<input type="radio" v-model="picked" value="a">

<!-- `toggle` 为 true 或 false -->
<input type="checkbox" v-model="toggle">

<!-- 当选中第一个选项时，`selected` 为字符串 "abc" -->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

但是有时我们可能想把值绑定到 Vue 实例的一个动态属性上，这时可以用 `v-bind` 实现，并且这个属性的值可以不是字符串。

复选框

```
<input
  type="checkbox"
  v-model="toggle"
  true-value="yes"
  false-value="no"
>
```

```
// 当选中时
vm.toggle === 'yes'
// 当没有选中时
vm.toggle === 'no'
```

这里的 `true-value` 和 `false-value` 特性并不会影响输入控件的 `value` 特性，因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交，(比如“yes”或“no”)，请换用单选按钮。

单选按钮

```
<input type="radio" v-model="pick" v-bind:value="a">
```

```
// 当选中时
vm.pick === vm.a
```

选择框的选项

```
<select v-model="selected">
    <!-- 内联对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
```

```
// 当选中时
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

修饰符

`.lazy`

在默认情况下， `v-model` 在每次 `input` 事件触发后将输入框的值与数据进行同步 (除了上述输入法组合文字时)。你可以添加 `lazy` 修饰符，从而转变为使用 `change` 事件进行同步：

```
<!-- 在“change”时而非“input”时更新 -->
<input v-model.lazy="msg" >
```

`.number`

如果想自动将用户的输入值转为数值类型，可以给 `v-model` 添加 `number` 修饰符：

```
<input v-model.number="age" type="number">
```

这通常很有用，因为即使在 `type="number"` 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 `parseFloat()` 解析，则会返回原始的值。

`.trim`

如果要自动过滤用户输入的首尾空白字符，可以给 `v-model` 添加 `trim` 修饰符：

```
<input v-model.trim="msg">
```

在组件上使用 `v-model`

> 如果你还不熟悉 Vue 的组件，可以暂且跳过这里。

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue 的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 `v-model` 一起使用！要了解更多，请参阅组件指南中的自定义输入组件。

组件基础

基本示例

这里有一个 Vue 组件的示例：

```
// 定义一个名为 button-counter 的新组件
Vue.component('button-counter', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'
})
```

组件是可复用的 Vue 实例，且带有一个名字：在这个例子中是 `<button-counter>` 。我们可以在一个通过 `new Vue` 创建的 Vue 根实例中，把这个组件作为自定义元素来使用：

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
```

```
new Vue({ el: '#components-demo' })
```

{% raw %}
{% endraw %}
因为组件是可复用的 Vue 实例，所以它们与 `new Vue` 接收相同的选项，例如 `data` 、 `computed` 、 `watch` 、 `methods` 以及生命周期钩子等。仅有的例外是像 `el` 这样根实例特有的选项。

组件的复用

你可以将组件进行任意次数的复用：

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

{% raw %}
{% endraw %}
注意当点击按钮时，每个组件都会各自独立维护它的 `count` 。因为你每用一次组件，就会有一个它的新实例被创建。

`data` 必须是一个函数

当我们定义这个 `<button-counter>` 组件时，你可能会发现它的 `data` 并不是像这样直接提供一个对象：

```
data: {
  count: 0
}
```

取而代之的是，一个组件的 `data` 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝：

```
data: function () {
  return {
    count: 0
  }
}
```

如果 Vue 没有这条规则，点击一个按钮就可能会像如下代码一样影响到其它所有实例：
{% raw %}
{% endraw %}

组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织：

例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。
为了能在模板中使用，这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型：全局注册和局部注册。至此，我们的组件都只是通过 `Vue.component` 全局注册的：

```
Vue.component('my-component-name', {
  // ... options ...
})
```

全局注册的组件可以用在其被注册之后的任何 (通过 `new Vue`) 新创建的 Vue 根实例，也包括其组件树中的所有子组件的模板中。

到目前为止，关于组件注册你需要了解的就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把组件注册读完。

通过 Prop 向子组件传递数据

早些时候，我们提到了创建一个博文组件的事情。问题是如果你不能向这个组件传递某一篇博文的标题或内容之类的我们想展示的数据的话，它是没有办法使用的。这也正是 prop 的由来。

Prop 是你可以在组件上注册的一些自定义特性。当一个值传递给一个 prop 特性的时候，它就变成了那个组件实例的一个属性。为了给博文组件传递一个标题，我们可以用一个 `props` 选项将其包含在该组件可接受的 prop 列表中：

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 `data` 中的值一样。

一个 prop 被注册之后，你就可以像这样把数据作为一个自定义特性传递进来：

```
<blog-post title="My journey with Vue"></blog-post>
<blog-post title="Blogging with Vue"></blog-post>
<blog-post title="Why Vue is so fun"></blog-post>
```

{% raw %}
{% endraw %}

然而在一个典型的应用中，你可能在 `data` 里有一个博文的数组：

```
new Vue({
  el: '#blog-post-demo',
  data: {
    posts: [
      { id: 1, title: 'My journey with Vue' },
      { id: 2, title: 'Blogging with Vue' },
      { id: 3, title: 'Why Vue is so fun' }
    ]
  }
})
```

并想要为每篇博文渲染一个组件：

```
<blog-post
```

```
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title"
></blog-post>
```

如上所示，你会发现我们可以使用 `v-bind` 来动态传递 prop。这在你一开始不清楚要渲染的具体内容，比如从一个 API 获取博文列表的时候，是非常有用的。

到目前为止，关于 prop 你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把 prop 读完。

单个根元素

当构建一个 `<blog-post>` 组件时，你的模板最终会包含的东西远不止一个标题：

```
<h3>{{ title }}</h3>
```

最最起码，你会包含这篇博文的正文：

```
<h3>{{ title }}</h3>
<div v-html="content"></div>
```

然而如果你在模板中尝试这样写，Vue 会显示一个错误，并解释道 every component must have a single root element (每个组件必须只有一个根元素)。你可以将模板的内容包裹在一个父元素内，来修复这个问题，例如：

```
<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>
```

看起来当组件变得越来越复杂的时候，我们的博文不只需要标题和内容，还需要发布日期、评论等等。为每个相关的信息定义一个 prop 会变得很麻烦：

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title"
  v-bind:content="post.content"
  v-bind:publishedAt="post.publishedAt"
  v-bind:comments="post.comments"
></blog-post>
```

所以是时候重构一下这个 `<blog-post>` 组件了，让它变成接受一个单独的 `post` prop：

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:post="post"
></blog-post>
```

```
Vue.component('blog-post', {
  props: ['post'],
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <div v-html="post.content"></div>
    </div>
  `
})
```

上述的这个和一些接下来的示例使用了 JavaScript 的[模板字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Template_literals)来让多行的模板更易读。它们在 IE 下并没有被支持，所以如果你需要在不 (经过 Babel 或 TypeScript 之类的工具) 编译的情况下支持 IE，请使用[折行转义字符](https://css-tricks.com/snippets/javascript/multiline-string-variables-in-javascript/)取而代之。

现在，不论何时为 `post` 对象添加一个新的属性，它都会自动地在 `<blog-post>` 内可用。

通过事件向父级组件发送消息

在我们开发 `<blog-post>` 组件时，它的一些功能可能要求我们和父级组件进行沟通。例如我们可能会引入一个可访问性的功能来放大博文的字号，同时让页面的其它部分保持默认的字号。

在其父组件中，我们可以通过添加一个 `postFontSize` 数据属性来支持这个功能：

```
new Vue({
  el: '#blog-posts-events-demo',
  data: {
    posts: [/* ... */],
    postFontSize: 1
  }
})
```

它可以在模板中用来控制所有博文的字号：

```
<div id="blog-posts-events-demo">
  <div :style="{ fontSize: postFontSize + 'em' }">
    <blog-post
      v-for="post in posts"
      v-bind:key="post.id"
```

```
      v-bind:post="post"
    ></blog-post>
  </div>
</div>
```

现在我们在每篇博文正文之前添加一个按钮来放大字号：

```
Vue.component('blog-post', {
  props: ['post'],
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <button>
        Enlarge text
      </button>
      <div v-html="post.content"></div>
    </div>
  `
})
```

问题是这个按钮不会做任何事：

```
<button>
  Enlarge text
</button>
```

当点击这个按钮时，我们需要告诉父级组件放大所有博文的文本。幸好 Vue 实例提供了一个自定义事件的系统来解决这个问题。我们可以调用内建的 **`$emit`** 方法并传入事件的名字，来向父级组件触发一个事件：

```
<button v-on:click="$emit('enlarge-text')">
  Enlarge text
</button>
```

然后我们可以用 `v-on` 在博文组件上监听这个事件，就像监听一个原生 DOM 事件一样：

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += 0.1"
></blog-post>
```

{% raw %}
{% endraw %}

使用事件抛出一个值

有的时候用一个事件来抛出一个特定的值是非常有用的。例如我们可能想让 `<blog-post>` 组件决定它的文本要放大多少。这时可以使用 `$emit` 的第二个参数来提供这个值：

```
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

然后当在父级组件监听这个事件的时候，我们可以通过 `$event` 访问到被抛出的这个值：

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += $event"
></blog-post>
```

或者，如果这个事件处理函数是一个方法：

```
<blog-post
  ...
  v-on:enlarge-text="onEnlargeText"
></blog-post>
```

那么这个值将会作为第一个参数传入这个方法：

```
methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

在组件上使用 `v-model`

自定义事件也可以用于创建支持 `v-model` 的自定义输入组件。记住：

```
<input v-model="searchText">
```

等价于：

```
<input
  v-bind:value="searchText"
  v-on:input="searchText = $event.target.value"
>
```

当用在组件上时，`v-model` 则会这样：

```
<custom-input
  v-bind:value="searchText"
  v-on:input="searchText = $event"
></custom-input>
```

为了让它正常工作，这个组件内的 `<input>` 必须：

- 将其 `value` 特性绑定到一个名叫 `value` 的 prop 上
- 在其 `input` 事件被触发时，将新的值通过自定义的 `input` 事件抛出

写成代码之后是这样的：

```
Vue.component('custom-input', {
  props: ['value'],
  template: `
    <input
      v-bind:value="value"
      v-on:input="$emit('input', $event.target.value)"
    >
  `
})
```

现在 `v-model` 就应该可以在这个组件上完美地工作起来了：

```
<custom-input v-model="searchText"></custom-input>
```

到目前为止，关于组件自定义事件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把自定义事件读完。

通过插槽分发内容

和 HTML 元素一样，我们经常需要向一个组件传递内容，像这样：

```
<alert-box>
  Something bad happened.
</alert-box>
```

可能会渲染出这样的东西：

{% raw %}
Something bad happened.
{% endraw %}

幸好，Vue 自定义的 `<slot>` 元素让这变得非常简单：

```
Vue.component('alert-box', {
  template: `
    <div class="demo-alert-box">
      <strong>Error!</strong>
      <slot></slot>
    </div>
  `
})
```

如你所见，我们只要在需要的地方加入插槽就行了——就这么简单！

到目前为止，关于插槽你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把插槽读完。

动态组件

有的时候，在不同组件之间进行动态切换是非常有用的，比如在一个多标签的界面里：

{% raw %}
{% endraw %}

上述内容可以通过 Vue 的 `<component>` 元素加一个特殊的 `is` 特性来实现：

```
<!-- 组件会在 `currentTabComponent` 改变时改变 -->
<component v-bind:is="currentTabComponent"></component>
```

在上述示例中，`currentTabComponent` 可以包括

- 已注册组件的名字，或
- 一个组件的选项对象

你可以在这里查阅并体验完整的代码，或在这个版本了解绑定组件选项对象，而不是已注册组件名的示例。

到目前为止，关于动态组件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把动态和异步组件读完。

解析 DOM 模板时的注意事项

有些 HTML 元素，诸如 `<ul>` 、 `<ol>` 、 `<table>` 和 `<select>` ，对于哪些元素可以出现在其内部是有严格限制的。而有些元素，诸如 `<li>` 、 `<tr>` 和 `<option>` ，只能出现在其它某些特定的元素内部。

这会导致我们使用这些有约束条件的元素时遇到一些问题。例如：

```
<table>
```

```
  <blog-post-row></blog-post-row>
</table>
```

这个自定义组件 `<blog-post-row>` 会被作为无效的内容提升到外部，并导致最终渲染结果出错。幸好这个特殊的 `is` 特性给了我们一个变通的办法：

```
<table>
  <tr is="blog-post-row"></tr>
</table>
```

需要注意的是如果我们从以下来源使用模板的话，这条限制是不存在的：

- 字符串 (例如： `template: '...'`)
- 单文件组件 (`.vue`)
- `<script type="text/x-template">`

到这里，你需要了解的解析 DOM 模板时的注意事项——实际上也是 Vue 的全部必要内容，大概就是这些了。恭喜你！接下来还有很多东西要去学习，不过首先，我们推荐你先休息一下，试用一下 Vue，自己随意做些好玩的东西。
如果你感觉已经掌握了这些知识，我们推荐你再回来把完整的组件指南，包括侧边栏中组件深入章节的所有页面读完。

深入了解组件

组件注册

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

组件名

在注册一个组件的时候，我们始终需要给它一个名字。比如在全局注册的时候我们已经看到了：

```
Vue.component('my-component-name', { /* ... */ })
```

该组件名就是 `Vue.component` 的第一个参数。
你给予组件的名字可能依赖于你打算拿它来做什么。当直接在 DOM 中使用一个组件 (而不是在字符串模板或单文件组件) 的时候，我们强烈推荐遵循 W3C 规范中的自定义组件名 (字母全小写且必须包含一个连字符)。这会帮助你避免和当前以及未来的 HTML 元素相冲突。
你可以在风格指南中查阅到关于组件名的其它建议。

组件名大小写

定义组件名的方式有两种：

使用 kebab-case

```
Vue.component('my-component-name', { /* ... */ })
```

当使用 kebab-case (短横线分隔命名) 定义一个组件时，你也必须在引用这个自定义元素时使用 kebab-case，例如 `<my-component-name>` 。

使用 PascalCase

```
Vue.component('MyComponentName', { /* ... */ })
```

当使用 PascalCase (驼峰式命名) 定义一个组件时，你在引用这个自定义元素时两种命名法都可以使用。也就是说 `<my-component-name>` 和 `<MyComponentName>` 都是可接受的。注意，尽管如此，直接在 DOM (即非字符串的模板) 中使用时只有 kebab-case 是有效的。

全局注册

到目前为止，我们只用过 `Vue.component` 来创建组件：

```
Vue.component('my-component-name', {
  // ... 选项 ...
```

```
})
```

这些组件是全局注册的。也就是说它们在注册之后可以用在任何新创建的 Vue 根实例（`new Vue`）的模板中。比如：

```
Vue.component('component-a', { /* ... */ })
Vue.component('component-b', { /* ... */ })
Vue.component('component-c', { /* ... */ })

new Vue({ el: '#app' })
```

```
<div id="app">
  <component-a></component-a>
  <component-b></component-b>
  <component-c></component-c>
</div>
```

在所有子组件中也是如此，也就是说这三个组件在各自内部也都可以相互使用。

局部注册

全局注册往往是不够理想的。比如，如果你使用一个像 webpack 这样的构建系统，全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

在这些情况下，你可以通过一个普通的 JavaScript 对象来定义组件：

```
var ComponentA = { /* ... */ }
var ComponentB = { /* ... */ }
var ComponentC = { /* ... */ }
```

然后在 `components` 选项中定义你想要使用的组件：

```
new Vue({
  el: '#app',
  components: {
    'component-a': ComponentA,
    'component-b': ComponentB
  }
})
```

对于 `components` 对象中的每个属性来说，其属性名就是自定义元素的名字，其属性值就是这个组件的选项对象。

注意局部注册的组件在其子组件中不可用。例如，如果你希望 `ComponentA` 在 `ComponentB` 中可用，则你需要这样写：

```
var ComponentA = { /* ... */ }

var ComponentB = {
  components: {
    'component-a': ComponentA
  },
  // ...
}
```

或者如果你通过 Babel 和 webpack 使用 ES2015 模块，那么代码看起来更像：

```
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  },
  // ...
}
```

注意在 ES2015+ 中，在对象中放一个类似 `ComponentA` 的变量名其实是 `ComponentA: ComponentA` 的缩写，即这个变量名同时是：

- 用在模板中的自定义元素的名称
- 包含了这个组件选项的变量名

模块系统

如果你没有通过 `import` / `require` 使用一个模块系统，也许可以暂且跳过这个章节。如果你使用了，那么我们会为你提供一些特殊的使用说明和注意事项。

在模块系统中局部注册

如果你还在阅读，说明你使用了诸如 Babel 和 webpack 的模块系统。在这些情况下，我们推荐创建一个 `components` 目录，并将每个组件放置在其各自的文件中。
然后你需要在局部注册之前导入每个你想使用的组件。例如，在一个假设的 `ComponentB.js` 或 `ComponentB.vue` 文件中：

```
import ComponentA from './ComponentA'
import ComponentC from './ComponentC'
```

```
export default {
  components: {
    ComponentA,
    ComponentC
  },
  // ...
}
```

现在 `ComponentA` 和 `ComponentC` 都可以在 `ComponentB` 的模板中使用了。

基础组件的自动化全局注册

可能你的许多组件只是包裹了一个输入框或按钮之类的元素，是相对通用的。我们有时候会把它们称为基础组件，它们会在各个组件中被频繁的用到。
所以会导致很多组件里都会有一个包含基础组件的长列表：

```
import BaseButton from './BaseButton.vue'
import BaseIcon from './BaseIcon.vue'
import BaseInput from './BaseInput.vue'

export default {
  components: {
    BaseButton,
    BaseIcon,
    BaseInput
  }
}
```

而只是用于模板中的一小部分：

```
<BaseInput
  v-model="searchText"
  @keydown.enter="search"
/>
<BaseButton @click="search">
  <BaseIcon name="search"/>
</BaseButton>
```

幸好如果你使用了 webpack (或在内部使用了 webpack 的 Vue CLI 3+)，那么就可以使用 `require.context` 只全局注册这些非常通用的基础组件。这里有一份可以让你在应用入口文件 (比如 `src/main.js`) 中全局导入基础组件的示例代码：

```
import Vue from 'vue'
```

```
import upperFirst from 'lodash/upperFirst'
import camelCase from 'lodash/camelCase'

const requireComponent = require.context(
  // 其组件目录的相对路径
  './components',
  // 是否查询其子目录
  false,
  // 匹配基础组件文件名的正则表达式
  /Base[A-Z]\w+\.(vue|js)$/
)

requireComponent.keys().forEach(fileName => {
  // 获取组件配置
  const componentConfig = requireComponent(fileName)

  // 获取组件的 PascalCase 命名
  const componentName = upperFirst(
    camelCase(
      // 剥去文件名开头的 `./` 和结尾的扩展名
      fileName.replace(/^\.\/(.*)\.\w+$/, '$1')
    )
  )

  // 全局注册组件
  Vue.component(
    componentName,
    // 如果这个组件选项是通过 `export default` 导出的,
    // 那么就会优先使用 `.default`,
    // 否则回退到使用模块的根。
    componentConfig.default || componentConfig
  )
})
```

记住全局注册的行为必须在根 Vue 实例 (通过 `new Vue`) 创建之前发生。这里有一个真实项目情景下的示例。

Prop

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

Prop 的大小写 (camelCase vs kebab-case)

HTML 中的特性名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。这意味着当你使用 DOM 中的模板时，camelCase (驼峰命名法) 的 prop 名需要使用其等价的 kebab-case (短横线分隔命名) 命名：

```
Vue.component('blog-post', {
  // 在 JavaScript 中是 camelCase 的
  props: ['postTitle'],
  template: '<h3>{{ postTitle }}</h3>'
})
```

```
<!-- 在 HTML 中是 kebab-case 的 -->
<blog-post post-title="hello!"></blog-post>
```

重申一次，如果你使用字符串模板，那么这个限制就不存在了。

Prop 类型

到这里，我们只看到了以字符串数组形式列出的 prop：

```
props: ['title', 'likes', 'isPublished', 'commentIds', 'author']
```

但是，通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些属性的名称和值分别是 prop 各自的名称和类型：

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object
}
```

这不仅为你的组件提供了文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。你会在这个页面接下来的部分看到类型检查和其它 prop 验证。

传递静态或动态 Prop

像这样，你已经知道了可以像这样给 prop 传入一个静态的值：

```
<blog-post title="My journey with Vue"></blog-post>
```

你也知道 prop 可以通过 `v-bind` 动态赋值，例如：

```
<!-- 动态赋予一个变量的值 -->
<blog-post v-bind:title="post.title"></blog-post>

<!-- 动态赋予一个复杂表达式的值 -->
<blog-post v-bind:title="post.title + ' by ' + post.author.name"></blog-post>
```

在上述两个示例中，我们传入的值都是字符串类型的，但实际上任何类型的值都可以传给一个 prop。

传入一个数字

```
<!-- 即便 `42` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->
<blog-post v-bind:likes="42"></blog-post>

<!-- 用一个变量进行动态赋值。-->
<blog-post v-bind:likes="post.likes"></blog-post>
```

传入一个布尔值

```
<!-- 包含该 prop 没有值的情况在内，都意味着 `true`。-->
<blog-post is-published></blog-post>

<!-- 即便 `false` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->
<blog-post v-bind:is-published="false"></blog-post>

<!-- 用一个变量进行动态赋值。-->
<blog-post v-bind:is-published="post.isPublished"></blog-post>
```

传入一个数组

```
<!-- 即便数组是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>

<!-- 用一个变量进行动态赋值。-->
<blog-post v-bind:comment-ids="post.commentIds"></blog-post>
```

传入一个对象

```
<!-- 即便对象是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->
<blog-post v-bind:author="{ name: 'Veronica', company: 'Veridian Dynamics' }"></
blog-post>

<!-- 用一个变量进行动态赋值。-->
<blog-post v-bind:author="post.author"></blog-post>
```

传入一个对象的所有属性

如果你想要将一个对象的所有属性都作为 prop 传入，你可以使用不带参数的 `v-bind` (取代 `v-bind:prop-name`)。例如，对于一个给定的对象 `post` ：

```
post: {
  id: 1,
  title: 'My Journey with Vue'
}
```

下面的模板：

```
<blog-post v-bind="post"></blog-post>
```

等价于：

```
<blog-post
  v-bind:id="post.id"
  v-bind:title="post.title"
></blog-post>
```

单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。
额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。
这里有两种常见的试图改变一个 prop 的情形：

1. **这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来使用。**在这种情况下，最好定义一个本地的 data 属性并将这个 prop 用作其初始值：

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

2. **这个 prop 以一种原始的值传入且需要进行转换。**在这种情况下，最好使用这个 prop 的值来定义一个计算属性：

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

注意在 JavaScript 中对象和数组是通过引用传入的，所以对于一个数组或对象类型的 prop 来说，在子组件中改变这个对象或数组本身**将会**影响到父组件的状态。

Prop 验证

我们可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

为了定制 prop 的验证方式，你可以为 `props` 中的值提供一个带有验证需求的对象，而不是一个字符串数组。例如：

```
Vue.component('my-component', {
  props: {
    // 基础的类型检查 (`null` 匹配任何类型)
    propA: Number,
    // 多个可能的类型
    propB: [String, Number],
    // 必填的字符串
    propC: {
      type: String,
      required: true
    },
    // 带有默认值的数字
    propD: {
      type: Number,
      default: 100
    },
    // 带有默认值的对象
```

```
    propE: {
      type: Object,
      // 对象或数组默认值必须从一个工厂函数获取
      default: function () {
        return { message: 'hello' }
      }
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
        // 这个值必须匹配下列字符串中的一个
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    }
  }
})
```

当 prop 验证失败的时候，(开发环境构建版本的) Vue 将会产生一个控制台的警告。

注意那些 prop 会在一个组件实例创建**之前**进行验证，所以实例的属性 (如 `data`、`computed` 等) 在 `default` 或 `validator` 函数中是不可用的。

类型检查

`type` 可以是下列原生构造函数中的一个：

- `String`
- `Number`
- `Boolean`
- `Array`
- `Object`
- `Date`
- `Function`
- `Symbol`

额外的， `type` 还可以是一个自定义的构造函数，并且通过 `instanceof` 来进行检查确认。例如，给定下列现成的构造函数：

```
function Person (firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
}
```

你可以使用：

```
Vue.component('blog-post', {
  props: {
    author: Person
  }
})
```

来验证 `author` prop 的值是否是通过 `new Person` 创建的。

非 Prop 的特性

一个非 prop 特性是指传向一个组件，但是该组件并没有相应 prop 定义的特性。

因为显式定义的 prop 适用于向一个子组件传入信息，然而组件库的作者并不总能预见组件会被用于怎样的场景。这也是为什么组件可以接受任意的特性，而这些特性会被添加到这个组件的根元素上。

例如，想象一下你通过一个 Bootstrap 插件使用了一个第三方的 `<bootstrap-date-input>` 组件，这个插件需要在其 `<input>` 上用到一个 `data-date-picker` 特性。我们可以将这个特性添加到你的组件实例上：

```
<bootstrap-date-input data-date-picker="activated"></bootstrap-date-input>
```

然后这个 `data-date-picker="activated"` 特性就会自动添加到 `<bootstrap-date-input>` 的根元素上。

替换/合并已有的特性

想象一下 `<bootstrap-date-input>` 的模板是这样的：

```
<input type="date" class="form-control">
```

为了给我们的日期选择器插件定制一个主题，我们可能需要像这样添加一个特别的类名：

```
<bootstrap-date-input
  data-date-picker="activated"
  class="date-picker-theme-dark"
></bootstrap-date-input>
```

在这种情况下，我们定义了两个不同的 `class` 的值：

- `form-control`，这是在组件的模板内设置好的
- `date-picker-theme-dark`，这是从组件的父级传入的

对于绝大多数特性来说，从外部提供给组件的值会替换掉组件内部设置好的值。所以如果传入 `type="text"`

就会替换掉 `type="date"` 并把它破坏！庆幸的是，`class` 和 `style` 特性会稍微智能一些，即两边的值会被合并起来，从而得到最终的值：`form-control date-picker-theme-dark`。

禁用特性继承

如果你不希望组件的根元素继承特性，你可以在组件的选项中设置 `inheritAttrs: false`。例如：

```
Vue.component('my-component', {
  inheritAttrs: false,
  // ...
})
```

这尤其适合配合实例的 `$attrs` 属性使用，该属性包含了传递给一个组件的特性名和特性值，例如：

```
{
  class: 'username-input',
  placeholder: 'Enter your username'
}
```

有了 `inheritAttrs: false` 和 `$attrs`，你就可以手动决定这些特性会被赋予哪个元素。在撰写基础组件的时候是常会用到的：

```
Vue.component('base-input', {
  inheritAttrs: false,
  props: ['label', 'value'],
  template: `
    <label>
      {{ label }}
      <input
        v-bind="$attrs"
        v-bind:value="value"
        v-on:input="$emit('input', $event.target.value)"
      >
    </label>
  `
})
```

这个模式允许你在使用基础组件的时候更像是使用原始的 HTML 元素，而不会担心哪个元素是真正的根元素：

```
<base-input
  v-model="username"
  class="username-input"
  placeholder="Enter your username"
></base-input>
```

自定义事件

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

事件名

不同于组件和 prop，事件名不存在任何自动化的大小写转换。而是触发的事件名需要完全匹配监听这个事件所用的名称。举个例子，如果触发一个 camelCase 名字的事件：

```
this.$emit('myEvent')
```

则监听这个名字的 kebab-case 版本是不会有任何效果的：

```
<my-component v-on:my-event="doSomething"></my-component>
```

不同于组件和 prop，事件名不会被用作一个 JavaScript 变量名或属性名，所以就没有理由使用 camelCase 或 PascalCase 了。并且 `v-on` 事件监听器在 DOM 模板中会被自动转换为全小写 (因为 HTML 是大小写不敏感的)，所以 `v-on:myEvent` 将会变成 `v-on:myevent` ——导致 `myEvent` 不可能被监听到。
因此，我们推荐你始终使用 kebab-case 的事件名。

自定义组件的 `v-model`

> 2.2.0+ 新增

一个组件上的 `v-model` 默认会利用名为 `value` 的 prop 和名为 `input` 的事件，但是像单选框、复选框等类型的输入控件可能会将 `value` 特性用于不同的目的。 `model` 选项可以用来避免这样的冲突：

