中南大学升华工作室Vue培训

49.234.77.58/index.php/2019/11/17/中南大学升华工作室vue培训

XZLang 2019年11月17日

2019年升华工作室网络部Vue培训

2019年升华工作室网络部Vue培训
Vue学习方法：
阅读文档 **https://cn.vuejs.org/**（推荐）
菜鸟教程
相关书籍
练习：
https://jsfiddle.net

认识JS

Vue (读音 /vjuː/，类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

Vue.js 安装

1、下载独立版本

2、CDN方法：推荐国内：https://cdn.jsdelivr.net/npm/vue

Vue.js 目录结构

Vue.js 起步

每个 Vue 应用都是通过用 `Vue` 函数创建一个新的 **Vue 实例**开始的：

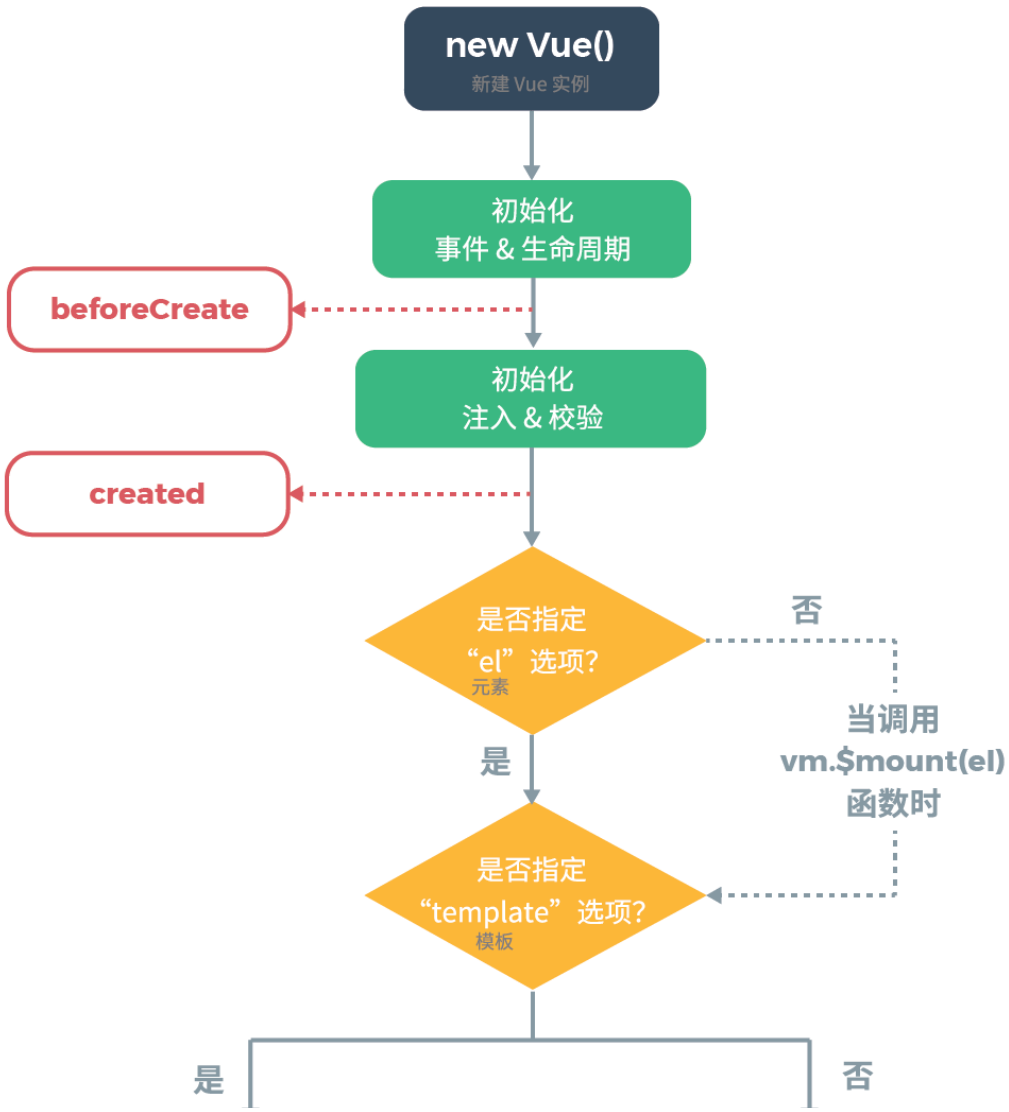
```
var vm = new Vue({
  // 选项
})
```

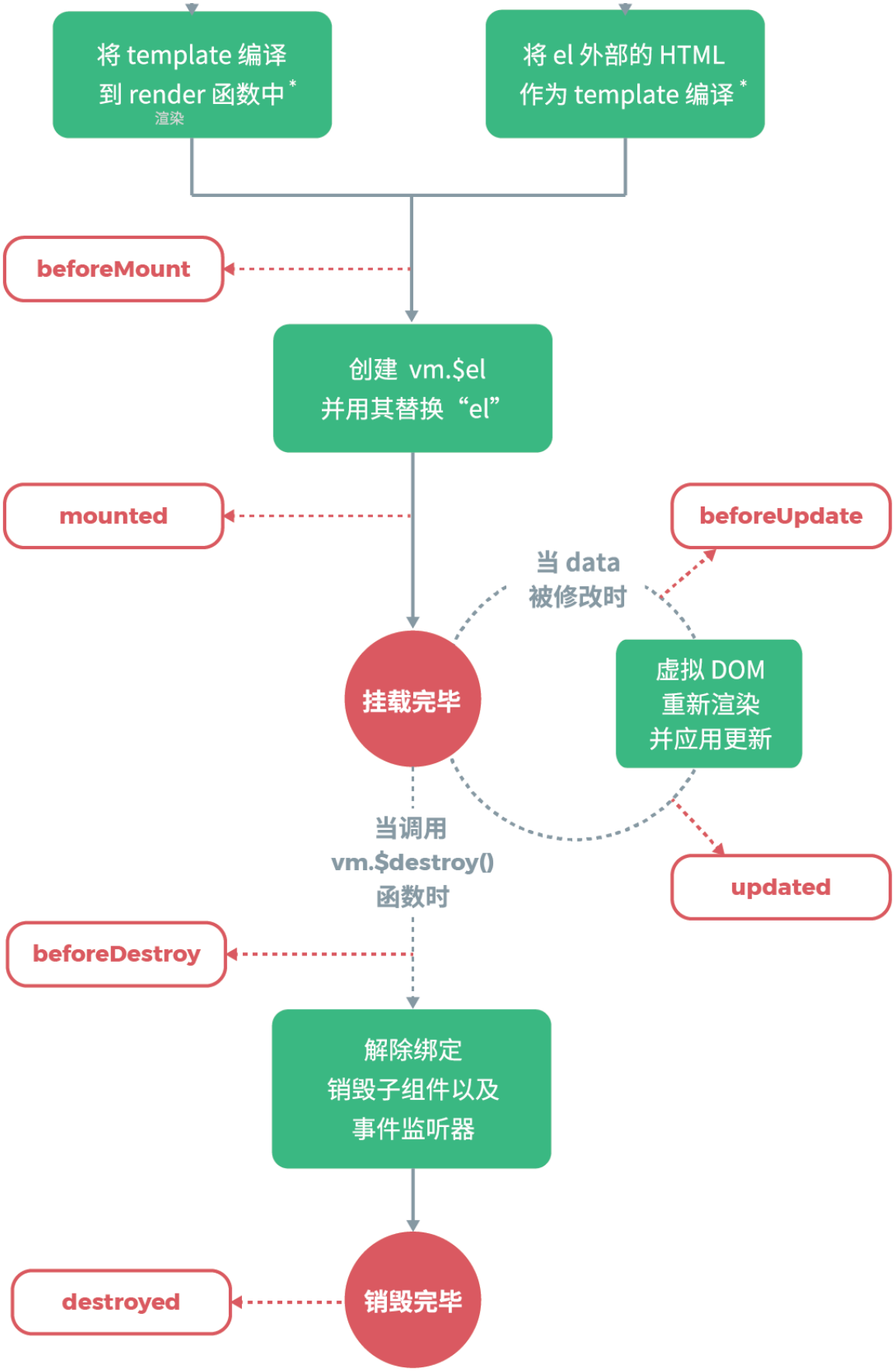
当创建一个 Vue 实例时，你可以传入一个**选项对象**。

> 数据与方法：当一个 Vue 实例被创建时，它将 `data` 对象中的所有的属性加入到 Vue 的**响应式系统**中。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。
> 例外是使用 `Object.freeze()`，这会阻止修改现有的属性，也意味着响应系统无法再*追踪*变化。

```
<div id="vue_det">
   <h1>site : {{site}}</h1>
   <h1>url : {{url}}</h1>
   <h1>{{details()}}</h1>
</div>
<script type="text/javascript">
   var vm = new Vue({
      el: '#vue_det',
      data: {
         site: "菜鸟教程",
         url: "www.runoob.com",
         alexa: "10000"
      },
      methods: {
         details: function() {
            return  this.site + " - 学的不仅是技术，更是梦想！";
         }
      }
   })
</script>
```

生命周期钩子





* 如果使用构造生成文件（例如构造单文件组件），
模板编译将提前执行

```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` 指向 vm 实例
