### 基础

　　Vue.js允许自定义指令，实质上是让你教 Vue一些新技巧：怎样将数据的变化映射到 DOM 的行为。可以使用Vue.directive(id, definition) 的方法传入指令 id 和定义对象来注册一个全局自定义指令。定义对象需要提供一些钩子函数（全部可选）：

bind： 仅调用一次，当指令第一次绑定元素的时候。

update： 第一次是紧跟在 bind 之后调用，获得的参数是绑定的初始值；以后每当绑定的值发生变化就会被调用，获得新值与旧值两个参数。

unbind：仅调用一次，当指令解绑元素的时候。

1. Vue.directive('my-directive', {
2. bind: function () {
3. // 准备工作
4. // 例如，添加事件处理器或只需要运行一次的高耗任务
5. },
6. update: function (newValue, oldValue) {
7. // 值更新时的工作
8. // 也会以初始值为参数调用一次
9. },
10. unbind: function () {
11. // 清理工作
12. // 例如，删除 bind() 添加的事件监听器
13. }
14. })

在注册之后，便可以在 Vue.js 模板中这样用（记着添加前缀 v-）：

1. <div v-my-directive="someValue"></div>

当只需要 update 函数时，可以传入一个函数替代定义对象：

1. Vue.directive('my-directive', function (value) {
2. // 这个函数用作 update()
3. })

js代码如下：

1. Vue.directive('my-directive', {
2. bind: function () {
3. console.log('my-directive')
4. },
5. update: function (value) {
6. this.el.innerHTML ='value - '+ value
7. }
8. })
9. var demo = new Vue({
10. el: '#demo',
11. data: {
12. my-directive: 'hello!'
13. }
14. })

### 指令实例属性

　　所有的钩子函数将被复制到实际的指令对象中，钩子内 this 指向这个指令对象。这个对象暴露了一些有用的属性：

　　el: 指令绑定的元素。  
　　vm: 拥有该指令的上下文 ViewModel。  
　　expression: 指令的表达式，不包括参数和过滤器。  
　　arg: 指令的参数。  
　　name: 指令的名字，不包含前缀。  
　　modifiers: 一个对象，包含指令的修饰符。  
　　descriptor: 一个对象，包含指令的解析结果。

1. <div id="demo" v-demo:hello.a.b="msg"></div>

js代码：

1. Vue.directive('demo', {
2. bind: function () {
3. console.log('demo bound!')
4. },
5. update: function (value) {
6. this.el.innerHTML =
7. 'name - ' + this.name + '<br>' +
8. 'expression - ' + this.expression + '<br>' +
9. 'argument - ' + this.arg + '<br>' +
10. 'modifiers - ' + JSON.stringify(this.modifiers) + '<br>' +
11. 'value - ' + value
12. }
13. })
14. var demo = new Vue({
15. el: '#demo',
16. data: {
17. msg: 'hello!'
18. }
19. })

### 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，指令可以使用任意合法的 JavaScript 表达式：

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

js代码如下：

1. Vue.directive('demo', function (value) {
2. document.write(value.color+" "+value.text);
3. })

### 字面修饰符

　　当指令使用了字面修饰符literal，它的值将按普通字符串处理并传递给 update 方法。update 方法将只调用一次，因为普通字符串不能响应数据变化。

1. <div id="demo" v-demo.literal="foo bar baz">

js代码如下：

1. Vue.directive('demo', function (value) {
2. document.write(value) // "foo bar baz"
3. })

### 元素指令

　　有时候，我们可能想要我们的指令可以以自定义元素的形式被使用，而不是作为一个特性。这与Angular 的 E 类指令的概念非常相似。元素指令可以看做是一个轻量的自定义组件（后面会讲到）。你可以像下面这样注册一个自定义的元素指令：

1. Vue.elementDirective('my-directive', {
2. // API 同普通指令
3. bind: function () {
4. // 操作 this.el...
5. }
6. })

页面应用为：

1. <my-directive></my-directive>

### 基础

　　可以用全局方法 Vue.filter() 注册一个自定义过滤器，它接收两个参数：过滤器ID和过滤器函数。过滤器函数以值为参数，返回转换后的值：

1. Vue.filter('split0', function (value) {
2. return value.split('.')[0];
3. })

html代码：

1. <div id="demo">{{msg|split0}}</div>

js代码：

1. Vue.filter('split0', function (value) {
2. return value.split('.')[0];
3. })
4. var vm=new Vue({
5. el: '#example',
6. data:{
7. msg:'2.0'}
8. })

过滤器函数可以接收任意数量的参数：

1. Vue.filter('wrap', function (value, begin, end) {
2. return begin + value + end;
3. })
4. <!-- 'hello' => 'before hello after' -->
5. <span>{{msg | wrap 'before' 'after'}}</span>