```
Vue.component('base-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean
  },
  template: `
    <input
      type="checkbox"
      v-bind:checked="checked"
      v-on:change="$emit('change', $event.target.checked)"
    >
```

```
  `
})
```

现在在这个组件上使用 `v-model` 的时候：

```
<base-checkbox v-model="lovingVue"></base-checkbox>
```

这里的 `lovingVue` 的值将会传入这个名为 `checked` 的 prop。同时当 `<base-checkbox>` 触发一个 `change` 事件并附带一个新的值的时候，这个 `lovingVue` 的属性将会被更新。
注意你仍然需要在组件的 `props` 选项里声明 `checked` 这个 prop。

将原生事件绑定到组件

你可能有很多次想要在一个组件的根元素上直接监听一个原生事件。这时，你可以使用 `v-on` 的 `.native` 修饰符：

```
<base-input v-on:focus.native="onFocus"></base-input>
```

在有的时候这是很有用的，不过在你尝试监听一个类似 `<input>` 的非常特定的元素时，这并不是个好主意。比如上述 `<base-input>` 组件可能做了如下重构，所以根元素实际上是一个 `<label>` 元素：

```
<label>
  {{ label }}
  <input
    v-bind="$attrs"
    v-bind:value="value"
    v-on:input="$emit('input', $event.target.value)"
  >
</label>
```

这时，父级的 `.native` 监听器将静默失败。它不会产生任何报错，但是 `onFocus` 处理函数不会如你预期地被调用。
为了解决这个问题，Vue 提供了一个 `$listeners` 属性，它是一个对象，里面包含了作用在这个组件上的所有监听器。例如：

```
{
  focus: function (event) { /* ... */ }
  input: function (value) { /* ... */ },
}
```

有了这个 `$listeners` 属性，你就可以配合 `v-on="$listeners"` 将所有的事件监听器指向这个组件

的某个特定的子元素。对于类似 `<input>` 的你希望它也可以配合 `v-model` 工作的组件来说，为这些监听器创建一个类似下述 `inputListeners` 的计算属性通常是非常有用的：

```
Vue.component('base-input', {
  inheritAttrs: false,
  props: ['label', 'value'],
  computed: {
    inputListeners: function () {
      var vm = this
      // `Object.assign` 将所有的对象合并为一个新对象
      return Object.assign({},
        // 我们从父级添加所有的监听器
        this.$listeners,
        // 然后我们添加自定义监听器,
        // 或覆写一些监听器的行为
        {
          // 这里确保组件配合 `v-model` 的工作
          input: function (event) {
            vm.$emit('input', event.target.value)
          }
        }
      )
    }
  },
  template: `
    <label>
      {{ label }}
      <input
        v-bind="$attrs"
        v-bind:value="value"
        v-on="inputListeners"
      >
    </label>
  `
})
```

现在 `<base-input>` 组件是一个完全透明的包裹器了，也就是说它可以完全像一个普通的 `<input>` 元素一样使用了：所有跟它相同的特性和监听器的都可以工作。

`.sync` 修饰符

> 2.3.0+ 新增

在有些情况下，我们可能需要对一个 prop 进行“双向绑定”。不幸的是，真正的双向绑定会带来维护上的问题，因为子组件可以修改父组件，且在父组件和子组件都没有明显的改动来源。

这也是为什么我们推荐以 `update:myPropName` 的模式触发事件取而代之。举个例子，在一个包含

`title` prop 的假设的组件中，我们可以用以下方法表达对其赋新值的意图：

```
this.$emit('update:title', newTitle)
```

然后父组件可以监听那个事件并根据需要更新一个本地的数据属性。例如：

```
<text-document
  v-bind:title="doc.title"
  v-on:update:title="doc.title = $event"
></text-document>
```

为了方便起见，我们为这种模式提供一个缩写，即 `.sync` 修饰符：

```
<text-document v-bind:title.sync="doc.title"></text-document>
```

注意带有 `.sync` 修饰符的 `v-bind` 不能和表达式一起使用 (例如 `v-bind:title.sync="doc.title + '!'"` 是无效的)。取而代之的是，你只能提供你想要绑定的属性名，类似 `v-model` 。

当我们用一个对象同时设置多个 prop 的时候，也可以将这个 `.sync` 修饰符和 `v-bind` 配合使用：

```
<text-document v-bind.sync="doc"></text-document>
```

这样会把 `doc` 对象中的每一个属性 (如 `title`) 都作为一个独立的 prop 传进去，然后各自添加用于更新的 `v-on` 监听器。

将 `v-bind.sync` 用在一个字面量的对象上，例如 `v-bind.sync="{ title: doc.title }"` ，是无法正常工作的，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

插槽

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

插槽内容

Vue 实现了一套内容分发的 API，这套 API 基于当前的 Web Components 规范草案，将 `<slot>` 元素作为承载分发内容的出口。

它允许你像这样合成组件：

```
<navigation-link url="/profile">
  Your Profile
</navigation-link>
```

然后你在 `<navigation-link>` 的模板中可能会写为：

```
<a
  v-bind:href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```

当组件渲染的时候，这个 `<slot>` 元素将会被替换为“Your Profile”。插槽内可以包含任何模板代码，包括 HTML：

```
<navigation-link url="/profile">
  <!-- 添加一个 Font Awesome 图标 -->
  <span class="fa fa-user"></span>
  Your Profile
</navigation-link>
```

甚至其它的组件：

```
<navigation-link url="/profile">
  <!-- 添加一个图标的组件 -->
  <font-awesome-icon name="user"></font-awesome-icon>
  Your Profile
</navigation-link>
```

如果 `<navigation-link>` 没有包含一个 `<slot>` 元素，则任何传入它的内容都会被抛弃。

具名插槽

有些时候我们需要多个插槽。例如，一个假设的 `<base-layout>` 组件的模板如下：

```
<div class="container">
  <header>
    <!-- 我们希望把页头放这里 -->
  </header>
  <main>
    <!-- 我们希望把主要内容放这里 -->
  </main>
  <footer>
    <!-- 我们希望把页脚放这里 -->
  </footer>
</div>
```

对于这样的情况， `<slot>` 元素有一个特殊的特性： `name` 。这个特性可以用来定义额外的插槽：

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

在向具名插槽提供内容的时候，我们可以在一个父组件的 `<template>` 元素上使用 `slot` 特性：

```
<base-layout>
  <template slot="header">
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template slot="footer">
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

另一种 `slot` 特性的用法是直接用在一个普通的元素上：

```
<base-layout>
  <h1 slot="header">Here might be a page title</h1>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
</base-layout>
```

我们还是可以保留一个未命名插槽，这个插槽是默认插槽，也就是说它会作为所有未匹配到插槽的内容的统一出口。上述两个示例渲染出来的 HTML 都将会是：

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

插槽的默认内容

有的时候为插槽提供默认的内容是很有用的。例如，一个 `<submit-button>` 组件可能希望这个按钮的默认内容是“Submit”，但是同时允许用户覆写为“Save”、“Upload”或别的内容。
你可以在组件模板里的 `<slot>` 标签内部指定默认的内容来做到这一点。

```
<button type="submit">
  <slot>Submit</slot>
</button>
```

如果父组件为这个插槽提供了内容，则默认的内容会被替换掉。

编译作用域

当你想在插槽内使用数据时，例如：

```
<navigation-link url="/profile">
```

```
  Logged in as {{ user.name }}
</navigation-link>
```

该插槽可以访问跟这个模板的其它地方相同的实例属性 (也就是说“作用域”是相同的)。但这个插槽不能访问 `<navigation-link>` 的作用域。例如尝试访问 `url` 是不会工作的。牢记一条准则：

> 父组件模板的所有东西都会在父级作用域内编译；子组件模板的所有东西都会在子级作用域内编译。

作用域插槽

> 2.1.0+ 新增

有的时候你希望提供的组件带有一个可从子组件获取数据的可复用的插槽。例如一个简单的 `<todo-list>` 组件的模板可能包含了如下代码：

```
<ul>
  <li
    v-for="todo in todos"
    v-bind:key="todo.id"
  >
    {{ todo.text }}
  </li>
</ul>
```

但是在我们应用的某些部分，我们希望每个独立的待办项渲染出和 `todo.text` 不太一样的东西。这也是作用域插槽的用武之地。

为了让这个特性成为可能，你需要做的全部事情就是将待办项内容包裹在一个 `<slot>` 元素上，然后将所有和其上下文相关的数据传递给这个插槽：在这个例子中，这个数据是 `todo` 对象：

```
<ul>
  <li
    v-for="todo in todos"
    v-bind:key="todo.id"
  >
    <!-- 我们为每个 todo 准备了一个插槽, -->
    <!-- 将 `todo` 对象作为一个插槽的 prop 传入。-->
    <slot v-bind:todo="todo">
      <!-- 回退的内容 -->
      {{ todo.text }}
    </slot>
  </li>
</ul>
```

现在当我们使用 `<todo-list>` 组件的时候，我们可以选择为待办项定义一个不一样的 `<template>` 作为替代方案，并且可以通过 `slot-scope` 特性从子组件获取数据：

```
<todo-list v-bind:todos="todos">
  <!-- 将 `slotProps` 定义为插槽作用域的名字 -->
  <template slot-scope="slotProps">
    <!-- 为待办项自定义一个模板，-->
    <!-- 通过 `slotProps` 定制每个待办项。-->
    <span v-if="slotProps.todo.isComplete">✓</span>
    {{ slotProps.todo.text }}
  </template>
</todo-list>
```

> 在 2.5.0+，`slot-scope` 不再限制在 `<template>` 元素上使用，而可以用在插槽内的任何元素或组件上。

解构 `slot-scope`

如果一个 JavaScript 表达式在一个函数定义的参数位置有效，那么这个表达式实际上就可以被 `slot-scope` 接受。也就是说你可以在支持的环境下 (单文件组件或现代浏览器)，在这些表达式中使用 ES2015 解构语法。例如：

```
<todo-list v-bind:todos="todos">
  <template slot-scope="{ todo }">
    <span v-if="todo.isComplete">✓</span>
    {{ todo.text }}
  </template>
</todo-list>
```

这会使作用域插槽变得更干净一些。

动态组件 & 异步组件

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

在动态组件上使用 `keep-alive`

我们之前曾经在一个多标签的界面中使用 `is` 特性来切换不同的组件：

```
<component v-bind:is="currentTabComponent"></component>
```

当在这些组件之间切换的时候，你有时会想保持这些组件的状态，以避免反复重渲染导致的性能问题。例如我们来展开说一说这个多标签界面：

{% raw %}
{% endraw %}

你会注意到，如果你选择了一篇文章，切换到 Archive 标签，然后再切换回 Posts，是不会继续展示你之前选择的文章的。这是因为你每次切换新标签的时候，Vue 都创建了一个新的 `currentTabComponent` 实例。

重新创建动态组件的行为通常是非常有用的，但是在这个案例中，我们更希望那些标签的组件实例能够被在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 `<keep-alive>` 元素将其动态组件包裹起来。

```
<!-- 失活的组件将会被缓存！-->
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
```

来看看修改后的结果：

{% raw %}
{% endraw %}

现在这个 Posts 标签保持了它的状态 (被选中的文章) 甚至当它未被渲染时也是如此。你可以在这个 fiddle 查阅到完整的代码。

注意这个 `` 要求被切换到的组件都有自己的名字，不论是通过组件的 `name` 选项还是局部/全局注册。

你可以在 API 参考文档 查阅更多关于 `<keep-alive>` 的细节。

异步组件

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，Vue 允许你以一个工厂函数的方式定义你的组件，这个工厂函数会异步解析你的组件定义。Vue 只有在这个组件需要被渲染的时候才会触发该工厂函数，且会把结果缓存起来供未来重渲染。例如：

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    // 向 `resolve` 回调传递组件定义
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

如你所见，这个工厂函数会收到一个 `resolve` 回调，这个回调函数会在你从服务器得到组件定义的时候被调用。你也可以调用 `reject(reason)` 来表示加载失败。这里的 `setTimeout` 是为了演示用的，如何获取组件取决于你自己。一个推荐的做法是将异步组件和 webpack 的 code-splitting 功能一起配合使用：

```
Vue.component('async-webpack-example', function (resolve) {
  // 这个特殊的 `require` 语法将会告诉 webpack
  // 自动将你的构建代码切割成多个包，这些包
  // 会通过 Ajax 请求加载
  require(['./my-async-component'], resolve)
})
```

你也可以在工厂函数中返回一个 `Promise` ，所以把 webpack 2 和 ES2015 语法加在一起，我们可以写成这样：

```
Vue.component(
  'async-webpack-example',
  // 这个 `import` 函数会返回一个 `Promise` 对象。
  () => import('./my-async-component')
)
```

当使用局部注册的时候，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')
  }
})
```

如果你是一个 Browserify 用户同时喜欢使用异步组件，很不幸这个工具的作者[明确表示](https://github.com/substack/node-browserify/issues/58#issuecomment-21978224)异步加载“并不会被 Browserify 支持”，至少官方不会。Browserify 社区已经找到了[一些变通方案](https://github.com/vuejs/vuejs.org/issues/620)，这些方案可能会对已存在的复杂应用有帮助。对于其它的

场景，我们推荐直接使用 webpack，以拥有内建的被作为第一公民的异步支持。

处理加载状态

> 2.3.0+ 新增

这里的异步组件工厂函数也可以返回一个如下格式的对象：

```
const AsyncComponent = () => ({
  // 需要加载的组件 (应该是一个 `Promise` 对象)
  component: import('./MyComponent.vue'),
  // 异步组件加载时使用的组件
  loading: LoadingComponent,
  // 加载失败时使用的组件
  error: ErrorComponent,
  // 展示加载时组件的延时时间。默认值是 200 (毫秒)
  delay: 200,
  // 如果提供了超时时间且组件加载也超时了，
  // 则使用加载失败时使用的组件。默认值是：`Infinity`
  timeout: 3000
})
```

> 注意如果你希望在 Vue Router 的路由组件中使用上述语法的话，你必须使用 Vue Router 2.4.0+ 版本。

处理边界情况

> 该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

这里记录的都是和处理边界情况有关的功能，即一些需要对 Vue 的规则做一些小调整的特殊情况。不过注意这些功能都是有劣势或危险的场景的。我们会在每个案例中注明，所以当你使用每个功能的时候请稍加留意。

访问元素 & 组件

在绝大多数情况下，我们最好不要触达另一个组件实例内部或手动操作 DOM 元素。不过也确实在一些情况下做这些事情是合适的。

访问根实例

在每个 `new Vue` 实例的子组件中，其根实例可以通过 `$root` 属性进行访问。例如，在这个根实例中：

```
// Vue 根实例
new Vue({
  data: {
    foo: 1
  },
  computed: {
    bar: function () { /* ... */ }
  },
  methods: {
    baz: function () { /* ... */ }
  }
})
```

所有的子组件都可以将这个实例作为一个全局 store 来访问或使用。

```
// 获取根组件的数据
this.$root.foo

// 写入根组件的数据
this.$root.foo = 2

// 访问根组件的计算属性
this.$root.bar

// 调用根组件的方法
this.$root.baz()
```

对于 demo 或非常小型的有少量组件的应用来说这是很方便的。不过这个模式扩展到中大型应用来说就不然了。

因此在绝大多数情况下，我们强烈推荐使用 Vuex 来管理应用的状态。

访问父级组件实例

和 `$root` 类似， `$parent` 属性可以用来从一个子组件访问父组件的实例。它提供了一种机会，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。

在绝大多数情况下，触达父级组件会使得你的应用更难调试和理解，尤其是当你变更了父级组件的数据的时候。当我们稍后回看那个组件的时候，很难找出那个变更是从哪里发起的。

另外在一些可能适当的时候，你需要特别地共享一些组件库。举个例子，在和 JavaScript API 进行交互而不渲染 HTML 的抽象组件内，诸如这些假设性的 Google 地图组件一样：

```
<google-map>
  <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
</google-map>
```

这个 `<google-map>` 组件可以定义一个 `map` 属性，所有的子组件都需要访问它。在这种情况下 `<google-map-markers>` 可能想要通过类似 `this.$parent.getMap` 的方式访问那个地图，以便为其添加一组标记。你可以在这里查阅这种模式。

请留意，尽管如此，通过这种模式构建出来的那个组件的内部仍然是容易出现问题的。比如，设想一下我们添加一个新的 `<google-map-region>` 组件，当 `<google-map-markers>` 在其内部出现的时候，只会渲染那个区域内的标记：

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

那么在 `<google-map-markers>` 内部你可能发现自己需要一些类似这样的 hack：

```
var map = this.$parent.map || this.$parent.$parent.map
```

很快它就会失控。这也是我们针对需要向任意更深层级的组件提供上下文信息时推荐依赖注入的原因。

访问子组件实例或子元素

尽管存在 prop 和事件，有的时候你仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，你可以通过 `ref` 特性为这个子组件赋予一个 ID 引用。例如：

```
<base-input ref="usernameInput"></base-input>
```

现在在你已经定义了这个 `ref` 的组件里，你可以使用：

```
this.$refs.usernameInput
```

来访问这个 `<base-input>` 实例，以便不时之需。比如程序化地从一个父级组件聚焦这个输入框。在刚才那个例子中，该 `<base-input>` 组件也可以使用一个类似的 `ref` 提供对内部这个指定元素的访问，例如：

```
<input ref="input">
```

甚至可以通过其父级组件定义方法：

```
methods: {
  // 用来从父级组件聚焦输入框
  focus: function () {
    this.$refs.input.focus()
  }
}
```

这样就允许父级组件通过下面的代码聚焦 `<base-input>` 里的输入框：

```
this.$refs.usernameInput.focus()
```

当 `ref` 和 `v-for` 一起使用的时候，你得到的引用将会是一个包含了对应数据源的这些子组件的数组。

`$refs` 只会在组件渲染完成之后生效，并且它们不是响应式的。这只意味着一个直接的子组件封装的“逃生舱”——你应该避免在模板或计算属性中访问 `$refs` 。

依赖注入

在此之前，在我们描述访问父级组件实例的时候，展示过一个类似这样的例子：

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

在这个组件里，所有 `<google-map>` 的后代都需要访问一个 `getMap` 方法，以便知道要跟那个地图进行交互。不幸的是，使用 `$parent` 属性无法很好的扩展到更深层级的嵌套组件上。这也是依赖注入的用武之地，它用到了两个新的实例选项： `provide` 和 `inject` 。

`provide` 选项允许我们指定我们想要提供给后代组件的数据/方法。在这个例子中，就是

`<google-map>` 内部的 `getMap` 方法：

```
provide: function () {
  return {
    getMap: this.getMap
  }
}
```

然后在任何后代组件里，我们都可以使用 `inject` 选项来接收指定的我们想要添加在这个实例上的属性：

```
inject: ['getMap']
```

你可以在这里看到完整的示例。相比 `$parent` 来说，这个用法可以让我们在任意后代组件中访问 `getMap`，而不需要暴露整个 `<google-map>` 实例。这允许我们更好的持续研发该组件，而不需要担心我们可能会改变/移除一些子组件依赖的东西。同时这些组件之间的接口是始终明确定义的，就和 `props` 一样。

实际上，你可以把依赖注入看作一部分“大范围有效的 prop”，除了：

- 祖先组件不需要知道哪些后代组件使用它提供的属性
- 后代组件不需要知道被注入的属性来自哪里

然而，依赖注入还是有负面影响的。它将你的应用以目前的组件组织方式耦合了起来，使重构变得更加困难。同时所提供的属性是非响应式的。这是出于设计的考虑，因为使用它们来创建一个中心化规模化的数据跟使用 `$root` 做这件事都是不够好的。如果你想要共享的这个属性是你的应用特有的，而不是通用化的，或者如果你想在祖先组件中更新所提供的数据，那么这意味着你可能需要换用一个像 Vuex 这样真正的状态管理方案了。

你可以在 API 参考文档学习更多关于依赖注入的知识。

程序化的事件侦听器

现在，你已经知道了 `$emit` 的用法，它可以被 `v-on` 侦听，但是 Vue 实例同时在其事件接口中提供了其它的方法。我们可以：

- 通过 `$on(eventName, eventHandler)` 侦听一个事件
- 通过 `$once(eventName, eventHandler)` 一次性侦听一个事件
- 通过 `$off(eventName, eventHandler)` 停止侦听一个事件

你通常不会用到这些，但是当你需要在一个组件实例上手动侦听事件时，它们是派得上用场的。它们也可以用于代码组织工具。例如，你可能经常看到这种集成一个第三方库的模式：

```
// 一次性将这个日期选择器附加到一个输入框上
```

```
// 它会被挂载到 DOM 上。
mounted: function () {
  // Pikaday 是一个第三方日期选择器的库
  this.picker = new Pikaday({
    field: this.$refs.input,
    format: 'YYYY-MM-DD'
  })
},
// 在组件被销毁之前，
// 也销毁这个日期选择器。
beforeDestroy: function () {
  this.picker.destroy()
}
```

这里有两个潜在的问题：

- 它需要在这个组件实例中保存这个 `picker` ，如果可以的话最好只有生命周期钩子可以访问到它。这并不算严重的问题，但是它可以被视为杂物。
- 我们的建立代码独立于我们的清理代码，这使得我们比较难于程序化地清理我们建立的所有东西。

你应该通过一个程序化的侦听器解决这两个问题：

```
mounted: function () {
  var picker = new Pikaday({
    field: this.$refs.input,
    format: 'YYYY-MM-DD'
  })

  this.$once('hook:beforeDestroy', function () {
    picker.destroy()
  })
}
```

使用了这个策略，我甚至可以让多个输入框元素同时使用不同的 Pikaday，每个新的实例都程序化地在后期清理它自己：

```
mounted: function () {
  this.attachDatepicker('startDateInput')
  this.attachDatepicker('endDateInput')
},
methods: {
  attachDatepicker: function (refName) {
    var picker = new Pikaday({
      field: this.$refs[refName],
      format: 'YYYY-MM-DD'
    })
```

```
    this.$once('hook:beforeDestroy', function () {
      picker.destroy()
    })
  }
}
```

查阅这个 fiddle 可以了解到完整的代码。注意，即便如此，如果你发现自己不得不在单个组件里做很多建立和清理的工作，最好的方式通常还是创建更多的模块化组件。在这个例子中，我们推荐创建一个可复用的 `<input-datepicker>` 组件。

想了解更多程序化侦听器的内容，请查阅实例方法 / 事件相关的 API。

注意 Vue 的事件系统不同于浏览器的 EventTarget API。尽管它们工作起来是相似的，但是 `$emit` 、 `$on` ，和 `$off` 并不是 `dispatchEvent` 、 `addEventListener` 和 `removeEventListener` 的别名。

循环引用

递归组件

组件是可以在它们自己的模板中调用自身的。不过它们只能通过 `name` 选项来做这件事：

```
name: 'unique-name-of-my-component'
```

当你使用 `Vue.component` 全局注册一个组件时，这个全局的 ID 会自动设置为该组件的 `name` 选项。

```
Vue.component('unique-name-of-my-component', {
  // ...
})
```

稍有不慎，递归组件就可能导致无限循环：

```
name: 'stack-overflow',
template: '<div><stack-overflow></stack-overflow></div>'
```

类似上述的组件将会导致“max stack size exceeded”错误，所以请确保递归调用是条件性的 (例如使用一个最终会得到 `false` 的 `v-if`)。

组件之间的循环引用

假设你需要构建一个文件目录树，像访达或资源管理器那样的。你可能有一个 `<tree-folder>` 组件，模板是这样的：

```
<p>
  <span>{{ folder.name }}</span>
  <tree-folder-contents :children="folder.children"/>
</p>
```

还有一个 `<tree-folder-contents>` 组件，模板是这样的：

```
<ul>
  <li v-for="child in children">
    <tree-folder v-if="child.children" :folder="child"/>
    <span v-else>{{ child.name }}</span>
  </li>
</ul>
```

当你仔细观察的时候，你会发现这些组件在渲染树中互为对方的后代和祖先——一个悖论！当通过 `Vue.component` 全局注册组件的时候，这个悖论会被自动解开。如果你是这样做的，那么你可以跳过这里。

然而，如果你使用一个模块系统依赖/导入组件，例如通过 webpack 或 Browserify，你会遇到一个错误：

```
Failed to mount component: template or render function not defined.
```

为了解释这里发生了什么，我们先把两个组件称为 A 和 B。模块系统发现它需要 A，但是首先 A 依赖 B，但是 B 又依赖 A，但是 A 又依赖 B，如此往复。这变成了一个循环，不知道如何不经过其中一个组件而完全解析出另一个组件。为了解决这个问题，我们需要给模块系统一个点，在那里“A 反正是需要 B 的，但是我们不需要先解析 B。”

在我们的例子中，把 `<tree-folder>` 组件设为了那个点。我们知道那个产生悖论的子组件是 `<tree-folder-contents>` 组件，所以我们会等到生命周期钩子 `beforeCreate` 时去注册它：

```
beforeCreate: function () {
  this.$options.components.TreeFolderContents = require('./tree-folder-contents
.vue').default
}
```

或者，在本地注册组件的时候，你可以使用 webpack 的异步 `import`：

```
components: {
  TreeFolderContents: () => import('./tree-folder-contents.vue')
}
```

这样问题就解决了！

模板定义的替代品

内联模板

当 `inline-template` 这个特殊的特性出现在一个子组件上时，这个组件将会使用其里面的内容作为模板，而不是将其作为被分发的内容。这使得模板的撰写工作更加灵活。

```
<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>
```

不过，`inline-template` 会让你模板的作用域变得更加难以理解。所以作为最佳实践，请在组件内优先选择 `template` 选项或 `.vue` 文件里的一个 `<template>` 元素来定义模板。

X-Templates

另一个定义模板的方式是在一个 `<script>` 元素中，并为其带上 `text/x-template` 的类型，然后通过一个 id 将模板引用过去。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'
})
```

这些可以用于模板特别大的 demo 或极小型的应用，但是其它情况下请避免使用，因为这会将模板和该组件的其它定义分离开。

控制更新

感谢 Vue 的响应式系统，它始终知道何时进行更新 (如果你用对了的话)。不过还是有一些边界情况，你想要强制更新，尽管表面上看响应式的数据没有发生改变。也有一些情况是你想阻止不必要的更新。

强制更新

如果你发现你自己需要在 Vue 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事。
你可能还没有留意到数组或对象的变更检测注意事项，或者你可能依赖了一个未被 Vue 的响应式系统追踪的状态。

然而，如果你已经做到了上述的事项仍然发现在极少数的情况下需要手动强制更新，那么你可以通过 `$forceUpdate` 来做这件事。

通过 `v-once` 创建低开销的静态组件

渲染普通的 HTML 元素在 Vue 中是非常快速的，但有的时候你可能有一个组件，这个组件包含了大量静态内容。在这种情况下，你可以在根元素上添加 `v-once` 特性以确保这些内容只计算一次然后缓存起来，就像这样：

```
Vue.component('terms-of-service', {
  template: `
    <div v-once>
      <h1>Terms of Service</h1>
      ... a lot of static content ...
    </div>
  `
})
```

再说一次，试着不要过度使用这个模式。当你需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非你非常留意渲染变慢了，不然它完全是没有必要的——再加上它在后期会带来很多困惑。例如，设想另一个开发者并不熟悉 `v-once` 或漏看了它在模板中，他们可能会花很多个小时去找出模板为什么无法正确更新。

过渡 & 动画

进入/离开 & 列表过渡

概述

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。

包括以下工具：

- 在 CSS 过渡和动画中自动应用 class
- 可以配合使用第三方 CSS 动画库，如 Animate.css
- 在过渡钩子函数中使用 JavaScript 直接操作 DOM
- 可以配合使用第三方 JavaScript 动画库，如 Velocity.js

在这里，我们只会讲到进入、离开和列表的过渡，你也可以看下一节的 管理过渡状态。

单元素/组件的过渡

Vue 提供了 `transition` 的封装组件，在下列情形中，可以给任何元素和组件添加进入/离开过渡

- 条件渲染 (使用 `v-if`)
- 条件展示 (使用 `v-show`)
- 动态组件
- 组件根节点

这里是一个典型的例子：

```
<div id="demo">
  <button v-on:click="show = !show">
    Toggle
  </button>
  <transition name="fade">
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#demo',
  data: {
    show: true
  }
})
```

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to /* .fade-leave-active below version 2.1.8 */ {
  opacity: 0;
}
```

{% raw %}

hello

{% endraw %}

当插入或删除包含在 `transition` 组件中的元素时，Vue 将会做以下处理：

1. 自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。
2. 如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。
3. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作 (插入/删除) 在下一帧中立即执行。(注意：此指浏览器逐帧动画机制，和 Vue 的 `nextTick` 概念不同)

过渡的类名

在进入/离开的过渡中，会有 6 个 class 切换。

1. `v-enter` ：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
2. `v-enter-active` ：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。
3. `v-enter-to` : 2.1.8版及以上 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 `v-enter` 被移除)，在过渡/动画完成之后移除。
4. `v-leave` : 定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。
5. `v-leave-active` ：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。
6. `v-leave-to` : 2.1.8版及以上 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 `v-leave` 被删除)，在过渡/动画完成之后移除。

对于这些在过渡中切换的类名来说，如果你使用一个没有名字的 `<transition>` ，则 `v-` 是这些类名的默认前缀。如果你使用了 `<transition name="my-transition">` ，那么 `v-enter` 会替换为 `my-transition-enter` 。

`v-enter-active` 和 `v-leave-active` 可以控制进入/离开过渡的不同的缓和曲线，在下面章节会有个示例说明。

CSS 过渡

常用的过渡都是使用 CSS 过渡。

下面是一个简单例子：

```
<div id="example-1">
  <button @click="show = !show">
    Toggle render
  </button>
  <transition name="slide-fade">
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-1',
  data: {
    show: true
  }
})
```

```
/* 可以设置不同的进入和离开动画 */
/* 设置持续时间和动画函数 */
.slide-fade-enter-active {
  transition: all .3s ease;
}
.slide-fade-leave-active {
  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}
.slide-fade-enter, .slide-fade-leave-to
/* .slide-fade-leave-active for below version 2.1.8 */ {
  transform: translateX(10px);
  opacity: 0;
}
```

{% raw %}

hello

{% endraw %}

CSS 动画

CSS 动画用法同 CSS 过渡，区别是在动画中 `v-enter` 类名在节点插入 DOM 后不会立即删除，而是在 `animationend` 事件触发时删除。

示例：(省略了兼容性前缀)

```
<div id="example-2">
```

```
  <button @click="show = !show">Toggle show</button>
  <transition name="bounce">
    <p v-if="show">Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mau
ris facilisis enim libero, at lacinia diam fermentum id. Pellentesque habitant
morbi tristique senectus et netus.</p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-2',
  data: {
    show: true
  }
})
```

```
.bounce-enter-active {
  animation: bounce-in .5s;
}
.bounce-leave-active {
  animation: bounce-in .5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(1);
  }
}
```

{% raw %}

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi tristique senectus et netus.

