

# Vue.js 2.0 安装教程

## 兼容性

Vue.js 不支持 IE8 及其以下版本，因为 Vue.js 使用了 IE8 不能实现的 ECMAScript 5 特性。Vue.js 支持所有兼容 ECMAScript 5 的浏览器。

## 更新日志

每个版本的更新日志见 [GitHub](#)。

## 独立版本

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

开发环境不要用最小压缩版，不然就失去了错误提示和警告！

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，22.86kb min+gzip

## CDN

推荐 [unpkg](#)，会保持和 npm 发布的最新的版本一致。可以在 [unpkg.com/vue/](#) 浏览 npm 包资源。

也可以从 [jsdelivr](#) 或 [cdnjs](#) 获取，不过这两个服务版本更新可能略滞后。

## NPM

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

1. # 最新稳定版

2. `$ npm install vue`

## 独立构建 vs 运行时构建

有两种构建方式，独立构建和运行时构建。

- 

独立构建包括编译和支持 `template` 选项。

- 

- 

运行时构建不包括模板编译，不支持 `template` 选项。运行时构建，可以用 `render` 选项，但它只在单文件组件中起作用，因为单文件组件的模板是在构建时预编译到 `render` 函数中，运行时构建只有独立构建大小的 30%，只有 16 Kb min+gzip 大小。

- 

默认 NPM 包导出的是 构建。为了使用独立构建，在 webpack 配置中添加下面的别名：

```
1.     resolve: {
2.     alias: {
3.       'vue$': 'vue/dist/vue.js'
4.     }
5.   }
```

对于 Browserify，可以用 `aliasify`

不要用 `import Vue from 'vue/dist/vue.js'` - 用一些工具或第三方库引入 Vue，这可能会导致应用程序在同一时间加载运行时和独立构建并造成错误。

## CSP 环境

有些环境，如 Google Chrome Apps，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。独立的构建取决于该功能编译模板，所以无法使用这些环境。

另一方面，运行时构建的是完全兼容 CSP 的。当通过 **Webpack + vue-loader** 或者 **Browserify + vueify** 构建时，在 CSP 环境中模板将被完美预编译到 **render** 函数中。

## 命令行工具

Vue.js 提供一个**官方命令行工具**，可用于快速搭建大型单页应用。该工具提供开箱即用的构建工具配置，带来现代化的前端开发流程。只需一分钟即可启动带热重载、保存时静态检查以及可用于生产环境的构建配置的项目：

1. # 全局安装 vue-cli
2. \$ npm install --global vue-cli
3. # 创建一个基于 webpack 模板的新项目
4. \$ vue init webpack my-project
5. # 安装依赖，走你
6. \$ cd my-project
7. \$ npm install
8. \$ npm run dev

## 开发版本

：在发布后构建的文件在 Github 仓库的 **/dist** 文件夹。为了使用 Github 上 Vue 最新的资源，你得自己构建。

1. git clone https://github.com/vuejs/vue.git node\_modules/vue
2. cd node\_modules/vue
3. npm install
4. npm run build

## Bower

1. # 最新稳定版本
2. \$ bower install vue

## AMD 模块加载器

独立下载版本或通过 Bower 安装 的版本已用 UMD 包装, 因此它们可以直接用作 AMD 模块。

## Vue 介绍

## Vue.js 是什么

Vue.js (读音 /vjuː/, 类似于 ) 是一套构建用户界面的。与其他重量级框架不同的是, Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层, 并且非常容易学习, 非常容易与其它库或已有项目整合。另一方面, Vue 完全有能力驱动采用单文件组件和 [Vue 生态系统支持的库](#) 开发的复杂单页应用。

Vue.js 的目标是通过尽可能简单的 API 实现和。

如果你是有经验的前端开发者, 想知道 Vue.js 与其它库/框架的区别, 查看对比其它框架。

如果你是初学者请看这里 [Vue2.0 新手入门 — 从环境搭建到发布](#)

## 起步

尝试 Vue.js 最简单的方法是使用 [JSFiddle Hello World 例子](#)。请在浏览器新标签页中打开它, 跟着我们查看一些基础示例。如果你喜欢用包管理器下载/安装, 查看安装教程。

## 声明式渲染

Vue.js 的核心是一个允许你采用简洁的模板语法来声明式的将数据渲染进 DOM 的系统:

```
1. <div id="app">
2.   {{ message }}
3. </div>
1. var app = new Vue({
2.   el: '#app',
3.   data: {
4.     message: 'Hello Vue!'
5.   }
6. })
```

[效果预览](#) »

我们已经生成了我们的第一个 Vue 应用! 看起来这跟单单渲染一个字符串模板非常类似, 但是 Vue.js 在背后做了大量工作。现在数据和 DOM 已经被绑定在一起, 所有的元素都是。

我们如何知道？打开你的浏览器的控制台，并修改 `app.message`，你将看到上例相应地更新。

除了绑定插入的文本内容，我们还可以采用这样的方式绑定 DOM 元素属性：

```
1. <div id="app-2">
2.   <span v-bind:title="message">
3.     Hover your mouse over me for a few seconds to see my dynamically bound title!
4.   </span>
5. </div>
1. var app2 = new Vue({
2.   el: '#app-2',
3.   data: {
4.     message: 'You loaded this page on ' + new Date()
5.   }
6. })
```

效果预览 »

这里我们遇到点新东西。你看到的 `v-bind` 属性被称为。指令带有前缀 `v-`，以表示它们是 Vue.js 提供的特殊属性。可能你已经猜到了，它们会在渲染过的 DOM 上应用特殊的响应式行为。这个指令的简单含义是说：将这个元素节点的 `title` 属性和 Vue 实例的 `message` 属性绑定到一起。

你再次打开浏览器的控制台输入 `app2.message = 'some new message'`，你就会再一次看到这个绑定了 `title` 属性的 HTML 已经进行了更新。

## 条件与循环

控制切换一个元素的显示也相当简单：

```
1. <div id="app-3">
2.   <p v-if="seen">Now you see me</p>
3. </div>
1. var app3 = new Vue({
2.   el: '#app-3',
```

```
3.         data: {
4.             seen: true
5.         }
6.     })
```

[效果预览 »](#)

继续在控制台设置 `app3.seen = false`，你会发现 “Now you see me” 消失了。

这个例子演示了我们不仅可以绑定 DOM 文本到数据，也可以绑定 DOM 到数据。而且，Vue.js 也提供一个强大的过渡效果系统，可以在 Vue 插入/删除元素时自动应用过渡效果。

也有一些其它指令，每个都有特殊的功能。例如，`v-for` 指令可以绑定数据到数据来渲染一个列表：

```
1.     <div id="app-4">
2.         <ol>
3.             <li v-for="todo in todos">
4.                 {{ todo.text }}
5.             </li>
6.         </ol>
7.     </div>
1.     var app4 = new Vue({
2.         el: '#app-4',
3.         data: {
4.             todos: [
5.                 { text: 'Learn JavaScript' },
6.                 { text: 'Learn Vue' },
7.                 { text: 'Build something awesome' }
8.             ]
9.         }
10.    })
```

[效果预览 »](#)

在控制台里，输入 `app4.todos.push({ text: 'New item' })`。你会发现列表中多了一栏新内容。

## 处理用户输入

为了让用户和你的应用进行互动，我们可以用 `v-on` 指令绑定一个监听事件用于调用我们 Vue 实例中定义的方法：

```
1.     <div id="app-5">
2.         <p>{{ message }}</p>
3.         <button v-on:click="reverseMessage">Reverse Message</button>
4.     </div>
1.     var app5 = new Vue({
2.         el: '#app-5',
3.         data: {
4.             message: 'Hello Vue.js!'
5.         },
6.         methods: {
7.             reverseMessage: function () {
8.                 this.message = this.message.split('').reverse().join('')
9.             }
10.        }
11.    })
```

效果预览 »

在 `reverseMessage` 方法中，我们在没有接触 DOM 的情况下更新了应用的状态 - 所有的 DOM 操作都由 Vue 来处理，你写的代码只需要关注基本逻辑。

Vue 也提供了 `v-model` 指令，它使得在表单输入和应用状态中做双向数据绑定变得非常轻巧。

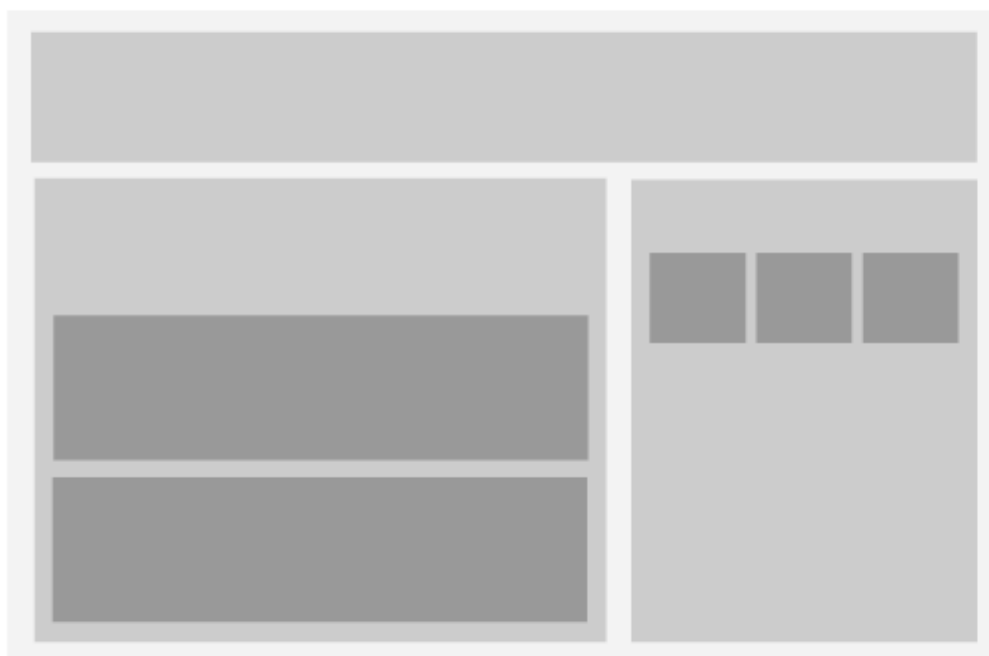
```
1.     <div id="app-6">
2.         <p>{{ message }}</p>
3.         <input v-model="message">
```

```
4.     </div>
1.     var app6 = new Vue({
2.       el: '#app-6',
3.       data: {
4.         message: 'Hello Vue!'
5.       }
6.     })
```

[效果预览 »](#)

## 用组件构建（应用）

组件系统是 Vue.js 另一个重要概念，因为它提供了一种抽象，让我们可以用独立可复用的小组件来构建大型应用。如果我们考虑到这点，几乎任意类型的应用的界面都可以抽象为一个组件树：





在 Vue 里，一个组件实质上是一个拥有预定义选项的一个 Vue 实例：

```
1. // Define a new component called todo-item
2. Vue.component('todo-item', {
3.   template: '<li>This is a todo</li>'
4. })
```

现在你可以另一个组件模板中写入它：

```
1. <ul>
2.   <!--
3.     Create an instance of the todo-item component
4.   -->
5.   <todo-item></todo-item>
6. </ul>
```

但是这样会为每个 todo 渲染同样的文本，这看起来并不是很酷。我们应该将数据从父作用域传到子组件。让我们来修改一下组件的定义，使得它能够接受一个 prop 字段：

```
1. Vue.component('todo-item', {
2.   // The todo-item component now accepts a
3.   // "prop", which is like a custom attribute.
4.   // This prop is called todo.
5.   props: ['todo'],
6.   template: '<li>{{ todo.text }}</li>'
7. })
```

现在，我们可以使用 `v-bind` 指令将 todo 传到每一个重复的组件中：

```
1. <div id="app-7">
2.   <ol>
3.     <!--
4.       Now we provide each todo-item with the todo object
5.       it's representing, so that its content can be dynamic
6.     -->
7.     <todo-item v-for="todo in todos" v-bind:todo="todo"></todo-item>
8.   </ol>
```

```

9.     </div>
1.     Vue.component('todo-item', {
2.         props: ['todo'],
3.         template: '<li>{{ todo.text }}</li>'
4.     })
5.     var app7 = new Vue({
6.         el: '#app-7',
7.         data: {
8.             todos: [
9.                 { text: 'Learn JavaScript' },
10.                { text: 'Learn Vue' },
11.                { text: 'Build something awesome' }
12.            ]
13.        }
14.    })

```

效果预览 »

这只是一个假设的例子，但是我们已经将应用分割成了两个更小的单元，子元素通过 *props* 接口实现了与父亲元素很好的解耦。我们现在可以在不影响到父应用的基础上，进一步为我们的 *todo* 组件改进更多复杂的模板和逻辑。

在一个大型应用中，为了使得开发过程可控，有必要将应用整体分割成一个个的组件。在后面的教程中我们将详述组件，不过这里有一个（假想）的例子，看看使用了组件的应用模板是什么样的：

```

1.     <div id="app">
2.         <app-nav></app-nav>
3.         <app-view>
4.             <app-sidebar></app-sidebar>
5.             <app-content></app-content>
6.         </app-view>
7.     </div>

```

## 与自定义元素的关系

你可能已经注意到 Vue.js 组件非常类似于——它是 **Web 组件规范** 的一部分。实际上 Vue.js 的组件语法参考了该规范。例如 Vue 组件实现了 **Slot API** 与 **is** 特性。但是，有几个关键的不同：

1.

Web 组件规范仍然远未完成，并且没有浏览器实现。相比之下，Vue.js 组件不需要任何补丁，并且在所有支持的浏览器（IE9 及更高版本）之下表现一致。必要时，Vue.js 组件也可以放在原生自定义元素之内。

2.

3.

Vue.js 组件提供了原生自定义元素所不具备的一些重要功能，比如组件间的数据流，自定义事件系统，以及动态的、带特效的组件替换。

4.

## 准备好探索更广阔的世界了？

我们刚才简单介绍了 Vue.js 核心的一些最基本的特征 - 本指南的其余部分将用更详尽的篇幅去描述其他的一些高级特性，所以一定要阅读完所有的内容哦！

### Vue 实例

## 构造器

每个 Vue.js 应用都是通过构造函数 **Vue** 创建一个 启动的：

```
1.   var vm = new Vue({  
2.     // 选项  
3.   })
```

虽然没有完全遵循 **MVVM 模式**，Vue 的设计无疑受到了它的启发。因此在文档中经常会使用 **vm** 这个变量名表示 Vue 实例。

在实例化 `Vue` 时，需要传入一个，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 API 文档中查看。

可以扩展 `Vue` 构造器，从而用预定义选项创建可复用的：

```
1.     var MyComponent = Vue.extend({
2.       // 扩展选项
3.     })
4.     // 所有的 `MyComponent` 实例都将以预定义的扩展选项被创建
5.     var myComponentInstance = new MyComponent()
```

尽管可以命令式地创建扩展实例，不过在多数情况下建议将组件构造器注册为一个自定义元素，然后声明式地用在模板中。我们将在后面详细说明组件系统。现在你只需知道所有的 `Vue.js` 组件其实都是被扩展的 `Vue` 实例。

## 属性与方法

每个 `Vue` 实例都会其 `data` 对象里所有的属性：

```
1.     var data = { a: 1 }
2.     var vm = new Vue({
3.       data: data
4.     })
5.     vm.a === data.a // -> true
6.     // 设置属性也会影响到原始数据
7.     vm.a = 2
8.     data.a // -> 2
9.     // ... 反之亦然
10.    data.a = 3
11.    vm.a // -> 3
```

注意只有这些被代理的属性是。如果在实例创建之后添加新的属性到实例上，它不会触发视图更新。我们将在后面详细讨论响应系统。

除了 `data` 属性，`Vue` 实例暴露了一些有用的实例属性与方法。这些属性与方法都有前缀

`$`，以便与代理的 `data` 属性区分。例如：

```
1.     var data = { a: 1 }
```

```
2.     var vm = new Vue({
3.       el: '#example',
4.       data: data
5.     })
6.     vm.$data === data // -> true
7.     vm.$el === document.getElementById('example') // -> true
8.     // $watch 是一个实例方法
9.     vm.$watch('a', function (newVal, oldVal) {
10.      // 这个回调将在 `vm.a` 改变后调用
11.    })
```

注意，不要在实例属性或者回调函数中（如 `vm.$watch('a', newVal => this.myMethod())`）使用箭头函数。因为箭头函数绑定父上下文，所以 `this` 不会像预想的一样是 Vue 实例，而是 `this.myMethod` 未被定义。

实例属性和方法的完整列表中查阅 API 参考。

## 实例生命周期

```
1.     var vm = new Vue({
2.       data: {
3.         a: 1
4.       },
5.       created: function () {
6.         // `this` 指向 vm 实例
7.         console.log('a is: ' + this.a)
8.       }
9.     })
10.    // -> "a is: 1"
```

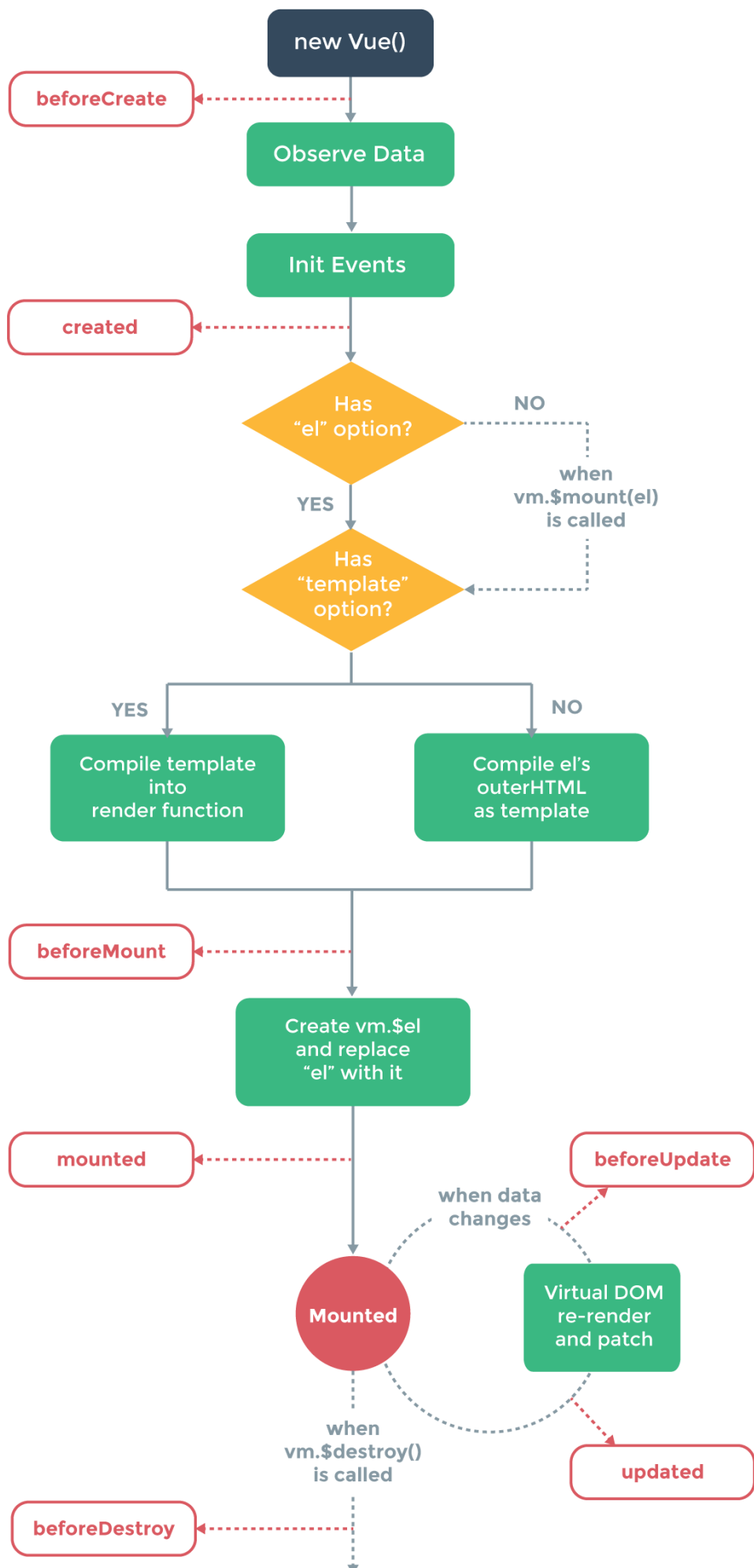
There are also other hooks which will be called at different stages of the instance's lifecycle, for example `mounted`, `updated`, and `destroyed`. All lifecycle hooks are called with their `this` context pointing to the Vue instance invoking it. You may have been wondering where the

concept of “controllers” lives in the Vue world and the answer is: there are no controllers. Your custom logic for a component would be split among these lifecycle hooks.

也有一些其它的钩子，在实例生命周期的不同阶段调用，如 `mounted`、`updated`、`destroyed`。钩子的 `this` 指向调用它的 Vue 实例。一些用户可能会问 Vue.js 是否有“控制器”的概念？答案是，没有。组件的自定义逻辑可以分布在这些钩子中。

## 生命周期图示

下图说明了实例的生命周期。你不需要立马弄明白所有的东西，不过以后它会有帮助。



## Lifecycle hooks

生命周期钩子应该算 vue 这次升级中 broken changes 最多的一部分了，对照 1.0 的[文档](#)和[release note](#)，作了下面这张表

vue 1.0+	vue 2.0	Description
init	beforeCreate	组件实例刚被创建，组件属性计算之前，如 data 属性等
created	created	组件实例创建完成，属性已绑定，但 DOM 还未生成，\$el 属性还不存在
beforeCompile	beforeMount	模板编译/挂载之前
compiled	mounted	模板编译/挂载之后
ready	mounted	模板编译/挂载之后（不保证组件已在 document 中）
-	beforeUpdate	组件更新之前
-	updated	组件更新之后
-	activated	for keep-alive，组件被激活时调用
-	deactivated	for keep-alive，组件被移除时调用
attached	-	不用了还说啥哪...
detached	-	那就不说了吧...
beforeDestory	beforeDestory	组件销毁前调用
destoryed	destoryed	组件销毁后调用

手册网  
shouce.ren

## 模板语法

Vue.js 使用了基于 HTML 的模版语法，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，在应用状态改变时，Vue 能够智能地计算出重新渲染组件的最小代价并应用到 DOM 操作上。

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所有的数据绑定：

1. `<span v-once>This will never change: {{ msg }}</span>`

## 纯 HTML

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

1. `<div v-html="rawHtml"></div>`

被插入的内容都会被当做 HTML —— 数据绑定会被忽略。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。组件更适合担任 UI 重用与复合的基本单元。

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。请只对可信内容使用 HTML 插值，对用户提供的插值。

## 属性

Mustache 不能在 HTML 属性中使用，应使用 `v-bind` 指令：

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

这对布尔值的属性也有效 —— 如果条件被求值为 `false` 的话该属性会被移除：

1. `<button v-bind:disabled="someDynamicCondition">Button</button>`

## 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。

1. `{{ number + 1 }}`
2. `{{ ok ? 'YES' : 'NO' }}`
3. `{{ message.split('').reverse().join('') }}`
4. `<div v-bind:id="list-1 + 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，所以下面的例子都生效。

1. `<!-- 这是语句，不是表达式 -->`
2. `{{ var a = 1 }}`
3. `<!-- 流控制也不会生效，请使用三元表达式 -->`
4. `{{ 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`。你不在应该在模板表达式中试图访问用户定义的全局变量。

## 过滤器

Vue.js 允许你自定义过滤器，被用作一些常见的文本格式化。过滤器应该被添加在 的尾部，由“管道符”指示：

1. `{{ message | capitalize }}`

Vue 2.x 中，过滤器只能在 `mustache` 绑定中使用。为了在指令绑定中实现同样的行为，你应该使用计算属性。

过滤器函数总接受表达式的值作为第一个参数。

1. `new Vue({`
2. `// ...`
3. `filters: {`
4.  `capitalize: function (value) {`
5.  `if (!value) return ''`
6.  `value = value.toString()`
7.  `return value.charAt(0).toUpperCase() + value.slice(1)`
8.  `}`
9. `}`
10. `})`

过滤器可以串联：

1. `{{ message | filterA | filterB }}`

过滤器是 JavaScript 函数，因此可以接受参数：

1. `{{ message | filterA('arg1', arg2) }}`

这里，字符串 `'arg1'` 将传给过滤器作为第二个参数，`arg2` 表达式的值将被求值然后传给过滤器作为第三个参数。

## 指令

指令（Directives）是带有 `v-` 前缀的特殊属性。指令属性的值预期是（除了 `v-for`，之后再讨论）。指令的职责就是当其表达式的值改变时相应地将某些行为应用到 DOM 上。让我们回顾一下在介绍里的例子：

1. `<p v-if="seen">Now you see me</p>`

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

## 参数

一些指令能接受一个“参数”，在指令后以冒号指明。例如，`v-bind` 指令被用来响应地更新 HTML 属性：

1. `<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` 属性与表达式 `url` 的值绑定。

另一个例子是 `v-on` 指令，它用于监听 DOM 事件：

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

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，用于指出一个指定应该以特殊方式

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()`：

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

之后当我们更深入地了解 `v-on` 与 `v-model` 时，会看到更多修饰符的使用。

## 缩写

`v-` 前缀在模板中是作为一个标示 Vue 特殊属性的明显标识。当你使用 Vue.js 为现有的标记添加动态行为时，它会很有用，但对于一些经常使用的指令来说有点繁琐。同时，当搭建

Vue.js 管理所有模板的 SPA 时，`v-` 前缀也变得没那么重要了。因此，Vue.js 为两个最为常用的指令提供了特别的缩写：

## `v-bind` 缩写

1. `<!-- 完整语法 -->`
2. `<a v-bind:href="url"></a>`
3. `<!-- 缩写 -->`
4. `<a :href="url"></a>`

