VUE官网笔记整理

# 基础

## 安装

兼容：VUE不兼容IE8以下的浏览器

插件：VueDevtools

CND方式:<scriptsrc="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script> //通过script直接引入

NPM方式：npm install vue //最新稳定版

CLI（命令行工具）：npmivue-cli

## 介绍

Script引入：<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>//此为开发版本

<script src="https://cdn.jsdelivr.net/npm/vue"></script>//此为生产版本

## 声明式渲染

<div id="app">

<p>{{ message }}</p>//直接用模板渲染数据

<input type="text" name="" id="" value="" v-model="message"/>//用model双向数据绑定

<p v-bind:title="message2">悬停查看提示信息</p>//bind数据绑定属性

<p v-if="seen">you can see me</p> //判断是否显示

<ul>

<li v-for="item in list">{{item.text}}</li>//for循环数据

<mycomponentv-for="item in list" v-bind:data='item' v-bind:key="item.id"></mycomponet> //vue组件

</ul>

<button v-on:click="reset">reset message</button> //按钮事件

</div>

Vue.component('mycomponent',{

props: ['data'],

template: '<li>{{data.text}}</li>'

}) //设置自定义组件

varapp = new Vue({

el: '#app',

data: {

message: 'Hello Vue!', //定义数据

message2: '页面加载于 '+ new Date().toLocaleString(),

seen: true,

list: [

{text: 'asd',id: '01'},

{text: 'bdf',id: '02'},

{text: 'dagad',id: '03'}

]

},

methods: {//挂载

reset: function(){

var\_self =this;

\_self.message= \_self.message.split('').reverse().join('')

}

}

})

## 模板语法

### 缩写

v-bind:attr='' === :attr=''

v-on:click='' === @click=''

## 计算属性和侦听器

### 计算属性

<p>Original message: "{{ reversedMessage }}"</p>

computed: {  
reversedMessage: function () {  
returnthis.message.split('').reverse().join('')  
 }  
 }

当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新

#### 计算属性缓存

计算属性是基于它们的依赖进行缓存的。只在相关依赖发生改变时它们才会重新求值。例如

computed: {  
 now: function () {  
returnDate.now()  
 }  
}

Date.now() 不是响应式依赖，所以重复使用now获取到的都是同一个时间，相比之下，每当触发重新渲染时，调用方法将**总会**再次执行函数。

#### 计算属性与侦听器

watch: {  
firstName: function (val) {  
this.fullName = val + ' ' + this.lastName  
 },  
lastName: function (val) {  
this.fullName = this.firstName + ' ' + val  
 }  
 }

computed: {  
fullName: function () {  
returnthis.firstName + ' ' + this.lastName  
 }  
 }

当你有一些数据需要随着其它数据变动而变动时，更好的做法是使用计算属性而不是命令式的 watch 回调。

### 侦听器

需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的

<inputv-model="question">  
watch: {  
 // 如果 `question` 发生改变，这个函数就会运行  
 question: function (newQuestion, oldQuestion) {  
this.answer = 'Waiting for you to stop typing...'  
this.debouncedGetAnswer()  
 }  
 }

## Class与style绑定

### 绑定HTML Class

#### 对象语法

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

active 这个 class 存在与否将取决于数据属性 isActive 的 [**truthiness**](https://developer.mozilla.org/zh-CN/docs/Glossary/Truthy)

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

Class与:class**共存**，切:class可以存在多个变量

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

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

也可不必写在**内联**里

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

也可以是**计算属性**

#### 数组语法

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

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

可以是三元

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

数组中写三元语法

#### 组件上使用

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

<my-componentclass="baz boo"></my-component>

直接添加，也可以使用对象或数组语法

### 绑定内联样式

#### 对象语法

CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用单引号括起来) 来命名：

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

直接绑定对象

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

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

#### 数组语法

可以将多个样式绑定到元素上

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

#### 自动添加前缀

当 v-bind:style 使用需要添加[**浏览器引擎前缀**](https://developer.mozilla.org/zh-CN/docs/Glossary/Vendor_Prefix)的 CSS 属性时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

#### 多重值

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

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

只会渲染数组中最后一个被浏览器支持的值

## 条件渲染

### v-if

<h1v-if="ok">Yes</h1>

<h1v-else>No</h1>

#### Template元素上使用

切换多个元素时，可以把一个 <template> 元素当做不可见的包裹元素，并在上面使用 v-if。最终的渲染结果将不包含 <template> 元素。

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

#### v-else

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

#### v-else-if

版本：**2.1.0 新增**

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

#### 用key管理可复用的元素

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

<templatev-if="loginType === 'username'">  
<label>Username</label>  
<inputplaceholder="Enter your username">  
</template>  
<templatev-else>  
<label>Email</label>  
<inputplaceholder="Enter your email address">  
</template>

再重复的input上增加key，切换时输入框都将被重新渲染

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

### v-show

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

<h1v-show="ok">Hello!</h1>

### v-if与v-show

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块

v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好

### v-if与v-for

**不推荐**同时使用 v-if 和 v-for

## 列表渲染

### 遍历数组

items 是源数据数组并且 item 是数组元素迭代的别名

<li v-for="item in items">  
 {{ item.message }}  
</li>

data: {  
 items: [  
 { message: 'Foo' },  
 { message: 'Bar' }  
 ]  
 }

v-for 还支持一个可选的第二个参数为当前项的索引

<liv-for="(item, index) in items">  
 {{ parentMessage }} - {{ index }} - {{ item.message }}  
</li>

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

<divv-for="item of items"></div>

### 遍历对象

<liv-for="value in object">  
 {{ value }}  
</li>

object: {  
firstName: 'John',  
lastName: 'Doe',  
 age: 30  
}

第二个的参数为键名

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

第三个参数为索引

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

Ps:在遍历对象时，是按 Object.keys() 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的

### key

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue将不会移动 DOM 元素来匹配数据项的顺序，而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素.

这个默认的模式是高效的，但是只适用于**不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出**。

它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性。理想的 key 值是每项都有的唯一 id。

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

### 数组更新检测

#### 变异方法

以下方法会触发视图更新

push、pop、shift、unshift、splice、sort、reverse

#### 替换数组

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

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

#### 注意事项

由于 JavaScript 的限制，Vue不能检测以下变动的数组：

1，当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue

2，,当你修改数组的长度时，例如：vm.items.length = newLength

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

可以使用以下方法

|  |
| --- |
| // Vue.set Vue.set(vm.items, indexOfItem, newValue) |

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

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

### 对象更改检测注意事项

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

可以使用Vue.set(object, key, value)

使用 Object.assign() 或 \_.extend()时应该使用：

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

### 显示过滤/排序结果

可以创建返回过滤或排序数组的计算属性

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

data: {  
 numbers: [ 1, 2, 3, 4, 5 ]  
},  
computed: {  
evenNumbers: function () {  
returnthis.numbers.filter(function (number) {  
return number % 2 === 0  
 })  
 }  
}

在计算属性不适用的情况下可以使用method方法

methods: {  
 even: function (numbers) {  
returnnumbers.filter(function (number) {  
return number % 2 === 0  
 })  
 }  
}

### 有范围的v-for

<spanv-for="n in 10">{{ n }} </span>

### Template上使用v-for

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

### v-for与v-if

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

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

### 一个组件的v-for

可以正常使用，只不过需要通过组件的porp传入值

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

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

is="todo-item" 属性可以避开一些潜在的浏览器解析错误

<ul>  
<li  
is="todo-item"  
v-for="(todo, index) in todos"  
v-bind:key="todo.id"  
v-bind:title="todo.title"  
v-on:remove="todos.splice(index, 1)"  
></li>  
</ul>

## 事件处理

### 监听事件

<buttonv-on:click="counter += 1">Add 1</button>

### 事件处理方法

<buttonv-on:click="greet">Greet</button>

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

### 内联处理器中的方法

<buttonv-on:click="say('hi')">Say hi</button>

methods: {  
 say: function (message) {  
 alert(message)  
 }  
 }

可以用特殊变量 $event 把它传入方法

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

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

### 事件修饰符

Vue提供了以下修饰符

.stop 阻止单击事件继续传播

.prevent 阻止默认事件，例如提交是不刷新页面

.captrue事件捕获模式，触发的事件现在此处处理，然后才由内部元素处理

.self 仅在当前元素触发

.once 只会触发一次ps:2.1.3新增

.passive 触发事件的默认行为，主要用在移动端滚动事件

修饰符顺序很重要v-on:click.prevent.self 会阻止**所有的点击**，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

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

### 按键修饰符

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

别名

<!--同上 -->  
<inputv-on:keyup.enter="submit">

全部默认按键别名有

.enter

.tab

.delete 捕获删除与退格键

.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**](https://cn.vuejs.org/v2/api/#keyCodes)：

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

#### 自动匹配按键修饰符

可直接将 [**KeyboardEvent.key**](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values) 暴露的任意有效按键名转换为 kebab-case 来作为修饰符：

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

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

### 系统修饰键

2.1.0新增

.ctrl

.alt

.shift

.meta

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

实例

<!-- Alt + C -->  
<input @keyup.alt.67="clear">  
<!-- Ctrl + Click -->  
<div @click.ctrl="doSomething">Do something</div>

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

#### .exact修饰符

2.5.0新增

.exact 修饰符允许你控制由精确的系统修饰符组合触发的事件。

<!--即使 Alt 或 Shift 被一同按下时也会触发 -->  
<button @click.ctrl="onClick">A</button>  
  
<!-- 有且只有 Ctrl 被按下的时候才触发 -->  
<button @click.ctrl.exact="onCtrlClick">A</button>  
  
<!-- 没有任何系统修饰符被按下的时候才触发 -->  
<button @click.exact="onClick">A</button>

#### 鼠标按钮修饰符

2.2.0新增

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

.left

.right

.middle

### 为什么在HTML中监听事件

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

1，扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。

2，因为你无须在 JavaScript 里手动绑定事件，你的ViewModel代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。

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

## 表单输入绑定

### 基础用法

v-model, 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

Ps:v-model 会忽略所有表单元素的 value、checked、selected 特性的初始值而总是将Ps:Vue实例的数据作为数据来源

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

#### 文本

<input v-model="message" placeholder="edit me">

#### 多行文本

<textarea v-model="message" placeholder="add multiple lines"></textarea>

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

#### 复选框

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

多个复选框，绑定到同一个数组

data: {  
checkedNames: []  
}

#### 单选框

<input type="radio" id="one" value="One" v-model="picked">

#### 下拉框

<select v-model="selected">  
<option disabled value="">请选择</option>  
<option>A</option>  
<option>B</option>  
<option>C</option>  
</select>

Ps：如果 v-model 表达式的初始值未能匹配任何选项，<select> 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 change 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

多选时 (绑定到一个数组)：

<select v-model="selected" multiple style="width: 50px;">  
<option>A</option>  
<option>B</option>  
<option>C</option>  
</select>

data: {  
 selected: []  
 }

v-for渲染的动态选项

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

data: {  
 selected: 'A',  
 options: [  
 { text: 'One', value: 'A' },  
 { text: 'Two', value: 'B' },  
 { text: 'Three', value: 'C' }  
 ]  
 }

### 绑定值

v-model 绑定的值通常是静态字符串 (对于复选框也可以是布尔值)

绑定动态属性用 v-bind 实现，并且这个属性的值可以不是字符串

#### 复选框

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

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

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

#### 单选按钮

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

// 当选中时  
vm.pick === vm.a

#### 下拉框选项

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

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

### 修饰符

#### .lazy

在默认情况下，v-model 在每次 input 事件触发后将输入框的值与数据进行同步 (除了[**上述**](https://cn.vuejs.org/v2/guide/forms.html#vmodel-ime-tip)输入法组合文字时)。你可以添加 lazy 修饰符，从而转变为使用 change事件进行同步：