{% endraw %}

自定义过渡的类名

我们可以通过以下特性来自定义过渡类名：

- `enter-class`
- `enter-active-class`
- `enter-to-class` (2.1.8+)

- `leave-class`
- `leave-active-class`
- `leave-to-class` (2.1.8+)

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如 Animate.css 结合使用十分有用。

示例：

```
<link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" ty
pe="text/css">

<div id="example-3">
  <button @click="show = !show">
    Toggle render
  </button>
  <transition
    name="custom-classes-transition"
    enter-active-class="animated tada"
    leave-active-class="animated bounceOutRight"
  >
    <p v-if="show">hello</p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-3',
  data: {
    show: true
  }
})
```

{% raw %}

hello

{% endraw %}

同时使用过渡和动画

Vue 为了知道过渡的完成，必须设置相应的事件监听器。它可以是 `transitionend` 或 `animationend`，这取决于给元素应用的 CSS 规则。如果你使用其中任何一种，Vue 能自动识别类型并设置监听。

但是，在一些场景中，你需要给同一个元素同时设置两种过渡动效，比如 `animation` 很快的被触发并完成了，而 `transition` 效果还没结束。在这种情况中，你就需要使用 `type` 特性并设置 `animation` 或 `transition` 来明确声明你需要 Vue 监听的类型。

显性的过渡持续时间

> 2.2.0 新增

在很多情况下，Vue 可以自动得出过渡效果的完成时机。默认情况下，Vue 会等待其在过渡效果的根元素的第一个 `transitionend` 或 `animationend` 事件。然而也可以不这样设定——比如，我们可以拥有一个精心编排的一系列过渡效果，其中一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

在这种情况下你可以用 `<transition>` 组件上的 `duration` 属性定制一个显性的过渡持续时间 (以毫秒计)：

```
<transition :duration="1000">...</transition>
```

你也可以定制进入和移出的持续时间：

```
<transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

JavaScript 钩子

可以在属性中声明 JavaScript 钩子

```
<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"

  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>
```

```
// ...
methods: {
  // --------
  // 进入中
  // --------

  beforeEnter: function (el) {
    // ...
  },
```

```
    // 当与 CSS 结合使用时
    // 回调函数 done 是可选的
    enter: function (el, done) {
      // ...
      done()
    },
    afterEnter: function (el) {
      // ...
    },
    enterCancelled: function (el) {
      // ...
    },

    // --------
    // 离开时
    // --------

    beforeLeave: function (el) {
      // ...
    },
    // 当与 CSS 结合使用时
    // 回调函数 done 是可选的
    leave: function (el, done) {
      // ...
      done()
    },
    afterLeave: function (el) {
      // ...
    },
    // leaveCancelled 只用于 v-show 中
    leaveCancelled: function (el) {
      // ...
    }
  }
```

这些钩子函数可以结合 CSS `transitions/animations` 使用，也可以单独使用。

当只用 JavaScript 过渡的时候，**在 `enter` 和 `leave` 中必须使用 `done` 进行回调**。否则，它们将被同步调用，过渡会立即完成。

推荐对于仅使用 JavaScript 过渡的元素添加 `v-bind:css="false"`，Vue 会跳过 CSS 的检测。这也可以避免过渡过程中 CSS 的影响。

一个使用 Velocity.js 的简单例子：

```
<!--
Velocity 和 jQuery.animate 的工作方式类似，也是用来实现 JavaScript 动画的一个很棒的选择
-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>
```

```
<div id="example-4">
  <button @click="show = !show">
    Toggle
  </button>
  <transition
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
    v-bind:css="false"
  >
    <p v-if="show">
      Demo
    </p>
  </transition>
</div>
```

```
new Vue({
  el: '#example-4',
  data: {
    show: false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
      el.style.transformOrigin = 'left'
    },
    enter: function (el, done) {
      Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })
      Velocity(el, { fontSize: '1em' }, { complete: done })
    },
    leave: function (el, done) {
      Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })
      Velocity(el, { rotateZ: '100deg' }, { loop: 2 })
      Velocity(el, {
        rotateZ: '45deg',
        translateY: '30px',
        translateX: '30px',
        opacity: 0
      }, { complete: done })
    }
  }
})
```

{% raw %}

Demo

{% endraw %}

初始渲染的过渡

可以通过 `appear` 特性设置节点在初始渲染的过渡

```
<transition appear>
  <!-- ... -->
</transition>
```

这里默认和进入/离开过渡一样，同样也可以自定义 CSS 类名。

```
<transition
  appear
  appear-class="custom-appear-class"
  appear-to-class="custom-appear-to-class" (2.1.8+)
  appear-active-class="custom-appear-active-class"
>
  <!-- ... -->
</transition>
```

自定义 JavaScript 钩子：

```
<transition
  appear
  v-on:before-appear="customBeforeAppearHook"
  v-on:appear="customAppearHook"
  v-on:after-appear="customAfterAppearHook"
  v-on:appear-cancelled="customAppearCancelledHook"
>
  <!-- ... -->
</transition>
```

多个元素的过渡

我们之后讨论多个组件的过渡，对于原生标签可以使用 `v-if` / `v-else` 。最常见的多标签过渡是一个列表和描述这个列表为空消息的元素：

```
<transition>
  <table v-if="items.length > 0">
    <!-- ... -->
  </table>
  <p v-else>Sorry, no items found.</p>
</transition>
```

可以这样使用，但是有一点需要注意：

当有**相同标签名**的元素切换时，需要通过 `key` 特性设置唯一的值来标记以让 Vue 区分它们，否则 Vue 为了效率只会替换相同标签内部的内容。即使在技术上没有必要，**给在 `` 组件中的多个元素设置 key 是一个更好的实践。**

示例：

```
<transition>
  <button v-if="isEditing" key="save">
    Save
  </button>
  <button v-else key="edit">
    Edit
  </button>
</transition>
```

在一些场景中，也可以通过给同一个元素的 `key` 特性设置不同的状态来代替 `v-if` 和 `v-else` ，上面的例子可以重写为：

```
<transition>
  <button v-bind:key="isEditing">
    {{ isEditing ? 'Save' : 'Edit' }}
  </button>
</transition>
```

使用多个 `v-if` 的多个元素的过渡可以重写为绑定了动态属性的单个元素过渡。例如：

```
<transition>
  <button v-if="docState === 'saved'" key="saved">
    Edit
  </button>
  <button v-if="docState === 'edited'" key="edited">
    Save
  </button>
  <button v-if="docState === 'editing'" key="editing">
    Cancel
  </button>
</transition>
```

可以重写为：

```
<transition>
  <button v-bind:key="docState">
    {{ buttonMessage }}
  </button>
</transition>
```

```
// ...
computed: {
  buttonMessage: function () {
    switch (this.docState) {
      case 'saved': return 'Edit'
      case 'edited': return 'Save'
      case 'editing': return 'Cancel'
    }
  }
}
```

过渡模式

这里还有一个问题，试着点击下面的按钮：

{% raw %}
{% endraw %}
在 "on" 按钮和 "off" 按钮的过渡中，两个按钮都被重绘了，一个离开过渡的时候另一个开始进入过渡。这是 `<transition>` 的默认行为 - 进入和离开同时发生。

在元素绝对定位在彼此之上的时候运行正常：

{% raw %}
{% endraw %}
然后，我们加上 translate 让它们运动像滑动过渡：

{% raw %}
{% endraw %}
同时生效的进入和离开的过渡不能满足所有要求，所以 Vue 提供了 过渡模式

- `in-out` ：新元素先进行过渡，完成之后当前元素过渡离开。
- `out-in` ：当前元素先进行过渡，完成之后新元素过渡进入。

用 `out-in` 重写之前的开关按钮过渡：

```
<transition name="fade" mode="out-in">
  <!-- ... the buttons ... -->
</transition>
```

{% raw %}
{% endraw %}
只用添加一个简单的特性，就解决了之前的过渡问题而无需任何额外的代码。

`in-out` 模式不是经常用到，但对于一些稍微不同的过渡效果还是有用的。

将之前滑动淡出的例子结合：

{% raw %}

{% endraw %}
很酷吧？

多个组件的过渡

多个组件的过渡简单很多 - 我们不需要使用 `key` 特性。相反，我们只需要使用动态组件：

```
<transition name="component-fade" mode="out-in">
  <component v-bind:is="view"></component>
</transition>
```

```
new Vue({
  el: '#transition-components-demo',
  data: {
    view: 'v-a'
  },
  components: {
    'v-a': {
      template: '<div>Component A</div>'
    },
    'v-b': {
      template: '<div>Component B</div>'
    }
  }
})
```

```
.component-fade-enter-active, .component-fade-leave-active {
  transition: opacity .3s ease;
}
.component-fade-enter, .component-fade-leave-to
/* .component-fade-leave-active for below version 2.1.8 */ {
  opacity: 0;
}
```

{% raw %}
AB
{% endraw %}

列表过渡

目前为止，关于过渡我们已经讲到：

- 单个节点
- 同一时间渲染多个节点中的一个

那么怎么同时渲染整个列表，比如使用 `v-for` ？在这种场景中，使用 `<transition-group>` 组件。在我们深入例子之前，先了解关于这个组件的几个特点：

- 不同于 `<transition>` ，它会以一个真实元素呈现：默认为一个 `<span>` 。你也可以通过 `tag` 特性更换为其他元素。
- 过渡模式不可用，因为我们不再相互切换特有的元素。
- 内部元素 总是需要 提供唯一的 `key` 属性值。

列表的进入/离开过渡

现在让我们由一个简单的例子深入，进入和离开的过渡使用之前一样的 CSS 类名。

```
<div id="list-demo" class="demo">
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list" tag="p">
    <span v-for="item in items" v-bind:key="item" class="list-item">
      {{ item }}
    </span>
  </transition-group>
</div>
```

```
new Vue({
  el: '#list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
  }
})
```

```
.list-item {
  display: inline-block;
  margin-right: 10px;
}
```

```
.list-enter-active, .list-leave-active {
  transition: all 1s;
}
.list-enter, .list-leave-to
/* .list-leave-active for below version 2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
```

{% raw %}
{{ item }}
{% endraw %}

这个例子有个问题，当添加和移除元素的时候，周围的元素会瞬间移动到他们的新布局的位置，而不是平滑的过渡，我们下面会解决这个问题。

列表的排序过渡

`<transition-group>` 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 ** `v-move` 特性**，它会在元素的改变定位的过程中应用。像之前的类名一样，可以通过 `name` 属性来自定义前缀，也可以通过 `move-class` 属性手动设置。

`v-move` 对于设置过渡的切换时机和过渡曲线非常有用，你会看到如下的例子：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min
.js"></script>

<div id="flip-list-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <transition-group name="flip-list" tag="ul">
    <li v-for="item in items" v-bind:key="item">
      {{ item }}
    </li>
  </transition-group>
</div>
```

```
new Vue({
  el: '#flip-list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9]
  },
  methods: {
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})
```

```
.flip-list-move {
  transition: transform 1s;
}
```

{% raw %}

- {{ item }}

{% endraw %}

这个看起来很神奇，内部的实现，Vue 使用了一个叫 FLIP 简单的动画队列

使用 transforms 将元素从之前的位置平滑过渡新的位置。

我们将之前实现的例子和这个技术结合，使我们列表的一切变动都会有动画过渡。

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min
.js"></script>

<div id="list-complete-demo" class="demo">
  <button v-on:click="shuffle">Shuffle</button>
  <button v-on:click="add">Add</button>
  <button v-on:click="remove">Remove</button>
  <transition-group name="list-complete" tag="p">
    <span
      v-for="item in items"
      v-bind:key="item"
      class="list-complete-item"
    >
      {{ item }}
    </span>
  </transition-group>
</div>
```

```
new Vue({
  el: '#list-complete-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
```

```
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})
```

```
.list-complete-item {
  transition: all 1s;
  display: inline-block;
  margin-right: 10px;
}
.list-complete-enter, .list-complete-leave-to
/* .list-complete-leave-active for below version 2.1.8 */ {
  opacity: 0;
  transform: translateY(30px);
}
.list-complete-leave-active {
  position: absolute;
}
```

{% raw %}

{{ item }}
{% endraw %}

需要注意的是使用 FLIP 过渡的元素不能设置为 `display: inline` 。作为替代方案，可以设置为 `display: inline-block` 或者放置于 flex 中

FLIP 动画不仅可以实现单列过渡，多维网格也同样可以过渡：

{% raw %}

Lazy Sudoku

Keep hitting the shuffle button until you win.

{{ cell.number }}
{% endraw %}

列表的交错过渡

通过 data 属性与 JavaScript 通信 ，就可以实现列表的交错过渡：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min
.js"></script>

<div id="staggered-list-demo">
  <input v-model="query">
  <transition-group
    name="staggered-fade"
    tag="ul"
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
```

```
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <li
      v-for="(item, index) in computedList"
      v-bind:key="item.msg"
      v-bind:data-index="index"
    >{{ item.msg }}</li>
  </transition-group>
</div>
```

```
new Vue({
  el: '#staggered-list-demo',
  data: {
    query: '',
    list: [
      { msg: 'Bruce Lee' },
      { msg: 'Jackie Chan' },
      { msg: 'Chuck Norris' },
      { msg: 'Jet Li' },
      { msg: 'Kung Fury' }
    ]
  },
  computed: {
    computedList: function () {
      var vm = this
      return this.list.filter(function (item) {
        return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1
      })
    }
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
      el.style.height = 0
    },
    enter: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
        Velocity(
          el,
          { opacity: 1, height: '1.6em' },
          { complete: done }
        )
      }, delay)
    },
    leave: function (el, done) {
      var delay = el.dataset.index * 150
      setTimeout(function () {
```

```
        Velocity(
          el,
          { opacity: 0, height: 0 },
          { complete: done }
        )
      }, delay)
    }
  }
})
```

{% raw %}

- {{ item.msg }}

{% endraw %}

可复用的过渡

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将 `<transition>` 或者 `<transition-group>` 作为根组件，然后将任何子组件放置在其中就可以了。

使用 template 的简单例子：

```
Vue.component('my-special-transition', {
  template: '\
    <transition\
      name="very-special-transition"\
      mode="out-in"\
      v-on:before-enter="beforeEnter"\
      v-on:after-enter="afterEnter"\
    >\
      <slot></slot>\
    </transition>\
  ',
  methods: {
    beforeEnter: function (el) {
      // ...
    },
    afterEnter: function (el) {
      // ...
    }
  }
})
```

函数组件更适合完成这个任务：

```
Vue.component('my-special-transition', {
  functional: true,
  render: function (createElement, context) {
```

```
    var data = {
      props: {
        name: 'very-special-transition',
        mode: 'out-in'
      },
      on: {
        beforeEnter: function (el) {
          // ...
        },
        afterEnter: function (el) {
          // ...
        }
      }
    }
    return createElement('transition', data, context.children)
  }
})
```

动态过渡

在 Vue 中即使是过渡也是数据驱动的！动态过渡最基本的例子是通过 `name` 特性来绑定动态值。

```
<transition v-bind:name="transitionName">
  <!-- ... -->
</transition>
```

当你想用 Vue 的过渡系统来定义的 CSS 过渡/动画 在不同过渡间切换会非常有用。

所有过渡特性都可以动态绑定，但我们不仅仅只有特性可以利用，还可以通过事件钩子获取上下文中的所有数据，因为事件钩子都是方法。这意味着，根据组件的状态不同，你的 JavaScript 过渡会有不同的表现。

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>

<div id="dynamic-fade-demo" class="demo">
  Fade In: <input type="range" v-model="fadeInDuration" min="0" v-bind:max="maxFadeDuration">
  Fade Out: <input type="range" v-model="fadeOutDuration" min="0" v-bind:max="maxFadeDuration">
  <transition
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <p v-if="show">hello</p>
  </transition>
```

```
  <button
    v-if="stop"
    v-on:click="stop = false; show = false"
  >Start animating</button>
  <button
    v-else
    v-on:click="stop = true"
  >Stop it!</button>
</div>
```

```
new Vue({
  el: '#dynamic-fade-demo',
  data: {
    show: true,
    fadeInDuration: 1000,
    fadeOutDuration: 1000,
    maxFadeDuration: 1500,
    stop: true
  },
  mounted: function () {
    this.show = false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 1 },
        {
          duration: this.fadeInDuration,
          complete: function () {
            done()
            if (!vm.stop) vm.show = false
          }
        }
      )
    },
    leave: function (el, done) {
      var vm = this
      Velocity(el,
        { opacity: 0 },
        {
          duration: this.fadeOutDuration,
          complete: function () {
            done()
            vm.show = true
          }
```

```
          }
        )
      }
    }
  })
```

最后，创建动态过渡的最终方案是组件通过接受 props 来动态修改之前的过渡。一句老话，唯一的限制是你的想象力。

状态过渡

Vue 的过渡系统提供了非常多简单的方法设置进入、离开和列表的动效。那么对于数据元素本身的动效呢，比如：

- 数字和运算
- 颜色的显示
- SVG 节点的位置
- 元素的大小和其他的属性

所有的原始数字都被事先存储起来，可以直接转换到数字。做到这一步，我们就可以结合 Vue 的响应式和组件系统，使用第三方库来实现切换元素的过渡状态。

状态动画与侦听器

通过侦听器我们能监听到任何数值属性的数值更新。可能听起来很抽象，所以让我们先来看看使用 GreenSock 一个例子：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/1.20.3/TweenMax.min.js"
></script>

<div id="animated-number-demo">
  <input v-model.number="number" type="number" step="20">
  <p>{{ animatedNumber }}</p>
</div>
```

```
new Vue({
  el: '#animated-number-demo',
  data: {
    number: 0,
    tweenedNumber: 0
  },
  computed: {
    animatedNumber: function() {
      return this.tweenedNumber.toFixed(0);
    }
  },
  watch: {
    number: function(newValue) {
      TweenLite.to(this.$data, 0.5, { tweenedNumber: newValue });
    }
  }
})
```

{% raw %}

{{ animatedNumber }}

{% endraw %}

当你把数值更新时，就会触发动画。这个是一个不错的演示，但是对于不能直接像数字一样存储的值，比如 CSS 中的 color 的值，通过下面的例子我们来通过 Tween.js 和 Color.js 实现一个例子：

```
<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>
<script src="https://cdn.jsdelivr.net/npm/color-js@1.0.3"></script>

<div id="example-7">
  <input
    v-model="colorQuery"
    v-on:keyup.enter="updateColor"
    placeholder="Enter a color"
  >
  <button v-on:click="updateColor">Update</button>
  <p>Preview:</p>
  <span
    v-bind:style="{ backgroundColor: tweenedCSSColor }"
    class="example-7-color-preview"
  ></span>
  <p>{{ tweenedCSSColor }}</p>
</div>
```

```
var Color = net.brehaut.Color

new Vue({
  el: '#example-7',
  data: {
    colorQuery: '',
    color: {
      red: 0,
      green: 0,
      blue: 0,
      alpha: 1
    },
    tweenedColor: {}
  },
  created: function () {
    this.tweenedColor = Object.assign({}, this.color)
  },
  watch: {
    color: function () {
      function animate () {
        if (TWEEN.update()) {
```

```
            requestAnimationFrame(animate)
          }
        }

        new TWEEN.Tween(this.tweenedColor)
          .to(this.color, 750)
          .start()

        animate()
      }
    },
    computed: {
      tweenedCSSColor: function () {
        return new Color({
          red: this.tweenedColor.red,
          green: this.tweenedColor.green,
          blue: this.tweenedColor.blue,
          alpha: this.tweenedColor.alpha
        }).toCSS()
      }
    },
    methods: {
      updateColor: function () {
        this.color = new Color(this.colorQuery).toRGB()
        this.colorQuery = ''
      }
    }
  })
```

```
.example-7-color-preview {
  display: inline-block;
  width: 50px;
  height: 50px;
}
```

{% raw %}

Preview:

{{ tweenedCSSColor }}

{% endraw %}

动态状态过渡

就像 Vue 的过渡组件一样，数据背后状态过渡会实时更新，这对于原型设计十分有用。当你修改一些变量，即使是一个简单的 SVG 多边形也可实现很多难以想象的效果。

(此处例子略，详见官方文档)

上述 demo 背后的代码可以通过这个 fiddle 进行详阅。

把过渡放到组件里

管理太多的状态过渡会很快的增加 Vue 实例或者组件的复杂性，幸好很多的动画可以提取到专用的子组件。

我们来将之前的示例改写一下：

```
<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>

<div id="example-8">
  <input v-model.number="firstNumber" type="number" step="20"> +
  <input v-model.number="secondNumber" type="number" step="20"> =
  {{ result }}
  <p>
    <animated-integer v-bind:value="firstNumber"></animated-integer> +
    <animated-integer v-bind:value="secondNumber"></animated-integer> =
    <animated-integer v-bind:value="result"></animated-integer>
  </p>
</div>
```

```
// 这种复杂的补间动画逻辑可以被复用
// 任何整数都可以执行动画
// 组件化使我们的界面十分清晰
// 可以支持更多更复杂的动态过渡
// 策略。
Vue.component('animated-integer', {
  template: '<span>{{ tweeningValue }}</span>',
  props: {
    value: {
      type: Number,
      required: true
    }
  },
  data: function () {
    return {
      tweeningValue: 0
    }
  },
  watch: {
    value: function (newValue, oldValue) {
      this.tween(oldValue, newValue)
    }
  },
  mounted: function () {
    this.tween(0, this.value)
  },
  methods: {
```

```
    tween: function (startValue, endValue) {
      var vm = this
      function animate () {
        if (TWEEN.update()) {
          requestAnimationFrame(animate)
        }
      }

      new TWEEN.Tween({ tweeningValue: startValue })
        .to({ tweeningValue: endValue }, 500)
        .onUpdate(function (object) {
          vm.tweeningValue = object.tweeningValue.toFixed(0)
        })
        .start()

      animate()
    }
  }
})

// 所有的复杂度都已经从 Vue 的主实例中移除！
new Vue({
  el: '#example-8',
  data: {
    firstNumber: 20,
    secondNumber: 40
  },
  computed: {
    result: function () {
      return this.firstNumber + this.secondNumber
    }
  }
})
```

我们能在组件中结合使用这一节讲到各种过渡策略和 Vue 内建的过渡系统。总之，对于完成各种过渡动效几乎没有阻碍。

赋予设计以生命

只要一个动画，就可以带来生命。不幸的是，当设计师创建图标、logo 和吉祥物的时候，他们交付的通常都是图片或静态的 SVG。所以，虽然 GitHub 的章鱼猫、Twitter 的小鸟以及其它许多 logo 类似于生灵，它们看上去实际上并不是活着的。

Vue 可以帮到你。因为 SVG 的本质是数据，我们只需要这些动物兴奋、思考或警戒的样例。然后 Vue 就可以辅助完成这几种状态之间的过渡动画，来制作你的欢迎页面、加载指示、以及更加带有情感的提示。

Sarah Drasner 展示了下面这个 demo，这个 demo 结合了时间和交互相关的状态改变：

查看 CodePen 上 Sarah Drasner (@sdras) 的例子 Vue-controlled Wall-E.