## `v-on` 缩写

1. `<!-- 完整语法 -->`
2. `<a v-on:click="doSomething"></a>`
3. `<!-- 缩写 -->`
4. `<a @click="doSomething"></a>`

它们看起来可能与普通的 HTML 略有不同，但 `:` 与 `@` 对于属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。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。

## 列表渲染

### `v-for`

我们用 `v-for` 指令根据一组数组的选项列表进行渲染。`v-for` 指令需要以 `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。

## 基本用法

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

```
4.     </li>
5.     </ul>
1.     var example1 = new Vue({
2.       el: '#example-1',
3.       data: {
4.         items: [
5.           {message: 'foo' },
6.           {message: 'Bar' }
7.         ]
8.       }
9.     })
```

效果预览 »

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。

```
1.     <ul id="example-2">
2.       <li v-for="(item, index) in items">
3.         {{ parentMessage }} - {{ index }} - {{ item.message }}
4.       </li>
5.     </ul>
1.     var example2 = new Vue({
2.       el: '#example-2',
3.       data: {
4.         parentMessage: 'Parent',
5.         items: [
6.           { message: 'Foo' },
7.           { message: 'Bar' }
8.         ]
9.       }
10.    })
```

效果预览 »

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：

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

## Template v-for

如同 `v-if` 模板，你也可以用带有 `v-for` 的 `<template>` 标签来渲染多个元素块。例如：

```
1. <ul>
2.   <template v-for="item in items">
3.     <li>{{ item.msg }}</li>
4.     <li class="divider"></li>
5.   </template>
6. </ul>
```

## 对象迭代 v-for

你也可以用 `v-for` 通过一个对象的属性来迭代。

```
1. <ul id="repeat-object" class="demo">
2.   <li v-for="value in object">
3.     {{ value }}
4.   </li>
5. </ul>

1. new Vue({
2.   el: '#repeat-object',
3.   data: {
4.     object: {
5.       FirstName: 'John',
6.       LastName: 'Doe',
7.       Age: 30
8.     }
9.   }
10. })
```

[效果预览 »](#)

你也可以提供第二个的参数为键名：

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

第三个参数为索引：

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

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。

## 整数迭代 v-for

`v-for` 也可以取整数。在这种情况下，它将重复多次模板。

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

[效果预览 »](#)

## 组件 和 v-for

了解组件相关知识，查看 [组件](#)。

在自定义组件里，你可以像任何普通元素一样用 `v-for`。

1. `<my-component v-for="item in items"></my-component>`

然而他不能自动传递数据到组件里，因为组件有自己独立的作用域。为了传递迭代数据到组件里，我们要用 `props`：

1. `<my-component`
2. `v-for="(item, index) in items"`
3. `v-bind:item="item"`
4. `v-bind:index="index">`

5. `</my-component>`

不自动注入 `item` 到组件里的原因是，因为这使得组件会紧密耦合到 `v-for` 如何运作。在一些情况下，明确数据的来源可以使组件可重用。

下面是一个简单的 todo list 完整的例子：

```
1. <div id="todo-list-example">
2.   <input
3.     v-model="newTodoText"
4.     v-on:keyup.enter="addNewTodo"
5.     placeholder="Add a todo"
6.   >
7.   <ul>
8.     <li
9.       is="todo-item"
10.      v-for="(todo, index) in todos"
11.      v-bind:title="todo"
12.      v-on:remove="todos.splice(index, 1)"
13.    ></li>
14.  </ul>
15. </div>

1. Vue.component('todo-item', {
2.   template: `
3.     <li>
4.       {{ title }}
5.       <button v-on:click="$emit('remove')">X</button>
6.     </li>
7.   `,
8.   props: ['title']
9. })
10. new Vue({
11.   el: '#todo-list-example',
```



```
12.     data: {
13.       newTodoText: "",
14.       todos: [
15.         'Do the dishes',
16.         'Take out the trash',
17.         'Mow the lawn'
18.       ]
19.     },
20.     methods: {
21.       addNewTodo: function () {
22.         this.todos.push(this.newTodoText)
23.         this.newTodoText = ""
24.       }
25.     }
26.   })
```

效果预览 »

## 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，而不是移动 DOM 元素来匹配数据项的顺序，Vue 将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"`。

这个默认的模式是有效的，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）：

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

3. `</div>`

建议尽可能使用 `v-for` 来提供 `key`，除非迭代 DOM 内容足够简单，或者你是故意要依赖于默认行为来获得性能提升。

因为它是 Vue 识别节点的一个通用机制，`key` 并不特别与 `v-for` 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` 数组调用突变方法：

```
example1.items.push({ message: 'Baz' })。
```

### 重塑数组

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()`、`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：

1. `example1.items = example1.items.filter(function (item) {`

2. `return item.message.match(/Foo/)`
3. `}}`

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 实现了一些智能启发式方法来最大化 DOM 元素重用，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. 当你直接设置一个项的索引时，例如：`vm.items[indexOfItem] = newValue`
2. 当你修改数组的长度时，例如：`vm.items.length = newLength`

为了避免第一种情况，以下两种方式将达到像 `vm.items[indexOfItem] = newValue` 的效果， 同时也将触发状态更新：

1. `// Vue.set`
2. `Vue.set(example1.items, indexOfItem, newValue)`
1. `// Array.prototype.splice``
2. `example1.items.splice(indexOfItem, 1, newValue)`

避免第二种情况，使用 `splice`：

1. `example1.items.splice(newLength)`

## 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：

1. `<li v-for="n in evenNumbers">{{ n }}</li>`
1. `data: {`
2. `numbers: [ 1, 2, 3, 4, 5 ]`
3. `},`
4. `computed: {`
5. `evenNumbers: function () {`

```
6.     return this.numbers.filter(function (number) {
7.     return number % 2 === 0
8.     })
9.     }
10.    }
```

或者，您也可以使用在计算属性是不可行的 `method` 方法 (例如，在嵌套 `v-for` 循环中)：

```
1.     <li v-for="n in even(numbers)">{{ n }}</li>
1.     data: {
2.     numbers: [ 1, 2, 3, 4, 5 ]
3.     },
4.     methods: {
5.     even: function (numbers) {
6.     return numbers.filter(function (number) {
7.     return number % 2 === 0
8.     })
9.     }
10.    }
```

## 计算属性

---

在模板中绑定表达式是非常便利的，但是它们实际上只用于简单的操作。在模板中放入太多的逻辑会让模板过重且难以维护。例如：

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

在这种情况下，模板不再简单和清晰。在实现反向显示 `message` 之前，你应该确认它。这个问题在你不止一次反向显示 `message` 的时候变得更加糟糕。

这就是为什么任何复杂逻辑，你都应当使用。

## 基础例子

```
1. <div id="example">
2.   <p>Original message: "{{ message }}"</p>
3.   <p>Computed reversed message: "{{ reversedMessage }}"</p>
4. </div>
5.
1. var vm = new Vue({
2.   el: '#example',
3.   data: {
4.     message: 'Hello'
5.   },
6.   computed: {
7.     // a computed getter
8.     reversedMessage: function () {
9.       // `this` points to the vm instance
10.      return this.message.split('').reverse().join('')
11.    }
12.  }
13. })
```

结果：

Original message: "Hello"

Computed reversed message: "olleH"

[效果预览](#) »

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。

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

你可以打开浏览器的控制台，修改 `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，依赖于 `vm.reversedMessage` 的绑定也会更新。而且最妙的是我们是声明式地创建这种依赖关系：计算属性的 `getter` 是干净无副作用的，因此也是易于测试和理解的。

## 计算缓存 vs Methods

你可能已经注意到我们可以通过调用表达式中的 `method` 来达到同样的效果：

```
1. <p>Reversed message: "{{ reverseMessage() }}"</p>
1. // in component
2. methods: {
3.   reverseMessage: function () {
4.     return this.message.split('').reverse().join('')
5.   }
6. }
```

不经过计算属性，我们可以在 `method` 中定义一个相同的函数来替代它。对于最终的结果，两种方式确实是相同的。然而，不同的是。计算属性只有在它的相关依赖发生改变时才会重新取值。这就意味着只要 `message` 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。

这也同样意味着如下计算属性将不会更新，因为 `Date.now()` 不是响应式依赖：

```
1. computed: {
2.   now: function () {
3.     return Date.now()
4.   }
5. }
```

相比而言，每当重新渲染的时候，`method` 调用执行函数。

我们为什么需要缓存？假设我们有一个重要的计算属性，这个计算属性需要一个巨大的数组遍历和做大量的计算。然后我们可能有其他的计算属性依赖于它。如果没有缓存，我们将不可避免的多次执行它的 getter！如果你不希望有缓存，请用 method 替代。

## 计算属性 vs Watched Property

Vue.js 提供了一个方法 `$watch`，它用于观察 Vue 实例上的数据变动。当一些数据需要根据其它数据变化时，`$watch` 很诱人——特别是如果你来自 AngularJS。不过，通常更好的办法是使用计算属性而不是一个命令式的 `$watch` 回调。思考下面例子：

```
1. <div id="demo">{{ fullName }}</div>
2.
3. var vm = new Vue({
4.   el: '#demo',
5.   data: {
6.     firstName: 'Foo',
7.     lastName: 'Bar',
8.     fullName: 'Foo Bar'
9.   },
10.  watch: {
11.    firstName: function (val) {
12.      this.fullName = val + ' ' + this.lastName
13.    },
14.    lastName: function (val) {
15.      this.fullName = this.firstName + ' ' + val
16.    }
17.  })
```

上面代码是命令式的和重复的。跟计算属性对比：

```
1. var vm = new Vue({
2.   el: '#demo',
3.   data: {
4.     firstName: 'Foo',
```

```
5.         lastName: 'Bar'
6.     },
7.     computed: {
8.         fullName: function () {
9.             return this.firstName + ' ' + this.lastName
10.        }
11.    }
12. })
```

这样更好，不是吗？

[效果预览 »](#)

## 计算 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`：

```
1.     // ...
2.     computed: {
3.         fullName: {
4.             // getter
5.             get: function () {
6.                 return this.firstName + ' ' + this.lastName
7.             },
8.             // setter
9.             set: function (newValue) {
10.                 var names = newValue.split(' ')
11.                 this.firstName = names[0]
12.                 this.lastName = names[names.length - 1]
13.             }
14.         }
15.     }
16.     // ...
```



现在在运行 `vm.fullName = 'John Doe'` 时，`setter` 会被调用，`vm.firstName` 和 `vm.lastName` 也会被对应更新。

## 观察 Watchers

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 `watcher`。这是为什么 Vue 提供一个更通用的方法通过 `watch` 选项，来响应数据的变化。当你想要在数据变化响应时，执行异步操作或昂贵操作时，这是很有用的。

例如：

```
1. <div id="watch-example">
2.   <p>
3.     Ask a yes/no question:
4.     <input v-model="question">
5.   </p>
6.   <p>{{ answer }}</p>
7. </div>
8.
9. <!-- Since there is already a rich ecosystem of ajax libraries -->
10. <!-- and collections of general-purpose utility methods, Vue core -->
11. <!-- is able to remain small by not reinventing them. This also -->
12. <!-- gives you the freedom to just use what you're familiar with. -->
13. <script src="https://unpkg.com/axios@0.12.0/dist/axios.min.js"></script>
14. <script src="https://unpkg.com/lodash@4.13.1/lodash.min.js"></script>
15. <script>
16.   var watchExampleVM = new Vue({
17.     el: '#watch-example',
18.     data: {
19.       question: "",
20.       answer: 'I cannot give you an answer until you ask a question!'
21.     },
22.     watch: {
```

```
15.     // 如果 question 发生改变，这个函数就会运行
16. question: function (newQuestion) {
17.     this.answer = 'Waiting for you to stop typing...'
18.     this.getAnswer()
19. }
20. },
21. methods: {
22.     // _.debounce 是一个通过 lodash 限制操作频率的函数。
23.     // 在这个例子中，我们希望限制访问 yesno.wtf/api 的频率
24.     // ajax 请求直到用户输入完毕才会发出
25.     // 学习更多关于 _.debounce function (and its cousin
26.     // _.throttle), 参考: https://lodash.com/docs#debounce
27.     getAnswer: _.debounce(
28.         function () {
29.             var vm = this
30.             if (this.question.indexOf('?') === -1) {
31.                 vm.answer = 'Questions usually contain a question mark. ;-)'
32.                 return
33.             }
34.             vm.answer = 'Thinking...'
35.             axios.get('https://yesno.wtf/api')
36.                 .then(function (response) {
37.                     vm.answer = _.capitalize(response.data.answer)
38.                 })
39.                 .catch(function (error) {
40.                     vm.answer = 'Error! Could not reach the API. ' + error
41.                 })
42.         },
43.         // 这是我们为用户停止输入等待的毫秒数
44.         500
```

```
45.     )
46.     }
47.     })
48.     </script>
```

[效果预览 »](#)

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，并直到我们得到最终结果时，才设置中间状态。这是计算属性无法做到的。

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

## Class 与 Style 绑定

数据绑定一个常见需求是操作元素的 `class` 列表和它的内联样式。因为它们都是属性，我们可以用 `v-bind` 处理它们：只需要计算出表达式最终的字符串。不过，字符串拼接麻烦又易错。因此，在 `v-bind` 用于 `class` 和 `style` 时，Vue.js 专门增强了它。表达式的结果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`。

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

上面的语法表示 `class` `active` 的更新将取决于数据属性 `isActive` 是否为真值。

我们也可以在对象中传入更多属性用来动态切换多个 `class`。此外，`v-bind:class` 指令可以与普通的 `class` 属性共存。如下模板：

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

如下 data:

```
1.      data: {  
2.      isActive: true,  
3.      hasError: false  
4.      }
```

渲染为:

```
1.      <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"`。

你也可以直接绑定数据里的一个对象:

```
1.      <div v-bind:class="classObject"></div>  
  
1.      data: {  
2.      classObject: {  
3.      active: true,  
4.      'text-danger': false  
5.      }  
6.      }
```

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

```
1.      <div v-bind:class="classObject"></div>  
  
1.      data: {  
2.      isActive: true,  
3.      error: null  
4.      },  
5.      computed: {  
6.      classObject: function () {  
7.      return {  
8.      active: this.isActive && !this.error,  
9.      'text-danger': this.error && this.error.type === 'fatal',
```

```
10.     }
11.     }
12.     }
```

## 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：

```
1.     <div v-bind:class="[activeClass, errorClass]">
1.         data: {
2.             activeClass: 'active',
3.             errorClass: 'text-danger'
4.         }
```

渲染为：

```
1.     <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：

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

此例始终添加 `errorClass`，但是只有在 `isActive` 是 `true` 时添加 `activeClass`。

不过，当有多个条件 `class` 时这样写有些繁琐。可以在数组语法中使用对象语法：

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

## 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，其实它是一个 JavaScript 对象。

CSS 属性名可以用驼峰式（camelCase）或短横分隔命名（kebab-case）：

```
1.     <div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
1.         data: {
2.             activeColor: 'red',
3.             fontSize: 30
4.         }
```

直接绑定到一个样式对象通常更好，让模板更清晰：

```
1. <div v-bind:style="styleObject"></div>
1. data: {
2.   styleObject: {
3.     color: 'red',
4.     fontSize: '13px'
5.   }
6. }
```

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一个元素上：

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

## 自动添加前缀

当 `v-bind:style` 使用需要特定前缀的 CSS 属性时，如 `transform`，Vue.js 会自动侦测并添加相应的前缀。

## 表单控件绑定

## 基础用法

你可以用 `v-model` 指令在表单控件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，并特别处理一些极端的例子。

`v-model` 并不关心表单控件初始化所生成的值。因为它会选择 Vue 实例数据来作为具体的值。

## 文本

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

效果预览 »

## 多行文本

1. `<span>Multiline message is:</span>`
2. `<p style="white-space: pre">{{ message }}</p>`
3. `<br>`
4. `<textarea v-model="message" placeholder="add multiple lines"></textarea>`

在文本区域插值( `<textarea></textarea>` ) 并不会生效, 应用 `v-model` 来代替

效果预览 »

## 复选框

单个勾选框, 逻辑值 :

1. `<input type="checkbox" id="checkbox" v-model="checked">`
2. `<label for="checkbox">{{ checked }}</label>`

效果预览 »

多个勾选框, 绑定到同一个数组 :

1. `<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">`
2. `<label for="jack">Jack</label>`
3. `<input type="checkbox" id="john" value="John" v-model="checkedNames">`
4. `<label for="john">John</label>`
5. `<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">`
6. `<label for="mike">Mike</label>`
7. `<br>`
8. `<span>Checked names: {{ checkedNames }}</span>`

1. `new Vue({`
2. `el: '...',`
3. `data: {`

```
4.         checkedNames: []
5.     }
6. })
```

[效果预览 »](#)

## 单选按钮

```
1.     <input type="radio" id="one" value="One" v-model="picked">
2.     <label for="one">One</label>
3.
4.     <input type="radio" id="two" value="Two" v-model="picked">
5.     <label for="two">Two</label>
6.
7.     <br>
8.     <span>Picked: {{ picked }}</span>
```

[效果预览 »](#)

## 选择列表

单选列表:

```
1.     <select v-model="selected">
2.     <option>A</option>
3.     <option>B</option>
4.     <option>C</option>
5.     </select>
6.     <span>Selected: {{ selected }}</span>
```

[效果预览 »](#)

多选列表（绑定到一个数组）：

```
1.     <select v-model="selected" multiple>
2.     <option>A</option>
3.     <option>B</option>
4.     <option>C</option>
5.     </select>
6.     <br>
```



7. `<span>Selected: {{ selected }}</span>`

[效果预览 »](#)

动态选项，用 `v-for` 渲染：

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

```
1. new Vue({
2.   el: '...',
3.   data: {
4.     selected: 'A',
5.     options: [
6.       { text: 'One', value: 'A' },
7.       { text: 'Two', value: 'B' },
8.       { text: 'Three', value: 'C' }
9.     ]
10.   }
11. })
```

[效果预览 »](#)

## 绑定 value

对于单选按钮，勾选框及选择列表选项，`v-model` 绑定的 `value` 通常是静态字符串（对于勾选框是逻辑值）：

```
1. <!-- 当选中时，`picked` 为字符串 "a" -->
2. <input type="radio" v-model="picked" value="a">
3. <!-- `toggle` 为 true 或 false -->
4. <input type="checkbox" v-model="toggle">
```

```
5.      <!-- 当选中时，`selected` 为字符串 "abc" -->
6.      <select v-model="selected">
7.      <option value="abc">ABC</option>
8.      </select>
```

但是有时我们想绑定 `value` 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

```
1.      <input
2.      type="checkbox"
3.      v-model="toggle"
4.      v-bind:true-value="a"
5.      v-bind:false-value="b">
1.      // 当选中时
2.      vm.toggle === vm.a
3.      // 当没有选中时
4.      vm.toggle === vm.b
```

## 单选按钮

```
1.      <input type="radio" v-model="pick" v-bind:value="a">
1.      // 当选中时
2.      vm.pick === vm.a
```

## 选择列表设置

```
1.      <select v-model="selected">
2.      <!-- 内联对象字面量 -->
3.      <option v-bind:value="{ number: 123 }">123</option>
4.      </select>
1.      // 当选中时
2.      typeof vm.selected // -> 'object'
3.      vm.selected.number // -> 123
```

## 修饰符

### .lazy

在默认情况下，`v-model` 在 `input` 事件中同步输入框的值与数据，但你可以添加一个修饰符 `lazy`，从而转变为在 `change` 事件中同步：

1. `<!-- 在 "change" 而不是 "input" 事件中更新 -->`
2. `<input v-model.lazy="msg" >`

### .number

如果想自动将用户的输入值转为 `Number` 类型(如果原值的转换结果为 `NaN` 则返回原值)，可以添加一个修饰符 `number` 给 `v-model` 来处理输入值：

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

这通常很有用，因为在 `type="number"` 时 `HTML` 中输入的值也总是会返回字符串类型。

### .trim

如果要自动过滤用户输入的首尾空格，可以添加 `trim` 修饰符到 `v-model` 上过滤输入：

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

## 事件处理器

## 监听事件

可以用 `v-on` 指令监听 `DOM` 事件来触发一些 `JavaScript` 代码。

示例：

1. `<div id="example-1">`
2. `<button v-on:click="counter += 1">增加 1</button>`
3. `<p>这个按钮被点击了 {{ counter }} 次。</p>`

```
4.     </div>
1.     var example1 = new Vue({
2.       el: '#example-1',
3.       data: {
4.         counter: 0
5.       }
6.     })
```

效果预览 »

## 方法事件处理器

许多事件处理的逻辑都很复杂，所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。

因此 `v-on` 可以接收一个定义的方法来调用。

示例：

```
1.     <div id="example-2">
2.       <!-- `greet` 是在下面定义的方法名 -->
3.       <button v-on:click="greet">Greet</button>
4.     </div>
1.     var example2 = new Vue({
2.       el: '#example-2',
3.       data: {
4.         name: 'Vue.js'
5.       },
6.       // 在 `methods` 对象中定义方法
7.       methods: {
8.         greet: function (event) {
9.           // `this` 在方法里指当前 Vue 实例
10.          alert('Hello ' + this.name + '!')
11.          // `event` 是原生 DOM 事件
12.          alert(event.target.tagName)
```

```
13.     }
14.     }
15.   })
16.   // 也可以用 JavaScript 直接调用方法
17.   example2.greet() // -> 'Hello Vue.js!'
```

效果预览 »

## 内联处理器方法

除了直接绑定到一个方法，也可以用内联 JavaScript 语句：

```
1.     <div id="example-3">
2.       <button v-on:click="say('hi')">Say hi</button>
3.       <button v-on:click="say('what')">Say what</button>
4.     </div>
1.     new Vue({
2.       el: '#example-3',
3.       methods: {
4.         say: function (message) {
5.           alert(message)
6.         }
7.       }
8.     })
```

效果预览 »

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

```
1.     <button v-on:click="warn('Form cannot be submitted yet.', $event)">Submit</button>
1.     // ...
2.     methods: {
3.       warn: function (message, event) {
4.         // 现在我们可以访问原生事件对象
5.         if (event) event.preventDefault()
```

```
6.     alert(message)
7.     }
8.     }
```

## 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 `methods` 中轻松实现这点，但更好的方式是：`methods` 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。通过由点(.)表示的指令后缀来调用修饰符。

- `.stop`
- `.prevent`
- `.capture`
- `.self`

```
1.     <!-- 阻止单击事件冒泡 -->
2.     <a v-on:click.stop="doThis"></a>
3.     <!-- 提交事件不再重载页面 -->
4.     <form v-on:submit.prevent="onSubmit"></form>
5.     <!-- 修饰符可以串联 -->
6.     <a v-on:click.stop.prevent="doThat"></a>
7.     <!-- 只有修饰符 -->
8.     <form v-on:submit.prevent></form>
9.     <!-- 添加事件侦听器时使用时间捕获模式 -->
10.    <div v-on:click.capture="doThis">...</div>
11.    <!-- 只当事件在该元素本身（而不是子元素）触发时触发回调 -->
12.    <div v-on:click.self="doThat">...</div>
```

## 按键修饰符

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

1. `<!-- 只有在 keyCode 是 13 时调用 vm.submit() -->`
2. `<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：

1. `<!-- 同上 -->`
2. `<input v-on:keyup.enter="submit">`
3. `<!-- 缩写语法 -->`
4. `<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：

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

## 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）传统理念。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.

3.

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。

4.

5.

当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何自己清理它们。

6.

