



Vue.js教程

极客学院出版

前言

Vue.js 是用于构建交互式的 Web 界面的库。它提供了 MVVM 数据绑定和一个可组合的组件系统，具有简单、灵活的 API。从技术上讲，Vue.js 集中在 MVVM 模式上的视图模型层，并通过双向数据绑定连接视图和模型。实际的 DOM 操作和输出格式被抽象出来成指令和过滤器。相比其它的 MVVM 框架，Vue.js 更容易上手。

Vue.js 是一个用于创建 Web 交互界面的库。它让你通过简单而灵活的 API 创建由数据驱动的 UI 组件。

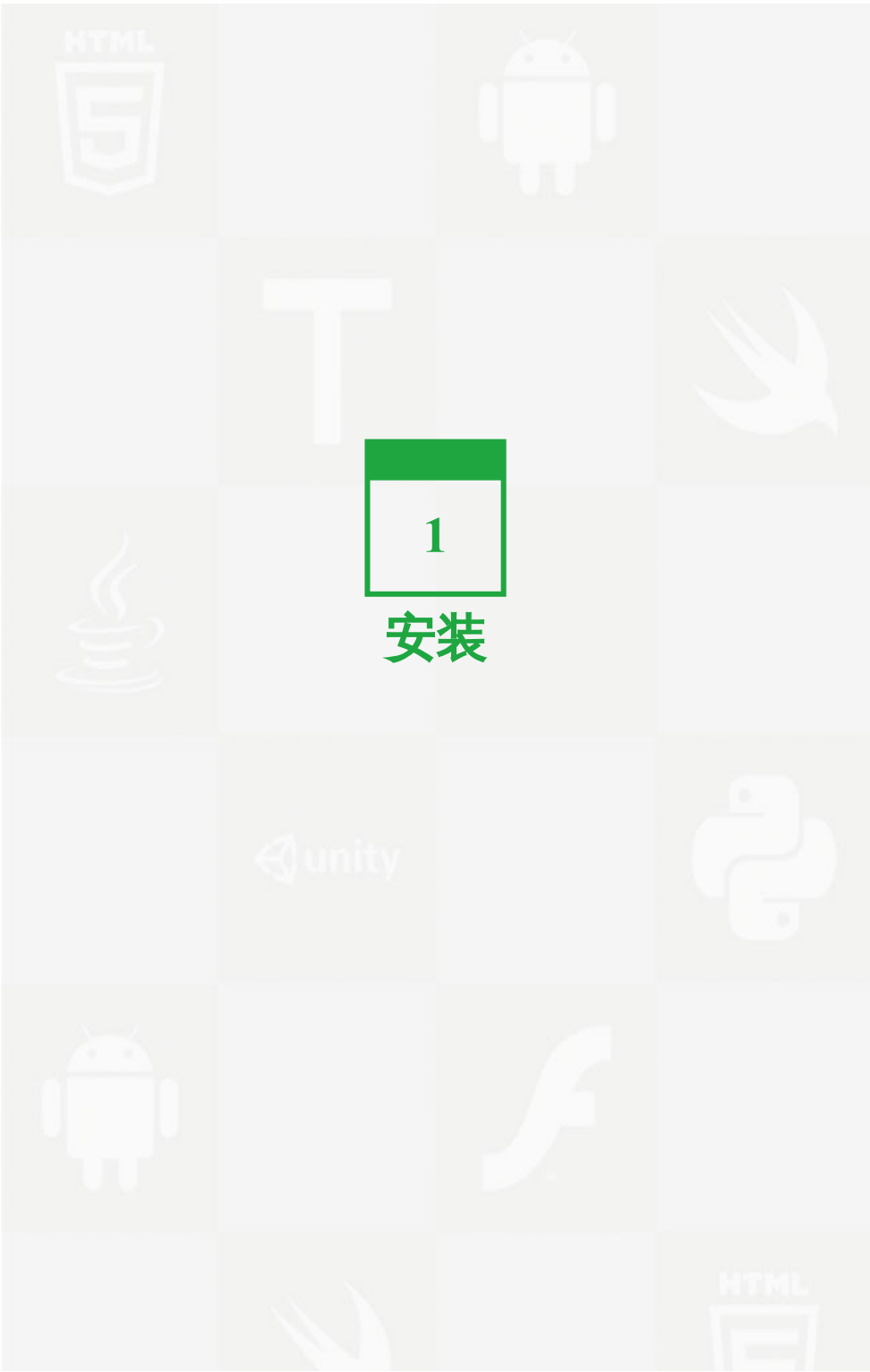
适用人群

即便您已经熟悉了一些这类的库或框架，我们还是推荐您继续阅读接下来的概览，因为您对它们的认识也许和它们在 Vue.js 语境下的定义不尽相同。

学习前提

Vue.js 的 API 是参考了 AngularJS、KnockoutJS、Ractive.js、Rivets.js。所以建议学习前，对上述 4 个框架系统做些了解。

鸣谢：<http://cn.vuejs.org/guide/installation.html>



兼容性提示：Vue.js 不支持 IE8 及其以下版本。

独立版本

直接下载并用 `<script>` 标签引入，`Vue` 就会被注册为一个全局变量。

CDN

也可以在 `jsdelivr` 或 `cdnjs` 获取 (版本更新可能会略滞后)。

CSP 兼容版本

部分环境，诸如 Google Chrome Apps，强制要求内容安全策略 (CSP) 并且不允许使用 `new Function()` 来进行表达式求值。在此情况下，你可以用 `CSP 兼容版本` 代替。

NPM

```
$ npm install vue  
`# 获取CSP兼容版本 :  
`$ npm install vue@csp  
`# 获取最新开发版本(来自于GitHub):  
$ npm install yyx990803/vue#dev
```

Bower

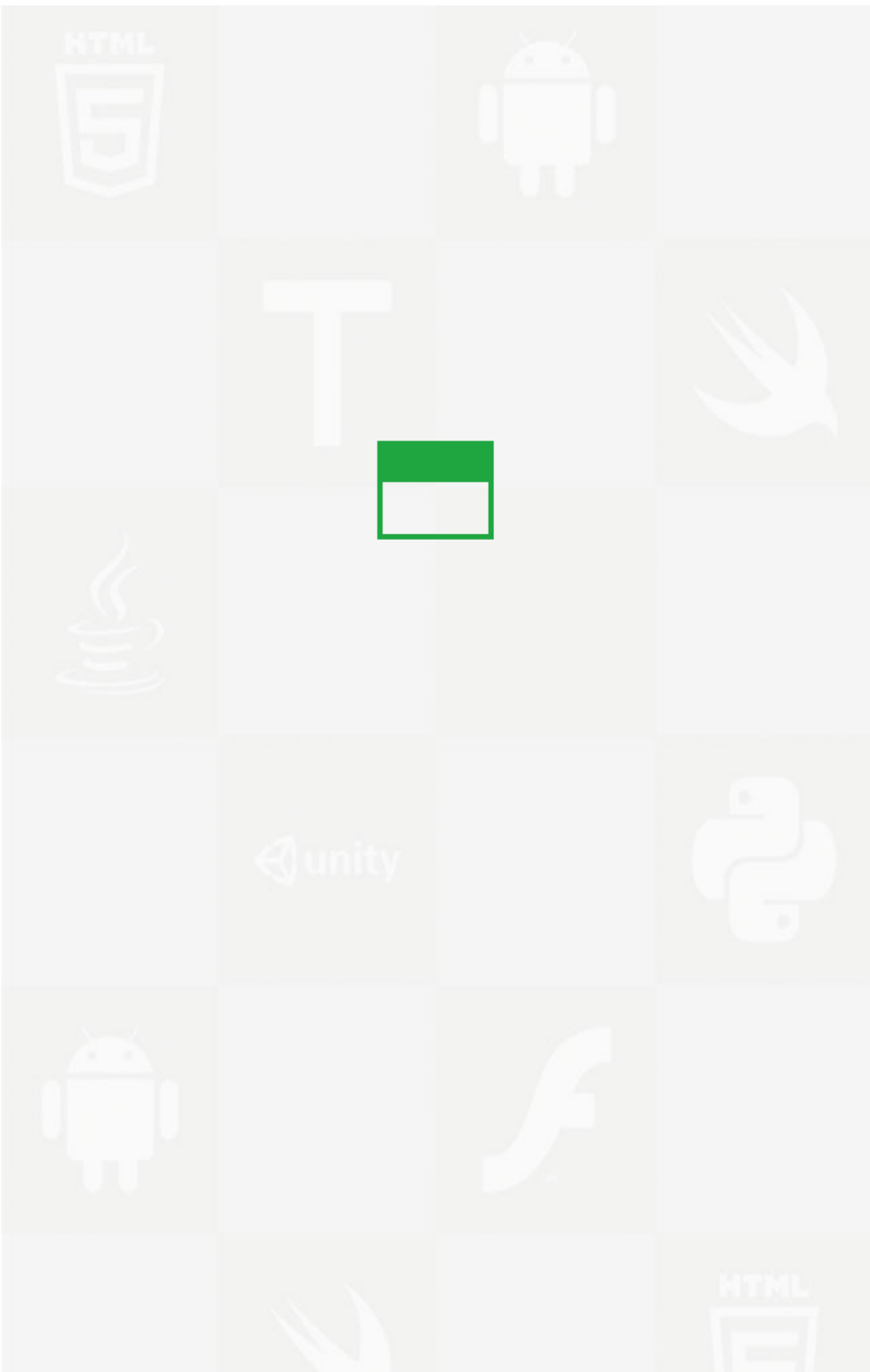
```
`# Bower 只能够获得稳定版本  
$ bower install vue
```

AMD 模块加载器

直接下载或通过 Bower 安装的版本已经用 UMD 接口包装过，可以直接作为 AMD 模块使用。

准备好了吗？

走起！



起步

介绍

Vue.js 是一个用于创建 web 交互界面的库。

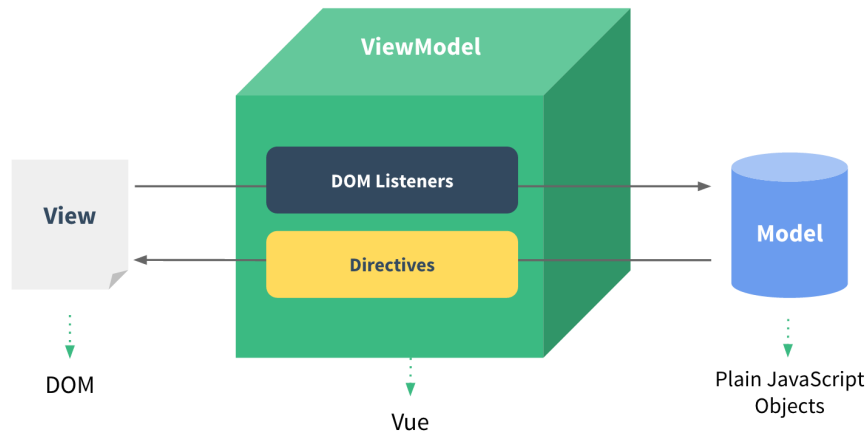
从技术角度讲，Vue.js 专注于 MVVM 模型的 ViewModel 层。它通过双向数据绑定把 View 层和 Model 层连接了起来。实际的 DOM 封装和输出格式都被抽象为了 Directives 和 Filters。

从哲学角度讲，Vue 希望通过一个尽量简单的 API 来提供反应式的数据绑定和可组合、复用的视图组件。它不是一个大而全的框架——它只是一个简单灵活的视图层。您可以独立使用它快速开发原型、也可以混合别的库做更多的事情。它同时和诸如 Firebase 这一类的 BaaS 服务有着天然的契合度。

Vue.js 的 API 设计深受 AngularJS、KnockoutJS、Ractive.js 和 Rivets.js 的影响。尽管有不少相似之处，但我们相信 Vue.js 能够在简约和功能之间的微妙平衡中体现出其独有的价值。

即便您已经熟悉了一些这类的库或框架，我们还是推荐您继续阅读接下来的概览，因为您对它们的认识也许和它们在 Vue.js 语境下的定义不尽相同。

概念综述



ViewModel

一个同步 Model 和 View 的对象。在 `Vue.js` 中，每个 `Vue` 实例都是一个 `ViewModel`。它们是通过构造函数 `Vue` 或其子类被创建出来的。

```
var vm = new Vue({ /* options */ })
```

这是您作为一个开发者在使用 `Vue.js` 时主要打交道的对象。更多的细节请移步至 **Vue 构造函数**。

视图 (View)

被 `Vue` 实例管理的 DOM 节点。

```
vm.$el // The View
```

Vue.js 使用基于 DOM 的模板。每个 Vue 实例都关联着一个相应的 DOM 元素。当一个 Vue 实例被创建时，它会递归遍历根元素的所有子结点，同时完成必要的数据库绑定。当这个视图被编译之后，它就会自动响应数据的变化。

在使用 Vue.js 时，除了自定义指令 (稍后会有解释)，您几乎不必直接接触 DOM。当数据发生变化时，视图将会自动触发更新。这些更新的粒度精确到一个文字节点。同时为了更好的性能，这些更新是批量异步执行的。

模型 (Model)

一个轻微改动过的原生 JavaScript 对象。

```
vm.$data // The Model
```

Vue.js 中的模型就是普通的 JavaScript 对象——也可以称为**数据对象**。一旦某对象被作为 Vue 实例中的数据，它就成为一个“反应式”的对象了。你可以操作它们的属性，同时正在观察它的 Vue 实例也会收到提示。Vue.js 把数据对象的属性都转换成了 ES5 中的 `getter/setters`，以此达到无缝的数据观察效果：无需脏值检查，也不需要刻意给 Vue 任何更新视图的信号。每当数据变化时，视图都会在下一帧自动更新。

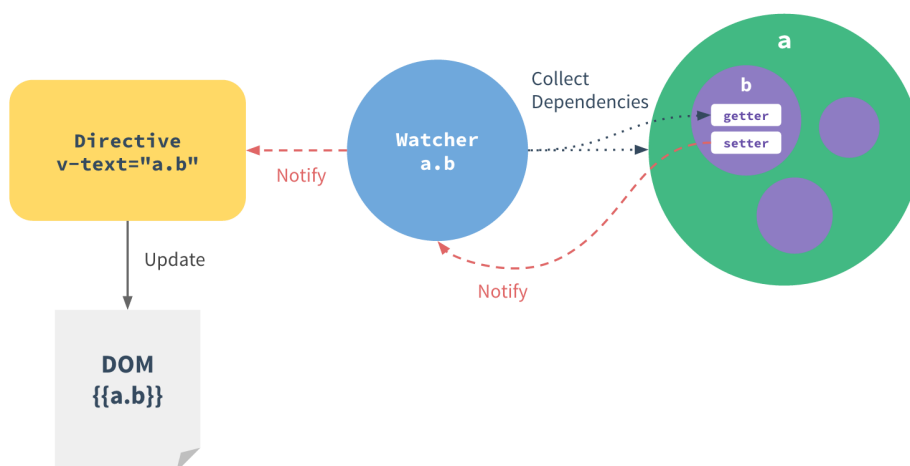
Vue 实例代理了它们观察到的数据对象的所有属性。所以一旦一个对象 `{ a: 1 }` 被观察，那么 `vm.$data.a` 和 `vm.a` 将会返回相同的值，而设置 `vm.a = 2` 则也会修改 `vm.$data`。