可复用性 & 组合

混入

基础

混入 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项。

例子：

```
// 定义一个混入对象
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// 定义一个使用混入对象的组件
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // => "hello from mixin!"
```

选项合并

当组件和混入对象含有同名选项时，这些选项将以恰当的方式混合。

比如，数据对象在内部会进行浅合并 (一层属性深度)，在和组件的数据发生冲突时以组件数据优先。

```
var mixin = {
  data: function () {
    return {
      message: 'hello',
      foo: 'abc'
    }
  }
}

new Vue({
  mixins: [mixin],
  data: function () {
    return {
```

```
      message: 'goodbye',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
    // => { message: "goodbye", foo: "abc", bar: "def" }
  }
})
```

同名钩子函数将混合为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用。

```
var mixin = {
  created: function () {
    console.log('混入对象的钩子被调用')
  }
}

new Vue({
  mixins: [mixin],
  created: function () {
    console.log('组件钩子被调用')
  }
})

// => "混入对象的钩子被调用"
// => "组件钩子被调用"
```

值为对象的选项，例如 `methods` ， `components` 和 `directives` ，将被混合为同一个对象。两个对象键名冲突时，取组件对象的键值对。

```
var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
```

```
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "from self"
```

注意： `Vue.extend()` 也使用同样的策略进行合并。

全局混入

也可以全局注册混入对象。注意使用！ 一旦使用全局混入对象，将会影响到 所有 之后创建的 Vue 实例。使用恰当时，可以为自定义对象注入处理逻辑。

```
// 为自定义的选项 'myOption' 注入一个处理器。
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// => "hello!"
```

谨慎使用全局混入对象，因为会影响到每个单独创建的 Vue 实例 (包括第三方模板)。大多数情况下，只应当应用于自定义选项，就像上面示例一样。也可以将其用作 [Plugins](plugins.html) 以避免产生重复应用

自定义选项合并策略

自定义选项将使用默认策略，即简单地覆盖已有值。如果想让自定义选项以自定义逻辑合并，可以向 `Vue.config.optionMergeStrategies` 添加一个函数：

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // return mergedVal
}
```

对于大多数对象选项，可以使用 `methods` 的合并策略：

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

更多高级的例子可以在 Vuex 的 1.x 混入策略里找到：

```
const merge = Vue.config.optionMergeStrategies.computed
Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {
  if (!toVal) return fromVal
  if (!fromVal) return toVal
  return {
    getters: merge(toVal.getters, fromVal.getters),
    state: merge(toVal.state, fromVal.state),
    actions: merge(toVal.actions, fromVal.actions)
  }
}
```

自定义指令

简介

除了核心功能默认内置的指令（ `v-model` 和 `v-show` ），Vue 也允许注册自定义指令。注意，在 Vue2.0 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通 DOM 元素进行底层操作，这时候就会用到自定义指令。举个聚焦输入框的例子，如下：
当页面加载时，该元素将获得焦点 (注意： `autofocus` 在移动版 Safari 上不工作)。事实上，只要你在打开这个页面后还没点击过任何内容，这个输入框就应当还是处于聚焦状态。现在让我们用指令来实现这个功能：

```
// 注册一个全局自定义指令 `v-focus`
Vue.directive('focus', {
  // 当被绑定的元素插入到 DOM 中时……
  inserted: function (el) {
    // 聚焦元素
    el.focus()
  }
})
```

如果想注册局部指令，组件中也接受一个 `directives` 的选项：

```
directives: {
  focus: {
    // 指令的定义
    inserted: function (el) {
      el.focus()
    }
  }
}
```

然后你可以在模板中任何元素上使用新的 `v-focus` 属性，如下：

```
<input v-focus>
```

钩子函数

一个指令定义对象可以提供如下几个钩子函数 (均为可选)：

- `bind` ：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
- `inserted` ：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。
- `update` ：所在组件的 VNode 更新时调用，但是可能发生在其子 VNode 更新之前。指令的值可能发生

了改变，也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新 (详细的钩子函数参数见下)。

- `componentUpdated` ：指令所在组件的 VNode 及其子 VNode 全部更新后调用。
- `unbind` ：只调用一次，指令与元素解绑时调用。

接下来我们来看一下钩子函数的参数 (即 `el` 、 `binding` 、 `vnode` 和 `oldVnode`)。

钩子函数参数

指令钩子函数会被传入以下参数：

- `el` ：指令所绑定的元素，可以用来直接操作 DOM 。
- `binding` ：一个对象，包含以下属性：
  - `name` ：指令名，不包括 `v-` 前缀。
  - `value` ：指令的绑定值，例如： `v-my-directive="1 + 1"` 中，绑定值为 `2` 。
  - `oldValue` ：指令绑定的前一个值，仅在 `update` 和 `componentUpdated` 钩子中可用。无论值是否改变都可用。
  - `expression` ：字符串形式的指令表达式。例如 `v-my-directive="1 + 1"` 中，表达式为 `"1 + 1"` 。
  - `arg` ：传给指令的参数，可选。例如 `v-my-directive:foo` 中，参数为 `"foo"` 。
  - `modifiers` ：一个包含修饰符的对象。例如： `v-my-directive.foo.bar` 中，修饰符对象为 `{ foo: true, bar: true }` 。
- `vnode` ：Vue 编译生成的虚拟节点。移步 VNode API 来了解更多详情。
- `oldVnode` ：上一个虚拟节点，仅在 `update` 和 `componentUpdated` 钩子中可用。

除了 `el` 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 [`dataset`](https://developer.mozilla.org/zh-CN/docs/Web/API/HTMLElement/dataset) 来进行。

这是一个使用了这些属性的自定义钩子样例：

```
<div id="hook-arguments-example" v-demo:foo.a.b="message"></div>
```

```
Vue.directive('demo', {
  bind: function (el, binding, vnode) {
    var s = JSON.stringify
    el.innerHTML =
      'name: '       + s(binding.name) + '<br>' +
      'value: '      + s(binding.value) + '<br>' +
      'expression: ' + s(binding.expression) + '<br>' +
      'argument: '   + s(binding.arg) + '<br>' +
      'modifiers: '  + s(binding.modifiers) + '<br>' +
```

```
      'vnode keys: ' + Object.keys(vnode).join(', ')
  }
})

new Vue({
  el: '#hook-arguments-example',
  data: {
    message: 'hello!'
  }
})
```

函数简写

在很多时候，你可能想在 `bind` 和 `update` 时触发相同行为，而不关心其它的钩子。比如这样写:

```
Vue.directive('color-swatch', function (el, binding) {
  el.style.backgroundColor = binding.value
})
```

对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法的 JavaScript 表达式。

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
```

```
Vue.directive('demo', function (el, binding) {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text)  // => "hello!"
})
```

渲染函数 & JSX

基础

Vue 推荐在绝大多数情况下使用 template 来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力，这时你可以用 render 函数，它比 template 更接近编译器。

让我们深入一个简单的例子，这个例子里 `render` 函数很实用。假设我们要生成锚点标题 (anchored headings)：

```
<h1>
  <a name="hello-world" href="#hello-world">
    Hello world!
  </a>
</h1>
```

对于上面的 HTML，我们决定这样定义组件接口：

```
<anchored-heading :level="1">Hello world!</anchored-heading>
```

当我们开始写一个只能通过 `level` prop 动态生成 heading 标签的组件时，你可能很快想到这样实现：

```
<script type="text/x-template" id="anchored-heading-template">
  <h1 v-if="level === 1">
    <slot></slot>
  </h1>
  <h2 v-else-if="level === 2">
    <slot></slot>
  </h2>
  <h3 v-else-if="level === 3">
    <slot></slot>
  </h3>
  <h4 v-else-if="level === 4">
    <slot></slot>
  </h4>
  <h5 v-else-if="level === 5">
    <slot></slot>
  </h5>
  <h6 v-else-if="level === 6">
    <slot></slot>
  </h6>
</script>
```

```
Vue.component('anchored-heading', {
  template: '#anchored-heading-template',
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

在这种场景中使用 template 并不是最好的选择：首先代码冗长，为了在不同级别的标题中插入锚点元素，我们需要重复地使用 `<slot></slot>` 。

虽然模板在大多数组件中都非常好用，但是在这里它就不是很简洁的了。那么，我们来尝试使用 `render` 函数重写上面的例子：

```
Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level,   // 标签名称
      this.$slots.default // 子元素数组
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

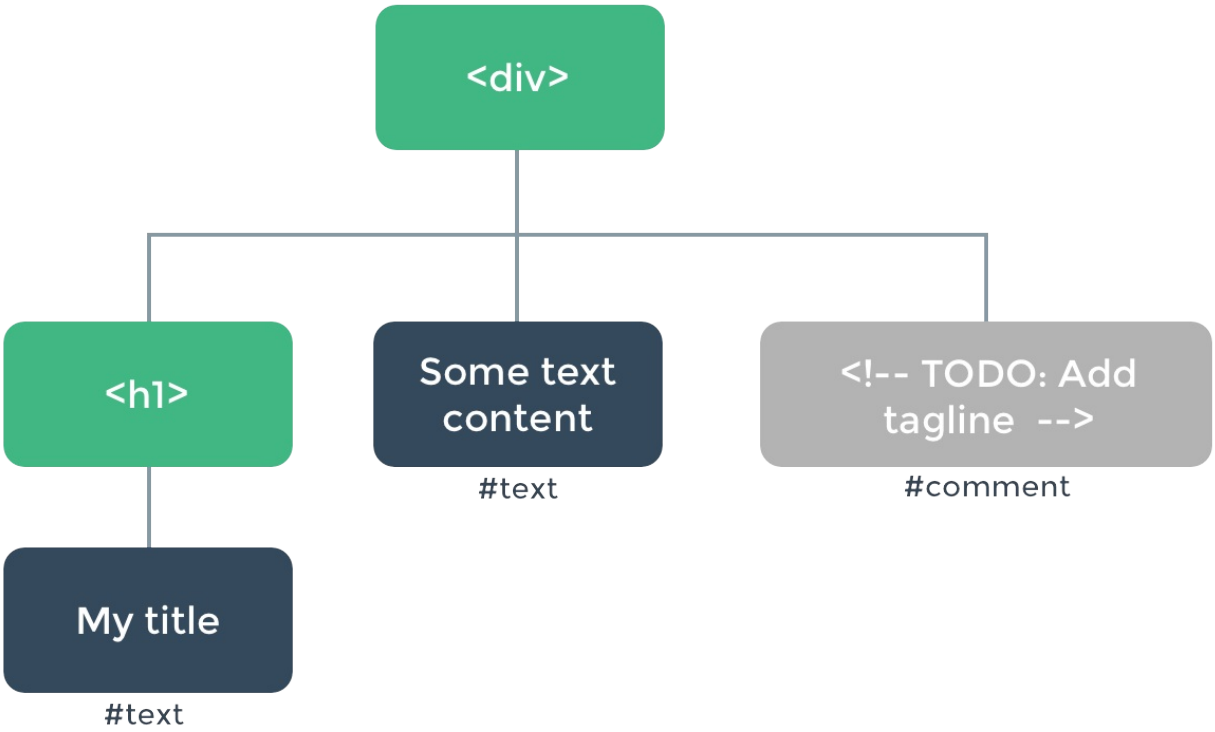
简单清晰很多！简单来说，这样代码精简很多，但是需要非常熟悉 Vue 的实例属性。在这个例子中，你需要知道，向组件中传递不带 `slot` 特性的子元素时，比如 `anchored-heading` 中的 `Hello world!` ，这些子元素被存储在组件实例中的 `$slots.default` 中。如果你还不了解，在深入 render 函数之前推荐阅读实例属性 API。

节点、树以及虚拟 DOM

在深入渲染函数之前，了解一些浏览器的工作原理是很重要的。以下面这段 HTML 为例：

```
<div>
  <h1>My title</h1>
  Some text content
  <!-- TODO: 添加标签行 -->
</div>
```

当浏览器读到这些代码时，它会建立一个“DOM 节点”树来保持追踪，如同你会画一张家谱树来追踪家庭成员的发展一样。
HTML 的 DOM 节点树如下图所示：



每个元素都是一个节点。每片文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点 (也就是说每个部分可以包含其它的一些部分)。
高效的更新所有这些节点会是比较困难的，不过所幸你不必再手动完成这个工作了。你只需要告诉 Vue 你希望页面上的 HTML 是什么，这可以是在一个模板里：

```
<h1>{{ blogTitle }}</h1>
```

或者一个渲染函数里：

```
render: function (createElement) {
  return createElement('h1', this.blogTitle)
}
```

在这两种情况下，Vue 都会自动保持页面的更新，即便 `blogTitle` 发生了改变。

虚拟 DOM

Vue 通过建立一个虚拟 DOM 对真实 DOM 发生的变化保持追踪。请仔细看这行代码：

```
return createElement('h1', this.blogTitle)
```

`createElement` 到底会返回什么呢？其实不是一个实际的 DOM 元素。它更准确的名字可能是 `createNodeDescription` ，因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点，及其子节点。我们把这样的节点描述为“虚拟节点 (Virtual Node)” ，也常简写它为“VNode” 。“虚拟 DOM”是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼。

`createElement` 参数

接下来你需要熟悉的是如何在 `createElement` 函数中生成模板。这里是 `createElement` 接受的参数：

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  // 一个 HTML 标签字符串，组件选项对象，或者
  // 解析上述任何一种的一个 async 异步函数。必需参数。
  'div',

  // {Object}
  // 一个包含模板相关属性的数据对象
  // 你可以在 template 中使用这些特性。可选参数。
  {
    // (详情见下一节)
  },

  // {String | Array}
  // 子虚拟节点 (VNodes)，由 `createElement()` 构建而成，
  // 也可以使用字符串来生成“文本虚拟节点”。可选参数。
  [
    '先写一些文字',
    createElement('h1', '一则头条'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
  ]
)
```

深入 data 对象

有一点要注意：正如在模板语法中， `v-bind:class` 和 `v-bind:style` ，会被特别对待一样，在 VNode 数据对象中，下列属性名是级别最高的字段。该对象也允许你绑定普通的 HTML 特性，就像 DOM 属性

一样，比如 `innerHTML` (这会取代 `v-html` 指令)。

```
{
  // 和`v-bind:class`一样的 API
  // 接收一个字符串、对象或字符串和对象组成的数组
  'class': {
    foo: true,
    bar: false
  },
  // 和`v-bind:style`一样的 API
  // 接收一个字符串、对象或对象组成的数组
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // 普通的 HTML 特性
  attrs: {
    id: 'foo'
  },
  // 组件 props
  props: {
    myProp: 'bar'
  },
  // DOM 属性
  domProps: {
    innerHTML: 'baz'
  },
  // 事件监听器基于 `on`
  // 所以不再支持如 `v-on:keyup.enter` 修饰器
  // 需要手动匹配 keyCode。
  on: {
    click: this.clickHandler
  },
  // 仅用于组件，用于监听原生事件，而不是组件内部使用
  // `vm.$emit` 触发的事件。
  nativeOn: {
    click: this.nativeClickHandler
  },
  // 自定义指令。注意，你无法对 `binding` 中的 `oldValue`
  // 赋值，因为 Vue 已经自动为你进行了同步。
  directives: [
    {
      name: 'my-custom-directive',
      value: '2',
      expression: '1 + 1',
      arg: 'foo',
      modifiers: {
        bar: true
      }
```

```
    }
  ],
  // 作用域插槽格式
  // { name: props => VNode | Array<VNode> }
  scopedSlots: {
    default: props => createElement('span', props.text)
  },
  // 如果组件是其他组件的子组件，需为插槽指定名称
  slot: 'name-of-slot',
  // 其他特殊顶层属性
  key: 'myKey',
  ref: 'myRef',
  // 如果你在渲染函数中向多个元素都应用了相同的 ref 名，
  // 那么 `$refs.myRef` 会变成一个数组。
  refInFor: true
}
```

完整示例

有了这些知识，我们现在可以完成我们最开始想实现的组件：

```
var getChildrenTextContent = function (children) {
  return children.map(function (node) {
    return node.children
      ? getChildrenTextContent(node.children)
      : node.text
  }).join('')
}

Vue.component('anchored-heading', {
  render: function (createElement) {
    // 创建 kebab-case 风格的ID
    var headingId = getChildrenTextContent(this.$slots.default)
      .toLowerCase()
      .replace(/\W+/g, '-')
      .replace(/(^\-|\-$)/g, '')

    return createElement(
      'h' + this.level,
      [
        createElement('a', {
          attrs: {
            name: headingId,
            href: '#' + headingId
          }
        }, this.$slots.default)
      ]
    )
  },
```

```
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

约束

VNodes 必须唯一

组件树中的所有 VNodes 必须是唯一的。这意味着，下面的 render function 是无效的：

```
render: function (createElement) {
  var myParagraphVNode = createElement('p', 'hi')
  return createElement('div', [
    // 错误-重复的 VNodes
    myParagraphVNode, myParagraphVNode
  ])
}
```

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现。例如，下面这个例子 render 函数完美有效地渲染了 20 个相同的段落：

```
render: function (createElement) {
  return createElement('div',
    Array.apply(null, { length: 20 }).map(function () {
      return createElement('p', 'hi')
    })
  )
}
```

使用 JavaScript 代替模板功能

`v-if` 和 `v-for`

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的 render 函数就不会提供专有的替代方法。比如，在 template 中使用的 `v-if` 和 `v-for`：

```
<ul v-if="items.length">
  <li v-for="item in items">{{ item.name }}</li>
</ul>
<p v-else>No items found.</p>
```

这些都会在 render 函数中被 JavaScript 的 `if` / `else` 和 `map` 重写：

```
props: ['items'],
render: function (createElement) {
  if (this.items.length) {
    return createElement('ul', this.items.map(function (item) {
      return createElement('li', item.name)
    }))
  } else {
    return createElement('p', 'No items found.')
  }
}
```

`v-model`

render 函数中没有与 `v-model` 的直接对应 - 你必须自己实现相应的逻辑：

```
props: ['value'],
render: function (createElement) {
  var self = this
  return createElement('input', {
    domProps: {
      value: self.value
    },
    on: {
      input: function (event) {
        self.$emit('input', event.target.value)
      }
    }
  })
}
```

这就是深入底层的代价，但与 `v-model` 相比，这可以让你更好地控制交互细节。

事件 & 按键修饰符

对于 `.passive` 、 `.capture` 和 `.once` 事件修饰符, Vue 提供了相应的前缀可以用于 `on` ：

Modifier(s)	Prefix
`.passive`	`&`
`.capture`	`!`
`.once`	`~`
`.capture.once` or `.once.capture`	`~!`

例如:

```
on: {
  '!click': this.doThisInCapturingMode,
  '~keyup': this.doThisOnce,
  '~!mouseover': this.doThisOnceInCapturingMode
}
```

对于其他的修饰符，前缀不是很重要，因为你可以在事件处理函数中使用事件方法：

Modifier(s)	Equivalent in Handler
`.stop`	`event.stopPropagation()`
`.prevent`	`event.preventDefault()`
`.self`	`if (event.target !== event.cu`
Keys: `.enter`, `.13`	`if (event.keyCode !== 13) ret` (change `13` to another key code for other key modifiers)
Modifiers Keys: `.ctrl`, `.alt`, `.shift`, `.meta`	`if (!event.ctrlKey) return` (change `ctrlKey` to `altKey`, `shiftKey`, or `metaKey`, respectively)

这里是一个使用所有修饰符的例子：

```
on: {
  keyup: function (event) {
    // 如果触发事件的元素不是事件绑定的元素
    // 则返回
    if (event.target !== event.currentTarget) return
    // 如果按下去的不是 enter 键或者
    // 没有同时按下 shift 键
    // 则返回
    if (!event.shiftKey || event.keyCode !== 13) return
    // 阻止 事件冒泡
    event.stopPropagation()
    // 阻止该元素默认的 keyup 事件
    event.preventDefault()
    // ...
  }
}
```

插槽

你可以通过 `this.$slots` 访问静态插槽的内容，得到的是一个 VNodes 数组：

```
render: function (createElement) {
  // `<div><slot></slot></div>`
  return createElement('div', this.$slots.default)
}
```

也可以通过 `this.$scopedSlots` 访问作用域插槽，得到的是一个返回 VNodes 的函数：

```
props: ['message'],
render: function (createElement) {
  // `<div><slot :text="message"></slot></div>`
  return createElement('div', [
    this.$scopedSlots.default({
      text: this.message
    })
  ])
}
```

如果要用渲染函数向子组件中传递作用域插槽，可以利用 VNode 数据对象中的 `scopedSlots` 域：

```
render: function (createElement) {
  return createElement('div', [
    createElement('child', {
      // 在数据对象中传递 `scopedSlots`
      // 格式：{ name: props => VNode | Array<VNode> }
      scopedSlots: {
        default: function (props) {
          return createElement('span', props.text)
        }
      }
    })
  ])
}
```

JSX

如果你写了很多 `render` 函数，可能会觉得下面这样的代码写起来很痛苦：

```
createElement(
  'anchored-heading', {
    props: {
      level: 1
    }
  }, [
```

```
    createElement('span', 'Hello'),
    ' world!'
  ]
)
```

特别是模板如此简单的情况下：

```
<anchored-heading :level="1">
  <span>Hello</span> world!
</anchored-heading>
```

这就是为什么会有一个 Babel 插件，用于在 Vue 中使用 JSX 语法，它可以让我们回到更接近于模板的语法上。

```
import AnchoredHeading from './AnchoredHeading.vue'

new Vue({
  el: '#demo',
  render: function (h) {
    return (
      <AnchoredHeading level={1}>
        <span>Hello</span> world!
      </AnchoredHeading>
    )
  }
})
```

将 `h` 作为 `createElement` 的别名是 Vue 生态系统中的一个通用惯例，实际上也是 JSX 所要求的，如果在作用域中 `h` 失去作用，在应用中会触发报错。

更多关于 JSX 映射到 JavaScript，阅读 使用文档。

函数式组件

之前创建的锚点标题组件是比较简单，没有管理或者监听任何传递给他的状态，也没有生命周期方法。它只是一个接收参数的函数。

在这个例子中，我们标记组件为 `functional`，这意味它是无状态 (没有响应式数据)，无实例 (没有 `this` 上下文)。

一个函数式组件就像这样：

```
Vue.component('my-component', {
  functional: true,
  // Props 可选
  props: {
    // ...
```

```
  },
  // 为了弥补缺少的实例
  // 提供第二个参数作为上下文
  render: function (createElement, context) {
    // ...
  }
})
```

> 注意：在 2.3.0 之前的版本中，如果一个函数式组件想要接受 props，则 `props` 选项是必须的。在 2.3.0 或以上的版本中，你可以省略 `props` 选项，所有组件上的特性都会被自动解析为 props。

在 2.5.0 及以上版本中，如果你使用了单文件组件，那么基于模板的函数式组件可以这样声明：

```
<template functional>
</template>
```

组件需要的一切都是通过上下文传递，包括：

- `props`：提供所有 prop 的对象
- `children`：VNode 子节点的数组
- `slots`：返回所有插槽的对象的函数
- `data`：传递给组件的数据对象，作为 `createElement` 的第二个参数传入组件
- `parent`：对父组件的引用
- `listeners`：(2.3.0+) 一个包含了所有在父组件上注册的事件侦听器的对象。这只是一个指向 `data.on` 的别名。
- `injections`：(2.3.0+) 如果使用了 `inject` 选项，则该对象包含了应当被注入的属性。

在添加 `functional: true` 之后，锚点标题组件的 render 函数之间简单更新增加 `context` 参数，`this.$slots.default` 更新为 `context.children`，之后 `this.level` 更新为 `context.props.level`。

因为函数式组件只是一个函数，所以渲染开销也低很多。然而，对持久化实例的缺乏也意味着函数式组件不会出现在 Vue devtools 的组件树里。

在作为包装组件时它们也同样非常有用，比如，当你需要做这些时：

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

下面是一个依赖传入 props 的值的 `smart-list` 组件例子，它能代表更多具体的组件：

```
var EmptyList = { /* ... */ }
```

```
var TableList = { /* ... */ }
var OrderedList = { /* ... */ }
var UnorderedList = { /* ... */ }

Vue.component('smart-list', {
  functional: true,
  props: {
    items: {
      type: Array,
      required: true
    },
    isOrdered: Boolean
  },
  render: function (createElement, context) {
    function appropriateListComponent () {
      var items = context.props.items

      if (items.length === 0)           return EmptyList
      if (typeof items[0] === 'object') return TableList
      if (context.props.isOrdered)      return OrderedList

      return UnorderedList
    }

    return createElement(
      appropriateListComponent(),
      context.data,
      context.children
    )
  }
})
```

向子元素或子组件传递特性和事件

在普通组件中，没有被定义为 prop 的特性会自动添加到组件的根元素上，将现有的同名特性替换或与其智能合并。

然而函数式组件要求你显式定义该行为：

```
Vue.component('my-functional-button', {
  functional: true,
  render: function (createElement, context) {
    // 完全透明的传入任何特性、事件监听器、子结点等。
    return createElement('button', context.data, context.children)
  }
})
```

向 `createElement` 通过传入 `context.data` 作为第二个参数，我们就把

`my-functional-button` 上面所有的特性和事件监听器都传递下去了。事实上这是非常透明的，那些事件甚至并不要求 `.native` 修饰符。

如果你使用基于模板的函数式组件，那么你还需要手动添加特性和监听器。因为我们可以访问到其独立的上下文内容，所以我们可以使用 `data.attrs` 传递任何 HTML 特性，也可以使用 `listeners` (即 *`data.on`* 的别名) 传递任何事件监听器。

```
<template functional>
  <button
    class="btn btn-primary"
    v-bind="data.attrs"
    v-on="listeners"
  >
    <slot/>
  </button>
</template>
```

`slots()` 和 `children` 对比

你可能想知道为什么同时需要 `slots()` 和 `children` 。 `slots().default` 不是和 `children` 类似的吗？在一些场景中，是这样，但是如果是函数式组件和下面这样的 children 呢？

```
<my-functional-component>
  <p slot="foo">
    first
  </p>
  <p>second</p>
</my-functional-component>
```

对于这个组件， `children` 会给你两个段落标签，而 `slots().default` 只会传递第二个匿名段落标签， `slots().foo` 会传递第一个具名段落标签。同时拥有 `children` 和 `slots()` ，因此你可以选择让组件通过 `slot()` 系统分发或者简单的通过 `children` 接收，让其他组件去处理。

模板编译

你可能有兴趣知道，Vue 的模板实际是编译成了 render 函数。这是一个实现细节，通常不需要关心，但如果你想看看模板的功能是怎样被编译的，你会发现会非常有趣。下面是一个使用 `Vue.compile` 来实时编译模板字符串的简单 demo：

插件

插件通常会为 Vue 添加全局功能。插件的范围没有限制——一般有下面几种：

1. 添加全局方法或者属性，如: vue-custom-element
2. 添加全局资源：指令/过滤器/过渡等，如 vue-touch
3. 通过全局 mixin 方法添加一些组件选项，如: vue-router
4. 添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。
5. 一个库，提供自己的 API，同时提供上面提到的一个或多个功能，如 vue-router

使用插件

通过全局方法 `Vue.use()` 使用插件。它需要在你调用 `new Vue()` 启动应用之前完成：

```
// 调用 `MyPlugin.install(Vue)`
Vue.use(MyPlugin)

new Vue({
  //... options
})
```

也可以传入一个选项对象：

```
Vue.use(MyPlugin, { someOption: true })
```

`Vue.use` 会自动阻止多次注册相同插件，届时只会注册一次该插件。

Vue.js 官方提供的一些插件 (例如 `vue-router`) 在检测到 `Vue` 是可访问的全局变量时会自动调用 `Vue.use()` 。然而在例如 CommonJS 的模块环境中，你应该始终显式地调用 `Vue.use()` ：

```
// 用 Browserify 或 webpack 提供的 CommonJS 模块环境时
var Vue = require('vue')
var VueRouter = require('vue-router')

// 不要忘了调用此方法
Vue.use(VueRouter)
```

awesome-vue 集合了来自社区贡献的数以千计的插件和库。

开发插件

Vue.js 的插件应该有一个公开方法 `install` 。这个方法的第一个参数是 `Vue` 构造器，第二个参数是一个

可选的选项对象：

```
MyPlugin.install = function (Vue, options) {
  // 1. 添加全局方法或属性
  Vue.myGlobalMethod = function () {
    // 逻辑...
  }

  // 2. 添加全局资源
  Vue.directive('my-directive', {
    bind (el, binding, vnode, oldVnode) {
      // 逻辑...
    }
    ...
  })

  // 3. 注入组件
  Vue.mixin({
    created: function () {
      // 逻辑...
    }
    ...
  })

  // 4. 添加实例方法
  Vue.prototype.$myMethod = function (methodOptions) {
    // 逻辑...
  }
}
```

过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 `v-bind` 表达式 (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示：

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

你可以在一个组件的选项中定义本地的过滤器：

```
filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

或者在创建 Vue 实例之前全局定义过滤器：

```
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
  // ...
})
```

下面这个例子用到了 `capitalize` 过滤器：

{% raw %}

{{ message | capitalize }}

{% endraw %}

过滤器函数总接收表达式的值 (之前的操作链的结果) 作为第一个参数。在上述例子中， `capitalize` 过滤器函数将会收到 `message` 的值作为第一个参数。

过滤器可以串联：

```
{{ message | filterA | filterB }}
```

在这个例子中， `filterA` 被定义为接收单个参数的过滤器函数，表达式 `message` 的值将作为参数传入到函数中。然后继续调用同样被定义为接收单个参数的过滤器函数 `filterB` ，将 `filterA` 的结果传递到 `filterB` 中。

过滤器是 JavaScript 函数，因此可以接收参数：

```
{{ message | filterA('arg1', arg2) }}
```

这里， `filterA` 被定义为接收三个参数的过滤器函数。其中 `message` 的值作为第一个参数，普通字符串 `'arg1'` 作为第二个参数，表达式 `arg2` 的值作为第三个参数。

工具

生产环境部署

开启生产环境模式

开发环境下，Vue 会提供很多警告来帮你对付常见的错误与陷阱。而在生产环境下，这些警告语句却没有用，反而会增加应用的体积。此外，有些警告检查还有一些小的运行时开销，这在生产环境模式下是可以避免的。

不使用构建工具

如果用 Vue 完整独立版本，即直接用 `<script>` 元素引入 Vue 而不提前进行构建，请记得在生产环境下使用压缩后的版本（`vue.min.js`）。两种版本都可以在安装指导中找到。

使用构建工具

当使用 webpack 或 Browserify 类似的构建工具时，Vue 源码会根据 `process.env.NODE_ENV` 决定是否启用生产环境模式，默认情况为开发环境模式。在 webpack 与 Browserify 中都有方法来覆盖此变量，以启用 Vue 的生产环境模式，同时在构建过程中警告语句也会被压缩工具去除。这些所有 `vue-cli` 模板中都预先配置好了，但了解一下怎样配置会更好。

webpack

在 webpack 4+ 中，你可以使用 `mode` 选项：

```
module.exports = {
  mode: 'production'
}
```

但是在 webpack 3 及其更低版本中，你需要使用 DefinePlugin：

```
var webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
}
```

Browserify

- 在运行打包命令时将 `NODE_ENV` 设置为 `"production"`。这等于告诉 `vueify` 避免引入热重载

和开发相关的代码。

- 对打包后的文件进行一次全局的 envify 转换。这使得压缩工具能清除掉 Vue 源码中所有用环境变量条件包裹起来的警告语句。例如：