关于我们 联系我们 留言板

手册网

## 条件渲染

### v-if

在字符串模板中，如 Handlebars ，我们得像这样写一个条件块：

1. `<!-- Handlebars 模板 -->`
2. `{{#if ok}}`
3. `<h1>Yes</h1>`
4. `{{/if}}`

在 Vue.js ，我们使用 `v-if` 指令实现同样的功能：

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

也可以用 `v-else` 添加一个“else”块：

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

### template v-if



因为 `v-if` 是一个指令，需要将它添加到一个元素上。但是如果我们想切换多个元素呢？此时我们可以把一个 `<template>` 元素当做包装元素，并在上面使用 `v-if`，最终的渲染结果不会包含它。

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

## v-else

可以用 `v-else` 指令给 `v-if` 或 `v-show` 添加一个“else”块：

```
1. <div v-if="Math.random() > 0.5">
2.   Sorry
3. </div>
4. <div v-else>
5.   Not sorry
6. </div>
```

`v-else` 元素必须紧跟在 `v-if` 或 `v-show` 元素的后面——否则它不能被识别。

## v-show

另一个根据条件展示元素的选项是 `v-show` 指令。用法大体上一样：

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

不同的是有 `v-show` 的元素会始终渲染并保持在 DOM 中。`v-show` 是简单的切换元素的 CSS 属性 `display`。

注意 `v-show` 不支持 `<template>` 语法。

## v-if vs. v-show

`v-if` 是真实的条件渲染，因为它会确保条件块在切换当中适当地销毁与重建条件块内的事件监听器和子组件。

`v-if` 也是：如果在初始渲染时条件为假，则什么也不做——在条件第一次变为真时才开始局部编译（编译会被缓存起来）。

相比之下，`v-show` 简单得多——元素始终被编译并保留，只是简单地基于 CSS 切换。

一般来说，`v-if` 有更高的切换消耗而 `v-show` 有更高的初始渲染消耗。因此，如果需要频繁切换使用 `v-show` 较好，如果在运行时条件不大可能改变则使用 `v-if` 较好。

关于我们 联系我们 留言板

手册网

## 组件

## 什么是组件？

组件（Component）是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素，封装可重用的代码。在较高层面上，组件是自定义元素，Vue.js 的编译器为它添加特殊功能。在有些情况下，组件也可以是原生 HTML 元素的形式，以 `is` 特性扩展。

## 使用组件

## 注册

之前说过，我们可以通过以下方式创建一个 Vue 实例：

```
1. new Vue({  
2.   el: '#some-element',  
3.   // 选项  
4. })
```

要注册一个全局组件，你可以使用 `Vue.component(tagName, options)`。例如：

```
1. Vue.component('my-component', {
```

2.        // 选项
3.        })

对于自定义标签名，Vue.js 不强制要求遵循 **W3C 规则**（小写，并且包含一个短杠），尽管遵循这个规则比较好。

组件在注册之后，便可以在父实例的模块中以自定义元素

`<my-component></my-component>` 的形式使用。要确保在初始化根实例 注册了组件：

1.        <div id="example">
2.        <my-component></my-component>
3.        </div>
1.        // 注册
2.        Vue.component('my-component', {
3.        template: '<div>A custom component!</div>'
4.        })
5.        // 创建根实例
6.        new Vue({
7.        el: '#example'
8.        })

渲染为：

1.        <div id="example">
2.        <div>A custom component!</div>
3.        </div>

效果预览 »

## 局部注册

不必在全局注册每个组件。通过使用组件实例选项注册，可以使组件仅在另一个实例/组件的作用域中可用：

1.        var Child = {
2.        template: '<div>A custom component!</div>'
3.        }

```
4.     new Vue({
5.       // ...
6.       components: {
7.         // <my-component> 将只在父模板可用
8.         'my-component': Child
9.       }
10.    })
```

这种封装也适用于其它可注册的 Vue 功能，如指令。

## DOM 模版解析说明

当使用 DOM 作为模版时（例如，将 `el` 选项挂载到一个已存在的元素上），你会受到 HTML 的一些限制，因为 Vue 只有在浏览器解析和标准化 HTML 后才能获取模版内容。尤其像这些元素 `<ul>`，`<ol>`，`<table>`，`<select>` 限制了能被它包裹的元素，`<option>` 只能出现在其它元素内部。

在自定义组件中使用这些受限制的元素时会导致一些问题，例如：

```
1.     <table>
2.       <my-row>...</my-row>
3.     </table>
```

自定义组件 `<my-row>` 被认为是无效的内容，因此在渲染的时候会导致错误。变通的方案是使用特殊的 `is` 属性：

```
1.     <table>
2.       <tr is="my-row"></tr>
3.     </table>
```

- `<script type="text/x-template">`
- JavaScript 内联模版字符串
- `.vue` 组件

因此，有必要的话请使用字符串模版。

## `data` 必须是函数

使用组件时，大多数选项可以被传入到 Vue 构造器中，有一个例外：`data` 必须是函数。

实际上，如果你这么做：

```
1.   Vue.component('my-component', {
2.     template: '<span>{{ message }}</span>',
3.     data: {
4.       message: 'hello'
5.     }
6.   })
```

那么 Vue 会在控制台发出警告，告诉你在组件中 `data` 必须是一个函数。最好理解这种规则的存在意义。

```
1.   <div id="example-2">
2.     <simple-counter></simple-counter>
3.     <simple-counter></simple-counter>
4.     <simple-counter></simple-counter>
5.   </div>
1.   var data = { counter: 0 }
2.   Vue.component('simple-counter', {
3.     template: '<button v-on:click="counter += 1">{{ counter }}</button>',
4.     // data 是一个函数，因此 Vue 不会警告，
5.     // 但是我们为每一个组件返回了同一个对象引用
6.     data: function () {
7.       return data
8.     }
9.   })
10.  new Vue({
11.    el: '#example-2'
12.  })
```

[效果预览 »](#)

由于这三个组件共享了同一个 `data`，因此增加一个 `counter` 会影响所有组件！我们可以通过为每个组件返回新的 `data` 对象来解决这个问题：

```
1.      data: function () {  
2.      return {  
3.      counter: 0  
4.      }  
5.      }
```

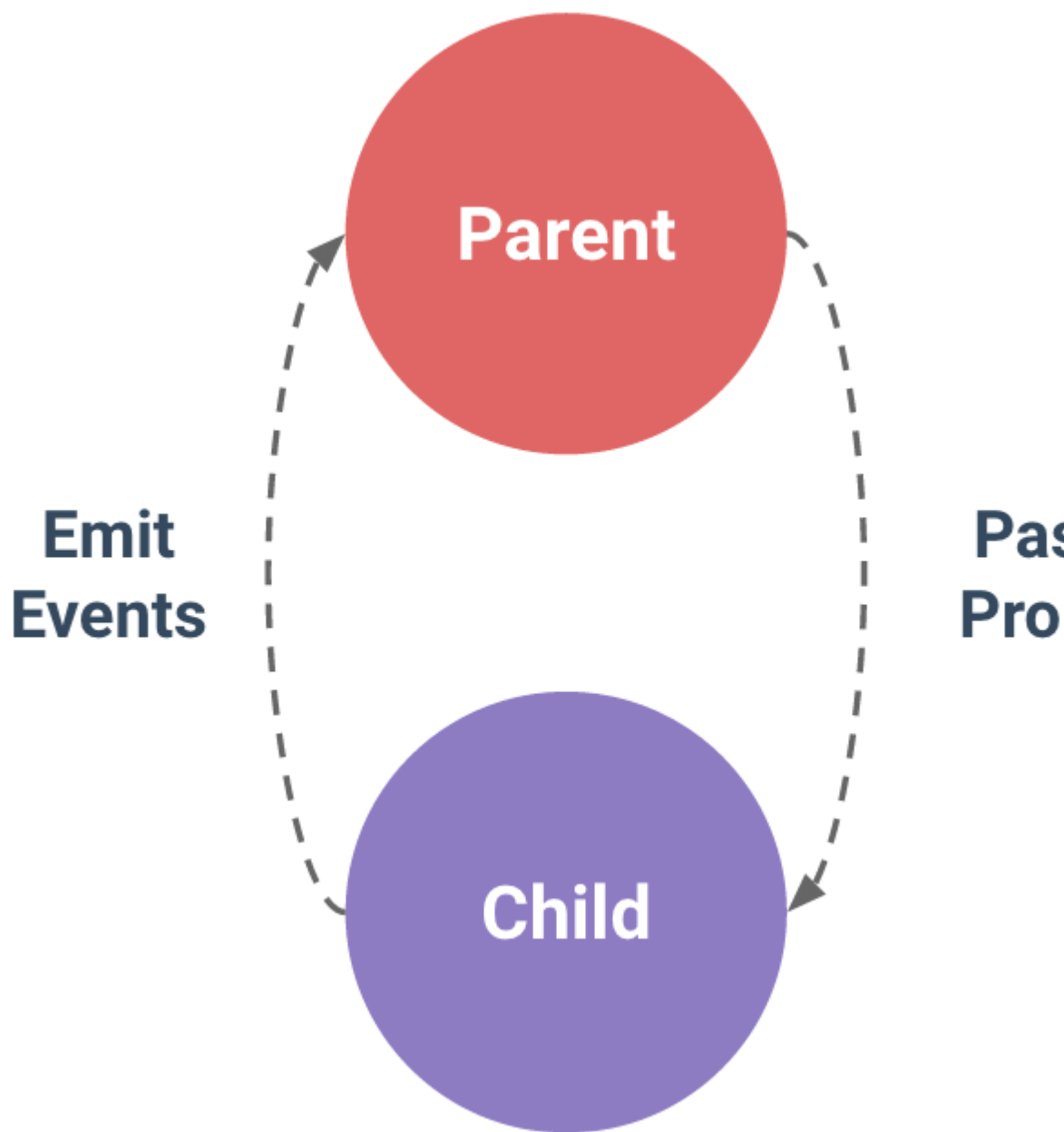
现在每个 `counter` 都有它自己内部的状态了：

[效果预览 »](#)

## 构成组件

组件意味着协同工作, 通常父子组件会是这样的关系: 组件 A 在它的模版中使用了组件 B。它们之间必然需要相互通信: 父组件要给予组件传递数据, 子组件需要将它内部发生的事情告知给父组件。然而, 在一个良好定义的接口中尽可能将父子组件解耦是很重要的。这保证了每个组件可以在相对隔离的环境中书写和理解, 也大幅提高了组件的可维护性和可重用性。

在 `Vue.js` 中, 父子组件的关系可以总结为。父组件通过 向下传递数据给予组件, 子组件通过 给父组件发送消息。看看它们是怎么工作的。



## Props

### 使用 Props 传递数据

组件实例的作用域是。这意味着不能并且不应该在子组件的模板内直接引用父组件的数据。可以使用 `props` 把数据传给子组件。

`prop` 是父组件用来传递数据的一个自定义属性。子组件需要显式地用 `props` 选项 声明 “prop”：

```

1.   Vue.component('child', {
2.     // 声明 props
3.     props: ['message'],
4.     // 就像 data 一样，prop 可以用在模板内
5.     // 同样也可以在 vm 实例中像 “this.message” 这样使用
6.     template: '<span>{{ message }}</span>'
7.   })

```

然后向它传入一个普通字符串：

```
1.   <child message="hello!"></child>
```

效果预览 »

## camelCase vs. kebab-case

HTML 特性不区分大小写。当使用非字符串模版时，名字形式为 camelCase 的 prop 用作特性时，需要转为 kebab-case（短横线隔开）：

```

1.   Vue.component('child', {
2.     // camelCase in JavaScript
3.     props: ['myMessage'],
4.     template: '<span>{{ myMessage }}</span>'
5.   })
1.   <!-- kebab-case in HTML -->
2.   <child my-message="hello!"></child>

```

再次说明，如果你使用字符串模版，不用在意这些限制。

## 动态 Props

类似于用 `v-bind` 绑定 HTML 特性到一个表达式，也可以用 `v-bind` 绑定动态 props 到父组件的数据。每当父组件的数据变化时，也会传导给子组件：

```

1.   <div>
2.     <input v-model="parentMsg">
3.     <br>
4.     <child v-bind:my-message="parentMsg"></child>
5.   </div>

```



使用 `v-bind` 的缩写语法通常更简单：

1. `<child :my-message="parentMsg"></child>`

效果预览 »

## 字面量语法 vs 动态语法

初学者常犯的一个错误是使用字面量语法传递数值：

1. `<!-- 传递了一个字符串 "1" -->`
2. `<comp some-prop="1"></comp>`

因为它是一个字面 `prop`，它的值以字符串 `"1"` 而不是以实际的数字传下去。如果想传递一个实际的 JavaScript 数字，需要使用 `v-bind`，从而让它的值被当作 JavaScript 表达式计算：

1. `<!-- 传递实际的数字 -->`
2. `<comp v-bind:some-prop="1"></comp>`

## 单向数据流

`prop` 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来。这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。

另外，每次父组件更新时，子组件的所有 `prop` 都会更新为最新值。这意味着你在子组件内部改变 `prop`。如果你这么做了，Vue 会在控制台给出警告。

通常有两种改变 `prop` 的情况：

- 1.

`prop` 作为初始值传入，子组件之后只是将它的初始值作为本地数据的初始值使用；

- 2.

- 3.

`prop` 作为需要被转变的原始值传入。

- 4.

更确切的说这两种情况是：

- 1.

定义一个局部 `data` 属性，并将 `prop` 的初始值作为局部数据的初始值。

2.

3.

定义一个 `computed` 属性，此属性从 `prop` 的值计算得出。

4.

注意在 JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 `prop` 是一个对象或数组，在子组件内部改变它父组件的状态。

## Prop 验证

组件可以为 `props` 指定验证要求。如果未指定验证要求，Vue 会发出警告。当组件给其他人使用时这很有用。

`prop` 是一个对象而不是字符串数组时，它包含验证要求：

```
1.   Vue.component('example', {
2.     props: {
3.       // 基础类型检测（`null` 意思是任何类型都可以）
4.       propA: Number,
5.       // 多种类型
6.       propB: [String, Number],
7.       // 必传且是字符串
8.       propC: {
9.         type: String,
10.        required: true
11.      },
12.      // 数字，有默认值
13.      propD: {
14.        type: Number,
15.        default: 100
16.      },
17.      // 数组 / 对象的默认值应当由一个工厂函数返回
18.      propE: {
```

```
19.     type: Object,
20.     default: function () {
21.       return { message: 'hello' }
22.     }
23.   },
24.   // 自定义验证函数
25.   propF: {
26.     validator: function (value) {
27.       return value > 10
28.     }
29.   }
30. }
31. })
```

`type` 可以是下面原生构造器：

- String
- Number
- Boolean
- Function
- Object
- Array

`type` 也可以是一个自定义构造器，使用 `instanceof` 检测。

当 `prop` 验证失败了，Vue 将拒绝在子组件上设置此值，如果使用的是开发版本会抛出一条警告。

## 自定义事件

我们知道，父组件是使用 `props` 传递数据给子组件，但如果子组件要把数据传递回去，应该怎样做？那就是自定义事件！

## 使用 `v-on` 绑定自定义事件

每个 Vue 实例都实现了事件接口 (Events interface)，即：

- 使用 `$on(eventName)` 监听事件
- 使用 `$emit(eventName)` 触发事件

Note that Vue's event system is separate from the browser's `EventTarget API`. Though they work similarly, `$on` and `$emit` are aliases for `addEventListener` and `dispatchEvent`.

另外，父组件可以在使用子组件的地方直接用 `v-on` 来监听子组件触发的事件。

下面是一个例子：

```
1. <div id="counter-event-example">
2.   <p>{{ total }}</p>
3.   <button-counter v-on:increment="incrementTotal"></button-counter>
4.   <button-counter v-on:increment="incrementTotal"></button-counter>
5. </div>
1. Vue.component('button-counter', {
2.   template: '<button v-on:click="increment">{{ counter }}</button>',
3.   data: function () {
4.     return {
5.       counter: 0
6.     }
7.   },
8.   methods: {
9.     increment: function () {
10.      this.counter += 1
11.      this.$emit('increment')
12.    }
13.   },
14. })
15. new Vue({
16.   el: '#counter-event-example',
17.   data: {
```

```

18.         total: 0
19.     },
20.     methods: {
21.         incrementTotal: function () {
22.             this.total += 1
23.         }
24.     }
25. })

```

在本例中，子组件已经和它外部完全解耦了。它所做的只是触发一个父组件关心的内部事件。

[效果预览 »](#)

## 给组件绑定原生事件

有时候，你可能想在某个组件的根元素上监听一个原生事件。可以使用 `.native` 修饰

`v-on`。例如：

```

1. <my-component v-on:click.native="doTheThing"></my-component>

```

## 使用自定义事件的表单输入组件

自定义事件也可以用来创建自定义的表单输入组件，使用 `v-model` 来进行数据双向绑定。

牢记，表单控件进行数据绑定时的语法：

```

1. <input v-model="something">

```

仅仅是一个语法糖：

```

1. <input v-bind:value="something" v-on:input="something = $event.target.value">

```

所以在组件中使用时，它相当于下面的简写：

```

1. <input v-bind:value="something" v-on:input="something = arguments[0]">

```

所以要让组件的 `v-model` 生效，它必须：

- 接受一个 `value` 属性
- 在有新的 `value` 时触发 `input` 事件

实战看看：

```
1.      <div id="v-model-example">
2.      <p>{{ message }}</p>
3.      <my-input
4.      label="Message"
5.      v-model="message"
6.      ></my-input>
7.      </div>
1.      Vue.component('my-input', {
2.      template: '\
3.      <div class="form-group">\
4.      <label v-bind:for="randomId">{{ label }}</label>\
5.      <input v-bind:id="randomId" v-bind:value="value" v-on:input="onInput">\
6.      </div>\
7.      ',
8.      props: ['value', 'label'],
9.      data: function () {
10.     return {
11.     randomId: 'input-' + Math.random()
12.     }
13.     },
14.     methods: {
15.     onInput: function (event) {
16.     this.$emit('input', event.target.value)
17.     }
18.     },
19.     })
20.     new Vue({
21.     el: '#v-model-example',
22.     data: {
```

```
23.     message: 'hello'
24.   }
25. })
```

这个接口不仅仅可以用来连接组件内部的表单输入，也很容易集成你自己创造的输入类型。想象一下：

```
1.     <voice-recognizer v-model="question"></voice-recognizer>
2.     <webcam-gesture-reader v-model="gesture"></webcam-gesture-reader>
3.     <webcam-retinal-scanner v-model="retinalImage"></webcam-retinal-scanner>
```

[效果预览](#) »

## 非父子组件通信

有时候非父子关系的组件也需要通信。在简单的场景下，使用一个空的 Vue 实例作为中央事件总线：

```
1.     var bus = new Vue()
1.     // 触发组件 A 中的事件
2.     bus.$emit('id-selected', 1)
1.     // 在组件 B 创建的钩子中监听事件
2.     bus.$on('id-selected', function (id) {
3.         // ...
4.     })
```

在更多复杂的情况下，你应该考虑使用专门的状态管理模式。

## 使用 Slots 分发内容

在使用组件时，常常要像这样组合它们：

```
1.     <app>
2.     <app-header></app-header>
3.     <app-footer></app-footer>
4.     </app>
```

注意两点：

- 1.

`<app>` 组件不知道它的挂载点会有什么内容。挂载点的内容是由 `<app>` 的父组件决定的。

2.

3.

`<app>` 组件很可能有它自己的模版。

4.

为了让组件可以组合，我们需要一种方式来混合父组件的内容与子组件自己的模板。这个过程被称为（或“transclusion”如果你熟悉 Angular）。Vue.js 实现了一个内容分发 API，参照了当前 [Web 组件规范草案](#)，使用特殊的 `<slot>` 元素作为原始内容的插槽。

## 编译作用域

在深入内容分发 API 之前，我们先明确内容的编译作用域。假定模板为：

1. `<child-component>`
2. `{{ message }}`
3. `</child-component>`

`message` 应该绑定到父组件的数据，还是绑定到子组件的数据？答案是父组件。组件作用域简单地说是：

父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译。

一个常见错误是试图在父组件模板内将一个指令绑定到子组件的属性/方法：

1. `<!-- 无效 -->`
2. `<child-component v-show="someChildProperty"></child-component>`

假定 `someChildProperty` 是子组件的属性，上例不会如预期那样工作。父组件模板不应该知道子组件的状态。

如果要绑定子组件内的指令到一个组件的根节点，应当在它的模板内这么做：

1. `Vue.component('child-component', {`
2. `// 有效，因为是在正确的作用域内`
3. `template: '<div v-show="someChildProperty">Child</div>',`
4. `data: function () {`



```
5.     return {
6.       someChildProperty: true
7.     }
8.   }
9. })
```

类似地，分发内容是在父组件作用域内编译。

## 单个 Slot

除非子组件模板包含至少一个 `<slot>` 插口，否则父组件的内容将会被。当子组件模板只有一个没有属性的 `slot` 时，父组件整个内容片段将插入到 `slot` 所在的 DOM 位置，并替换掉 `slot` 标签本身。

最初在 `<slot>` 标签中的任何内容都被视为。备用内容在子组件的作用域内编译，并且只有在宿主元素为空，且没有要插入的内容时才显示备用内容。

假定 `my-component` 组件有下面模板：

```
1.   <div>
2.     <h2>I'm the child title</h2>
3.     <slot>
4.       如果没有分发内容则显示我。
5.     </slot>
6.   </div>
```

父组件模版：

```
1.   <div>
2.     <h1>I'm the parent title</h1>
3.     <my-component>
4.       <p>This is some original content</p>
5.       <p>This is some more original content</p>
6.     </my-component>
7.   </div>
```

渲染结果：

```
1.      <div>
2.      <h1>I'm the parent title</h1>
3.      <div>
4.      <h2>I'm the child title</h2>
5.      <p>This is some original content</p>
6.      <p>This is some more original content</p>
7.      </div>
8.      </div>
```

## 具名 Slots

`<slot>` 元素可以用一个特殊的属性 `name` 来配置如何分发内容。多个 `slot` 可以有不同的名字。具名 `slot` 将匹配内容片段中有对应 `slot` 特性的元素。

仍然可以有一个匿名 `slot`，它是，作为找不到匹配的内容片段的备用插槽。如果没有默认的 `slot`，这些找不到匹配的内容片段将被抛弃。

例如，假定我们有一个 `app-layout` 组件，它的模板为：

```
1.      <div class="container">
2.      <header>
3.      <slot name="header"></slot>
4.      </header>
5.      <main>
6.      <slot></slot>
7.      </main>
8.      <footer>
9.      <slot name="footer"></slot>
10.     </footer>
11.     </div>
```

父组件模版：

```
1.      <app-layout>
2.      <h1 slot="header">Here might be a page title</h1>
```

```
3.     <p>A paragraph for the main content.</p>
4.     <p>And another one.</p>
5.     <p slot="footer">Here's some contact info</p>
6.     </app-layout>
```

渲染结果为：

```
1.     <div class="container">
2.
3.     <h1>Here might be a page title</h1>
4.     </header>
5.     <main>
6.     <p>A paragraph for the main content.</p>
7.     <p>And another one.</p>
8.     </main>
9.     <footer>
10.    <p>Here's some contact info</p>
11.    </footer>
12.    </div>
```

在组合组件时，内容分发 API 是非常有用的机制。

## 动态组件

多个组件可以使用同一个挂载点，然后动态地在它们之间切换。使用保留的 `<component>`

元素，动态地绑定到它的 `is` 特性：

```
1.     var vm = new Vue({
2.       el: '#example',
3.       data: {
4.         currentView: 'home'
5.       },
6.       components: {
7.         home: { /* ... */ },
```

```

8.     posts: { /* ... */ },
9.     archive: { /* ... */ }
10.   }
11. })
1.     <component v-bind:is="currentView">
2.     <!-- 组件在 vm.currentview 变化时改变! -->
3.     </component>

```

也可以直接绑定到组件对象上：

```

1.     var Home = {
2.     template: '<p>Welcome home!</p>'
3.   }
4.     var vm = new Vue({
5.     el: '#example',
6.     data: {
7.     currentView: Home
8.   }
9. })

```

## keep-alive

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 `keep-alive` 指令参数：

```

1.     <keep-alive>
2.     <component :is="currentView">
3.     <!-- 非活动组件将被缓存! -->
4.     </component>
5.     </keep-alive>

```

在 API 参考查看更多 `<keep-alive>` 的细节。

## 杂项

## 编写可复用组件

在编写组件时，记住是否要复用组件有好处。一次性组件跟其它组件紧密耦合没关系，但是可复用组件应当定义一个清晰的公开接口。

Vue 组件的 API 来自三部分 - props, events 和 slots :

- 允许外部环境传递数据给组件
- 
- 允许组件触发外部环境的副作用
- 
- 允许外部环境将额外的内容组合在组件中。
- 

使用 `v-bind` 和 `v-on` 的简写语法，模板的缩进清楚且简洁：

## 子组件索引

尽管有 props 和 events ，但是有时仍然需要在 JavaScript 中直接访问子组件。为此可以使用 `ref` 为子组件指定一个索引 ID 。例如：

1. `<div id="parent">`
2. `<user-profile ref="profile"></user-profile>`
3. `</div>`
1. `var parent = new Vue({ el: '#parent' })`
2. `// 访问子组件`
3. `var child = parent.$refs.profile`

当 `ref` 和 `v-for` 一起使用时， `ref` 是一个数组或对象，包含相应的子组件。

`$refs` 只在组件渲染完成后才填充，并且它是非响应式的。它仅作为一个直接访问子组件的应急方案——应当避免在模版或计算属性中使用 `$refs`。

## 异步组件

在大型应用中，我们可能需要将应用拆分为多个小模块，按需从服务器下载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：

```
1.   Vue.component('async-example', function (resolve, reject) {
2.     setTimeout(function () {
3.       resolve({
4.         template: '<div>I am async!</div>'
5.       })
6.     }, 1000)
7.   })
```

工厂函数接收一个 `resolve` 回调，在收到从服务器下载的组件定义时调用。也可以调用

`reject(reason)` 指示加载失败。这里 `setTimeout` 只是为了演示。怎么获取组件完全由你决定。推荐配合使用 `Webpack` 的代码分割功能：

```
1.   Vue.component('async-webpack-example', function (resolve) {
2.     // 这个特殊的 require 语法告诉 webpack
3.     // 自动将编译后的代码分割成不同的块，
4.     // 这些块将通过 Ajax 请求自动下载。
5.     require(['./my-async-component'], resolve)
6.   })
```

你可以使用 Webpack 2 + ES2015 的语法返回一个 `Promise` resolve 函数：

```
1.   Vue.component(
2.     'async-webpack-example',
3.     () => System.import('./my-async-component')
4.   )
```

如果你是 用户,可能就无法使用异步组件了,它的作者已经表明 Browserify 是不支持异步加载的。如果这个功能对你很重要,请使用 Webpack。

## 组件命名约定

当注册组件 (或者 props) 时, 可以使用 kebab-case , camelCase , 或 TitleCase 。Vue 不关心这个。

```
1.      // 在组件定义中
2.      components: {
3.      // 使用 camelCase 形式注册
4.      'kebab-cased-component': { /* ... */ },
5.      'camelCasedComponent': { /* ... */ },
6.      'TitleCasedComponent': { /* ... */ }
7.    }
```

在 HTML 模版中, 请使用 kebab-case 形式 :

```
1.      <!-- 在 HTML 模版中始终使用 kebab-case -->
2.      <kebab-cased-component></kebab-cased-component>
3.      <camel-cased-component></camel-cased-component>
4.      <title-cased-component></title-cased-component>
```

当使用字符串模式时, 可以不受 HTML 的 case-insensitive 限制。这意味实际上在模版中, 你可以使用 camelCase 、 PascalCase 或者 kebab-case 来引用你的组件和 prop :

```
1.      <!-- 在字符串模版中可以用任何你喜欢的方式! -->
2.      <my-component></my-component>
3.      <myComponent></myComponent>
4.      <MyComponent></MyComponent>
```

如果组件未经 `slot` 元素传递内容, 你甚至可以在组件名后使用 `/` 使其自闭合 :

```
1.      <my-component/>
```

当然, 这只在字符串模版中有效。因为自闭的自定义元素是无效的 HTML , 浏览器原生的解析器也无法识别它。

## 递归组件

组件在它的模板内可以递归地调用自己, 不过, 只有当它有 name 选项时才可以 :

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

When you register a component globally using `Vue.component`, the global ID is automatically set as the component's `name` option.

```
1.      Vue.component('unique-name-of-my-component', {  
2.        // ...  
3.      })
```

If you're not careful, recursive components can also lead to infinite loops:

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

上面组件会导致一个错误 “max stack size exceeded”，所以要确保递归调用有终止条件（比如递归调用时使用 `v-if` 并让他最终返回 `false`）。

## 内联模版

如果子组件有 `inline-template` 特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板更灵活。

```
1.      <my-component inline-template>  
2.      <div>  
3.        <p>These are compiled as the component's own template.</p>  
4.        <p>Not parent's transclusion content.</p>  
5.      </div>  
6.    </my-component>
```

但是 `inline-template` 让模板的作用域难以理解。最佳实践是使用 `template` 选项在组件内定义模板或者在 `.vue` 文件中使用 `template` 元素。

## X-Templates

另一种定义模版的方式是在 JavaScript 标签里使用 `text/x-template` 类型，并且指定一个 id。例如：

```
1.      <script type="text/x-template" id="hello-world-template">  
2.        <p>Hello hello hello</p>
```



```
3.     </script>
1.     Vue.component('hello-world', {
2.       template: '#hello-world-template'
3.     })
```

这在有很多模版或者小的应用中有用，否则应该避免使用，因为它将模版和组件的其他定义隔离了。

## 使用 `v-once` 的低级静态组件 (Cheap Static Component)

尽管在 Vue 中渲染 HTML 很快，不过当组件中包含静态内容时，可以考虑使用 `v-once` 将渲染结果缓存起来，就像这样：

```
1.     Vue.component('terms-of-service', {
2.       template: '\
3.         <div v-once>\
4.           <h1>Terms of Service</h1>\
5.           ... a lot of static content ... \
6.         </div>\
7.       '
8.     })
```

关于我们 联系我们 留言板

手册网

## 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。你来之这里的目的是看看 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，伴随于此，有配套的路由和负责处理全局状态管理的库。

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。我们需要指出 React 比 Vue 更好的地方，像是他们的生态系统和丰富的自定义渲染器。

React 社区为我们准确进行平衡的考量提供了[非常积极地帮助](#)，特别感谢来自 React 团队的 Dan Abramov。他非常慷慨的花费时间来贡献专业知识，帮助我们完善这篇文档，最后我们对最终结果都[十分满意](#)。

有了上面这些内容，我们希望你能对下面这两个库的比较内容的公正性感到放心。

## 性能简介

到目前为止，针对现实情况的测试中，Vue 的性能是优于 React 的。如果你对此表示怀疑，请继续阅读。我们会解释为什么会这样（并且会提供一个与 React 团队共同约定的比较基准）。

### 渲染性能

在渲染用户界面的时候，DOM 的操作成本是最高的，不幸的是没有库可以让这些原始操作变得更快。

我们能做到的最好效果就是：

1. 尽量减少 DOM 操作。Vue 和 React 都使用虚拟 DOM 来实现，并且两者工作的效果一样好。
2. 尽量减少除 DOM 操作以外的其他操作。这是 Vue 和 React 所不同的地方。

在 React 中，我们设定渲染一个元素的额外开销是 1，而平均渲染一个组件的开销是 2。那么在 Vue 中，一个元素的开销更像是 0.1，但是平均组件的开销将会是 4，这是由于我们需要设定响应系统所导致的。

这意味着：在典型的应用中，由于需要渲染的元素比组件的数量是更多的，因此 Vue 的性能表现将会远优于 React。然而，在极端情况下，比如每个组件只渲染一个元素，Vue 就会通常更慢一些。当然接下来还有其他的原因。

Vue 和 React 也提供功能性组件，这些组件因为都是没有声明，没有实例化的，因此会花费更少的开销。当这些都用于关键性能的场景时，Vue 将会更快。为了证明这点，我们建立了一个简单的[参照项目](#)，它负责渲染 10,000 个列表项 100 次。我们鼓励你基于此去尝试运行一下。然而在实际上，由于浏览器和硬件的差异甚至 JavaScript 引擎的不同，结果都会相应有所不同。

如果你懒得去做，下面的数值是来自于一个 2014 年产的 MacBook Air 并在 Chrome 52 版本下运行所产生的。为了避免偶然性，每个参照项目都分别运行 20 次并取自最好的结果：

	Vue	React
<b>Fastest</b>	23ms	63ms
<b>Median</b>	42ms	81ms
<b>Average</b>	51ms	94ms
<b>95th Perc.</b>	73ms	164ms
<b>Slowest</b>	343ms	453ms

## 更新性能

在 React 中，你需要在处处去实现 `shouldComponentUpdate`，并且用不可变数据结构才能实现最优化的渲染。在 Vue 中，组件的依赖被自动追踪，所以当这些依赖项变动时，它才会更新。唯一需要注意的可能需要进一步优化的一点是在长列表中，需要在每项上添加一个 `key` 属性。

这意味着，未经优化的 Vue 相比未经优化的 React 要快的多。由于 Vue 改进过渲染性能，甚至全面优化过的 React 通常也会慢于开箱即用的 Vue。

## 开发中

显然，在生产环境中的性能是至关重要的，目前为止我们所具体讨论的便是针对此环境。但开发过程中的表现也不容小视。不错的是用 Vue 和 React 开发大多数应用的速度都是足够快的。

然而，假如你要开发一个对性能要求比较高的数据可视化或者动画的应用时，你需要了解到下面这点：在开发中，Vue 每秒最高处理 10 帧，而 React 每秒最高处理不到 1 帧。

这是由于 React 有大量的检查机制，这会让它提供许多有用的警告和错误提示信息。我们同样认为这些是很重要的，但是我们在实现这些检查时，也更加密切地关注了性能方面。

## HTML & CSS

在 React 中，它们都是 JavaScript 编写的，听起来这十分简单和优雅。然而不幸的事实是，JavaScript 内的 HTML 和 CSS 会产生很多痛点。在 Vue 中我们采用 Web 技术并在其上扩展。接下来将通过一些实例向你展示这意味的是什么。

## 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。这有一个[通过 React 社区审核过的例子](#)：

```
1.      render () {
2.          let { items } = this.props
3.          let children
4.          if ( items.length > 0 ) {
5.              children = (
6.                  <ul>
7.                      {items.map( item =>
8.                          <li key={item.id}>{item.name}</li>
9.                      )}
10.                 </ul>
11.             )
12.          } else {
13.              children = <p>No items found.</p>
14.          }
15.          return (
16.              <div className = 'list-container'>
17.                  {children}
18.              </div>
19.          )
20.      }
```

JSX 的渲染功能有下面这些优势：

- 你可以使用完整的编程语言 JavaScript 功能来构建你的视图页面。
- 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）。

在 Vue 中，由于有时需要用这些功能，我们也提供了渲染功能 并且支持了 JSX。然而，对于大多数组件来说，渲染功能是不推荐使用了。

在这方面，我们提供的是更简单的模板：

```
1.      <template>
2.          <div class="list-container">
3.              <ul v-if="items.length">
4.                  <li v-for="item in items">
5.                      {{ item.name }}
6.                  </li>
7.              </ul>
8.              <p v-else>No items found.</p>
9.          </div>
10.     </template>
```

优点如下：

- 在写模板的过程中，样式风格已定并涉及更少的功能实现。
- 模板总是会被声明的。
- 模板中任何 HTML 语法都是有效的。
- 阅读起来更贴合英语（比如，for each item in items）。
- 不需要高级版本的 JavaScript 语法，来增加可读性。

这样，不仅开发人员更容易编写代码，设计人员和开发人员也可以更容易的分析代码和贡献代码。

这还没有结束。Vue 拥抱 HTML，而不是用 JavaScript 去重塑它。在模板内，Vue 也允许你用预处理器比如 Pug（原名 Jade）。

React 生态也有一个[项目](#)允许你写模板，但是存在一些缺点：

- 功能远没有 Vue 模板系统丰富。
- 需要从组件文件中分离出 HTML 代码。
- 这是个第三方库，而非官方支持，可能未来核心库更新就不再支持。

## CSS 的组件作用域

除非你把组件分布在多个文件上(例如 [CSS Modules](#))，要不在 React 中作用域内的 CSS 就会产生警告。非常简单的 CSS 还可以工作，但是稍微复杂点的，比如悬停状态、媒体查询、伪类选择符等要么通过沉重的依赖来重做要么就直接不能用。

而 Vue 可以让你在每个单文件组件中完全访问 CSS。

```
1.     <style scoped>
2.         @media (min-width: 250px) {
3.             .list-container:hover {
4.                 background: orange;
5.             }
6.         }
7.     </style>
```

这个可选 `scoped` 属性会自动添加一个唯一的属性（比如 `data-v-1`）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-1]:hover`。

最后，就像 HTML 一样，你可以选择自己偏爱的 CSS 预处理器编写 CSS。这可以让你围绕设计为中心展开工作，而不是引入专门的库来增加你应用的体积和复杂度。

## 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://unpkg.com/vue/dist/vue.js"></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。

## 本地渲染

ReactNative 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 是阿里的跨平台用户界面开发框架，Weex 的 JavaScript 框架运行时用的就是 Vue。这意味着在 Weex 的帮助下，你使用 Vue 语法开发的组件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。

在现在，Weex 还在积极发展，成熟度也不能和 ReactNative 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。

## 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。

## Angular 1

Vue 的一些语法和 Angular 的很相似（例如 `v-if` vs `ng-if`）。因为 Angular 是 Vue 早期开发的灵感来源。然而，Angular 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 Angular 1 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 Angular 1 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

Angular 1 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 Angular 中两者有不少相混的地方。

## 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 Angular 1 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。Angular 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 2 和 Vue 用相似的设计解决了一些 Angular 1 中存在的问题。

## Angular 2

我们单独将 Angular 2 作分类，因为它完全是一个全新的框架。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 1 面向的是较小的应用程序，Angular 2 已转移焦点，面向的是大型企业应用。在这一点上 TypeScript 经常会被引用，它对那些喜欢用 Java 或者 C# 等类型安全的语言的人是非常有用的。



Vue 也十分适合制作[企业应用](#)，你也可以通过使用[官方类型](#)或[用户贡献的装饰器](#)来支持 TypeScript，这完全是自由可选的。

## 大小和性能

在性能方面，这两个框架都非常的快。但目前尚没有足够的数据用例来具体展示。如果你一定要量化这些数据，你可以查看[第三方参照](#)，它表明 Vue 2 相比 Angular2 是更快的。

在大小方面，虽然 Angular 2 使用 tree-shaking 和离线编译技术使代码体积减小了许多。但包含编译器和全部功能的 Vue2(23kb) 相比 Angular 2(50kb) 还是要小的多。但是要注意，用 Angular 2 的 App 的体积缩减是使用了 tree-shaking 移除了那些框架中没有用到的功能，但随着功能引入的不断增多，尺寸会变得越来越来大。

## 灵活性

Vue 相比于 Angular 2 则更加灵活，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构建。有人可能喜欢用统一的方式来构建，也有很多开发者喜欢这种灵活自由的方式。

## 学习曲线

开始使用 Vue，你使用的是熟悉的 HTML、符合 ES5 规则的 JavaScript（也就是纯 JavaScript）。有了这些基本的技能，你可以快速地掌握它(指南)并投入开发。

Angular 2 的学习曲线是非常陡峭的。即使不包括 TypeScript，它的[开始指南](#)中所用的就有 ES2015 标准的 JavaScript，18 个 NPM 依赖包，4 个文件和超过 3 千多字的介绍，这一切都是为了完成个 Hello World。而 [Vue's Hello World](#) 就非常简单。甚至我们并不用花费一整个页面去介绍它。

## 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 甩开 Ember 几条街，即使是 Ember 2.0 的最新 Glimmer 引擎。Vue 能够自动批量更新，而 Ember 在关键性能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 1.0 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。

Polymer 自定义的元素是用 HTML 文件来创建的，这会限制使用 JavaScript/CSS（和被现代浏览器普遍支持的语言特性）。相比之下，Vue 的单文件组件允许你非常容易的使用 ES2015 和你想用的 CSS 预编译处理器。

在部署生产环境时，Polymer 建议使用 HTML Imports 加载所有资源。而这要求服务器和客户端都支持 Http 2.0 协议，并且浏览器实现了此标准。这是否可行就取决于你的目标用户和部署环境了。如果状况不佳，你必须用 Vulcanizer 工具来打包 Polymer 元素。而在这方面，Vue 可以结合异步组件的特性和 Webpack 的代码分割特性来实现懒加载（lazy-loaded）。这同时确保了对旧浏览器的兼容且又能更快加载。

而 Vue 和 Web Component 标准进行深层次的整合也是完全可行的，比如使用 Custom Elements、Shadow DOM 的样式封装。然而在我们做出严肃的实现承诺之前，我们目前仍在等待相关标准成熟，进而再广泛应用于主流的浏览器中。

## Riot

Riot 2.0 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：

- 根据真实条件来渲染，Riot 根据是否有分支简单显示或隐藏所有内容。
- 功能更加强大的路由机制，Riot 的路由功能的 API 是极少的。
- 更多成熟工具的支持。Vue 提供官方支持 Webpack、Browserify 和 SystemJS，而 Riot 是依靠社区来建立集成系统。
- 过渡效果系统。Riot 现在还没有提供。
- 更好的性能。Riot 尽管声称其使用了虚拟 DOM，但实际上用的还是脏检查机制，因此和 Angular 1 患有相同的性能问题。

## 高级教程

### Render 函数

## 基础

Vue 推荐使用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，这就是，它比 `template` 更接近编译器。

让我们先深入一个使用 `render` 函数的简单例子，假设你想生成一个带锚链接的标题：

```
1.      <h1>
2.          <a name="hello-world" href="#hello-world">
3.              Hello world!
4.          </a>
5.      </h1>
```

在 HTML 层，我们决定这样定义组件接口：

```
1.      <anchored-heading :level="1">Hello world!</anchored-heading>
```

当我们开始写一个通过 `level` prop 动态生成 heading 标签的组件，你可很快能想到这样实现：

```
1. <script type="text/x-template" id="anchored-heading-template">
2.   <div>
3.     <h1 v-if="level === 1">
4.       <slot></slot>
5.     </h1>
6.     <h2 v-if="level === 2">
7.       <slot></slot>
8.     </h2>
9.     <h3 v-if="level === 3">
10.      <slot></slot>
11.    </h3>
12.    <h4 v-if="level === 4">
13.      <slot></slot>
14.    </h4>
15.    <h5 v-if="level === 5">
16.      <slot></slot>
17.    </h5>
18.    <h6 v-if="level === 6">
19.      <slot></slot>
20.    </h6>
21.  </div>
22. </script>

1. Vue.component('anchored-heading', {
2.   template: '#anchored-heading-template',
3.   props: {
4.     level: {
5.       type: Number,
6.       required: true
```

```
7.         }
8.     }
9. })
```

`template` 在这种场景中就表现的有些冗余了。虽然我们重复使用 `<slot></slot>` 来接收每一个级别的标题标签，在标题标签中添加相同的锚点元素。但是些都会被包裹在一个无用的 `div` 中，因为组件必须有根节点。

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：

```
1.   Vue.component('anchored-heading', {
2.     render: function (createElement) {
3.         return createElement(
4.             'h' + this.level,    // tag name  标签名称
5.             this.$slots.default // 子组件中的阵列
6.         )
7.     },
8.     props: {
9.         level: {
10.             type: Number,
11.             required: true
12.         }
13.     }
14. })
```

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，你需要知道当你不使用 `slot` 属性向组件中传递内容时，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，

`createElement` 参数

第二件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：

```
1.      // @returns {VNode}
2.      createElement(
3.          // {String | Object | Function}
4.          // 一个 HTML 标签，组件设置，或一个函数
5.          // 必须 Return 上述其中一个
6.          'div',
7.          // {Object}
8.          // 一个对应属性的数据对象
9.          // 您可以在 template 中使用.可选项.
10.         {
11.             // (下一章，将详细说明相关细节)
12.         },
13.         // {String | Array}
14.         // 子节点(VNodes). 可选项.
15.         [
16.             createElement('h1', 'hello world'),
17.             createElement(MyComponent, {
18.                 props: {
19.                     someProp: 'foo'
20.                 }
21.             }),
22.             'bar'
23.         ]
24.     )
```

## 完整数据对象

有一件事要注意：在 templates 中，`v-bind:class` 和 `v-bind:style`，会有特别的处理，他们在 VNode 数据对象中，为最高级配置。

```
1.      {
2.          // 和`v-bind:class`一样的 API
3.          'class': {
4.              foo: true,
5.              bar: false
6.          },
7.          // 和`v-bind:style`一样的 API
8.          style: {
9.              color: 'red',
10.             fontSize: '14px'
11.          },
12.          // 正常的 HTML 特性
13.          attrs: {
14.              id: 'foo'
15.          },
16.          // 组件 props
17.          props: {
18.              myProp: 'bar'
19.          },
20.          // DOM 属性
21.          domProps: {
22.              innerHTML: 'baz'
23.          },
24.          // 事件监听器基于 "on"
25.          // 所以不再支持如 v-on:keyup.enter 修饰器
26.          // 需要手动匹配 keyCode。
27.          on: {
28.              click: this.clickHandler
29.          },
30.          // 仅对于组件，用于监听原生事件，而不是组件使用 vm.$emit 触发的事件。
```

```

31.     nativeOn: {
32.         click: this.nativeClickHandler
33.     },
34.     // 自定义指令. 注意事项: 不能对绑定的旧值设值
35.     // Vue 会为您持续追踪
36.     directives: [
37.         {
38.             name: 'my-custom-directive',
39.             value: '2'
40.             expression: '1 + 1',
41.             arg: 'foo',
42.             modifiers: {
43.                 bar: true
44.             }
45.         }
46.     ],
47.     // 如果子组件有定义 slot 的名称
48.     slot: 'name-of-slot'
49.     // 其他特殊顶层属性
50.     key: 'myKey',
51.     ref: 'myRef'
52. }

```

## 完整示例

有了这方面的知识，我们现在可以完成我们最开始想实现的组件：

```

1.     var getChildrenTextContent = function (children) {
2.         return children.map(function (node) {
3.             return node.children
4.                 ? getChildrenTextContent(node.children)
5.                 : node.text

```



```

6.         }).join("")
7.     }
8.     Vue.component('anchored-heading', {
9.         render: function (createElement) {
10.             // create kebabCase id
11.             var headingId = getChildrenTextContent(this.$slots.default)
12.                 .toLowerCase()
13.                 .replace(/\W+/g, '-')
14.                 .replace(/(^|-|-$)/g, "")
15.             return createElement(
16.                 'h' + this.level,
17.                 [
18.                     createElement('a', {
19.                         attrs: {
20.                             name: headingId,
21.                             href: '#' + headingId
22.                         }
23.                     }, this.$slots.default)
24.                 ]
25.             )
26.         },
27.         props: {
28.             level: {
29.                 type: Number,
30.                 required: true
31.             }
32.         }
33.     })

```

## 约束

## VNodes 必须唯一

所有组件树中的 VNodes 必须唯一。这意味着，下面的 render function 是无效的：

```
1.     render: function (createElement) {
2.         var myParagraphVNode = createElement('p', 'hi')
3.         return createElement('div', [
4.             // Yikes - duplicate VNodes!
5.             myParagraphVNode, myParagraphVNode
6.         ])
7.     }
```

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 个重复的段落：

```
1.     render: function (createElement) {
2.         return createElement('div',
3.             Array.apply(null, { length: 20 }).map(function () {
4.                 return createElement('p', 'hi')
5.             })
6.         )
7.     }
```

## 使用 JavaScript 代替模板功能

无论什么都可以使用原生的 JavaScript 来实现，Vue 的 render 函数不会提供专用的 API。

比如，template 中的 `v-if` 和 `v-for`：

```
1.     <ul v-if="items.length">
2.         <li v-for="item in items">{{ item.name }}</li>
3.     </ul>
4.     <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：

```
1.     render: function (createElement) {
2.         if (this.items.length) {
```

```

3.         return createElement('ul', this.items.map(function (item) {
4.             return createElement('li', item.name)
5.         })))
6.     } else {
7.         return createElement('p', 'No items found.')
8.     }
9. }

```

## JSX

如果你写了很多 `render` 函数，可能会觉得痛苦：

```

1.     createElement(
2.         'anchored-heading', {
3.             props: {
4.                 level: 1
5.             }
6.         }, [
7.             createElement('span', 'Hello'),
8.             ' world!'
9.         ]
10.    )

```

特别是模板如此简单的情况下：

```

1.     <anchored-heading :level="1">
2.         <span>Hello</span> world!
3.     </anchored-heading>

```

这就是会有一个 `Babel plugin` 插件，用于在 `Vue` 中使用 `JSX` 语法的原因，它可以让我们回到于更接近模板的语法上。

```

1.     import AnchoredHeading from './AnchoredHeading.vue'
2.     new Vue({
3.         el: '#demo',
4.         render (h) {

```

```
5.         return (  
6.             <AnchoredHeading level={1}>  
7.                 <span>Hello</span> world!  
8.             </AnchoredHeading>  
9.         )  
10.     }  
11. })
```

将 `h` 作为 `createElement` 的别名是一个通用惯例，你会发现在 `Vue` 生态系统中，实际上必须用到 `JSX`，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](#)。

## 函数化组件

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 `data`），无实例（没有 `this` 上下文）。

一个 就像这样：

```
1. Vue.component('my-component', {  
2.     functional: true,  
3.     // 为了弥补缺少的实例  
4.     // 提供第二个参数作为上下文  
5.     render: function (createElement, context) {  
6.         // ...  
7.     },  
8.     // Props 可选  
9.     props: {  
10.         // ...  
11.     }  
12. })
```

组件需要的一切都是通过上下文传递，包括：

- `props`: 提供 props 的对象
- `children`: VNode 子节点的数组
- `slots`: slots 对象
- `data`: 传递给组件的 data 对象
- `parent`: 对父组件的引用

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`。

函数化组件只是一个函数，所以渲染开销也低很多。但同样它也有完整的组件封装，你需要知道这些， 比如：

- 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：

```
1.   var EmptyList = { /* ... */ }
2.   var TableList = { /* ... */ }
3.   var OrderedList = { /* ... */ }
4.   var UnorderedList = { /* ... */ }
5.   Vue.component('smart-list', {
6.     functional: true,
7.     render: function (createElement, context) {
8.       function appropriateListComponent () {
9.         var items = context.props.items
10.        if (items.length === 0)           return EmptyList
11.        if (typeof items[0] === 'object') return TableList
```

```
12.         if (context.props.isOrdered)      return OrderedList
13.         return UnorderedList
14.     }
15.     return createElement(
16.         appropriateListComponent(),
17.         context.data,
18.         context.children
19.     )
20. },
21. props: {
22.     items: {
23.         type: Array,
24.         required: true
25.     },
26.     isOrdered: Boolean
27. }
28. })
```

## `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？

```
1.     <my-functional-component>
2.       <p slot="foo">
3.         first
4.       </p>
5.       <p>second</p>
6.     </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

---

## 开发插件

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-element`

2.

3.

添加全局资源 : 指令/过滤器/过渡等, 如 `vue-touch`

4.

5.

通过全局 `mixin` 方法添加一些组件选项, 如: `vuex`

6.

7.

添加 Vue 实例方法, 通过把它们添加到 `Vue.prototype` 上实现。

8.

9.

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`

10.