    console.log('a is: ' + this.a)
  }
})
```

Vue.js 模板语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML ，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

插值，指令，用户输入

插值：文本，HTML，属性，使用 JavaScript 表达式，

```
<span>Message: {{ msg }}</span>
<span v-once>这个将不会改变: {{ msg }}</span>
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
<div v-bind:class="{'class1': use}">
{{9*9}}
{{ message.split('').reverse().join('') }}
```

指令：v-if，v-for，v-bind，v-on，v-model...

指令 (Directives) 是带有 `v-` 前缀的特殊特性。指令特性的值预期是**单个 JavaScript 表达式** (`v-for` 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。

```
<a v-bind:[attributeName]="url"> ... </a>
```

这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的 Vue 实例有一个 data 属性 attributeName，其值为 "href"，那么这个绑定将等价于 v-bind:href。

同样地，你可以使用动态参数为一个动态的事件名绑定处理函数：

```
<a v-on:[eventName]="doSomething"> ... </a>
```

在这个示例中，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。

Vue.js 条件语句

`v-if` 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 truthy 值的时候被渲染。

因为 `v-if` 是一个指令，所以必须将它添加到一个元素上。但是如果想切换多个元素呢？此时可以把一个 `<template>` 元素当做不可见的包裹元素，并在上面使用 `v-if`。最终的渲染结果将不包含 `<template>` 元素。

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做除了使 Vue 变得非常快之外，还有其它一些好处。例如，如果你允许用户在不同的登录方式之间切换：

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