### 双向过滤器

　　我们使用过滤器都是在把来自模型的值显视在视图之前转换它。不过也可以定义一个过滤器，在把来自视图（< input > 元素）的值写回模型之前转化它，如下代码：

1. Vue.filter('currencyDisplay', {
2. // model -> view
3. // 在更新 `<input>` 元素之前格式化值
4. read: function(val) {
5. return '￥'+val.toFixed(2)
6. },
7. // view -> model
8. // 在写回数据之前格式化值
9. write: function(val, oldVal) {
10. var number = +val.replace(/[^\d.]/g, '')
11. return isNaN(number) ? 0 : parseFloat(number.toFixed(2))
12. }
13. })

html代码如下：

1. <div id="example">
2. <input type="text" v-model="msg|currencyDisplay"><br>
3. <span>{{msg}}</span>
4. </div>

### 动态参数

　　如果一个过滤器参数没有被引号包裹，它会在当前 vm 的数据作用域里当做表达式进行动态求值。此外，过滤器函数的this 上下文永远是调用它的当前 vm。

1. <input v-model="userInput">
2. <span>{{msg | concat userInput}}</span>

js代码：

1. Vue.filter('concat', function (value, input) {
2. // 这里 `input` === `this.userInput`
3. return value + input
4. })

### 方法处理器

　　可以用 v-on 指令监听 DOM 事件：

1. <div id="example">
2. <button v-on:click="greet">Greet</button>
3. </div>

我们绑定了一个单击事件处理器到一个方法 greet。下面在 Vue 实例中定义这个方法：

1. var vm = new Vue({
2. el: '#example',
3. data: {
4. name: 'Vue.js'
5. },
6. // 在 'methods' 对象中定义方法
7. methods: {
8. greet: function (event) {
9. // 方法内 'this'指向 vm
10. alert('Hello ' + this.name + '!')
11. // 'event' 是原生 DOM 事件
12. alert(event.target.tagName)
13. }
14. }
15. })

### 内联语句处理器

　　除了直接绑定到一个方法，也可以用内联 JavaScript 语句：

1. <div id="example-2">
2. <button v-on:click="say('hi')">Say Hi</button>
3. <button v-on:click="say('what')">Say What</button>
4. </div>

js代码:

1. new Vue({
2. el: '#example',
3. methods: {
4. say: function (msg) {
5. alert(msg)
6. }
7. }
8. })

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

1. <button v-on:click="say('hello!', $event)">Submit</button>

js代码：

1. new Vue({
2. el: '#example',
3. methods: {
4. say: function (msg, event) {
5. // 现在我们可以访问原生事件对象
6. event.preventDefault()
7. }
8. }
9. })

### 事件修饰符

　　在事件处理器中经常需要调用 event.preventDefault() 或 event.stopPropagation()。尽管我们在方法内可以轻松做到，不过让方法是纯粹的数据逻辑而不处理 DOM 事件细节会更好。为了解决这个问题，Vue.js 为 v-on 提供两个事件修饰符：.prevent 与 .stop。

1. <!-- 阻止单击事件冒泡 -->
2. <a v-on:click.stop="doThis"></a>
3. <!-- 提交事件不再重载页面 -->
4. <form v-on:submit.prevent="onSubmit"></form>
5. <!-- 修饰符可以串联 -->
6. <a v-on:click.stop.prevent="doThat"></a>
7. <!-- 只有修饰符 -->
8. <form v-on:submit.prevent></form>

### 按键修饰符

在监听键盘事件时，我们经常需要检测 keyCode。Vue.js允许为v-on添加按键修饰符：

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

记住所有的 keyCode 比较困难，Vue.js 为最常用的按键提供别名：

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

常用按键别名：enter、tab、delete、esc、space、up、down、left、right。

### 简介

　　通过 Vue.js 的过渡系统，你可以轻松的为 DOM 节点被插入/移除的过程添加过渡动画效果。Vue 将会在适当的时机添加/移除 CSS 类名来触发 CSS3 过渡/动画效果，你也可以提供相应的 JavaScript 钩子函数在过渡过程中执行自定义的 DOM 操作。

　　以 v-transition="my-transition 这个指令为例，当带有这个指令的 DOM 节点被插入或移除时，Vue 将会：

　　1.用 my-transition这个 ID 去查找是否有注册过的 JavaScript 钩子对象。这个对象可以是由 Vue.transition(id, hooks) 全局注册，或是通过 transitions 选项定义在当前的组件内部。如果找到此对象，则会在过渡动画不同的阶段调用相应的钩子。