Vue.js 的插件应当有一个公开方法 `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：

```
1.     MyPlugin.install = function (Vue, options) {
2.         // 1. 添加全局方法或属性
3.         Vue.myGlobalMethod = function () {
4.             // 逻辑...
5.         }
6.         // 2. 添加全局资源
7.         Vue.directive('my-directive', {
8.             bind (el, binding, vnode, oldVnode) {
9.                 // 逻辑...
10.            }
11.            ...
12.        })
13.        // 3. 注入组件
14.        Vue.mixin({
15.            created: function () {
16.                // 逻辑...
17.            }
18.            ...
19.        })
20.        // 4. 添加事例方法
21.        Vue.prototype.$myMethod = function (options) {
22.            // 逻辑...
23.        }
24.    }
```



# 使用插件

通过全局方法 `Vue.use()` 使用插件:

1. `// 调用 `MyPlugin.install(Vue)``
2. `Vue.use(MyPlugin)`

也可以传入一个选项对象:

1. `Vue.use(MyPlugin, { someOption: true })`

`Vue.use` 会自动阻止注册相同插件多次, 届时只会注册一次该插件。

一些插件, 如 `vue-router` 如果 `Vue` 是全局变量则自动调用 `Vue.use()`。不过在模块环

境中应当始终显式调用 `Vue.use()`:

1. `// 通过 Browserify 或 Webpack 使用 CommonJS 兼容模块`
2. `var Vue = require('vue')`
3. `var VueRouter = require('vue-router')`
4. `// 不要忘了调用此方法`
5. `Vue.use(VueRouter)`

`awesome-vue` 集合了来自社区贡献的数以千计的插件和库。

## 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。

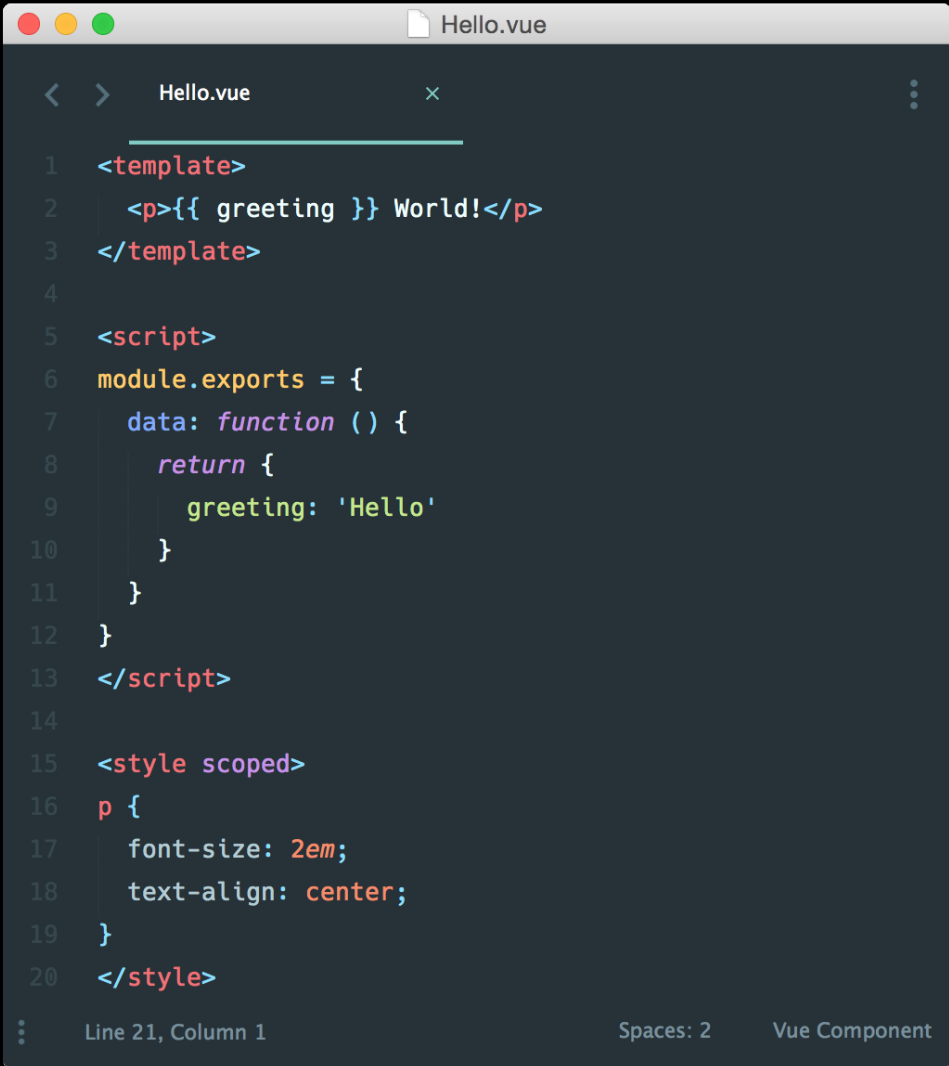
这种方案在只是使用 JavaScript 增强某个视图的中小型项目中表现得很好。然而在更复杂的项目中, 或者当你的前端完全采用 JavaScript 驱动的时候, 以下弊端就显现出来:

- 强制要求每个 `component` 中的命名不得重复
- 缺乏语法高亮, 在 HTML 有多行的时候, 需要用到丑陋的 `\`

- 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏
- 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：



```
1 <template>
2   <p>{{ greeting }} World!</p>
3 </template>
4
5 <script>
6   module.exports = {
7     data: function () {
8       return {
9         greeting: 'Hello'
10      }
11    }
12  }
13 </script>
14
15 <style scoped>
16   p {
17     font-size: 2em;
18     text-align: center;
19   }
20 </style>
```

Line 21, Column 1      Spaces: 2      Vue Component

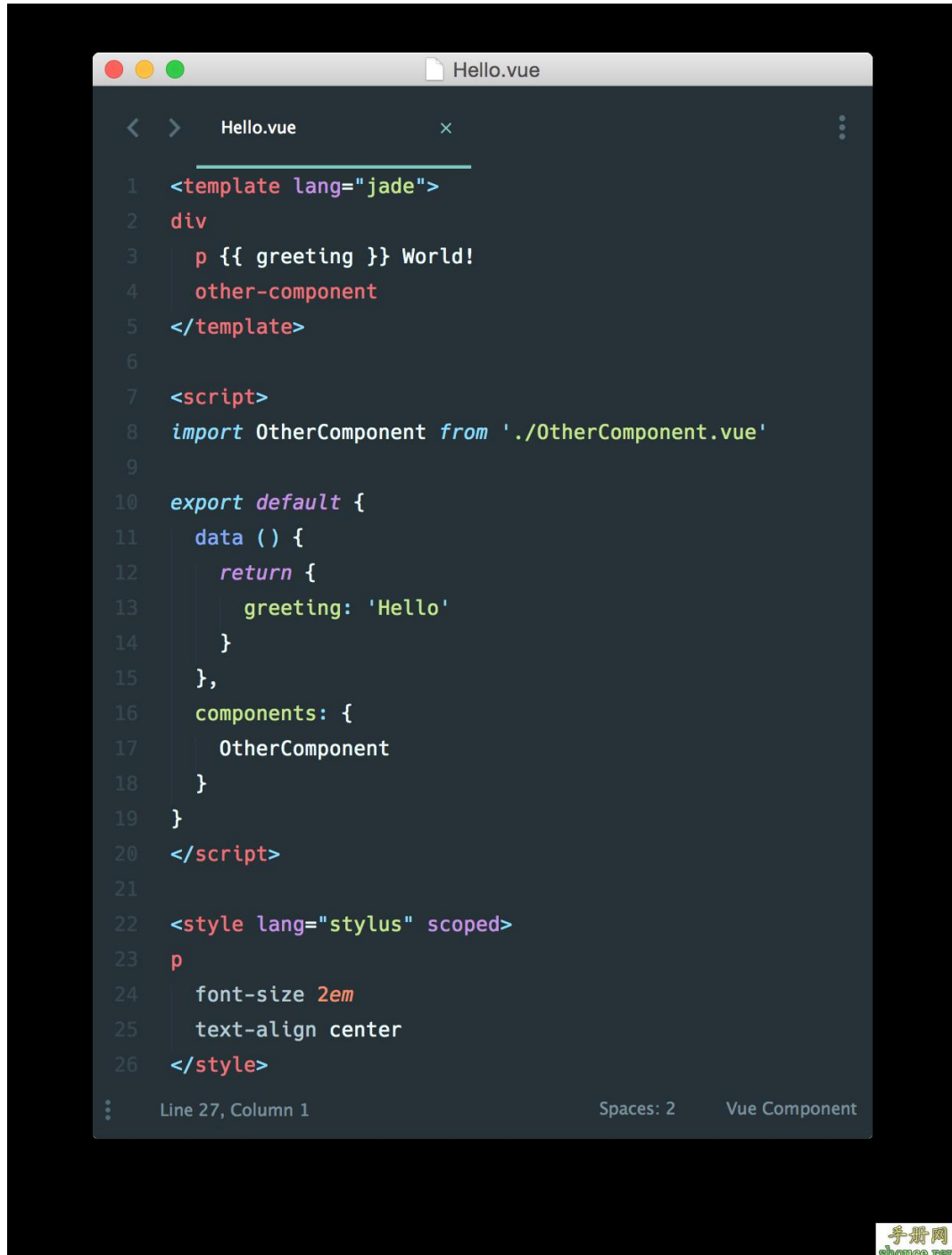
手册网  
shouce.wen

现在我们获得：

- 完整语法高亮

- CommonJS 模块
- 组件化的 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 Jade, 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 Buble, TypeScript, SCSS, PostCSS - 或者其他任何能够帮助你提高生产力的预处理器。

## 起步

## 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：

- ：阅读 [Getting Started guide](#) 直到 *10: Uninstalling global packages* 章节。
- 
- ：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。
- 

在你花一些时日了解这些资源之后，我们建议你参考 [webpack-simple](#)。只要遵循指示，你就能很快的运行一个用到 `.vue` 组件，ES2015 和 热重载(hot-reloading) 的 Vue 项目！

这个模板使用 [Webpack](#)，一个能将多个模块打包成最终应用的模块打包工具。[这个视频](#) 介绍了 Webpack 的更多相关信息。学习了这些基础知识后，你可能想看看 [这个在 Egghead.io 上的 高级 Webpack 课程](#)。

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。这个 [webpack-simple](#) 模板已经为

你准备好了所有的东西，但是如果你想了解更多关于 `.vue` 组件和 Webpack 如何一起运转的信息，你可以阅读 [vue-loader 的文档](#)。

## 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](https://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。

Webpack 中，每个模块在构建前被加载器转变，Vue 官方插件 [vue-loader](#) 用来转变 `.vue`

单文件组件。[webpack-simple](#) 模板已经准备好了一切，如果要更多了解 `.vue` 如何和 Webpack 配合工作，请看 [vue-loader 文档](#)

## 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

在测试的代码结构方面，你不必在你的组件中做任何特殊的事情使它们可测试。主要导出原始设置就可以了：

```
1.      <template>
2.        <span>{{ message }}</span>
3.      </template>
4.      <script>
5.        export default {
6.          data () {
7.            return {
8.              message: 'hello!'
9.            }
10.          },
11.          created () {
12.            this.message = 'bye!'
13.          }
14.        }
15.      </script>
```

当测试的组件时，所要做做的就是导入对象和 `Vue` 然后使用许多常见的断言：

```
1.      // 导入 Vue.js 和组件，进行测试
2.      import Vue from 'vue'
3.      import MyComponent from 'path/to/MyComponent.vue'
4.      // 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。
5.      describe('MyComponent', () => {
```

```

6.      // 检查原始组件选项
7.      it('has a created hook', () => {
8.          expect(typeof MyComponent.created).toBe('function')
9.      })
10.     // 评估原始组件选项中的函数的结果
11.     it('sets the correct default data', () => {
12.         expect(typeof MyComponent.data).toBe('function')
13.         const defaultData = MyComponent.data()
14.         expect(defaultData.message).toBe('hello!')
15.     })
16.     // 检查 mount 中的组件实例
17.     it('correctly sets the message when created', () => {
18.         const vm = new Vue(MyComponent).$mount()
19.         expect(vm.message).toBe('bye!')
20.     })
21.     // 创建一个实例并检查渲染输出
22.     it('renders the correct message', () => {
23.         const Ctor = Vue.extend(MyComponent)
24.         const vm = new Ctor().$mount()
25.         expect(vm.$el.textContent).toBe('bye!')
26.     })
27. })

```

## 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：

```

1.     <template>
2.         <p>{{ msg }}</p>
3.     </template>
4.     <script>

```

```
5.     export default {
6.         props: ['msg']
7.     }
8. </script>
```

你可以在不同的 props 中，通过 `propsData` 选项断言它的渲染输出：

```
1.     import Vue from 'vue'
2.     import MyComponent from './MyComponent.vue'
3.     // 挂载元素并返回已渲染的文本的工具函数
4.     function getRenderedText (Component, propsData) {
5.         const Ctor = Vue.extend(Component)
6.         const vm = new Ctor({ propsData }).$mount()
7.         return vm.$el.textContent
8.     }
9.     describe('MyComponent', () => {
10.         it('render correctly with different props', () => {
11.             expect(getRenderedText(MyComponent, {
12.                 msg: 'Hello'
13.             })).toBe('Hello')
14.             expect(getRenderedText(MyComponent, {
15.                 msg: 'Bye'
16.             })).toBe('Bye')
17.         })
18.     })
```

## 主张异步更新

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：

```
1.     // 在状态更新后检查生成的 HTML
2.     it('updates the rendered message when vm.message updates', done => {
```

```
3.     const vm = new Vue(MyComponent).$mount()
4.     vm.message = 'foo'
5.     // 在状态改变后和断言 DOM 更新前等待一刻
6.     Vue.nextTick(() => {
7.         expect(vm.$el.textContent).toBe('foo')
8.         done()
9.     })
10. }
```

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。

关于我们 联系我们 留言板

手册网

## 服务端渲染

## 需要服务端渲染（SSR）吗？

在开始服务端渲染前，我们先看看它能给我们带来什么，以及什么时候需要用它。

## SEO（搜索引擎优化）

谷歌和 Bing 可以很好地索引同步的 JavaScript 应用。*同步*在这里是个关键词。如果应用启动时有一个加载动画，然后内容通过 ajax 获取，那爬虫不会等待他们加载完成。

这意味着在异步获取内容的页面上很需要进行搜索引擎优化的时候，服务端渲染就很重要。

## 客户端的网络比较慢

用户可能在网络比较慢的情况下从远处访问网站 - 或者通过比较差的带宽。这些情况下，尽量减少页面请求数量，来保证用户尽快看到基本的内容。

可以用 Webpack 的代码拆分 避免强制用户下载整个单页面应用，但是，这样也远没有下载个单独的预先渲染过的 HTML 文件性能高。

## 客户端运行在老的(或者直接没有)JavaScript 引擎上



对于世界上的一些地区人，可能只能用 1998 年产的电脑访问互联网的方式使用计算机。而 Vue 只能运行在 IE9 以上的浏览器，你可以也想为那些老式浏览器提供基础内容 - 或者是在命令行中使用 **Lynx** 的时髦的黑客。

## 服务端渲染 对比 预渲染(Prerendering)

如果你只是用服务端渲染来改善一个少数的营销页面（如 首页，关于，联系 等等）的 SEO，那你可以用替换。预渲染不像服务器渲染那样即时编译 HTML，预渲染只是在构建时为了特定的路由生成特定的几个静态页面。其优势是预渲染的设置更加简单，可以保持前端是一个完整的静态站。

你用 webpack 可以很简单地通过 **prerender-spa-plugin** 来添加预渲染，它被广泛地用在 Vue 应用上 - 事实上，创建者也是 Vue 核心团队成员之一。

## Hello World

准备在行动中体验服务端渲染吧。服务端渲染(即 SSR)听起来很复杂，不过一个简单的 Node 脚本只需要 3 步就可以实现这个功能:

```
1.      // 步骤 1: 创建一个 Vue 实例
2.      var Vue = require('vue')
3.      var app = new Vue({
4.          render: function (h) {
5.              return h('p', 'hello world')
6.          }
7.      })
8.      // 步骤 2: 创建一个渲染器
9.      var renderer = require('vue-server-renderer').createRenderer()
10.     // 步骤 3: 将 Vue 实例 渲染成 HTML
11.     renderer.renderToString(app, function (error, html) {
12.         if (error) throw error
13.         console.log(html)
14.         // => <p server-rendered="true">hello world</p>
15.     })
```

这并不困难。当然这个示例比大部分应用都简单。我们不必担心：

- 一个 Web 服务器
- 流式响应

- 组件缓存
- 构建过程
- 路由
- Vuex 状态管理

这个指南的其余部分，我们将探讨这些功能怎样运作。一旦你理解了基础，我们会提供更多细节和进一步的示例来帮助你解决意外情况。

## 通过 Express Web 服务器实现简单的服务端渲染

如果没有一个 Web 服务器，很难说是服务端渲染，所以我们来补充它。我们将构建一个非常简单的服务端渲染应用，只用 ES5，也不带其他构建步骤或 Vue 插件。

启动一个应用告诉用户他们在一个页面上花了多少时间。

```
1.     new Vue({
2.       template: '<div>你已经在这花了 {{ counter }} 秒。</div>',
3.       data: {
4.         counter: 0
5.       },
6.       created: function () {
7.         var vm = this
8.         setInterval(function () {
9.           vm.counter += 1
10.        }, 1000)
11.       }
12.     })
```

为了适应服务端渲染，我们需要进行一些修改，让它可以在浏览器和 Node 中渲染：

- 在浏览器中，将我们的应用实例添加到全局上下文（`window`）上，我们可以安装它。
- 在 Node 中，导出一个工厂函数让我们可以为每个请求创建应用实例。

实现这个需要一点模板：

```
1.     // assets/app.js
2.     (function () { 'use strict'
3.       var createApp = function () {
```

```

4.          // -----
5.          // 开始常用的应用代码
6.          // -----
7.          // 主要的 Vue 实例必须返回，并且有一个根节点在 id "app" 上，这样客户端可
           以加载它。
8.          return new Vue({
9.              template: '<div id="app">你已经在这花了 {{ counter }} 秒。</div>',
10.             data: {
11.                 counter: 0
12.             },
13.             created: function () {
14.                 var vm = this
15.                 setInterval(function () {
16.                     vm.counter += 1
17.                 }, 1000)
18.             }
19.         })
20.         // -----
21.         // 结束常用的应用代码
22.         // -----
23.     }
24.     if (typeof module !== 'undefined' && module.exports) {
25.         module.exports = createApp
26.     } else {
27.         this.app = createApp()
28.     }
29.     }).call(this)

```

现在有了应用代码，接着加一个 html 文件。

```

1.     <!-- index.html -->
2.     <!DOCTYPE html>

```

```
3.     <html>
4.     <head>
5.         <title>My Vue App</title>
6.         <script src="/assets/vue.js"></script>
7.     </head>
8.     <body>
9.         <div id="app"></div>
10.        <script src="/assets/app.js"></script>
11.        <script>app.$mount('#app')</script>
12.    </body>
13. </html>
```

主要引用 `assets` 文件夹中我们先前创建的 `app.js`，以及 `vue.js` 文件，我们就有了一个可以运行的单页面应用

然后为了实现服务端渲染，在服务端需要加一个步骤。

```
1.     // server.js
2.     'use strict'
3.     var fs = require('fs')
4.     var path = require('path')
5.     // 定义全局的 Vue 为了服务端的 app.js
6.     global.Vue = require('vue')
7.     // 获取 HTML 布局
8.     var layout = fs.readFileSync('./index.html', 'utf8')
9.     // 创建一个渲染器
10.    var renderer = require('vue-server-renderer').createRenderer()
11.    // 创建一个 Express 服务器
12.    var express = require('express')
13.    var server = express()
14.    // 部署静态文件夹为 "assets"文件夹
15.    server.use('/assets', express.static(
16.        path.resolve(__dirname, 'assets')
```

```
17.     ))
18.     // 处理所有的 Get 请求
19.     server.get('*', function (request, response) {
20.         // 渲染我们的 Vue 应用为一个字符串
21.         renderer.renderToString(
22.             // 创建一个应用实例
23.             require('./assets/app')(),
24.             // 处理渲染结果
25.             function (error, html) {
26.                 // 如果渲染时发生了错误
27.                 if (error) {
28.                     // 打印错误到控制台
29.                     console.error(error)
30.                     // 告诉客户端错误
31.                     return response
32.                         .status(500)
33.                         .send('Server Error')
34.                 }
35.                 // 发送布局和 HTML 文件
36.                 response.send(layout.replace('<div id="app"></div>', html))
37.             }
38.         )
39.     })
40.     // 监听 5000 端口
41.     server.listen(5000, function (error) {
42.         if (error) throw error
43.         console.log('Server is running at localhost:5000')
44.     })
```

这样就完成了。整个示例，克隆下来深度实验。一旦它在本地运行时，你可以通过在页面右击选择[页面资源](#)（或类似操作）确认服务渲染真的运行了。可以在 body 中看到：

```
1.      <div id="app" server-rendered="true">You have been here for 0 seconds&period;</div>
```

代替:

```
1.      <div id="app"></div>
```

## 流式响应

Vue 还支持渲染，优先选择适用于支持流的 Web 服务器。允许 HTML 一边生成一般写入相应流，而不是在最后一次全部写入。其结果是请求服务速度更快，没有缺点！

为了使上一节应用代码适用流式渲染，可以简单的替换 `server.get('*',...)` 为下面的代码：

```
1.      // 拆分布局成两段 HTML
2.      var layoutSections = layout.split('<div id="app"></div>')
3.      var preAppHTML = layoutSections[0]
4.      var postAppHTML = layoutSections[1]
5.      // 处理所有的 Get 请求
6.      server.get('*', function (request, response) {
7.          // 渲染我们的 Vue 实例作为流
8.          var stream = renderer.renderToStream(require('./assets/app'))()
9.          // 将预先的 HTML 写入响应
10.         response.write(preAppHTML)
11.         // 每当新的块被渲染
12.         stream.on('data', function (chunk) {
13.             // 将块写入响应
14.             response.write(chunk)
15.         })
16.         // 当所有的块被渲染完成
17.         stream.on('end', function () {
18.             // 将 post-app HTML 写入响应
19.             response.end(postAppHTML)
20.         })
21.         // 当渲染时发生错误
22.         stream.on('error', function (error) {
```

```
23.         // 打印错误到控制台
24.         console.error(error)
25.         // 告诉服务端发生了错误
26.         return response
27.         .status(500)
28.         .send('Server Error')
29.     })
30. }
```

这不比之前的版本复杂，甚至这对你来说都不是个新概念。我们做了：

1. 建立流
2. 在应用响应前写入 HTML
3. 在可获得时将应用 HTML 写入响应
4. 在响应最后写入 HTML
5. 处理任何错误

## 组件缓存

Vue 的服务端渲染默认非常快，但是您可以通过缓存渲染好的组件进一步提高性能。这被认为是一种先进的功能，但是，如果缓存了错误的组件（或者正确的组件带有错误的内容）将导致应用渲染出错。特别注意：

不应该缓存组件包含子组件依赖全局状态（例如来自 `vuex` 的状态）。如果这么做，子组件（事实上是整个子树）也会被缓存。所以要特别注意带有 `slots` 片段或者子组件的情况。

## 设置

在警告情况之外的，我们可以用下面的方法缓存组件。

首先，你需要提供给渲染器一个 **缓存对象**。这有个简单的示例使用 `lru-cache`

```
1.     var createRenderer = require('vue-server-renderer').createRenderer
2.     var lru = require('lru-cache')
3.     var renderer = createRenderer({
4.         cache: lru(1000)
5.     })
```

这将缓存高达 1000 个独立的渲染。对于更进一步缓存到内容中的配置，看 `lru-cache` 设置

然后对于你想缓存的组件，你可以为他们提供：

- 一个唯一的名字
- 一个 `serverCacheKey` 函数，返回一个唯一的组件作用域

例如:

```
1.   Vue.component({
2.     name: 'list-item',
3.     template: '<li>{{ item.name }}</li>',
4.     props: ['item'],
5.     serverCacheKey: function (props) {
6.       return props.item.type + '::' + props.item.id
7.     }
8.   })
```

## 缓存的理想组件

任何纯组件可以被安全缓存 - 这是保证给任何组件传递一样的数据产生相同的 HTML。这些场景的例子包括：

- 静态的组件 (例如 总是尝试一样的 HTML,所以 `serverCacheKey` 函数可以被返回 `true`)
- 列表组件 (当有大量列表, 缓存他们可以改善性能)
- 通用 UI 组件 (例如 buttons, alerts, 等等 - 至少他们通过 props 获取数据而不是 slots 或者子组件)

## 构建过程，路由，和 Vuex 状态管理

现在，应该理解服务端渲染背后的基本概念了。但是，构建过程、路由、Vuex 每一个都有自己的注意事项。

要真正掌握复杂应用下的服务端渲染，我们推荐深度熟悉以下资源：

- [vue-server-renderer 文档](#):更多细节在这里, 和更多先进的主题一起的文档。 例如 [preventing cross-request contamination](#) 和 [添加独立的服务构建](#)
- [vue-hackernews-2.0](#): 明确整合了 所有主要的 Vue 库和概念在单个应用中

## 过渡效果



---

## 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 entering/leaving 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：

```
1. <div id="demo">
2.   <button v-on:click="show = !show">
3.     Toggle
4.   </button>
5.   <transition name="fade">
6.     <p v-if="show">hello</p>
7.   </transition>
8. </div>

1. new Vue({
2.   el: '#demo',
```

```
3.     data: {
4.     show: true
5.     }
6.     })

1.     .fade-enter-active, .fade-leave-active {
2.     transition: opacity .5s
3.     }
4.     .fade-enter, .fade-leave-active {
5.     opacity: 0
6.     }
```

效果预览 »

元素封装成过渡组件之后，在遇到插入或删除时，Vue 将

1.

自动嗅探目标元素是否有 CSS 过渡或动画，并在合适时添加/删除 CSS 类名。

2.

3.

如果过渡组件设置了过渡的 JavaScript 钩子函数，会在相应的阶段调用钩子函数。

4.

5.

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，与 Vue，和 Vue 的 `nextTick` 概念不同）

6.

## 过渡的-CSS-类名

会有 4 个(CSS)类名在 enter/leave 的过渡中切换

1.

`v-enter`: 定义进入过渡的开始状态。在元素被插入时生效，在下一个帧移除。

2.

3.

`v-enter-active`: 定义进入过渡的结束状态。在元素被插入时生效，在

`transition/animation` 完成之后移除。

4.

5.

`v-leave`: 定义离开过渡的开始状态。在离开过渡被触发时生效，在下一个帧移除。

6.

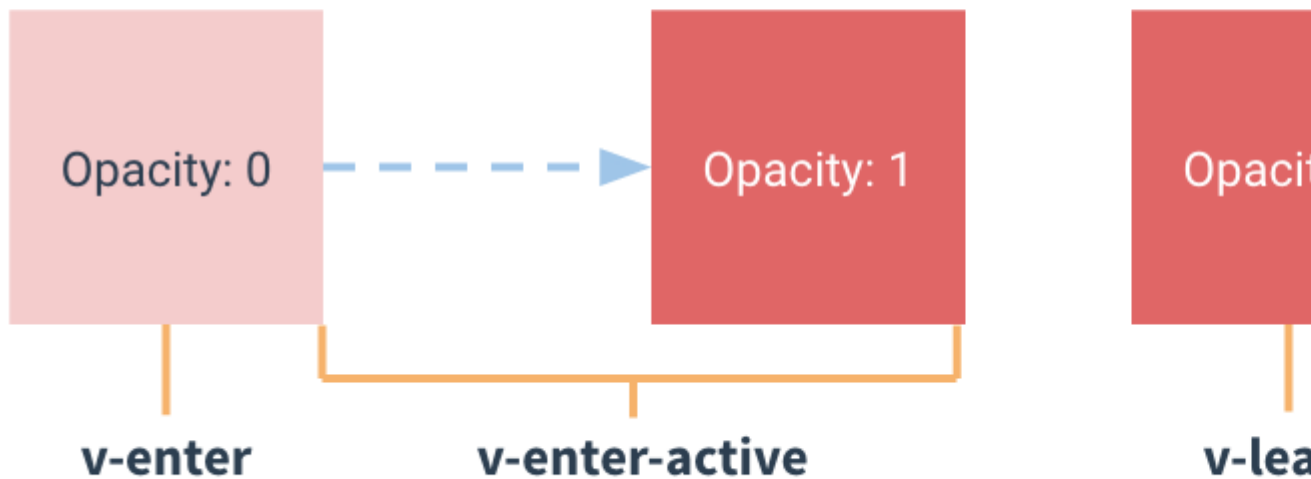
7.