这样也不总是符合实际需求，所以 Vue 为你提供了一种方式来表达“这两个元素是完全独立的，不要复用它们”。只需添加一个具有唯一值的 `key` 属性即可：

```
<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>
```

v-if vs v-show

```
<h1 v-show="ok">Hello!</h1>
```

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

`v-if` 也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

相比之下，`v-show` 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 `v-show` 较好；如果在运行时条件很少改变，则使用 `v-if` 较好。

Vue.js 循环语句

我们可以用 `v-for` 指令基于一个数组来渲染一个列表。`v-for` 指令需要使用 `item in items` 形式的特殊语法，其中 `items` 是源数据数组，而 `item` 则是被迭代的数组元素的**别名**。

```
<ul id="example-1">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

在 `v-for` 块中，我们可以访问所有父作用域的属性。`v-for` 还支持一个可选的第二个参数，即当前项的索引。

```
<ul id="example-2">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>
```

你也可以用 `v-for` 来遍历一个对象的属性。

```
<ul id="v-for-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
new Vue({
  el: '#v-for-object',
  data: {
    object: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
})
```

变异方法 (mutation method)

Vue 将被侦听的数组的变异方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

你可以打开控制台，然后对前面例子的 `items` 数组尝试调用变异方法。比如 `example1.items.push({ message: 'Baz' })`。

替换数组

变异方法，顾名思义，会改变调用了这些方法的原始数组。相比之下，也有非变异 (non-mutating method) 方法，例如 `filter()`、`concat()` 和 `slice()`。它们不会改变原始数组，而**总是返回一个新数组**。当使用非变异方法时，可以用新数组替换旧数组：