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

#### .number

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

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

如果这个值无法被 parseFloat() 解析，则会返回原始的值。

#### .trim

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

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

### 在组件上使用v-model

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 v-model 一起使用

## 组件基础

### 基本示例

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

组件是可复用的Vue实例，且带有一个名字

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

|  |
| --- |
| newVue({ el: '#components-demo' }) |

通过初始化根实例来吧组件引入

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

### 组件的复用

可以将组件进行任意次数的复用

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

你每用一次组件，就会有一个它的新**实例**被创建

#### Data必须是一个函数

**一个组件的 data 选项必须是一个函数**，因此每个实例可以维护一份被返回对象的独立的拷贝，没有这条规则，点击一个按钮会影响到其它所有实例

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

### 组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织



为了能在模板中使用，这些组件必须先注册以便Vue能够识别。这里有两种组件的注册类型：**全局注册**和**局部注册**。

通过 Vue.component 全局注册的

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

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

### 通过prop向子组件传递数据

Prop 是你可以在组件上注册的一些自定义特性。当一个值传递给一个 prop 特性的时候，它就变成了那个组件实例的一个属性。

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

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。

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

批量渲染组件时

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

### 单个根元素

**every component must have a single root element (每个组件必须只有一个根元素)**

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

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

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

### 通过事件向父组件发送消息

调用内建的 [**$emit 方法**](https://cn.vuejs.org/v2/api/#vm-emit)并传入事件的名字，来向父级组件触发一个事件

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

然后用 v-on 在组件上监听事件，就像监听一个原生 DOM 事件一样

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

#### 使用事件抛出一个值

$emit 的第二个参数来提供这个值

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

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

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

或者，事件处理函数是一个方法：

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

这个值将会作为第一个参数传入这个方法：

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

#### 在组件上使用v-model

为了让它正常工作，这个组件内的 <input> 必须：

1，将其 value 特性绑定到一个名叫 value 的 prop 上

2，在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

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

<custom-inputv-model="searchText"></custom-input>

### 通过插槽分发内容

Vue自定义的 <slot> 元素让这变得非常简单

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

### 动态组件

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

currentTabComponent 可以包括

1，已注册组件的名字，或,

2，一个组件的选项对象

### 解析DOM模板时的注意事项

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

Ps：需要注意的是**如果我们从以下来源使用模板的话，这条限制是不存在的**：

1，字符串 (例如：template: '...')

2，[单文件组件 (.vue)](https://cn.vuejs.org/v2/guide/single-file-components.html)

3，[<script type="text/x-template">](https://cn.vuejs.org/v2/guide/components-edge-cases.html#X-Templates)

# 深入了解组件

## 组件注册

### 组件名

在注册一个组件的时候，我们始终需要给它一个名字

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

Ps：组件名 (字母全小写且必须包含一个连字符)。这会帮助你避免和当前以及未来的 HTML 元素相冲突

#### 组件名大小写

定义组件名的方式有两种

Kebab-case(短横线隔开)

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

pascalBase(首字母大写命名—驼峰式)

### 全局注册

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

子组件之间也可互相用

### 局部注册

**局部注册的组件在其子组件中不可用**

importComponentAfrom'./ComponentA.vue'

components: {  
ComponentA  
}

### 模块系统

#### 在模块系统中局部注册

importComponentAfrom'./ComponentA'  
importComponentCfrom'./ComponentC'  
  
exportdefault {  
 components: {  
ComponentA,  
ComponentC  
 },  
// ...  
}

#### 基础组件的自动化全局注册

如果你使用了webpack (或在内部使用了webpack的 [**Vue CLI 3+**](https://github.com/vuejs/vue-cli))，那么就可以使用 require.context 只全局注册这些非常通用的基础组件

importVuefrom'vue'  
importupperFirstfrom'lodash/upperFirst'  
importcamelCasefrom'lodash/camelCase'  
  
constrequireComponent = require.context(  
// 其组件目录的相对路径  
'./components',  
// 是否查询其子目录  
false,  
// 匹配基础组件文件名的正则表达式  
 /Base[A-Z]\w+\.(vue|js)$/  
)  
  
requireComponent.keys().forEach(fileName => {  
// 获取组件配置  
constcomponentConfig = requireComponent(fileName)  
  
// 获取组件的PascalCase命名  
constcomponentName = upperFirst(  
camelCase(  
// 剥去文件名开头的 `./` 和结尾的扩展名  
fileName.replace(/^\.\/(.\*)\.\w+$/, '$1')  
 )  
 )  
  
// 全局注册组件  
Vue.component(  
componentName,  
// 如果这个组件选项是通过 `export default` 导出的，  
// 那么就会优先使用 `.default`，  
// 否则回退到使用模块的根。  
componentConfig.default || componentConfig  
 )  
})

Ps:**全局注册的行为必须在根Vue实例 (通过 new Vue) 创建之前发生**

## Prop

### Prop的大小写(camelCase vs kebab-case)

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

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

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

### Prop类型

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

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

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

也可简写

### 传递静态或动态Prop

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

#### 传入一个数字

<!--这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-postv-bind:likes="42"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-postv-bind:likes="post.likes"></blog-post>

#### 传入一个布尔值

<!--包含该 prop 没有值的情况在内，都意味着 `true`。-->  
<blog-postis-published></blog-post>  
<!--即便 `false` 是静态的，我们仍然需要 `v-bind` 来告诉Vue -->  
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-postv-bind:is-published="false"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-postv-bind:is-published="post.isPublished"></blog-post>

#### 传入一个数组

<!--这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-postv-bind:comment-ids="[234, 266, 273]"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-postv-bind:comment-ids="post.commentIds"></blog-post>

#### 传入一个对象

<!--即便对象是静态的，我们仍然需要 `v-bind` 来告诉Vue -->  
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-postv-bind:author="{ name: 'Veronica', company: 'Veridian Dynamics' }"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-postv-bind:author="post.author"></blog-post>

#### 传入一个对象的所有属性

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

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

<blog-postv-bind="post"></blog-post>

等价于：

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

### 单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你**不**应该在一个子组件内部改变 prop。如果你这样做了，Vue会在浏览器的控制台中发出警告

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

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

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

|  |
| --- |
| props: ['size'], computed: { normalizedSize: function () { returnthis.size.trim().toLowerCase()  } } |

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

### Prop验证

Vue.component('my-component', {  
 props: {  
// 基础的类型检查 (`null` 匹配任何类型)  
propA: Number,  
// 多个可能的类型  
propB: [String, Number],  
// 必填的字符串  
propC: {  
 type: String,  
 required: true  
 },  
// 带有默认值的数字  
propD: {  
 type: Number,  
default: 100  
 },  
// 带有默认值的对象  
propE: {  
 type: Object,  
// 对象或数组默认值必须从一个工厂函数获取  
default: function () {  
return { message: 'hello' }  
 }  
 },  
// 自定义验证函数  
propF: {  
 validator: function (value) {  
// 这个值必须匹配下列字符串中的一个  
return ['success', 'warning', 'danger'].indexOf(value) !== -1  
 }  
 }  
 }  
})

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

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

#### 类型检查

String，Number，Boolean，Array，Object，Date，Function，Symbol

通过 instanceof 来进行检查

例如，给定下列现成的构造函数：

functionPerson (firstName, lastName) {  
this.firstName = firstName  
this.lastName = lastName  
}

props: {  
 author: Person  
}

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

### 非Prop特性

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

例如

<bootstrap-date-inputdata-date-picker="activated"></bootstrap-date-input>

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

#### 替换/合并已有的特性

从外部提供给组件的值会替换掉组件内部设置好的值。例如传入的input的type的值是可变的，但传入的class以及style的值是会合并的

#### 禁用特性继承

如果你**不**希望组件的根元素继承特性，你可以在组件的选项中设置

inheritAttrs: false

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

有了 inheritAttrs: false 和 $attrs，你就可以手动决定这些特性会被赋予哪个元素。在撰写[基础组件](https://cn.vuejs.org/v2/style-guide/#%E5%9F%BA%E7%A1%80%E7%BB%84%E4%BB%B6%E5%90%8D-%E5%BC%BA%E7%83%88%E6%8E%A8%E8%8D%90)的时候是常会用到的：

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

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

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

## 自定义事件

### 事件名

触发的事件名需要完全匹配监听这个事件所用的名称

事件名不会被用作一个 JavaScript 变量名或属性名，所以使用 kebab-case 的事件名

### 自定义组件的v-model

2.2.0新增

一个组件上的 v-model 默认会利用名为 value 的 prop 和名为 input 的事件，但是像单选框、复选框等类型的输入控件可能会将 value 特性用于[**不同的目的**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox#Value)。model 选项可以用来避免这样的冲突

template: `  
<input  
 type="checkbox"  
v-bind:checked="checked"  
v-on:change="$emit('change', $event.target.checked)"  
>  
 `

<base-checkboxv-model="lovingVue"></base-checkbox>

这里的 lovingVue 的值将会传入这个名为 checked 的 prop。同时当 <base-checkbox> 触发一个 change 事件并附带一个新的值的时候，这个 lovingVue 的属性将会被更新

Ps：注意你仍然需要在组件的 props 选项里声明 checked 这个 prop。

### 将原生事件绑定到组件

Vue提供了一个 $listeners 属性，它是一个对象，里面包含了作用在这个组件上的所有监听器

computed: {  
inputListeners: function () {  
varvm = this  
// `Object.assign` 将所有的对象合并为一个新对象  
returnObject.assign({},  
// 我们从父级添加所有的监听器  
this.$listeners,  
// 然后我们添加自定义监听器，  
// 或覆写一些监听器的行为  
 {  
// 这里确保组件配合 `v-model` 的工作  
 input: function (event) {  
vm.$emit('input', event.target.value)  
 }  
 }  
 )  
 }  
 },

所有跟它相同的特性和监听器的都可以工作

### .sync修饰符

2.3.0新增

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

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

可以设置多个

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

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

## 插槽

### 插槽内容

<slot> 元素作为承载分发内容的出口

<navigation-linkurl="/profile">  
 Your Profile  
</navigation-link>

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

如果组件**没有**包含一个 <slot> 元素，则任何传入它的内容都会被抛弃

### 具名插槽

当需要多个插槽时，这样的情况，<slot> 元素有一个特殊的特性：name

<divclass="container">  
<header>  
<slotname="header"></slot>  
</header>  
<main>  
<slot></slot>  
</main>  
<footer>  
<slotname="footer"></slot>  
</footer>  
</div>

<templateslot="header">  
<h1>Here might be a page title</h1>  
</template>

或者

<h1slot="header">Here might be a page title</h1>

可以保留一个未命名插槽，这个插槽是**默认插槽**，也就是说它会作为所有未匹配到插槽的内容的统一出口

### 插槽的默认内容

可以在组件模板里的 <slot> 标签内部指定默认的内容来做到这一点

<slot>Submit</slot>

父组件为这个插槽提供了内容，则默认的内容会被替换掉

### 编译作用域

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

在插槽内使用数据时

<navigation-linkurl="/profile">  
 Logged in as {{ user.name }}  
</navigation-link>

请按照上述规则编写

### 作用域插槽

2.1.0新增

当需要在插槽内使用数据

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

可以选择为待办项定义一个不一样的 <template> 作为替代方案，并且可以通过 slot-scope 特性从子组件获取数据

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

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

#### 解构slot-scope

如果一个 JavaScript 表达式在一个函数定义的参数位置有效，那么这个表达式实际上就可以被 slot-scope 接受。也就是说你可以在支持的环境下 ([**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html)或[**现代浏览器**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#%E6%B5%8F%E8%A7%88%E5%99%A8%E5%85%BC%E5%AE%B9))，在这些表达式中使用 [**ES2015 解构语法**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#%E8%A7%A3%E6%9E%84%E5%AF%B9%E8%B1%A1)

<todo-listv-bind:todos="todos">  
<templateslot-scope="{ todo }">  
<spanv-if="todo.isComplete">✓</span>  
 {{ todo.text }}  
</template>  
</todo-list>

PS：**推荐使用**

## 动态组件&异步组件

### 在组件上使用keep-alive

<keep-alive>  
<componentv-bind:is="currentTabComponent"></component>  
</keep-alive>