`v-leave-active`: 定义离开过渡的结束状态。在离开过渡被触发时生效，在

`transition/animation` 完成之后移除。

8.

## Enter



对于这些在 `enter/leave` 过渡中切换的类名, `v-` 是这些类名的前缀。使用

`<name="my-transition">` 可以重置前缀, 比如 `v-enter` 替换为

`my-transition-enter`。

`v-enter-active` 和 `v-leave-active` 可以控制 进入/离开 过渡的不同阶段, 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：

1. `<div id="example-1">`
2. `<button @click="show = !show">`
3. `Toggle` render
4. `</button>`
5. `<transition name="slide-fade">`

```

6.     <p v-if="show">hello</p>
7.
8.     </transition>
9.
10.    </div>
11.
12.    new Vue({
13.      el: '#example-1',
14.      data: {
15.        show: true
16.      },
17.    })
18.
19.    /* 可以设置不同的进入和离开动画 */
20.
21.    /* 设置持续时间和动画函数 */
22.
23.    .slide-fade-enter-active {
24.      transition: all .3s ease;
25.    }
26.
27.    .slide-fade-leave-active {
28.      transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
29.    }
30.
31.    .slide-fade-enter, .slide-fade-leave-active {
32.      padding-left: 10px;
33.      opacity: 0;
34.    }

```

效果预览 »

## 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)

```

1.     <div id="example-2">
2.       <button @click="show = !show">Toggle show</button>

```

```
3.     <transition name="bounce">
4.     <p v-if="show">Look at me!</p>
5.     </transition>
6. </div>
```

```
1. new Vue({
2.   el: '#example-2',
3.   data: {
4.     show: true
5.   }
6. })

1. .bounce-enter-active {
2.   animation: bounce-in .5s;
3. }
4. .bounce-leave-active {
5.   animation: bounce-out .5s;
6. }
7. @keyframes bounce-in {
8.   0% {
9.     transform: scale(0);
10.  }
11.   50% {
12.     transform: scale(1.5);
13.   }
14.   100% {
15.     transform: scale(1);
16.   }
17. }
18. @keyframes bounce-out {
19.   0% {
20.     transform: scale(1);
```

```
21.     }
22.     50% {
23.         transform: scale(1.5);
24.     }
25.     100% {
26.         transform: scale(0);
27.     }
28. }
```

[效果预览 »](#)

## 自定义过渡类名

我们可以通过以下特性来自定义过渡类名：

- `enter-class`
- `enter-active-class`
- `leave-class`
- `leave-active-class`

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：

```
1.     <link href="https://unpkg.com/animate.css@3.5.1/animate.min.css" rel="stylesheet"
    type="text/css">
2.     <div id="example-3">
3.         <button @click="show = !show">
4.             Toggle render
5.         </button>
6.         <transition
7.             name="custom-classes-transition"
8.             enter-active-class="animated tada">
```

```
9.         leave-active-class="animated bounceOutRight"
10.     >
11.     <p v-if="show">hello</p>
12. </transition>
13. </div>
1. new Vue({
2.   el: '#example-3',
3.   data: {
4.     show: true
5.   }
6. })
```

效果预览 »

## 同时使用 Transitions 和 Animations

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。在这种情况下，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。

## JavaScript 钩子

可以在属性中声明 JavaScript 钩子

```
1. <transition
2.   v-on:before-enter="beforeEnter"
3.   v-on:enter="enter"
4.   v-on:after-enter="afterEnter"
5.   v-on:enter-cancelled="enterCancelled"
6.   v-on:before-leave="beforeLeave"
```



```
7.      v-on:leave="leave"
8.      v-on:after-leave="afterLeave"
9.      v-on:leave-cancelled="leaveCancelled"
10.    >
11.    <!-- ... -->
12.  </transition>
1.      // ...
2.      methods: {
3.        // -----
4.        //  进入中
5.        // -----
6.        beforeEnter: function (el) {
7.          // ...
8.        },
9.        // 此回调函数是可选项的设置
10.       // 与 CSS 结合时使用
11.        enter: function (el, done) {
12.          // ...
13.          done()
14.        },
15.        afterEnter: function (el) {
16.          // ...
17.        },
18.        enterCancelled: function (el) {
19.          // ...
20.        },
21.        // -----
22.        // 离开时
23.        // -----
24.        beforeLeave: function (el) {
```

```

25.      // ...
26.    },
27.    // 此回调函数是可选项的设置
28.    // 与 CSS 结合时使用
29.    leave: function (el, done) {
30.      // ...
31.      done()
32.    },
33.    afterLeave: function (el) {
34.      // ...
35.    },
36.    // leaveCancelled 只用于 v-show 中
37.    leaveCancelled: function (el) {
38.      // ...
39.    }
40.  }

```

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，`enterleavedone`。否则，它们会被同步调用，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：

```

1.      <!--
2.      Velocity works very much like jQuery.animate and is
3.      a great option for JavaScript animations
4.      -->
5.      <script
src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>
6.      <div id="example-4">

```

7. <button @click="show = !show">

8. Toggle

9. </button>

10. <transition

11. v-on:before-enter="beforeEnter"

12. v-on:enter="enter"

13. v-on:leave="leave"

14. v-bind:css="false"

15. >

16. <p v-if="show">

17. Demo

18. </p>

19. </transition>

20. </div>

1. new Vue({

2. el: '#example-4',

3. data: {

4. show: false

5. },

6. methods: {

7. beforeEnter: function (el) {

8. el.style.opacity = 0

9. },

10. enter: function (el, done) {

11. Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })

12. Velocity(el, { fontSize: '1em' }, { complete: done })

13. },

14. leave: function (el, done) {

15. Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })

16. Velocity(el, { rotateZ: '100deg' }, { loop: 2 })

```
17.         Velocity(el, {
18.             rotateZ: '45deg',
19.             translateY: '30px',
20.             translateX: '30px',
21.             opacity: 0
22.         }, { complete: done })
23.     }
24. }
25. })
```

效果预览 »

## 初始渲染的过渡

可以通过 `appear` 特性设置节点的在初始渲染的过渡

```
1.     <transition appear>
2.     <!-- ... -->
3.     </transition>
```

这里默认和进入和离开过渡一样，同样也可以自定义 CSS 类名。

```
1.     <transition
2.     appear
3.     appear-class="custom-appear-class"
4.     appear-active-class="custom-appear-active-class"
5.     >
6.     <!-- ... -->
7.     </transition>
```

自定义 JavaScript 钩子：

```
1.     <transition
2.     appear
3.     v-on:before-appear="customBeforeAppearHook"
4.     v-on:appear="customAppearHook"
5.     v-on:after-appear="customAfterAppearHook"
```

```
6.     >
7.     <!-- ... -->
8.     </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：

```
1.     <transition>
2.     <table v-if="items.length > 0">
3.     <!-- ... -->
4.     </table>
5.     <p v-else>Sorry, no items found.</p>
6.     </transition>
```

可以这样使用，但是有一点需要注意：

当有的元素切换时，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，`<transition>`

示例：

```
1.     <transition>
2.     <button v-if="isEditing" key="save">
3.     Save
4.     </button>
5.     <button v-else key="edit">
6.     Edit
7.     </button>
8.     </transition>
```

在一些场景中，也可以给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：

```
1.     <transition>
```

```

2.     <button v-bind:key="isEditing">
3.     {{ isEditing ? 'Save' : 'Edit' }}
4.     </button>
5.     </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：

```

1.     <transition>
2.     <button v-if="docState === 'saved'" key="saved">
3.     Edit
4.     </button>
5.     <button v-if="docState === 'edited'" key="edited">
6.     Save
7.     </button>
8.     <button v-if="docState === 'editing'" key="editing">
9.     Cancel
10.    </button>
11.    </transition>

```

可以重写为：

```

1.     <transition>
2.     <button v-bind:key="docState">
3.     {{ buttonMessage }}
4.     </button>
5.     </transition>

1.    // ...
2.    computed: {
3.      buttonMessage: function () {
4.        switch (docState) {
5.          case 'saved': return 'Edit'
6.          case 'edited': return 'Save'
7.          case 'editing': return 'Cancel'

```

```
8.         }
9.         }
10.        }
```

## 过渡模式

这里还有一个问题，试着点击下面的按钮：

[效果预览 »](#)

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：

[效果预览 »](#)

然后，我们加上 `translate` 让它们运动像滑动过渡：

[效果预览 »](#)

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

- 

`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：

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

[效果预览 »](#)

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：

[效果预览 »](#)

很酷吧？

## 多个组件的过渡

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

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

1.     new Vue({
2.       el: '#transition-components-demo',
3.       data: {
4.         view: 'v-a'
5.       },
6.       components: {
7.         'v-a': {
8.           template: '<div>Component A</div>'
9.         },
10.        'v-b': {
11.          template: '<div>Component B</div>'
12.        }
13.      }
14.    })

1.     .component-fade-enter-active, .component-fade-leave-active {
2.       transition: opacity .3s ease;
3.     }
4.     .component-fade-enter, .component-fade-leave-active {
5.       opacity: 0;
```



6.        }

[效果预览 »](#)

## 列表过渡

目前为止，关于过渡我们已经讲到：

- 单个节点
- 一次渲染多个节点

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>`。你可以通过 `tag` 特性更换为其他元素。
- 元素 指定唯一的 `key` 特性值

## 列表的进入和离开过渡

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。

```
1.     <div id="list-demo" class="demo">
2.     <button v-on:click="add">Add</button>
3.     <button v-on:click="remove">Remove</button>
4.     <transition-group name="list" tag="p">
5.     <span v-for="item in items" v-bind:key="item" class="list-item">
6.         {{ item }}
7.     </span>
8. </transition-group>
9. </div>
1. new Vue({
2.   el: '#list-demo',
3.   data: {
4.     items: [1,2,3,4,5,6,7,8,9],
5.     nextNum: 10
```

```
6.     },
7.     methods: {
8.     randomIndex: function () {
9.     return Math.floor(Math.random() * this.items.length)
10.    },
11.    add: function () {
12.    this.items.splice(this.randomIndex(), 0, this.nextNum++)
13.    },
14.    remove: function () {
15.    this.items.splice(this.randomIndex(), 1)
16.    },
17.    }
18.    })
1.    .list-item {
2.    display: inline-block;
3.    margin-right: 10px;
4.    }
5.    .list-enter-active, .list-leave-active {
6.    transition: all 1s;
7.    }
8.    .list-enter, .list-leave-active {
9.    opacity: 0;
10.    transform: translateY(30px);
11.    }
```

效果预览 »

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。

## 列表的位移过渡

`<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`, 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 :

```
1.      <script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>
2.      <div id="flip-list-demo" class="demo">
3.      <button v-on:click="shuffle">Shuffle</button>
4.      <transition-group name="flip-list" tag="ul">
5.      <li v-for="item in items" v-bind:key="item">
6.          {{ item }}
7.      </li>
8.      </transition-group>
9.      </div>
1.      new Vue({
2.          el: '#flip-list-demo',
3.          data: {
4.              items: [1,2,3,4,5,6,7,8,9]
5.          },
6.          methods: {
7.              shuffle: function () {
8.                  this.items = _.shuffle(this.items)
9.              }
10.          }
11.      })
1.      .flip-list-move {
2.          transition: transform 1s;
3.      }
```

## 效果预览 »

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。

```
1.      <script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>
2.      <div id="list-complete-demo" class="demo">
3.      <button v-on:click="shuffle">Shuffle</button>
4.      <button v-on:click="add">Add</button>
5.      <button v-on:click="remove">Remove</button>
6.      <transition-group name="list-complete" tag="p">
7.      <span
8.      v-for="item in items"
9.      v-bind:key="item"
10.     class="list-complete-item"
11.     >
12.     {{ item }}
13.   </span>
14. </transition-group>
15. </div>
1.   new Vue({
2.     el: '#list-complete-demo',
3.     data: {
4.       items: [1,2,3,4,5,6,7,8,9],
5.       nextNum: 10
6.     },
7.     methods: {
8.       randomIndex: function () {
9.         return Math.floor(Math.random() * this.items.length)
10.      },
11.      add: function () {
```

```

12.     this.items.splice(this.randomIndex(), 0, this.nextNum++)
13. },
14.     remove: function () {
15.         this.items.splice(this.randomIndex(), 1)
16.     },
17.     shuffle: function () {
18.         this.items = _.shuffle(this.items)
19.     }
20. }
21. })

1.     .list-complete-item {
2.         transition: all 1s;
3.         display: inline-block;
4.         margin-right: 10px;
5.     }
6.     .list-complete-enter, .list-complete-leave-active {
7.         opacity: 0;
8.         transform: translateY(30px);
9.     }
10.    .list-complete-leave-active {
11.        position: absolute;
12.    }

```

效果预览 »

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，多维网格的过渡也同样简单：

效果预览 »

## 列表的渐进过渡

通过 data 属性与 JavaScript 通信，就可以实现列表的渐进过渡：

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

2.     <div id="staggered-list-demo">

3.     <input v-model="query">

4.     <transition-group

5.     name="staggered-fade"

6.     tag="ul"

7.     v-bind:css="false"

8.     v-on:before-enter="beforeEnter"

9.     v-on:enter="enter"

10.    v-on:leave="leave"

11.    >

12.    <li

13.    v-for="(item, index) in computedList"

14.    v-bind:key="item.msg"

15.    v-bind:data-index="index"

16.    >{{ item.msg }}</li>

17.  </transition-group>

18. </div>

1.  new Vue({

2.    el: '#staggered-list-demo',

3.    data: {

4.      query: "",

5.      list: [

6.        { msg: 'Bruce Lee' },

7.        { msg: 'Jackie Chan' },

8.        { msg: 'Chuck Norris' },

9.        { msg: 'Jet Li' },

10.       { msg: 'Kung Fury' }
```

```
11.         ]
12.     },
13.     computed: {
14.         computedList: function () {
15.             var vm = this
16.             return this.list.filter(function (item) {
17.                 return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1
18.             })
19.         }
20.     },
21.     methods: {
22.         beforeEnter: function (el) {
23.             el.style.opacity = 0
24.             el.style.height = 0
25.         },
26.         enter: function (el, done) {
27.             var delay = el.dataset.index * 150
28.             setTimeout(function () {
29.                 Velocity(
30.                     el,
31.                     { opacity: 1, height: '1.6em' },
32.                     { complete: done }
33.                 )
34.             }, delay)
35.         },
36.         leave: function (el, done) {
37.             var delay = el.dataset.index * 150
38.             setTimeout(function () {
39.                 Velocity(
40.                     el,
```

```

41.             { opacity: 0, height: 0 },
42.             { complete: done }
43.         )
44.     }, delay)
45. }
46. }
47. })

```

效果预览 »

## 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：

```

1.   Vue.component('my-special-transition', {
2.     template: '\
3.       <transition\
4.         name="very-special-transition"\
5.         mode="out-in"\
6.         v-on:before-enter="beforeEnter"\
7.         v-on:after-enter="afterEnter"\
8.       >\
9.       <slot></slot>\
10.     </transition>\
11.   ',
12.   methods: {
13.     beforeEnter: function (el) {
14.       // ...
15.     },
16.     afterEnter: function (el) {

```



```
17.         // ...
18.     }
19. }
20. })
```

函数组件更适合完成这个任务：

```
1.     Vue.component('my-special-transition', {
2.         functional: true,
3.         render: function (createElement, context) {
4.             var data = {
5.                 props: {
6.                     name: 'very-special-transition'
7.                     mode: 'out-in'
8.                 },
9.                 on: {
10.                    beforeEnter: function (el) {
11.                        // ...
12.                    },
13.                    afterEnter: function (el) {
14.                        // ...
15.                    }
16.                }
17.            }
18.            return createElement('transition', data, context.children)
19.        }
20.    })
```

## 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。

```
1.     <transition v-bind:name="transitionName">
```

2.               <!-- ... -->

3.               </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。

所有的过渡特性都是动态绑定。它不仅是简单的特性，通过事件的钩子函数方法，可以在获取到相应上下文数据。这意味着，可以根据组件的状态通过 JavaScript 过渡设置不同的过渡效果。

1.               <script

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

2.               <div id="dynamic-fade-demo">

3.               Fade In:   <input    type="range"    v-model="fadeInDuration"    min="0"  
v-bind:max="maxFadeDuration">

4.               Fade Out:   <input    type="range"    v-model="fadeOutDuration"    min="0"  
v-bind:max="maxFadeDuration">

5.               <transition

6.               v-bind:css="false"

7.               v-on:before-enter="beforeEnter"

8.               v-on:enter="enter"

9.               v-on:leave="leave"

10.              >

11.              <p v-if="show">hello</p>

12.              </transition>

13.              <button v-on:click="stop = true">Stop it!</button>

14.              </div>

1.               new Vue({

2.               el: '#dynamic-fade-demo',

3.               data: {

4.                show: true,

5.                fadeInDuration: 1000,

6.                fadeOutDuration: 1000,

7.                maxFadeDuration: 1500,

8.                stop: false

9.               },

```
10.     mounted: function () {
11.         this.show = false
12.     },
13.     methods: {
14.         beforeEnter: function (el) {
15.             el.style.opacity = 0
16.         },
17.         enter: function (el, done) {
18.             var vm = this
19.             Velocity(el,
20.                 { opacity: 1 },
21.                 {
22.                     duration: this.fadeInDuration,
23.                     complete: function () {
24.                         done()
25.                         if (!vm.stop) vm.show = false
26.                     }
27.                 }
28.             )
29.         },
30.         leave: function (el, done) {
31.             var vm = this
32.             Velocity(el,
33.                 { opacity: 0 },
34.                 {
35.                     duration: this.fadeOutDuration,
36.                     complete: function () {
37.                         done()
38.                         vm.show = true
39.                     }

```

```
40.         }
41.     )
42. }
43. }
44. })
```

效果预览 »

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。

## 过渡状态

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 watcher

通过 `watcher` 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 `Tweenjs` 一个例子：

```
1. <script src="https://unpkg.com/tween.js@16.3.4"></script>
2. <div id="animated-number-demo">
3.   <input v-model.number="number" type="number" step="20">
4.   <p>{{ animatedNumber }}</p>
5. </div>
1. new Vue({
2.   el: '#animated-number-demo',
3.   data: {
```

```

4.         number: 0,
5.         animatedNumber: 0
6.     },
7.     watch: {
8.         number: function(newValue, oldValue) {
9.             var vm = this
10.            function animate (time) {
11.                requestAnimationFrame(animate)
12.                TWEEN.update(time)
13.            }
14.            new TWEEN.Tween({ tweeningNumber: oldValue })
15.                .easing(TWEEN.Easing.Quadratic.Out)
16.                .to({ tweeningNumber: newValue }, 500)
17.                .onUpdate(function () {
18.                    vm.animatedNumber = this.tweeningNumber.toFixed(0)
19.                })
20.                .start()
21.            animate()
22.        }
23.    }
24. })

```

效果预览 »

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 Color.js 实现一个例子：

```

1.     <script src="https://unpkg.com/tween.js@16.3.4"></script>
2.     <script src="https://unpkg.com/color.js@1.0.3/color.js"></script>
3.     <div id="example-7">
4.         <input
5.             v-model="colorQuery"
6.             v-on:keyup.enter="updateColor"

```

```
7.         placeholder="Enter a color"
8.     >
9.     <button v-on:click="updateColor">Update</button>
10.    <p>Preview:</p>
11.    <span
12.        v-bind:style="{ backgroundColor: tweenedCSSColor }"
13.        class="example-7-color-preview"
14.    ></span>
15.    <p>{{ tweenedCSSColor }}</p>
16.</div>
```

```
1. var Color = net.brehaut.Color
2. new Vue({
3.     el: '#example-7',
4.     data: {
5.         colorQuery: "",
6.         color: {
7.             red: 0,
8.             green: 0,
9.             blue: 0,
10.            alpha: 1
11.        },
12.        tweenedColor: {}
13.    },
14.    created: function () {
15.        this.tweenedColor = Object.assign({}, this.color)
16.    },
17.    watch: {
18.        color: function () {
19.            function animate (time) {
20.                requestAnimationFrame(animate)
```

```

21.         TWEEN.update(time)
22.     }
23.     new TWEEN.Tween(this.tweenedColor)
24.         .to(this.color, 750)
25.         .start()
26.         .animate()
27.     }
28. },
29. computed: {
30.     tweenedCSSColor: function () {
31.         return new Color({
32.             red: this.tweenedColor.red,
33.             green: this.tweenedColor.green,
34.             blue: this.tweenedColor.blue,
35.             alpha: this.tweenedColor.alpha
36.         }).toCSS()
37.     }
38. },
39. methods: {
40.     updateColor: function () {
41.         this.color = new Color(this.colorQuery).toRGB()
42.         this.colorQuery = ""
43.     }
44. }
45. })

1. .example-7-color-preview {
2.     display: inline-block;
3.     width: 50px;
4.     height: 50px;
5. }

```

效果预览 »

## 动态状态转换

就像 Vue 的过渡组件一样，数据背后状态转换会实时更新，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 多边形也可是实现很多难以想象的效果。

效果预览 »

## 通过组件组织过渡

管理太多的状态转换的很快会接近到 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：

```
1.      <script src="https://unpkg.com/tween.js@16.3.4"></script>
2.      <div id="example-8">
3.          <input v-model.number="firstNumber" type="number" step="20"> +
4.          <input v-model.number="secondNumber" type="number" step="20"> =
5.          {{ result }}
6.      <p>
7.          <animated-integer v-bind:value="firstNumber"></animated-integer> +
8.          <animated-integer v-bind:value="secondNumber"></animated-integer> =
9.          <animated-integer v-bind:value="result"></animated-integer>
10.     </p>
11. </div>

1.      // 这种复杂的补间动画逻辑可以被复用
2.      // 任何整数都可以执行动画
3.      // 组件化使我们的界面十分清晰
4.      // 可以支持更多更复杂的动态过渡
5.      // strategies.
6.      Vue.component('animated-integer', {
7.          template: '<span>{{ tweeningValue }}</span>',
8.          props: {
9.              value: {
10.                  type: Number,
```



```
11.         required: true
12.     }
13. },
14.     data: function () {
15.         return {
16.             tweeningValue: 0
17.         }
18.     },
19.     watch: {
20.         value: function (newValue, oldValue) {
21.             this.tween(oldValue, newValue)
22.         }
23.     },
24.     mounted: function () {
25.         this.tween(0, this.value)
26.     },
27.     methods: {
28.         tween: function (startValue, endValue) {
29.             var vm = this
30.             function animate (time) {
31.                 requestAnimationFrame(animate)
32.                 TWEEN.update(time)
33.             }
34.             new TWEEN.Tween({ tweeningValue: startValue })
35.                 .to({ tweeningValue: endValue }, 500)
36.                 .onUpdate(function () {
37.                     vm.tweeningValue = this.tweeningValue.toFixed(0)
38.                 })
39.                 .start()
40.             animate()
```

```
41.         }
42.     }
43. })
44. // All complexity has now been removed from the main Vue instance!
45. new Vue({
46.     el: '#example-8',
47.     data: {
48.         firstNumber: 20,
49.         secondNumber: 40
50.     },
51.     computed: {
52.         result: function () {
53.             return this.firstNumber + this.secondNumber
54.         }
55.     }
56. })
```

## 混合

---

## 基础

混合是一种灵活的分布式复用 **Vue** 组件的方式。混合对象可以包含任意组件选项。以组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项。

例子：

```
1. // 定义一个混合对象
2. var myMixin = {
3.     created: function () {
4.         this.hello()
5.     },
6.     methods: {
```

```

7.         hello: function () {
8.             console.log('hello from mixin!')
9.         }
10.    }
11. }
12. // 定义一个使用混合对象的组件
13. var Component = Vue.extend({
14.     mixins: [myMixin]
15. })
16. var component = new Component() // -> "hello from mixin!"

