**进入/离开 & 列表过渡**

**概述**

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

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

在这里，我们只会讲到进入、离开和列表的过渡，你也可以看下一节的 [**管理过渡状态**](https://cn.vuejs.org/v2/guide/transitioning-state.html)。

**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：

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

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

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

Toggle

hello

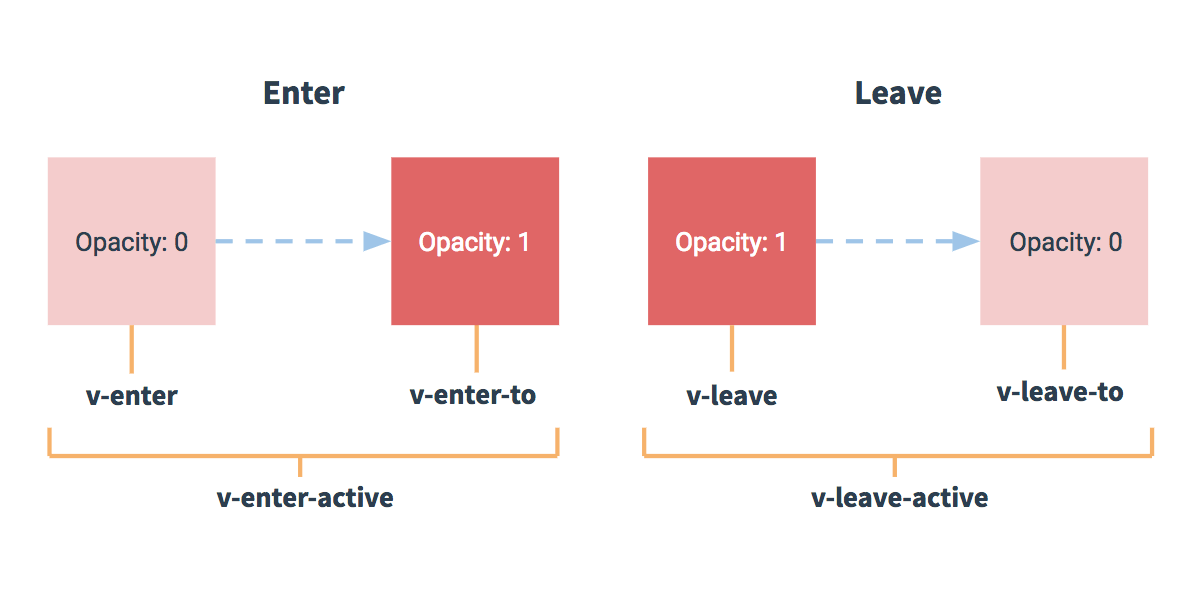
当插入或删除包含在 transition 组件中的元素时，Vue 将会做以下处理：

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-钩子)，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。在元素被插入时生效，在下一个帧移除。
2. v-enter-active：定义过渡的状态。在元素整个过渡过程中作用，在元素被插入时生效，在 transition/animation 完成之后移除。这个类可以被用来定义过渡的过程时间，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。在元素被插入一帧后生效 (于此同时 v-enter 被删除)，在 transition/animation 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生效，在下一个帧移除。
5. v-leave-active：定义过渡的状态。在元素整个过渡过程中作用，在离开过渡被触发后立即生效，在 transition/animation 完成之后移除。这个类可以被用来定义过渡的过程时间，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。在离开过渡被触发一帧后生效 (于此同时 v-leave 被删除)，在 transition/animation 完成之后移除。



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 <transition 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：

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

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

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

Toggle render

hello

**CSS 动画**

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

示例：(省略了兼容性前缀)

|  |
| --- |
| <div id="example-2">  <button @click="show = !show">Toggle show</button>  <transition name="bounce">  <p v-if="show">Look at me!</p>  </transition>  </div> |

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

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

Toggle show

Look at me!