　　2.自动探测目标元素是否应用了 CSS 过渡效果或者动画效果，并在适当的时机添加/移除 CSS 类名。

　　3.如果没有提供 JavaScript 钩子函数，也没有检测到相应的 CSS 过渡/动画效果，DOM 的插入/移除会在下一帧立即执行。

1. <div v-if="show" transition="my-transition"></div>

　　所有的 Vue.js 过渡效果只有在该 DOM 操作是通过 Vue.js 触发时才会生效。触发的方式可以是通过内置指令，比如v-if，或是通过 Vue 实例的方法，比如 vm.$appendTo()。

### CSS 过渡

一个典型的 CSS 过渡效果定义如下：

1. <div v-if="show" v-transition="expand">hello</div>

然后为.expand-transition，.expand-enter 和.expand-leave 添加 CSS 规则:

1. /\* 必需 \*/
2. .expand-transition {
3. transition: all .3s ease;
4. height: 30px;
5. padding: 10px;
6. background-color: #eee;
7. overflow: hidden;
8. }
9. /\* .expand-enter 定义进入的开始状态 \*/
10. /\* .expand-leave 定义离开的结束状态 \*/
11. .expand-enter, .expand-leave {
12. height: 0;
13. padding: 0 10px;
14. opacity: 0;
15. }

另外，可以提供 JavaScript 钩子:

1. Vue.transition('expand', {
2. beforeEnter: function (el) {
3. el.textContent = 'beforeEnter'
4. },
5. enter: function (el) {
6. el.textContent = 'enter'
7. },
8. afterEnter: function (el) {
9. el.textContent = 'afterEnter'
10. },
11. enterCancelled: function (el) {
12. // handle cancellation
13. },
14. beforeLeave: function (el) {
15. el.textContent = 'beforeLeave'
16. },
17. leave: function (el) {
18. el.textContent = 'leave'
19. },
20. afterLeave: function (el) {
21. el.textContent = 'afterLeave'
22. },
23. leaveCancelled: function (el) {
24. // handle cancellation
25. }
26. })

　　说简单点就是我们在操作某个元素时，当v-if的值为发生变化时，就会引起transition的变化，从而使的CSS样式改变。

### 过渡的CSS类名

　　类名的添加和切换取决于 transition特性的值。比如 transition="fade"，会有三个 CSS 类名：

　　1）.fade-transition 始终保留在元素上。

　　2）.fade-enter 定义进入过渡的开始状态。只应用一帧然后立即删除。

　　3）.fade-leave 定义离开过渡的结束状态。在离开过渡开始时生效，在它结束后删除。

　　如果 transition 特性没有值，类名默认是 .v-transition, .v-enter 和 .v-leave。

### 过渡流程详解

　　根据我们第一节中的代码来讲解过渡流程：

　　当 show 属性改变时，Vue.js 将相应地插入或删除div元素，按照如下规则改变过渡的 CSS 类名：

　　如果 show 变为 false，Vue.js 将：

1. 用 beforeLeave 钩子；
2. 添加 v-leave 类名到元素上以触发过渡；
3. 调用 leave 钩子；
4. 等待过渡结束（监听 transitionend 事件）；
5. 从 DOM 中删除元素并删除 v-leave 类名；
6. 调用 afterLeave 钩子。

　　如果 show 变为 true，Vue.js 将：

1. 调用 beforeEnter 钩子；
2. 添加 v-enter 类名到元素上；
3. 把它插入 DOM；
4. 调用 enter 钩子；
5. 强制一次CSS 布局，让 v-enter 确实生效。然后删除 v-enter 类名，以触发过渡，回到元素的原始状态；
6. 等待过渡结束；
7. 调用afterEnter钩子。

### 渐近过渡

　　transition 与 v-for 一起用时可以创建渐近过渡。给过渡元素添加一个特性 stagger, enter-stagger 或 leave-stagger， 如下代码：

1. <div v-for="list" transition stagger="100"></div>

或者，提供一个钩子 stagger, enter-stagger 或 leave-stagger，以更好的控制. 如下代码：

1. Vue.transition('stagger', {
2. stagger: function (index) {
3. // 每个过渡项目增加 50ms 延时
4. // 但是最大延时限制为 300ms
5. return Math.min(300, index \* 50)
6. }
7. })

如下实例html代码：

1. <div id="demo">
2. <input v-model="query">
3. <ul>
4. <li v-for="item in list | filterBy query"
5. transition="staggered"
6. stagger="100">
7. {{item.msg}}
8. </li>
9. </ul>
10. </div>

js代码：

1. new Vue({
2. el: '#demo',
3. data: {
4. query: '',
5. list: [
6. { msg: 'Bruce Lee' },
7. { msg: 'Jackie Chan' },
8. { msg: 'Chuck Norris' },
9. { msg: 'Jet Li' },
10. { msg: 'Kung Fury' }
11. ]
12. }
13. })