数据对象是被就地转化的，所以根据引用修改数据和修改 `vm.$data` 具有相同的效果。这也意味着多个 Vue 实例可以观察同一份数据。在较大型的应用程序中，我们

也推荐将 Vue 实例作为纯粹的视图看待，同时把数据处理逻辑放在更独立的外部数据层。

值得提醒的是，一旦数据被观察，Vue.js 就不会再检测到新加入或删除的属性了。作为弥补，我们会为被观察的对象增加 `$add`，`$set` 和 `$delete` 方法。

下面是对 Vue.js 数据观测机制实现的高层概览：



指令 (Directives)

带特殊前缀的 HTML 特性，可以让 Vue.js 对一个 DOM 元素做各种处理。

```
<div v-text="message"></div>
```

这里的 div 元素有一个 `v-text` 指令，其值为 `message`。Vue.js 会让该 div 的文本内容与 Vue 实例中的 `message` 属性值保持一致。

Directives 可以封装任何 DOM 操作。比如 `v-attr` 会操作一个元素的特性；`v-repeat` 会基于数组来复制一个元素；`v-on` 会绑定事件等。稍后会有更多的介绍。

Mustache 风格绑定

你也可以使用 mustache 风格的绑定，不管在文本中还是在属性中。它们在底层会被转换成 `v-text` 和 `v-attr` 的指令。比如：

```
<div id="person-{{id}}">Hello {{name}}!</div>
```

这很方便，不过有一些注意事项：

- 一个 `` 的 `src` 属性在赋值时会产生一个 HTTP 请求，所以当模板在第一次被解析时会产生一个 404 请求。这种情况下更适合用 `v-attr`。
- Internet Explorer 在解析 HTML 时会移除非法的内联 `style` 属性，所以如果你想支持 IE，请在绑定内联 CSS 的时候始终使用 `v-style`。

你可以使用三对花括号来回避 HTML 代码，而这种写法会在底层转换为 `v-html`：

```
{{{ safeHTMLString }}}}
```

不过这种用法会留下 XSS 攻击的隐患，所以建议只对绝对信任的数据来源使用三对花括号的写法，或者先通过自定义的过滤器 (filter) 对不可信任的 HTML 进行过滤。

最后，你可以在你的 mustache 绑定里加入 `*` 来注明这是一个一次性撰写的数据，这样的话它就不会响应后续的数据变化：

```
{{* onlyOnce }}
```

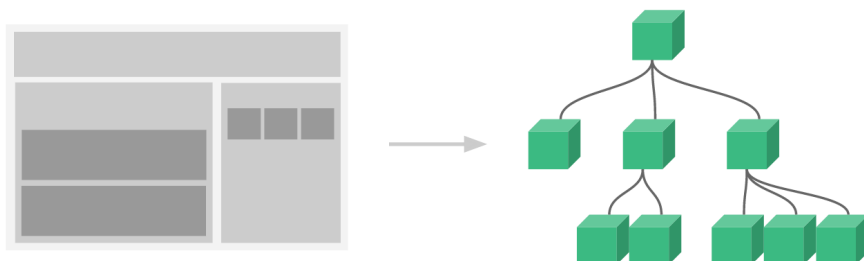
过滤器 (Filters)

过滤器是用于在更新视图之前处理原始值的函数。它们通过一个“管道”在指令或绑定中进行处理：

```
<div>{{message | capitalize}}</div>
```

这样在 div 的文本内容被更新之前，message 的值会先传给 capitalize 函数处理。更多内容可移步至[深入了解过滤器 \(Filters\)](#)。

组件 (Components)



在 Vue.js，每个组件都是一个简单的 Vue 实例。一个树形嵌套的各种组件就代表了你的应用程序的各种接口。通过 Vue.extend 返回的自定义构造函数可以把这些组件实例化，不过更推荐的声明式的用法是通过 Vue.component(id, constructor) 注册这些组件。一旦组件被注册，它们就可以在 Vue 实例的模板中以自定义元素形式使用了。

```
<my-component>  
  <!-- internals handled by my-component -->  
</my-component>
```

这个简单的机制使得我们可以以类似 **Web Components** 的声明形式对 Vue 组件进行复用和组合，同时无需最新版的浏览器或笨重的 polyfills。通过将一个应用程序拆分成简单的组件，代码库可以被尽可能的解耦，同时

更易于维护。更多关于组件的内容，请移步至[组件系统](#)。

简单示例

```
<div id="demo">
  <h1>{{title | uppercase}}</h1>
  <ul>
    <li
      v-repeat="todos"
      v-on="click: done = !done"
      class="{{done ? 'done' : ''}}">
      {{content}}
    </li>
  </ul>
</div>
```

```
var demo = new Vue({
  el: '#demo',
  data: {
    title: 'todos',
    todos: [
      {
        done: true,
        content: 'Learn JavaScript'
      },
      {
        done: false,
        content: 'Learn Vue.js'
      }
    ]
  }
})
```

Result

TODOS

- Learn JavaScript

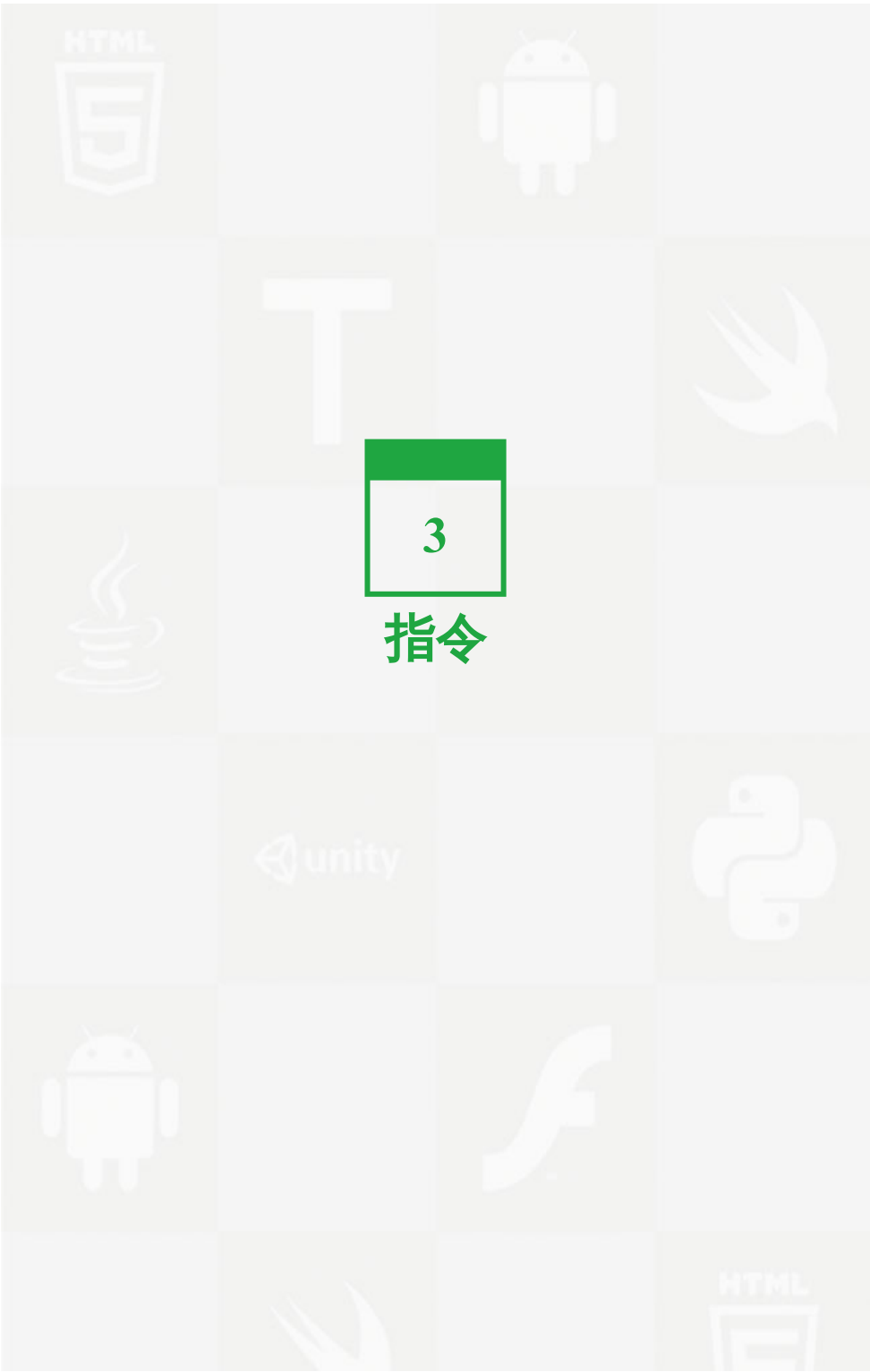
- Learn Vue.js

也可以通过 **jsfiddle** 查看。

你可以点击一个 todo 来开关它，也可以打开你的浏览器命令行直接操作 `demo` 对象：比如改变 `demo.title`、向 `demo.todos` 里放入一个新的对象、或开关某个 todo 的 `done` 状态值。

也许你现在脑子里有很多问题，别担心，我们稍后都会一一提到的。

下一步：[深入了解指令 \(Directives\)](#)。



概要

如果你没有用过 AngularJS，你可能不太清楚指令 (directive) 是什么。一个指令的本质是模板中出现的特殊标记，让处理模板的库知道需要对这里的 DOM 元素进行一些对应的处理。Vue.js 的指令概念相比 Angular 要简单得多。Vue.js 中的指令只会以带前缀的 HTML 特性 (attribute) 的形式出现：

```
<element  
  prefix-directiveId="[argument:] expression [|  
  filters...]">  
</element>
```

简单示例

```
<div v-text="message"></div>
```

这里的前缀是默认的 `v-`。指令的 ID 是 `text`，表达式是 `message`。这个指令告诉 Vue.js，当 Vue 实例的 `message` 属性改变时，更新该 `div` 元素的 `textContent`。

内联表达式

```
<div v-text="'hello ' + user.firstName + ' ' + user.lastName"></div>
```

这里我们使用了一个计算表达式 (computed expression)，而不仅仅是简单的属性名。Vue.js 会自动跟踪表达式中依赖的属性并在这些依赖发生变化时触发指令更新。同时，因为有异步批处理更新机制，哪怕多个依赖同时变化，表达式也只会触发一次。

你应该明智地使用表达式，并避免在你的模板里放入过多的逻辑，尤其是有副作用的语句 (事件监听表达式除外)。为了防止在模板内滥用逻辑，Vue.js 把内联表达式限制为**一条语句**。如果需要绑定更复杂的操作，请使用[计算属性](#)。

- 出于安全考虑，在内联表达式中你只能访问当前上下文中 Vue 实例的属性和方法。

参数

```
<div v-on="click : clickHandler"></div>
```

有些指令需要在路径或表达式前加一个参数。在这个例子中 `click` 参数代表了我们希望 `v-on` 指令监听到点击事件之后调用该 ViewModel 实例的 `clickHandler` 方法。

过滤器

过滤器可以紧跟在指令的路径或表达式之后，在更新 DOM 之前对值进行进一步处理。过滤器像 shell 脚本一样跟在一个管道符 (|) 之后。更多内容请移步至 [深入了解过滤器](#)。

多重指令从句

你可以在同一个特性里多次绑定同一个指令。这些绑定用逗号分隔，它们在底层被分解为多个指令实例进行绑定。

```
<div v-on="
  click    : onClick,
  keyup    : onKeyUp,
  keydown  : onKeyDown
">
</div>
```

字面量指令

有些指令不会创建数据绑定——它们的值只是一个字符串字面量。比如 `v-ref` 指令：

```
<my-component v-ref="some-string-id"></my-component>
```

这里的 `"some-string-id"` 并不是一个反应式的表达式——Vue.js 不会尝试去观测组件中的对应数据。

在有些情况下，你也可以使用 Mustache 风格绑定来使得字面量指令“反应化”：

```
<div v-show="showMsg" v-transition="{{dynamicTransitionId}}"></div>
```

但是，请注意只有 `v-transition` 指令具有此特性。Mustache 表达式在其他字面量指令中，例如 `v-ref` 和 `v-el`，只会被计算一次。它们在编译完成后将不会再响应数据的变化。