**自定义过渡的类名**

我们可以通过以下特性来自定义过渡类名：

* enter-class
* enter-active-class
* enter-to-class (2.1.8+)
* leave-class
* leave-active-class
* leave-to-class (2.1.8+)

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

示例：

|  |
| --- |
| <link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" type="text/css">  <div id="example-3">  <button @click="show = !show">  Toggle render  </button>  <transition  name="custom-classes-transition"  enter-active-class="animated tada"  leave-active-class="animated bounceOutRight"  >  <p v-if="show">hello</p>  </transition>  </div> |

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

Toggle render

hello

**同时使用过渡和动画**

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

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

**显性的过渡持续时间**

**2.2.0 新增**

在很多情况下，Vue 可以自动得出过渡效果的完成时机。默认情况下，Vue 会等待其在过渡效果的根元素的第一个 transitionend 或 animationend 事件。然而也可以不这样设定——比如，我们可以拥有一个精心编排的一序列过渡效果，其中一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

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

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

你也可以定制进入和移出的持续时间：

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

**JavaScript 钩子**

可以在属性中声明 JavaScript 钩子

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

|  |
| --- |
| // ...  methods: {  // --------  // 进入中  // --------  beforeEnter: function (el) {  // ...  },  // 此回调函数是可选项的设置  // 与 CSS 结合时使用  enter: function (el, done) {  // ...  done()  },  afterEnter: function (el) {  // ...  },  enterCancelled: function (el) {  // ...  },  // --------  // 离开时  // --------  beforeLeave: function (el) {  // ...  },  // 此回调函数是可选项的设置  // 与 CSS 结合时使用  leave: function (el, done) {  // ...  done()  },  afterLeave: function (el) {  // ...  },  // leaveCancelled 只用于 v-show 中  leaveCancelled: function (el) {  // ...  }  } |

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

当只用 JavaScript 过渡的时候，**在 enter 和 leave 中，回调函数 done 是必须的**。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：

|  |
| --- |
| <!--  Velocity works very much like jQuery.animate and is  a great option for JavaScript animations  -->  <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="example-4">  <button @click="show = !show">  Toggle  </button>  <transition  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  v-bind:css="false"  >  <p v-if="show">  Demo  </p>  </transition>  </div> |

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

Toggle

**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

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

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

自定义 JavaScript 钩子：

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

**多个元素的过渡**

我们之后讨论[**多个组件的过渡**](https://cn.vuejs.org/v2/guide/transitions.html#多个组件的过渡)，对于原生标签可以使用 v-if/v-else 。最常见的多标签过渡是一个列表和描述这个列表为空消息的元素：

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

可以这样使用，但是有一点需要注意：

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

示例：

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

在一些场景中，也可以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：

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

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

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

可以重写为：

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

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

**过渡模式**

这里还有一个问题，试着点击下面的按钮：

off

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：

off

然后，我们加上 translate 让它们运动像滑动过渡：

off

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

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

用 out-in 重写之前的开关按钮过渡：

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

off

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：

off

很酷吧？

**多个组件的过渡**

多个组件的过渡简单很多 - 我们不需要使用 key 特性。相反，我们只需要使用[**动态组件**](https://cn.vuejs.org/v2/guide/components.html#动态组件)：

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

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

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

A B

Component A

**列表过渡**

目前为止，关于过渡我们已经讲到：

* 单个节点
* 同一时间渲染多个节点中的一个

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

* 不同于 <transition>，它会以一个真实元素呈现：默认为一个 <span>。你也可以通过 tag 特性更换为其他元素。
* 内部元素 **总是需要** 提供唯一的 key 属性值

**列表的进入/离开过渡**

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

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

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

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

Add Remove

123456789

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

**列表的排序过渡**

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

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

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>  <div id="flip-list-demo" class="demo">  <button v-on:click="shuffle">Shuffle</button>  <transition-group name="flip-list" tag="ul">  <li v-for="item in items" v-bind:key="item">  {{ item }}  </li>  </transition-group>  </div> |

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

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

Shuffle

这个看起来很神奇，内部的实现，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。

我们将之前实现的例子和这个技术结合，使我们列表的一切变动都会有动画过渡。

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>  <div id="list-complete-demo" class="demo">  <button v-on:click="shuffle">Shuffle</button>  <button v-on:click="add">Add</button>  <button v-on:click="remove">Remove</button>  <transition-group name="list-complete" tag="p">  <span  v-for="item in items"  v-bind:key="item"  class="list-complete-item"  >  {{ item }}  </span>  </transition-group>  </div> |

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

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

Shuffle Add Remove

123456789

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

FLIP 动画不仅可以实现单列过渡，多维网格也[**同样可以过渡**](https://jsfiddle.net/chrisvfritz/sLrhk1bc/)：

**Lazy Sudoku**

Keep hitting the shuffle button until you win.