```

## 选项合并

当组件和混合对象含有同名选项时，这些选项将以恰当的方式混合。比如，同名钩子函数将混合为一个数组，因此都将被调用。另外，混合对象的钩子将在组件自身钩子调用：

```

1.     var mixin = {
2.         created: function () {
3.             console.log('mixin hook called')
4.         }
5.     }
6.     new Vue({
7.         mixins: [mixin],
8.         created: function () {
9.             console.log('component hook called')
10.        }
11.    })
12. // -> "混合对象的钩子被调用"
13. // -> "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。

```

1.     var mixin = {

```

```

2.     methods: {
3.         foo: function () {
4.             console.log('foo')
5.         },
6.         conflicting: function () {
7.             console.log('from mixin')
8.         }
9.     }
10. }
11. var vm = new Vue({
12.     mixins: [mixin],
13.     methods: {
14.         bar: function () {
15.             console.log('bar')
16.         },
17.         conflicting: function () {
18.             console.log('from self')
19.         }
20.     }
21. })
22. vm.foo() // -> "foo"
23. vm.bar() // -> "bar"
24. vm.conflicting() // -> "from self"

```

注意： `Vue.extend()` 也使用同样的策略进行合并。

## 全局混合

也可以全局注册混合对象。 注意使用！ 一旦使用全局混合对象，将会影响到 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。

```

1.     // 为自定义的选项 'myOption' 注入一个处理器。
2.     Vue.mixin({

```

```

3.         created: function () {
4.             var myOption = this.$options.myOption
5.             if (myOption) {
6.                 console.log(myOption)
7.             }
8.         }
9.     })
10.    new Vue({
11.        myOption: 'hello!'
12.    })
13.    // -> "hello!"

```

谨慎使用全局混合对象，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` 以避免产生重复应用

## 自定义选项混合策略

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混合，可以向 `Vue.config.optionMergeStrategies` 添加一个函数：

```

1.    Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
2.        // return mergedVal
3.    }

```

对于大多数对象选项，可以使用 `methods` 的合并策略：

```

1.    var strategies = Vue.config.optionMergeStrategies
2.    strategies.myOption = strategies.methods

```

更多高级的例子可以在 `Vuex 1.x` 的混合策略里找到：

```

1.    const merge = Vue.config.optionMergeStrategies.computed
2.    Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {
3.        if (!toVal) return fromVal
4.        if (!fromVal) return toVal
5.        return {

```

```
6.         getters: merge(toVal.getters, fromVal.getters),
7.         state: merge(toVal.state, fromVal.state),
8.         actions: merge(toVal.actions, fromVal.actions)
9.     }
10. }
```

---

---

## 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：

```
1.     const NotFound = { template: '<p>Page not found</p>' }
2.     const Home = { template: '<p>home page</p>' }
3.     const About = { template: '<p>about page</p>' }
4.     const routes = {
5.         '/': Home,
6.         '/about': About
7.     }
8.     new Vue({
9.         el: '#app',
10.        data: {
11.            currentRoute: window.location.pathname
12.        },
13.        computed: {
```

```
14.     ViewComponent () {  
15.         return routes[this.currentRoute] || NotFound  
16.     }  
17. },  
18.     render (h) { return h(this.ViewComponent) }  
19. })
```

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](#)。

## 深入响应式原理

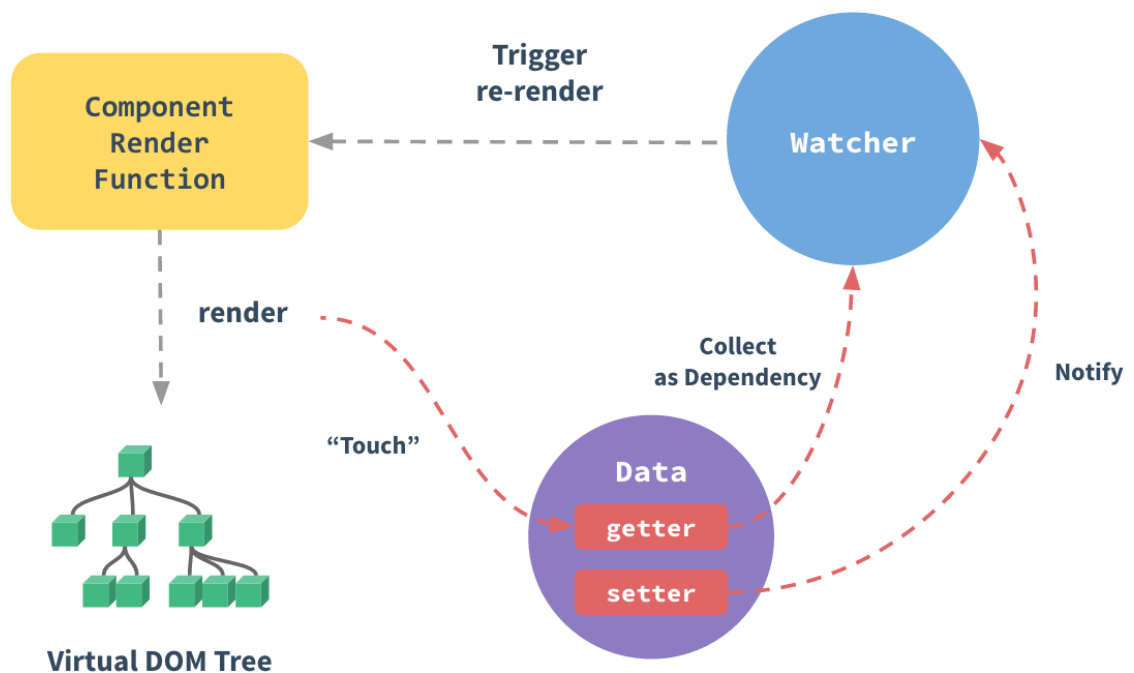
大部分的基础内容我们已经讲到了, 现在讲点底层内容。Vue 最显著的一个功能是响应系统 —— 模型只是普通对象, 修改它则更新视图。这会让状态管理变得非常简单且直观, 不过理解它的原理以避免一些常见的陷阱也是很重要的。在本节中, 我们将开始深挖 Vue 响应系统的底层细节。

## 如何追踪变化

把一个普通 Javascript 对象传给 Vue 实例来作为它的 `data` 选项, Vue 将遍历它的属性, 用 `Object.defineProperty` 将它们转为 `getter/setter`。这是 ES5 的特性, 不能打补丁实现, 这便是为什么 Vue 不支持 IE8 以及更低版本浏览器的原因。

用户看不到 `getter/setters`, 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 `程序实例`, 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。



手册网  
shouce.wen

## 变化检测问题

受现代 Javascript 的限制(以及 `Object.observe` 的废弃), Vue 在初始化实例时将属性转为 `getter/setter`, 所以属性必须在 `data` 对象上才能让 Vue 转换它, 这样才能让它是响应的。例如：

```
1. var vm = new Vue({
2.   data:{
3.     a:1
4.   }
5. })
6. // `vm.a` 是响应的
7. vm.b = 2
8. // `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 properties)。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：

```
1. 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：

```
1.      this.$set(this.someObject, 'b', 2)
```

有时你想向已有对象上添加一些属性，例如使用 `Object.assign()` 或 `_.extend()` 方法来添加属性。但是，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：

```
1.      // 代替 `Object.assign(this.someObject, { a: 1, b: 2 })`  
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。

由于 `Vue` 不允许动态添加根级响应式属性，所以你必须初始化实例前声明根级响应式属性，哪怕只是一个空值：

```
1.      var vm = new Vue({  
2.      data: {  
3.      // 声明 message 为一个空值字符串  
4.      message: "",  
5.      },  
6.      template: '<div>{{ message }}</div>'  
7.      })  
8.      // 之后设置 `message`  
9.      vm.message = 'Hello!'
```

如果你不在 `data` 对象中声明 `message`，`Vue` 将发出警告表明你的渲染方法正试图访问一个不存在的属性。

这样的限制在背后是有其技术原因的，在依赖项跟踪系统中，它消除了一类边界情况，也使 `Vue` 实例在类型检查系统的帮助下运行的更高效。在代码可维护性方面上这也是重要的一点：

`data` 对象就像组件状态的 Schema，在它上面声明所有的属性让组织代码更易于被其他开发者或是自己回头重新阅读时更加快速地理解。

## 异步更新队列

你应该注意到 `Vue` 执行 DOM 更新是，只要观察到数据变化，`Vue` 就开始一个队列，将同一事件循环内所有的数据变化缓存起来。如果一个 `watcher` 被多次触发，只会推入一次到队列中。然后，在接下来的事件循环中，`Vue` 刷新队列并仅执行必要的 DOM 更新。`Vue` 在

内部使用 `Promise.then` 和 `MutationObserver` 为可用的异步队列调用回调

`setTimeout(fn, 0)`.

例如，当你设置 `vm.someData = 'new value'`，该组件不会马上被重新渲染。当刷新队列时，这个组件会在下一次事件循环清空队列时更新。我们基本不用关心这个过程，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。一般来讲，Vue 鼓励开发者沿着数据驱动的思路，尽量避免直接接触 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)`。这样回调在 DOM 更新完成后就会调用。例如：

```
1. <div id="example">{{message}}</div>
1. var vm = new Vue({
2.   el: '#example',
3.   data: {
4.     message: '123'
5.   }
6. })
7. vm.message = 'new message' // 更改数据
8. vm.$el.textContent === 'new message' // false
9. Vue.nextTick(function () {
10.   vm.$el.textContent === 'new message' // true
11. })
```

`vm.$nextTick()` 这个实例方法在组件内使用特别方便，因为它不需要全局 `Vue`，它的回

调 `this` 将自动绑定到当前的 Vue 实例上：

```
1. Vue.component('example', {
2.   template: '<span>{{ message }}</span>',
3.   data: function () {
4.     return {
5.       message: 'not updated'
6.     }
7.   }
8. })
```

```
7.     },
8.     methods: {
9.       updateMessage: function () {
10.        this.message = 'updated'
11.        console.log(this.$el.textContent) // => 'not updated'
12.        this.$nextTick(function () {
13.          console.log(this.$el.textContent) // => 'updated'
14.        })
15.      }
16.    }
17.  })
```

## 生产环境部署

## 删除警告

为了减少文件大小，Vue 精简独立版本已经删除了所有警告，但是当你使用 Webpack 或 Browserify 等工具时，你需要一些额外的配置实现这点。

## Webpack

使用 Webpack 的 `DefinePlugin` 来指定生产环境，以便在压缩时可以让 UglifyJS 自动删除代码块内的警告语句。例如配置：

```
1.     var webpack = require('webpack')
2.     module.exports = {
3.       // ...
4.       plugins: [
5.         // ...
6.         new webpack.DefinePlugin({
7.           'process.env': {
8.             NODE_ENV: '"production"'
9.           }
10.        })
11.      ]
12.    }
```

```

10.         }},
11.         new webpack.optimize.UglifyJsPlugin({
12.             compress: {
13.                 warnings: false
14.             }
15.         })
16.     ]
17. }

```

## Browserify

- 运行打包命令，设置 `NODE_ENV` 为 `"production"`。等于告诉 `vueify` 避免引入热重载和开发相关代码。
- 使用一个全局 `envify` 转换你的 `bundle` 文件。这可以精简掉包含在 `Vue` 源码中所有环境变量条件相关代码块内的警告语句。例如：

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

- 使用 `vueify` 中包含的 `extract-css` 插件，提取样式到单独的 `css` 文件。

```
1. NODE_ENV=production browserify -g envify -p [ vueify/plugins/extract-css -o build.css ]
-e main.js | uglifyjs -c -m > build.js
```

## 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）。利用这个钩子函数和错误跟踪服务（如 `Sentry`，它为 `Vue` 提供官方集成），可能是个不错的主意。

## 提取 CSS

使用单文件组件时，`<style>` 标签在开发运行过程中会被动态实时注入。在生产环境中，你可能需要从所有组件中提取样式到单独的 `CSS` 文件中。有关如何实现的详细信息，请查阅 `vue-loader` 和 `vueify` 相应文档。

`vue-cli` 已经配置好了官方的 `webpack` 模板。

## 状态管理

---

## 类 Flux 状态管理的官方实现

由于多个状态分散的跨越在许多组件和交互间各个角落，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**，无需配置即可访问时光旅行。

## 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：

```
1.      const sourceOfTruth = {}
2.      const vmA = new Vue({
3.          data: sourceOfTruth
4.      })
5.      const vmB = new Vue({
6.          data: sourceOfTruth
7.      })
```

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。现在我们有唯一的实际来源，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：

```
1.      var store = {
2.          debug: true,
3.          state: {
4.              message: 'Hello!'
```

```

5.         },
6.         setMessageAction (newValue) {
7.             this.debug && console.log('setMessageAction triggered with', newValue)
8.             this.state.message = newValue
9.         },
10.        clearMessageAction () {
11.            this.debug && console.log('clearMessageAction triggered')
12.            this.state.message = 'action B triggered'
13.        }
14.    }

```

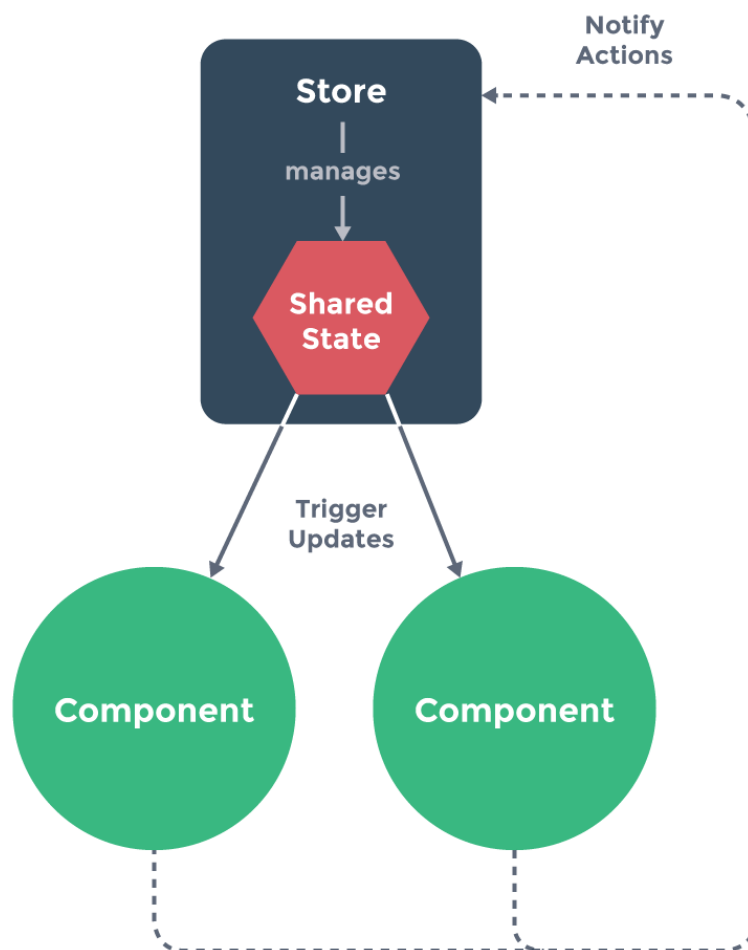
需要注意，所有 store 中 state 的改变，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的 mutation 将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

```

1.    var vmA = new Vue({
2.        data: {
3.            privateState: {},
4.            sharedState: store.state
5.        }
6.    })
7.    var vmB = new Vue({
8.        data: {
9.            privateState: {},
10.            sharedState: store.state
11.        }
12.    })

```



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。

`vuex` 给我们带来了整个循环机制, 如果你读了这么久, 不妨去尝试一下它!

## 自定义指令

## 简介

除了默认设置的核心指令(`v-model` 和 `v-show`),Vue 也允许注册自定义指令。注意,在 Vue2.0 里面,代码复用的主要形式和抽象是组件——然而,有的情况下,你仍然需要对纯 DOM 元素进行底层操作,这时候就会用到自定义指令。

下面这个例子将聚焦一个 `input` 元素,像这样:

[效果预览 »](#)

当页面加载时,元素将获得焦点。事实上,你访问后还没点击任何内容,`input` 就获得了焦点。现在让我们完善这个指令:

```
1.      // 注册一个全局自定义指令 v-focus
2.      Vue.directive('focus', {
3.          // 当绑定元素插入到 DOM 中。
4.          inserted: function (el) {
5.              // 聚焦元素
6.              el.focus()
7.          }
8.      })
```

也可以注册局部指令,组件中接受一个 `directives` 的选项:

```
1.      directives: {
2.          focus: {
3.              // 指令的定义---
4.          }
5.      }
```

然后你可以在模板中任何元素上使用新的 `v-focus` 属性:

```
1.      <input v-focus>
```

## 钩子函数

指令定义函数提供了几个钩子函数(可选):

-



**bind**: 只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。

- 
- 

**inserted**: 被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。

- 
- 

**update**: 被绑定元素所在的模板更新时调用，而不论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的模板更新（详细的钩子函数参数见下）。

- 
- 

**componentUpdated**: 被绑定元素所在模板完成一次更新周期时调用。

- 
- 

**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 (包括 **el**, **binding**, **vnode**, **oldVnode**) 。

## 钩子函数参数

钩子函数被赋予了以下参数：

- **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。

- : 指令的绑定值, 例如: `v-my-directive="1 + 1"`, value 的值是 `2`。
- : 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。
- : 绑定值的字符串形式。例如 `v-my-directive="1 + 1"`, expression 的值是 `"1 + 1"`。
- : 传给指令的参数。例如 `v-my-directive:foo`, arg 的值是 `"foo"`。
- : 一个包含修饰符的对象。例如: `v-my-directive.foo.bar`, 修饰符对象 `modifiers` 的值是 `{ foo: true, bar: true }`。
- : Vue 编译生成的虚拟节点, 查阅 VNode API 了解更多详情。
- : 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, 尽量不要修改他们。如果需要在钩子之间共享数据, 建议通过元素的 `dataset` 来进行。

一个使用了这些参数的自定义钩子样例:

```

1. <div id="hook-arguments-example" v-demo:hello.a.b="message"></div>
1. Vue.directive('demo', {
2.   bind: function (el, binding, vnode) {
3.     var s = JSON.stringify
4.     el.innerHTML =
5.       'name: ' + s(binding.name) + '<br>' +
6.       'value: ' + s(binding.value) + '<br>' +
7.       'expression: ' + s(binding.expression) + '<br>' +
8.       'argument: ' + s(binding.arg) + '<br>' +
9.       'modifiers: ' + s(binding.modifiers) + '<br>' +
10.      'vnode keys: ' + Object.keys(vnode).join(',')
11.   }

```

```
12.     })
13.     new Vue({
14.       el: '#hook-arguments-example',
15.       data: {
16.         message: 'hello!'
17.       }
18.     })
```

效果预览 »

## 函数简写

大多数情况下，我们可能想在 `bind` 和 `update` 钩子上做重复动作，并且不想关心其它的钩子函数。可以这样写：

```
1.     Vue.directive('color-swatch', function (el, binding) {
2.       el.style.backgroundColor = binding.value
3.     })
```

## 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类型的 Javascript 表达式。

```
1.     <div v-demo="{ color: 'white', text: 'hello!' }"></div>
1.     Vue.directive('demo', function (el, binding) {
2.       console.log(binding.value.color) // => "white"
3.       console.log(binding.value.text)  // => "hello!"
4.     })
```

从 Vue 1.x 迁移

## FAQ

哇，非常长的一页！是否意味着 Vue2.0 已经完全不同了呢，

是否需要从头学起呢，Vue1.0 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，

怎么开始做项目迁移？

1.

从运行 `migration helper` 这个工具开始。我们非常谨慎地把一个高级 Vue 开发工具简化并重新编译成了一个命令行工具。当这个工具发现了一个弃用的用法之后，就会给出通知和建议，并附上关于详细信息的链接。

2.

3.

然后，看看侧边栏给出的关于这一页的内容。如果你发现有的地方有影响，而该工具没有给出提示的，请检查并解决一下该项。

4.

5.

如果有测试的话，测试一边看看还有什么问题。如果没有测试的话，打开 app，随机翻一下，看一下有什么报错或者警告信息。

6.

7.

至此，你的 app 基本已经迁移完毕了。如果你有更多想了解的，可以阅读一下本节剩下的部分。

8.

从 1.0 迁移到 2.0 要花多长时间？

取决于以下几个方面：

- 

要迁移的 app 的规模。（小到中型的基本上一天内就可以搞定）

- 

- 

为了要要 Vue2.0 的新功能分心了多少次。不是说你们，我们构建 Vue2.0 的时候经常发生这种事。

- 

- 

使用了哪些弃用的功能。基本上大部分弃用的功能可以通过 **find-and-replace** 来实现升级，但有一些还是要花点时间。如果你没有遵循最佳实践，那么 Vue2.0 会强迫你去遵循。这有利于项目的长期运行，但是也意味着重构（也许有些需要重构的东西已经过时）。

- 

迁移到 Vue 2 ，我也需要更新 Vuex 和 Vue-Router ？

只有 Vue-Router 2 是可编译的，可以遵循 **Vue-Router** 迁移路径 来处理。幸运地是，大多数应用不含有许多路由代码，所以迁移不用超过一小时。

对于 Vuex ， 甚至 0.8 版本和 Vue 2 一起都是可以编译的，所以不必强制更新。促使你立即更新的理由是 **Vuex 2** 有更先进的功能，比如模块和减少的样板文件。

## 模板

## 片段实例 移除

每个组件有且仅有一个根节点。不再支持片段实例，如果你有这样的模板：

1. `<p>foo</p>`
2. `<p>bar</p>`

最好把它包裹到一个简单的容器里面去：

1. `<div>`
2. `<p>foo</p>`
3. `<p>bar</p>`
4. `</div>`

## 升级方式

升级后，为你的 app 运行端对端测试，并关注关于多个根节点的。

## 生命周期钩子

`beforeCompile` 移除

用 `created` 钩子来代替。

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

`compiled` 替换

用 `mounted` 钩子来代替。

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

`attached` 移除

依赖其它钩子使用自定义的 `dom` 内部方法，例如：

1. `attached: function () {`
2. `doSomething()`
3. `}`

现在可以这样做：

1. `mounted: function () {`
2. `this.$nextTick(function () {`
3. `doSomething()`

```
4.         })
5.     }
```

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

`detached` 移除

用自定义的 `dom` 内部的其他钩子代替，例如：

```
1.         detached: function () {
2.             doSomething()
3.         }
```

可以用以下方式代替：

```
1.         destroyed: function () {
2.             this.$nextTick(function () {
3.                 doSomething()
4.             })
5.         }
```

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

`init` 换名

用新的 `beforeCreate` 钩子代替，他们本质上是一样的。为了与其他生命周期的钩子命名保持一致性，所以重新命名了这个钩子。

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

`ready` 替换

使用新的 `mounted` 钩子代替，应该注意的是，通过使用 `mounted` 钩子，并不能保证该实例已经插入文档。所以还应该在钩子函数中包含 `Vue.nextTick/vm.$nextTick` 例如：

```
1.     mounted: function () {  
2.         this.$nextTick(function () {  
3.             // 保证 this.$el 已经插入文档  
4.         })  
5.     }
```

## 升级方式

运行 `migration helper` 工具找到所有使用这个钩子的实例

### `v-for`

## `v-for` 数组参数的顺序 改变

当含有 `index` 时，以前传递的参数顺序是 `:(index, value)`。现在变成了 `:(value, index)`，这样可以与 js 的新数组方法：`forEach`，`map` 保持一致。

## 升级方式

运行 `migration helper` 来找到使用弃用参数顺序的实例。注意，该工具将不会标记以 `position` 或者 `num` 来命名 `index` 参数。

## `v-for` 对象参数的顺序 改变

当包含 `key` 时，对象的参数顺序是 `(key, value)`。现在改为了 `(value, key)`，这样可以和通用的对象迭代器（比如 `lodash` 的迭代器）保持一致。

## 升级方式



运行 `migration helper` 来找到使用弃用参数顺序的实例。注意，该工具将不会标记以 `name`

或者 `property` 来命名 `key` 参数。

## `$index` and `$key` 移除

隐式声明的 `$index` 的 `$key` 两个变量在新版里面已经弃用了，取代的是在 `v-for` 中显式地声明。这可以使无经验的 Vue 开发者更好地理解代码，同样也可以使得在处理嵌套循环时更加清晰。

## 升级方式

运行 `migration helper` 来找到使用弃用变量的实例。如果有些没有找到，也可以参考 比如

`Uncaught ReferenceError: $index is not defined`

## `track-by` 替换

`track-by` 被 `key` 取代，和其他参数一样，如果没有 `v-bind` 或者 `:` 前缀，它将被作为一个字符串。大多数情况下， 我们想要能够动态绑定完整的表达式，而不是一个 `key`。例如：

1. `<div v-for="item in items" track-by="id">`

现在应该写成：

1. `<div v-for="item in items" v-bind:key="item.id">`

## 升级方式

运行 `migration helper` 找到含 `track-by` 的实例。

## `v-for` 排序值 改变

显然 `v-for="number in 10"` 将使得 `number` 从 0 到 9 迭代，现在变成了从 1 到 10。

## 升级方式

以正则 `\w+ in \d+` 搜索整个代码，当出现在 `v-for` 里面时，检查一下，对你是否有影响。

## 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：

```
1.      props: {
2.        username: {
3.          type: String,
4.          coerce: function (value) {
5.            return value
6.              .toLowerCase()
7.              .replace(/\s+/, '-')
8.          }
9.        }
10.     }
```

现在应该写为：

```
1.      props: {
2.        username: String,
3.      },
4.      computed: {
5.        normalizedUsername: function () {
6.          return this.username
7.            .toLowerCase()
8.            .replace(/\s+/, '-')
9.        }
10.     }
```

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。

## 升级方式

运行 `migration helper` 工具找出包含 `coerce` 选项的实例。

## `twoWay` Prop 的参数 移除

`Props` 现在只能单项传递。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

## 升级方式

运行 `migration helper` 工具，找出含有 `twoWay` 参数的实例。

## `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

## 升级方式

运行 `migration helper` 工具找到使用 `.once` 和 `.sync` 修饰符的实例。

## 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` 值可以用以下选项代替：

- 通过 `data` 属性，用 `prop` 去设置一个 `data` 属性的默认值。
- 通过 `computed` 属性。

## 升级方式

运行端对端测试，查看关于 `prop` 修改的。

## 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`。

## 升级方式

运行端对端测试，将会弹出 来通知你使用 `props` 的根实例已经失效。

## Built-In 指令

### `v-bind` 真/假值 改变

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 (如 `v-if` 和 `v-show`)，他们依然遵循 js 对真假值判断的一般规则。

## 升级方式

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

## 用 `v-on` 监听原生事件 改变

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：

1. `<my-component v-on:click.native="doSomething"></my-component>`

## 升级方式

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

## 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：

[效果预览 »](#)

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：

1. `<!--`
2. 通过使用 `lodash` 或其它库的 `debounce` 函数，
3. 我们相信 `debounce` 实现是一流的，

4. 并且可以随处使用它，不仅仅是在模板中。

5. -->

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

7. `<div id="debounce-search-demo">`

8.     `<input v-model="searchQuery" placeholder="Type something">`

9.     `<strong>{{ searchIndicator }}</strong>`

10. `</div>`

```
1. new Vue({
2.   el: '#debounce-search-demo',
3.   data: {
4.     searchQuery: '',
5.     searchQueryIsDirty: false,
6.     isCalculating: false
7.   },
8.   computed: {
9.     searchIndicator: function () {
10.      if (this.isCalculating) {
11.        return '🔄 Fetching new results'
12.      } else if (this.searchQueryIsDirty) {
13.        return '... Typing'
14.      } else {
15.        return '👌 Done'
16.      }
17.    }
18.  },
19.  watch: {
20.    searchQuery: function () {
21.      this.searchQueryIsDirty = true
22.      this.expensiveOperation()
23.    }
24.  }
25.})
```

```

24.     },
25.     methods: {
26.         // 这是 debounce 实现的地方。
27.         expensiveOperation: _.debounce(function () {
28.             this.isCalculating = true
29.             setTimeout(function () {
30.                 this.isCalculating = false
31.                 this.searchQueryIsDirty = false
32.             }.bind(this), 1000)
33.         }, 500)
34.     }
35. })

```

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 `throttle` 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。

