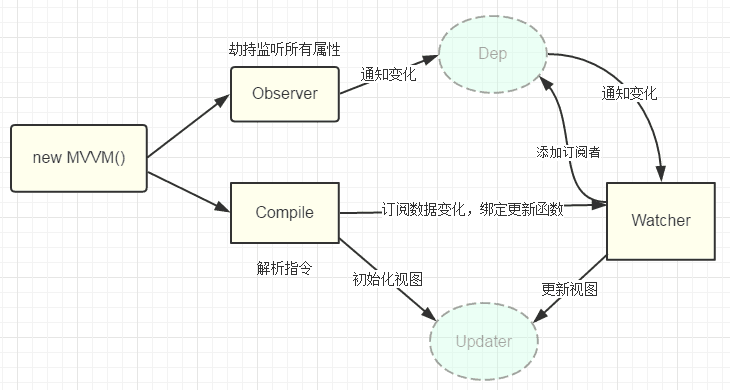
# 基础

## ·计算属性和侦听器

* 对于任何复杂逻辑，你都应当使用**计算属性**
* **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。
* computed 对象内的方法如果在初始化时绑定到元素上的事件会先执行一次这个方法 ，而 methods 内的方法则不会； 所以取决你是否需要缓存。
* 计算属性默认只有 getter ，不过在需要时你也可以提供一个 setter ：
* 侦听器watch：当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。可以限制操作的频率。
* 不能用箭头函数来定义侦听器watcher。因为箭头函数已经绑定了父级作用域的上下文，所以 this 将不会按照期望指向vue 实例。

**VUE双向绑定原理**



## Class 与Style 绑定

表达式结果的类型除了字符串之外，还可以是对象或数组。建议用对象的形式

### 绑定 HTML Class

#### 对象语法： :class={},

结果取决于各数据属性的truthiness

绑定的数据对象不必内联定义在模板里：

常用模式: 绑定一个返回对象的[**计算属性**](https://cn.vuejs.org/v2/guide/computed.html)

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

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

#### 数组语法： :class = [ ]

在数组语法中也可以使用对象语法：用于切换多个class

#### 组件语法

当在一个自定义组件上使用 class 属性时，这些类将被添加到该组件的根元素上面。这个元素上已经存在的类不会被覆盖。

### 绑定内联样式

* v-bind:style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。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：

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

* v-bind:style 的数组语法可以将多个样式对象应用到同一个元素上：

<div v-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。

## 条件渲染

* v-if 与v-else和v-else-if必须紧紧跟随。
* v-show切换元素的display属性。与v-if相比，频繁切换的场景适合v-show, 切换较少而渲染成本较高的场景适合v-if。
* 用 key 管理可复用的元素：由于vue倾向复用已有元素的关系。

场景：各个元素切换的时候是独立的，不要复用。

方法：指定一个具有唯一值的key：，这样切换时候元素会被重新渲染

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

注意，<label> 元素仍然会被高效地复用，因为它们没有添加 key 属性。

* v-for 比 v-if 具有更高的优先级

## 列表渲染 v-for

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

它也可以用于强制替换元素/组件而不是重复使用它。当你遇到如下场景时它可能会很有用：

* 完整地触发组件的生命周期钩子
* 触发过渡

### 数组更新检测

#### 变异方法：

push() pop() shift() unshift() splice() sort() reverse()等，会改变原数组

#### 非变异方法：

Filter() concat() slice()等，返回新数组，因此需要用新数组替换旧数组。

#### Vue不能检测的数组变动：

* 利用索引直接设置一个项：vm.items[indexOfItem] = newValue

解决：全局方法vue.set；原型方法Array.prototype.splice(); 实例方法vm.$set

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

解决： splice

### 对象更改检测

**Vue 不能检测对象属性的添加或删除。使用全局或者实例的set方法**

**赋予多个属性：**Object.assign() 或 \_.extend()方法时， 用两个对象的属性创建一个新的对象

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

### 显示过滤/排序方法

* 可以创建返回过滤或排序数组的计算属性。
* 在计算属性不适用的情况下 (例如，在嵌套 v-for 循环中) 你可以使用一个 method 方法：

### v-for 接受整数：重复此整数次模板

### v-for 在templete中渲染多个元素

### v-for with 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
* 如果你的目的是有条件地跳过循环的执行，那么可以将 v-if 置于外层元素 (或 [**<template>**](https://cn.vuejs.org/v2/guide/conditional.html#在-lt-template-gt-中配合-v-if-条件渲染一整组))上。

