一 深入响应式原理

1. 如何追踪变化？

2. 变化检测问题

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

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

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

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

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

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

有时你想向已有对象上添加一些属性，例如使用 Object.assign() 或 \_.extend() 方法来添加属性。但是，添加到对象上的新属性不会触发更新。在这种情况下可以创建一个新的对象，让它包含原对象的属性和新的属性：

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

3. 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!' |

如果你在 data 选项中未声明 message，Vue 将警告你渲染函数在试图访问的属性不存在。

4. 异步更新队列

可能你还没有注意到，Vue 异步执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会一次推入到队列中。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 MutationObserver，如果执行环境不支持，会采用 setTimeout(fn, 0) 代替。

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

|  |
| --- |
| <div id="example">{{message}}</div> |
| var vm = new Vue({  el: '#example',  data: {  message: '123'  }  })  vm.message = 'new message' // 更改数据  vm.$el.textContent === 'new message' // false，可以用console.log()输出查看结果  Vue.nextTick(function () {  vm.$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: 'not updated'  }  },  methods: {  updateMessage: function () {  this.message = 'updated'  console.log(this.$el.textContent) // => '没有更新'  this.$nextTick(function () {  console.log(this.$el.textContent) // => '更新完成'  })  }  }  }) |

二 过渡效果

1. 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：

1.1 在 CSS 过渡和动画中自动应用 class

1.2 可以配合使用第三方 CSS 动画库，如 Animate.css

1.3 在过渡钩子函数中使用 JavaScript 直接操作 DOM

1.4 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.

2. 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（进入）/leaving（离开） 过渡

● 条件渲染 （使用 v-if）

● 条件展示 （使用 v-show）

● 动态组件

● 组件根节点

例如：

|  |
| --- |
| <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-active {  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：

● 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。

● 如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。

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

2.1 过渡的-CSS-类名

会有 4 个(CSS)类名在 enter/leave 的过渡中切换

●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。

●v-enter-active: 定义进入过渡的结束状态。在元素被插入时生效，在 transition/animation 完成之后移除。

●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。

●v-leave-active: 定义离开过渡的结束状态。在离开过渡被触发时生效，在 transition/animation 完成之后移除。

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。

2.2 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-active {  transform: translateX(10px);  opacity: 0;  } |

2.3 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。

例如：

|  |
| --- |
| <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-out .5s;  }  @keyframes bounce-in {  0% {  transform: scale(0);  }  50% {  transform: scale(1.5);  }  100% {  transform: scale(1);  }  }  @keyframes bounce-out {  0% {  transform: scale(1);  }  50% {  transform: scale(1.5);  }  100% {  transform: scale(0);  }  } |

2.4 自定义过渡类名

我们可以通过以下特性来自定义过渡类名：

● enter-class

● enter-active-class

● leave-class

● leave-active-class

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

|  |
| --- |
| <link href="https://unpkg.com/animate.css@3.5.1/animate.min.css" 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  }  }) |

2.5 同时使用transitions 和 animations

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

但是，在一些场景中，你需要给同一个元素同时设置两种过渡动效，比如 animation 很快的被触发并完成了，而 transition 效果还没结束。在这种情况中，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。

2.6 JavaScript 钩子

可以在属性中声明 JavaScript 钩子

|  |
| --- |
| <transition  v-on:before-enter="beforeEnter" //插入之前  v-on:enter="enter" //插入  v-on:after-enter="afterEnter" //插入之后  v-on:enter-cancelled="enterCancelled" //取消插入  v-on:before-leave="beforeLeave" //离开之前  v-on:leave="leave" //离开  v-on:after-leave="afterLeave" //离开之后  v-on:leave-cancelled="leaveCancelled" //取消离开  >  <!-- ... -->  </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。

|  |
| --- |
| <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 })  }  }  }) |

3. 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-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"  >  <!-- ... -->  </transition> |

4. 多个元素过渡

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。

|  |
| --- |
| <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 (docState) {  case 'saved': return 'Edit'  case 'edited': return 'Save'  case 'editing': return 'Cancel'  }  }  } |

5. 过渡模式

● 默认：离开和进入同时发生

● 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。

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

6. 多组件过渡

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

|  |
| --- |
| <transition name="component-fade" mode="out-in">  <component v-bind:is="view"></component>  </transition> |
| new Vue({  el: '#transition-components-demo',  data: {  view: 'v-a'  },  components: {  'v-a': {  template: '<div>Component A</div>'  },  'v-b': {  template: '<div>Component B</div>'  }  }  }) |
| .component-fade-enter-active, .component-fade-leave-active {  transition: opacity .3s ease;  }  .component-fade-enter, .component-fade-leave-active {  opacity: 0;  } |