**列表的交错过渡**

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

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="staggered-list-demo">  <input v-model="query">  <transition-group  name="staggered-fade"  tag="ul"  v-bind:css="false"  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  >  <li  v-for="(item, index) in computedList"  v-bind:key="item.msg"  v-bind:data-index="index"  >{{ item.msg }}</li>  </transition-group>  </div> |

|  |
| --- |
| new Vue({  el: '#staggered-list-demo',  data: {  query: '',  list: [  { msg: 'Bruce Lee' },  { msg: 'Jackie Chan' },  { msg: 'Chuck Norris' },  { msg: 'Jet Li' },  { msg: 'Kung Fury' }  ]  },  computed: {  computedList: function () {  var vm = this  return this.list.filter(function (item) {  return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1  })  }  },  methods: {  beforeEnter: function (el) {  el.style.opacity = 0  el.style.height = 0  },  enter: function (el, done) {  var delay = el.dataset.index \* 150  setTimeout(function () {  Velocity(  el,  { opacity: 1, height: '1.6em' },  { complete: done }  )  }, delay)  },  leave: function (el, done) {  var delay = el.dataset.index \* 150  setTimeout(function () {  Velocity(  el,  { opacity: 0, height: 0 },  { complete: done }  )  }, delay)  }  }  }) |



* Bruce Lee
* Jackie Chan
* Chuck Norris
* Jet Li
* Kung Fury

**可复用的过渡**

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将 <transition> 或者 <transition-group> 作为根组件，然后将任何子组件放置在其中就可以了。

使用 template 的简单例子：

|  |
| --- |
| Vue.component('my-special-transition', {  template: '\  <transition\  name="very-special-transition"\  mode="out-in"\  v-on:before-enter="beforeEnter"\  v-on:after-enter="afterEnter"\  >\  <slot></slot>\  </transition>\  ',  methods: {  beforeEnter: function (el) {  // ...  },  afterEnter: function (el) {  // ...  }  }  }) |

函数组件更适合完成这个任务：

|  |
| --- |
| Vue.component('my-special-transition', {  functional: true,  render: function (createElement, context) {  var data = {  props: {  name: 'very-special-transition',  mode: 'out-in'  },  on: {  beforeEnter: function (el) {  // ...  },  afterEnter: function (el) {  // ...  }  }  }  return createElement('transition', data, context.children)  }  }) |

**动态过渡**

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

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

当你想用 Vue 的过渡系统来定义的 CSS 过渡/动画 在不同过渡间切换会非常有用。

所有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。

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="dynamic-fade-demo" class="demo">  Fade In: <input type="range" v-model="fadeInDuration" min="0" v-bind:max="maxFadeDuration">  Fade Out: <input type="range" v-model="fadeOutDuration" min="0" v-bind:max="maxFadeDuration">  <transition  v-bind:css="false"  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  >  <p v-if="show">hello</p>  </transition>  <button  v-if="stop"  v-on:click="stop = false; show = false"  >Start animating</button>  <button  v-else  v-on:click="stop = true"  >Stop it!</button>  </div> |

|  |
| --- |
| new Vue({  el: '#dynamic-fade-demo',  data: {  show: true,  fadeInDuration: 1000,  fadeOutDuration: 1000,  maxFadeDuration: 1500,  stop: true  },  mounted: function () {  this.show = false  },  methods: {  beforeEnter: function (el) {  el.style.opacity = 0  },  enter: function (el, done) {  var vm = this  Velocity(el,  { opacity: 1 },  {  duration: this.fadeInDuration,  complete: function () {  done()  if (!vm.stop) vm.show = false  }  }  )  },  leave: function (el, done) {  var vm = this  Velocity(el,  { opacity: 0 },  {  duration: this.fadeOutDuration,  complete: function () {  done()  vm.show = true  }  }  )  }  }  }) |

Fade In:  Fade Out:

hello

Start animating

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。

# 混合

## 基础

混合 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。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：

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

## 选项合并

当组件和混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 **之前** 调用 ：

|  |
| --- |
| var mixin = {  created: function () {  console.log('混合对象的钩子被调用')  }  }  new Vue({  mixins: [mixin],  created: function () {  console.log('组件钩子被调用')  }  })  // => "混合对象的钩子被调用"  // => "组件钩子被调用" |