#### 一个组件的 v-for

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

然而，任何数据都不会被自动传递到组件里，因为组件有自己独立的作用域。为了把迭代数据传递到组件里，我们要用 props 。。不自动将 item 注入到组件里的原因是，这会使得组件与 v-for 的运作紧密耦合。明确组件数据的来源能够使组件在其他场合重复使用。

## 事件处理： v-on

### 事件处理方法：v-on接收一个需要调用的方法名称

### 内联处理器中的方法：

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

|  |
| --- |
| <button v-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

上面更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。这样减少在调用方法中处理。

* .stop .prevent .capture .self .once .passive

使用修饰符时，顺序很重要

v-on:click.prevent.self 会阻止**所有的点击**， v-on:click.self.prevent 只会阻止对元素自身的点击。

<!-- 阻止单击事件继续传播 -->  
<a v-on:click.stop="doThis"></a>  
<!-- 提交事件不再重载页面 -->  
<form v-on:submit.prevent="onSubmit"></form>  
<!-- 修饰符可以串联 -->  
<a v-on:click.stop.prevent="doThat"></a>  
<!-- 只有修饰符 -->  
<form v-on:submit.prevent></form>  
<!-- 添加事件监听器时使用事件捕获模式 -->  
<!-- 即元素自身触发的事件先在此处理，然后才交由内部元素进行处理 -->  
<div v-on:click.capture="doThis">...</div>  
<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->  
<!-- 即事件不是从内部元素触发的 -->  
<div v-on:click.self="doThat">...</div>

<!-- 点击事件将只会触发一次 -->  
<a v-on:click.once="doThis"></a>

<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->  
<!-- 而不会等待 `onScroll` 完成 -->  
<!-- 这其中包含 `event.preventDefault()` 的情况 -->  
<div v-on:scroll.passive="onScroll">...</div>

### 按键修饰符

* .enter .tab .delete (捕获“删除”和“退格”键) .esc .space
* .up .down .left .right

### 系统修饰键

* .ctrl .alt .shift .meta

## 表单输入绑定

### 常用方法

v-model在<input>, <textarea>, <select>元素上建立双向绑定。 忽略所有表单元素的value. Checked, selected特性的初始值而总是将 Vue 实例的数据作为数据来源。你应该通过 JavaScript 在组件的 data 选项中声明初始值。

* 多行文本：在文本区域插值 (<textarea></textarea>) 并不会生效，应用 v-model 来代替。
* 复选框：

checkbox单个复选框的数据绑定到布尔值；

多个复选框的数据绑定到同一个数组。

* 单选按钮：radio
* 选择框：select， option 单选时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（用键盘crtl配合？？）

### 值绑定：

对于单选按钮，复选框及选择框的选项，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)：

但是有时我们可能想把值绑定到 Vue 实例的一个动态属性上，这时可以用 v-bind 实现，并且这个属性的值可以不是字符串。

#### 复选框：

#### 单选按钮

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

|  |
| --- |
| // 当选中时 vm.pick === vm.a |

选择框的选项

### 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

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

* **.number** 自动将用户的输入值转为数值类型

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

* **.trim** 自动过滤用户输入的首尾空白字符

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

### 在组件上使用v-model：

## 组件基础

### 组件的复用

#### data必须是一个函数

data(){ return {…} } 或者 data: function(){ return{ …}}

### 组件的组织：全局注册与局部注册

### 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。

### 每个组件必须只有一个根元素

### 通过事件向父级组件发送消息： $emit()

vm.$emit( eventName, […args] )

eventName： 传回给父级的事件名

…args: 抛出的值用于被父级接收调用

在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
* 如果这个事件处理函数是一个方法，那么这个值将会作为第一个参数传入这个方法