7. 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

7.1 列表的进入和离开过渡

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

7.2 列表的位移过渡

<transition-group> 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 v-move 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;  } |

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

7.3 列表渐进过渡

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

|  |
| --- |
| <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)  }  }  }) |

8. 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。

|  |
| --- |
| 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)  }  }) |

9. 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">  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-on:click="stop = true">Stop it!</button>  </div> |
| new Vue({  el: '#dynamic-fade-demo',  data: {  show: true,  fadeInDuration: 1000,  fadeOutDuration: 1000,  maxFadeDuration: 1500,  stop: false  },  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  }  }  )  }  }  }) |

三 Render函数

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">  <div>  <h1 v-if="level === 1">  <slot></slot>  </h1>  <h2 v-if="level === 2">  <slot></slot>  </h2>  <h3 v-if="level === 3">  <slot></slot>  </h3>  <h4 v-if="level === 4">  <slot></slot>  </h4>  <h5 v-if="level === 5">  <slot></slot>  </h5>  <h6 v-if="level === 6">  <slot></slot>  </h6>  </div>  </script> |
| Vue.component('anchored-heading', {  template: '#anchored-heading-template',  props: {  level: {  type: Number,  required: true  }  }  }) |
| 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  }  }  }) |

1. createElement参数

第二件你需要熟悉的是如何在 createElement 函数中生成模板。这里是 createElement 接受的参数：

|  |
| --- |
| 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)，可以是一个字符串或者一个数组. 可选参数.  [  createElement('h1', 'hello world'),  createElement(MyComponent, {  props: {  someProp: 'foo'  }  }),  'bar'  ]  ) |
| 个人理解: |
| createElement(html标签,值,子节点); |

1.1 深入data object参数

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。

|  |
| --- |
| {  //和v-bind:class 一样的api  “class”:{  foo:’true’,  bar:’false’  },  //和v-bind:style 一样的api  style:{  color:red,  font-size:30px;  },  //正常的html特性  attrs:{  id:’foo’,  title:’鼠标悬浮查看title属性’  },  //组件props  props:{  myProp:”bar”  },  //Dom 属性值  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 => h('span', props.text)  },  // 如果组件是其他组件的子组件，需为slot指定名称  slot: 'name-of-slot'  // 其他特殊顶层属性  key: 'myKey',  ref: 'myRef'  } |
| 完整实例: |

四 自定义指令

1. 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

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

|  |
| --- |
| // 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()  }  })  或者  //局部指令  directives: {  focus: {  // 指令的定义---  }  }  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> |

2. 钩子函数

指令定义函数提供了几个钩子函数（可选）

|  |  |
| --- | --- |
| bind | 只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。 |
| inserted | 被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。 |
| update | 被绑定元素所在的模板更新时调用，而不论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的模板更新（详细的钩子函数参数见下） |
| compentUpdated | 被绑定元素所在模板完成一次更新周期时调用。 |
| unbind | 只调用一次， 指令与元素解绑时调用。 |

接下来我们来看一下钩子函数的参数 (包括 el，binding，vnode，oldVnode) 。

3. 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 | |
| oldVnode | 一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。 | |
| 除了 el 之外，其它参数都应该是只读的，尽量不要修改他们。如果需要在钩子之间共享数据，建议通过元素的 dataset 来进行。 | | |
| 示例 | | |
| <div id="hook-arguments-example" v-demo:hello.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!'  }  }) | | |

1. 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  }) |

4. 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!"  }) |

五 混合

1. 基础

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

|  |
| --- |
| 示例 |
| // 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!" |

2. 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。 |

3. 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 以避免产生重复应用 |

4.自定义选项混合策略

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

|  |
| --- |
| 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 1.x的混合策略里找到:  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)  }  } |

六 插件

1. 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：

◎ 添加全局方法或者属性，如: vue-element

◎ 添加全局资源：指令/过滤器/过渡等，如 vue-touch

◎ 通过全局 mixin方法添加一些组件选项，如: vuex

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

◎ 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

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 (options) {  // 逻辑...  }  } |

2. 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 会自动阻止注册相同插件多次，届时只会注册一次该插件。

一些插件，如 vue-router 如果 Vue 是全局变量则自动调用 Vue.use() 。不过在模块环境中应当始终显式调用 Vue.use() :

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

七 单文件组件

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

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

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

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

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

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

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

这是一个文件名为 Hello.vue 的简单实例：