完整的字面量指令列表可以在 [API 索引](#) 中找到。

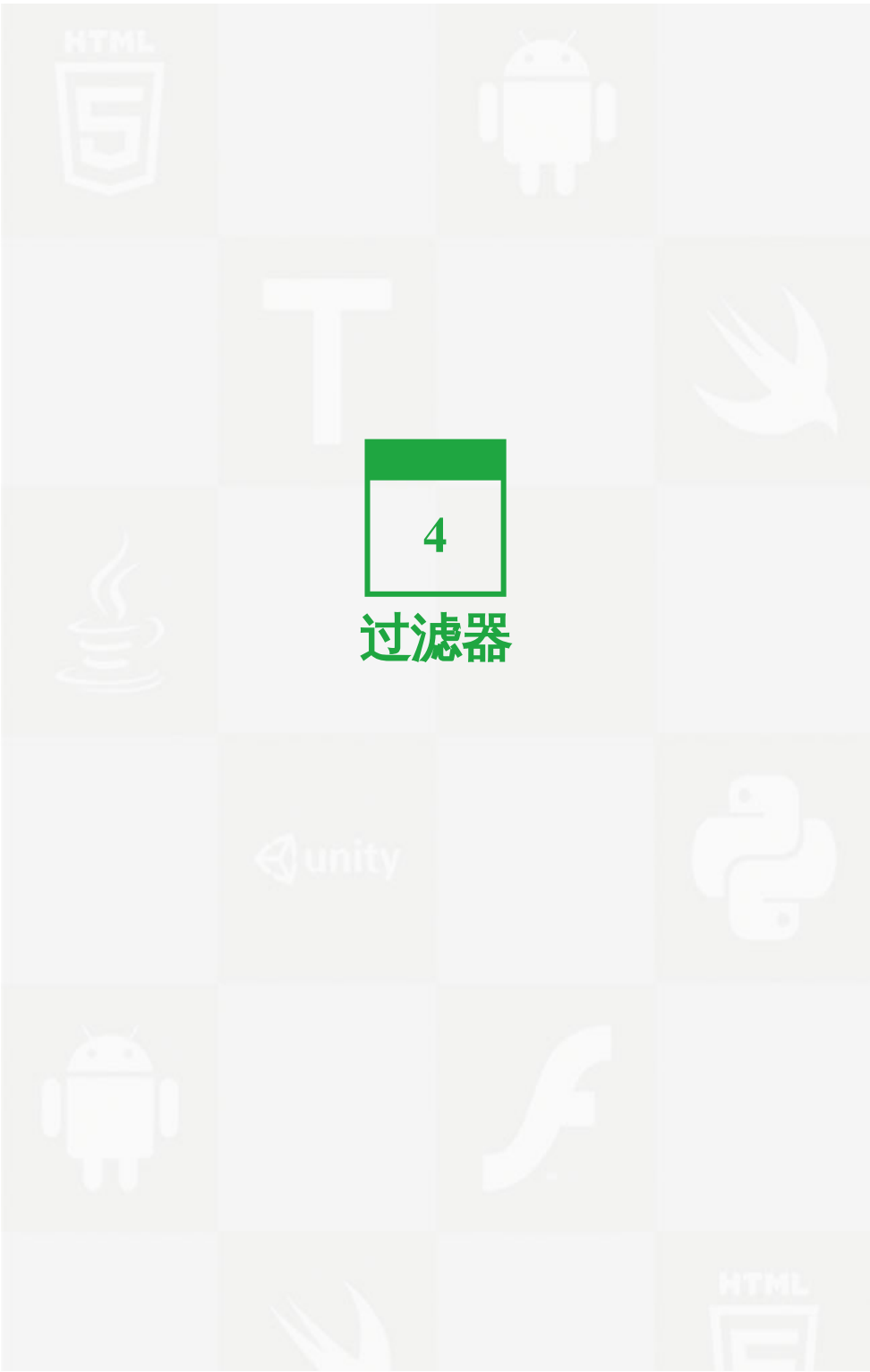
空指令

有些指令并不需要判断特性的值 —— 这些操作对某个元素处理且仅处理一次。比如 `v-pre` 指令：

```
<div v-pre>
  <!-- 内部模板将不会被编译 -->
</div>
```

完整的空指令列表可以在 [API 索引](#) 中找到。

下一步：[过滤器](#)。



概要

一个 Vue.js 的过滤器本质上是一个函数，这个函数会接收一个值，将其处理并返回。过滤器在指令中由一个管道符 (|) 标记，并可以跟随一个或多个参数：

```
<element directive="expression | filterId  
[args...]"></element>
```

示例

过滤器必须放置在一个指令的值的最后：

```
<span v-text="message | capitalize"></span>
```

你也可以用在 mustache 风格的绑定的内部：

```
<span>{{message | uppercase}}</span>
```

可以串联多个过滤器：

```
<span>{{message | lowercase | reverse}}</span>
```

参数

一些过滤器是可以接受参数的。参数用空格分隔开：

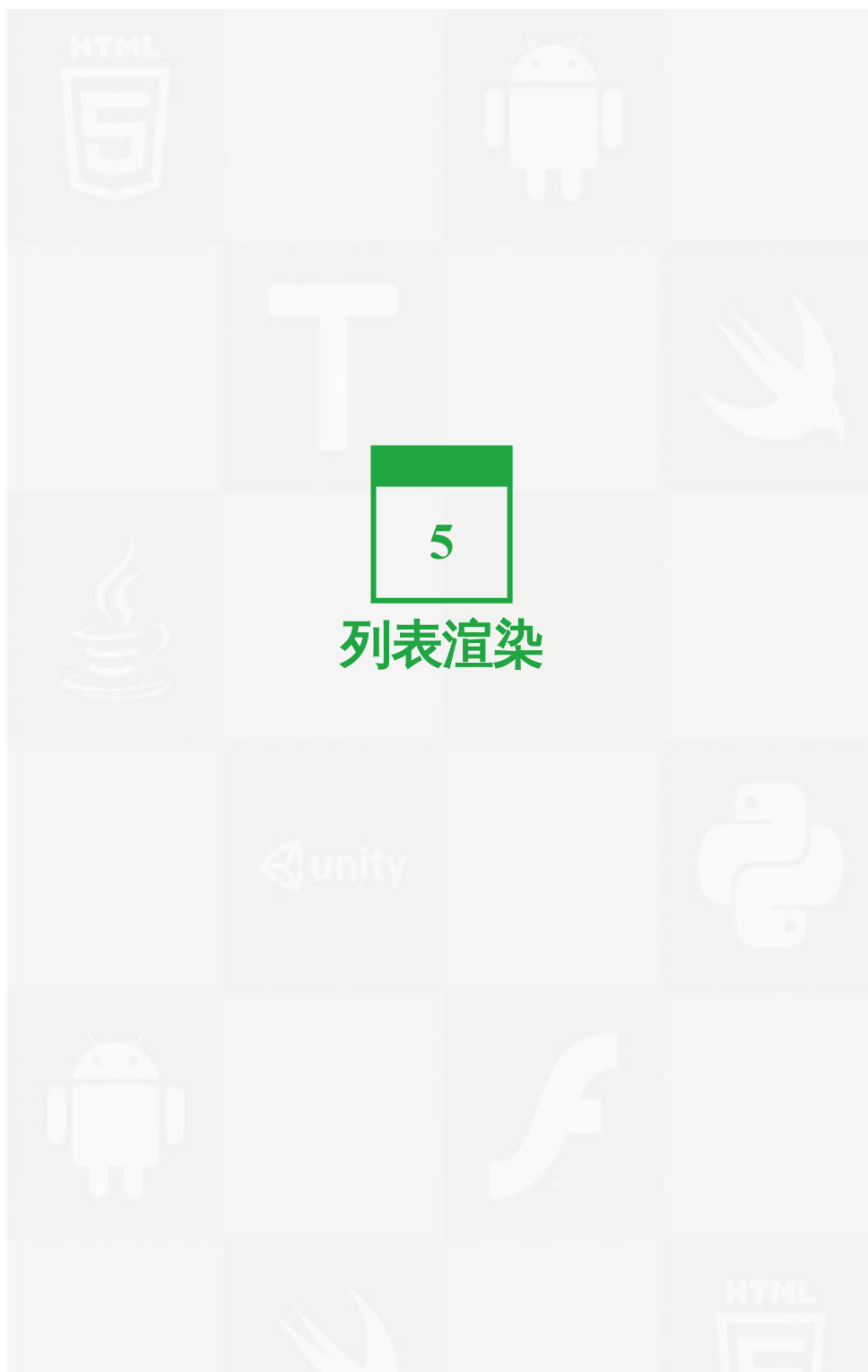
```
<span>{{order | pluralize 'st' 'nd' 'rd' 'th'}}</span>
```

```
<input v-on="keyup: submitForm | key 'enter'">
```

纯字符串参数需要用引号包裹。无引号的参数会作为表达式在当前数据作用域内动态计算。在后面讲到自定义过滤器时，我们进一步讨论这些细节。

上述示例的具体用法参见完整的内建过滤器列表。

现在你已经了解了指令和过滤器，接下来我们趁热打铁，看看如何 [展示一个列表](#)。



你可以使用 `v-repeat` 指令来基于 ViewModel 上的对象数组渲染列表。对于数组中的每个对象，该指令将创建一个以该对象作为其 `$data` 对象的子 Vue 实例。这些子实例继承父实例的数据作用域，因此在重复的模板元素中你既可以访问子实例的属性，也可以访问父实例的属性。此外，你还可以通过 `$index` 属性来获取当前实例对应的数组索引。

示例：

```
<ul id="demo">
  <li v-repeat="items" class="item-{{$index}}">
    {{$index}} - {{parentMsg}} {{childMsg}}
  </li>
</ul>
```

```
var demo = new Vue({
  el: '#demo',
  data: {
    parentMsg: 'Hello',
    items: [
      { childMsg: 'Foo' },
      { childMsg: 'Bar' }
    ]
  }
})
```

结果：

- 0 - Hello Foo
- 1 - Hello Bar

块级重复

有时我们可能需要重复一个包含多个节点的块，这时可以用 `<template>` 标签包裹这个块。这里的 `<template>` 标签只起到语义上的包裹作用，其本身不会被渲染出来。例如：

```
<ul>
  <template v-repeat="list">
    <li>{{msg}}</li>
    <li class="divider"></li>
  </template>
</ul>
```

简单值数组

简单值 (primitive value) 即字符串、数字、boolean 等非对象的值。对于包含简单值的数组，你可用 `$value` 直接访问值：

```
<ul id="tags">
  <li v-repeat="tags">
    {{$value}}
  </li>
</ul>
```

```
new Vue({
  el: '#tags',
  data: {
    tags: ['JavaScript', 'MVVM', 'Vue.js']
  }
})
```

结果：

- JavaScript
- MVVM
- Vue.js

使用别名

有时我们可能想要更明确地访问当前作用域的变量而不是隐式地回退到父作用域。你可以通过提供一个参数给 `v-repeat` 指令并用它作为将被迭代项的别名：

```
<ul id="users">
  <li v-repeat="user in users">
    {{user.name}} - {{user.email}}
  </li>
</ul>
```

```
new Vue({
  el: '#users',
  data: {
    users: [
      { name: 'Foo Bar', email: 'foo@bar.com' },
      { name: 'John Doh', email: 'john@doh.com' }
    ]
  }
})
```

结果：

- Foo Bar - foo@bar.com
- John Doh - john@doh.com

这里的 `user in users` 语法只在 `Vue 0.12.8` 及其以上版本中可用。老版本需要使用 `user : users` 语法。

在 `v-repeat` 中使用别名会让模板可读性更强，同时性能更好。

变异方法

Vue.js 内部对被观察数组的变异方法 (mutating methods, 包括 `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `sort()` 和 `reverse()`) 进行了拦截, 因此调用这些方法也将自动触发视图更新。

```
// 以下操作会触发 DOM 更新
demo.items.unshift({ childMsg: 'Baz' })
demo.items.pop()
```

扩展方法

Vue.js 给被观察数组添加了两个便捷方法：`$set()` 和 `$remove()`。

你应该避免直接通过索引来设置数据绑定数组中的元素，比如 `demo.items[0] = {}`，因为这些改动是无法被 Vue.js 侦测到的。你应该使用扩展的 `$set()` 方法：

```
// 不要用 `demo.items[0] = ...`  
demo.items.$set(0, { childMsg: 'Changed!' })
```

`$remove()` 只是 `splice()` 方法的语法糖。它将移除给定索引处的元素。当参数不是数值时，`$remove()` 将在数组中搜索该值并删除第一个发现的对应元素。

```
// 删除索引为 0 的元素。  
demo.items.$remove(0)
```

替换数组

当你使用非变异方法，比如 `filter()`，`concat()` 或 `slice()`，返回的数组将是一个不同的实例。在此情况下，你可以用新数组替换旧的数组：

```
demo.items = demo.items.filter(function (item)
{
  return item.childMsg.match(/Hello/)
})
```

你可能会认为这将导致整个列表的 DOM 被销毁并重新渲染。但别担心，Vue.js 能够识别一个数组元素是否已有关联的 Vue 实例，并尽可能地进行有效复用。

使用 track-by

在某些情况下，你可能需要以全新的对象（比如 API 调用所返回的对象）去替换数组。如果你的每一个数据对象都有一个唯一的 id 属性，那么你可以使用 `track-by` 特性参数给 Vue.js 一个提示，这样它就可以复用已有的具有相同 id 的 Vue 实例和 DOM 节点。

例如，如果你的数据长这样：

```
{
  items: [
    { _uid: '88f869d', ... },
    { _uid: '7496c10', ... }
  ]
}
```

那么你可以像这样给出提示：

```
<div v-repeat="items" track-by="_uid">
  <!-- content -->
</div>
```

在替换 `items` 数组时，Vue.js 如果碰到一个有 `_uid: '88f869d'` 的新对象，它就会知道可以直接复用有同样 `_uid` 的已有实例。

如果没有唯一的 id 可以用来追踪，也可以使用 `track-by="$index"`，也就是原地复用实例和 DOM 节点。请务必小心使用此功能，因为在追踪 `$index` 时，Vue 不会移动子实例及 DOM 节点，只是按他们第一次被创建时的顺序复用。有两种情形应避免使用 `track-by="$index"`：

1. 当重复的块包含表单输入框，且输入框绑定的值可能导致数组被重新排序；
2. 在重复组件时，组件除了接收的数组数据外，还包含私有的状态。

在使用全新数据重新渲染大型 `v-repeat` 列表时，合理使用 `track-by` 能极大地提升性能。

遍历对象

你也可以使用 `v-repeat` 遍历一个对象的所有属性。每个重复的实例会有一个特殊的属性 `$key`。对于简单值，你也可以象访问数组中的简单值那样使用 `$value` 属性。

```
<ul id="repeat-object">
  <li v-repeat="primitiveValues">{{ $key }} :
  {{ $value }}</li>
  <li>===</li>
  <li v-repeat="objectValues">{{ $key }} :
  {{ msg }}</li>
</ul>
```

```
new Vue({
  el: '#repeat-object',
  data: {
    primitiveValues: {
      FirstName: 'John',
      LastName: 'Doe',
      Age: 30
    },
    objectValues: {
      one: {
        msg: 'Hello'
      },
      two: {
        msg: 'Bye'
      }
    }
  }
})
```

结果：

- `FirstName : John`
- `LastName : Doe`
- `Age : 30`
- `===`
- `one : Hello`
- `two : Bye`

在 ECMAScript 5 中，对于给对象添加一个新属性，或是从对象中删除一个属性时，没有机制可以检测到这两种情况。针对这个问题，Vue.js 中的被观察对象会被添加三个扩展方法: `$add(key, value)` , `$set(key, value)` 和 `$delete(key)`。这些方法可以被用于在添加 / 删除观察对象的属性时触发对应的视图更新。方法 `$add` 和 `$set` 的不同之处在于当指定的键已经在对象中存在时 `$add` 将提前返回，所以调用 `obj.$add(key)` 并不会以 `undefined` 覆盖已有的值。