Ps:注意这个 <keep-alive> 要求被切换到的组件都有自己的名字，不论是通过组件的 name 选项还是局部/全局注册

### 异步组件

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

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

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

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

当使用[**局部注册**](https://cn.vuejs.org/v2/guide/components.html#%E6%9C%AC%E5%9C%B0%E6%B3%A8%E5%86%8C)的时候，你也可以直接提供一个返回 Promise 的函数：

|  |
| --- |
| new Vue({ // ...  components: { 'my-component': () =>import('./my-async-component')  } }) |

Ps：如果你是一个 **Browserify** 用户同时喜欢使用异步组件，很不幸这个工具的作者[**明确表示**](https://github.com/substack/node-browserify/issues/58#issuecomment-21978224)异步加载“并不会被 Browserify 支持”，至少官方不会。Browserify 社区已经找到了[**一些变通方案**](https://github.com/vuejs/vuejs.org/issues/620)，这些方案可能会对已存在的复杂应用有帮助。对于其它的场景，我们推荐直接使用 webpack，以拥有内建的被作为第一公民的异步支持。

#### 处理加载状态

2.3.0新增

这里的异步组件工厂函数也可以返回一个如下格式的对象：

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

**注意如果你希望在**[**Vue Router**](https://github.com/vuejs/vue-router)**的路由组件中使用上述语法的话，你必须使用 Vue Router 2.4.0+ 版本。**

## 处理边界情况

### 访问元素&组件

#### 访问根实例

每个 new Vue 实例的子组件中，其根实例可以通过 $root 属性进行访问

// 获取根组件的数据  
this.$root.foo  
// 写入根组件的数据  
this.$root.foo = 2  
// 访问根组件的计算属性  
this.$root.bar  
// 调用根组件的方法  
this.$root.baz()

PS：对于 demo 或非常小型的有少量组件的应用来说这是很方便的。不过这个模式扩展到中大型应用来说就不然了。因此在绝大多数情况下，我们强烈推荐使用 [**Vuex**](https://github.com/vuejs/vuex) 来管理应用的状态

#### 访问父组件实例

$parent 属性可以用来从一个子组件访问父组件的实例

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

this.$parent.map

推荐[**依赖注入**](https://cn.vuejs.org/v2/guide/components-edge-cases.html#%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5)

#### 访问子组件实例或子元素

通过 ref 特性为这个子组件赋予一个 ID 引用

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

this.$refs.usernameInput

获取元素

<input ref="input">

this.$refs.input.focus()

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

#### 依赖注入

provide和 inject

provide 选项允许我们指定我们想要**提供**给后代组件的数据/方法

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

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

inject: ['getMap']

PS：然而，依赖注入还是有负面影响的。它将你的应用以目前的组件组织方式耦合了起来，使重构变得更加困难。同时所提供的属性是非响应式的。这是出于设计的考虑，因为使用它们来创建一个中心化规模化的数据跟[**使用 $root**](https://cn.vuejs.org/v2/guide/components-edge-cases.html#%E8%AE%BF%E9%97%AE%E6%A0%B9%E5%AE%9E%E4%BE%8B)做这件事都是不够好的。如果你想要共享的这个属性是你的应用特有的，而不是通用化的，或者如果你想在祖先组件中更新所提供的数据，那么这意味着你可能需要换用一个像 [**Vuex**](https://github.com/vuejs/vuex) 这样真正的状态管理方案了。

### 程序化的事件侦听器

通过 $on(eventName, eventHandler) 侦听一个事件

通过 $once(eventName, eventHandler) 一次性侦听一个事件

通过 $off(eventName, eventHandler) 停止侦听一个事件

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

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

mounted: function () {  
this.attachDatepicker('startDateInput')  
this.attachDatepicker('endDateInput')  
},

methods: {  
 attachDatepicker: function (refName) {  
var picker = new Pikaday({  
 field: this.$refs[refName],  
 format: 'YYYY-MM-DD'  
 })  
  
this.$once('hook:beforeDestroy', function () {  
 picker.destroy()  
 })  
 }  
}

PS：注意 Vue 的事件系统不同于浏览器的 [**EventTarget API**](https://developer.mozilla.org/zh-CN/docs/Web/API/EventTarget)。尽管它们工作起来是相似的，但是 $emit、$on, 和 $off 并不是 dispatchEvent、addEventListener 和 removeEventListener 的别名

### 循环引用

#### 递归组件

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

当你使用 Vue.component 全局注册一个组件时，这个全局的 ID 会自动设置为该组件的 name 选项，稍有不慎，递归组件就可能导致无限循环。类似上述的组件将会导致“max stack size exceeded”错误，所以请确保递归调用是条件性的 (例如使用一个最终会得到 false 的 v-if)

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

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

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

#### 组件之间的循环引用

当两个组件项目依赖时会遇到一个错误：

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

可以在beforeCreate 时去注册它

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

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

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

### 模板定义的代替品

#### 内联模板

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

<my-component inline-template>  
<div>  
<p>These are compiled as the component's own template.</p>  
<p>Not parent's transclusion content.</p>  
</div>  
</my-component>

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

#### X-Tepmlate

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

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

Vue.component('hello-world', {  
 template: '#hello-world-template'  
})

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

### 控制更新

#### 强制更新

PS：如果你发现你自己需要在 Vue 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事

可以通过 [**$forceUpdate**](https://cn.vuejs.org/v2/api/#vm-forceUpdate) 来做这件事

#### 通过v-once创建低开销的静态组件

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

PS：再说一次，试着不要过度使用这个模式。当你需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非你非常留意渲染变慢了，不然它完全是没有必要的——再加上它在后期会带来很多困惑

# 过渡&动画

## 进入离开&列表过渡

### 概述

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。包括以下：

1，在 CSS 过渡和动画中自动应用 class

2，可以配合使用第三方 CSS 动画库，如 Animate.css

3，在过渡钩子函数中使用 JavaScript 直接操作 DOM

4，可以配合使用第三方 JavaScript 动画库，如 Velocity.js

### 单元素/组件的过渡

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

1，条件渲染 (使用 v-if)

2，条件展示 (使用 v-show)

3，动态组件

4，组件根节点

<transition name="fade">  
<p v-if="show">hello</p>  
</transition>

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

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

1，自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名

2，如果过渡组件提供了 [**JavaScript 钩子函数**](https://cn.vuejs.org/v2/guide/transitions.html#JavaScript-%E9%92%A9%E5%AD%90)，这些钩子函数将在恰当的时机被调用

3，如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作 (插入/删除) 在下一帧中立即执行。(注意：此指浏览器逐帧动画机制，和 Vue 的 nextTick 概念不同)

#### 过渡的类名

在进入/离开的过渡中，会有 6 个 class 切换

1，v-enter：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除

2，v-enter-active：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数

3，v-enter-to: **2.1.8版及以上** 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除)，在过渡/动画完成之后移除

4，v-leave: 定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除

5，v-leave-active：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数

6，v-leave-to: **2.1.8版及以上** 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除)，在过渡/动画完成之后移除



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

v-enter-active 和 v-leave-active 可以控制进入/离开过渡的不同的缓和曲线

#### CSS过渡

常用的过渡都是使用 CSS 过渡

<transition name="slide-fade">  
<p v-if="show">hello</p>  
</transition>

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

#### 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

<transitionname="bounce">  
<pv-if="show">Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi tristique senectus et netus.</p>  
</transition>

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

#### 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

1，enter-class

2，enter-active-class

3，enter-to-class (2.1.8+)

4，leave-class

5，leave-active-class

6，leave-to-class (2.1.8+)

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如 [**Animate.css**](https://daneden.github.io/animate.css/) 结合使用十分有用

<transition  
 name="custom-classes-transition"  
 enter-active-class="animated tada"  
 leave-active-class="animated bounceOutRight"  
>  
<p v-if="show">hello</p>  
</transition>

#### 同时使用过渡和动画

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

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

#### 显性的过渡持续时间

2.2.0新增

可以用 <transition> 组件上的 duration 属性定制一个显性的过渡持续时间 (以毫秒计)

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

也可以定制进入和移出的持续时间

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

#### Javascript钩子

可以在属性中声明 JavaScript 钩子

<transition  
 v-on:before-enter="beforeEnter"  
 v-on:enter="enter"  
 v-on:after-enter="afterEnter"  
 v-on:enter-cancelled="enterCancelled"  
  
 v-on:before-leave="beforeLeave"  
 v-on:leave="leave"  
 v-on:after-leave="afterLeave"  
 v-on:leave-cancelled="leaveCancelled"  
>  
<!-- ... -->  
</transition>

methods: {  
// --------  
// 进入中  
// --------  
 beforeEnter: function (el) {  
// ...  
 },  
// 当与 CSS 结合使用时  
// 回调函数 done 是可选的  
 enter: function (el, done) {  
// ...  
 done()  
 },  
 afterEnter: function (el) {  
// ...  
 },  
 enterCancelled: function (el) {  
// ...  
 },  
// --------  
// 离开时  
// --------  
 beforeLeave: function (el) {  
// ...  
 },  
// 当与 CSS 结合使用时  
// 回调函数 done 是可选的  
 leave: function (el, done) {  
// ...  
 done()  
 },  
 afterLeave: function (el) {  
// ...  
 },  
// leaveCancelled 只用于 v-show 中  
 leaveCancelled: function (el) {  
// ...  
 }  
}

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

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

PS：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。

### 初始渲染的过渡

可以通过 appear 特性设置节点在初始渲染的过渡

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

可以自定义 CSS 类名

<transition  
 appear  
 appear-class="custom-appear-class"  
 appear-to-class="custom-appear-to-class" (2.1.8+)  
 appear-active-class="custom-appear-active-class"  
>

自定义 JavaScript 钩子

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

### 多个元素的过渡

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

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

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

可以简写为

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

或

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

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

#### 过渡模式

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

1，in-out：新元素先进行过渡，完成之后当前元素过渡离开

2，out-in：当前元素先进行过渡，完成之后新元素过渡进入

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

### 多个组件的过渡

多个组件的过渡简单很多 - 我们不需要使用 key 特性。相反，我们只需要使用[**动态组件**](https://cn.vuejs.org/v2/guide/components.html#%E5%8A%A8%E6%80%81%E7%BB%84%E4%BB%B6)

<transitionname="component-fade"mode="out-in">  
<componentv-bind:is="view"></component>  
</transition>

data: {  
 view: 'v-a'  
 },  
 components: {  
'v-a': {  
 template: '<div>Component A</div>'  
 },  
'v-b': {  
 template: '<div>Component B</div>'  
 }  
 }

### 列表过渡

<transition-group> 组件

**特点：1，**不同于 <transition>，它会以一个真实元素呈现：默认为一个 <span>。你也可以通过 tag 特性更换为其他元素。

2，[**过渡模式**](https://cn.vuejs.org/v2/guide/transitions.html#%E8%BF%87%E6%B8%A1%E6%A8%A1%E5%BC%8F)不可用，因为我们不再相互切换特有的元素

**3，**内部元素 **总是需要** 提供唯一的 key 属性值

#### 列表的进入/离开过渡

<transition-group name="list" tag="p">  
<span v-for="item in items" v-bind:key="item" class="list-item">  
 {{ item }}  
</span>  
</transition-group>

#### 列表的排序过渡

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

<transition-group name="flip-list" tag="ul">  
<li v-for="item in items" v-bind:key="item">  
 {{ item }}  
</li>  
</transition-group>

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

Vue 使用了一个叫 [**FLIP**](https://aerotwist.com/blog/flip-your-animations/) 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

**PS：**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

#### 列表的交错过渡

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

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

<transition-group  
 name="staggered-fade"  
 tag="ul"  
 v-bind:css="false"  
 v-on:before-enter="beforeEnter"  
 v-on:enter="enter"  
 v-on:leave="leave"  
>  
<li  
 v-for="(item, index) in computedList"  
 v-bind:key="item.msg"  
 v-bind:data-index="index"  
>{{ item.msg }}</li>  
</transition-group>

methods: {  
 beforeEnter: function (el) {  
 el.style.opacity = 0  
 el.style.height = 0  
 },  
 enter: function (el, done) {  
var delay = el.dataset.index \* 150  
 setTimeout(function () {  
 Velocity(  
 el,  
 { opacity: 1, height: '1.6em' },  
 { complete: done }  
 )  
 }, delay)  
 },  
 leave: function (el, done) {  
var delay = el.dataset.index \* 150  
 setTimeout(function () {  
 Velocity(  
 el,  
 { opacity: 0, height: 0 },  
 { complete: done }  
 )  
 }, delay)  
 }  
 }