值为对象的选项，例如 methods, components 和 directives，将被混合为同一个对象。两个对象键名冲突时，取组件对象的键值对。

|  |
| --- |
| var mixin = {  methods: {  foo: function () {  console.log('foo')  },  conflicting: function () {  console.log('from mixin')  }  }  }  var vm = new Vue({  mixins: [mixin],  methods: {  bar: function () {  console.log('bar')  },  conflicting: function () {  console.log('from self')  }  }  })  vm.foo() // => "foo"  vm.bar() // => "bar"  vm.conflicting() // => "from self" |

注意：Vue.extend() 也使用同样的策略进行合并。

## 全局混合

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 **所有** 之后创建的 Vue 实例。使用恰当时，可以为自定义对象注入处理逻辑。

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

谨慎使用全局混合对象，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**](https://cn.vuejs.org/v2/guide/plugins.html) 以避免产生重复应用

## 自定义选项合并策略

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

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

对于大多数对象选项，可以使用 methods 的合并策略：

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

更多高级的例子可以在 [**Vuex**](https://github.com/vuejs/vuex) 的 1.x 混合策略里找到：

|  |
| --- |
| const merge = Vue.config.optionMergeStrategies.computed  Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {  if (!toVal) return fromVal  if (!fromVal) return toVal  return {  getters: merge(toVal.getters, fromVal.getters),  state: merge(toVal.state, fromVal.state),  actions: merge(toVal.actions, fromVal.actions)  }  } |

# 自定义指令

## 简介

除了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：



当页面加载时，元素将获得焦点 (注意：autofocus 在移动版 Safari 上不工作)。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：

|  |
| --- |
| // 注册一个全局自定义指令 v-focus  Vue.directive('focus', {  // 当绑定元素插入到 DOM 中。  inserted: function (el) {  // 聚焦元素  el.focus()  }  }) |

也可以注册局部指令，组件中接受一个 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：

|  |
| --- |
| <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：所在组件的 VNode 更新时调用，**但是可能发生在其孩子的 VNode 更新之前**。指令的值可能发生了改变也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新 (详细的钩子函数参数见下)。
* componentUpdated：所在组件的 VNode **及其孩子的 VNode** 全部更新时调用。
* 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。
  + **value**：指令的绑定值，例如：v-my-directive="1 + 1", value 的值是 2。
  + **oldValue**：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。
  + **expression**：绑定值的字符串形式。例如 v-my-directive="1 + 1" ，expression 的值是 "1 + 1"。
  + **arg**：传给指令的参数。例如 v-my-directive:foo，arg 的值是 "foo"。
  + **modifiers**：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 }。
* **vnode**：Vue 编译生成的虚拟节点，查阅 [**VNode API**](https://cn.vuejs.org/v2/api/#VNode-接口) 了解更多详情。
* **oldVnode**：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

除了 el 之外，其它参数都应该是只读的，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**](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset) 来进行。

一个使用了这些参数的自定义钩子样例：

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

|  |
| --- |
| Vue.directive('demo', {  bind: function (el, binding, vnode) {  var s = JSON.stringify  el.innerHTML =  'name: ' + s(binding.name) + '<br>' +  'value: ' + s(binding.value) + '<br>' +  'expression: ' + s(binding.expression) + '<br>' +  'argument: ' + s(binding.arg) + '<br>' +  'modifiers: ' + s(binding.modifiers) + '<br>' +  'vnode keys: ' + Object.keys(vnode).join(', ')  }  })  new Vue({  el: '#hook-arguments-example',  data: {  message: 'hello!'  }  }) |

name: "demo"  
value: "hello!"  
expression: "message"  
argument: "foo"  
modifiers: {"a":true,"b":true}  
vnode keys: tag, data, children, text, elm, ns, context, functionalContext, functionalOptions, functionalScopeId, key, componentOptions, componentInstance, parent, raw, isStatic, isRootInsert, isComment, isCloned, isOnce, asyncFactory, asyncMeta, isAsyncPlaceholder

## 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:

|  |
| --- |
| 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类型的 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，这就是 **render 函数**，它比 template 更接近编译器。

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

在 HTML 层，我们决定这样定义组件接口：

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

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

|  |
| --- |
| <script type="text/x-template" id="anchored-heading-template">  <h1 v-if="level === 1">  <slot></slot>  </h1>  <h2 v-else-if="level === 2">  <slot></slot>  </h2>  <h3 v-else-if="level === 3">  <slot></slot>  </h3>  <h4 v-else-if="level === 4">  <slot></slot>  </h4>  <h5 v-else-if="level === 5">  <slot></slot>  </h5>  <h6 v-else-if="level === 6">  <slot></slot>  </h6>  </script> |

|  |
| --- |
| Vue.component('anchored-heading', {  template: '#anchored-heading-template',  props: {  level: {  type: Number,  required: true  }  }  }) |