### 通过插槽分发内容

### [动态组件](动态组件&异步组件#_动态组件&异步组件)

通过 Vue 的 <component> 元素加一个特殊的 is 特性来实现

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

在上述示例中，currentTabComponent 可以包括

* 已注册组件的名字，或
* 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。

特殊的 is 特性给了我们一个变通的办法

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**：

* 字符串 (例如：template: '...')
* [**单文件组件 (.vue)**](https://cn.vuejs.org/v2/guide/single-file-components.html)
* [**<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：

### [组件名大小写](https://cn.vuejs.org/v2/guide/components-registration.html#组件名大小写)

两种方式。直接在 DOM (即非字符串的模板) 中使用时只有 kebab-case 是有效的。

### 全局注册

在注册之后可以用在任何新创建的 Vue 根实例 (new Vue) 的模板中。

### 局部注册

ES2015写法：

import ComponentA from './ComponentA.vue'  
  
export default {  
 components: {  
 ComponentA  
 },  
 // ...  
}

注意在 ES2015+ 中，在对象中放一个类似 ComponentA 的变量名其实是 ComponentA: ComponentA 的缩写，即这个变量名同时是：

* 用在模板中的自定义元素的名称 左边的componentA（在模板中转成kebab-case）
* 包含了这个组件选项的变量名 右边的componentA

### 模块系统

### 基础组件的自动化全局注册

基础组件：内容较少，被其他组件频繁调用。使用 require.context 只全局注册这些非常通用的基础组件。 [webpack require context 说明](https://www.jianshu.com/p/78f7b19932cb)

记住**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：

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。如果你使用字符串模板，那么这个限制就不存在了。

### Prop类型：

以对象形式列出 prop，指定各个prop的名称与类型

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

### 传递静态prop：

用v-bind来告诉vue右边是一个js表达式而不是字符串。

<blog-post v-bind:likes="42"></blog-post>

**传入一个对象的所有属性**：使用不带参数的v-bind。

### 单向数据流：

只有父级prop更新会向下流动到子组件。也不应该在一个子组件内部改变prop。这里有两种常见的试图改变一个 prop 的情形：

* **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：
* **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。

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：

定制方法是提供一个对象

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  
 }  
 }  
 }  
})

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

type可以是八大原生构造函数中的一个；也可以自定义构造函数，并通过instanceof进行检测。

String Number Boolean Array Object Date Function Symbol

### 非prop特性：

指传向一个组件，但是该组件并没有相应 prop 定义的特性。

#### 替换/合并已有的特性

绝大部分特性而言，外部传入的会替换掉组件内部的；class和style特性稍好，会合并两边的值。

#### 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。配合实例的 $attrs 属性，在撰写[**基础组件**](https://cn.vuejs.org/v2/style-guide/#基础组件名-强烈推荐)的时候是常会用到的，可以手动决定这些特性会被赋予哪个元素。

## 自定义事件

推荐始终使用kebab-case的事件名

**事件名：**事件名不存在任何自动化的大小写转换。而是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。

### 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，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：

### 将原生事件绑定到组件：.native修饰符

### 将原生事件绑定到组件

原因似乎是无法监听非子元素（后代元素）

$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。

{  
 focus: function (event) { /\* ... \*/ }  
 input: function (value) { /\* ... \*/ },  
}

就可以配合 v-on="$listeners" 将所有的事件监听器指向这个组件的某个特定的子元素。

Vue.component('base-input', {  
 inheritAttrs: false,  
 props: ['label', 'value'],  
 computed: {  
 inputListeners: function () {  
 var vm = this //单向数据流第一种情况  
 // `Object.assign` 将所有的对象合并为一个新对象  
 return Object.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)  
 }  
 }  
 )  
 }  
 },  
 template: `  
 <label>  
 {{ label }}  
 <input  
 v-bind="$attrs"  
 v-bind:value="value"  
 v-on="inputListeners"  
 >  
 </label>  
 `  
})