### 动态过渡

在 Vue 中即使是过渡也是数据驱动的！动态过渡最基本的例子是通过 name 特性来绑定动态值

<transition v-bind:name="transitionName">  
<!-- ... -->  
</transition>

所有过渡特性都可以动态绑定，但我们不仅仅只有特性可以利用，还可以通过事件钩子获取上下文中的所有数据，因为事件钩子都是方法。这意味着，根据组件的状态不同，你的 JavaScript 过渡会有不同的表现。

<transition  
 v-bind:css="false"  
 v-on:before-enter="beforeEnter"  
 v-on:enter="enter"  
 v-on:leave="leave"  
>  
<p v-if="show">hello</p>  
</transition>

methods: {  
 beforeEnter: function (el) {  
 el.style.opacity = 0  
 },  
 enter: function (el, done) {  
var vm = this  
 Velocity(el,  
 { opacity: 1 },  
 {  
 duration: this.fadeInDuration,  
 complete: function () {  
 done()  
if (!vm.stop) vm.show = false  
 }  
 }  
 )  
 },  
 leave: function (el, done) {  
var vm = this  
 Velocity(el,  
 { opacity: 0 },  
 {  
 duration: this.fadeOutDuration,  
 complete: function () {  
 done()  
 vm.show = true  
 }  
 }  
 )  
 }  
 }

最后，创建动态过渡的最终方案是组件通过接受 props 来动态修改之前的过渡。一句老话，唯一的限制是你的想象力

## 状态过渡

元素本身的动效：1，数字和运算。2，颜色的显示。3，SVG 节点的位置。4，元素的大小和其他的属性

所有的原始数字都被事先存储起来，可以直接转换到数字。做到这一步，我们就可以结合 Vue 的响应式和组件系统，使用第三方库来实现切换元素的过渡状态

### 状态动画与侦听器

通过侦听器我们能监听到任何数值属性的数值更新

<input v-model.number="number" type="number" step="20">  
<p>{{ animatedNumber }}</p>

computed: {  
 animatedNumber: function() {  
returnthis.tweenedNumber.toFixed(0);  
 }  
 },  
 watch: {  
 number: function(newValue) {  
 TweenLite.to(this.$data, 0.5, { tweenedNumber: newValue });  
 }  
 }

当改变的是例如样式颜色这样的值时

<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>  
<script src="https://cdn.jsdelivr.net/npm/color-js@1.0.3"></script>

<span  
 v-bind:style="{ backgroundColor: tweenedCSSColor }"  
 class="example-7-color-preview"  
></span>

data: {  
 colorQuery: '',  
 color: {  
 red: 0,  
 green: 0,  
 blue: 0,  
 alpha: 1  
 },  
 tweenedColor: {}  
 },  
 created: function () {  
this.tweenedColor = Object.assign({}, this.color)  
 },  
 watch: {  
 color: function () {  
functionanimate () {  
if (TWEEN.update()) {  
 requestAnimationFrame(animate)  
 }  
 }  
  
new TWEEN.Tween(this.tweenedColor)  
 .to(this.color, 750)  
 .start()  
  
 animate()  
 }  
 },  
 computed: {  
 tweenedCSSColor: function () {  
returnnew Color({  
 red: this.tweenedColor.red,  
 green: this.tweenedColor.green,  
 blue: this.tweenedColor.blue,  
 alpha: this.tweenedColor.alpha  
 }).toCSS()  
 }  
 },  
 methods: {  
 updateColor: function () {  
this.color = new Color(this.colorQuery).toRGB()  
this.colorQuery = ''  
 }  
 }

### 动态状态过渡

就像 Vue 的过渡组件一样，数据背后状态过渡会实时更新，这对于原型设计十分有用。当你修改一些变量，即使是一个简单的 SVG 多边形也可实现很多难以想象的效果。

### 把过渡放到组件里

管理太多的状态过渡会很快的增加 Vue 实例或者组件的复杂性，幸好很多的动画可以提取到专用的子组件

Vue.component('animated-integer', {  
 template: '<span>{{ tweeningValue }}</span>',  
 props: {  
 value: {  
 type: Number,  
 required: true  
 }  
 },  
 data: function () {  
return {  
 tweeningValue: 0  
 }  
 },  
 watch: {  
 value: function (newValue, oldValue) {  
this.tween(oldValue, newValue)  
 }  
 },  
 mounted: function () {  
this.tween(0, this.value)  
 },  
 methods: {  
 tween: function (startValue, endValue) {  
var vm = this  
functionanimate () {  
if (TWEEN.update()) {  
 requestAnimationFrame(animate)  
 }  
 }  
  
new TWEEN.Tween({ tweeningValue: startValue })  
 .to({ tweeningValue: endValue }, 500)  
 .onUpdate(function (object) {  
 vm.tweeningValue = object.tweeningValue.toFixed(0)  
 })  
 .start()  
  
 animate()  
 }  
 }  
})

我们能在组件中结合使用这一节讲到各种过渡策略和 Vue [**内建的过渡系统**](https://cn.vuejs.org/v2/guide/transitions.html)。总之，对于完成各种过渡动效几乎没有阻碍

### 赋予设计以生命

只要一个动画，就可以带来生命。不幸的是，当设计师创建图标、logo 和吉祥物的时候，他们交付的通常都是图片或静态的 SVG。所以，虽然 GitHub 的章鱼猫、Twitter 的小鸟以及其它许多 logo 类似于生灵，它们看上去实际上并不是活着的。

Vue 可以帮到你。因为 SVG 的本质是数据，我们只需要这些动物兴奋、思考或警戒的样例。然后 Vue 就可以辅助完成这几种状态之间的过渡动画，来制作你的欢迎页面、加载指示、以及更加带有情感的提示

# 可用性&混合

## 混入

### 基础

混入 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项

var myMixin = {  
 created: function () {  
this.hello()  
 },  
 methods: {  
 hello: function () {  
console.log('hello from mixin!')  
 }  
 }  
}  
  
// 定义一个使用混入对象的组件  
var Component = Vue.extend({  
 mixins: [myMixin]  
})  
  
var component = new Component()

### 选项合并

当组件和混入对象含有同名选项时，这些选项将以恰当的方式混合

数据对象在内部会进行浅合并 (一层属性深度)，在和组件的数据发生冲突时以组件数据优先

同名钩子函数将混合为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子**之前**调用

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

PS：Vue.extend() 也使用同样的策略进行合并

### 全局混入

也可以全局注册混入对象。注意使用！一旦使用全局混入对象，将会影响到 **所有** 之后创建的 Vue 实例。使用恰当时，可以为自定义对象注入处理逻辑

// 为自定义的选项 'myOption' 注入一个处理器。  
Vue.mixin({  
 created: function () {  
var myOption = this.$options.myOption  
if (myOption) {  
console.log(myOption)  
 }  
 }  
})  
  
new Vue({  
 myOption: 'hello!'  
})

### 自定义选项合并策略

自定义选项将使用默认策略，即简单地覆盖已有值。如果想让自定义选项以自定义逻辑合并，可以向 Vue.config.optionMergeStrategies 添加一个函数

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

对于大多数对象选项，可以使用 methods 的合并策略

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

## 自定义指令

### 简介

例如需要一进页面就获取焦点

Vue.directive('focus', {  
// 当被绑定的元素插入到 DOM 中时……  
 inserted: function (el) {  
// 聚焦元素  
 el.focus()  
 }  
})

如果想注册局部指令，组件中也接受一个 directives 的选项

directives: {  
 focus: {  
// 指令的定义  
 inserted: function (el) {  
 el.focus()  
 }  
 }  
}

然后你可以在模板中任何元素上使用新的 v-focus 属性

<inputv-focus>

### 钩子函数

一个指令定义对象可以提供如下几个钩子函数 (均为可选)

1，bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置

2，inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)

3，update：所在组件的 VNode 更新时调用，**但是可能发生在其子 VNode 更新之前**。指令的值可能发生了改变，也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新 (详细的钩子函数参数见下)

4，componentUpdated：指令所在组件的 VNode **及其子 VNode** 全部更新后调用

5，unbind：只调用一次，指令与元素解绑时调用

### 钩子函数参数

指令钩子函数会被传入以下参数

1，el：指令所绑定的元素，可以用来直接操作 DOM

2，binding：一个对象，包含以下属性：1，name：指令名，不包括 v- 前缀。2，value：指令的绑定值，例如：v-my-directive="1 + 1" 中，绑定值为 2。3，oldValue：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。4，expression：字符串形式的指令表达式。例如 v-my-directive="1 + 1"中，表达式为 "1 + 1"。5，arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。6，modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为 { foo: true, bar: true }。

3，vnode：Vue 编译生成的虚拟节点。移步 [**VNode API**](https://cn.vuejs.org/v2/api/#VNode-%E6%8E%A5%E5%8F%A3) 来了解更多详情

4，oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用

PS：除了 el 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 [**dataset**](https://developer.mozilla.org/zh-CN/docs/Web/API/HTMLElement/dataset) 来进行

<div id="hook-arguments-example" v-demo:foo.a.b="message"></div>

Vue.directive('demo', {  
 bind: function (el, binding, vnode) {  
var s = JSON.stringify  
 el.innerHTML =  
'name: ' + s(binding.name) + '<br>' +  
'value: ' + s(binding.value) + '<br>' +  
'expression: ' + s(binding.expression) + '<br>' +  
'argument: ' + s(binding.arg) + '<br>' +  
'modifiers: ' + s(binding.modifiers) + '<br>' +  
'vnode keys: ' + Object.keys(vnode).join(', ')  
 }  
})

### 函数简写

在很多时候，你可能想在 bind 和 update 时触发相同行为，而不关心其它的钩子

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

### 对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法的 JavaScript 表达式

<div v-demo="{ color: 'white', text: 'hello!' }"></div>

Vue.directive('demo', function (el, binding) {  
console.log(binding.value.color) // => "white"  
console.log(binding.value.text) // => "hello!"  
})

## 渲染函数&JSX

### 基础

Vue 推荐在绝大多数情况下使用 template 来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力，这时你可以用 **render 函数**，它比 template 更接近编译器