迭代值域

`v-repeat` 也可以接受一个整数。在这种情况下，它将重复显示一个模板多次。

```
<div id="range">
  <div v-repeat="val">Hi! {{$index}}</div>
</div>```

```new Vue({
 el: '#range',
 data: {
 val: 3
 }
});
```

结果：

- Hi! 0
- Hi! 1
- Hi! 2

## 数组过滤器

---

有时候我们只需要显示一个数组的过滤或排序过的版本，而不需要实际改变或重置原始数据。Vue 提供了两个内置的过滤器来简化此类需求：`filterBy` 和 `orderBy`。可查看 **方法文档** 获得更多细节。

下一步: [事件监听](#)。



6

事件监听

你可以使用 `v-on` 指令来绑定并监听 DOM 事件。绑定的内容可以是一个当前实例上的方法 (后面无需跟括号) 或一个内联表达式。如果提供的是一个方法，则原生的 DOM event 会被作为第一个参数传入，同时这个 event 会带有 `targetVM` 属性，指向触发该事件的相应的 ViewModel：

```
<div id="demo">
 <a v-on="click: onClick">触发一个方法函数
 <a v-on="click: n++">触发一个表达式
</div>
```

```
new Vue({
 el: '#demo',
 data: {
 n: 0
 },
 methods: {
 onClick: function (e) {
 console.log(e.target.tagName) // "A"
 console.log(e.targetVM === this) // true
 }
 }
})
```

## 执行表达式

---

当在 `v-repeat` 里使用 `v-on` 时，`targetVM` 显得很有用，因为 `v-repeat` 会创建大量 `ViewModel`。但是，通过执行表达式的方式，把代表当前 `ViewModel` 数据对象的别名传进去，会更方便直观一些：

```
<ul id="list">
 <li v-repeat="item in items" v-on="click:
toggle(item)">
 {{item.text}}


```

```
new Vue({
 el: '#list',
 data: {
 items: [
 { text: 'one', done: true },
 { text: 'two', done: false }
]
 },
 methods: {
 toggle: function (item) {
 item.done = !item.done
 }
 }
})
```

当你想要在表达式中访问原来的 DOM event，你可以传递一个 `$event` 参数进去：

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



```
/* ... */
{
 methods: {
 submit: function (msg, e) {
 e.stopPropagation()
 }
 }
}
/* ... */
```

## key 过滤器

---

当监听键盘事件时，我们常常需要判断常用的 `key code`。Vue.js 提供了一个特殊的只能用在 `v-on` 指令的过滤器：`key`。它接收一个表示 `key code` 的参数并完成判断：

```
<!-- 只有当 keyCode 等于 13 时才调用方法 -->
<input v-on="keyup:submit | key 13">
```

它也预置了一些常用的按键名：

```
<!-- 效果同上 -->
<input v-on="keyup:submit | key 'enter'">
```

查看 API 参考：[key 过滤器的全部预设参数](#)。

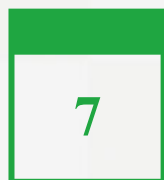
## 为什么要在 HTML 中写监听器？

---

你可能会注意到整个事件监听的方式违背了“separation of concern”的传统理念。不必担心，因为所有的 Vue.js 时间处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护困难。实际上，使用 `v-on` 还有更多好处：

1. 它便于在 HTML 模板中轻松定位 JS 代码里的对应方法实现。
2. 因为你无须在 JS 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦。这会更易于测试。
3. 当一个 ViewModel 被销毁时，所有的事件监听都会被自动移除。你无须担心如何自行清理它们。

下一步：[处理表单](#)。



## 处理表单

## 基本用法

---

你可以在表单的 `input` 元素上使用 `v-model` 指令来创建双向数据绑定。它会根据 `input` 元素的类型自动选取正确的绑定模式。

### 示例

```
<form id="demo">
 <!-- text -->
 <p>
 <input type="text" v-model="msg">
 {{msg}}
 </p>
 <!-- checkbox -->
 <p>
 <input type="checkbox" v-model="checked">
 {{checked ? "yes" : "no"}}
 </p>
 <!-- radio buttons -->
 <p>
 <input type="radio" name="picked"
value="one" v-model="picked">
 <input type="radio" name="picked"
value="two" v-model="picked">
 {{picked}}
 </p>
 <!-- select -->
 <p>
 <select v-model="selected">
 <option>one</option>
 <option>two</option>
 </select>
 {{selected}}
 </p>
 <!-- multiple select -->
 <p>
```

```
<select v-model="multiSelect" multiple>
 <option>one</option>
 <option>two</option>
 <option>three</option>
</select>
{{multiSelect}}
</p>
<p><pre>data: {{$data | json 2}}</pre></p>
</form>
```

```
new Vue({
 el: '#demo',
 data: {
 msg : 'hi!',
 checked : true,
 picked : 'one',
 selected : 'two',
 multiSelect: ['one', 'three']
 }
})
```

**效果**

hi!

☒ yes

☒ ☐ one

two

data:

```
{
 "msg": "hi!",
 "checked": true,
 "picked": "one",
 "selected": "two",
 "multiSelect": [
 "one",
 "three"
]
}
```

## 惰性更新

---

默认情况下，`v-model` 会在每个 `input` 事件之后同步输入的数据。你可以添加一个 `lazy` 特性，将其改变为在 `change` 事件之后才进行同步。

```
<!-- 在 "change" 而不是 "input" 事件触发后进行
同步 -->
<input v-model="msg" lazy>
```



## 转换为数字

---

如果你希望将用户的输入自动转换为数字，你可以在 `v-model` 所在的 `input` 上添加一个 `number` 特性。

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

## 动态 select 选项

---

当你需要为一个 `<select>` 元素动态渲染列表选项时，推荐将 `options` 特性和 `v-model` 指令配合使用，这样当选项动态改变时，`v-model` 会正确地同步：

```
<select v-model="selected" options="myOptions">
</select>
```

在你的数据里，`myOptions` 应该是一个指向选项数组的路径或是表达式。

该数组可以包含普通字符串或对象。对象的格式应为 `{text:'', value:''}`。这允许你把展示的文字和其背后对应的值区分开来。

```
[
 { text: 'A', value: 'a' },
 { text: 'B', value: 'b' }
]
```

会渲染成：

```
<select>
 <option value="a">A</option>
 <option value="b">B</option>
</select>
```

另外，数组里对象的格式也可以是 `{label:'', options:[...]}`。这样的数据会被渲染成为一个 `<optgroup>`：

```
[
 { label: 'A', options: ['a', 'b'] },
 { label: 'B', options: ['c', 'd'] }
]
```

会渲染成：

```
<select>
 <optgroup label="A">
 <option value="a">a</option>
 <option value="b">b</option>
 </optgroup>
 <optgroup label="B">
 <option value="c">c</option>
 <option value="d">d</option>
 </optgroup>
</select>
```

你的原始数据很有可能不是这里所要求的格式，因此在动态生成选项时必须进行一些数据转换。为了简化这种转换，`options` 特性支持过滤器。将数据的转换逻辑做成一个可复用的 **自定义过滤器**。通常来说是个好主意：

```
Vue.filter('extract', function (value,
keyToExtract) {
 return value.map(function (item) {
 return item[keyToExtract]
 })
})
```

```
<select
 v-model="selectedUser"
 options="users | extract 'name'">
</select>
```

上述过滤器将像 `[{ name: 'Bruce' }, { name: 'Chuck' }]` 这样的原始数据转化为 `['Bruce', 'Chuck']`，从而符合动态选项的格式要求。

## 输入 Debounce

---

在一次输入被同步到模型之前，`debounce` 特性允许你设置一个每次用户事件后的等待延迟。如果在这个延迟到期之前用户再次输入，则不会立刻触发更新，而是重置延迟的等待时间。当每次更新前你要执行繁重作业时会有用，例如一个基于 `ajax` 的自动补全功能。

```
<input v-model="msg" debounce="500">
```

### 结果

A screenshot of a web application. It features a light gray background. In the center, there is a blue text label "edit me". Below this label, there is a white rectangular input field with a thin gray border, containing the text "edit me".

注意 `debounce` 参数并不对用户的输入事件进行 `debounce`：它只对底层数据的“写入”操作起作用。因此当使用 `debounce` 时，你应该用 `vm.$watch()` 而不是 `v-on` 来响应数据变化。

下一步：[计算属性](#)



Vue.js 的内联表达式非常方便，但它最合适的使用场景是简单的布尔操作或字符串拼接。如果涉及更复杂的逻辑，你应该使用**计算属性**。

在 Vue.js 中，你可以通过 `computed` 选项定义计算属性：

```
var demo = new Vue({
 data: {
 firstName: 'Foo',
 lastName: 'Bar'
 },
 computed: {
 fullName: {
 // getter 应返回计算后的值
 get: function () {
 return this.firstName + ' ' +
this.lastName
 },
 // setter 是可选的
 set: function (newValue) {
 var names = newValue.split(' ')
 this.firstName = names[0]
 this.lastName = names[names.length - 1]
 }
 }
 }
})
demo.fullName // 'Foo Bar'
```

当你只需要 `getter` 的时候，你可以直接提供一个函数：

```
computed: {
 fullName: function () {
 return this.firstName + ' ' + this.lastName
 }
}
```

一个计算属性本质上是一个被 `getter/setter` 函数定义了的属性。计算属性使用起来和一般属性一样，只是在访问它的时候，你会得到 `getter` 函数返回的值，改变它的时候，你会触发 `setter` 函数，新值将会作为 `setter` 的参数被传入。

在 `0.12.8` 之前，计算属性仅仅体现为一个取值的行为——每次你访问它的时候，`getter` 都会重新求值。在 `0.12.8` 中对此做了改进——计算属性的值会被缓存，只有在需要时才会重新计算。

接下来，让我们学习一下如何[编写自定义指令](#)。

9

## 自定义指令



## 基础

---

Vue.js 允许你注册自定义指令，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。你可以使用 `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。

### Example

```
“Vue.directive('my-directive', { bind: function () { // 做绑定的准备工作 // 比如添加事件监听器，或是其他只需要执行一次的复杂操作 }, update: function (newValue, oldValue) { // 根据获得的新值执行对应的更新 // 对于初始值也会被调用一次 }, unbind: function () { // 做清理工作 // 比如移除在 bind() 中添加的事件监听器 } })
```

一旦注册好自定义指令，你就可以在 `Vue.js` 模板中像这样来使用它（需要添加 `Vue.js` 的指令前缀，默认为 ``v-``）：

```
`<div v-my-directive="someValue"></div>`
```

如果你只需要 ``update`` 函数，你可以只传入一个函数，而不用传定义对象：

```
```Vue.directive('my-directive', function
(value) {
  // 这个函数会被作为 update() 函数使用
})```
```

所有的钩子函数会被复制到实际的**指令对象**中，而这个指令对象将会是所有钩子函数的 `this` 上下文环境。指令对象上暴露了一些有用的公开属性：

- ****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
- ****raw****：未被解析的原始表达式
- ****name****：不带前缀的指令名

>这些属性是只读的，不要修改它们。你也可以给指令对象附加自定义的属性，但是注意不要覆盖已有的内部属性。

使用指令对象属性的示例：

```
`<div id="demo" v-demo="LightSlateGray : msg">
</div>`
```

```
```Vue.directive('demo', {
 bind: function () {
 this.el.style.color = '#fff'
 this.el.style.backgroundColor = this.arg
 },
 update: function (value) {
 this.el.innerHTML =
 'name - ' + this.name + '
' +
 'raw - ' + this.raw + '
' +
 'expression - ' + this.expression +
 '
' +
 }
})```
```

```

 'argument - ' + this.arg + '
' +
 'value - ' + value
 }
})
var demo = new Vue({
 el: '#demo',
 data: {
 msg: 'hello!'
 }
})``

```

**\*\*Result\*\***

```

- name - demo
- raw - LightSlateGray : msg
- expression - msg
- argument - LightSlateGray
- value - hello!

```

### ### 多重从句

同一个特性内部，逗号分隔的多个从句将被绑定为多个指令实例。在下面的例子中，指令会被创建和调用两次：

```

`<div v-demo="color: 'white', text: 'hello!'">
</div>`

```

如果想要用单个指令实例处理多个参数，可以利用字面量对象作为表达式：

```

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

```

```

``Vue.directive('demo', function (value) {
 console.log(value) // Object {color: 'white',
text: 'hello!'}
})``

```

## ## 字面指令

如果在创建自定义指令的时候传入 ``isLiteral: true``，那么特性值就会被看成直接字符串，并被赋值给该指令的 ``expression``。字面指令不会试图建立数据监视。

**\*\*Example\*\* :**

```
<div v-literal-dir="foo"></div>`

``Vue.directive('literal-dir', {
 isLiteral: true,
 bind: function () {
 console.log(this.expression) // 'foo'
 }
})``
```

## ### 动态字面指令

然而，在字面指令含有 ``Mustache`` 标签的情形下，指令的行为如下：

- 指令实例会有一个属性，``this._isDynamicLiteral`` 被设为 ``true``；
- 如果没有提供 ``update`` 函数，``Mustache`` 表达式只会被求值一次，并将该值赋给 ``this.expression``。不会对表达式进行数据监视。
- 如果提供了 ``update`` 函数，指令将会为表达式建立一个数据监视，并且在计算结果变化的时候调用 ``update``。

## ## 双向指令

如果你的指令想向 Vue 实例写回数据，你需要传入

`twoWay: true` 。该选项允许在指令中使用  
`this.set(value)`。

```
```Vue.directive('example', {
  twoWay: true,
  bind: function () {
    this.handler = function () {
      // 把数据写回 vm
      // 如果指令这样绑定 v-example="a.b.c",
      // 这里将会给 `vm.a.b.c` 赋值
      this.set(this.el.value)
    }.bind(this)
    this.el.addEventListener('input',
this.handler)
  },
  unbind: function () {
    this.el.removeEventListener('input',
this.handler)
  }
})```
```

内联语句

传入 `acceptStatement: true` 可以让自定义指令像
`v-on` 一样接受内联语句：

```
`<div v-my-directive="a++"></div>`
```

```
```Vue.directive('my-directive', {
 acceptStatement: true,
 update: function (fn) {
 // the passed in value is a function which
when called,
 // will execute the "a++" statement in the
owner vm's
 // scope.
 }
})```
```

但是请明智地使用此功能，因为通常我们希望避免在模板中产生副作用。

## ## 深度数据观察

如果你希望在一个对象上使用自定义指令，并且当对象内部嵌套的属性发生变化时也能够触发指令的`update`函数，那么你就要在指令的定义中传入`deep: true`。