```
example1.items = example1.items.filter(function (item) {
  return item.message.match(/Foo/)
})
```

由于 JavaScript 的限制，Vue **不能**检测以下数组的变动：

1. 当你利用索引直接设置一个数组项时，例如：`vm.items[indexOfItem] = newValue`
2. 当你修改数组的长度时，例如：`vm.items.length = newLength`

举个例子：

```
var vm = new Vue({
  data: {
    items: ['a', 'b', 'c']
  }
})
vm.items[1] = 'x' // 不是响应性的
vm.items.length = 2 // 不是响应性的
```

为了解决第一类问题，以下两种方式都可以实现和 `vm.items[indexOfItem] = newValue` 相同的效果，同时也将在响应式系统内触发状态更新：

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

你也可以使用 `vm.$set` 实例方法，该方法是全局方法 `Vue.set` 的一个别名：

```
vm.$set(vm.items, indexOfItem, newValue)
```

为了解决第二类问题，你可以使用 `splice`：

```
vm.items.splice(newLength)
```

还是由于 JavaScript 的限制，Vue 不能检测对象属性的添加或删除：

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` 现在是响应式的

vm.b = 2
// `vm.b` 不是响应式的
```

对于已经创建的实例，Vue 不允许动态添加根级别的响应式属性。但是，可以使用 `Vue.set(object, propertyName, value)` 方法向嵌套对象添加响应式属性。例如，对于：

```
var vm = new Vue({
  data: {
    userProfile: {
      name: 'Anika'
    }
  }
})
```

你可以添加一个新的 `age` 属性到嵌套的 `userProfile` 对象：

```
Vue.set(vm.userProfile, 'age', 27)
```

你还可以使用 `vm.$set` 实例方法，它只是全局 `Vue.set` 的别名：

```
vm.$set(vm.userProfile, 'age', 27)
```

有时你可能需要为已有对象赋值多个新属性，比如使用 `Object.assign()` 或 `_.extend()`。在这种情况下，你应该用两个对象的属性创建一个新的对象。所以，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：

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

Vue.js 计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。

计算属性默认只有 getter **，不过在需要时你也可以提供一个** setter

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})

//setter
computed: {
  fullName: {
    // getter
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // setter
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}
```

计算属性缓存 vs **方法**

计算属性 vs **侦听属性**

Vue.js 监听属性

watch **选项提供了一个更通用的方法，来响应数据的变化。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。**

```
<div id = "computed_props">
千米 : <input type = "text" v-model = "kilometers">
米 : <input type = "text" v-model = "meters">
</div>
<p id="info"></p>
<script type = "text/javascript">
var vm = new Vue({
  el: '#computed_props',
  data: {
    kilometers : 0,
    meters:0
  },
  methods: {
  },
  computed :{
  },
  watch : {
    kilometers:function(val) {
      this.kilometers = val;
      this.meters = this.kilometers * 1000
    },
    meters : function (val) {
      this.kilometers = val/ 1000;
      this.meters = val;
    }
  }
});
// $watch 是一个实例方法
vm.$watch('kilometers', function (newValue, oldValue) {
// 这个回调将在 vm.kilometers 改变后调用
document.getElementById ("info").innerHTML = "修改前值为: " + oldValue + "，修改后值为: " +
newValue;
})
</script>
```

Vue.js 样式绑定

我们可以传给 `v-bind:class` 一个对象，以动态地切换 class：

```
<div v-bind:class="{ active: isActive }"></div>
```

上面的语法表示 `active` 这个 class 存在与否将取决于数据属性 `isActive` 的 truthiness

你可以在对象中传入更多属性来动态切换多个 class。此外， `v-bind:class` 指令也可以与普通的 class 属性共存。