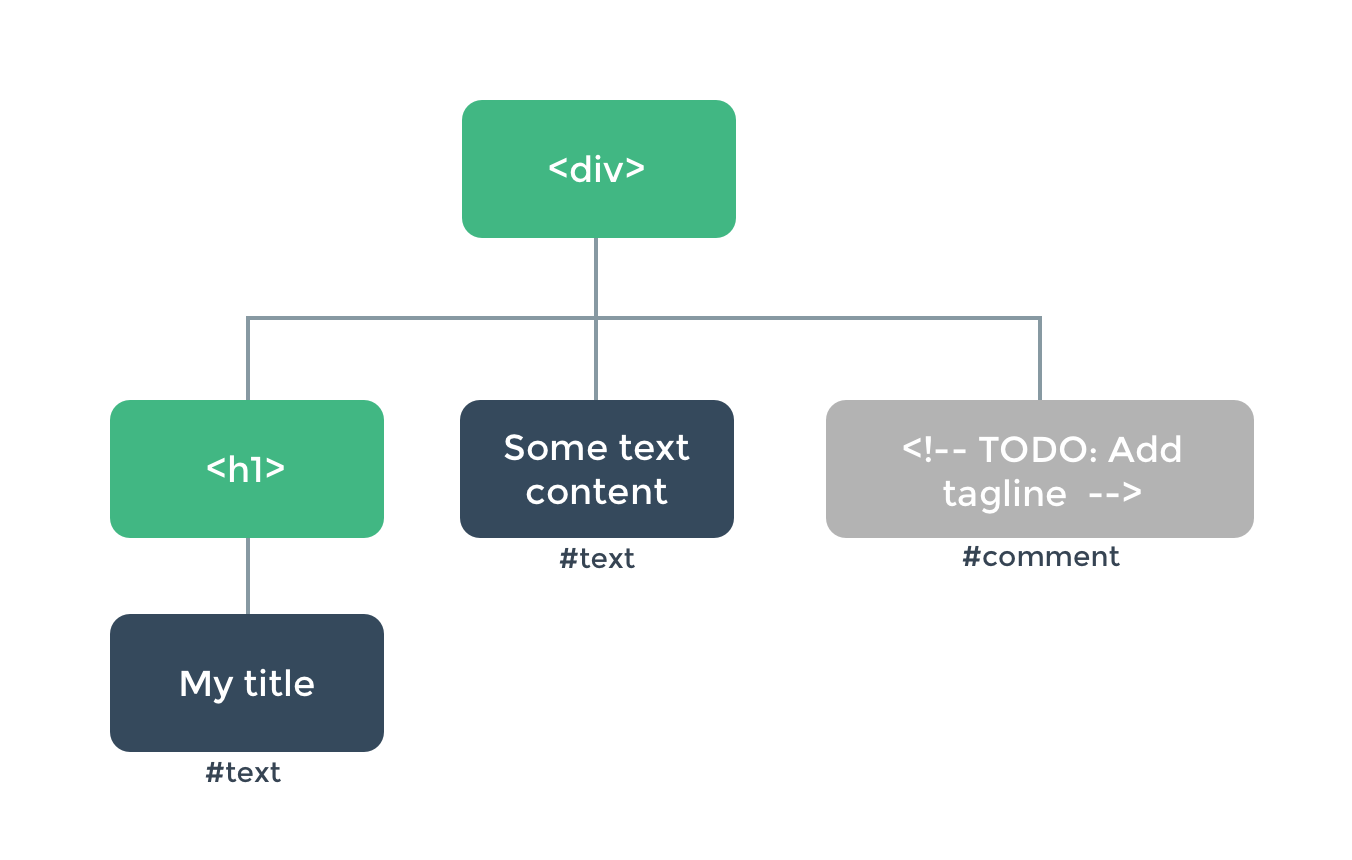
<h1>  
<a name="hello-world" href="#hello-world">  
 Hello world!  
</a>  
</h1>

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

Vue.component('anchored-heading', {  
 render: function (createElement) {  
return createElement(  
'h' + this.level, // 标签名称  
this.$slots.default // 子元素数组  
 )  
 },  
 props: {  
 level: {  
 type: Number,  
 required: true  
 }  
 }  
})

向组件中传递不带 slot 特性的子元素时，比如 anchored-heading 中的 Hello world!，这些子元素被存储在组件实例中的 $slots.default 中

### 节点、树以及虚拟DOM



每个元素都是一个节点。每片文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点 (也就是说每个部分可以包含其它的一些部分)

#### 虚拟DOM

Vue 通过建立一个**虚拟 DOM** 对真实 DOM 发生的变化保持追踪

return createElement('h1', this.blogTitle)

createElement 到底会返回什么呢？其实不是一个实际的 DOM 元素。它更准确的名字可能是 createNodeDescription，因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点，及其子节点。我们把这样的节点描述为“虚拟节点 (Virtual Node)”，也常简写它为“VNode”。“虚拟 DOM”是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼

### createElement参数

// @returns {VNode}  
createElement(  
// {String | Object | Function}  
  // 一个 HTML 标签字符串，组件选项对象，或者  
// 解析上述任何一种的一个 async 异步函数。必需参数。  
  'div',  
  
// {Object}  
  // 一个包含模板相关属性的数据对象  
// 你可以在 template 中使用这些特性。可选参数。  
 {  
    // (详情见下一节)  
 },  
  
// {String | Array}  
  // 子虚拟节点 (VNodes)，由 `createElement()` 构建而成，  
// 也可以使用字符串来生成“文本虚拟节点”。可选参数。  
 [  
'先写一些文字',  
 createElement('h1', '一则头条'),  
 createElement(MyComponent, {  
 props: {  
 someProp: 'foobar'  
 }  
 })  
 ]  
)

#### 深入data对象

{  
// 和`v-bind:class`一样的 API  
// 接收一个字符串、对象或字符串和对象组成的数组  
'class': {  
 foo: true,  
 bar: false  
 },  
// 和`v-bind:style`一样的 API  
// 接收一个字符串、对象或对象组成的数组  
 style: {  
 color: 'red',  
 fontSize: '14px'  
 },  
// 普通的 HTML 特性  
 attrs: {  
 id: 'foo'  
 },  
// 组件 props  
 props: {  
 myProp: 'bar'  
 },  
// DOM 属性  
 domProps: {  
 innerHTML: 'baz'  
 },  
// 事件监听器基于 `on`  
// 所以不再支持如 `v-on:keyup.enter` 修饰器  
// 需要手动匹配 keyCode。  
 on: {  
 click: this.clickHandler  
 },  
  // 仅用于组件，用于监听原生事件，而不是组件内部使用  
// `vm.$emit` 触发的事件。  
 nativeOn: {  
 click: this.nativeClickHandler  
 },  
  // 自定义指令。注意，你无法对 `binding` 中的 `oldValue`  
// 赋值，因为 Vue 已经自动为你进行了同步。  
  directives: [  
 {  
 name: 'my-custom-directive',  
 value: '2',  
 expression: '1 + 1',  
 arg: 'foo',  
 modifiers: {  
 bar: true  
 }  
 }  
 ],  
  // 作用域插槽格式  
// { name: props => VNode | Array<VNode> }  
 scopedSlots: {  
default: props => createElement('span', props.text)  
 },  
  // 如果组件是其他组件的子组件，需为插槽指定名称  
  slot: 'name-of-slot',  
// 其他特殊顶层属性  
 key: 'myKey',  
 ref: 'myRef',  
// 如果你在渲染函数中向多个元素都应用了相同的 ref 名，  
// 那么 `$refs.myRef` 会变成一个数组。  
 refInFor: true  
}

#### 完整示例

var getChildrenTextContent = function (children) {  
return children.map(function (node) {  
return node.children  
 ? getChildrenTextContent(node.children)  
 : node.text  
 }).join('')  
}  
  
Vue.component('anchored-heading', {  
 render: function (createElement) {  
    // 创建 kebab-case 风格的ID  
   var headingId = getChildrenTextContent(this.$slots.default)  
 .toLowerCase()  
 .replace(/\W+/g, '-')  
 .replace(/(^\-|\-$)/g, '')  
  
return createElement(  
'h' + this.level,  
 [  
 createElement('a', {  
 attrs: {  
 name: headingId,  
 href: '#' + headingId  
 }  
 }, this.$slots.default)  
 ]  
 )  
 },  
 props: {  
 level: {  
 type: Number,  
 required: true  
 }  
 }  
})

#### 约束

VNodes必须唯一

所以下面这个无效

render: function (createElement) {  
var myParagraphVNode = createElement('p', 'hi')  
return createElement('div', [  
    // 错误-重复的 VNodes  
    myParagraphVNode, myParagraphVNode  
 ])  
}

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现

render: function (createElement) {  
return createElement('div',  
Array.apply(null, { length: 20 }).map(function () {  
return createElement('p', 'hi')  
 })  
 )  
}

### 使用javascript代替模板功能

#### v-if和 v-for

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的 render 函数就不会提供专有的替代方法

render: function (createElement) {  
if (this.items.length) {  
return createElement('ul', this.items.map(function (item) {  
return createElement('li', item.name)  
 }))  
 } else {  
return createElement('p', 'No items found.')  
 }  
}

#### v-model

render 函数中没有与 v-model 的直接对应 - 你必须自己实现相应的逻辑

render: function (createElement) {  
var self = this  
return createElement('input', {  
 domProps: {  
 value: self.value  
 },  
 on: {  
 input: function (event) {  
 self.$emit('input', event.target.value)  
 }  
 }  
 })  
}

这就是深入底层的代价，但与 v-model 相比，这可以让你更好地控制交互细节。

#### 事件&按键修饰符

| **Modifier(s)** | **Prefix** |
| --- | --- |
| .passive | & |
| .capture | ! |
| .once | ~ |
| .capture.once or .once.capture | ~! |

例如

on: {  
'!click': this.doThisInCapturingMode,  
'~keyup': this.doThisOnce,  
'~!mouseover': this.doThisOnceInCapturingMode  
}

| **Modifier(s)** | **Equivalent in Handler** |
| --- | --- |
| .stop | event.stopPropagation() |
| .prevent | event.preventDefault() |
| .self | if (event.target !== event.currentTarget) return |
| Keys: .enter, .13 | if (event.keyCode !== 13) return (change 13 to [**another key code**](http://keycode.info/) for other key modifiers) |
| Modifiers Keys: .ctrl, .alt, .shift, .meta | if (!event.ctrlKey) return (change ctrlKey to altKey, shiftKey, or metaKey, respectively) |

on: {  
 keyup: function (event) {  
    // 如果触发事件的元素不是事件绑定的元素  
    // 则返回  
    if (event.target !== event.currentTarget) return  
    // 如果按下去的不是 enter 键或者  
    // 没有同时按下 shift 键  
    // 则返回  
    if (!event.shiftKey || event.keyCode !== 13) return  
    // 阻止 事件冒泡  
   event.stopPropagation()  
    // 阻止该元素默认的 keyup 事件  
    event.preventDefault()  
// ...  
 }  
}

#### 插槽

你可以通过 [**this.$slots**](https://cn.vuejs.org/v2/api/#vm-slots) 访问静态插槽的内容，得到的是一个 VNodes 数组

render: function (createElement) {  
// `<div><slot></slot></div>`  
return createElement('div', this.$slots.default)  
}

可以通过 [**this.$scopedSlots**](https://cn.vuejs.org/v2/api/#vm-scopedSlots) 访问作用域插槽，得到的是一个返回 VNodes 的函数：

render: function (createElement) {  
// `<div><slot :text="message"></slot></div>`  
return createElement('div', [  
this.$scopedSlots.default({  
 text: this.message  
 })  
 ])  
}

渲染函数向子组件中传递作用域插槽，可以利用 VNode 数据对象中的 scopedSlots 域

render: function (createElement) {  
return createElement('div', [  
 createElement('child', {  
// 在数据对象中传递 `scopedSlots`  
// 格式：{ name: props => VNode | Array<VNode> }  
 scopedSlots: {  
default: function (props) {  
return createElement('span', props.text)  
 }  
 }  
 })  
 ])  
}

### JSX

import AnchoredHeading from'./AnchoredHeading.vue'  
  
new Vue({  
 el: '#demo',  
 render: function (h) {  
return (  
<AnchoredHeading level={1}>  
<span>Hello</span> world!  
</AnchoredHeading>  
 )  
 }  
})

PS:将 h 作为 createElement 的别名是 Vue 生态系统中的一个通用惯例，实际上也是 JSX 所要求的，如果在作用域中 h 失去作用，在应用中会触发报错。

### 函数式组件

之前创建的锚点标题组件是比较简单，没有管理或者监听任何传递给他的状态，也没有生命周期方法。它只是一个接收参数的函数。  
在这个例子中，我们标记组件为 functional，这意味它是无状态 (没有[**响应式数据**](https://cn.vuejs.org/v2/api/#%E9%80%89%E9%A1%B9-%E6%95%B0%E6%8D%AE))，无实例 (没有 this 上下文)

一个**函数式组件**就像这样：

Vue.component('my-component', {  
 functional: true,  
// Props 可选  
 props: {  
// ...  
 },  
// 为了弥补缺少的实例  
// 提供第二个参数作为上下文  
 render: function (createElement, context) {  
// ...  
 }  
})

PS：**注意：在 2.3.0 之前的版本中，如果一个函数式组件想要接受 props，则 props选项是必须的。在 2.3.0 或以上的版本中，你可以省略 props 选项，所有组件上的特性都会被自动解析为 props。**