```
NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js
```

- 或者在 Gulp 中使用 envify：

```
// 使用 envify 的自定义模块来定制环境变量
var envify = require('envify/custom')

browserify(browserifyOptions)
  .transform(vueify)
  .transform(
    // 必填项，以处理 node_modules 里的文件
    { global: true },
    envify({ NODE_ENV: 'production' })
  )
  .bundle()
```

- 或者配合 Grunt 和 grunt-browserify 使用 envify：

```
// 使用 envify 自定义模块指定环境变量
var envify = require('envify/custom')

browserify: {
  dist: {
    options: {
```

// 该函数用来调整 grunt-browserify 的默认指令

configure: b => b

.transform('vueify')

.transform(

// 用来处理 `node_modules` 文件

{ global: true },

envify({ NODE_ENV: 'production' })

)

.bundle()

}

}

}

```
#### Rollup

使用 [rollup-plugin-replace](https://github.com/rollup/rollup-plugin-replace):

``` js
const replace = require('rollup-plugin-replace')
rollup({
// ...
plugins: [
  replace({
    'process.env.NODE_ENV': JSON.stringify( 'production' )
  })
]
}).then(...)
```

模板预编译

当使用 DOM 内模板或 JavaScript 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法。

预编译模板最简单的方式就是使用单文件组件——相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串。

如果你使用 webpack，并且喜欢分离 JavaScript 和模板文件，你可以使用 vue-template-loader，它也可以在构建过程中把模板文件转换成为 JavaScript 渲染函数。

提取组件的 CSS

当使用单文件组件时，组件内的 CSS 会以 `<style>` 标签的方式通过 JavaScript 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式内容闪烁 (fouc)”。将所有组件的 CSS 提取到同一个文件可以避免这个问题，也会让 CSS 更好地进行压缩和缓存。

查阅这个构建工具各自的文档来了解更多：

- webpack + vue-loader（`vue-cli` 的 webpack 模板已经预先配置好）
- Browserify + vueify
- Rollup + rollup-plugin-vue

跟踪运行时错误

如果在组件渲染时出现运行错误，错误将会被传递至全局 `Vue.config.errorHandler` 配置函数 (如果已设置)。利用这个钩子函数来配合错误跟踪服务是个不错的主意。比如 Sentry，它为 Vue 提供了官方集成。
```

单文件组件

介绍

在很多 Vue 项目中，我们使用 `Vue.component` 来定义全局组件，紧接着用 `new Vue({ el: '#container '})` 在每个页面内指定一个容器元素。
这种方式在很多中小规模的项目中运作的很好，在这些项目里 JavaScript 只被用来加强特定的视图。但当在更复杂的项目中，或者你的前端完全由 JavaScript 驱动的时候，下面这些缺点将变得非常明显：

- 全局定义 (Global definitions) 强制要求每个 component 中的命名不得重复
- 字符串模板 (String templates) 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 `\`
- 不支持 CSS (No CSS support) 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏
- 没有构建步骤 (No build step) 限制只能使用 HTML 和 ES5 JavaScript, 而不能使用预处理器，如 Pug (formerly Jade) 和 Babel

文件扩展名为 `.vue` 的 single-file components(单文件组件) 为以上所有问题提供了解决方法，并且还可以使用 webpack 或 Browserify 等构建工具。
这是一个文件名为 `Hello.vue` 的简单实例：

现在我们获得：

- 完整语法高亮
- CommonJS 模块
- 组件作用域的 CSS

正如我们说过的，我们可以使用预处理器来构建简洁和功能更丰富的组件，比如 Pug，Babel (with ES2015 modules)，和 Stylus。

这些特定的语言只是例子，你可以只是简单地使用 Babel，TypeScript，SCSS，PostCSS - 或者其他任何能够帮助你提高生产力的预处理器。如果搭配 `vue-loader` 使用 webpack，它也是把 CSS Modules 当作第一公民来对待的。

怎么看待关注点分离？

一个重要的事情值得注意，**关注点分离不等于文件类型分离。**在现代 UI 开发中，我们已经发现相比于把代码库分离成三个大的层次并将其相互交织起来，把它们划分为松散耦合的组件再将其组合起来更合理一些。在一个组件里，其模板、逻辑和样式是内部耦合的，并且把他们搭配在一起实际上使得组件更加内聚且更可维护。
即便你不喜欢单文件组件，你仍然可以把 JavaScript、CSS 分离成独立的文件然后做到热重载和预编译。

```
<!-- my-component.vue -->
<template>
  <div>This will be pre-compiled</div>
</template>
<script src="./my-component.js"></script>
<style src="./my-component.css"></style>
```

起步

例子沙箱

如果你希望深入了解并开始使用单文件组件，请来 CodeSandbox 看看这个简单的 todo 应用。

针对刚接触 JavaScript 模块开发系统的用户

有了 `.vue` 组件，我们就进入了高级 JavaScript 应用领域。如果你没有准备好的话，意味着还需要学会使用一些附加的工具：

- Node Package Manager (NPM)：阅读 Getting Started guide 直到 10: Uninstalling global packages章节。
- Modern JavaScript with ES2015/16：阅读 Babel 的 Learn ES2015 guide。你不需要立刻记住每一个方法，但是你可以保留这个页面以便后期参考。

在你花一天时间了解这些资源之后，我们建议你参考 webpack 模板。只要遵循指示，你就能很快地运行一个用到 `.vue` 组件，ES2015 和热重载 (hot-reloading) 的 Vue 项目!
想学习更多 webpack 的知识，请移步它们的官方文档以及 webpack learning academy。在 webpack 中，每个模块被打包到 bundle 之前都由一个相应的“loader”来转换，Vue 也提供 vue-loader 插件来执行 `.vue` 单文件组件 的转换。

针对高级用户

无论你更钟情 webpack 或是 Browserify，我们为简单的和更复杂的项目都提供了一些文档模板。我们建议浏览 github.com/vuejs-templates，找到你需要的部分，然后参考 README 中的说明，使用 vue-cli 工具生成新的项目。
模板中使用 webpack，一个模块加载器加载多个模块然后构建成最终应用。为了进一步了解 webpack，可以看官方介绍视频。如果你有基础，可以看 在 Egghead.io 上的 webpack 进阶教程。

单元测试

配置和工具

任何兼容基于模块的构建系统都可以正常使用，但如果你需要一个具体的建议，可以使用 Karma 进行自动化测试。它有很多社区版的插件，包括对 Webpack 和 Browserify 的支持。更多详细的安装步骤，请参考各项目的安装文档，通过这些 Karma 配置的例子可以快速帮助你上手 (Webpack 配置，Browserify 配置)。

简单的断言

你不必为了可测性在组件中做任何特殊的操作，导出原始设置就可以了：

```
<template>
  <span>{{ message }}</span>
</template>

<script>
  export default {
    data () {
      return {
        message: 'hello!'
      }
    },
    created () {
      this.message = 'bye!'
    }
  }
</script>
```

然后随着 Vue 导入组件的选项，你可以使用许多常见的断言：

```
// 导入 Vue.js 和组件，进行测试
import Vue from 'vue'
import MyComponent from 'path/to/MyComponent.vue'

// 这里是一些 Jasmine 2.0 的测试，你也可以使用你喜欢的任何断言库或测试工具。

describe('MyComponent', () => {
  // 检查原始组件选项
  it('has a created hook', () => {
    expect(typeof MyComponent.created).toBe('function')
  })

  // 评估原始组件选项中的函数的结果
  it('sets the correct default data', () => {
```

```
    expect(typeof MyComponent.data).toBe('function')
    const defaultData = MyComponent.data()
    expect(defaultData.message).toBe('hello!')
  })

  // 检查 mount 中的组件实例
  it('correctly sets the message when created', () => {
    const vm = new Vue(MyComponent).$mount()
    expect(vm.message).toBe('bye!')
  })

  // 创建一个实例并检查渲染输出
  it('renders the correct message', () => {
    const Constructor = Vue.extend(MyComponent)
    const vm = new Constructor().$mount()
    expect(vm.$el.textContent).toBe('bye!')
  })
})
```

编写可被测试的组件

很多组件的渲染输出由它的 props 决定。事实上，如果一个组件的渲染输出完全取决于它的 props，那么它会让测试变得简单，就好像断言不同参数的纯函数的返回值。看下面这个例子：

```
<template>
  <p>{{ msg }}</p>
</template>

<script>
  export default {
    props: ['msg']
  }
</script>
```

你可以在不同的 props 中，通过 `propsData` 选项断言它的渲染输出：

```
import Vue from 'vue'
import MyComponent from './MyComponent.vue'

// 挂载元素并返回已渲染的文本的工具函数
function getRenderedText (Component, propsData) {
  const Constructor = Vue.extend(Component)
  const vm = new Constructor({ propsData: propsData }).$mount()
  return vm.$el.textContent
}

describe('MyComponent', () => {
```

```
  it('renders correctly with different props', () => {
    expect(getRenderedText(MyComponent, {
      msg: 'Hello'
    })).toBe('Hello')

    expect(getRenderedText(MyComponent, {
      msg: 'Bye'
    })).toBe('Bye')
  })
})
```

断言异步更新

由于 Vue 进行 异步更新 DOM 的情况，一些依赖 DOM 更新结果的断言必须在 `Vue.nextTick` 回调中进行：

```
// 在状态更新后检查生成的 HTML
it('updates the rendered message when vm.message updates', done => {
  const vm = new Vue(MyComponent).$mount()
  vm.message = 'foo'

  // 在状态改变后和断言 DOM 更新前等待一刻
  Vue.nextTick(() => {
    expect(vm.$el.textContent).toBe('foo')
    done()
  })
})
```

我们计划做一个通用的测试工具集，让不同策略的渲染输出 (例如忽略子组件的基本渲染) 和断言变得更简单。关于更深入的 Vue 单元测试的内容，请移步 vue-test-utils 以及我们关于 Vue 组件的单元测试的 cookbook 文章。

TypeScript 支持

在 Vue 2.5.0 中，我们大大改进了类型声明以更好地使用默认的基于对象的 API。同时此版本也引入了一些其它变化，需要开发者作出相应的升级。阅读博客文章了解更多详情。

发布为 NPM 包的官方声明文件

静态类型系统能帮助你有效防止许多潜在的运行时错误，而且随着你的应用日渐丰满会更加显著。这就是为什么 Vue 不仅仅为 Vue core 提供了针对 TypeScript 的官方类型声明，还为 Vue Router 和 Vuex 也提供了相应的声明文件。

而且，我们已经把它们发布到了 NPM，最新版本的 TypeScript 也知道该如何自己从 NPM 包里解析类型声明。这意味着只要你成功地通过 NPM 安装了，就不再需要任何额外的工具辅助，即可在 Vue 中使用 TypeScript 了。

推荐配置

```
// tsconfig.json
{
  "compilerOptions": {
    // 与 Vue 的浏览器支持保持一致
    "target": "es5",
    // 这可以对 `this` 上的数据属性进行更严格的推断
    "strict": true,
    // 如果使用 webpack 2+ 或 rollup，可以利用 tree-shake:
    "module": "es2015",
    "moduleResolution": "node"
  }
}
```

注意你需要引入 `strict: true` (或者至少 `noImplicitThis: true` ，这是 `strict` 模式的一部分) 以利用组件方法中 `this` 的类型检查，否则它会始终被看作 `any` 类型。

参阅 TypeScript 编译器选项文档 (英) 了解更多。

开发工具链

工程创建

Vue CLI 3 可以使用 TypeScript 生成新工程。创建方式：

```
# 1. 如果没有安装 Vue CLI 就先安装
npm install --global @vue/cli
```

```
# 2. 创建一个新工程，并选择 "Manually select features (手动选择特性)" 选项
vue create my-project-name
```

编辑器支持

要使用 TypeScript 开发 Vue 应用程序，我们强烈建议您使用 Visual Studio Code，它为 TypeScript 提供了极好的“开箱即用”支持。如果你正在使用单文件组件 (SFC)，可以安装提供 SFC 支持以及其他更多实用功能的 Vetur 插件。

WebStorm 同样为 TypeScript 和 Vue 提供了“开箱即用”的支持。

基本用法

要让 TypeScript 正确推断 Vue 组件选项中的类型，您需要使用 `Vue.component` 或 `Vue.extend` 定义组件：

```
import Vue from 'vue'
const Component = Vue.extend({
  // 类型推断已启用
})

const Component = {
  // 这里不会有类型推断，
  // 因为TypeScript不能确认这是Vue组件的选项
}
```

基于类的 Vue 组件

如果您在声明组件时更喜欢基于类的 API，则可以使用官方维护的 vue-class-component 装饰器：

```
import Vue from 'vue'
import Component from 'vue-class-component'

// @Component 修饰符注明了此类为一个 Vue 组件
@Component({
  // 所有的组件选项都可以放在这里
  template: '<button @click="onClick">Click!</button>'
})
export default class MyComponent extends Vue {
  // 初始数据可以直接声明为实例的属性
  message: string = 'Hello!'

  // 组件方法也可以直接声明为实例的方法
  onClick (): void {
    window.alert(this.message)
  }
```

```
}
```

增强类型以配合插件使用

插件可以增加 Vue 的全局/实例属性和组件选项。在这些情况下，在 TypeScript 中制作插件需要类型声明。庆幸的是，TypeScript 有一个特性来补充现有的类型，叫做模块补充 (module augmentation)。

例如，声明一个 `string` 类型的实例属性 `$myProperty` ：

```
// 1. 确保在声明补充的类型之前导入 'vue'
import Vue from 'vue'

// 2. 定制一个文件，设置你想要补充的类型
//    在 types/vue.d.ts 里 Vue 有构造函数类型
declare module 'vue/types/vue' {
// 3. 声明为 Vue 补充的东西
  interface Vue {
    $myProperty: string
  }
}
```

在你的项目中包含了上述作为声明文件的代码之后 (像 `my-property.d.ts`)，你就可以在 Vue 实例上使用 `$myProperty` 了。

```
var vm = new Vue()
console.log(vm.$myProperty) // 将会顺利编译通过
```

你也可以声明额外的属性和组件选项：

```
import Vue from 'vue'

declare module 'vue/types/vue' {
  // 可以使用 `VueConstructor` 接口
  // 来声明全局属性
  interface VueConstructor {
    $myGlobal: string
  }
}

// ComponentOptions 声明于 types/options.d.ts 之中
declare module 'vue/types/options' {
  interface ComponentOptions<V extends Vue> {
    myOption?: string
  }
}
```

上述的声明允许下面的代码顺利编译通过：

```
// 全局属性
console.log(Vue.$myGlobal)

// 额外的组件选项
var vm = new Vue({
  myOption: 'Hello'
})
```

标注返回值

因为 Vue 的声明文件天生就具有循环性，TypeScript 可能在推断某个方法的类型的时候存在困难。因此，你可能需要在 `render` 或 `computed` 里的方法上标注返回值。

```
import Vue, { VNode } from 'vue'

const Component = Vue.extend({
  data () {
    return {
      msg: 'Hello'
    }
  },
  methods: {
    // 需要标注有 `this` 参与运算的返回值类型
    greet (): string {
      return this.msg + ' world'
    }
  },
  computed: {
    // 需要标注
    greeting(): string {
      return this.greet() + '!'
    }
  },
  // `createElement` 是可推导的，但是 `render` 需要返回值类型
  render (createElement): VNode {
    return createElement('div', this.greeting)
  }
})
```

如果你发现类型推导或成员补齐不工作了，标注某个方法也许可以帮助你解决这个问题。使用 `--noImplicitAny` 选项将会帮助你找到这些未标注的方法。

规模化

路由

官方路由

对于大多数单页面应用，都推荐使用官方支持的 vue-router 库。更多细节可以看 vue-router 文档。

从零开始简单的路由

如果只需要非常简单的路由而不需要引入整个路由库，可以动态渲染一个页面级的组件像这样：

```
const NotFound = { template: '<p>Page not found</p>' }
const Home = { template: '<p>home page</p>' }
const About = { template: '<p>about page</p>' }

const routes = {
  '/': Home,
  '/about': About
}

new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})
```

结合 HTML5 History API，你可以建立一个非常基本但功能齐全的客户端路由器。可以直接看实例应用

整合第三方路由

如果有非常喜欢的第三方路由，如 Page.js 或者 Director，整合很简单。这有个用了 Page.js 的复杂示例。

状态管理

类 Flux 状态管理的官方实现

由于状态零散地分布在许多组件和组件之间的交互中，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 vuex：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 vue-devtools，无需配置即可进行时光旅行调试。

React 的开发者请参考以下信息

如果你是来自 React 的开发者，你可能会对 Vuex 和 Redux 间的差异表示关注，Redux 是 React 生态环境中最流行的 Flux 实现。Redux 事实上无法感知视图层，所以它能够轻松的通过一些简单绑定和 Vue 一起使用。Vuex 区别在于它是一个专门为 Vue 应用所设计。这使得它能够更好地和 Vue 进行整合，同时提供简洁的 API 和改善过的开发体验。

简单状态管理起步使用

经常被忽略的是，Vue 应用中原始 `数据` 对象的实际来源 - 当访问数据对象时，一个 Vue 实例只是简单的代理访问。所以，如果你有一处需要被多个实例间共享的状态，可以简单地通过维护一份数据来实现共享：

```
const sourceOfTruth = {}

const vmA = new Vue({
  data: sourceOfTruth
})

const vmB = new Vue({
  data: sourceOfTruth
})
```

现在当 `sourceOfTruth` 发生变化，`vmA` 和 `vmB` 都将自动的更新引用它们的视图。子组件们的每个实例也会通过 `this.$root.$data` 去访问。现在我们有了唯一的数据来源，但是，调试将会变为噩梦。任何时间，我们应用中的任何部分，在任何数据改变后，都不会留下变更过的记录。

为了解决这个问题，我们采用一个简单的 store 模式：

```
var store = {
  debug: true,
  state: {
    message: 'Hello!'
  },
  setMessageAction (newValue) {
    if (this.debug) console.log('setMessageAction triggered with', newValue)
```

```
    this.state.message = newValue
  },
  clearMessageAction () {
    if (this.debug) console.log('clearMessageAction triggered')
    this.state.message = ''
  }
}
```

需要注意，所有 store 中 state 的改变，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的 mutation 将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

```
var vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})

var vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```

重要的是，注意你不应该在 action 中 替换原始的状态对象 - 组件和 store 需要引用同一个共享对象，mutation 才能够被观察

接着我们继续延伸约定，组件不允许直接修改属于 store 实例的 state，而应执行 action 来分发 (dispatch) 事件通知 store 去改变，我们最终达成了 Flux 架构。这样约定的好处是，我们能够记录所有 store 中发生的 state 改变，同时实现能做到记录变更 (mutation)、保存状态快照、历史回滚/时光旅行的先进的调试工具。

说了一圈其实又回到了vuex，如果你已经读到这儿，或许可以去尝试一下！

服务端渲染

SSR 完全指南

在 2.3 发布后我们发布了一份完整的构建 Vue 服务端渲染应用的指南。这份指南非常深入，适合已经熟悉 Vue, webpack 和 Node.js 开发的开发者阅读。请移步 ssr.vuejs.org。

Nuxt.js

从头搭建一个服务端渲染的应用是相当复杂的。幸运的是，我们有一个优秀的社区项目 Nuxt.js 让这一切变得非常简单。Nuxt 是一个基于 Vue 生态的更高层的框架，为开发服务端渲染的 Vue 应用提供了极其便利的开发体验。更酷的是，你甚至可以用它来做为静态站生成器。推荐尝试。

内在

深入响应式原理

深入响应式原理

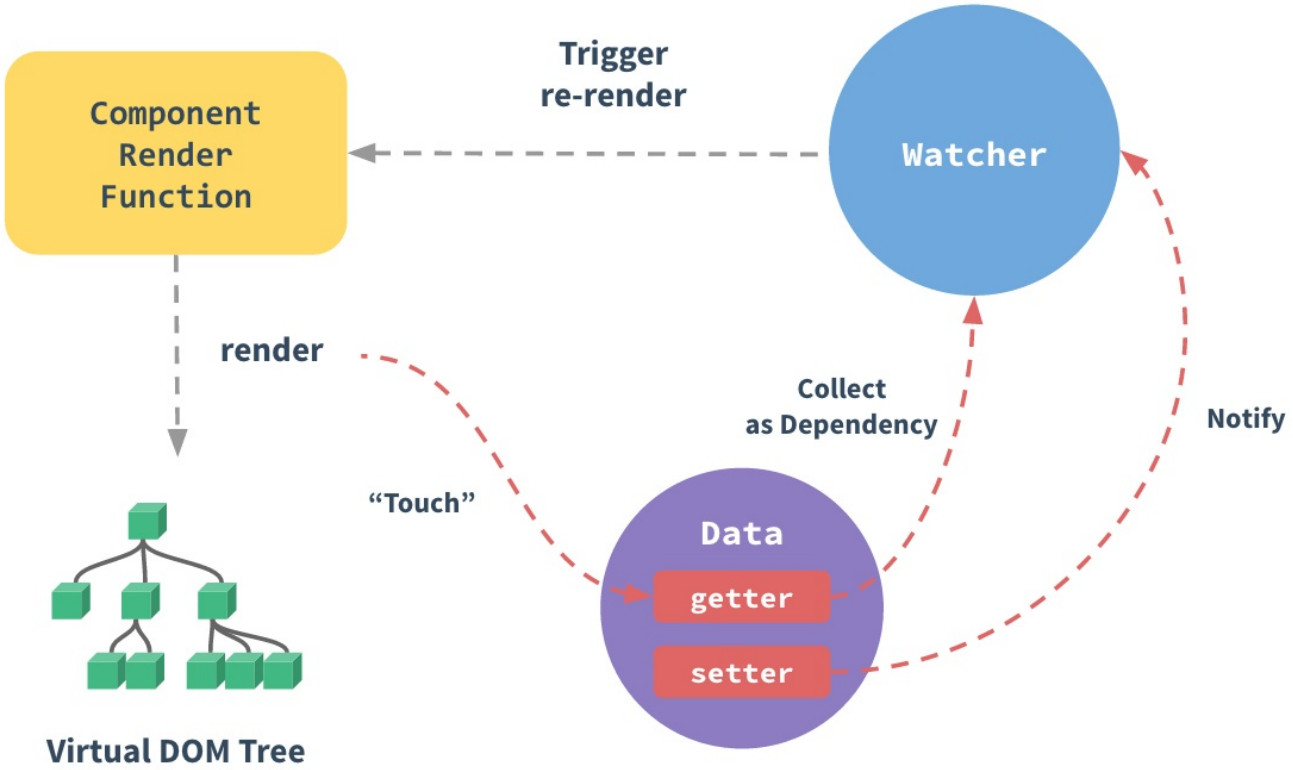
现在是时候深入一下了！Vue 最独特的特性之一，是其非侵入性的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以回避一些常见的问题。在这个章节，我们将进入一些 Vue 响应式系统的底层的细节。

如何追踪变化

当你把一个普通的 JavaScript 对象传给 Vue 实例的 `data` 选项，Vue 将遍历此对象所有的属性，并使用 Object.defineProperty 把这些属性全部转为 getter/setter。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器。

这些 getter/setter 对用户来说是不可见的，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。这里需要注意的问题是浏览器控制台在打印数据对象时 getter/setter 的格式化并不同，所以你可能需要安装 vue-devtools 来获取更加友好的检查接口。

每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 `setter` 被调用时，会通知 `watcher` 重新计算，从而致使它关联的组件得以更新。



检测变化的注意事项

受现代 JavaScript 的限制 (而且 `Object.observe` 也已经被废弃)，Vue 不能检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行 `getter/setter` 转化过程，所以属性必须在 `data` 对象上

存在才能让 Vue 转换它，这样才能让它是响应的。例如：

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

// `vm.a` 是响应的

vm.b = 2
// `vm.b` 是非响应的
```

Vue 不允许在已经创建的实例上动态添加新的根级响应式属性 (root-level reactive property)。然而它可以使用 `Vue.set(object, key, value)` 方法将响应属性添加到嵌套的对象上：

```
Vue.set(vm.someObject, 'b', 2)
```

您还可以使用 `vm.$set` 实例方法，这也是全局 `Vue.set` 方法的别名：

```
this.$set(this.someObject,'b',2)
```

有时你想向一个已有对象添加多个属性，例如使用 `Object.assign()` 或 `_.extend()` 方法来添加属性。但是，这样添加到对象上的新属性不会触发更新。在这种情况下可以创建一个新的对象，让它包含原对象的属性和新的属性：

```
// 代替 `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

也有一些数组相关的问题，之前已经在列表渲染中讲过。

声明响应式属性

由于 Vue 不允许动态添加根级响应式属性，所以你必须在初始化实例前声明根级响应式属性，哪怕只是一个空值：

```
var vm = new Vue({
  data: {
    // 声明 message 为一个空值字符串
    message: ''
  },
```

```
  template: '<div>{{ message }}</div>'
})
// 之后设置 `message`
vm.message = 'Hello!'
```

如果你未在 data 选项中声明 `message` ，Vue 将警告你渲染函数正在试图访问的属性不存在。
这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例在类型检查系统的帮助下运行的更高效。而且在代码可维护性方面也有一点重要的考虑： `data` 对象就像组件状态的概要，提前声明所有的响应式属性，可以让组件代码在以后重新阅读或其他开发人员阅读时更易于被理解。

异步更新队列

可能你还没有注意到，Vue 异步执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作上非常重要。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部尝试对异步队列使用原生的 `Promise.then` 和 `MessageChannel` ，如果执行环境不支持，会采用 `setTimeout(fn, 0)` 代替。
例如，当你设置 `vm.someData = 'new value'` ，该组件不会立即重新渲染。当刷新队列时，组件会在事件循环队列清空时的下一个“tick”更新。多数情况我们不需要关心这个过程，但是如果你想在 DOM 状态更新后做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员沿着“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们确实要这么做。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 `Vue.nextTick(callback)` 。这样回调函数在 DOM 更新完成后就会调用。例如：

```
<div id="example">{{message}}</div>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    message: '123'
  }
})
vm.message = 'new message' // 更改数据
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // true
})
```

在组件内使用 `vm.$nextTick()` 实例方法特别方便，因为它不需要全局 `Vue` ，并且回调函数中的 `this` 将自动绑定到当前的 Vue 实例上：

```
Vue.component('example', {
  template: '<span>{{ message }}</span>',
  data: function () {
    return {
      message: '没有更新'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = '更新完成'
      console.log(this.$el.textContent) // => '没有更新'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => '更新完成'
      })
    }
  }
})
```

因为 `$nextTick()` 返回一个 Promise 对象，所以你可以使用新的 ES2016 async/await 语法完成相同的事情：

```
  methods: {
    async updateMessage: function () {
      this.message = 'updated'
      console.log(this.$el.textContent) // => '未更新'
      await this.$nextTick()
      console.log(this.$el.textContent) // => '已更新'
    }
  }
```

迁移

从 Vue 1.x 迁移

FAQ

> 哇，非常长的一页！是否意味着 Vue 2.0 已经完全不同了呢，是否需要从头学起呢，Vue 1.0 的项目是不是没法迁移了？

非常开心地告诉你，并不是！几乎 90% 的 API 和核心概念都没有变。因为本节包含了很多详尽的阐述以及许多迁移的例子，所以显得有点长。不用担心，你不必从头到尾把本节读一遍！

> 我该从哪里开始项目迁移呢？

1. 首先，在当前项目下运行迁移工具。我们非常谨慎地把高级 Vue 升级过程简化为使用一个简单的命令行工具。当工具识别出旧有的特性后，就会告知你并给出建议，同时附上关于详细信息的链接。
2. 然后，浏览本页面的侧边栏列出的内容。如果发现有的标题对你的项目有影响，但是迁移工具没有给出提示，请检查自己的项目。
3. 如果你的项目有测试代码，运行并查看仍然失败的地方。如果没有测试代码，在浏览器中打开你的程序，通过导航环顾并留意那些报错或警告信息。
4. 现在，你的应用程序应该已彻底完成迁移。如果你渴望了解更多，可以阅读本页面剩余部分 - 或者从介绍部分，从头开始深入新的文档和改进过的指南。由于你已经熟悉一些核心概念，所以许多部分已经被删除掉。