## Upgrade Path

运行 `migration helper` 工具找出使用 `debounce` 参数的实例。

## 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：

1. `<input v-model="name" lazy>`
2. `<input v-model="age" type="number" number>`

现在写成这样：

1. `<input v-model.lazy="name">`
2. `<input v-model.number="age" type="number">`

## 升级方式

运行 `migration helper` 工具找到这些弃用参数。

## 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：

1. `<input v-model="text" value="foo">`

在 `data` 选项中有下面写法的：

1. `data: {`
2. `text: 'bar'`
3. `}`

将渲染 `model` 为 `'bar'` 而不是 `'foo'`。同样，对 `<textarea>` 已有的值来说：

1. `<textarea v-model="text">`
2. `hello world`
3. `</textarea>`

必须保证 `text` 初值为 `"hello world"`

## 升级方式

升级后运行端对端测试，注意关于 `v-model` 内联参数的

## `v-model` with `v-for` Iterated Primitive Values 移除

像这样的写法将失效：

1. `<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：

1. `strings.map(function (str) {`
2. `return createElement('input', ...)`
3. `})`



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:

1. `<input v-for="obj in objects" v-model="obj.str">`

## 升级方式

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提示。

## 带有 `!important` 的 `v-bind:style` 移除

这样写将失效:

1. `<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:

1. `<p v-bind:style="color: ' + myColor + ' !important'">hello</p>`

## 升级方式

运行 [迁移帮助工具](#)。找到含有 `!important` 的 style 绑定对象。

## `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:

1. `<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` :

1. `<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。

## 升级方式

运行 `migration helper` 找出实例中的 `v-el` 和 `v-ref` 。

## `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` 来代替。例如 :

1. `<p v-if="foo">Foo</p>`
2. `<p v-else v-show="bar">Not foo, but bar</p>`

现在应该写出这样 :

1. `<p v-if="foo">Foo</p>`
2. `<p v-if="!foo && bar">Not foo, but bar</p>`

## 升级方式

运行 `migration helper` 找出实例中存在的 `v-else` 以及 `v-show` 。

## 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，最好是把模板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。

### 升级方式

运行 `migration helper` 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。

## 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：

1. `<p v-my-directive.literal="foo bar baz"></p>`

只是：

1. `<p v-my-directive="foo bar baz"></p>`

### 升级方式

运行 `migration helper` 找到实例中使用 ``.literal`` 修饰符的地方。

# 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](#)。

## 升级方式

运行 `migration helper` 找到使用 `transition` 属性的地方。

## 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。

## 升级方式

运行 `migration helper` 工具找到使用 `transition` 属性的地方。

## 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。

## 升级方式

运行 `migration helper` 找到使用 `transition` 属性的地方。升级期间，你可以“过渡”到新的过渡策略。

# 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：

1. `// v-on:keyup.f1 不可用`
2. `Vue.config.keyCodes.f1 = 112`

## 升级方式

运行 `migration helper` 找到过时的 `keyCode` 配置

## `$dispatch` 和 `$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` `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：

1. `Todos`
2.     |-- `NewTodoInput`
3.     |-- `Todo`
4.         |-- `DeleteTodoButton`

可以通过单独的事件中心管理组件间的通信：

1.     // 将在各处使用该事件中心
2.     // 组件通过它来通信
3.     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：

1.     // `NewTodoInput`
2.     // ...
3.     methods: {
4.         addTodo: function () {
5.             eventHub.\$emit('add-todo', { text: this.newTodoText })
6.             this.newTodoText = ''
7.         }
8.     }
1.     // `DeleteTodoButton`
2.     // ...
3.     methods: {
4.         deleteTodo: function (id) {
5.             eventHub.\$emit('delete-todo', id)
6.         }
7.     }
1.     // `Todos`

```

2.      // ...
3.      created: function () {
4.          eventHub.$on('add-todo', this.addTo)
5.          eventHub.$on('delete-todo', this.deleteTodo)
6.      },
7.      // 最好在组件销毁前
8.      // 清除事件监听
9.      beforeDestroy: function () {
10.          eventHub.$off('add-todo', this.addTo)
11.          eventHub.$off('delete-todo', this.deleteTodo)
12.      },
13.      methods: {
14.          addTo: function (newTodo) {
15.              this.todos.push(newTodo)
16.          },
17.          deleteTodo: function (todoId) {
18.              this.todos = this.todos.filter(function (todo) {
19.                  return todo.id !== todoId
20.              })
21.          }
22.      }

```

在简单的情况下这样做可以代替 `$dispatch` 和 `$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](#)。

## 升级方式

运行 `migration helper` 工具找出使用 `$dispatch` 和 `$broadcast` 的实例。

## 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

```
1. <input v-on:keyup="doStuff | debounce 500">
1. methods: {
2.   doStuff: function () {
3.     // ...
4.   }
5. }
```

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:

```
1. <input v-on:keyup="doStuff">
1. methods: {
2.   doStuff: _.debounce(function () {
3.     // ...
4.   }, 500)
5. }
```

这种写法的更多优点详见: `v-model` 示例.

### 替换 `limitBy` 过滤器

不再这样写:

```
1. <p v-for="item in items | limitBy 10">{{ item }}</p>
```

在 `computed` 属性中使用 `js` 内置方法: `.slice` `method`:

```
1. <p v-for="item in filteredItems">{{ item }}</p>
```



```
1.     computed: {
2.         filteredItems: function () {
3.             return this.items.slice(0, 10)
4.         }
5.     }
```

## 替换 `filterBy` 过滤器

不再这样写：

```
1.     <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：

```
1.     <p v-for="user in filteredUsers">{{ user.name }}</p>
1.     computed: {
2.         filteredUsers: function () {
3.             var self = this
4.             return self.users.filter(function (user) {
5.                 return user.name.indexOf(self.searchQuery) !== -1
6.             })
7.         }
8.     }
```

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：

```
1.     var self = this
2.     self.users.filter(function (user) {
3.         var searchRegex = new RegExp(self.searchQuery, 'i')
4.         return user.isActive && (
5.             searchRegex.test(user.name) ||
6.             searchRegex.test(user.email)
7.         )
8.     })
```

## 替换 `orderBy` 过滤器

不这样写：

```
1. <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`)：

```
1. <p v-for="user in orderedUsers">{{ user.name }}</p>
```

```
1. computed: {  
2.   orderedUsers: function () {  
3.     return _orderBy(this.users, 'name')  
4.   }  
5. }
```

甚至可以字段排序：

```
1. _orderBy(this.users, ['name', 'last_login'], ['asc', 'desc'])
```

## 升级方式

运行 `migration helper` 工具找到指令中使用的过滤器。如果有些没找到，看看。

## 过滤器参数符号 改变

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：

```
1. <p>{{ date | formatDate 'YY-MM-DD' timeZone }}</p>
```

现在用圆括号括起来并用逗号分隔：

```
1. <p>{{ date | formatDate('YY-MM-DD', timeZone) }}</p>
```

## 升级方式

运行 `migration helper` 工具找到老式的调用符号，如果有遗漏，请看。

## 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

```
1.      text[0].toUpperCase() + text.slice(1)
```

## 替换 `uppercase` 过滤器

```
1.      text.toUpperCase()
```

## 替换 `lowercase` 过滤器

```
1.      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：

```
1.      function pluralizeKnife (count) {
2.          if (count === 0) {
3.              return 'no knives'
4.          } else if (count === 1) {
5.              return '1 knife'
6.          } else {
7.              return count + 'knives'
8.          }
9.      }
```

## Replacing the `currency` Filter

对于简单的问题,可以这样做：

1. `'$' + 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`)。解决这个问题可以使用 `accounting` 库来实现更多可靠的货币格式化。

## 升级方式

运行 `migration helper` 工具找到舍弃的过滤器。如果有些遗漏，请参考。

## Two-Way Filters replaced

Some users have enjoyed using two-way filters with `v-model` to create interesting inputs with very little code. While *seemingly* simple however, two-way filters can also hide a great deal of complexity - and even encourage poor UX by delaying state updates. Instead, components wrapping an input are recommended as a more explicit and feature-rich way of creating custom inputs.

As an example, we'll now walk the migration of a two-way currency filter:

It mostly works well, but the delayed state updates can cause strange behavior. For example, click on the `Result` tab and try entering `9.999` into one of those inputs. When the input loses focus, its value will update to `$10.00`. When looking at the calculated total however, you'll see that `9.999` is what's stored in our data. The version of reality that the user sees is out of sync!

To start transitioning towards a more robust solution using Vue 2.0, let's first wrap this filter in a new `<currency-input>` component:

This allows us add behavior that a filter alone couldn't encapsulate, such as selecting the content of an input on focus. Now the next step will be to extract the business logic from the filter. Below, we pull everything out into an external `currencyValidator` object:

This increased modularity not only makes it easier to migrate to Vue 2, but also allows currency parsing and formatting to be:

- unit tested in isolation from your Vue code

- used by other parts of your application, such as to validate the payload to an API endpoint

Having this validator extracted out, we've also more comfortably built it up into a more robust solution. The state quirks have been eliminated and it's actually impossible for users to enter anything wrong, similar to what the browser's native number input tries to do.

We're still limited however, by filters and by Vue 1.0 in general, so let's complete the upgrade to Vue 2.0:

You may notice that:

- Every aspect of our input is more explicit, using lifecycle hooks and DOM events in place of the hidden behavior of two-way filters.
- We can now use `v-model` directly on our custom inputs, which is not only more consistent with normal inputs, but also means our component is Vuex-friendly.
- Since we're no longer using filter options that require a value to be returned, our currency work could actually be done asynchronously. That means if we had a lot of apps that had to work with currencies, we could easily refactor this logic into a shared microservice.

## Upgrade Path

Run the `migration helper` on your codebase to find examples of filters used in directives like `v-model`. If you miss any, you should also see .

## Slots

### 重名的 Slots 移除

同一模板中的重名 `<slot>` 已经弃用。当一个 `slot` 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，可一用 `prop` 来传递。

### 升级方式

更新后运行测试，查看 关于重名 slots 的提示 `v-model`。

## 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。

### 升级方式

运行 `migration helper` 找到选择 slots 标签 CSS 选择器(例如：`[slot="my-slot-name"]`)。

## 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：

1. `<keep-alive>`
2. `<component v-bind:is="view"></component>`
3. `</keep-alive>`

这样可以在含多种状态子组件中使用 `<keep-alive>`：

1. `<keep-alive>`
2. `<todo-list v-if="todos.length > 0"></todo-list>`
3. `<no-todos-gif v-else></no-todos-gif>`
4. `</keep-alive>`

当 `<keep-alive>` 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：

1. `<transition>`

2. `<keep-alive>`
3. `<component v-bind:is="view"></component>`
4. `</keep-alive>`
5. `</transition>`

## 升级方式

运行 `migration helper` 工具找到 `keep-alive` 属性。

## 计算插值 todaymark

### 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：

1. `<button class="btn btn-{{ size }}"></button>`

应该写成行内表达式：

1. `<button v-bind:class="'btn btn-' + size"></button>`

或者计算属性：

1. `<button v-bind:class="buttonClasses"></button>`
1. `computed: {`
2. `buttonClasses: function () {`
3. `return 'btn btn-' + size`
4. `}`
5. `}`

## 升级方式

运行 `migration helper` 找到属性内部的计算插值

### HTML 计算插值 移除

HTML 的计算插值 (`{{{ foo }}})` 已经弃用，取代的是 `v-html` 指令。

## 升级方式

运行 `migration helper` 找到 HTML 计算插值。

## 单次绑定<sup>替换</sup>

单次绑定 (`{{* foo }}`) 已经弃用取代的是 `v-once directive`。

## 升级方式

运行 `migration helper` 工具找到单次绑定使用位置。

## 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。

## 升级方式

运行测试，如果有依赖于老方法的观察器将弹出。

`vm.$set` 改变

`vm.$set` 只是 `Vue.set` 的别名。

## 升级方式



运行 `migration helper` 工具找到过时的用法

`vm.$delete` 改变

`vm.$delete` 现在只是： `Vue.delete` 别名。

## 升级方式

运行 `migration helper` 工具找到过时的用法

`Array.prototype.$set` 弃用

用 `Vue.set` 代替

## 升级方式

运行 `migration helper` 工具找到数组上的 `.$set`。如有遗漏请参考。

`Array.prototype.$remove` 移除

用 `Array.prototype.splice` 代替，例如：

```
1.      methods: {  
2.          removeTodo: function (todo) {  
3.              var index = this.todos.indexOf(todo)  
4.              this.todos.splice(index, 1)  
5.          }  
6.      }
```

或者更好的方法，直接给除去的方法一个 `index` 参数：

```
1.      methods: {  
2.          removeTodo: function (index) {  
3.              this.todos.splice(index, 1)  
4.          }  
5.      }
```

4.        }
5.        }

## 升级方式

运行 `migration helper` 工具找到数组上的 `.$remove`。如有遗漏请参考

## 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。

## 升级方式

运行 `migration helper` 找到实例中的 `Vue.set` 或 `Vue.delete`。如有遗漏请参考。

## 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）。

## 升级方式

运行 `migration helper` 工具找到覆盖 `vm.$data` 的位置。如有遗漏请参考。

`vm.$get` 移除

可以直接取回响应数据。

## 升级方式

运行 `migration helper` 工具找到 `vm.$get` 的位置。如有遗漏请参考 。

## 围绕 DOM 的实例方法

`vm.$appendTo` 移除

使用 DOM 原生方法:

1. `myElement.appendChild(vm.$el)`

### 升级方式

运行 `migration helper` 工具找到 `vm.$appendTo` 的位置。如果有遗漏可以参考。

`vm.$before` 移除

使用 DOM 原生方法：

1. `myElement.parentNode.insertBefore(vm.$el, myElement)`

### 升级方式

运行 `migration helper` 工具找到 `vm.$before`。如有遗漏，请参考 。

`vm.$after` 移除

使用 DOM 原生方法：

1. `myElement.parentNode.insertBefore(vm.$el, myElement.nextSibling)`

如果 `myElement` 是最后一个节点也可以这样写：

1. `myElement.parentNode.appendChild(vm.$el)`

### 升级方式

运行 `migration helper` 找到 `vm.$after` 的位置。如有遗漏，请参考。

`vm.$remove` 移除

使用 DOM 原生方法：

1. `vm.$el.remove()`

## 升级方式

运行 `migration helper` 找到 `vm.$remove`。如有遗漏，请参考。

## 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](#)。

## 升级方式

运行 `migration helper` 工具找到使用 `vm.$eval` 的位置。如有遗漏请参考。

`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](#)。

## 升级方式

运行 `migration helper` 找到 `vm.$interpolate`。如有遗漏请参考。

`vm.$log` 移除

请使用 [Vue Devtools](#) 感受最佳 debug 体验。

## 升级方式

运行 [migration helper](#) 找到 `vm.$log`。如有遗漏请参考。

## 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：

```
1.     new Vue({
2.       el: '#app',
3.       template: '<div id="app"> ... </div>'
4.     })
```

或者使用渲染函数：

```
1.     new Vue({
2.       el: '#app',
3.       render: function (h) {
4.         h('div', {
5.           attrs: {
6.             id: 'app',
7.           }
8.         }, /* ... */)
9.       }
10.    })
```

## 升级方式

运行 `migration helper` 工具找到 `replace: false` 使用的位置。

## 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。

## 升级方式

运行 `migration helper` 找到包含 `Vue.config.debug` 的地方。

`Vue.config.async` 移除

异步操作现在需要渲染性能的支持。

## 升级方式

运行 `migration helper` 工具找到使用 `Vue.config.async` 的实例。

`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。

## 升级方式

运行 `migration helper` 工具找到使用 `Vue.config.delimiters` 的实例。

`Vue.config.unsafeDelimiters` 移除

HTML 插入 替换为 `v-html`.

## 升级方式

运行 `migration helper` 工具找到 `Vue.config.unsafeDelimiters`。然后 `helper` 工具也会找到 HTML 插入的实例，可以用 ``v-html`` 来替换。

## 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。

## 升级方式

更新后运行测试在中找到关于带有 `Vue.extend` 的 `el`。

### `Vue.elementDirective` 移除

用组件来代替

## 升级方式

运行 `migration helper` 工具找到包含 `Vue.elementDirective` 的实例。

### `Vue.partial` 移除

Partials have been removed in favor of more explicit data flow between components, using props. Unless you're using a partial in a performance-critical area, the recommendation is to simply use a normal component instead. If you were dynamically binding the `name` of a partial, you can use a dynamic component.

If you happen to be using partials in a performance-critical part of your app, then you should upgrade to functional components. They must be in a plain JS/JSX file (rather than in a `.vue` file) and are stateless and instanceless, just like partials. This makes rendering extremely fast.

A benefit of functional components over partials is that they can be much more dynamic, because they grant you access to the full power of JavaScript. There is a cost to this power however. If you've never used a component framework with render functions before, they may take a bit longer to learn.

## 升级方式

运行 `migration helper` 工具找到包含 `Vue.partial` 的实例

[关于我们](#) [联系我们](#) [留言板](#)

[手册网](#)

## 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：

1. `router.start({`
2. `template: '<router-view></router-view>'`
3. `}, '#app')`

你只需要传一个路由属性给 Vue 实例：



```
1.     new Vue({
2.       el: '#app',
3.       router: router,
4.       template: '<router-view></router-view>'
5.     })
```

或者，如果你使用的是运行时构建 (runtime-only) 方式：

```
1.     new Vue({
2.       el: '#app',
3.       router: router,
4.       render: h => h('router-view')
5.     })
```

## 升级路径

运行 [迁移助手](#) 找到 `router.start` 被调用的示例。

## 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：

```
1.     router.map({
2.       '/foo': {
3.         component: Foo
4.       },
5.       '/bar': {
6.         component: Bar
7.       }
8.     })
```

会以这种方式定义：

```
1.     var router = new VueRouter({
2.       routes: [
```

```

3.         { path: '/foo', component: Foo },
4.         { path: '/bar', component: Bar }
5.     ]
6. })

```

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。

## 升级路径

运行 [迁移助手](#) 找到 `router.map` 被调用的示例。

`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：

```

1.     // 普通的路由
2.     var routes = [
3.         // ...
4.     ]
5.     // 动态生成的路由
6.     marketingPages.forEach(function (page) {
7.         routes.push({
8.             path: '/marketing/' + page.slug
9.             component: {
10.                 extends: MarketingComponent
11.                 data: function () {
12.                     return { page: page }
13.                 }
14.             }
15.         })
16.     })
17.     var router = new Router({
18.         routes: routes

```

19.            })

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：

```
1.            router.match = createMatcher(  
2.            [{  
3.                path: '/my/new/path',  
4.                component: MyComponent  
5.            }].concat(router.options.routes)  
6.            )
```

## 升级路径

运行 [迁移助手](#) 找到 `router.on` 被调用的示例。

`subRoutes` 换名

出于 Vue Router 和其他路由库一致性的考虑，重命名为 `children`

## 升级路径

运行 [迁移助手](#) 找到 `subRoutes` 选项的示例。

`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：

```
1.            router.redirect({  
2.                '/tos': '/terms-of-service'  
3.            })
```

成像下面的 `routes` 配置里定义的样子：

```
1.            {  
2.                path: '/tos',  
3.                redirect: '/terms-of-service'  
4.            }
```

## 升级路径

运行 [迁移助手](#) 找到 `router.redirect` 被调用的示例。

### `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：

```
1. router.alias({
2.   '/manage': '/admin'
3. })
```

配置这个样子：

```
1. {
2.   path: '/admin',
3.   component: AdminPanel,
4.   alias: '/manage'
5. }
```

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：

```
1. alias: ['/manage', '/administer', '/administrate']
```

## 升级路径

运行[迁移助手](#)找到 `router.alias` 被调用的示例。

## 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：

```
1. '/admin': {
2.   component: AdminPanel,
3.   requiresAuth: true
4. }
```

你现在需要把它更新成：

```

1.      {
2.        path: '/admin',
3.        component: AdminPanel,
4.        meta: {
5.          requiresAuth: true
6.        }
7.      }

```

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：

```

1.      if (route.meta.requiresAuth) {
2.        // ...
3.      }

```

## 升级路径

运行 [迁移助手](#) 找到任意的路由不在 `meta` 作用域下的示例。

## 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+`。

## 升级路径

运行 [迁移助手](#) 找到弃用路由语法的示例。

## 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：

```

1.      <a v-link="/about">About</a>

```

你需要把它更新成：

1. `<router-link to="/about">About</router-link>`

Note that `target="_blank"` is not supported on `<router-link>`, so if you need to open a link in a new tab, you have to use `<a>` instead.

## 升级路径

运行 迁移助手 找到 `v-link` 指令的示例。

### `v-link-active` 替换

The `v-link-active` directive has also been replaced by the `tag` attribute on the `<router-link>` component. So for example, you'll update this:

`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：

1. `<li v-link-active>`
2. `<a v-link="/about">About</a>`
3. `</li>`

成这个写法:

1. `<router-link tag="li" to="/about">`
2. `<a>About</a>`
3. `</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。

## 升级路径

运行 迁移助手 找到 `v-link-active` 指令的示例

## 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。

## 升级路径

运行 [迁移助手](#)，找到 `router.go` 和 `router.push` 指令被调用的示例。

## 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。

## 升级路径

运行 [迁移助手](#) 找到 `hashbang: false` 选项的示。

`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：

```
1.      var router = new VueRouter({  
2.          history: 'true'  
3.      })
```

成这个写法：

```
1.      var router = new VueRouter({  
2.          mode: 'history'  
3.      })
```

## 升级路径

运行 [迁移助手](#) 找到 `history: true` 选项的示。

`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：

```
1.      var router = new VueRouter({
```

2.           abstract: 'true'
3.           })

成这个写法：

1.           var router = new VueRouter({
2.           mode: 'abstract'
3.           })

## 升级路径

运行 [迁移助手](#) 找到 `abstract: true` 选项的示例。

## 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：

1.           saveScrollPosition: true
- 你可以替换为：
1.           scrollBehavior: function (to, from, savedPosition) {
  2.           return savedPosition || { x: 0, y: 0 }
  3.           }

## 升级路径

运行 [迁移路径](#) 找到 `saveScrollPosition: true` 选项的示例。

### `root` 换名

为了与 HTML 的 `<base>` 标签保持一致性，重命名为 `base`。

## 升级路径

运行 [迁移路径](#) 找到 `root` 选项的示例。



`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。

## 升级路径

运行 [迁移路径](#) 找到 `transitionOnLoad: true` 选项的示例。

`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。

## 升级路径

运行 [迁移指令](#) 找到 `suppressTransitionError: true` 选项的示例。

## 路由挂钩

`activate` 替换

使用 `beforeRouteEnter` 这一组件进行替代。

## 升级路径

运行 [迁移路径](#) 找到 `beforeRouteEnter` 钩子的示例。

`canActivate` 替换

使用 `beforeEnter` 在路由中作为替代。

## 升级路径

运行 [迁移路径](#) 找到 `canActivate` 钩子的示例。

`deactivate` 移除

使用 `beforeDestroy` 或者 `destroyed` 钩子作为替代。

## 升级路径

运行 [迁移路径](#) 找到 `deactivate` 钩子的示例。

`canDeactivate` 替换

在组件中使用 `beforeRouteLeave` 作为替代。

## 升级路径

运行 [迁移路径](#) 找到 `canDeactivate` 钩子的示例。

`canReuse: false` 移除

在新的 Vue 路由中将不再被使用。

## 升级路径

运行 [迁移助手](#) 找到 `canReuse: false` 选项的示例。

`data` 替换

`$route` 属性是响应式的，所有你可以就使用一个 `watcher` 去响应路由的改变，就像这样：

```
1.     watch: {  
2.         '$route': 'fetchData'  
3.     },  
4.     methods: {  
5.         fetchData: function () {  
6.             // ...  
7.         }  
8.     }
```

## 升级路径

运行 [迁移助手](#) 找到 `data` 钩子的示例。

## `$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：

```
1.      data: function () {
2.          return {
3.              posts: [],
4.              isLoading: false,
5.              fetchError: null
6.          }
7.      },
8.      watch: {
9.          '$route': function () {
10.              var self = this
11.              self.isLoading = true
12.              self.fetchData().then(function () {
13.                  self.isLoading = false
14.              })
15.          }
16.      },
17.      methods: {
18.          fetchData: function () {
19.              var self = this
20.              return axios.get('/api/posts')
21.                  .then(function (response) {
22.                      self.posts = response.data.posts
23.                  })
24.                  .catch(function (error) {
25.                      self.fetchError = error
26.                  })
27.          }
28.      }
29.    }
30.  },
31.  components: {}
32.}
```

27.        }

28.        }

## 从 Vuex 0.6.x 迁移到 1.0

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。

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。，因此如果你想了解更多，请查阅 [Vuex 2.0 文档](#)。

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

传入字符串属性路径的 `store.watch` 废弃

`store.watch` 现在只接受函数。因此，下面例子你需要替换：

```
1.       store.watch('user.notifications', callback)
```

为：

```
1.       store.watch(  
2.        // 当返回结果改变...  
3.       function (state) {  
4.        return state.user.notifications  
5.       },
```

```
6.      // 执行回调函数
7.      callback
8.    )
```

这帮助你更加完善的控制那些需要监听的响应式属性。

## 升级方法

在代码库运行[迁移工具](#)，查找在 `store.watch` 中使用字符串作为第一个参数的事例。

## 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：

```
1.    var myPlugin = store => {
2.      store.subscribe(function (mutation, state) {
3.        // Do something...
4.      })
5.    }
```

更多信息请查阅[插件文档](#)的示例。

## 升级方式

在代码库运行[迁移工具](#)，查找使用了 `store.on`, `store.off`, `store.emit` 的事例。

## 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：

```
1.    const myPlugins = store => {
2.      store.subscribe('mutation', (mutation, state) => {
3.        // Do something...
```

4.                ))

5.                }

更多详情, 请查阅 [插件文档](#)。

## 升级方法

在代码库运行[迁移工具](#), 查找使用了 `middlewares` 选项的事例。

**End!**