在 2.5.0 及以上版本中，如果你使用了[**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html)，那么基于模板的函数式组件可以这样声明：

<template functional>  
</template>

组件需要的一切都是通过上下文传递

1，props：提供所有 prop 的对象

2，children: VNode 子节点的数组

3，slots: 返回所有插槽的对象的函数

4，data：传递给组件的[**数据对象**](https://cn.vuejs.org/v2/guide/render-function.html#%E6%B7%B1%E5%85%A5-data-%E5%AF%B9%E8%B1%A1)，作为 createElement 的第二个参数传入组件

parent：对父组件的引用

5，listeners: (2.3.0+) 一个包含了所有在父组件上注册的事件侦听器的对象。这只是一个指向 data.on 的别名。

6，injections: (2.3.0+) 如果使用了 [**inject**](https://cn.vuejs.org/v2/api/#provide-inject) 选项，则该对象包含了应当被注入的属性。

在添加 functional: true 之后，锚点标题组件的 render 函数之间简单更新增加 context 参数，this.$slots.default 更新为 context.children，之后this.level 更新为 context.props.level

因为函数式组件只是一个函数，所以渲染开销也低很多。然而，对持久化实例的缺乏也意味着函数式组件不会出现在 [**Vue devtools**](https://github.com/vuejs/vue-devtools) 的组件树里

下面是一个依赖传入 props 的值的 smart-list 组件例子

var EmptyList = { /\* ... \*/ }  
var TableList = { /\* ... \*/ }  
var OrderedList = { /\* ... \*/ }  
var UnorderedList = { /\* ... \*/ }  
  
Vue.component('smart-list', {  
 functional: true,  
 props: {  
 items: {  
 type: Array,  
 required: true  
 },  
 isOrdered: Boolean  
 },  
 render: function (createElement, context) {  
functionappropriateListComponent () {  
var items = context.props.items  
  
if (items.length === 0) return EmptyList  
if (typeof items[0] === 'object') return TableList  
if (context.props.isOrdered) return OrderedList  
  
return UnorderedList  
 }  
  
return createElement(  
 appropriateListComponent(),  
 context.data,  
 context.children  
 )  
 }  
})

#### 向子元素或子组件传递特性和事件

然而函数式组件要求你显式定义该行为：

Vue.component('my-functional-button', {  
 functional: true,  
 render: function (createElement, context) {  
// 完全透明的传入任何特性、事件监听器、子结点等。  
return createElement('button', context.data, context.children)  
 }  
})

向 createElement 通过传入 context.data 作为第二个参数，我们就把 my-functional-button 上面所有的特性和事件监听器都传递下去了。事实上这是非常透明的，那些事件甚至并不要求 .native 修饰符

如果你使用基于模板的函数式组件，那么你还需要手动添加特性和监听器。因为我们可以访问到其独立的上下文内容，所以我们可以使用 data.attrs 传递任何 HTML 特性，也可以使用 listeners (即 *data.on* 的别名) 传递任何事件监听器

<template functional>  
<button  
 class="btn btn-primary"  
 v-bind="data.attrs"  
 v-on="listeners"  
>  
<slot/>  
</button>  
</template>

#### Slots和children对比

你可能想知道为什么同时需要 slots() 和 children。slots().default 不是和 children 类似的吗？在一些场景中，是这样，但是如果是函数式组件和下面这样的 children 呢？

<my-functional-component>  
<p slot="foo">  
 first  
</p>  
<p>second</p>  
</my-functional-component>

对于这个组件，children 会给你两个段落标签，而 slots().default 只会传递第二个匿名段落标签，slots().foo 会传递第一个具名段落标签。同时拥有 children和 slots() ，因此你可以选择让组件通过 slot() 系统分发或者简单的通过 children 接收，让其他组件去处理

### 模板编译

Vue 的模板实际是编译成了 render 函数

1，render

function anonymous(

) {

with(this){return \_c('div',[\_m(0),(message)?\_c('p',[\_v(\_s(message))]):\_c('p',[\_v("No message.")])])}

}

StaticRenderFns

\_m(0): function anonymous(

) {

with(this){return \_c('header',[\_c('h1',[\_v("I'm a template!")])])}

}

## 插件

插件通常会为 Vue 添加全局功能。插件的范围没有限制——一般有下面几种

1，添加全局方法或者属性，如: [**vue-custom-element**](https://github.com/karol-f/vue-custom-element)

2，添加全局资源：指令/过滤器/过渡等，如 [**vue-touch**](https://github.com/vuejs/vue-touch)

3，通过全局 mixin 方法添加一些组件选项，如: [**vue-router**](https://github.com/vuejs/vue-router)

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

5，一个库，提供自己的 API，同时提供上面提到的一个或多个功能，如 [**vue-router**](https://github.com/vuejs/vue-router)

### 使用插件

通过全局方法 Vue.use() 使用插件。它需要在你调用 new Vue() 启动应用之前完成：

Vue.use(MyPlugin)  
  
new Vue({  
//... options  
})

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

Vue.use 会自动阻止多次注册相同插件，届时只会注册一次该插件

ue.js 官方提供的一些插件 (例如 vue-router) 在检测到 Vue 是可访问的全局变量时会自动调用 Vue.use()。然而在例如 CommonJS 的模块环境中，你应该始终显式地调用 Vue.use()

// 用 Browserify 或 webpack 提供的 CommonJS 模块环境时  
var Vue = require('vue')  
var VueRouter = require('vue-router')  
  
// 不要忘了调用此方法  
Vue.use(VueRouter)

**PS：**[**awesome-vue**](https://github.com/vuejs/awesome-vue#components--libraries) 集合了来自社区贡献的数以千计的插件和库

### 开发插件

Vue.js 的插件应该有一个公开方法 install。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象

MyPlugin.install = function (Vue, options) {  
// 1. 添加全局方法或属性  
 Vue.myGlobalMethod = function () {  
// 逻辑...  
 }  
  
// 2. 添加全局资源  
 Vue.directive('my-directive', {  
 bind (el, binding, vnode, oldVnode) {  
// 逻辑...  
 }  
 ...  
 })  
  
// 3. 注入组件  
 Vue.mixin({  
 created: function () {  
// 逻辑...  
 }  
 ...  
 })  
  
// 4. 添加实例方法  
 Vue.prototype.$myMethod = function (methodOptions) {  
// 逻辑...  
 }  
}

## 过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：**双花括号插值和 v-bind 表达式** (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示

<!-- 在双花括号中 -->  
{{ message | capitalize }}  
  
<!-- 在 `v-bind` 中 -->  
<div v-bind:id="rawId | formatId"></div>

你可以在一个组件的选项中定义本地的过滤器

filters: {  
 capitalize: function (value) {  
if (!value) return''  
 value = value.toString()  
return value.charAt(0).toUpperCase() + value.slice(1)  
 }  
}

在创建 Vue 实例之前全局定义过滤器

Vue.filter('capitalize', function (value) {  
if (!value) return''  
 value = value.toString()  
return value.charAt(0).toUpperCase() + value.slice(1)  
})

过滤器函数总接收表达式的值 (之前的操作链的结果) 作为第一个参数。在上述例子中，capitalize 过滤器函数将会收到 message 的值作为第一个参数

过滤器可以串联

{{ message | filterA | filterB }}

在这个例子中，filterA 被定义为接收单个参数的过滤器函数，表达式 message 的值将作为参数传入到函数中。然后继续调用同样被定义为接收单个参数的过滤器函数 filterB，将 filterA 的结果传递到 filterB 中

过滤器是 JavaScript 函数，因此可以接收参数

{{ message | filterA('arg1', arg2) }}

filterA 被定义为接收三个参数的过滤器函数。其中 message 的值作为第一个参数，普通字符串 'arg1' 作为第二个参数，表达式 arg2 的值作为第三个参数

# 工具

## 生产环境部署

### 开启生产环境模式

开发环境下，Vue 会提供很多警告来帮你对付常见的错误与陷阱。而在生产环境下，这些警告语句却没有用，反而会增加应用的体积。此外，有些警告检查还有一些小的运行时开销，这在生产环境模式下是可以避免的。

#### 不使用构建工具

如果用 Vue 完整独立版本，即直接用 <script> 元素引入 Vue 而不提前进行构建，请记得在生产环境下使用压缩后的版本 (vue.min.js)。两种版本都可以在[**安装指导**](https://cn.vuejs.org/v2/guide/installation.html#%E7%9B%B4%E6%8E%A5%E7%94%A8-lt-script-gt-%E5%BC%95%E5%85%A5)中找到。

#### 使用构建工具

当使用 webpack 或 Browserify 类似的构建工具时，Vue 源码会根据 process.env.NODE\_ENV 决定是否启用生产环境模式，默认情况为开发环境模式。在 webpack 与 Browserify 中都有方法来覆盖此变量，以启用 Vue 的生产环境模式，同时在构建过程中警告语句也会被压缩工具去除。这些所有 vue-cli 模板中都预先配置好了，但了解一下怎样配置会更好。

webpack

在 webpack 4+ 中，你可以使用 mode 选项

module.exports = {  
 mode: 'production'  
}

但是在 webpack 3 及其更低版本中，你需要使用 [**DefinePlugin**](https://webpack.js.org/plugins/define-plugin/)

var webpack = require('webpack')  
  
module.exports = {  
 // ...  
 plugins: [  
 // ...  
 new webpack.DefinePlugin({  
 'process.env.NODE\_ENV': JSON.stringify('production')  
 })  
 ]  
}

browserify

在运行打包命令时将 NODE\_ENV 设置为 "production"。这等于告诉 vueify 避免引入热重载和开发相关的代码

对打包后的文件进行一次全局的 [**envify**](https://github.com/hughsk/envify) 转换。这使得压缩工具能清除掉 Vue 源码中所有用环境变量条件包裹起来的警告语句

NODE\_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js

在 Gulp 中使用 [**envify**](https://github.com/hughsk/envify)

// 使用 envify 的自定义模块来定制环境变量  
var envify = require('envify/custom')  
  
browserify(browserifyOptions)  
 .transform(vueify)  
 .transform(  
 // 必填项，以处理 node\_modules 里的文件  
 { global: true },  
 envify({ NODE\_ENV: 'production' })  
 )  
 .bundle()

配合 Grunt 和 [**grunt-browserify**](https://github.com/jmreidy/grunt-browserify) 使用 [**envify**](https://github.com/hughsk/envify)

// 使用 envify 自定义模块指定环境变量  
var envify = require('envify/custom')  
  
browserify: {  
 dist: {  
 options: {  
        // 该函数用来调整 grunt-browserify 的默认指令  
        configure: b => b  
 .transform('vueify')  
 .transform(  
            // 用来处理 `node\_modules` 文件  
 { global: true },  
 envify({ NODE\_ENV: 'production' })  
 )  
 .bundle()  
 }  
 }  
}

Rollup

使用 [**rollup-plugin-replace**](https://github.com/rollup/rollup-plugin-replace)

const replace = require('rollup-plugin-replace')  
rollup({  
 // ...  
 plugins: [  
 replace({  
 'process.env.NODE\_ENV': JSON.stringify( 'production' )  
 })  
 ]  
}).then(...)