> 将 Vue 1.x 版本的应用程序迁移到 2.0 要花多长时间？

这取决于几个因素：

- 取决于你应用程序的规模 (中小型的基本上一天内就可以搞定)。
- 取决于你分心和开始 2.0 最酷的新功能的次数。☺ 无法判断时间，我们构建 2.0 应用的时候也经常发生这种事！
- 取决于你使用了哪些旧有的特性。大部分可以通过查找和替换 (find-and-replace) 来实现升级，但有一些可能还是要花点时间。如果你没有遵循最佳实践，Vue 2.0 会尽力强迫你去遵循。这有利于项目的长期运行，但也可能意味着重大重构 (尽管有些需要重构的部分可能已经过时)。

> 如果我升级到到 Vue 2 ，我还必须同时升级 Vuex 和 Vue Router？

只有 Vue Router 2 与 Vue 2 保持兼容，所以 Vue Router 是需要升级的，你必须遵循 Vue Router 迁移方式来处理。幸运的是，大多数应用没有很多 router 相关代码，所以迁移可能不会超过一个小时。
对于 Vuex ，版本 0.8+ 与 Vue 2 保持兼容，所以部分不必强制升级。可以促使你立即升级的唯一理由，是你想要使用那些 Vuex 2 中新的高级特性，比如模块 (modules) 和减少的样板文件 (reduced boilerplate)。

模板

片段实例 移除

每个组件必须只有一个根元素。不再允许片段实例，如果你有这样的模板：

```
<p>foo</p>
<p>bar</p>
```

最好把整个内容都简单包裹到一个新的元素里，如下所示：

```
<div>
  <p>foo</p>
  <p>bar</p>
</div>
```

生命周期钩子函数

`beforeCompile` 移除

使用 `created` 钩子函数替代。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`compiled` 替换

使用 `mounted` 钩子函数替代。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`attached` 移除

使用其他钩子函数内置的 DOM 检测 (DOM check) 方法。例如，替换如下：

```
attached: function () {
  doSomething()
}
```

可以这样使用：

```
mounted: function () {
  this.$nextTick(function () {
    doSomething()
  })
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`detached` 移除

使用其他钩子函数内的 DOM 检测 (DOM check) 方法。例如，替换如下：

```
detached: function () {
  doSomething()
}
```

可以这样使用：

```
destroyed: function () {
  this.$nextTick(function () {
    doSomething()
  })
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`init` 重新命名

使用新的 `beforeCreate` 钩子函数替代，本质上 beforeCreate 和 init 完全相同。init 被重新命名是为了和其他的生命周期方法的命名方式保持一致。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`ready` 替换

使用新的 `mounted` 钩子函数替代。应该注意的是，使用 `mounted` 并不能保证钩子函数中的 this.$el 在 document 中。为此还应该引入 `Vue.nextTick` / `vm.$nextTick` 。例如：

```
mounted: function () {
  this.$nextTick(function () {
    // 代码保证 this.$el 在 document 中
  })
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出所有使用此钩子函数的示例。</p>
</div>
```

`v-for`

`v-for` 遍历数组时的参数顺序 变更

当包含 `index` 时，之前遍历数组时的参数顺序是 `(index, value)` 。现在是 `(value, index)` ，来和 JavaScript 的原生数组方法 (例如 `forEach` 和 `map`) 保持一致。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出那些使用旧有参数顺序的示例。注意，如果你将你的 index 参数命名为一些不通用的名字 (
例如 <code>position</code> 或 <code>num</code>)，迁移工具将不会把它们标记出来。</p>
</div>
```

`v-for` 遍历对象时的参数顺序 变更

当包含 `key` 时，之前遍历对象的参数顺序是 `(key, value)` 。现在是 `(value, key)` ，来和常见的对象迭代器 (例如 lodash) 保持一致。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出那些使用旧有参数顺序的示例。注意，如果你将你的 key 参数命名为一些不通用的名字 (例
如 <code>name</code> 或 <code>property</code>)，迁移工具将不会把它们标记出来。</p>
</div>
```

`$index` and `$key` 移除

已经移除了 `$index` 和 `$key` 这两个隐式声明变量，以便在 `v-for` 中显式定义。这可以使没有太多 Vue 开发经验的开发者更好地阅读代码，并且在处理嵌套循环时也能产生更清晰的行为。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出使用这些移除变量的示例。如果你没有找到，也可以在<strong>控制台错误</strong>中查
找 (例如 <code>Uncaught ReferenceError: $index is not defined</code>)。</p>
</div>
```

`track-by` 替换

`track-by` 已经替换为 `key` ，它的工作方式与其他属性一样，没有 `v-bind` 或者 `:` 前缀，它会被作为一个字符串处理。多数情况下，你需要使用具有完整表达式的动态绑定 (dynamic binding) 来替换静态的 key。例如，替换：

```
<div v-for="item in items" track-by="id">
```

你现在应该写为：

```
<div v-for="item in items" v-bind:key="item.id">
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具
</a>来找出那些使用<code>track-by</code>的示例。</p>
```

```
</div>
```

`v-for` 范围值 变更

之前，`v-for="number in 10"` 的 `number` 从 0 开始到 9 结束，现在从 1 开始，到 10 结束。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库中使用正则 <code>/\w+ in \d+/</code>搜索。当出现在 <code>v-for</code>
中，请检查是否受到影响。</p>
</div>
```

Props

`coerce` Prop 的参数 移除

如果需要检查 prop 的值，创建一个内部的 computed 值，而不再在 props 内部去定义，例如：

```
props: {
  username: {
    type: String,
    coerce: function (value) {
      return value
        .toLowerCase()
        .replace(/\s+/, '-')
    }
  }
}
```

现在应该写为：

```
props: {
  username: String,
},
computed: {
  normalizedUsername: function () {
    return this.username
      .toLowerCase()
      .replace(/\s+/, '-')
  }
}
```

这样有一些好处：

- 你可以对保持原始 prop 值的操作权限。
- 通过给予验证后的值一个不同的命名，强制开发者使用显式申明。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找出包含 <code>coerce</code> 选项的实例。</p>
</div>
```

`twoWay` Prop 的参数 移除

Props 现在只能单向传递。为了对父组件产生反向影响，子组件需要显式地传递一个事件而不是依赖于隐式地双向绑定。详见：

- 自定义组件事件
- 自定义输入组件 (使用组件事件)
- 全局状态管理

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>, 找出含有 <code>twoWay</code> 参数的实例。</p>
</div>
```

`v-bind` 的 `.once` 和 `.sync` 修饰符 移除

Props 现在只能单向传递。为了对父组件产生反向影响，子组件需要显式地传递一个事件而不是依赖于隐式地双向绑定。详见：

- 自定义组件事件
- 自定义输入组件 (使用组件事件)
- 全局状态管理

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到使用 <code>.once</code> 和 <code>.sync</code> 修饰符的实例。</p>
</div>
```

修改 Props 弃用

组件内修改 prop 是反模式 (不推荐的) 的。比如，先声明一个 prop ，然后在组件中通过

`this.myProp = 'someOtherValue'` 改变 prop 的值。根据渲染机制，当父组件重新渲染时，子组件的内部 prop 值也将被覆盖。

大多数情况下，改变 prop 值可以用以下选项替代：

- 通过 data 属性，用 prop 去设置一个 data 属性的默认值。
- 通过 computed 属性。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行端对端测试，查看关于 prop 修改的<strong>控制台警告信息</strong>。</p>
</div>
```

根实例的 Props 替换

对于一个根实例来说 (比如：用 `new Vue({ ... })` 创建的实例)，只能用 `propsData` 而不是 `props` 。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行端对端测试，将会弹出 <strong>failed tests</strong> 来通知你使用 <code>props</code> 的根实例已经失效。</p>
</div>
```

计算属性

`cache: false` 弃用

在 Vue 未来的大版本中，计算属性的缓存验证将会被移除。把不缓存的计算属性转换为方法可以得到和之前相同的结果。

示例：

```
template: '<p>message: {{ timeMessage }}</p>',
computed: {
  timeMessage: {
    cache: false,
    get: function () {
      return Date.now() + this.message
    }
  }
}
```

或者使用组件方法：

```
template: '<p>message: {{ getTimeMessage() }}</p>',
methods: {
  getTimeMessage: function () {
    return Date.now() + this.message
  }
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
<code>cache: false</code> 的选项。</p>
</div>
```

Built-In 指令

`v-bind` 真/假值 变更

在2.0中使用 `v-bind` 时，只有 `null` ， `undefined` ，和 `false` 被看作是假。这意味着， `0` 和空字符串将被作为真值渲染。比如 `v-bind:draggable="''"` 将被渲染为 `draggable="true"` 。
对于枚举属性，除了以上假值之外，字符串 `"false"` 也会被渲染为 `attr="false"` 。
注意，对于其它钩子函数 (如 `v-if` 和 `v-show`)，他们依然遵循 js 对真假值判断的一般规则。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行端到端测试，如果你 app 的任何部分有可能被这个升级影响到，将会弹出<strong>failed
tests</strong></p>
</div>
```

用 `v-on` 监听原生事件 变更

现在在组件上使用 `v-on` 只会监听自定义事件 (组件用 `$emit` 触发的事件)。如果要监听根元素的原生事件，可以使用 `.native` 修饰符，比如：

```
<my-component v-on:click.native="doSomething"></my-component>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行端对端测试，如果你 app 的任何部分有可能被这个升级影响到，将会弹出<strong>failed
tests</strong> </p>
</div>
```

带有 `debounce` 的 `v-model` 移除

Debouncing 曾经被用来控制 Ajax 请求及其它高耗任务的频率。Vue 中 `v-model` 的 `debounce` 属性参数使得在一些简单情况下非常容易实现这种控制。但实际上，这是控制了状态更新的频率，而不是控制高耗时任务本身。这是个微小的差别，但是会随着应用增长而显现出局限性。

例如在设计一个搜索提示时的局限性：

```
<script src="https://cdn.jsdelivr.net/lodash/4.13.1/lodash.js"></script>
<div id="debounce-search-demo" class="demo">
  <input v-model="searchQuery" placeholder="Type something">
  <strong>{{ searchIndicator }}</strong>
</div>
<script>
new Vue({
  el: '#debounce-search-demo',
  data: {
    searchQuery: '',
    searchQueryIsDirty: false,
    isCalculating: false
  },
  computed: {
    searchIndicator: function () {
      if (this.isCalculating) {
        return '⟳ Fetching new results'
      } else if (this.searchQueryIsDirty) {
        return '... Typing'
      } else {
        return '✓ Done'
      }
    }
  },
  watch: {
    searchQuery: function () {
      this.searchQueryIsDirty = true
      this.expensiveOperation()
    }
  },
  methods: {
    expensiveOperation: _.debounce(function () {
      this.isCalculating = true
      setTimeout(function () {
        this.isCalculating = false
        this.searchQueryIsDirty = false
      }.bind(this), 1000)
    }, 500)
  }
})
</script>
```

使用 `debounce` 参数，便无法观察 "Typing" 的状态。因为无法对输入状态进行实时检测。然而，通过将 `debounce` 与 Vue 解耦，可以仅仅只延迟我们想要控制的操作，从而避开这些局限性：

```
<!--
通过使用 lodash 或者其它库的 debounce 函数，
我们相信 debounce 实现是一流的，
并且可以随处使用它，不仅仅是在模板中。
-->
<script src="https://cdn.jsdelivr.net/lodash/4.13.1/lodash.js"></script>
<div id="debounce-search-demo">
  <input v-model="searchQuery" placeholder="Type something">
  <strong>{{ searchIndicator }}</strong>
</div>
```

```
new Vue({
  el: '#debounce-search-demo',
  data: {
    searchQuery: '',
    searchQueryIsDirty: false,
    isCalculating: false
  },
  computed: {
    searchIndicator: function () {
      if (this.isCalculating) {
        return '⟳ Fetching new results'
      } else if (this.searchQueryIsDirty) {
        return '... Typing'
      } else {
        return '✓ Done'
      }
    }
  },
  watch: {
    searchQuery: function () {
      this.searchQueryIsDirty = true
      this.expensiveOperation()
    }
  },
  methods: {
    // 这是 debounce 实现的地方。
    expensiveOperation: _.debounce(function () {
      this.isCalculating = true
      setTimeout(function () {
        this.isCalculating = false
        this.searchQueryIsDirty = false
      }.bind(this), 1000)
```

```
    }, 500)
  }
})
```

这种方式的另外一个优点是：当包裹函数执行时间与延时时间相当时，将会等待较长时间。比如，当给出搜索建议时，要等待用户输入停止一段时间后才给出建议，这个体验非常差。其实，这时候更适合用 throttling 函数。因为现在你可以自由的使用类似 lodash 之类的库，所以很快就可以用 throttling 重构项目。

```
<div class="upgrade-path">
  <h4>Upgrade Path</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找出使用 <code>debounce</code> 参数的 实例。</p>
</div>
```

使用 `lazy` 或者 `number` 参数的 `v-model` 。替换

`lazy` 和 `number` 参数现在以修饰符的形式使用，这样看起来更加清晰，而不是这样：

```
<input v-model="name" lazy>
<input v-model="age" type="number" number>
```

现在写成这样：

```
<input v-model.lazy="name">
<input v-model.number="age" type="number">
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到这些弃用参数。</p>
</div>
```

使用内联 `value` 的 `v-model` 移除

`v-model` 不再以内联 `value` 方式初始化的初值了，显然他将以实例的 data 相应的属性作为真正的初值。

这意味着以下元素：

```
<input v-model="text" value="foo">
```

在 data 选项中有下面写法的：

```
data: {
  text: 'bar'
}
```

将渲染 model 为 'bar' 而不是 'foo' 。同样，对 `<textarea>` 已有的值来说：

```
<textarea v-model="text">
  hello world
</textarea>
```

必须保证 `text` 初值为 "hello world"

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>升级后运行端对端测试，注意关于<code>v-model</code>内联参数的 <strong>console warnings</strong></p>
</div>
```

`v-model` with `v-for` Iterated Primitive Values 移除

像这样的写法将失效：

```
<input v-for="str in strings" v-model="str">
```

因为 `<input>` 将被编译成类似下面的 js 代码：

```
strings.map(function (str) {
  return createElement('input', ...)
})
```

这样， `v-model` 的双向绑定在这里就失效了。把 `str` 赋值给迭代器里的另一个值也没有用，因为它仅仅是函数内部的一个变量。

替代方案是，你可以使用对象数组，这样 `v-model` 就可以同步更新对象里面的字段了，例如：

```
<input v-for="obj in objects" v-model="obj.str">
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行测试，如果你的 app 有地方被这个更新影响到的话将会弹出<strong>failed tests</strong>提示。</p>
</div>
```

带有 `!important` 的 `v-bind:style` 移除

这样写将失效：

```
<p v-bind:style="{ color: myColor + ' !important' }">hello</p>
```

如果确实需要覆盖其它的 `!important` ，最好用字符串形式去写：

```
<p v-bind:style="'color: ' + myColor + ' !important'">hello</p>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper"> 迁移帮助工具。</
a>找到含有 <code>!important</code> 的 style 绑定对象。</p>
</div>
```

`v-el` 和 `v-ref` 替换

简单起见， `v-el` 和 `v-ref` 合并为一个 `ref` 属性了，可以在组件实例中通过 `$refs` 来调用。这意味着 `v-el:my-element` 将写成这样： `ref="myElement"` ， `v-ref:my-component` 变成了这样： `ref="myComponent"` 。绑定在一般元素上时， `ref` 指 DOM 元素，绑定在组件上时， `ref` 为一组件实例。

因为 `v-ref` 不再是一个指令了而是一个特殊的属性，它也可以被动态定义了。这样在和 `v-for` 结合的时候是很有用的：

```
<p v-for="item in items" v-bind:ref="'item' + item.id"></p>
```

以前 `v-el` / `v-ref` 和 `v-for` 一起使用将产生一个 DOM 数组或者组件数组，因为没法给每个元素一个特定名字。现在你还仍然可以这样做，给每个元素一个同样的 `ref` ：

```
<p v-for="item in items" ref="items"></p>
```

和 1.x 中不同， `$refs` 不是响应的，因为它们在渲染过程中注册/更新。只有监听变化并重复渲染才能使它们响应。

另一方面，设计 `$refs` 主要是提供给 js 程序访问的，并不建议在模板中过度依赖使用它。因为这意味着在实例之外去访问实例状态，违背了 Vue 数据驱动的思想。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找出实例中的 <code>v-el</code> 和 <code>v-ref</code> 。</p>
</div>
```

`v-show` 后面使用 `v-else` 移除

`v-else` 不能再跟在 `v-show` 后面使用。请在 `v-if` 的否定分支中使用 `v-show` 来替代。例如：

```
<p v-if="foo">Foo</p>
<p v-else v-show="bar">Not foo, but bar</p>
```

现在应该写出这样：

```
<p v-if="foo">Foo</p>
<p v-if="!foo && bar">Not foo, but bar</p>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找出实例中存在的 <code>v-else</code> 以及 <code>v-show</code>。</p>
</div>
```

自定义指令 简化

在新版中，指令的使用范围已经大大减小了：现在指令仅仅被用于低级的 DOM 操作。大多数情况下，最好是使用组件作为代码复用的抽象层。

显要的改变有如下几点：

- 指令不再拥有实例。意思是，在指令的钩子函数中不再拥有实例的 `this` 。替代的是，你可以在参数中接受你需要的任何数据。如果确实需要，可以通过 `el` 来访问实例。
- 类似 `acceptStatement` ， `deep` ， `priority` 等都已被弃用。为了替换 `双向` 指令，见 示例。
- 现在有些钩子的意义和以前不一样了，并且多了两个钩子函数。

幸运的是，新钩子更加简单，更加容易掌握。详见 自定义指令指南。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
```

```
定义指令的地方。在 helper 工具会把这些地方标记出来，因为很有可能这些地方需要重构。</p>
</div>
```

指令 `.literal` 修饰符 移除

`.literal` 修饰符已经被移除，为了获取一样的功能，可以简单地提供字符串修饰符作为值。

示例，如下更改：

```
<p v-my-directive.literal="foo bar baz"></p>
```

只是：

```
<p v-my-directive="'foo bar baz'"></p>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
实例中使用 `.literal` 修饰符的地方。</p>
</div>
```

过渡

`transition` 参数 替换

Vue 的过渡系统有了彻底的改变，现在通过使用 `<transition>` 和 `<transition-group>` 来包裹元素实现过渡效果，而不再使用 `transition` 属性。详见 Transitions guide。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
使用 <code>transition</code> 属性的地方。</p>
</div>
```

可复用的过渡 `Vue.transition` 替换

在新的过渡系统中，可以通过模板复用过渡效果。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
使用 <code>transition</code> 属性的地方。</p>
</div>
```

过渡的 `stagger` 参数 移除

如果希望在列表渲染中使用渐近过渡，可以通过设置元素的 `data-index` (或类似属性) 来控制时间。请参考这个例子。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到使用 <code>transition</code> 属性的地方。升级期间，你可以“过渡”到新的过渡策略。</p>
</div>
```

事件

`events` 选项 移除

`events` 选项被弃用。事件处理器现在在 `created` 钩子中被注册。参考详细示例 `$dispatch` and `$broadcast` 迁移指南

`Vue.directive('on').keyCodes` 替换

新的简明配置 `keyCodes` 的方式是通过 `Vue.config.keyCodes` 例如：

```
// v-on:keyup.f1 不可用
Vue.config.keyCodes.f1 = 112
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到过时的 <code>keyCode</code> 配置</p>
</div>
```

`$dispatch` 和 `$broadcast` 替换

`$dispatch` 和 `$broadcast` 已经被弃用。请使用更多简明清晰的组件间通信和更好的状态管理方案，如：Vuex。

因为基于组件树结构的事件流方式实在是让人难以理解，并且在组件结构扩展的过程中会变得越来越脆弱。这种事件方式确实不太好，我们也不希望在以后让开发者们太痛苦。并且 `$dispatch` 和 `$broadcast` 也没有解决兄弟组件间的通信问题。

对于 `$dispatch` 和 `$broadcast` 最简单的升级方式就是：通过使用事件中心，允许组件自由交流，无论组件处于组件树的哪一层。由于 Vue 实例实现了一个事件分发接口，你可以通过实例化一个空的 Vue 实例来实现这个目的。

这些方法的最常见用途之一是父子组件的相互通信。在这些情况下，你可以使用 `v-on` 监听子组件上 $emit 的变化。这可以允许你很方便的添加事件显性。
然而，如果是跨多层父子组件通信的话， `$emit` 并没有什么用。相反，用集中式的事件中间件可以做到简单的升级。这会让组件之间的通信非常顺利，即使是兄弟组件。因为 Vue 通过事件发射器接口执行实例，实际上你可以使用一个空的 Vue 实例。
比如，假设我们有个 todo 的应用结构如下：

```
Todos
├─ NewTodoInput
└─ Todo
   └─ DeleteTodoButton
```

可以通过单独的事件中心管理组件间的通信：

```
// 将在各处使用该事件中心
// 组件通过它来通信
var eventHub = new Vue()
```

然后在组件中，可以使用 `$emit` ， `$on` ， `$off` 分别来分发、监听、取消监听事件：

```
// NewTodoInput
// ...
methods: {
  addTodo: function () {
    eventHub.$emit('add-todo', { text: this.newTodoText })
    this.newTodoText = ''
  }
}
```

```
// DeleteTodoButton
// ...
methods: {
  deleteTodo: function (id) {
    eventHub.$emit('delete-todo', id)
  }
}
```

```
// Todos
// ...
created: function () {
  eventHub.$on('add-todo', this.addTodo)
  eventHub.$on('delete-todo', this.deleteTodo)
```

```
},
// 最好在组件销毁前
// 清除事件监听
beforeDestroy: function () {
  eventHub.$off('add-todo', this.addTodo)
  eventHub.$off('delete-todo', this.deleteTodo)
},
methods: {
  addTodo: function (newTodo) {
    this.todos.push(newTodo)
  },
  deleteTodo: function (todoId) {
    this.todos = this.todos.filter(function (todo) {
      return todo.id !== todoId
    })
  }
}
```

在简单的情况下这样做可以替代 `$dispatch` 和 `$broadcast` ，但是对于大多数复杂情况，更推荐使用一个专用的状态管理层如：Vuex。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找出使用 <code>$dispatch</code> 和 <code>$broadcast</code>的实例。</p>
</div>
```

过滤器

插入文本之外的过滤器 移除

现在过滤器只能用在插入文本中 (`{{ }}` tags)。我们发现在指令 (如： `v-model` ， `v-on` 等) 中使用过滤器使事情变得更复杂。像 `v-for` 这样的列表过滤器最好把处理逻辑作为一个计算属性放在 js 里面，这样就可以在整个模板中复用。

总之，能在原生 js 中实现的东西，我们尽量避免引入一个新的符号去重复处理同样的问题。下面是如何替换 Vue 内置过滤器：

替换 `debounce` 过滤器

不再这样写

```
<input v-on:keyup="doStuff | debounce 500">
```

```
methods: {
```

```
  doStuff: function () {
    // ...
  }
}
```

请使用 lodash's `debounce` (也有可能是 `throttle`) 来直接控制高耗任务。可以这样来实现上面的功能：

```
<input v-on:keyup="doStuff">
```

```
methods: {
  doStuff: _.debounce(function () {
    // ...
  }, 500)
}
```

这种写法的更多优点详见：[`v-model` 示例](#带有-debounce-的-v-model 移除)。

替换 `limitBy` 过滤器

不再这样写：

```
<p v-for="item in items | limitBy 10">{{ item }}</p>
```

在 computed 属性中使用 js 内置方法： `.slice` method：

```
<p v-for="item in filteredItems">{{ item }}</p>
```

```
computed: {
  filteredItems: function () {
    return this.items.slice(0, 10)
  }
}
```

替换 `filterBy` 过滤器

不再这样写：

```
<p v-for="user in users | filterBy searchQuery in 'name'">{{ user.name }}</p>
```

在 computed 属性中使用 js 内置方法 `.filter` method：

```
<p v-for="user in filteredUsers">{{ user.name }}</p>
```

```
computed: {
  filteredUsers: function () {
    var self = this
    return self.users.filter(function (user) {
      return user.name.indexOf(self.searchQuery) !== -1
    })
  }
}
```

js 原生的 `.filter` 同样能实现很多复杂的过滤器操作，因为可以在计算 computed 属性中使用所有 js 方法。比如，想要通过匹配用户名字和电子邮箱地址 (不区分大小写) 找到用户：

```
var self = this
self.users.filter(function (user) {
  var searchRegex = new RegExp(self.searchQuery, 'i')
  return user.isActive && (
    searchRegex.test(user.name) ||
    searchRegex.test(user.email)
  )
})
```

替换 `orderBy` 过滤器

不这样写：

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

而是在 computed 属性中使用 lodash's `orderBy` (或者可能是 `sortBy`)：

```
<p v-for="user in orderedUsers">{{ user.name }}</p>
```

```
computed: {
  orderedUsers: function () {
    return _.orderBy(this.users, 'name')
  }
}
```

甚至可以字段排序：

```
_.orderBy(this.users, ['name', 'last_login'], ['asc', 'desc'])
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到指令中使用的过滤器。如果有些没找到，看看<strong>控制台错误信息</strong>。</p>
</div>
```

过滤器参数符号 变更

现在过滤器参数形式可以更好地与 js 函数调用方式一致，因此不用再用空格分隔参数：

```
<p>{{ date | formatDate 'YY-MM-DD' timeZone }}</p>
```

现在用圆括号括起来并用逗号分隔：

```
<p>{{ date | formatDate('YY-MM-DD', timeZone) }}</p>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到老式的调用符号，如果有遗漏，请看<strong>控制台错误信息</strong>。</p>
</div>
```

内置文本过滤器 移除

尽管插入文本内部的过滤器依然有效，但是所有内置过滤器已经移除了。取代的是，推荐在每个区域使用更专业的库来解决。(比如用 `date-fns` 来格式化日期，用 `accounting` 来格式化货币)。

对于每个内置过滤器，我们大概总结了下该怎么替换。代码示例可能写在自定义 helper 函数，方法或计算属性中。

替换 `json` 过滤器

不用一个个改，因为 Vue 已经帮你自动格式化好了，无论是字符串，数字还是数组，对象。如果想用 js 的 `JSON.stringify` 功能去实现，你也可以把它写在方法或者计算属性里面。

替换 `capitalize` 过滤器

```
text[0].toUpperCase() + text.slice(1)
```

替换 `uppercase` 过滤器

```
text.toUpperCase()
```

替换 `lowercase` 过滤器

```
text.toLowerCase()
```

替换 `pluralize` 过滤器

NPM 上的 pluralize 库可以很好的实现这个功能。如果仅仅想将特定的词格式化成复数形式或者想给特定的值 ('0') 指定特定的输出，也可以很容易地自定义复数格式化过滤器：

```
function pluralizeKnife (count) {
  if (count === 0) {
    return 'no knives'
  } else if (count === 1) {
    return '1 knife'
  } else {
    return count + 'knives'
  }
}
```

Replacing the `currency` Filter

对于简单的问题，可以这样做：

