Vue官方文档学习知识点

# 列表渲染

如果你想添加新的响应式属性，不要像这样：

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

你应该这样做：（这种已经在项目中运用）

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

# 显示过滤/排序结果

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

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

在计算属性不适用的情况下 (例如，在嵌套 v-for 循环中)

<li v-for="n in even(numbers)">{{ n }}</li>

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

# v-for with v-if

当他们处于同一个节点时，v-for的优先级高于v-if，这意味着v-if将分别重复运行于每个v-for中

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

注意：2.2.0+ 的版本里，当在组件中使用 **v-for** 时，**key** 现在是必须的。

为了把迭代数据传递到组件里，我们要用 props（这个例子要多理解\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*）

<div id="todo-list-example">  
 <form v-on:submit.prevent="addNewTodo">  
 <label for="new-todo">Add a todo</label>  
 <input  
 v-model="newTodoText"  
 id="new-todo"  
 placeholder="E.g. Feed the cat"  
 >  
 <button>Add</button>  
 </form>  
 <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>  
</div>

注意：这里的 is="todo-item" 属性。这种做法在使用 DOM 模板时是十分必要的，因为在 <ul> 元素内只有 <li> 元素会被看作有效内容。这样做实现的效果与 <todo-item> 相同，但是可以避开一些潜在的浏览器解析错误。

Vue.component('todo-item', {  
 template: '\  
 <li>\  
 {{ title }}\  
 <button v-on:click="$emit(\'remove\')">Remove</button>\  
 </li>\  
 ',  
 props: ['title']  
})  
  
new Vue({  
 el: '#todo-list-example',  
 data: {  
 newTodoText: '',  
 todos: [  
 {  
 id: 1,  
 title: 'Do the dishes',  
 },  
 {  
 id: 2,  
 title: 'Take out the trash',  
 },  
 {  
 id: 3,  
 title: 'Mow the lawn'  
 }  
 ],  
 nextTodoId: 4  
 },  
 methods: {  
 addNewTodo: function () {  
 this.todos.push({  
 id: this.nextTodoId++,  
 title: this.newTodoText  
 })  
 this.newTodoText = ''  
 }  
 }  
})

知识点：

父组件里面的v-model="newTodoText"

# 事件处理方法

<div id="example-2">  
 <!-- `greet` 是在下面定义的方法名 -->  
 <button v-on:click="greet">Greet</button>  
</div>

var example2 = new Vue({  
 el: '#example-2',  
 data: {  
 name: 'Vue.js'  
 },  
 // 在 `methods` 对象中定义方法  
 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)  
 }  
 }  
 }  
})

理解event事件

# vue click.stop阻止点击事件继续传播

<div id="app">

<div v-on:click="dodo">

<button v-on:click="doThis">阻止单击事件继续传播</button>

</div>

</div>

<script>

var app = new Vue({

el: "#app",

data: {

name: "Vue.js"

},

methods: {

doThis: function () {

alert("noclick");

},

dodo: function () {

alert("dodo");

}

}

});

</script>

将会先弹出“noclick”,再弹出“dodo”。

<div id="app">

<div v-on:click="dodo">

<button v-on:click.stop="doThis">阻止单击事件继续传播</button>

</div>

</div>

<script>

var app = new Vue({

el: "#app",

data: {

name: "Vue.js"

},

methods: {

doThis: function () {

alert("noclick");

},

dodo: function () {

alert("dodo");

}

}

});

</script>

将只弹出‘noclick’

# 修饰符

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

请查看该文章：<https://blog.csdn.net/hhf235678/article/details/78595088>

# 组件基础

为每个相关的信息定义一个 prop 会变得很麻烦：

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

所以是时候重构一下这个 <blog-post> 组件了，让它变成接受一个单独的 post prop：

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

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

# 在组件上使用v-model

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

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

为了让它正常工作，这个组件内的 <input> 必须：

* 将其 value 特性绑定到一个名叫 value 的 prop 上
* 在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

# 解析DOM模板时的注意

<table>  
 <blog-post-row></blog-post-row>  
</table>

这个自定义组件 <blog-post-row> 会被作为无效的内容提升到外部，并导致最终渲染结果出错。幸好这个特殊的 is 特性给了我们一个变通的办法：

<table>  
 <tr is="blog-post-row"></tr>  
</table>

# 单项数据流

所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

# 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

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

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>  
 `  
})

# 自定义组件的v-model

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

Vue.component('base-checkbox', {  
 model: {  
 prop: 'checked',  
 event: 'change'  
 },  
 props: {  
 checked: Boolean  
 },  
 template: `  
 <input  
 type="checkbox"  
 v-bind:checked="checked"  
 v-on:change="$emit('change', $event.target.checked)"  
 >  
 `  
})

<base-checkbox v-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。

# .sync修饰符--‘prop’进行‘双向绑定’

this.$emit('update:title', newTitle)

<text-document  
 v-bind:title="doc.title"  
 v-on:update:title="doc.title = $event"  
></text-document>

为了方便起见，我们为这种模式提供一个缩写，即 .sync 修饰符：

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

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

# 作用域插槽(可从子组件获取数据)

<ul>  
 <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。-->  
 <slot v-bind:todo="todo">  
 <!-- 回退的内容 -->  
 {{ todo.text }}  
 </slot>  
 </li>  
</ul>

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

# 动态组件（is）

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

# 动态组件使用keep-alive

我们更希望那些标签的组件实例能够被在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 <keep-alive> 元素将其动态组件包裹起来。

<!-- 失活的组件将会被缓存！-->  
<keep-alive>  
 <component v-bind:is="currentTabComponent"></component>  
</keep-alive>

# 处理边界情况（\*\*\*\*\*\*\*）

## 访问根实例

根实例可以通过 $root 属性进行访问

所有的子组件都可以将这个实例作为一个全局 store 来访问或使用

// 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()

## 子组件访问父组件的实例（$parent）

它提供了一种机会，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。(但是这种方式不太好)，

推荐：需要向任意更深层级的组件提供上下文信息时推荐[**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)的原因。

## 访问子组件（ref特性）

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

this.$refs.usernameInput

# 侦听事件

* 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

# 依赖注入（provide和inject）

1. 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  
 }  
}

1. 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']

实际上，你可以把依赖注入看作一部分“大范围有效的 prop”

# 混入（mixins）

其实就是组件的复用功能，该混入对象可以包含组件的任意选项，当某个组件使用‘混入对象’时，所有该混入对象的选项都将被放到当前组件当中。当和组件的数据发生冲突时，以组件的为主，（一切以组件为主）

// 定义一个混入对象  
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() // => "hello from mixin!"

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。

用途：

* 1. 很多地方都会用到的公共方法，用混入的方法可以减少代码
  2. 需要在原有的基础上增加方法，可以使用混入的方法减少对源代码的污染

调用顺序：

混入对象的钩子将在组件自身钩子之前调用，如果遇到全局混入(`Vue.mixin`)，全局混入的执行顺序要前于混入和组件里的方法。

# 自定义指令

注册局部指令

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

# $nextTick(回调函数)（什么时候用它）

在DOM更新完成之后，再执行该回调函数，一般在修改数据之后使用该方法，以便获得更新后的DOM

Vue-Router

This.$route.params

1、嵌套路由

This.$router.push()

**2、路由模式的区别**：history和hash

history模式下，build之后本地 index.html 打开是无效的。

hash模式下，build之后本地 index.html 打开正常！

3、路由懒加载

const Foo = () => import('./Foo.vue')

const router = new VueRouter({

routes: [

{ path: '/foo', component: Foo }

]

})