### 模板预编译

当使用 DOM 内模板或 JavaScript 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法

预编译模板最简单的方式就是使用[**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html)——相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串

如果你使用 webpack，并且喜欢分离 JavaScript 和模板文件，你可以使用 [**vue-template-loader**](https://github.com/ktsn/vue-template-loader)，它也可以在构建过程中把模板文件转换成为 JavaScript 渲染函数

### 提取组件的css

当使用单文件组件时，组件内的 CSS 会以 <style> 标签的方式通过 JavaScript 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式内容闪烁 (fouc)”。将所有组件的 CSS 提取到同一个文件可以避免这个问题，也会让 CSS 更好地进行压缩和缓存。

查阅这个构建工具各自的文档来了解更多

[**webpack + vue-loader**](https://vue-loader.vuejs.org/zh-cn/configurations/extract-css.html) (vue-cli 的 webpack 模板已经预先配置好)

[**Browserify + vueify**](https://github.com/vuejs/vueify#css-extraction)

[**Rollup + rollup-plugin-vue**](https://vuejs.github.io/rollup-plugin-vue/#/en/2.3/?id=custom-handler)

### 跟踪运行时错误

如果在组件渲染时出现运行错误，错误将会被传递至全局 Vue.config.errorHandler配置函数 (如果已设置)。利用这个钩子函数来配合错误跟踪服务是个不错的主意。比如 [**Sentry**](https://sentry.io/)，它为 Vue 提供了[**官方集成**](https://sentry.io/for/vue/)

## 单文件组件

### 介绍

在很多 Vue 项目中，我们使用 Vue.component 来定义全局组件，紧接着用 new Vue({ el: '#container '}) 在每个页面内指定一个容器元素

这种方式在很多中小规模的项目中运作的很好，在这些项目里 JavaScript 只被用来加强特定的视图。但当在更复杂的项目中，或者你的前端完全由 JavaScript 驱动的时候，下面这些缺点将变得非常明显

1，**全局定义 (Global definitions)** 强制要求每个 component 中的命名不得重复

2，**字符串模板 (String templates)** 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 \

3，**不支持 CSS (No CSS support)** 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏

4，**没有构建步骤 (No build step)** 限制只能使用 HTML 和 ES5 JavaScript, 而不能使用预处理器，如 Pug (formerly Jade) 和 Babel

文件扩展名为 .vue 的 **single-file components(单文件组件)** 为以上所有问题提供了解决方法，并且还可以使用 webpack 或 Browserify 等构建工具

#### 怎么看待关注点分离

一个重要的事情值得注意，**关注点分离不等于文件类型分离。**在现代 UI 开发中，我们已经发现相比于把代码库分离成三个大的层次并将其相互交织起来，把它们划分为松散耦合的组件再将其组合起来更合理一些。在一个组件里，其模板、逻辑和样式是内部耦合的，并且把他们搭配在一起实际上使得组件更加内聚且更可维护

即便你不喜欢单文件组件，你仍然可以把 JavaScript、CSS 分离成独立的文件然后做到热重载和预编译

<template>  
 <div>This will be pre-compiled</div>  
</template>  
<script src="./my-component.js"></script>  
<style src="./my-component.css"></style>

### 起步

#### 例子沙箱

如果你希望深入了解并开始使用单文件组件，请来 CodeSandbox [**看看这个简单的 todo 应用**](https://codesandbox.io/s/o29j95wx9)

#### 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

**1，Node Package Manager (NPM)**：阅读 [**Getting Started guide**](https://docs.npmjs.com/getting-started/what-is-npm) 直到 10: Uninstalling global packages章节

**2，Modern JavaScript with ES2015/16**：阅读 Babel 的 [**Learn ES2015 guide**](https://babeljs.io/docs/learn-es2015/)。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

#### 针对高级用户

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**](https://github.com/vuejs/vue-cli) 工具生成新的项目

模板中使用 [**webpack**](https://webpack.js.org/)，一个模块加载器加载多个模块然后构建成最终应用。为了进一步了解 webpack，可以看 [**官方介绍视频**](https://www.youtube.com/watch?v=WQue1AN93YU)。如果你有基础，可以看 [**在 Egghead.io 上的 webpack 进阶教程**](https://egghead.io/courses/using-webpack-for-production-javascript-applications)

## 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**](https://karma-runner.github.io/) 进行自动化测试。它有很多社区版的插件，包括对 [**Webpack**](https://github.com/webpack/karma-webpack) 和 [**Browserify**](https://github.com/Nikku/karma-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**](https://github.com/vuejs-templates/webpack/blob/master/template/test/unit/karma.conf.js) 配置，[**Browserify**](https://github.com/vuejs-templates/browserify/blob/master/template/karma.conf.js) 配置)。

### 简单的断言

你不必为了可测性在组件中做任何特殊的操作，导出原始设置就可以了

<template>  
 <span>{{ message }}</span>  
</template>  
  
<script>  
 export default {  
 data () {  
 return {  
 message: 'hello!'  
 }  
 },  
 created () {  
 this.message = 'bye!'  
 }  
 }  
</script>

然后随着 Vue 导入组件的选项，你可以使用许多常见的断言

// 导入 Vue.js 和组件，进行测试  
import Vue from 'vue'  
import MyComponent from 'path/to/MyComponent.vue'  
  
// 这里是一些 Jasmine 2.0 的测试，你也可以使用你喜欢的任何断言库或测试工具。  
  
describe('MyComponent', () => {  
 // 检查原始组件选项  
 it('has a created hook', () => {  
 expect(typeof MyComponent.created).toBe('function')  
 })  
  
 // 评估原始组件选项中的函数的结果  
 it('sets the correct default data', () => {  
 expect(typeof MyComponent.data).toBe('function')  
 const defaultData = MyComponent.data()  
 expect(defaultData.message).toBe('hello!')  
 })  
  
 // 检查 mount 中的组件实例  
 it('correctly sets the message when created', () => {  
 const vm = new Vue(MyComponent).$mount()  
 expect(vm.message).toBe('bye!')  
 })  
  
 // 创建一个实例并检查渲染输出  
 it('renders the correct message', () => {  
 const Constructor = Vue.extend(MyComponent)  
 const vm = new Constructor().$mount()  
 expect(vm.$el.textContent).toBe('bye!')  
 })  
})

### 编写可被测试的组件

很多组件的渲染输出由它的 props 决定。事实上，如果一个组件的渲染输出完全取决于它的 props，那么它会让测试变得简单，就好像断言不同参数的纯函数的返回值

<template>  
 <p>{{ msg }}</p>  
</template>  
  
<script>  
 export default {  
 props: ['msg']  
 }  
</script>

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

import Vue from 'vue'  
import MyComponent from './MyComponent.vue'  
  
// 挂载元素并返回已渲染的文本的工具函数  
function getRenderedText (Component, propsData) {  
 const Constructor = Vue.extend(Component)  
 const vm = new Constructor({ propsData: propsData }).$mount()  
 return vm.$el.textContent  
}  
  
describe('MyComponent', () => {  
 it('renders correctly with different props', () => {  
 expect(getRenderedText(MyComponent, {  
 msg: 'Hello'  
 })).toBe('Hello')  
  
 expect(getRenderedText(MyComponent, {  
 msg: 'Bye'  
 })).toBe('Bye')  
 })  
})

### 断言异步更新

由于 Vue 进行 [**异步更新 DOM**](https://cn.vuejs.org/v2/guide/reactivity.html#%E5%BC%82%E6%AD%A5%E6%9B%B4%E6%96%B0%E9%98%9F%E5%88%97) 的情况，一些依赖 DOM 更新结果的断言必须在 Vue.nextTick 回调中进行

// 在状态更新后检查生成的 HTML  
it('updates the rendered message when vm.message updates', done => {  
 const vm = new Vue(MyComponent).$mount()  
 vm.message = 'foo'  
  
 // 在状态改变后和断言 DOM 更新前等待一刻  
 Vue.nextTick(() => {  
 expect(vm.$el.textContent).toBe('foo')  
 done()  
 })  
})

关于更深入的 Vue 单元测试的内容，请移步 [**Vue Test Utils**](https://vue-test-utils.vuejs.org/zh/) 以及我们关于 [**Vue 组件的单元测试**](https://cn.vuejs.org/v2/cookbook/unit-testing-vue-components.html)的 cookbook 文章

## TypeScript支持

### 发布为NPM包的官方声明文件

只要你成功地通过 NPM 安装了，就不再需要任何额外的工具辅助，即可在 Vue 中使用 TypeScript 了

### 推荐配置

// tsconfig.json  
{  
 "compilerOptions": {  
 // 与 Vue 的浏览器支持保持一致  
 "target": "es5",  
    // 这可以对 `this` 上的数据属性进行更严格的推断  
 "strict": true,  
    // 如果使用 webpack 2+ 或 rollup，可以利用 tree-shake:  
 "module": "es2015",  
 "moduleResolution": "node"  
 }  
}

注意你需要引入 strict: true (或者至少 noImplicitThis: true，这是 strict模式的一部分) 以利用组件方法中 this 的类型检查，否则它会始终被看作 any 类型。

### 开发工具链

#### 工程创建

[**Vue CLI 3**](https://github.com/vuejs/vue-cli) 可以使用 TypeScript 生成新工程。创建方式

# 1. 如果没有安装 Vue CLI 就先安装  
npm install --global @vue/cli  
  
# 2. 创建一个新工程，并选择 "Manually select features (手动选择特性)" 选项  
vue create my-project-name

#### 编辑器支持

使用 TypeScript 开发 Vue 应用程序，我们强烈建议您使用 [**Visual Studio Code**](https://code.visualstudio.com/)，它为 TypeScript 提供了极好的“开箱即用”支持。如果你正在使用[**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html) (SFC), 可以安装提供 SFC 支持以及其他更多实用功能的 [**Vetur 插件**](https://github.com/vuejs/vetur)

[**WebStorm**](https://www.jetbrains.com/webstorm/) 同样为 TypeScript 和 Vue 提供了“开箱即用”的支持

### 基本用法

要让 TypeScript 正确推断 Vue 组件选项中的类型，您需要使用 Vue.component 或 Vue.extend 定义组件

import Vue from 'vue'  
const Component = Vue.extend({  
 // 类型推断已启用  
})  
  
const Component = {  
 // 这里不会有类型推断，  
 // 因为TypeScript不能确认这是Vue组件的选项  
}

### 基于类的vue组件

如果您在声明组件时更喜欢基于类的 API，则可以使用官方维护的 [**vue-class-component**](https://github.com/vuejs/vue-class-component) 装饰器

mport Vue from 'vue'  
import Component from 'vue-class-component'  
  
// @Component 修饰符注明了此类为一个 Vue 组件  
@Component({  
 // 所有的组件选项都可以放在这里  
 template: '<button @click="onClick">Click!</button>'  
})  
export default class MyComponent extends Vue {  
 // 初始数据可以直接声明为实例的属性  
 message: string = 'Hello!'  
  
 // 组件方法也可以直接声明为实例的方法  
 onClick (): void {  
 window.alert(this.message)  
 }  
}

### 增强类型以配合插件使用

插件可以增加 Vue 的全局/实例属性和组件选项。在这些情况下，在 TypeScript 中制作插件需要类型声明。庆幸的是，TypeScript 有一个特性来补充现有的类型，叫做[**模块补充 (module augmentation)**](https://www.typescriptlang.org/docs/handbook/declaration-merging.html#module-augmentation)

// 1. 确保在声明补充的类型之前导入 'vue'  
import Vue from 'vue'  
  
// 2. 定制一个文件，设置你想要补充的类型  
// 在 types/vue.d.ts 里 Vue 有构造函数类型  
declare module 'vue/types/vue' {  
// 3. 声明为 Vue 补充的东西  
 interface Vue {  
 $myProperty: string  
 }  
}

var vm = new Vue()  
console.log(vm.$myProperty)

可以声明额外的属性和组件选项

import Vue from 'vue'  
  
declare module 'vue/types/vue' {  
  // 可以使用 `VueConstructor` 接口  
  // 来声明全局属性  
 interface VueConstructor {  
    $myGlobal: string  
  }  
}  
  
// ComponentOptions 声明于 types/options.d.ts 之中  
declare module 'vue/types/options' {  
 interface ComponentOptions<V extends Vue> {  
 myOption?: string  
 }  
}

上述的声明允许下面的代码顺利编译通过

// 全局属性  
console.log(Vue.$myGlobal)  
  
// 额外的组件选项  
var vm = new Vue({  
 myOption: 'Hello'  
})