```
<div v-my-directive="obj"></div>`

``Vue.directive('my-directive', {
 deep: true,
 update: function (obj) {
 // 当 obj 内部嵌套的属性变化时也会调用此函数
 }
})``
```

## ## 指令优先级

你可以选择给指令提供一个优先级数（默认是 0）。同一个元素上优先级越高的指令会比其他的指令处理得早一些。优先级一样的指令会按照其在元素特性列表中出现的顺序依次处理，但是不能保证这个顺序在不同的浏览器中是一致的。

通常来说作为用户，你并不需要关心内置指令的优先级，如果你感兴趣的话，可以参阅源码。逻辑控制指令`v-repeat`，`v-if`被视为“终结性指令”，它们在编译过程中始终拥有最高的优先级。

## ## 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：

```
```Vue.elementDirective('my-directive', {  
  // 和普通指令的 API 一致  
  bind: function () {  
    // 对 this.el 进行操作...  
  }  
})
```

使用时我们不再用这样的写法：

```
<div v-my-directive></div>
```

而是写成：

```
<my-directive></my-directive>
```

元素指令不能接受参数或表达式，但是它可以读取元素的特性，来决定它的行为。

与通常的指令有个很大的不同，元素指令是**终结性的**，这意味着，一旦 Vue 遇到一个元素指令，它将跳过对该元素和其子元素的编译 - 即只有该元素指令本身可以操作该元素及其子元素。

下面，我们来看怎样写一个 [自定义过滤器](#)。

10

自定义过滤器

基础

和自定义指令类似，你可以用全局方法

`Vue.filter()`，传递一个**过滤器 ID** 和一个**过滤器函数**来注册一个自定义过滤器。过滤器函数会接受一个参数值并返回将其转换后的值：

```
Vue.filter('reverse', function (value) {  
  return value.split('').reverse().join('')  
})
```

```
<!-- 'abc' => 'cba' -->  
<span v-text="message | reverse"></span>
```

过滤器函数也可以接受内联参数：

```
Vue.filter('wrap', function (value, begin, end)  
{  
  return begin + value + end  
})
```

```
<!-- 'hello' => 'before hello after' -->  
<span v-text="message | wrap 'before' 'after'">  
</span>
```

双向过滤器

到目前为止，我们使用过滤器都是把来自模型的值在显示到视图之前进行转换。其实我们也可以定义一个过滤器，在把来自视图的值（input 元素）在写回模型之前进行转换：

```
Vue.filter('check-email', {
  // 这里 read 可选，只是为了演示
  read: function (val) {
    return val
  },
  // write 函数会在数据写入到模型之前被调用
  write: function (val, oldVal) {
    return isEmail(val) ? val : oldVal
  }
})
```

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`。

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

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

在上面这个例子中，显然用内联表达式也可以达成相同的效果。但是面对更复杂的需求时，常常需要不止一个语句，这种情况下你就得把逻辑放到一个计算属性中或是一个自定义过滤器中。

内建的 `filterBy` 和 `orderBy` 过滤器都是根据当前 `Vue` 实例的状态对传入的数组进行处理。

很好！现在，是时候了解 `Vue` 的核心概念：[组件系统](#)了。

11

组件系统

使用组件

在 Vue.js 中，我们可以用 Vue 扩展出来的 ViewModel 子类当做可复用的组件。这在概念上与 **Web Components** 非常相似，不同之处在于 Vue 的组件无需任何 polyfill。要创建一个组件，只需调用 `Vue.extend()` 来生成一个 Vue 的子类构造函数：

```
// 扩展 Vue 得到一个可复用的构造函数
var MyComponent = Vue.extend({
  template: '<p>A custom component!</p>'
})
```

Vue 的构造函数可接收的大部分选项都能在 `Vue.extend()` 中使用，不过也有两个特例：`data` 和 `el`。由于每个 Vue 的实例都应该有自己的 `$data` 和 `$el`，我们显然不希望传递给 `Vue.extend()` 的值被所有通过这个构造函数创建的实例所共享。因此如果要定义组件初始化默认数据和元素的方式，应该传入一个函数：

```
var ComponentWithDefaultData = Vue.extend({
  data: function () {
    return {
      title: 'Hello!'
    }
  }
})
```

接下来，就可以用 `Vue.component()` 来注册这个构造函数了：

```
// 把构造函数注册到 my-component 这个 id
Vue.component('my-component', MyComponent)
```

为了更简单，也可以直接传入 option 对象来代替构造函数。如果接收到的是一个对象，`Vue.component()` 会为你隐式调用 `Vue.extend()`：

```
// 注意：该方法返回全局 Vue 对象，  
// 而非注册的构造函数  
Vue.component('my-component', {  
  template: '<p>A custom component!</p>'  
})
```

之后就能在父级实例的模板中使用注册过的组件了 (务必在初始化根实例之前注册组件)：

```
<!-- 父级模板 -->  
<my-component></my-component>
```

渲染结果：

```
<p>A custom component!</p>
```

你没有必要，也不应该全局注册所有组件。你可以限制一个组件仅对另一个组件及其后代可用，只要在另一个组件的 `components` 选项传入这个组件即可 (这种封装形式同样适用于其他资源，例如指令和过滤器)：

```
var Parent = Vue.extend({  
  components: {  
    child: {  
      // child 只能被  
      // Parent 及其后代组件使用  
    }  
  }  
})
```

理解 `Vue.extend()` 和 `Vue.component()` 的区别非常重要。由于 `Vue` 本身是一个构造函数，`Vue.extend()` 是一个**类继承方法**。它用来创建一个 `Vue` 的子类并返回其构造函数。而另一方面，`Vue.component()` 是一个类似 `Vue.directive()` 和 `Vue.filter()` 的资源注册方法。它作用是建立指定的构造函数与 ID 字符串间的

关系，从而让 Vue.js 能在模板中使用它。直接向 `Vue.component()` 传递 `options` 时，它会在内部调用 `Vue.extend()`。

Vue.js 支持两种不同风格的调用组件的 API：命令式的基于构造函数的 API，以及基于模板的声明式的 Web Components 风格 API。如果你感到困惑，想一下通过 `new Image()` 和通过 `` 标签这两种创建图片元素的方式。它们都在各自的适用场景下发挥着作用，为了尽可能灵活，Vue.js 同时提供这两种方式。

`table` 元素对能出现在其内部的元素类型有限制，因此自定义元素会被提到外部而且无法正常渲染。在那种情况下你可以使用指令式组件语法：`<tr v-component="my-component"></tr>`。

数据流

通过 prop 传递数据

默认情况下，组件有**独立作用域**。这意味着你无法在子组件的模板中引用父级的数据。为了传递数据到拥有独立作用域的子组件中，我们需要用到 `prop`。

一个“prop”是指组件的数据对象上的一个预期会从父级组件取得的字段。一个子组件需要通过 `prop` 选项显式声明它希望获得的 prop：

```
Vue.component('child', {
  // 声明 prop
  props: ['msg'],
  // prop 可以在模板内部被使用，
  // 也可以类似 this.msg 这样来赋值
  template: '<span>{{msg}}</span>'
})
```

然后，我们可以像这样向这个组件传递数据：

```
<child msg="hello!"></child>
```

结果：

- hello!

驼峰命名 vs. 连字符命名

HTML 特性是大小写不敏感的。当驼峰式的 prop 名在 HTML 中作为特性名出现时，你需要用对应的连字符

(短横) 分隔形式代替：

```
Vue.component('child', {
  props: ['myMessage'],
  template: '<span>{{myMessage}}</span>'
})
```

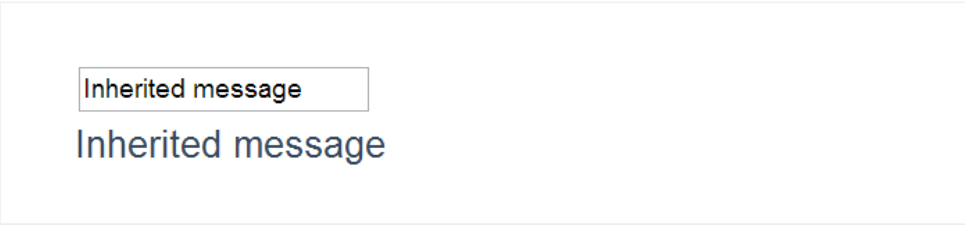
```
<!-- 重要：使用连字符分隔的名称！ -->
<child my-message="hello!"></child>
```

动态 prop

我们同样能够从父级向下传递动态数据。例如：

```
<div>
  <input v-model="parentMsg">
  <br>
  <child msg="{{parentMsg}}"></child>
</div>
```

结果：



暴露 `$data` 作为 prop 也是可行的。传入的值必须是一个对象，它会被用来替换组件默认的 `$data` 对象。

传递回调作为 prop

同样可以向下传递一个方法或语句作为子组件的一个回调方法。借此可以进行声明式的、解耦的父子通信：

```

Vue.component('parent', {
  // ...
  methods: {
    onChildLoaded: function (msg) {
      console.log(msg)
    }
  }
})
Vue.component('child', {
  // ...
  props: ['onLoad'],
  ready: function () {
    this.onLoad('message from child!')
  }
})

```

```

<!-- 父级模板 -->
<child on-load="{{onChildLoaded}}"></child>

```

prop 绑定类型

默认情况下，所有 prop 都会在子级和父级的属性之间建立一个**单向向下传递**的绑定关系：当父级的属性更新时，它将向下同步至子级，反之则不会。这种默认设定是为了防止子级组件意外篡改父级的状态，那将导致难以推导应用的数据流。不过也可以显式指定一个双向或者一次性的绑定：

对比这些语法：

```

<!-- 默认情况下，单向绑定 -->
<child msg="{{parentMsg}}"></child>
<!-- 显式双向绑定 -->
<child msg="@parentMsg"></child> <!-- 显示一次性绑定 -->
<child msg="{{* parentMsg}}"></child>

```

双向绑定会反向同步子级的 `msg` 属性到父级的 `parentMsg` 属性。一次性绑定在完成之后不会在父级和子级之间同步未来发生的变化。

注意如果传递的 `prop` 值是对象或数组，将会是引用传递。在子级改动对象或数组将会影响到父级的状态，这种情况会无视你使用的绑定的类型。

prop 规则

组件可以对接收到的 `prop` 声明一定的规则限制。在开发给他人使用的组件时这会很有用，因为对 `prop` 的有效性检验可以看做是组件 API 的一部分，并且能保证用户正确地使用了组件。与直接把 `prop` 定义成字符串不同，你需要使用包含验证规则的对象：

```
Vue.component('example', {
  props: {
    // 基本类型检查 (null 表示接受所有类型)
    onSomeEvent: Function,
    // 必需性检查
    requiredProp: {
      type: String,
      required: true
    },
    // 指定默认值
    propWithDefault: {
      type: Number,
      default: 100
    },
    // 对象或数组类型的默认值
    // 应该由工厂函数返回
    propWithObjectDefault: {
      type: Object,
      default: function () {
        return { msg: 'hello' }
      }
    }
  }
})
```

```
    }  
  },  
  // 双向 prop。  
  // 如果绑定类型不匹配将抛出警告。  
  twoWayProp: {  
    twoWay: true  
  },  
  // 自定义验证函数  
  greaterThanTen: {  
    validator: function (value) {  
      return value > 10  
    }  
  }  
}  
})
```

其中 `type` 可以是以下任一原生构造函数：

- `String`
- `Number`
- `Boolean`
- `Function`
- `Object`
- `Array`

另外，`type` 还可以是自定义构造函数，断言将会是一个 `instanceof` 检查。

如果 `prop` 检验不通过，Vue 会拒绝这次针对子组件的赋值，并且在使用开发版本时会抛出一个警告。

继承父级作用域

如果有需要，你也可以使用 `inherit: true` 选项来让子组件通过原型链继承父级的全部属性：

```
var parent = new Vue({
  data: {
    a: 1
  }
})
// $addChild() 是一个实例方法，
// 它允许你用代码创建子实例。
var child = parent.$addChild({
  inherit: true,
  data: {
    b: 2
  }
})
console.log(child.a) // -> 1
console.log(child.b) // -> 2
parent.a = 3
console.log(child.a) // -> 3
```

这里有一点需要注意：由于 Vue 实例上的数据属性都是 getter/setter，设置 `child.a = 2` 会直接改变 `parent.a` 的值，而非在子级创建一个新属性遮蔽父级中的属性：

```
child.a = 4
console.log(parent.a) // -> 4
console.log(child.hasOwnProperty('a')) // ->
false
```

作用域注意事项

当组件被用在父模板中时，例如：

```
<!-- 父模板 -->
<my-component v-show="active" v-
on="click:onClick"></my-component>
```

这里的命令 (`v-show` 和 `v-on`) 会在父作用域编译，所以 `active` 和 `onClick` 的取值取决于父级。任何子模板中的命令和插值都会`在子作用域中编译`。这样使得上下级组件间更好地分离。

阅读 **组件作用域** 了解更多细节。

组件生命周期

每一个组件，或者说 Vue 的实例，都有着自己的生命周期：它会被创建、编译、插入、移除，最终销毁。在这每一个时间点，实例都会触发相应的事件，而在创建实例或者定义组件时，我们可以传入生命周期钩子函数来响应这些事件。例如：

```
var MyComponent = Vue.extend({
  created: function () {
    console.log('An instance of MyComponent has
    been created!')
  }
})
```

查阅 API 文档中可用的 **生命周期钩子函数完整列表**。

动态组件

你可以使用内置的 `<component>` 元素在组件间动态切换来实现“页面切换”：

```
new Vue({
  el: 'body',
  data: {
    currentView: 'home'
  },
  components: {
    home: { /* ... / },
    posts: { / ... / },
    archive: { / ... */ }
  }
})
```

```
<component is="{{currentView}}">
  &lt;!-- 内容随 vm.currentview 一同改变！ -->
&gt;
&lt;/component&gt;</pre>
</div>
```

如果希望被切换出去的组件保持存活，从而保留它的当前状态或者避免反复重新渲染，你可以加上 `keep-alive`` 特性参数：

```
<component is="{{currentView}}" keep-alive>
  <!-- 不活跃的的组件会被缓存！ -->
</component>
```