在这种场景中使用 template 并不是最好的选择：首先代码冗长，为了在不同级别的标题中插入锚点元素，我们需要重复地使用 <slot></slot>。

虽然模板在大多数组件中都非常好用，但是在这里它就不是很简洁的了。那么，我们来尝试使用 render 函数重写上面的例子：

|  |
| --- |
| Vue.component('anchored-heading', {  render: function (createElement) {  return createElement(  'h' + this.level, // tag name 标签名称  this.$slots.default // 子组件中的阵列  )  },  props: {  level: {  type: Number,  required: true  }  }  }) |

简单清晰很多！简单来说，这样代码精简很多，但是需要非常熟悉 Vue 的实例属性。在这个例子中，你需要知道当你不使用 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，**在深入 render 函数之前推荐阅读**[**实例属性 API**](https://cn.vuejs.org/v2/api/#实例属性)**。**

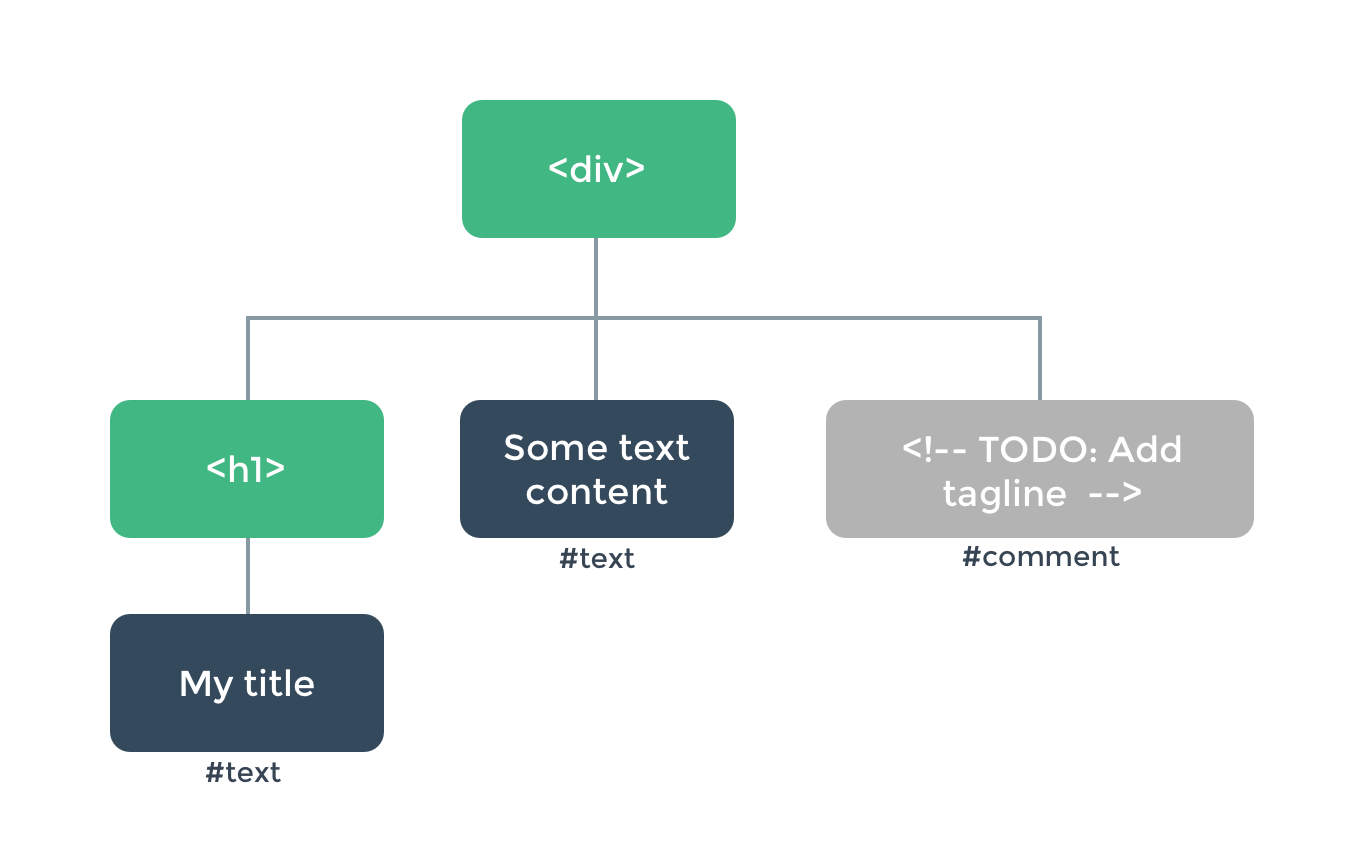
## 节点、树以及虚拟 DOM

在深入渲染函数之前，了解一些浏览器的工作原理是很重要的。以下面这段 HTML 为例：

|  |
| --- |
| <div>  <h1>My title</h1>  Some text content  <!-- TODO: Add tagline -->  </div> |

当浏览器读到这些代码时，它会建立一个[**“DOM 节点”树**](https://javascript.info/dom-nodes)来保持追踪，如同你会画一张家谱树来追踪家庭成员的发展一样。

HTML 的 DOM 节点树如下图所示：



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

高效的更新所有这些节点会是比较困难的，不过所幸你不必再手动完成这个工作了。你只需要告诉 Vue 你希望页面上的 HTML 是什么，这可以是在一个模板里：

|  |
| --- |
| <h1>{{ blogTitle }}</h1> |

或者一个渲染函数里：

|  |
| --- |
| render: function (createElement) {  return createElement('h1', this.blogTitle)  } |

在这两种情况下，Vue 都会自动保持页面的更新，即便 blogTitle 发生了改变。

### 虚拟 DOM

Vue 通过建立一个**虚拟 DOM** 对真实 DOM 发生的变化保持追踪。请近距离看一下这行代码：

|  |
| --- |
| return createElement('h1', this.blogTitle) |

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

## createElement 参数

接下来你需要熟悉的是如何在 createElement 函数中生成模板。这里是 createElement接受的参数：

|  |
| --- |