```
 .object1 {
  height:100px;
  width:100px;
 }
 .object2a {
  background: blue;
 }
 .object2b {
  background: red;
 }
<div id="app-4" class="object1" v-bind:class="object2">
</div>

<script>
var app4 = new Vue({
  el: '#app-4',
  data: {
    object2:{
  object2a: true,
  object2b: false
 }

  }
})
```

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

```
<div v-bind:style="styleObject"></div>
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

Vue.js 事件处理器

监听事件：

可以用 `v-on` 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

事件处理方法：

然而许多事件处理逻辑会更为复杂，所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。因此 `v-on` 还可以接收一个需要调用的方法名称。

```
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)
      }
    }
  }
})

// 也可以用 JavaScript 直接调用方法
example2.greet() // => 'Hello Vue.js!'
```

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：

在事件处理程序中调用 `event.preventDefault()` 或 `event.stopPropagation()` 是非常常见的需求。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题，Vue.js 为 `v-on` 提供了**事件修饰符**。之前提过，修饰符是由点开头的指令后缀来表示的。

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`
- `.passive`

```
<!-- 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 -->
<!-- 即内部元素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>
```

按键修饰符：

在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符：

```
<!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->
<input v-on:keyup.enter="submit">
```

你可以直接将 KeyboardEvent.key 暴露的任意有效按键名转换为 kebab-case 来作为修饰符。

```
<input v-on:keyup.page-down="onPageDown">
```

在上述示例中，处理函数只会在 `$event.key` 等于 `PageDown` 时被调用。

Vue.js 表单

v-model指令可以实现表单数据和vue数据对象的双向绑定。
单个复选框，绑定到布尔值
多个复选框，绑定到同一个数组
单选框
选择框
值绑定：

```
true-value="yes" false-value="no"
<input type="radio" v-model="pick" v-bind:value="a">
<select v-model="selected">
    <!-- 内联对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
// 当选中时
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

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：

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

.number
如果想自动将用户的输入值转为数值类型，可以给 v-model 添加 number 修饰符：

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

这通常很有用，因为即使在 type=“number” 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 parseFloat() 解析，则会返回原始的值。

.trim
如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符：

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

Vue.js 组件

Vue.js 自定义指令

bject’
vm.selected.number // => 123

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：

.number
如果想自动将用户的输入值转为数值类型，可以给 v-model 添加 number 修饰符：

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

这通常很有用，因为即使在 type=“number” 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 parseFloat() 解析，则会返回原始的值。

.trim
如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符：

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

Vue.js 组件

```
// 定义一个名为 button-counter 的新组件
Vue.component('button-counter', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'
})
```

组件是可复用的 Vue 实例，且带有一个名字：在这个例子中是 。我们可以在一个通过 new Vue 创建的 Vue 根实例中，把这个组件作为自定义元素来使用：

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
new Vue({ el: '#components-demo' })
```

因为组件是可复用的 Vue 实例，所以它们与 new Vue 接收相同的选项，例如 data、computed、watch、methods 以及生命周期钩子等。仅有的例外是像 el 这样根实例特有的选项。

你可以将组件进行任意次数的复用：

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

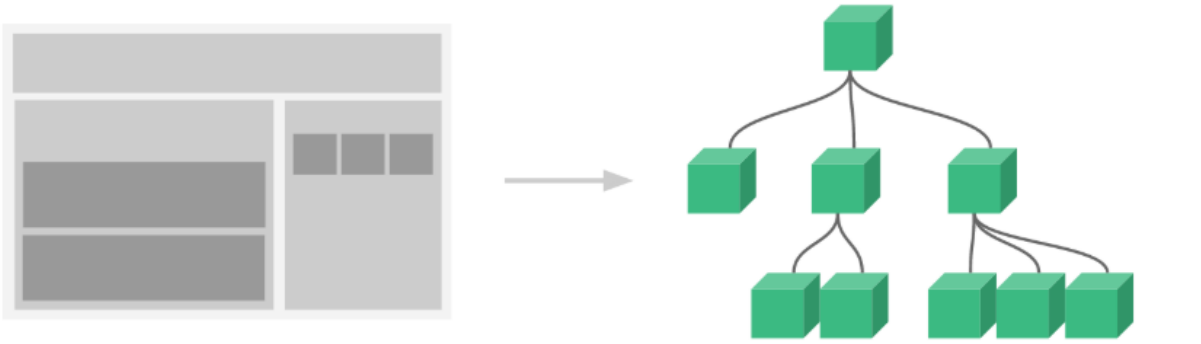
当我们定义这个 组件时，你可能会发现它的 data 并不是像这样直接提供一个对象：

```
data: {
  count: 0
}
```