```
'$' + price.toFixed(2)
```

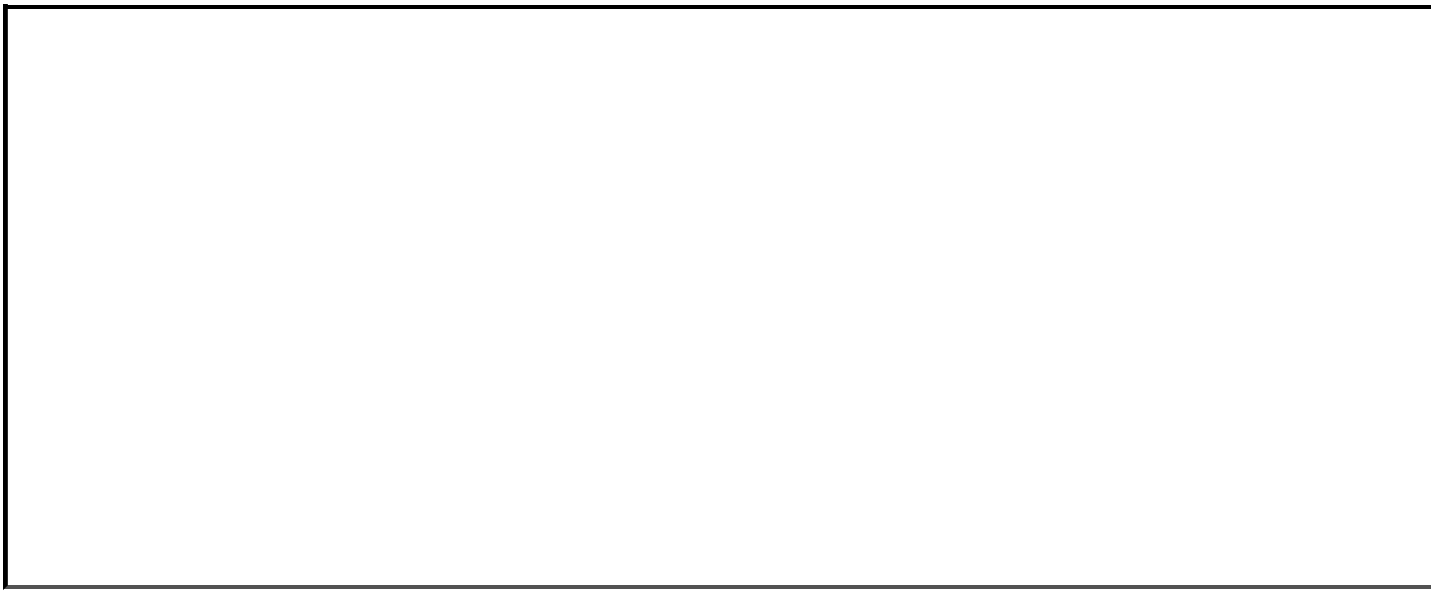
大多数情况下，仍然会有奇怪的现象 (比如 `0.035.toFixed(2)` 向上取舍得到 `0.04` ，但是 `0.045` 向下取舍却也得到 `0.04`)。解决这些问题可以使用 `accounting` 库来实现更多可靠的货币格式化。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到舍弃的过滤器。如果有些遗漏，请参考<strong>控制台错误信息</strong>。</p>
</div>
```

双向过滤器 替换

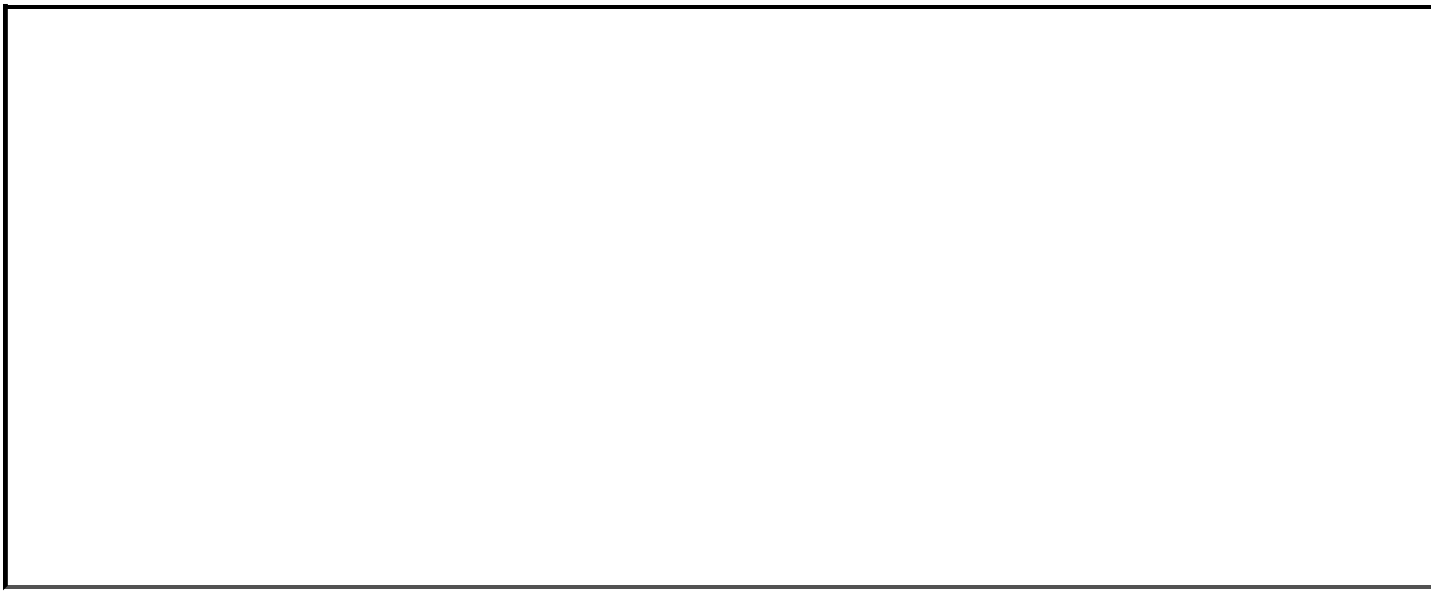
有些用户已经乐于通过 `v-model` 使用双向过滤器，以很少的代码创建有趣的输入。尽管表面上很简单，双向过滤器也会暗藏一些巨大的复杂性——甚至促使状态更新变得迟钝影响用户体验。推荐用包裹一个输入的组件取而代之，这样以更显性且功能更丰富的方式创建自定义的输入。

我们现在做一次双向汇率过滤器的迁移作为示范：

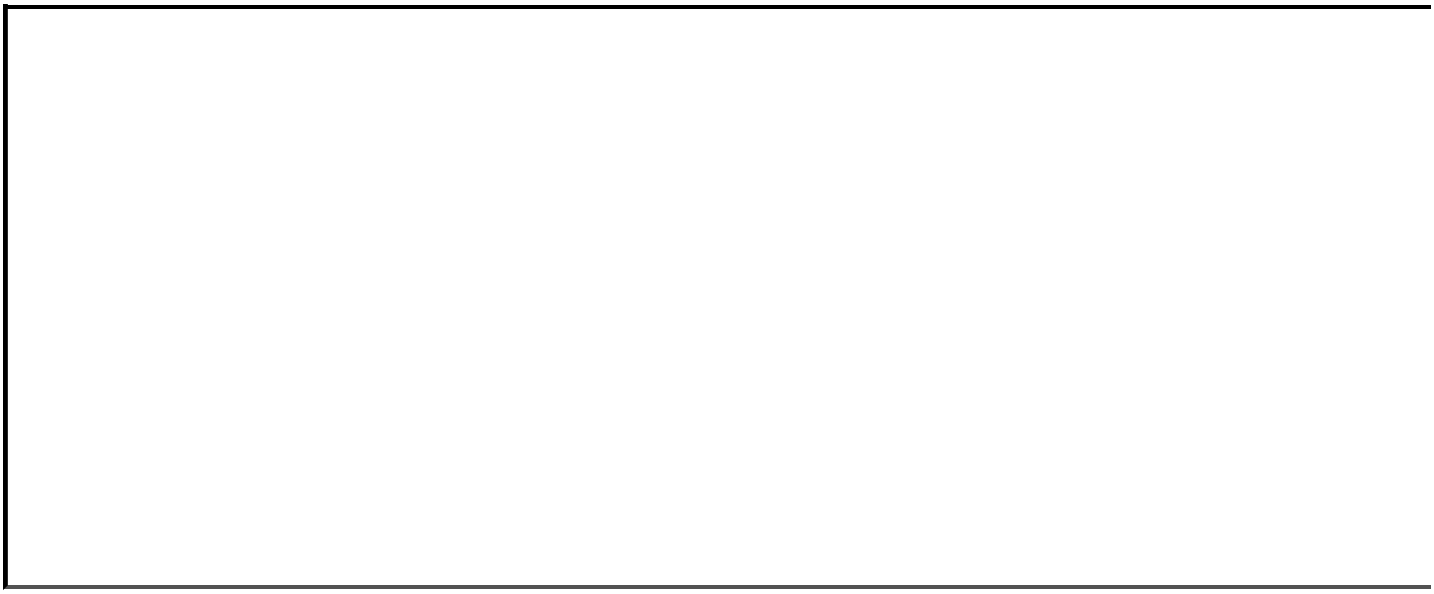


它基本工作良好，但是拖延的状态更新会导致奇怪的行为。比如，点击 `Result` 标签，试着在其中一个输入框中输入 `9.999` 。当输入框失去焦点的时候，其值将会更新到 `$10.00` 。然而当我们从整个计算器的角度看时，你会发现存储的数据是 `9.999` 。用户看到的已经不是真实的同步了！

为了过渡到一个更加健壮的 Vue 2.0 的方案，让我们首先在一个新的 `<currency-input>` 组件中包裹这个过滤器：



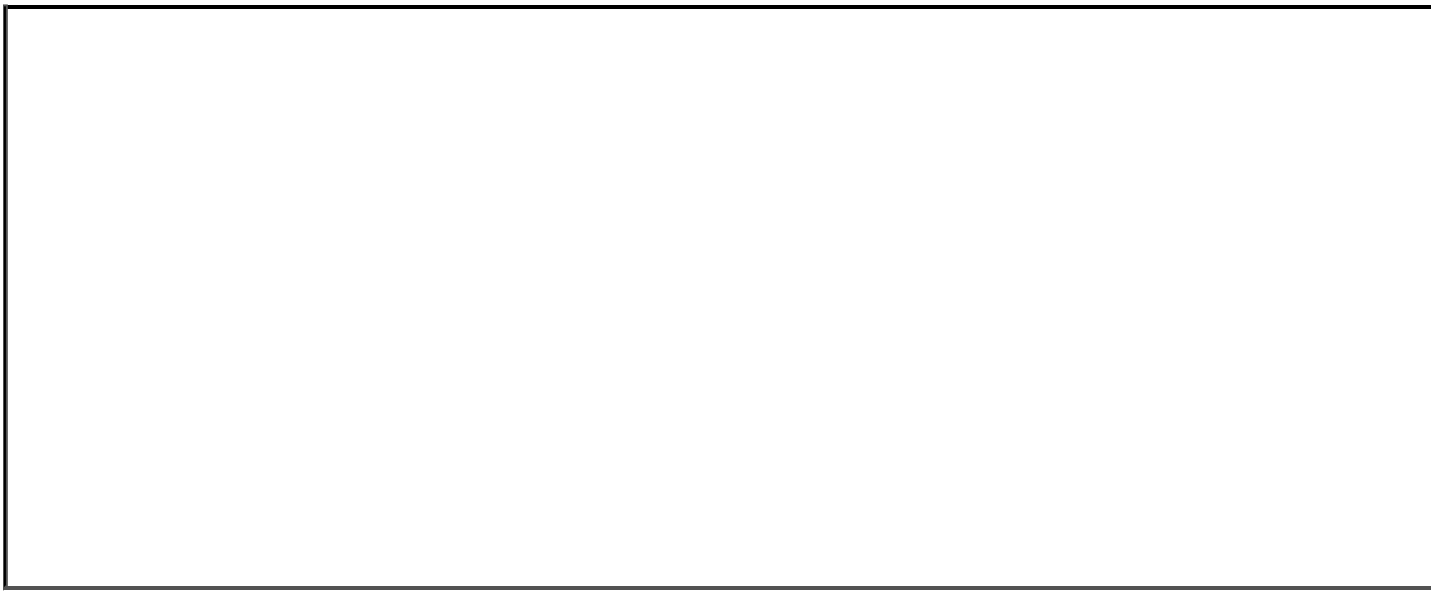
它允许我们添加独立过滤器无法封装的行为，比如选择输入框聚焦的内容。下一步我们从过滤器中提取业务逻辑。接下来是我们把所有的东西放到一个外部的 `currencyValidator` 对象中：



这会更加模块化，不只是更容易的迁移到 Vue 2，同时也允许汇率间隙和格式化：

- 从你的 Vue 代码中独立出来进行单元测试
- 在你的应用程序的别的部分中使用，比如验证验证一个 API 端的负荷

把这个验证器提取出来之后，我们也可以更舒适的把它构建到更健壮的解决方案中。那些古怪的状态也消除了，用户不再可能会输入错误，就像浏览器原生的数字输入框一样。
然而在 Vue 1.0 的过滤器中，我们仍然是受限的，所以还是完全升级到 Vue 2.0 吧：



你可能注意到了：

- 我们的输入框的各方面都更显性，使用生命周期钩子和 DOM 事件以替代双向过滤器的隐藏行为。
- 我们现在可以在自定义输入框中直接使用 `v-model` ，其不只是固定配合正常的输入框来使用，这也意味着我们的组件是对 Vuex 友好的。
- 因为我们已经不再要求过滤器选项必须要有一个返回值，所以实际上我们的汇率工作可以异步完成。这意味着如果我们有很多应用需要和汇率打交道，我们可以轻松的提炼这个逻辑并成为一个共享的微服务。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到在例如 <code>v-model</code> 的指令中使用过滤器的例子。如果你错过了，则应该会收到<strong>命令行报错</strong>。</p>
</div>
```

插槽

重名的插槽 移除

同一模板中的重名 `<slot>` 已经弃用。当一个插槽已经被渲染过了，那么就不能在同一模板其它地方被再次渲染了。如果要在不同位置渲染同一内容，可以用 prop 来传递。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>更新后运行测试，查看<strong>控制台警告信息</strong> 关于重名 slots 的提示 <code>v-model</code>。</p>
</div>
```

`slot` 样式参数 移除

通过具名 `<slot>` 插入的片段不再保持 `slot` 的参数。请用一个包裹元素来控制样式。或者用更高级方法：通过编程方式修改内容 ：render functions。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到选择 slots 标签 CSS 选择器 (例如：<code>[slot="my-slot-name"]</code>)。</p>
</div>
```

特殊属性

`keep-alive` 属性 替换

`keep-alive` 不再是一个特殊属性而是一个包裹组件，类似于 `<transition>` 比如：

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

这样可以在含多种状态子组件中使用 `<keep-alive>` ：

```
<keep-alive>
  <todo-list v-if="todos.length > 0"></todo-list>
  <no-todos-gif v-else></no-todos-gif>
</keep-alive>
```

当 `` 含有不同子组件时，应该分别影响到每一个子组件。不仅是第一个而是所有的子组件都将被忽略。

和 `<transition>` 一起使用时，确保把内容包裹在内：

```
<transition>
  <keep-alive>
    <component v-bind:is="view"></component>
  </keep-alive>
</transition>
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到<
code>keep-alive</code> 属性。</p>
</div>
```

计算插值

属性内部的计算插值 移除

属性内部的计算插值已经不能再使用了：

```
<button class="btn btn-{{ size }}"></button>
```

应该写成行内表达式：

```
<button v-bind:class="'btn btn-' + size"></button>
```

或者计算属性：

```
<button v-bind:class="buttonClasses"></button>
```

```
computed: {
  buttonClasses: function () {
    return 'btn btn-' + size
  }
```

```
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
属性内部的计算插值</p>
</div>
```

HTML 计算插值 移除

HTML 的计算插值（ `{{{ foo }}}` ）已经移除，取代的是 `v-html` 指令。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
 HTML 计算插值。</p>
</div>
```

单次绑定替换

单次绑定（ `{{* foo }}` ）已经被新的 `v-once` directive 取代。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找
到单次绑定使用位置。</p>
</div>
```

响应

`vm.$watch` changed

通过 `vm.$watch` 创建的观察器现在将在组件渲染时被激活。这样可以让你在组件渲染前更新状态，不用做不必要的更新。比如可以通过观察组件的 prop 变化来更新组件本身的值。

如果以前通过 `vm.$watch` 在组件更新后与 DOM 交互，现在就可以通过 `updated` 生命周期钩子来做这些。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行测试，如果有依赖于老方法的观察器将弹出 <strong>failed tests</strong>。</p>
</div>
```

`vm.$set` 变更

`vm.$set` 只是 `Vue.set` 的别名。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到过时的用法</p>
</div>
```

`vm.$delete` 变更

`vm.$delete` 现在只是： `Vue.delete` 别名。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到过时的用法</p>
</div>
```

`Array.prototype.$set` 弃用

用 `Vue.set` 替代

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到数组上的<code>.$set</code>。如有遗漏请参考<strong>控制台错误信息</strong>。</p>
</div>
```

`Array.prototype.$remove` 移除

用 `Array.prototype.splice` 替代，例如：

```
methods: {
  removeTodo: function (todo) {
    var index = this.todos.indexOf(todo)
    this.todos.splice(index, 1)
  }
}
```

或者更好的方法，直接给除去的方法一个 index 参数：

```
methods: {
```

```
  removeTodo: function (index) {
    this.todos.splice(index, 1)
  }
}
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到数组上的<code>.$remove</code>。如有遗漏请参考<strong>控制台错误信息</strong></p>
</div>
```

Vue 实例上的 `Vue.set` 和 `Vue.delete` 移除

`Vue.set` 和 `Vue.delete` 在实例上将不再起作用。现在都强制在实例的 data 选项中声明所有顶级响应值。如果删除实例属性或实例 `$data` 上的某个值，直接将它设置为 null 即可。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到实例中的 <code>Vue.set</code> 或 <code>Vue.delete</code> 。如有遗漏请参考<strong>控制台错误信息</strong>。</p>
</div>
```

替换 `vm.$data` 移除

现在禁止替换实例的 $data。这样防止了响应系统的一些极端情况并且让组件状态更加可控可预测 (特别是对于存在类型检查的系统)。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到覆盖 <code>vm.$data</code>的位置。如有遗漏请参考<strong>控制台警告信息</strong>。</p>
</div>
```

`vm.$get` 移除

可以直接取回响应数据。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到 <code>vm.$get</code> 的位置。如有遗漏请参考 <strong>控制台错误信息</strong>。</p>
</div>
```

围绕 DOM 的实例方法

`vm.$appendTo` 移除

使用 DOM 原生方法：

```
myElement.appendChild(vm.$el)
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
<code>vm.$appendTo</code> 的位置。如果有遗漏可以参考<strong>控制台错误信息</strong>。</
p>
</div>
```

`vm.$before` 移除

使用 DOM 原生方法：

```
myElement.parentNode.insertBefore(vm.$el, myElement)
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找
到 <code>vm.$before</code>。如有遗漏，请参考 <strong>控制台错误信息</strong>。</p>
</div>
```

`vm.$after` 移除

使用 DOM 原生方法：

```
myElement.parentNode.insertBefore(vm.$el, myElement.nextSibling)
```

如果 `myElement` 是最后一个节点也可以这样写：

```
myElement.parentNode.appendChild(vm.$el)
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
```

```
<code>vm.$after</code> 的位置。如有遗漏，请参考<strong>控制台错误信息</strong>。</p>
</div>
```

`vm.$remove` 移除

使用 DOM 原生方法：

```
vm.$el.remove()
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
<code>vm.$remove</code>。如有遗漏，请参考<strong>控制台错误信息</strong>。</p>
</div>
```

底层实例方法

`vm.$eval` 移除

尽量不要使用，如果必须使用该功能并且不肯定如何使用请参考 the forum。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
使用 <code>vm.$eval</code> 的位置。如有遗漏请参考<strong>控制台错误信息</strong>。</p>
</div>
```

`vm.$interpolate` 移除

尽量不要使用，如果必须使用该功能并且不肯定如何使用请参考 the forum。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到<
code>vm.$interpolate</code>。如有遗漏请参考<strong>控制台错误信息</strong>.</p>
</div>
```

`vm.$log` 移除

请使用 Vue Devtools 感受最佳 debug 体验。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
<code>vm.$log</code>。如有遗漏请参考<strong>控制台错误信息</strong>.</p>
</div>
```

实例 DOM 选项

`replace: false` 移除

现在组件总是会替换掉他们被绑定的元素。为了模仿 `replace: false` 的行为，可以用一个和将要替换元素类似的元素将根组件包裹起来：

```
new Vue({
  el: '#app',
  template: '<div id="app"> ... </div>'
})
```

或者使用渲染函数：

```
new Vue({
  el: '#app',
  render: function (h) {
    h('div', {
      attrs: {
        id: 'app',
      }
    }, /* ... */)
  }
})
```

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到
<code>replace: false</code>使用的位置。</p>
</div>
```

全局配置

`Vue.config.debug` 移除

不再需要，因为警告信息将默认在堆栈信息里输出。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到包含<code>Vue.config.debug</code>的地方。</p>
</div>
```

`Vue.config.async` 移除

异步操作现在需要渲染性能的支持。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到使用<code>Vue.config.async</code>的实例。</p>
</div>
```

`Vue.config.delimiters` 替换

以模板选项的方式使用。这样可以在使用自定义分隔符时避免影响第三方模板。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到使用<code>Vue.config.delimiters</code>的实例。</p>
</div>
```

`Vue.config.unsafeDelimiters` 移除

HTML 插值替换为 `v-html` 。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到<code>Vue.config.unsafeDelimiters</code>。然后 helper 工具也会找到 HTML 插入的实例，可以用`v-html`来替换。</p>
</div>
```

全局 API

带 `el` 的 `Vue.extend` 移除

el 选项不再在 `Vue.extend` 中使用。仅在实例创建参数中可用。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>更新后运行测试在<strong>控制台警告信息</strong>中找到关于带有<code>Vue.extend</code>的<code>el</code>。</p>
</div>
```

`Vue.elementDirective` 移除

用组件来替代

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到包含<code>Vue.elementDirective</code>的实例。</p>
</div>
```

`Vue.partial` 移除

Partials 已被移除，取而代之的是更明确的组件之间的数据流--props。除非你正在使用一个部分性能关键型区域，否则建议只使用一个 normal component 来代替。如果你是动态绑定部分的 `name` ，您可以使用 dynamic component。

如果你碰巧在你的应用程序的性能关键部分使用 partials，那么你应该升级到函数式组件。它们必须在纯 JS / JSX 文件中 (而不是在 `.vue` 文件中)，并且是无状态的和无实例的，就像 partials。这使得渲染极快。

函数式组件相对于 partials 一个好处是它们可以更具动态性，因为它们允许您访问 JavaScript 的全部功能。然而，这是有成本的。如果你从来没有使用过渲染式的组件框架，你可能需要花费更长的时间来学习它们。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</a>找到包含 <code>Vue.partial</code>的实例</p>
</div>
```

从 Vue Router 0.7.x 迁移

> 只有 Vue Router 2 是与 Vue 2 相互兼容的，所以如果你更新了 Vue ，你也需要更新 Vue Router 。这也是我们在主文档中将迁移路径的详情添加进来的原因。
>
> 有关使用 Vue Router 2 的完整教程，请参阅 Vue Router 文档。

Router 初始化

`router.start` 替换

不再会有一个特殊的 API 用来初始化包含 Vue Router 的 app ，这意味着不再是：

```
router.start({
  template: '<router-view></router-view>'
}, '#app')
```

你只需要传一个路由属性给 Vue 实例：

```
new Vue({
  el: '#app',
  router: router,
  template: '<router-view></router-view>'
})
```

或者，如果你使用的是运行时构建 (runtime-only) 方式：

```
new Vue({
  el: '#app',
  router: router,
  render: h => h('router-view')
})
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>router.start</code> 被调用的示例。</p>
</div>
```

Route 定义

`router.map` 替换

路由现在被定义为一个在 router 实例里的一个 `routes` 选项数组。所以这些路由：

```
router.map({
  '/foo': {
    component: Foo
  },
  '/bar': {
    component: Bar
  }
})
```

会以这种方式定义：

```
var router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
  ]
})
```

考虑到不同浏览器中遍历对象不能保证会使用相同的键值，这种数组的语法可以保证更多可预测的路由匹配。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>router.map</code> 被调用的示例。</p>
</div>
```

`router.on` 移除

如果你需要在启动的 app 时通过代码生成路由，你可以动态地向路由数组推送定义来完成这个操作。举个例子：

```
// 普通的路由
var routes = [
  // ...
]

// 动态生成的路由
marketingPages.forEach(function (page) {
  routes.push({
    path: '/marketing/' + page.slug
    component: {
      extends: MarketingComponent
      data: function () {
        return { page: page }
```

```
      }
    }
  })
})

var router = new Router({
  routes: routes
})
```

如果你需要在 router 被实例化后增加新的路由，你可以把 router 原来的匹配方式换成一个包括你新添的加路由的匹配方式：

```
router.match = createMatcher(
  [{
    path: '/my/new/path',
    component: MyComponent
  }].concat(router.options.routes)
)
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>router.on</code> 被调用的示例。</p>
</div>
```

`router.beforeEach` changed

`router.beforeEach` 现在是异步工作的，并且携带一个 `next` 函数作为其第三个参数。

```
router.beforeEach(function (transition) {
  if (transition.to.path === '/forbidden') {
    transition.abort()
  } else {
    transition.next()
  }
})
```

```
router.beforeEach(function (to, from, next) {
  if (to.path === '/forbidden') {
    next(false)
  } else {
    next()
  }
})
```

`subRoutes` 换名

出于 Vue Router 和其他路由库一致性的考虑，重命名为 `children`

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>subRoutes</code> 选项的示例。</p>
</div>
```

`router.redirect` 替换

现在用一个路由定义的选项作为代替。举个例子，你将会更新：

```
router.redirect({
  '/tos': '/terms-of-service'
})
```

成像下面的 `routes` 配置里定义的样子：

```
{
  path: '/tos',
  redirect: '/terms-of-service'
}
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>router.redirect</code> 被调用的示例。</p>
</div>
```

`router.alias` 替换

现在是你进行 alias 操作的路由定义里的一个选项。举个例子，你需要在你的 `routes` 定义里将：

```
router.alias({
  '/manage': '/admin'
})
```

配置这个样子：

```
{
```

```
  path: '/admin',
  component: AdminPanel,
  alias: '/manage'
}
```

如果你需要进行多次 alias 操作，你也可以使用一个数组语法去实现：

```
alias: ['/manage', '/administer', '/administrate']
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行<a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a>找到
<code>router.alias</code> 被调用的示例。</p>
</div>
```

任意的 Route 属性 替换

现在任意的 route 属性必须在新 meta 属性的作用域内，以避免和以后的新特性发生冲突。举个例子，如果你以前这样定义：

```
'/admin': {
  component: AdminPanel,
  requiresAuth: true
}
```

你现在需要把它更新成：

```
{
  path: '/admin',
  component: AdminPanel,
  meta: {
    requiresAuth: true
  }
}
```

如果在一个路由上访问一个属性，你仍然会通过 meta 。举个例子：

```
if (route.meta.requiresAuth) {
  // ...
}
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到任意的路由不在 meta 作用域下的示例。</p>
</div>
```

URL 中的 Query 数组 [] 语法 移除

当传递数组给 query 参数时，URL 语法不再是 `/foo?users[]=Tom&users[]=Jerry` ，取而代之，新语法是 `/foo?users=Tom&users=Jerry` ，此时 `$route.query.users` 将仍旧是一个数组，不过如果在该 query 中只有一个参数： `/foo?users=Tom` ，当直接访问该路由时，vue-router 将无法知道我们期待的 `users` 是个数组。因此，可以考虑添加一个计算属性并且在每个使用 `$route.query.users` 的地方以该计算属性代替：

```
export default {
  // ...
  computed: {
    // 此计算属性将始终是个数组
    users () {
      const users = this.$route.query.users
      return Array.isArray(users) ? users : [users]
    }
  }
}
```

Route 匹配

路由匹配现在使用 path-to-regexp 这个包，这将会使得工作与之前相比更加灵活。

一个或者更多的命名参数 改变

语法稍微有些许改变，所以以 `/category/*tags` 为例，应该被更新为 `/category/:tags+` 。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到弃用路由语法的示例。</p>
</div>
```

链接

`v-link` 替换

`v-link` 指令已经被一个新的 `<router-link>` 组件指令替代，这一部分的工作已经被 Vue 2 中的组件完成。这将意味着在任何情况下，如果你拥有这样一个链接：

```
<a v-link="'/about'">About</a>
```

你需要把它更新成：

```
<router-link to="/about">About</router-link>
```

注意：`<router-link>` 不支持 `target="_blank"` 属性，如果你想打开一个新标签页，你必须用 `<a>` 标签。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>v-link</code> 指令的示例。</p>
</div>
```

`v-link-active` 替换

`v-link-active` 也因为指定了一个在 `<router-link>` 组件上的 tag 属性而被弃用了。举个例子，你需要更新：

```
<li v-link-active>
  <a v-link="'/about'">About</a>
</li>
```

成这个写法：

```
<router-link tag="li" to="/about">
  <a>About</a>
</router-link>
```

`<a>` 标签将会成为真实的链接 (并且可以获取到正确的跳转)，但是激活的类将会被应用在外部的 `<li>` 标签上。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>v-link-active</code> 指令的示例</p>
</div>
```