| // @returns {VNode}  createElement(  // {String | Object | Function}   // 一个 HTML 标签字符串，组件选项对象，或者一个返回值类型为 String/Object 的函数，必要参数   'div',  // {Object}   // 一个包含模板相关属性的数据对象   // 这样，您可以在 template 中使用这些属性。可选参数。  {     // (详情见下一节)  },  // {String | Array}   // 子节点 (VNodes)，由 `createElement()` 构建而成，  // 或使用字符串来生成“文本节点”。可选参数。  [  '先写一些文字',  createElement('h1', '一则头条'),  createElement(MyComponent, {  props: {  someProp: 'foobar'  }  })  ]  ) |

### 深入 data 对象

有一件事要注意：正如在模板语法中，v-bind:class 和 v-bind:style ，会被特别对待一样，在 VNode 数据对象中，下列属性名是级别最高的字段。该对象也允许你绑定普通的 HTML 特性，就像 DOM 属性一样，比如 innerHTML (这会取代 v-html 指令)。

|  |
| --- |
| {  // 和`v-bind:class`一样的 API  'class': {  foo: true,  bar: false  },  // 和`v-bind:style`一样的 API  style: {  color: 'red',  fontSize: '14px'  },  // 正常的 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  },   // 仅对于组件，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。注意事项：不能对绑定的旧值设值   // Vue 会为您持续追踪   directives: [  {  name: 'my-custom-directive',  value: '2',  expression: '1 + 1',  arg: 'foo',  modifiers: {  bar: true  }  }  ],  // Scoped slots in the form of  // { 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'  } |

### 完整示例

有了这些知识，我们现在可以完成我们最开始想实现的组件：

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

### 约束

#### VNodes 必须唯一

组件树中的所有 VNodes 必须是唯一的。这意味着，下面的 render function 是无效的：

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

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现。例如，下面这个例子 render 函数完美有效地渲染了 20 个重复的段落：

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

## 使用 JavaScript 代替模板功能

### v-if 和 v-for

由于使用原生的 JavaScript 来实现某些东西很简单，Vue 的 render 函数没有提供专用的 API。比如，template 中的 v-if 和 v-for：

|  |
| --- |
| <ul v-if="items.length">  <li v-for="item in items">{{ item.name }}</li>  </ul>  <p v-else>No items found.</p> |