### .sync修饰符 语法糖

背景：对一个 prop 进行“双向绑定会带来维护上的问题，

推荐以 update:myPropName 的模式触发事件取而代之。

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

方便起见，提供这个.sync缩写。

<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。

* 当我们用一个对象同时设置多个 prop 的时候：

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

这样会把 doc 对象中的每一个属性 (如 title) 都作为一个独立的 prop 传进去，然后各自添加用于更新的 v-on 监听器

* 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。

## 插槽 <slot> ：作为承载分发内容的出口

当组件渲染的时候，这个 <slot> 元素将会被替换。插槽内可以包含任何模板代码，包括HTML，甚至是其他的组件。如果组件中没有包含一个<slot>元素，则任何传入他的内容都会被抛弃。

### 具名插槽：

<slot>有个属性：name。向具名插槽提供内容的时候，可以：

* 在父组件的<template>元素上使用slot属性。
* 直接用在一个普通的元素。

**默认插槽**：还会保留一个未命名插槽，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

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。

<button type="submit">  
 <slot>Submit</slot>  
</button>

如果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。

#### 作用域插槽

背景：你希望提供的组件带有一个可从子组件获取数据的可复用的插槽。

操作：

* 为每个待处理项准备一个插槽：将待处理项内容包裹在一个slot元素内，然后将所有和其上下文相关的数据传递给这个插槽。
* 这样便可以对每一个代办项定义一个不同的<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-list v-bind:todos="todos">  
 <!-- 将 `slotProps` 定义为插槽作用域的名字 -->

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

**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

[ES2015相关。](https://www.jdon.com/idea/js/use_destructuring_in_es2015.html)

## 动态组件&异步组件

### 在动态组件上使用 keep-alive

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。

* 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

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。

#### 处理加载状态

## 处理边界情况

### 访问元素&组件

在绝大多数情况下，我们最好不要触达另一个组件实例内部或手动操作 DOM 元素。不过也确实在一些情况下做这些事情是合适的。

#### 访问根实例

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

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。

* 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。

#### 访问父级组件实例

$parent 属性可以用来从一个子组件访问父组件的实例。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。

另外在一些可能适当的时候，你需要特别地共享一些组件库。比如这个地图组件：

<https://jsfiddle.net/chrisvfritz/ttzutdxh/> 想要通过类似 this.$parent.getMap 的方式访问那个地图，以便为其添加一组标记。

* 请留意，尽管如此，通过这种模式构建出来的那个组件的内部仍然是容易出现问题的。
* 这也是我们针对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#依赖注入)的原因。

#### 访问子组件实例或子元素：抛开prop和事件 ref和$refs

通过 ref 特性为这个子组件赋予一个 ID 引用。

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

现在在你已经定义了这个 ref 的组件里，你可以使用：

|  |
| --- |
| this.$refs.usernameInput |

* 可以实现出父级组件级联访问到子组件方法？？
* 当 ref 和 v-for 一起使用的时候，你得到的引用将会是一个包含了对应数据源的这些子组件的数组。
* $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 实例选项：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  
 }  
}

* 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']

<https://jsfiddle.net/chrisvfritz/tdv8dt3s/>

1. 这个用法可以让我们在任意后代组件中访问 getMap，而不需要暴露整个 <google-map> 实例。
2. 这允许我们更好的持续研发该组件，而不需要担心我们可能会改变/移除一些子组件依赖的东西。
3. 同时这些组件之间的接口是始终明确定义的，就和 props 一样。
4. 实际上，你可以把依赖注入看作一部分“大范围有效的 prop”，除了：
   * 祖先组件不需要知道哪些后代组件使用它提供的属性
   * 后代组件不需要知道被注入的属性来自哪里 ????
5. 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#访问根实例)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

常规方法： $emit 的用法，它可以被 v-on 侦听

其他方法： Vue 实例同时在其事件接口中提供。当你需要在一个组件实例上手动侦听事件时：

* 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

也可以用于代码组织工具

注意，即便如此，如果你发现自己不得不在单个组件里做很多建立和清理的工作，最好的方式通常还是创建更多的（可复用的）模块化组件。

* 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

组件只能通过 name 选项，在它们自己的模板中调用自身的。

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：

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)。

##### 组件之间的循环引用

假设你需要构建一个文件目录树。可能出现这些组件在渲染树中互为对方的后代和祖先。

当通过 Vue.component 全局注册组件的时候，这个悖论会被自动解开。如果你使用一个模块系统依赖/导入组件，例如通过 webpack 或 Browserify，你会遇到一个错误。