过渡控制

有两个额外的特性参数能够支持对需要渲染或过渡的组件进行高级控制。

`wait-for` 等待事件

等待即将进入的组件触发该事件后再插入 DOM。这就允许你等待数据异步加载完成后再触发过渡，避免显示空白内容。

这一特性可以用于静态和动态组件。注意：对于动态组件，所有有待渲染的组件都必须通过 `$emit` 触发指定事件，否则他们永远不会被插入。

示例：

```
<!-- 静态组件 -->
<my-component wait-for="data-loaded"></my-
component>
<!-- 动态组件 -->
<component is="{{view}}" wait-for="data-
loaded"></component>
```

```
````// 组件定义
{
 // 获取数据并在编译完成钩子函数中异步触发事
 件。
 // 这里jQuery只是用作演示。
 compiled: function () {
 var self = this
 $.ajax({
 // ...
 success: function (data) {
 self.$data = data
 self.$emit('data-loaded')
 }
 })
 }
}
```

transition-mode **过渡模式**

`transition-mode` 特性参数允许指定两个动态组件之间的过渡应如何进行。

默认情况下，进入组件和退出组件的过渡是同时进行的。这个特性参数允许设置成另外两种模式：

- `in-out`：先进后出；先执行新组件过渡，当前组件在新组件过渡结束后执行过渡并退出。
- `out-in`：先出后进；当前组件首先执行过渡并退出，新组件在当前组件过渡结束后执行过渡并进入。

**示例：**

## 列表与组件

---

对于一个对象数组，你可以把组件和 `v-repeat` 组合使用。这种场景下，对于数组中的每个对象，都以该对象为 `$data` 创建一个子组件，以指定组件作为构造函数。

```
<ul id="list-example">
 <user-profile v-
repeat="users"></user-
profile>

```

```
new Vue({
 el: '#list-example',
 data: {
 users: [
 {
 name: 'Chuck Norris',
 email: 'chuck@norris.com'
 },
 {
 name: 'Bruce Lee',
 email: 'bruce@lee.com'
 }
]
 },
 components: {
```

```
 'user-profile': {
 template: '
• {{name}} {{email}}
,
 }
 }
})
```

**结果：**

- Chuck Norris - chuck@norris.com
- Bruce Lee - bruce@lee.com

## 在组件循环中使用标识符

在组件中标识符语法同样适用，并且被循环的数据会被设置成组件的一个属性，以标识符作为键名：

>注意一旦组件跟 `v-repeat` 一同使用，同样的作用域规则也会被应用到该组件容器上的其他命令。结果就是，你在父级模板中将获取不到 `$index`；只能在组件自身的模板中获取。

>

>或者，你也可以通过循环 `<template>` 来建立一个媒介作用域，但是大多数情况下在组件内部使用 `$index` 是更好的实践。

## 子组件引用

---

某些情况下需要通过 JavaScript 访问嵌套的子组件。要实现这种操作，需要使用 `v-ref` 为子组件分配一个 ID。例如：

```
var parent = new Vue({ el: '#parent' })
// 访问子组件
var child = parent.$.profile
```

当 `v-ref` 与 `v-repeat` 一同使用时，会获得一个与数据数组对应的子组件数组。



## 事件系统

---

虽然你可以直接访问一个 Vue 实例的子级与父级，但是通过内建的事件系统进行跨组件通讯更为便捷。这还能使你的代码进一步解耦，变得更易于维护。一旦建立了上下级关系，就能使用组件的 **事件实例方法** 来分发和触发事件。

```
var parent = new Vue({
 template: '
 ',
 created: function () {
 this.$on('child-created', function (child) {
 console.log('new child created: ')
 console.log(child)
 })
 },
 components: {
 child: {
 created: function () {
 this.$dispatch('child-created', this)
 }
 }
 }
}).$mount()
```



## 私有资源

---

有时一个组件需要使用类似命令、过滤器和子组件这样的资源，但是又希望把这些资源封装起来以便自己在别处复用。这一点可以用私有资源实例化选项来实现。私有资源只能被拥有该资源的组件及其继承组件和子组件的实例访问。

```
// 全部5种类型的资源
var MyComponent = Vue.extend({
 directives: {
 // “id : 定义”键值对，与处理全局方法的方式
 相同
 'private-directive': function () {
 // ...
 }
 },
 filters: {
 // ...
 },
 components: {
 // ...
 },
 partials: {
 // ...
 },
 effects: {
 // ...
 }
})
```

```
}
})
```

>你可以通过设置 `Vue.config.strict = true` 阻止子组件访问父组件的私有资源。

又或者，可以用与全局资源注册方法类似的链式 API 为现有组件构造方法添加私有资源：

```
MyComponent
 .directive('...', {})
 .filter('...', function () {})
 .component('...', {})
 // ...
```

## 资源命名约定

有些资源，诸如组件和指令，会以 HTML 特性或自定义 HTML 标签的方式出现在模板中。因为 HTML 特性名和标签名都是**大小写不敏感**的，我们经常需要用连

字符（短横）连接命名取代驼峰命名。**从 0.12.9 开始**，我们支持将资源进行驼峰命名，同时在模板里用连字符命名法使用它们。

## 示例

```
// in a component definition
components: {
 // 用驼峰命名注册组件
 myComponent: { /*... */ }
}
```

这和 ES6 **对象字面量简写** 完美搭配：

```
import compA from './components/a';
import compB from './components/b';
export default {
 components: {
 // 在模板中以 和 的形式调用
 compA,
 compB
 }
}
```



## 内容插入

---

在创建可复用组件时，常常需要访问及复用宿主元素的原始内容，而它们并非组件本身的一部分（类似 Angular 的“transclusion”概念）。Vue.js 实现了一套内容插入机制，它和目前的 Web Components 规范草案兼容，使用特殊的 `<content>` 元素作为原始内容的插入点。

>关键提示：`transclude` 的内容会在父级作用域中编译，而非子级作用域。

### 单插入点

只有一个不带特性的 `<content>` 标签时，整个原始内容都会被插入到它在 DOM 中的位置并把它替换掉。原来在 `<content>` 标签内部的所有内容会被视为 **后备内容**。后备内容只有在宿主元素为空且没有要插入的内容时才会被显示。例如：

my-component 的模板：



This is my component!





This will only be displayed if no content is inserted

使用该组件的父标签：

```
<my-component>
```

```
 <p>This is some original
 content</p>
```

```
 <p>This is some more original
 content</p>
```

```
</my-component>
```

渲染结果如下：

```
<my-component>
```

```
 <h1>This is my component!</h1>
```

```
 <p>This is some original
 content</p>
```

```
 <p>This is some more original
 content</p>
```

```
</my-component>
```

## 多插入点

`<content>` 元素有一个特殊特性 `select`，需要赋值为一个 CSS 选择器。可以使用多个包含不同 `select` 特性的 `<content>` 插入点，它们会被原始内容中与选择器匹配的部分所替代。

>从 0.11.6 起，`<content>` 选择器只能匹配宿主节点的顶级子节点。从而表现与 Shadow DOM 规范一致，并且可以避免意外地选中嵌套的 `transclude` 内容中不需要的节点。

举例来说，假设有一个带有如下模板的 `multi-insertion` 组件：

...

父标签：

```
<multi-insertion>
```

```
 <p>One</p>
```

```
 <p>Two</p>
```

```
<p>Three</p>
</multi-insertion>
```

渲染结果如下：

```
<multi-insertion>
 <p>Three</p>
 <p>Two</p>
 <p>One</p>
</multi-insertion>
```

内容插入机制能很好地控制对原始内容的操作和显示，使组件极为灵活多变易于组合。

## 行内模板

---

在 0.11.6 中，为组件引入了一个特殊的特性参数：`inline-template`。当传递了这个参数时，组件会使用自己内部的内容作为模板而非 `transclude` 的内容。这会使模板编写更灵活。

```
<my-component inline-template>
```

```
 <p>这里的内容会作为组件本身的模板进行编译</p>
```

```
 <p>而不是作为父作用域的插入内容</p>
```

```
</my-component>
```

## 异步组件

---

异步组件只在 `Vue ^0.12.0` 版本中支持。

在大型项目中，我们可能需要把应用分割成小的组成部分，并且只在实际用到一个组件的时候加载它。为了让这种操作更容易，`Vue.js` 允许把组件定义成一个异步加载组件定义的工厂方法。`Vue.js` 只会在需要渲染该组件时才触发相应的工厂方法，并且会对加载结果进行缓存。例如：

```
```Vue.component('async-example', function
(resolve, reject) {
  setTimeout(function () {
    resolve({
      template: `
```

```
I am async!
```

```
,
    })
  }, 1000)
})``
```

工厂方法会收到一个 `resolve` 回调方法，应该在从服务器获得组件定义之后调用它。你也可以通过调用 `reject(reason)` 来提示加载失败。这里的 `setTimeout` 只是用作简单的演示；具体如何获取组件完全取决于你。有一种不错的手段是把异步组件和 Webpack 的**分块打包功能**结合使用：

```
Vue.component('async-webpack-example', function
(resolve) {
  // 这个特殊的 require 语法会通知 webpack
  // 自动把构建后的代码分割成
  // 会通过 ajax 请求自动加载的 bundle。
  require(['./my-async-component'], resolve)
})
```

下一节：[过渡效果](#)



12

过渡效果

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。

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 过渡效果

一个典型的 CSS 过渡效果定义如下：

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

你还需要定义 `.expand-transition`，`.expand-enter` 和 `.expand-leave` 三个 CSS 类：

```
.expand-transition {  
  transition: all .3s ease;  
  height: 30px;  
  padding: 10px;  
  background-color: #eee;  
  overflow: hidden;  
}  
.expand-enter, .expand-leave {  
  height: 0;  
  padding: 0 10px;  
  opacity: 0;  
}
```

同时，你也可以提供 JavaScript 钩子：

```
Vue.transition('expand', {  
  beforeEnter: function (el) {  
    el.textContent = 'beforeEnter'  
  },  
  enter: function (el) {  
    el.textContent = 'enter'  
  },  
  afterEnter: function (el) {  
    el.textContent = 'afterEnter'  
  },  
  enterCancelled: function (el) {  
    // handle cancellation  
  }  
})
```

```
    },  
    ,  
    beforeLeave: function (el) {  
      el.textContent = 'beforeLeave'  
    },  
    leave: function (el) {  
      el.textContent = 'leave'  
    },  
    afterLeave: function (el) {  
      el.textContent = 'afterLeave'  
    },  
    leaveCancelled: function (el) {  
      // handle cancellation  
    }  
  })  
})
```

结果



这里使用的 CSS 类名由 `v-transition` 指令的值所决定。以 `v-transition="fade"` 为例，CSS 类 `.fade-transition` 将会一直存在，而 `.fade-enter` 和 `.fade-leave` 将会在合适的时机自动被添加或移除。当 `v-transition` 指令没有提供值的时候，所使用的 CSS 类名将会是默认的 `.v-transition`，`.v-enter` 和 `.v-leave`。

当 `show` 属性变化时，Vue 会依据其当前的值来插入/移除 `<div>` 元素，并在合适的时机添加/移除对应的 CSS 类，具体如下：

- 当 `show` 变为 `false` 时，Vue 将会：

1. 调用 `beforeLeave` 钩子；

2. 在元素上应用 CSS 类 `.v-leave` 来触发过渡效果；
 3. 调用 `leave` 钩子；
 4. 等待过渡效果执行完毕；(监听 `transitionend` 事件)
 5. 从 DOM 中移除元素并且移除 CSS 类 `.v-leave`。
 6. 调用 `afterLeave` 钩子。
- 当 `show` 为 `true` 时，Vue 将会：
 1. 调用 `beforeEnter` 钩子；
 2. 在元素上应用 CSS 类 `.v-enter`；
 3. 将元素插入 DOM；
 4. 调用 `enter` 钩子；
 5. 应用 `.v-enter` 类, 然后强制 CSS 布局以保证 `.v-enter` 生效；最后移除 `.v-enter` 来触发元素过渡到原本的状态。
 6. 等待过渡效果执行完毕；
 7. 调用 `afterEnter` 钩子。

此外，如果一个正在执行进入的过渡效果的元素在过渡还未完成之前就被移除，则 `enterCancelled` 钩子将会被执行。这个钩子可以用于清理工作，比如移除在 `enter` 时创建的计时器。对于正在离开过渡中又被重新插入的元素同理。

上述所有的钩子函数执行时，其 `this` 都指向相应的 Vue 实例。如果一个元素本身是一个 Vue 实例的根节点，则此实例将被应用为 `this`；否则 `this` 指向该过渡指令所属的实例。

最后，`enter` 与 `leave` 钩子函数可以接受可选的第二个参数：一个回调函数。当你的函数签名中含有第二个参数时，即表示你期望使用此回调来显式地完成整个过渡过程，而不是依赖 Vue 去自动检测 CSS 过渡的 `transitionend` 事件。比如：

```
enter: function (el) {  
  // 无第二个参数  
  // 过渡效果的结束由 CSS 过渡结束事件来决定  
}
```

VS

```
enter: function (el, done) {  
  // 有第二个参数  
  // 过渡效果结束必须由手动调用 `done` 来决定  
}
```

当多个元素同时执行过渡效果时，Vue.js 会进行批量处理以保证只触发一次强制布局。

CSS 动画

CSS 动画通过与 CSS 过渡效果一样的方式进行调用，区别就是动画中 `.v-enter` 类并不会在节点插入 DOM 后马上移除，而是在 `animationend` 事件触发时移除。

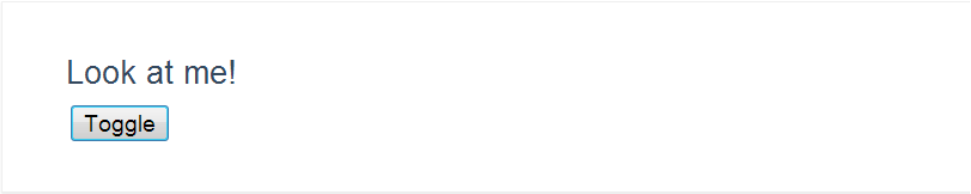
示例：（省略了兼容性前缀）

```
<span v-show="show" v-transition="bounce">Look  
at me!</span>
```

```
.bounce-enter {  
  animation: bounce-in .5s;  
}  
.bounce-leave {  
  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);  
  }  
}
```

结果



纯 JavaScript 过渡效果

你也可以只使用 JavaScript 钩子，不定义任何 CSS 过渡规则。当只使用 JavaScript 钩子时，`enter` 和 `leave` 钩子必须使用 `done` 回调，否则它们将会被同步调用，过渡将立即结束。下面的示例中我们使用 jQuery 来注册一个自定义的 JavaScript 过渡效果：

```
Vue.transition('fade', {
  enter: function (el, done) {
    // 此时元素已被插入 DOM
    // 动画完成时调用 done 回调
    $(el)
      .css('opacity', 0)
      .animate({ opacity: 1 }, 1000, done)
  },
  enterCancelled: function (el) {
    $(el).stop()
  },
  leave: function (el, done) {
    // 与 enter 钩子同理
    $(el).animate({ opacity: 0 }, 1000, done)
  },
  leaveCancelled: function (el) {
    $(el).stop()
  }
})
```