这些都会在 render 函数中被 JavaScript 的 if/else 和 map 重写：

|  |
| --- |
| 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 相应的 api - 你必须自己来实现相应的逻辑：

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

这就是深入底层要付出的，尽管麻烦了一些，但相对于 v-model 来说，你可以更灵活地控制。

### 事件 & 按键修饰符

对于 .passive、.capture 和 .once事件修饰符, Vue 提供了相应的前缀可以用于 on：

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

例如:

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

对于其他的修饰符，前缀不是很重要，因为你可以在事件处理函数中使用事件方法：

| **Modifier(s)** | **Equivalent in Handler** |
| --- | --- |
| .stop | event.stopPropagation() |
| .prevent | event.preventDefault() |
| .self | if (event.target !== event.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) |

这里是一个使用所有修饰符的例子：

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

### 插槽

你可以从 [**this.$slots**](https://cn.vuejs.org/v2/api/#vm-slots) 获取 VNodes 列表中的静态内容：

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

还可以从 [**this.$scopedSlots**](https://cn.vuejs.org/v2/api/#vm-scopedSlots) 中获得能用作函数的作用域插槽，这个函数返回 VNodes：

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

如果要用渲染函数向子组件中传递作用域插槽，可以利用 VNode 数据中的 scopedSlots域：

|  |
| --- |
| render (createElement) {  return createElement('div', [  createElement('child', {  // pass `scopedSlots` in the data object  // in the form of { name: props => VNode | Array<VNode> }  scopedSlots: {  default: function (props) {  return createElement('span', props.text)  }  }  })  ])  } |

## JSX

如果你写了很多 render 函数，可能会觉得痛苦：

|  |
| --- |
| createElement(  'anchored-heading', {  props: {  level: 1  }  }, [  createElement('span', 'Hello'),  ' world!'  ]  ) |

特别是模板如此简单的情况下：

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

这就是为什么会有一个 [**Babel 插件**](https://github.com/vuejs/babel-plugin-transform-vue-jsx)，用于在 Vue 中使用 JSX 语法的原因，它可以让我们回到更接近于模板的语法上。

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

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

更多关于 JSX 映射到 JavaScript，阅读 [**使用文档**](https://github.com/vuejs/babel-plugin-transform-vue-jsx#usage)。

## 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 data)，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,  // 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) {  // ...  },  // Props 可选  props: {  // ...  }  }) |

**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，所有组件上的属性都会被自动解析为 props。**

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
* listeners: (2.3.0+) 一个包含了组件上所注册的 v-on 侦听器的对象。这只是一个指向 data.on 的别名。
* injections: (2.3.0+) 如果使用了 [**inject**](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。

在作为包装组件时它们也同样非常有用，比如，当你需要做这些时：

* 程序化地在多个组件中选择一个
* 在将 children, props, data 传递给子组件之前操作它们。

下面是一个依赖传入 props 的值的 smart-list 组件例子，它能代表更多具体的组件：

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

### slots() 和 children 对比

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

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

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

## 模板编译

你可能有兴趣知道，Vue 的模板实际是编译成了 render 函数。这是一个实现细节，通常不需要关心，但如果你想看看模板的功能是怎样被编译的，你会发现会非常有趣。下面是一个使用 Vue.compile 来实时编译模板字符串的简单 demo：



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

## 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-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)

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：

|  |
| --- |
| 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...  }  } |

## 使用插件

通过全局方法 Vue.use() 使用插件：

|  |
| --- |
| // 调用 `MyPlugin.install(Vue)`  Vue.use(MyPlugin) |

也可以传入一个选项对象：

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

Vue.use 会自动阻止多次注册相同插件，届时只会注册一次该插件。

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

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

[**awesome-vue**](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。

# 过滤器

Vue.js 允许你自定义过滤器，可被用作一些常见的文本格式化。过滤器可以用在两个地方：**mustache 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”符指示：

|  |
| --- |
| <!-- in mustaches -->  {{ message | capitalize }}  <!-- in v-bind -->  <div v-bind:id="rawId | formatId"></div> |

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。在这个例子中，capitalize 过滤器函数将会收到 message 的值作为第一个参数。

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

过滤器可以串联：

|  |
| --- |
| {{ message | filterA | filterB }} |

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

过滤器是 JavaScript 函数，因此可以接收参数：

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

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