为了解决这个问题，我们需要给模块系统一个点，在那里“A 反正是需要 B 的，但是我们不需要先解析 B。”

在我们的例子中，把 <tree-folder> 组件设为了那个点（A）。我们知道那个产生悖论的子组件是 <tree-folder-contents> 组件(B)，所以我们会等到生命周期钩子 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')  
}

#### 模板定义的替代品

##### 内联模板 难以理解

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。

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-Templetes

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。例如：

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

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

* 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：手动强制更新或阻止更新

##### 强制更新：基本上是不必要的

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

组件包含了**大量**静态内容，这时在根元素上添加 v-once 特性以确保这些内容只计算一次然后缓存起来

不要过度使用这个模式：带来的便利有限，而且不利于后期协同维护

# 可复用性与组合

## 混入MIXIN

混入对象 var mixin = {}可以包含任意组件选项, 所有混入对象的选项将被混入该组件本身的选项。

### **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。
* 钩子函数：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。

注意：Vue.extend() 也使用同样的策略进行合并。

### 全局混入 Vue.mixin

**（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。）**

**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 OptionMergeStrategies

* 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

### 钩子函数（均为可选）

* 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。
* 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)。
* 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)。
* componentUpdated：指令所在组件的 VNode **及其子 VNode** 全部更新后调用。
* unbind：只调用一次，指令与元素解绑时调用。

### 钩子函数参数

* 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" 中，绑定值为 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：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"。
  + arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。
  + 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 }。
* vnode：Vue 编译生成的虚拟节点。移步 [**VNode API**](https://cn.vuejs.org/v2/api/#VNode-接口) 来了解更多详情。
* oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

## 渲染函数&JSX

## 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 Vue.use()

1. 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：
2. 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。

1. 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()：

[**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.install()

如果插件是一个对象，必须提供 install 方法。

如果插件是一个函数，它会被作为 install 方法。install 方法调用时，会将 Vue 作为参数传入。

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

* 过滤器可以用在两个地方：**双花括号插值和 v-bind 表达式。**被添加在 JavaScript 表达式的尾部，由“管道”符号指示。
* 可以在一个组件的选项中定义本地的过滤器：filters: { }
* 或者在创建 Vue 实例之前全局定义过滤器：Vue.filter( )

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。

* 因此过滤器可以串联。
* 因此过滤器自己可以接受参数，顺序排在前面的表达式之后：2，3，4.。。

# 路由ROUTER SPA建议使用vue-router库

## 起步

<router-link></router-link>是一个组件该组件用于设置一个导航链接，切换不同 HTML 内容。 **to** 属性为目标地址。

1. 如果使用模块化机制编程，导入Vue和VueRouter，要调用 Vue.use(VueRouter)
2. 定义路由组件：可通过模板template或者import命令导入vue文件

const Foo = { template: '<div>foo</div>' }

const Bar = { template: '<div>bar</div>' }

1. 定义路由：每个路由应该映射一个组件。 其中"component" 可以是
   * 通过 Vue.extend() 创建的组件构造器，
   * 或者，只是一个组件配置对象。
   * 我们晚点再讨论嵌套路由。

const routes = [

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

{ path: '/bar', component: Bar } ]

//路径对应to指定的地址，component为组件名称

1. 创建 router 实例 ，然后传 `routes` 配置

const router = new VueRouter({ routes // （缩写）相当于 routes: routes

1. 创建和挂载根实例。记得要通过 router 配置参数注入路由，从而让整个应用都有路由功能。

const app = new Vue({ router }).$mount('#app')

通过注入路由器，我们可以在任何组件内通过 this.$router 访问路由器，也可以通过 this.$route 访问当前路由。

留意一下 this.$router 和 router 使用起来完全一样。我们使用 this.$router 的原因是我们并不想在每个独立需要封装路由的组件中都导入路由。