编程导航

`router.go` 改变

为了与 HTML5 History API 保持一致性， `router.go` 已经被用来作为 后退/前进导航， `router.push` 用来导向特殊页面。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> ，找到 <code>router.go</code> 和 <code>router.push</code> 指令被调用的示例。</p>
</div>
```

路由选择：Modes

`hashbang: false` 移除

Hashbangs 将不再为谷歌需要去爬去一个网址，所以他们将不再成为哈希策略的默认选项。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>hashbang: false</code> 选项的示。</p>
</div>
```

`history: true` 替换

所有路由模型选项将被简化成一个单个的 `mode` 选项。你需要更新：

```
var router = new VueRouter({
  history: 'true'
})
```

成这个写法：

```
var router = new VueRouter({
  mode: 'history'
})
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找
```

```
到 <code>history: true</code> 选项的示。</p>
</div>
```

`abstract: true` 替换

所有路由模型选项将被简化成一个单个的 `mode` 选项。你需要更新：

```
var router = new VueRouter({
  abstract: 'true'
})
```

成这个写法：

```
var router = new VueRouter({
  mode: 'abstract'
})
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>abstract: true</code> 选项的示例。</p>
</div>
```

路由选项：Misc

`saveScrollPosition` 替换

它已经被替换为可以接受一个函数的 `scrollBehavior` 选项，所以滑动行为可以完全的被定制化处理 - 甚至为每次路由进行定制也可以满足。这将会开启很多新的可能，但是简单的复制旧的行为：

```
saveScrollPosition: true
```

你可以替换为：

```
scrollBehavior: function (to, from, savedPosition) {
  return savedPosition || { x: 0, y: 0 }
}
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>saveScrollPosition: true</code> 选项的示例。</p>
```

```
</div>
```

`root` 换名

为了与 HTML 的 `<base>` 标签保持一致性，重命名为 `base` 。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>root</code> 选项的示例。</p>
</div>
```

`transitionOnLoad` 移除

由于 Vue 的过渡系统 `appear` transition control 的存在，这个选项将不再需要。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>transitionOnLoad: true</code> 选项的示例。</p>
</div>
```

`suppressTransitionError` 移除

出于简化钩子的考虑被移除。如果你真的需要抑制过渡错误，你可以使用 `try … catch` 作为替代。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移指令</a> 找到 <code>suppressTransitionError: true</code> 选项的示例。</p>
</div>
```

路由挂钩

`activate` 替换

使用 `beforeRouteEnter` 这一组件进行替代。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>beforeRouteEnter</code> 钩子的示例。</p>
</div>
```

`canActivate` 替换

使用 `beforeEnter` 在路由中作为替代。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>canActivate</code> 钩子的示例。</p>
</div>
```

`deactivate` 移除

使用 `beforeDestroy` 或者 `destroyed` 钩子作为替代。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>deactivate</code> 钩子的示例。</p>
</div>
```

`canDeactivate` 替换

在组件中使用 `beforeRouteLeave` 作为替代。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移路径</a> 找到 <code>canDeactivate</code> 钩子的示例。</p>
</div>
```

`canReuse: false` 移除

在新的 Vue 路由中将不再被使用。

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>canReuse: false</code> 选项的示例。</p>
</div>
```

`data` 替换

`$route` 属性是响应式的，所以你可以就使用一个 watcher 去响应路由的改变，就像这样：

```
watch: {
  '$route': 'fetchData'
},
methods: {
  fetchData: function () {
    // ...
  }
}
```

```
<div class="upgrade-path">
  <h4>升级路径</h4>
  <p>运行 <a href="https://github.com/vuejs/vue-migration-helper">迁移助手</a> 找到 <code>data</code> 钩子的示例。</p>
</div>
```

`$loadingRouteData` 移除

定义你自己的属性 (例如： `isLoading`)，然后在路由上的 watcher 中更新加载状态。举个例子，如果使用 axios 获取数据：

```
data: function () {
  return {
    posts: [],
    isLoading: false,
    fetchError: null
  }
},
watch: {
  '$route': function () {
    var self = this
    self.isLoading = true
    self.fetchData().then(function () {
      self.isLoading = false
    })
  }
},
methods: {
  fetchData: function () {
    var self = this
    return axios.get('/api/posts')
      .then(function (response) {
        self.posts = response.data.posts
      })
      .catch(function (error) {
        self.fetchError = error
      })
  }
```

```
}
```

从 Vuex 0.6.x 迁移到 1.0

title: 从 Vuex 0.6.x 迁移到 1.0
type: guide
order: 703

> Vuex 2.0 已经发布了，但是这份指南只涵盖迁移到 1.0？这是打错了吗？此外，似乎 Vuex 1.0 和 2.0 也同时发布。这是怎么回事？我该用哪一个并且哪一个兼容 Vue 2.0呢？

Vuex 1.0 和 2.0 如下：

- 都完全支持 Vue 1.0 和 2.0
- 将在可预见的未来保留支持

然而它们的目标用户稍微有所不同。
Vuex 2.0 从根本上重新设计并且提供简洁的 API，用于帮助正在开始一个新项目的用户，或想要用客户端状态管理前沿技术的用户。此迁移指南不涵盖 Vuex 2.0 相关内容，因此如果你想了解更多，请查阅 Vuex 2.0 文档。
Vuex 1.0 主要是向下兼容，所以升级只需要很小的改动。推荐拥有大量现存代码库的用户，或只想尽可能平滑升级 Vue 2.0 的用户。这份指南致力促进这一过程，但仅包括迁移说明。完整使用指南请查阅 Vuex 1.0 文档。

带字符串属性路径的 `store.watch` 替换

`store.watch` 现在只接受函数。因此，下面例子你需要替换：

```
store.watch('user.notifications', callback)
```

为：

```
store.watch(
  // 当返回结果改变...
  function (state) {
    return state.user.notifications
  },
  // 执行回调函数
  callback
)
```

这帮助你更加完善的控制那些需要监听的响应式属性。

```
<div class="upgrade-path">
```

```
  <h4>升级方法</h4>
  <p>在代码库运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</
a>，查找在 <code>store.watch</code> 中使用字符串作为第一个参数的事例。</p>
</div>
```

Store 的事件触发器 移除

store 实例不再暴露事件触发器 (event emitter) 接口（ `on` ， `off` ， `emit` ）。如果你之前使用 store 作为全局的 event bus，迁移说明相关内容请查阅此章节。

为了替换正在使用观察 store 自身触发事件的这些接口，(例如： `store.on('mutation', callback)`)，我们引入新的方法 `store.subscribe` 。在插件中的典型使用方式如下：

```
var myPlugin = store => {
  store.subscribe(function (mutation, state) {
    // Do something...
  })
}
```

更多信息请查阅插件文档的示例。

```
<div class="upgrade-path">
  <h4>升级方式</h4>
  <p>在代码库运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</
a>，查找使用了 <code>store.on</code>, <code>store.off</code>, <code>store.emit</c
ode> 的事例。</p>
</div>
```

中间件 替换

中间件被替换为插件。插件是接收 store 作为仅有参数的基本函数，能够监听 store 中的 mutation 事件：

```
const myPlugins = store => {
  store.subscribe('mutation', (mutation, state) => {
    // Do something...
  })
}
```

更多详情，请查阅 插件文档。

```
<div class="upgrade-path">
```

```
  <h4>升级方法</h4>
  <p>在代码库运行<a href="https://github.com/vuejs/vue-migration-helper">迁移工具</
a>，查找使用了 <code>middlewares</code> 选项的事例。</p>
</div>
```

更多

对比其他框架

这个页面无疑是最难编写的，但我们认为它也是非常重要的。或许你曾遇到了一些问题并且已经用其他的框架解决了。你来这里的目的是看看 Vue 是否有更好的解决方案。这也是我们在此想要回答的。
客观来说，作为核心团队成员，显然我们会更偏爱 Vue，认为对于某些问题来讲用 Vue 解决会更好。如果没有这点信念，我们也就不会整天为此忙活了。但是在此，我们想尽可能地公平和准确地来描述一切。其他的框架也有显著的优点，例如 React 庞大的生态系统，或者像是 Knockout 对浏览器的支持覆盖到了 IE6。我们会尝试着把这些内容全部列出来。
我们也希望得到你的帮助，来使文档保持最新状态，因为 JavaScript 的世界进步的太快。如果你注意到一个不准确或似乎不太正确的地方，请提交问题让我们知道。

React

React 和 Vue 有许多相似之处，它们都有：

- 使用 Virtual DOM
- 提供了响应式 (Reactive) 和组件化 (Composable) 的视图组件。
- 将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库。

由于有着众多的相似处，我们会用更多的时间在这一块进行比较。这里我们不只保证技术内容的准确性，同时也兼顾了平衡的考量。我们需要承认 React 比 Vue 更好的地方，比如更丰富的生态系统。
下列部分章节会略微有些过时，因为最近 React 16+ 的发布，我们计划在不久的将来和 React 社区一起重写这部分内容。

运行时性能

React 和 Vue 都是非常快的，所以速度并不是在它们之中做选择的决定性因素。对于具体的数据表现，可以移步这个第三方 benchmark，它专注于渲染/更新非常简单的组件树的真实性能。

优化

在 React 应用中，当某个组件的状态发生变化时，它会以该组件为根，重新渲染整个组件子树。
如要避免不必要的子组件的重渲染，你需要在所有可能的地方使用 `PureComponent` ，或是手动实现 `shouldComponentUpdate` 方法。同时你可能会需要使用不可变的数据结构来使得你的组件更容易被优化。
然而，使用 `PureComponent` 和 `shouldComponentUpdate` 时，需要保证该组件的整个子树的渲染输出都是由该组件的 props 所决定的。如果不符合这个情况，那么此类优化就会导致难以察觉的渲染结果不一致。这使得 React 中的组件优化伴随着相当的心智负担。
在 Vue 应用中，组件的依赖是在渲染过程中自动追踪的，所以系统能精确知晓哪个组件确实需要被重渲染。你可以理解为每一个组件都已经自动获得了 `shouldComponentUpdate` ，并且没有上述的子树问题限制。

Vue 的这个特点使得开发者不再需要考虑此类优化，从而能够更好地专注于应用本身。

HTML & CSS

在 React 中，一切都是 JavaScript。不仅仅是 HTML 可以用 JSX 来表达，现在的潮流也越来越多地将 CSS 也纳入到 JavaScript 中来处理。这类方案有其优点，但也存在一些不是每个开发者都能接受的取舍。
Vue 的整体思想是拥抱经典的 Web 技术，并在其上进行扩展。我们下面会详细分析一下。

JSX vs Templates

在 React 中，所有的组件的渲染功能都依靠 JSX。JSX 是使用 XML 语法编写 JavaScript 的一种语法糖。
使用 JSX 的渲染函数有下面这些优势：

- 你可以使用完整的编程语言 JavaScript 功能来构建你的视图页面。比如你可以使用临时变量、JS 自带的流程控制、以及直接引用当前 JS 作用域中的值等等。
- 开发工具对 JSX 的支持相比于现有可用的其他 Vue 模板还是比较先进的 (比如，linting、类型检查、编辑器的自动完成)。

事实上 Vue 也提供了渲染函数，甚至支持 JSX。然而，我们默认推荐的还是模板。任何合乎规范的 HTML 都是合法的 Vue 模板，这也带来了一些特有的优势：

- 对于很多习惯了 HTML 的开发者来说，模板比起 JSX 读写起来更自然。这里当然有主观偏好的成分，但如果这种区别会导致开发效率的提升，那么它就有客观的价值存在。
- 基于 HTML 的模板使得将已有的应用逐步迁移到 Vue 更为容易。
- 这也使得设计师和新人开发者更容易理解和参与到项目中。
- 你甚至可以使用其他模板预处理器，比如 Pug 来书写 Vue 的模板。

有些开发者认为模板意味着需要学习额外的 DSL (Domain-Specific Language 领域特定语言) 才能进行开发——我们认为这种区别是比较肤浅的。首先，JSX 并不是免费的——它是基于 JS 之上的一套额外语法，因此也有它自己的学习成本。同时，正如同熟悉 JS 的人学习 JSX 会很容易一样，熟悉 HTML 的人学习 Vue 的模板语法也是很容易的。最后，DSL 的存在使得我们可以让开发者用更少的代码做更多的事，比如 `v-on` 的各种修饰符，在 JSX 中实现对应的功能会需要多得多的代码。
更抽象一点来看，我们可以把组件区分为两类：一类是偏视图表现的 (presentational)，一类则是偏逻辑的 (logical)。我们推荐在前者中使用模板，在后者中使用 JSX 或渲染函数。这两类组件的比例会根据应用类型的不同有所变化，但整体来说我们发现表现类的组件远远多于逻辑类组件。

组件作用域内的 CSS

除非你把组件分布在多个文件上 (例如 CSS Modules)，CSS 作用域在 React 中是通过 CSS-in-JS 的方案实现的 (比如 styled-components、glamorous 和 emotion)。这引入了一个新的面向组件的样式范例，它和普通的 CSS 撰写过程是有区别的。另外，虽然在构建时将 CSS 提取到一个单独的样式表是支持的，但 bundle 里通常还

是需要一个运行时程序来让这些样式生效。当你能够利用 JavaScript 灵活处理样式的同时，也需要权衡 bundle 的尺寸和运行时的开销。

如果你是一个 CSS-in-JS 的爱好者，许多主流的 CSS-in-JS 库也都支持 Vue (比如 styled-components-vue 和 vue-emotion)。这里 React 和 Vue 主要的区别是，Vue 设置样式的默认方法是单文件组件里类似 `style` 的标签。

单文件组件让你可以在同一个文件里完全控制 CSS，将其作为组件代码的一部分。

```
<style scoped>
  @media (min-width: 250px) {
    .list-container:hover {
      background: orange;
    }
  }
</style>
```

这个可选 `scoped` 属性会自动添加一个唯一的属性 (比如 `data-v-21e5b78`) 为组件内 CSS 指定作用域，编译的时候 `.list-container:hover` 会被编译成类似 `.list-container[data-v-21e5b78]:hover`。

最后，Vue 的单文件组件里的样式设置是非常灵活的。通过 vue-loader，你可以使用任意预处理器、后处理器，甚至深度集成 CSS Modules——全部都在 `<style>` 标签内。

规模

向上扩展

Vue 和 React 都提供了强大的路由来应对大型应用。React 社区在状态管理方面非常有创新精神 (比如 Flux、Redux)，而这些状态管理模式甚至 Redux 本身也可以非常容易的集成在 Vue 应用中。实际上，Vue 更进一步地采用了这种模式 (Vuex)，更加深入集成 Vue 的状态管理解决方案 Vuex 相信能为你带来更好的开发体验。

两者另一个重要差异是，Vue 的路由库和状态管理库都是由官方维护支持且与核心库同步更新的。React 则是选择把这些问题交给社区维护，因此创建了一个更分散的生态系统。但相对的，React 的生态系统相比 Vue 更加繁荣。

最后，Vue 提供了 Vue-cli 脚手架，能让你非常容易地构建项目，包含了 Webpack，Browserify，甚至 no build system。React 在这方面也提供了 create-react-app，但是现在还存在一些局限性：

- 它不允许在项目生成时进行任何配置，而 Vue 支持 Yeoman-like 定制。
- 它只提供一个构建单页面应用的单一模板，而 Vue 提供了各种用途的模板。
- 它不能用用户自建的模板构建项目，而自建模板对企业环境下预先建立协议是特别有用的。

而要注意的是这些限制是故意设计的，这有它的优势。例如，如果你的项目需求非常简单，你就不需要自定义生成过程。你能把它作为一个依赖来更新。如果阅读更多关于不同的设计理念。

向下扩展

React 学习曲线陡峭，在你开始学 React 前，你需要知道 JSX 和 ES2015，因为许多示例用的是这些语法。你需要学习构建系统，虽然你在技术上可以用 Babel 来实时编译代码，但是这并不推荐用于生产环境。
就像 Vue 向上扩展好比 React 一样，Vue 向下扩展后就类似于 jQuery。你只要把如下标签放到页面就可以运行：

```
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

然后你就可以编写 Vue 代码并应用到生产中，你只要用 min 版 Vue 文件替换掉就不用担心其他的性能问题。
由于起步阶段不需学 JSX，ES2015 以及构建系统，所以开发者只需不到一天的时间阅读指南就可以建立简单的应用程序。

原生渲染

React Native 能使你用相同的组件模型编写有本地渲染能力的 APP (iOS 和 Android)。能同时跨多平台开发，对开发者是非常棒的。相应地，Vue 和 Weex 会进行官方合作，Weex 是阿里巴巴发起的跨平台用户界面开发框架，同时也正在 Apache 基金会进行项目孵化，Weex 允许你使用 Vue 语法开发不仅仅可以运行在浏览器端，还能被用于开发 iOS 和 Android 上的原生应用的组件。
在现在，Weex 还在积极发展，成熟度也不能和 React Native 相抗衡。但是，Weex 的发展是由世界上最大的电子商务企业的需求在驱动，Vue 团队也会和 Weex 团队积极合作确保为开发者带来良好的开发体验。
另一个 Vue 的开发者们很快就会拥有的选项是 NativeScript，这是一个社区驱动的插件。

MobX

Mobx 在 React 社区很流行，实际上在 Vue 也采用了几乎相同的反应系统。在有限程度上，React + Mobx 也可以被认为是更繁琐的 Vue，所以如果你习惯组合使用它们，那么选择 Vue 会更合理。

Preact 和其它类 React 库

类 React 的库们往往尽可能地与 React 共享 API 和生态。因此上述比较对它们来说也同样适用。它们和 React 的不同往往在于更小的生态。因为这些库无法 100% 兼容 React 生态中的全部，部分工具和辅助库也可能无法使用。或者即使看上去能工作，但也有可能随时发生不兼容，除非你用的这个类 React 库官方与 React 保持严格一致。

AngularJS (Angular 1)

Vue 的一些语法和 AngularJS 的很相似 (例如 `v-if` vs `ng-if`)。因为 AngularJS 是 Vue 早期开发的灵感来源。然而，AngularJS 中存在的许多问题，在 Vue 中已经得到解决。

复杂性

在 API 与设计两方面上 Vue.js 都比 AngularJS 简单得多，因此你可以快速地掌握它的全部特性并投入开发。

灵活性和模块化

Vue.js 是一个更加灵活开放的解决方案。它允许你以希望的方式组织应用程序，而不是在任何时候都必须遵循 AngularJS 制定的规则，这让 Vue 能适用于各种项目。我们知道把决定权交给你是非常必要的。
这也就是为什么我们提供 webpack template，让你可以用几分钟，去选择是否启用高级特性，比如热模块加载、linting、CSS 提取等等。

数据绑定

AngularJS 使用双向绑定，Vue 在不同组件间强制使用单向数据流。这使应用中的数据流更加清晰易懂。

指令与组件

在 Vue 中指令和组件分得更清晰。指令只封装 DOM 操作，而组件代表一个自给自足的独立单元——有自己的视图和数据逻辑。在 AngularJS 中，每件事都由指令来做，而组件只是一种特殊的指令。

运行时性能

Vue 有更好的性能，并且非常非常容易优化，因为它不使用脏检查。
在 AngularJS 中，当 watcher 越来越多时会变得越来越慢，因为作用域内的每一次变化，所有 watcher 都要重新计算。并且，如果一些 watcher 触发另一个更新，脏检查循环 (digest cycle) 可能要运行多次。AngularJS 用户常常要使用深奥的技术，以解决脏检查循环的问题。有时没有简单的办法来优化有大量 watcher 的作用域。
Vue 则根本没有这个问题，因为它使用基于依赖追踪的观察系统并且异步队列更新，所有的数据变化都是独立触发，除非它们之间有明确的依赖关系。
有意思的是，Angular 和 Vue 用相似的设计解决了一些 AngularJS 中存在的问题。

Angular (原本的 Angular 2)

我们将新的 Angular 独立开来讨论，因为它是一个和 AngularJS 完全不同的框架。例如：它具有优秀的组件系统，并且许多实现已经完全重写，API 也完全改变了。

TypeScript

Angular 事实上必须用 TypeScript 来开发，因为它的文档和学习资源几乎全部是面向 TS 的。TS 有很多好处——静态类型检查在大规模的应用中非常有用，同时对于 Java 和 C# 背景的开发者也是非常提升开发效率的。
然而，并不是所有人都想用 TS——在中小型规模的项目中，引入 TS 可能并不会带来太多明显的优势。在这些情况下，用 Vue 会是更好的选择，因为在不用 TS 的情况下使用 Angular 会很有挑战性。
最后，虽然 Vue 和 TS 的整合可能不如 Angular 那么深入，我们也提供了官方的 类型声明 和 组件装饰器，并且知道有大量用户在生产环境中使用 Vue + TS 的组合。我们也和微软的 TS / VSCode 团队进行着积极的合作，目标是为 Vue + TS 用户提供更好的类型检查和 IDE 开发体验。

运行时性能

这两个框架都很快，有非常类似的 benchmark 数据。你可以浏览具体的数据做更细粒度的对比，不过速度应该

不是决定性的因素。

体积

在体积方面，最近的 Angular 版本中在使用了 AOT 和 tree-shaking 技术后使得最终的代码体积减小了许多。但即使如此，一个包含了 Vuex + Vue Router 的 Vue 项目 (gzip 之后 30kB) 相比使用了这些优化的 `angular-cli` 生成的默认项目尺寸 (~65KB) 还是要小得多。

灵活性

Vue 相比于 Angular 更加灵活，Vue 官方提供了构建工具来协助你构建项目，但它并不限制你去如何组织你的应用代码。有人可能喜欢有严格的代码组织规范，但也有开发者喜欢更灵活自由的方式。

学习曲线

要学习 Vue，你只需要有良好的 HTML 和 JavaScript 基础。有了这些基本的技能，你就可以非常快速地通过阅读 指南 投入开发。
Angular 的学习曲线是非常陡峭的——作为一个框架，它的 API 面积比起 Vue 要大得多，你也因此需要理解更多的概念才能开始有效率地工作。当然，Angular 本身的复杂度是因为它的设计目标就是只针对大型的复杂应用；但不可否认的是，这也使得它对于经验不甚丰富的开发者相当的不友好。

Ember

Ember 是一个全能框架。它提供了大量的约定，一旦你熟悉了它们，开发会变得很高效。不过，这也意味着学习曲线较高，而且并不灵活。这意味着在框架和库 (加上一系列松散耦合的工具) 之间做权衡选择。后者会更自由，但是也要求你做更多架构上的决定。
也就是说，我们最好比较的是 Vue 内核和 Ember 的模板与数据模型层：

- Vue 在普通 JavaScript 对象上建立响应，提供自动化的计算属性。在 Ember 中需要将所有东西放在 Ember 对象内，并且手工为计算属性声明依赖。
- Vue 的模板语法可以用全功能的 JavaScript 表达式，而 Handlebars 的语法和帮助函数相比来说非常受限。
- 在性能上，Vue 比 Ember 好很多，即使是 Ember 3.x 的最新 Glimmer 引擎。Vue 能够自动批量更新，而 Ember 在性能敏感的场景时需要手动管理。

Knockout

Knockout 是 MVVM 领域内的先驱，并且追踪依赖。它的响应系统和 Vue 也很相似。它在浏览器支持以及其他方面的表现也是让人印象深刻的。它最低能支持到 IE6，而 Vue 最低只能支持到 IE9。
随着时间的推移，Knockout 的发展已有所放缓，并且略显有点老旧了。比如，它的组件系统缺少完备的生命周期事件方法，尽管这些在现在是非常常见的。以及相比于 Vue 调用子组件的接口它的方法显得有点笨重。
如果你有兴趣研究，你还会发现二者在接口设计的理念上是不同的。这可以通过各自创建的 simple Todo List 体

现出来。或许有点主观，但是很多人认为 Vue 的 API 接口更简单结构更优雅。

Polymer

Polymer 是另一个由谷歌赞助的项目，事实上也是 Vue 的一个灵感来源。Vue 的组件可以粗略的类比于 Polymer 的自定义元素，并且两者具有相似的开发风格。最大的不同之处在于，Polymer 是基于最新版的 Web Components 标准之上，并且需要重量级的 polyfills 来帮助工作 (性能下降)，浏览器本身并不支持这些功能。相比而言，Vue 在支持到 IE9 的情况下并不需要依赖 polyfills 来工作。
在 Polymer 版本中，为了弥补性能，团队非常有限的使用数据绑定系统。例如，在 Polymer 中唯一支持的表达式只有布尔值否定和单一的方法调用，它的 computed 方法的实现也并不是很灵活。

Riot

Riot 3.0 提供了一个类似于基于组件的开发模型 (在 Riot 中称之为 Tag)，它提供了小巧精美的 API。Riot 和 Vue 在设计理念上可能有许多相似处。尽管相比 Riot ，Vue 要显得重一点，Vue 还是有很多显著优势的：

- 更好的性能。Riot 使用了 遍历 DOM 树 而不是虚拟 DOM，但实际上用的还是脏检查机制，因此和 AngularJS 患有相同的性能问题。
- 更多成熟工具的支持。Vue 提供官方支持 webpack 和 Browserify，而 Riot 是依靠社区来建立集成系统。

加入 Vue.js 社区

Vue.js 的社区正在急速增长中，如果你正在阅读本文，这说明你大概已经准备好加入 Vue.js 社区了。欢迎！
现在让我们来解答你能从社区中获得什么以及你能为社区做什么。

您将享有的资源

行为规范

这份行为规范是一个指南，它易于让我们所参与的技术社区更加繁荣。

获取帮助

- 论坛：咨询与 Vue 及其生态的相关问题的最佳地点。
- 聊天室：一个 Vue 开发者们相互认识和交流的实时聊天室。
- Meetup：想在当地找到像你一样的 Vue.js 爱好者吗？有兴趣成为社区领袖吗？这里就有你所需要的支持和帮助！
- GitHub：如果你想报告 bug 或者提出新特性需求，欢迎来 GitHub 提交 issue。我们也非常欢迎 pull request！

探索生态

- Awesome Vue：一览其他牛人发布的优秀资源。
- “Show and Tell”子论坛：又一个好地方，可以看看他人借助 Vue 生态完成的作品，以及他人为不断壮大的 Vue 生态的贡献。

您可以参与的方式

贡献代码

和所有的项目一样，贡献代码需要遵循规范。为了保证我们能尽快地帮助你解决问题或者接受你的 pull request，请先阅读这份贡献指南。
阅读之后，你应该已经准备好向 Vue 的核心仓库贡献代码了：

- vue：核心库
- vuex：类 Flux 的状态管理
- vue-router：为单页面应用提供的路由系统

……还有许多小型的官方同伴库。

分享 (并积累) 您的经验

除了在论坛或聊天室回答问题、分享资源外，还有一些其它的方式可以分享并增长你的见识：

- **开发学习资料。**我们常说，最好的学习方法就是教别人。如果你正在用 Vue 做一些有趣的事情，你可以写一篇博客、组织研讨会、甚至创建一个 gist 分享到社交平台上：这些都能加强你的专项知识。
- **关注 (watch) 你关心的仓库。**这样无论何时该仓库有新的动静，你都会第一时间收到通知，得到关于正在进行的讨论以及即将到来的新特性的新鲜情报。这是超棒的积累专业知识的方法，你最终将会有能力来解决问题 (issue) 并提交 pull request。

翻译文档

Vue 已经在全球范围内传播开来，核心团队成员甚至来自至少 6 个时区。论坛 已有 7 种语言，数字还在持续增长。我们许多文档都有积极维护的翻译。我们非常为 Vue 的国际影响力骄傲，但我们还能做得更好。
我希望现在你正在使用你的首选语言阅读这篇文档，如果不是，你愿意帮助我们翻译它吗？
如果你愿意的话，请随时 fork 这些文档或者官方维护的其他文档的仓库，然后开始翻译吧。只要你取得了进展，就请在主仓库开一个 issue 或者 pull request，我们将号召更多的贡献者来进行帮助。

成为社区领袖

在社区中，你可以做很多事情来帮助 Vue 的发展：

- **参加当地的 meetup。**不论是准备一个话题还是组织一个 workshop，你都可以通过帮助新老 Vue 开发者的发展来为社区带来很多价值。
- **自己组织 meetup。**如果你所在的地方没有人组织 meetup，你可以自己组织起来！要善用 vuemeetups.org 的资源！
- **帮助 meetup 组织者。**在举办活动时，帮助永远不嫌多，所以请帮助当地组织者让每个活动都能够成功举办。

对于如何参与当地的 Vue 社区，如果你有任何问题，请联系 hello@vuemeetups.org 或 @VueMeetups！

开发团队

核心团队

社区伙伴