CSS代码：

1. ul {
2. padding-left: 0;
3. font-family: Helvetica, Arial, sans-serif;
4. }
5. .staggered-transition {
6. transition: all .5s ease;
7. overflow: hidden;
8. margin: 0;
9. height: 20px;
10. }
11. .staggered-enter, .staggered-leave {
12. opacity: 0;
13. height: 0;
14. }

以上代码的效果为:当我们在文本框中输入值时，会看到列表逐渐消失，删除内容时，会看到列表显示。

### 使用组件

　　在Vue中，可以用 Vue.extend() 创建一个组件构造器：

1. var MyComponent = Vue.extend({
2. template:'..........' //选项
3. })

要把这个构造器用作组件，需要用 Vue.component(tag, constructor) 注册 ：

1. // 全局注册组件，tag 为 my-component
2. Vue.component('my-component', MyComponent)

　　在注册之后，组件便可以用在父实例的模块中，以自定义元素 < my-component > 的形式使用。要确保在初始化根实例之前注册了组件：

1. <div id="example">
2. <my-component></my-component>
3. </div>

js代码如下：

1. // 定义
2. var MyComponent = Vue.extend({
3. template: '<div>A custom component!</div>'
4. })
5. // 注册
6. Vue.component('my-component', MyComponent)
7. // 创建根实例
8. new Vue({
9. el: '#example'
10. })

渲染为：

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

### Props 传递数据

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

“prop” 是组件数据的一个字段，期望从父组件传下来。子组件需要显式地用 props 选项声明 props：

1. Vue.component('child', {
2. // 声明 props
3. props: ['msg'],
4. // prop 可以用在模板内
5. // 可以用 `this.msg` 设置
6. template: '<span>{{ msg }}</span>'
7. })

然后向它传入一个普通字符串：

1. <child msg="hello!"></child>

### 动态Props

　　HTML 特性不区分大小写。名字形式为 camelCase 的 prop 用作特性时，需要转为 kebab-case（短横线隔开）：

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

　　类似于绑定一个普通的特性到一个表达式，也可以用 v-bind 绑定动态 Props 到父组件的数据。每当父组件的数据变化时，也会传递给子组件：

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

### Prop 绑定类型

　　prop 默认是单向绑定：当父组件的属性变化时，将传导给子组件，但是反过来不会。这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。不过，也可以使用 .sync 或 .once 绑定修饰符显式地强制双向或单次绑定。

1. <!-- 默认为单向绑定 -->
2. <child v-bind:my-message="parentMsg"></child>
3. <!-- 双向绑定 -->
4. <child v-bind:my-message.sync="parentMsg"></child>
5. <!-- 单次绑定 -->
6. <child v-bind:my-message.once="parentMsg"></child>

　　双向绑定会把子组件的 my-message 属性同步回父组件的 parentMsg 属性。单次绑定在建立之后不会同步之后的变化。

### 自定义事件

　　Vue 实例实现了一个自定义事件接口，用于在组件树中通信。这个事件系统独立于原生 DOM 事件，做法也不同。

　　每个 Vue 实例都是一个事件触发器：

* 使用 $on() 监听事件；
* 使用 $emit() 在它上面触发事件；
* 使用 $dispatch() 派发事件，事件沿着父链冒泡；
* 使用 $broadcast() 广播事件，事件向下传导给所有的后代。

子组件可以用 this.$parent 访问它的父组件,而父组件有一个数组 this.$children，包含它所有的子元素。

1. <!-- 子组件模板 -->
2. <template id="child-template">
3. <input v-model="msg">
4. <button v-on:click="notify">Dispatch Event</button>
5. </template>
6. <!-- 父组件模板 -->
7. <div id="events-example">
8. <p>Messages: {{ messages | json }}</p>
9. <child></child>
10. </div>
11. // 注册子组件
12. // 将当前消息派发出去
13. Vue.component('child', {
14. template: '#child-template',
15. data: function () {
16. return { msg: 'hello' }
17. },
18. methods: {
19. notify: function () {
20. if (this.msg.trim()) {
21. this.$dispatch('child-msg', this.msg)
22. this.msg = ''
23. }
24. }
25. }
26. })
27. // 启动父组件
28. // 将收到消息时将事件推入一个数组
29. var parent = new Vue({
30. el: '#events-example',
31. data: {
32. messages: []
33. },
34. // 在创建实例时 `events` 选项简单地调用 `$on`
35. events: {
36. 'child-msg': function (msg) {
37. // 事件回调内的 `this` 自动绑定到注册它的实例上
38. this.messages.push(msg)
39. }
40. }
41. })