定义此过渡之后，你就可以通过给 `v-transition` 指定对应的 ID 来调用它：

```
<p v-transition="fade"></p>
```

如果一个只使用 JavaScript 过渡效果的元素恰巧也受到其它 CSS 过渡/动画规则的影响，这可能会对 Vue 的 CSS 过渡检测机制产生干扰。碰到这样的状况时，你可以通过给你的钩子对象添加 `css: false` 来禁止 CSS 检测。

渐进过渡效果

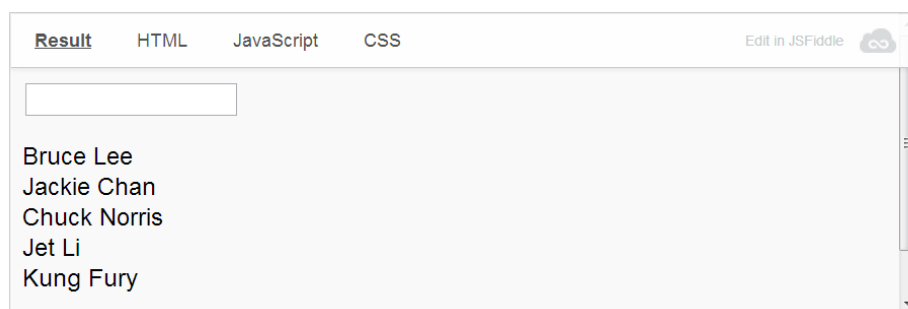
当同时使用 `v-transition` 和 `v-repeat` 时，我们可以为列表元素添加渐进的过渡效果，你只需要为你的过渡元素加上 `stagger`，`enter-stagger` 或者 `leave-stagger` 特性（以毫秒为单位）：

```
<div v-repeat="list" v-transition stagger="100">
</div>
```

或者你也可以提供 `stagger`，`enterStagger` 或 `eaveStagger` 钩子来进行更细粒度的控制：

```
Vue.transition('stagger', {
  stagger: function (index) {
    // 为每个过渡元素增加 50ms 的延迟，
    // 但是最大延迟为 300ms
    return Math.min(300, index * 50)
  }
})
```

示例：



下一节：[创建大型应用](#)。



13

创建大型应用

Vue.js 在设计思想上追求的是尽可能的灵活。它本身只是一个界面库，并不强制使用哪种架构。这对于快速原型开发很有用，但是对于经验欠缺的开发者，用 Vue.js 构建大型应用可能会是一个挑战。在这里我会针对在使用 Vue.js 时如何组织大型的项目提供一些略带个人偏好的建议。

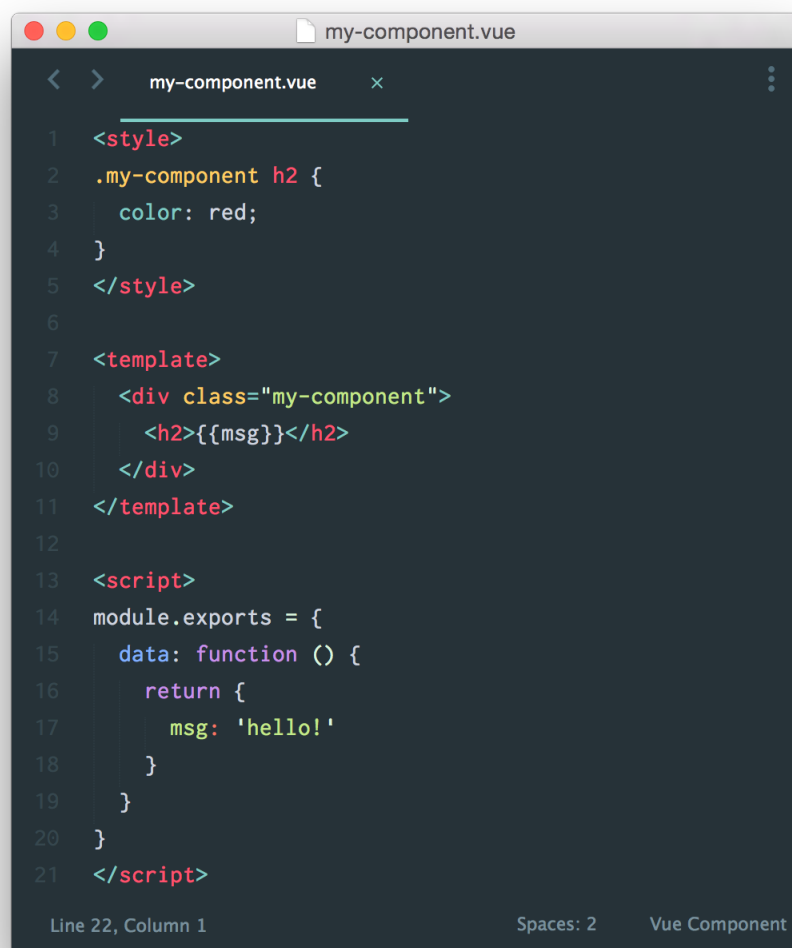
模块化

虽然独立构建的 `Vue.js` 可以被用作一个全局变量，但是它通常更适合配合一个模块化构建系统使用，这可以让你更好地组织代码。我的建议是用 CommonJS 模块格式编写源代码 (这是 Node.js 使用的格式，也是 `Vue.js` 源代码使用的格式)，并通过 **Webpack** 或 **Browserify** 把它们打包起来。

更重要的是，**Webpack** 和 **Browserify** 不仅仅是模块打包工具。两者都提供源码转换的 API，允许你将你的源码用其他的预处理程序进行转换。例如，你可以用 **babel-loader** 或 **babelify** 直接在你的模块中使用 ES6/7 的语法。

单文件组件

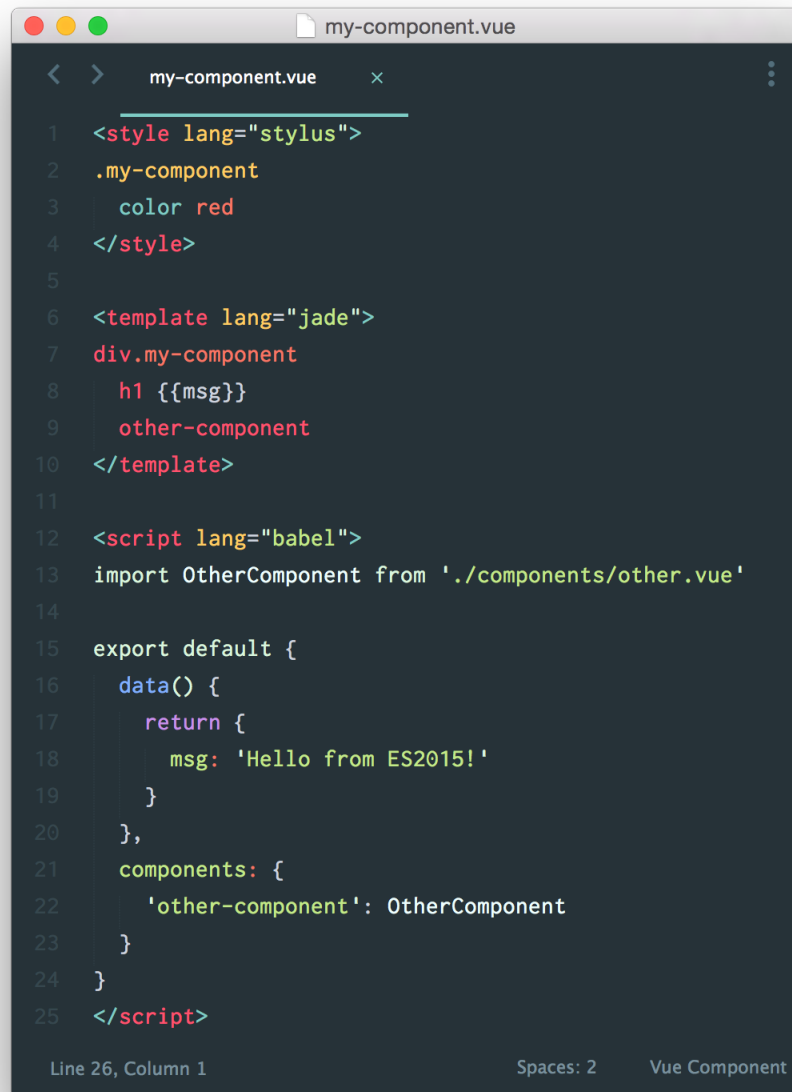
在一个典型的 Vue.js 项目里，我们会希望将我们的代码拆分成许多的组件，并且在一个组件里把它所包含的 CSS 样式、HTML 模板、JavaScript 逻辑封装在一起。就像上面提到的那样，借助 Webpack 或 Browserify，并配以相应的源码转换插件，我们就可以这样编写组件了：



```
1 <style>
2 .my-component h2 {
3   color: red;
4 }
5 </style>
6
7 <template>
8   <div class="my-component">
9     <h2>{{msg}}</h2>
10   </div>
11 </template>
12
13 <script>
14 module.exports = {
15   data: function () {
16     return {
17       msg: 'hello!'
18     }
19   }
20 }
21 </script>
```

Line 22, Column 1 Spaces: 2 Vue Component

如果你喜欢预处理器，你甚至可以这样写：



```
1 <style lang="stylus">
2   .my-component
3     color red
4 </style>
5
6 <template lang="jade">
7   div.my-component
8     h1 {{msg}}
9     other-component
10 </template>
11
12 <script lang="babel">
13   import OtherComponent from './components/other.vue'
14
15   export default {
16     data() {
17       return {
18         msg: 'Hello from ES2015!'
19       }
20     },
21     components: {
22       'other-component': OtherComponent
23     }
24   }
25 </script>
```

Line 26, Column 1 Spaces: 2 Vue Component

你可以用 Webpack + vue-loader 或 Browserify + vueify 来编译这些单文件的 Vue 组件。如果你同时使用预处理器，则推荐用 Webpack 来进行构建，因为 Webpack 的插件 API 提供了更好的文件依赖追踪和缓存支持。

在 GitHub 上可以找到上面所描述的工作流的代码示例：

- Webpack + vue-loader

- Browserify + vueify

路由

官方路由模块 `vue-router` 正在活跃开发中，即将发布。

你可以手动监听 `hashchange` 事件并利用一个动态的 `<component>` 来实现一些基本的路由逻辑。

示例：

```
<div id="app">
  <component is="{{currentView}}"></component>
</div>```

```Vue.component('home', { /* ... */ })
Vue.component('page1', { /* ... */ })
var app = new Vue({
 el: '#app',
 data: {
 currentView: 'home'
 }
})
// Switching pages in your route handler:
app.currentView = 'page1'
```

利用这种机制我们可以很容易地接入独立的路由库，比如 **Page.js** 或是 **Director**。



## 服务器通信

---

所有的 Vue 实例都可以直接通过 `JSON.stringify()` 序列化得到它们原始的 `$data`，并不需要做任何额外的工作。社区已经贡献了 **vue-resource** 插件，提供了与 RESTful API 协作的功能。你也可以使用任何你喜欢的 Ajax 组件，比如 `$.ajax` 或是 **SuperAgent**。Vue.js 也可以和诸如 Firebase 和 Parse 这样的 BaaS 服务完美配合。

## 单元测试

---

任何兼容 CommonJS 构建系统的测试工具都可以。建议使用 **Karma** 配合其 **CommonJS 预处理插件** 对你的代码进行模块化测试。

最佳实践是暴露出模块内的原始选项/函数。参考如下示例：

```
// my-component.js
module.exports = {
 template: '{{msg}}',
 data: function () {
 return {
 msg: 'hello!'
 }
 }
 created: function () {
 console.log('my-component created!')
 }
}
```

你可以在你的入口模块中如下使用这个文件：

```
// main.js
var Vue = require('vue')
var app = new Vue({
 el: '#app',
 data: { /* ... */ },
 components: {
 'my-component': require('./my-component')
 }
})
```

然后你可以如下测试该模块：

```
// Some Jasmine 2.0 tests
describe('my-component', function () {
 // require source module
 var myComponent = require('../src/my-component')
 it('should have a created hook', function ()
 {
 expect(typeof
myComponent.created).toBe('function')
 })
 it('should set correct default data',
function () {
 expect(typeof
myComponent.data).toBe('function')
 var defaultData = myComponent.data()
 expect(defaultData.message).toBe('hello!')
 })
})
```

因为 Vue.js 的指令异步响应数据的更新，当你需要在数据更新后断言 DOM 的状态时，你需要在一个 `Vue.nextTick` 回调里做这件事。

## 发布至生产环境

---

为了缩小体积，最小化后的独立版 Vue.js 已去除所有的警告信息，但当你用像 Browserify、Webpack 这样的工具构建 Vue.js 应用时，并没有一个明显的办法来去除这些警告。

从 0.12.8 开始，可以采用如下的方式配置你的构建工具来优化最后生成的代码体积：

### Webpack

使用 Webpack 的 **DefinePlugin** 来定义生产环境参数，这样警告相关的代码会变得永远不会被执行，从而在 UglifyJS 压缩的时候会被自动丢掉。比如：

```
var webpack = require('webpack')
module.exports = {
 // ...
 plugins: [
 // ...
 new webpack.DefinePlugin({
 'process.env': {
 NODE_ENV: '"production"'
 }
 }),
 new webpack.optimize.UglifyJsPlugin({
 compress: {
 warnings: false
 }
 })
]
}
```

```
]
}
```

## Browserify

只需要在打包命令中把 `NODE_ENV` 设置成 `"production"` 即可。Vue 会自动应用 **envify** 转换并跳过警告处理。比如：

```
NODE_ENV=production browserify -e main.js |
uglifyjs -c -m > build.js
```

## 应用示例

---

**Vue.js Hackernews Clone** 是一个应用的例子，它用 `Webpack + vue-loader` 代码组织、`Director.js` 做路由、HackerNews 官方的 `Firebase API` 为后端。这不算什么特别大的应用，但它结合并展示了本页面讨论到的各方面概念。

下一节：[扩展 Vue](#)

14

扩展 Vue

## Mixins

---

Mixin (混入) 是一种可以在多个 Vue 组件之间灵活复用特性的机制。你可以像写一个普通 Vue 组件的选项对象一样编写一个 mixin：

```
// mixin.js
module.exports = {
 created: function () {
 this.hello()
 },
 methods: {
 hello: function () {
 console.log('hello from mixin!')
 }
 }
}
```