点击过的导航链接都会加上样式

**class ="router-link-exact-active router-link-active"**。

## 动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。

一个“路径参数”使用冒号 : 标记。当匹配到一个路由时，参数值会被设置到 this.$route.params，可以在每个组件内使用。

一个路由中设置多段“路径参数”，对应的值都会设置到 $route.params 中

**$route 对象提供了很多有用的信息 查看**[**API 文档**](https://router.vuejs.org/zh/api/#路由对象)

### 响应路由参数的变化

当使用路由参数时，例如从 /user/foo 导航到 /user/bar，**原来的组件实例会被复用，也意味着组件的生命周期钩子不会再被调用。**

复用组件时，想对路由参数的变化作出响应的话，你可以

* + 简单地 watch (监测变化) $route 对象：
  + 或者使用 2.2 中引入的 beforeRouteUpdate 导航守卫：

### 高级匹配模式 查看它的 [文档](https://github.com/pillarjs/path-to-regexp#parameters)

### 匹配优先级： 定义顺序

## 嵌套路由

一个被渲染组件同样可以包含自己的嵌套 <router-view>. 要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置：

const router = new VueRouter({

routes: [

{ path: '/user/:id', component: User,

children: [

{

// 当 /user/:id/profile 匹配成功，

// UserProfile 会被渲染在 User 的 <router-view> 中

path: 'profile',

component: UserProfile

},

{

// 当 /user/:id/posts 匹配成功

// UserPosts 会被渲染在 User 的 <router-view> 中

path: 'posts',

component: UserPosts

}

]

}

]

})

**要注意，以 / 开头的嵌套路径会被当作根路径。 这让你充分的使用嵌套组件而无须设置嵌套的路径。**

当你访问 /user/foo 时，User 的出口是不会渲染任何东西，这是因为没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个 空的 子路由：{path : ‘’, component: UserHome}

## 编程式的导航 模仿[window.historyAPI](https://developer.mozilla.org/zh-CN/docs/Web/API/History_API/)

### router.push(location, onComplete?, onAbort?)

**注意：在 Vue 实例内部，你可以通过 $router 访问路由实例。因此你可以调用 this.$router.push。**



第一个参数可以为字符串，对象，命名的路由，或者

// 带查询参数，变成 /register? plan=private

router.push({ path: 'register', query: { plan: 'private' }})

**注意：如果提供了 path，params 会被忽略，上述例子中的 query 并不属于这种情况。取而代之的是下面例子的做法，你需要提供路由的 name 或手写完整的带有参数的 path**

**同样的规则也适用于 router-link 组件的 to 属性。**

const userId = 123

router.push({ name: 'user', params: { userId }}) // -> /user/123

router.push({ path: `/user/${userId}` }) // -> /user/123

// 这里的 params 不生效

router.push({ path: '/user', params: { userId }}) // -> /user

后两个为回调，这些回调将会在导航成功完成 (在所有的异步钩子被解析之后) 或终止 (导航到相同的路由、或在当前导航完成之前导航到另一个不同的路由) 的时候进行相应的调用。

* \*\*如果目的地和当前路由相同，只有参数发生了改变 (比如从一个用户资料到另一个 /users/1 -> /users/2)，你需要使用 beforeRouteUpdate 来响应这个变化 (比如抓取用户信息)

### router.replace(location, onComplete?, onAbort?)

它不会向 history 添加新记录，而是替换.

### router.go(n) n为步数

### **Vue Router 的导航方法 (push、 replace、 go) 在各类路由模式 (history、** hash 和 abstract) 下表现一致。

## 命名路由

<router-link :to="{ name: 'user', params: { userId: 23 }}">User</router-link>

router.push({ name: 'user', params: { userId: 123 }})

## 命名视图

懒加载 ：开启了以后不同的路由组件可以分成不同的代码，从而实现进入相应页面以后才加载相关代码。

**const** admin = resolve => require(['../views/admin.vue'], resolve);

# 规模化

# API

## 实例属性

### vm.$watch( expOrFn, callback, [options] )

回调函数得到的参数为新值和旧值.