### 标注返回值

因为 Vue 的声明文件天生就具有循环性，TypeScript 可能在推断某个方法的类型的时候存在困难。因此，你可能需要在 render 或 computed 里的方法上标注返回值

import Vue, { VNode } from 'vue'  
  
const Component = Vue.extend({  
 data () {  
 return {  
 msg: 'Hello'  
 }  
 },  
 methods: {  
 // 需要标注有 `this` 参与运算的返回值类型  
 greet (): string {  
 return this.msg + ' world'  
 }  
 },  
 computed: {  
 // 需要标注  
 greeting(): string {  
 return this.greet() + '!'  
 }  
 },  
 // `createElement` 是可推导的，但是 `render` 需要返回值类型  
 render (createElement): VNode {  
 return createElement('div', this.greeting)  
 }  
})

如果你发现类型推导或成员补齐不工作了，标注某个方法也许可以帮助你解决这个问题。使用 --noImplicitAny 选项将会帮助你找到这些未标注的方法

# 规模化

## 路由

### 官方路由

对于大多数单页面应用，都推荐使用官方支持的 [**vue-router 库**](https://github.com/vuejs/vue-router)。更多细节可以看 [**vue-router 文档**](https://router.vuejs.org/)

https://router.vuejs.org/zh/

### 从零开始简单的路由

const NotFound = { template: '<p>Page not found</p>' }  
const Home = { template: '<p>home page</p>' }  
const About = { template: '<p>about page</p>' }  
  
const routes = {  
 '/': Home,  
 '/about': About  
}  
  
new Vue({  
 el: '#app',  
 data: {  
 currentRoute: window.location.pathname  
 },  
 computed: {  
 ViewComponent () {  
 return routes[this.currentRoute] || NotFound  
 }  
 },  
 render (h) { return h(this.ViewComponent) }  
})

### 整合第三方路由

如果有非常喜欢的第三方路由，如 [**Page.js**](https://github.com/visionmedia/page.js) 或者 [**Director**](https://github.com/flatiron/director)，整合[**很简单**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/compare/master...pagejs)。这有个用了 Page.js 的[**复杂示例**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/tree/pagejs)

## 状态管理

### 类flux状态管理的官方实现

由于状态零散地分布在许多组件和组件之间的交互中，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 [**vuex**](https://github.com/vuejs/vuex)：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 [**vue-devtools**](https://github.com/vuejs/vue-devtools)，无需配置即可进行[**时光旅行调试**](https://raw.githubusercontent.com/vuejs/vue-devtools/master/media/demo.gif)

#### React的开发者请参考一下文档

如果你是来自 React 的开发者，你可能会对 Vuex 和 [**Redux**](https://github.com/reactjs/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**](https://yarnpkg.com/en/packages?q=redux%20vue&p=1)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

你有一处需要被多个实例间共享的状态，可以简单地通过维护一份数据来实现共享

const sourceOfTruth = {}  
  
const vmA = new Vue({  
 data: sourceOfTruth  
})  
  
const vmB = new Vue({  
 data: sourceOfTruth  
})

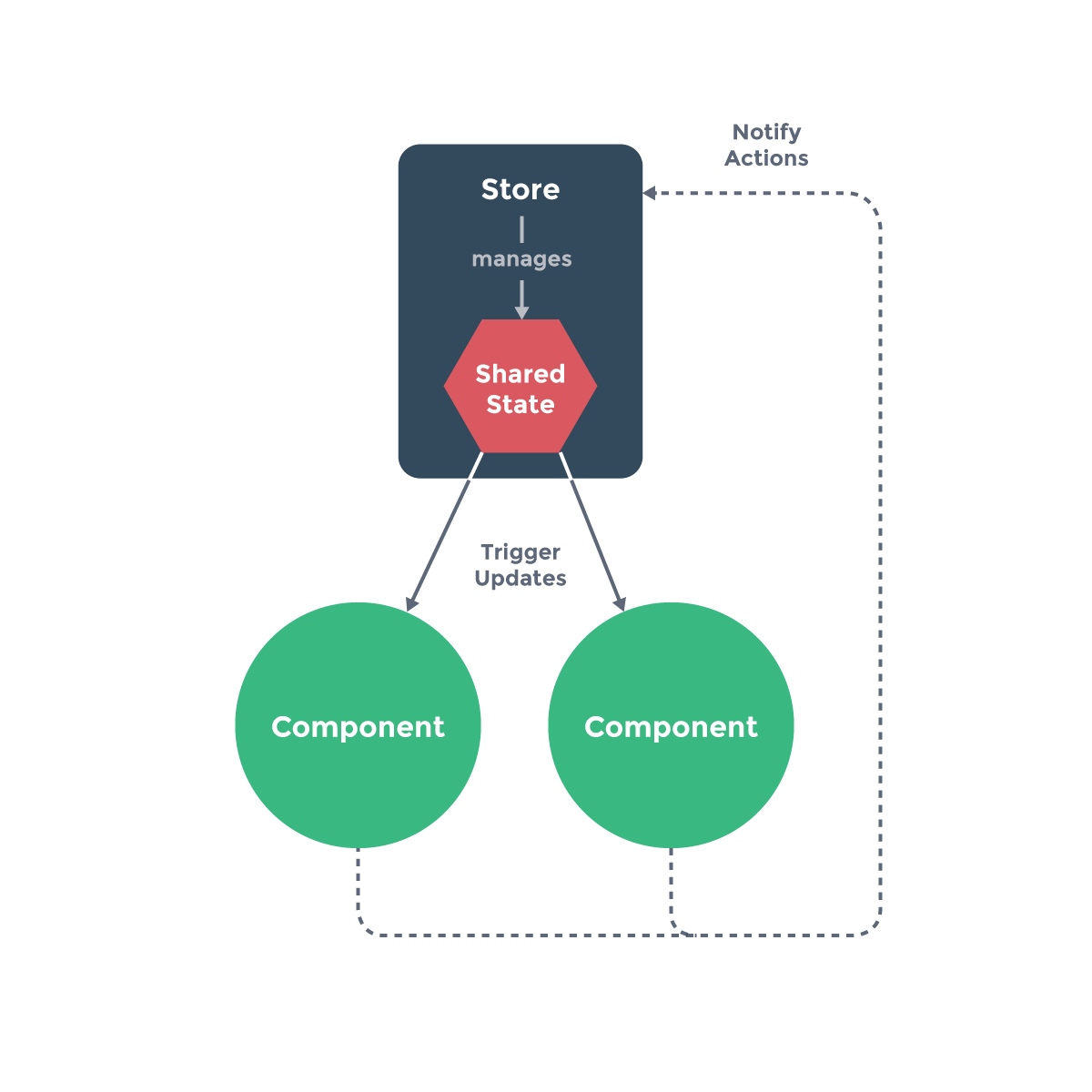
简单的 **store 模式**

var store = {  
 debug: true,  
 state: {  
 message: 'Hello!'  
 },  
 setMessageAction (newValue) {  
 if (this.debug) console.log('setMessageAction triggered with', newValue)  
 this.state.message = newValue  
 },  
 clearMessageAction () {  
 if (this.debug) console.log('clearMessageAction triggered')  
 this.state.message = ''  
 }  
}

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

每个实例/组件仍然可以拥有和管理自己的私有状态

var vmA = new Vue({  
 data: {  
 privateState: {},  
 sharedState: store.state  
 }  
})  
  
var vmB = new Vue({  
 data: {  
 privateState: {},  
 sharedState: store.state  
 }  
})



PS: 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**](https://facebook.github.io/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

## 服务端渲染

### SSR完全指南

在 2.3 发布后我们发布了一份完整的构建 Vue 服务端渲染应用的指南。这份指南非常深入，适合已经熟悉 Vue, webpack 和 Node.js 开发的开发者阅读。请移步 [**ssr.vuejs.org**](https://ssr.vuejs.org/zh/)

### Nuxt.js

从头搭建一个服务端渲染的应用是相当复杂的。幸运的是，我们有一个优秀的社区项目 [**Nuxt.js**](https://nuxtjs.org/) 让这一切变得非常简单。Nuxt 是一个基于 Vue 生态的更高层的框架，为开发服务端渲染的 Vue 应用提供了极其便利的开发体验。更酷的是，你甚至可以用它来做为静态站生成器。推荐尝试

# 内在

## 深入响应式原理

现在是时候深入一下了！Vue 最独特的特性之一，是其非侵入性的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以回避一些常见的问题。在这个章节，我们将进入一些 Vue 响应式系统的底层的细节

### 如何追踪变化

当你把一个普通的 JavaScript 对象传给 Vue 实例的 data 选项，Vue 将遍历此对象所有的属性，并使用 [**Object.defineProperty**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些属性全部转为 [**getter/setter**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects#%E5%AE%9A%E4%B9%89_getters_%E4%B8%8E_setters)

Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器

这些 getter/setter 对用户来说是不可见的，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。这里需要注意的问题是浏览器控制台在打印数据对象时 getter/setter 的格式化并不同，所以你可能需要安装 [**vue-devtools**](https://github.com/vuejs/vue-devtools) 来获取更加友好的检查接口

每个组件实例都有相应的 **watcher** 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新



### 检测变化的注意事项

受现代 JavaScript 的限制 (而且 Object.observe 也已经被废弃)，Vue **不能检测到对象属性的添加或删除**。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化过程，所以属性必须在 data 对象上存在才能让 Vue 转换它，这样才能让它是响应的

var vm = new Vue({  
 data:{  
 a:1  
 }  
})  
  
// `vm.a` 是响应的  
  
vm.b = 2  
// `vm.b` 是非响应的

Vue 不允许在已经创建的实例上动态添加新的根级响应式属性 (root-level reactive property)。然而它可以使用 Vue.set(object, key, value) 方法将响应属性添加到嵌套的对象上

Vue.set(vm.someObject, 'b', 2)

您还可以使用 vm.$set 实例方法，这也是全局 Vue.set 方法的别名

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

有时你想向一个已有对象添加多个属性，例如使用 Object.assign() 或 \_.extend() 方法来添加属性。但是，这样添加到对象上的新属性不会触发更新。在这种情况下可以创建一个新的对象，让它包含原对象的属性和新的属性

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

### 声明响应式属性

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

var vm = new Vue({  
 data: {  
 // 声明 message 为一个空值字符串  
 message: ''  
 },  
 template: '<div>{{ message }}</div>'  
})  
// 之后设置 `message`  
vm.message = 'Hello!'

如果你未在 data 选项中声明 message，Vue 将警告你渲染函数正在试图访问的属性不存在

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例在类型检查系统的帮助下运行的更高效。而且在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的概要，提前声明所有的响应式属性，可以让组件代码在以后重新阅读或其他开发人员阅读时更易于被理解

### 异步更新列队

可能你还没有注意到，Vue **异步**执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作上非常重要。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部尝试对异步队列使用原生的 Promise.then 和 MessageChannel，如果执行环境不支持，会采用 setTimeout(fn, 0) 代替

当你设置 vm.someData = 'new value' ，该组件不会立即重新渲染。当刷新队列时，组件会在事件循环队列清空时的下一个“tick”更新。多数情况我们不需要关心这个过程，但是如果你想在 DOM 状态更新后做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员沿着“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们确实要这么做。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用

<div id="example">{{message}}</div>

var vm = new Vue({  
 el: '#example',  
 data: {  
 message: '123'  
 }  
})  
vm.message = 'new message' // 更改数据  
vm.$el.textContent === 'new message' // false  
Vue.nextTick(function () {  
 vm.$el.textContent === 'new message' // true  
})

在组件内使用 vm.$nextTick() 实例方法特别方便，因为它不需要全局 Vue ，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上

Vue.component('example', {  
 template: '<span>{{ message }}</span>',  
 data: function () {  
 return {  
 message: '没有更新'  
 }  
 },  
 methods: {  
 updateMessage: function () {  
 this.message = '更新完成'  
 console.log(this.$el.textContent) // => '没有更新'  
 this.$nextTick(function () {  
 console.log(this.$el.textContent) // => '更新完成'  
 })  
 }  
 }  
})

因为 $nextTick() 返回一个 Promise 对象，所以你可以使用新的 [**ES2016 async/await**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function)语法完成相同的事情

methods: {  
 async updateMessage: function () {  
 this.message = 'updated'  
 console.log(this.$el.textContent) // => '未更新'  
 await this.$nextTick()  
 console.log(this.$el.textContent) // => '已更新'  
 }  
}