取而代之的是，一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝：

```
data: function () {
  return {
    count: 0
  }
}
```

通常一个应用会以一棵嵌套的组件树的形式来组织：



例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。

为了能在模板中使用，这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型：全局注册和局部注册。至此，我们的组件都只是通过 Vue.component 全局注册的：

```
Vue.component('my-component-name', {
  // ... options ...
})
```

全局注册的组件可以用在其被注册之后的任何 (通过 new Vue) 新创建的 Vue 根实例，也包括其组件树中的所有子组件的模板中。
早些时候，我们提到了创建一个博文组件的事情。问题是如果你不能向这个组件传递某一篇博文的标题或内容之类的我们想展示的数据的话，它是没有办法使用的。这也正是 prop 的由来。

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。为了给博文组件传递一个标题，我们可以用一个 props 选项将其包含在该组件可接受的 prop 列表中.

```
Vue.component('TITLE', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

当构建一个 组件时，你的模板最终会包含的东西远不止一个标题：

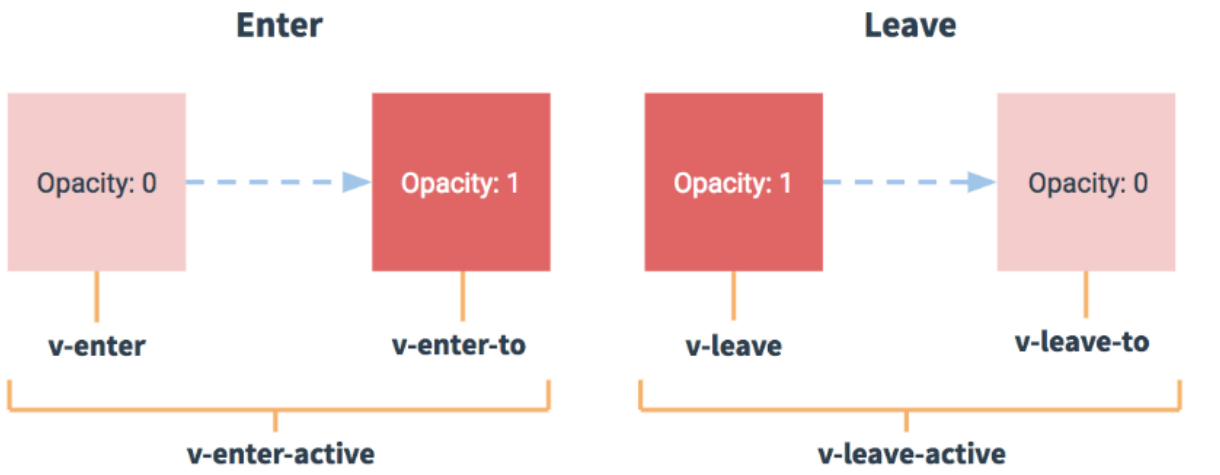
```
<h3>{{ title }}</h3>
```

最最起码，你会包含这篇博文的正文：

```
<h3>{{ title }}</h3>
<div v-html="content"></div>
```

然而如果你在模板中尝试这样写，Vue 会显示一个错误，并解释道 every component must have a single root element (每个组件必须只有一个根元素)。

Vue.js 过渡



单元素的过渡：
类名：
在进入/离开的过渡中，会有 6 个 class 切换。

对于这些在过渡中切换的类名来说，如果你使用一个没有名字的 ，则 v- 是这些类名的默认前缀。如果你使用了

```
<transition name="my-transition">
```

，那么 v-enter 会替换为 my-transition-enter。
Vue.js 自定义指令

Vue.js 路由