表达式只接受监督的键路径。对于更复杂的表达式，用一个函数取代。

* 注意：在变异 (不是替换) 对象或数组时，旧值将与新值相同，因为它们的引用指向同一个对象/数组。Vue 不会保留变异之前值的副本。

vm.$watch 返回一个取消观察函数，用来停止触发回调：

* **选项：deep**

为了发现对象内部值的变化，可以在选项参数中指定 deep: true 。注意监听数组的变动不需要这么做。

vm.$watch('someObject', callback, {  
 deep: true  
})  
vm.someObject.nestedValue = 123  
// callback is fired

* **选项：immediate** 立即以表达式的当前值触发回调：

vm.$watch('a', callback, {  
 immediate: true  
})// 立即以 `a` 的当前值触发回调

## 指令

### v-cloak

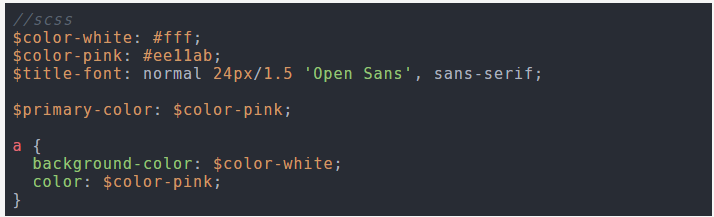
这个指令保持在元素上直到关联实例结束编译。和 CSS 规则如 [v-cloak] { display: none } 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。

<https://segmentfault.com/a/1190000008819667>

# SCSS笔记

## 变量及其运算

美元符号$定义变量，冒号: 指定值。变量可以指定给其他变量，可以是字符串（）



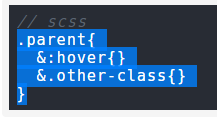
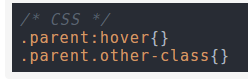
颜色使用十六进制码；

属性按照字典顺序

## 巢状语法NESTING（建议结构）

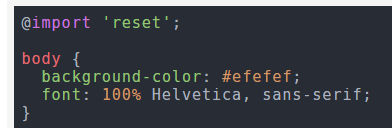
兩種情況，一種是指包含在某個DOM元素之內的，例如上面範例的ul li a。而另一種則是指同個DOM元素的但不同類別，例如ul.mylist，

后一种不带空格的情况需要加入&符号

 会编译为 

## 导入文件IMPORT

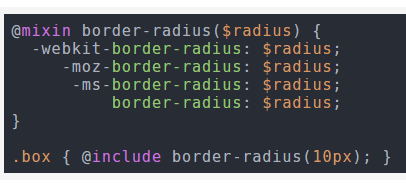
被导入的文件通常在文件名前加下划线\_以作区分，这样编译器会略过。导入语句中不需要写下划线，因为编译器会自动查找对应的带下划线的文件名。



注意和css的@import 有什么区别。

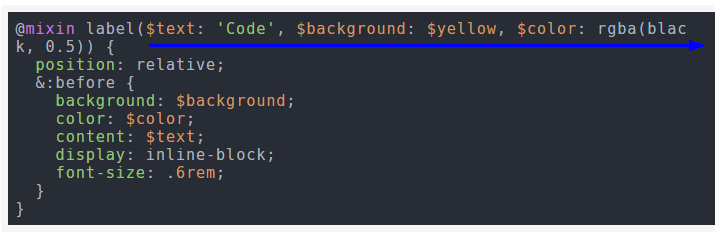
## 混合MIXIN

形式类似一组值的组合或者一个函数，如果你需要整個群組或整個類別的CSS套用 常用于套用不同浏览器的css声明



@minix標記需要對應到@include標記 @include可以出现在任意位置。

定义传入值可以加上预设值



## 函数FUNCTION

@function可以定義[自訂的函式](http://sass-lang.com/documentation/file.SASS_REFERENCE.html#function_directives)

也內建了許多工具函式，詳見[內建函式的清單](http://sass-lang.com/documentation/Sass/Script/Functions.html)

## 注释COMMENT

CSS只支持/\*…\*/ ，SCSS还支持//的单行注释。

/\* \*/ are preserved where possible in the output. SCSS also uses // for comments that are thrown away, like Sass. Unlike Sass, though, // comments in SCSS may appear anywhere and last only until the end of the line.