```
// test.js
var myMixin = require('./mixin')
var Component = Vue.extend({
 mixins: [myMixin]
})
var component = new Component() // -> "hello
from mixin!"
```

更多细节请参见 **API**。



## 使用插件进行扩展

---

通常插件会为 Vue 添加一个全局的功能。

### 撰写插件

你可以撰写以下几种典型类型的插件：

1. 添加一个或几个全局方法。比如 `vue-element`
2. 添加一个或几个全局资源：指令、过滤器、动画效果等。比如 `vue-touch`
3. 通过绑定到 `Vue.prototype` 的方式添加一些 Vue 实例方法。这里有个约定，就是 Vue 的实例方法应该带有 `$` 前缀，这样就不会和用户的数据和方法产生冲突了。

```
exports.install = function (Vue, options) {
 Vue.myGlobalMethod = ... // 1
 Vue.directive('my-directive', {}) // 2
 Vue.prototype.$myMethod = ... // 3
}
```

### 使用插件

假设我们使用的构建系统是 `CommonJS`，则需要作如下调用：

```
var vueTouch = require('vue-touch')
// use the plugin globally
Vue.use(vueTouch)
```

你也可以向插件里传递额外的选项：

```
Vue.use(require('my-plugin'), {
 /* pass in additional options */
})
```

## 现有的插件 & 工具

---

- `vue-resource` : 一个插件，为用 `XMLHttpRequest` 或 `JSONP` 生成网络请求、响应提供服务。
- `vue-validator` : 一个表单验证的插件。
- `vue-devtools` : 一个用来调试 Vue.js 应用程序的 Chrome 浏览器开发者工具扩展。
- `vue-touch` : : 添加基于 Hammer.js 的触摸手势的指令。
- `vue-element` : 用 Vue.js 注册 `Custom Elements` 。
- 用户贡献的工具列表

下一节: [最佳实践与技巧](#)。

15

## 细节与最佳实践

## 数据初始化

---

明确定义的数据模型更加适合 Vue 的数据观察模式。建议在定义组件时，在 `data` 选项中初始化所有需要进行动态观察的属性。例如，给定下面的模版：

```
<div id="demo">
 <p v-class="green: validation.valid">{{message}}
</p>
 <input v-model="message">
</div>
```

建议像这样初始化你的数据，而不是什么都不定义：

```
new Vue({
 el: '#demo',
 data: {
 message: '',
 validation: {
 valid: false
 }
 }
})
```

为什么要这样做呢？因为 Vue 是通过递归遍历初始数据中的所有属性，并用 `Object.defineProperty` 把它们转化为 `getter` 和 `setter` 来实现数据观察的。如果一个属性在实例创建时不存在于初始数据中，那么 Vue 就没有办法观察这个属性了。

当然，你也不需要为每一个可能存在的嵌套属性都进行初始定义。在初始化的时候可以将一个属性置为空对象，然后在后面的操作中设置为一个新的拥有嵌套结构的对象。只要这个新对象包含了应有的属性，Vue 依然能对这个新对象进行递归遍历，从而观察其内部属性。

## 添加和删除属性

---

正如前面所说的，Vue 会使用 `Object.defineProperty` 通过转化属性值来观察数据。不过，在 ECMAScript 5 中，当一个新的属性被添加到对象或者从对象中删除的时候，并没有办法可以检测到这两种情况。为了解决这个问题，Vue 会为被观察的对象添加三个扩展方法：

- `obj.$add(key, value)`
- `obj.$set(key, value)`
- `obj.$delete(key)`

通过调用这些方法给观察对象中添加或者删除属性，就能够触发所对应的 DOM 更新。`$add` 和 `$set` 的区别是，假如当前对象已经含有所使用的 `key`，`$add` 会直接返回。所以当使用 `$obj.$add(key)` 的时候不会将已经存在的值覆盖为 `undefined`。

另外需要注意的一点是，当你通过数组索引赋值来改动数组时（比如 `arr[1] = value`），Vue 是无法侦测到这类操作的。类似地，你可以使用扩展方法来确保 Vue.js 收到了通知。被观察的数组有两个扩展方法：

- `arr.$set(index, value)`
- `arr.$remove(index | value)`

Vue 组件实例也有相应的实例方法：

- `vm.$get(path)`
  - `vm.$set(path, value)`
-

- `vm.$add(key, value)`
- `vm.$delete(key, value)`

注意 `vm.$get` 和 `vm.set` 都接受路径。

尽管存在这些方法，但我强烈建议你只在必要的时候才动态添加可观察属性。为了理解你的组件状态，将 `data` 选项看做一个 `schema` 很有帮助。假如你清晰地列出一个组件中所有可能存在的属性，那么当你隔了几个月再来维护这个组件的时候，就可以更容易地理解这个组件可能包含怎样的状态。

## 理解异步更新

---

默认情况下，Vue 的 DOM 更新是**异步执行的**。理解这一点非常重要。当侦测到数据变化时，Vue 会打开一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。假如一个 watcher 在一个事件循环中被触发了多次，它只会被推送到队列中一次。然后，在进入下一次的事件循环时，Vue 会清空队列并进行必要的 DOM 更新。在内部，Vue 会使用 `MutationObserver` 来实现队列的异步处理，如果不支持则会回退到 `setTimeout(fn, 0)`。

举例来说，当你设置 `vm.someData = 'new value'`，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新。如果你想要根据更新的 DOM 状态去做某些事情，就必须留意这个细节。尽管 Vue.js 鼓励开发者用“数据驱动”的方式想问题，避免直接操作 DOM，但有时候你可能就是想要使用某个熟悉的 jQuery 插件。这种情况下怎么办呢？你可以在数据改变后，立刻调用

`Vue.nextTick(callback)`，并把你要做的事情放到回调函数里面。当 `Vue.nextTick` 的回调函数执行时，DOM 将会已经是更新后的状态了。

示例：

```
<div id="example">{{msg}}</div>
```

```
var vm = new Vue({
 el: '#example',
 data: {
 msg: '123'
 }
})
```



```
vm.msg = 'new message' // change data
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
 vm.$el.textContent === 'new message' // true
})
```

除此之外，也有一个实例方法 `vm.$nextTick()`。这个方法与全局的 `Vue.nextTick` 功能一样，但更方便在组件内部使用，因为它不需要全局的 `Vue` 变量，另外它的回调函数的 `this` 上下文会自动绑定到调用它的 `Vue` 实例：

```
Vue.component('example', {
 template: '{{msg}}',
 data: function () {
 return {
 msg: 'not updated'
 }
 },
 methods: {
 updateMessage: function () {
 this.msg = 'updated'
 console.log(this.$el.textContent) // =>
 'not updated'
 this.$nextTick(function () {
 console.log(this.$el.textContent) // =>
 'updated'
 })
 }
 }
})
```

## 组件作用域

---

每一个 Vue.js 组件都是一个拥有自己的独立作用域的 Vue 实例。在使用组件的时候，理解组件作用域机制非常重要。其规则概括来说就是：

在父模板中出现的，将在父模板作用域内编译；在子模板中出现的，将在子模板的作用域内编译。

一个常见的错误是，在父模版中尝试将一个指令绑定到子作用域里的属性或者方法上：

```
<div id="demo">
 <!-- 不起作用，因为作用域不对！ -->
 <child-component v-on="click: childMethod">
</child-component>
</div>
```

如果需要在子组件的根节点上绑定指令，应当将指令写在子组件的模板内：

```
Vue.component('child-component', {
 // 这次作用域对了
 template: '<div v-on="click: childMethod">Child</div>',
 methods: {
 childMethod: function () {
 console.log('child method invoked!')
 }
 }
})
```

注意，当组件和 `v-repeat` 一同使用时，`$index` 作为子作用域属性也会受到此规则的影响。

另外，父模板里组件节点内部的 HTML 内容被看做是“transclusion content”（插入内容）。除非子模版包含至少一个 `<content>` 出口，不然这些插入内容不会被渲染。需要留意的是，插入内容也是**在父作用域中编译**的：

```
<div>
 <child-component>
 <!-- 在父作用域里编译 -->
 <p>{{msg}}</p>
 </child-component>
</div>
```

你可以使用 `inline-template` 属性去明确内容在子模版的作用域中被编译：

```
<div>
 <child-component inline-template>
 <!-- 在子作用域里编译 -->
 <p>{{msg}}</p>
 </child-component>
</div>
```

更多关于内容插入的细节，请看组件一章的**内容插入**小节。

## 在多个实例之间通讯

---

一种常见的在 Vue 中进行父子通讯的方法是，通过 `props` 传递一个父方法作为一个回调到子组件中。这样使用时的回调传递可以被定义在父模版中，从而保持了组件之间 Javascript 实现细节上的解耦：

```
<div id="demo">
 <p>Child says: {{msg}}</p>
 <child-component send-message="{{onChildMsg}}">
</child-component>
</div>
```

```
new Vue({
 el: '#demo',
 data: {
 msg: ''
 },
 methods: {
 onChildMsg: function(msg) {
 this.msg = msg
 return 'Got it!'
 }
 },
 components: {
 'child-component': {
 props: [
 // you can use prop assertions to
 ensure the
 // callback prop is indeed a function.
 {
 name: 'send-message',
 type: Function,
 required: true
 }
]
 }
 }
})
```

```

],
 // props with hyphens are auto-camelized
 template:
 '<button v-on="click:onClick">Say Yeah!
</button>' +
 '<p>Parent responds: {{response}}</p>',
 // component `data` option must be a
 function
 data: function () {
 return {
 response: ''
 }
 },
 methods: {
 onClick: function () {
 this.response =
this.sendMessage('Yeah!')
 }
 }
 }
 }
})

```

## Result :

Child says:

Say Yeah!

Parent responds:

当你需要跨越多层嵌套的组件进行通讯时，你可以使用**事件系统**。另外，在构建大型应用时，用 Vue 搭配类似 **Flux** 的架构也是完全可行的。

## 片段实例

---

自 0.12.2 起，`replace` 参数默认为 `true`。这意味着：

组件的模板长什么样，渲染出来的 DOM 就是什么样。

父模板中调用组件的元素将会被组件本身的模板取代。因此，如果组件的模板包含多个顶级元素：

```
Vue.component('example', {
 template:
 '<div>A</div>' +
 '<div>B</div>'
})
```

或者模板只包含文本：

```
Vue.component('example', {
 template: 'Hello world'
})
```

在这两个情况下，实例将变成一个**片段实例** (fragment instance)，也即没有根元素的实例。它的 `$el` 指向一个锚节点（普通模式下是空的文本节点，debug 模式下是注释节点）。更重要的是，父模板组件元素上的指令、过渡效果和属性绑定（`props` 除外）将无效，因为生成的实例并没有根元素供它们绑定：

```
<!-- 指令不生效，因为没有根元素用来绑定 -->
<example v-show="ok" v-transition="fade">
</example>
`
`<!-- props 还是能够正常生效 -->
<example prop="{someData}"></example>
```

虽然片段实例也有其使用场景，但是大部分情况下，给组件模板一个根元素是推荐的做法。这样父模板组件元素上的指令和属性能正常运转，并且性能也会更好一点。

## 修改默认选项

---

通过修改全局的 `Vue.options` 对象，可以修改实例选项的默认值。例如，你可以设置 `Vue.options.replace = false`，使所有 Vue 实例都按照 `replace: false` 的规则被编译。请谨慎使用这个功能 - 最好是只在一个项目刚开始的时候使用它，因为它会影响所有 Vue 实例的行为。

下一节：[常见问题](#)。



16

常见问题

- **为什么 Vue.js 不支持 IE8?**

Vue.js 借助 ECMAScript 5 的 `Object.defineProperty` 才得以不靠脏检查实现原生对象即模型的 API。然而这一新特性在 IE8 里只在 DOM 元素上起作用，对原生 JavaScript 对象无效，而且无法通过 polyfill 来修正。

- **那么 Vue.js 修改了我的数据喽?**

是，也不是。Vue.js 只是将正常属性转化为 `getters` 和 `setters`，这样它才能在属性被访问或更改时得到通知。而在序列化数据时，结果是完全相同的。当然，这里有一些注意事项：

1. 当你使用 `console.log` 观察对象时你将只能看到一串 `getter/setters`。但你可以使用 `vm.$log()` 来打印出一个更直观的输出。
2. 你不能在数据对象里定义自己的 `getter/setters`。这通常来说不是问题，因为数据对象应该是从纯 JSON 中解析出来的，而且 Vue.js 提供了计算属性功能。
3. Vue.js 在被观察的对象上添加了一些拓展的属性或方法，比如 `__ob__`，`$add`，`$set` 以及 `$delete`。这些属性是不可遍历的，所以它们不会显示在 `for ... in ...` 循环中。只要你不覆盖它们，就不会有什么问题。

除此之外，访问属性和赋值都和原生对象一样，`JSON.stringify` 和 `for ... in ...` 循环也能够照常运行。99.9% 的情况下你都不需要考虑这些问题。

- **Vue.js 目前的状态怎样? 我能把它应用在生产环境中么?**

Vue.js 最新的 0.12 发布版本，经历了一些 API 变更，但这是 1.0 版本前的最后一个计划发布版本。与此同时，全球各地都已经有公司/产品将 Vue.js 使用在了生产环境中。Vue.js 的测试覆盖率和 bug 修复速度也在同类框架中是首屈一指的。参见：[使用了 Vue.js 的项目（不完全统计）](#)。

- **Vue.js 是完全免费的吗？**

Vue.js 基于 MIT 协议发布，是完全开源且免费的。

- **Vue.js 和 AngularJS 之间的区别是什么？**

下面是一些选择 Vue 而不是 Angular 的可能原因，当然它们可能不适用于每一个人：

1. Vue.js 是一个更加灵活开放的解决方案。它允许你以希望的方式组织你的应用程序，而不是任何时候都必须遵循 Angular 制定的规则。它仅仅是一个视图层，所以你可以将它嵌入一个现有页面而不一定要做成一个庞大的单页应用。在结合其他库方面它给了你更大的空间，但相应，你也需要做更多的架构决策。例如，Vue.js 核心默认不包含路由和 ajax 功能，并且通常假定你在用应用中使用了一个外部的模块构建系统。这可能是最重要的区别。
2. 在 API 和内部设计方面，Vue.js 比 Angular 简单得多，因此你可以快速地掌握它的全部特性并投入开发。
3. Vue.js 拥有更好的性能，因为它不使用脏检查。当 watcher 越来越多时，Angular 会变得越来越慢，因为作用域内的每一次数据变更，所有的 watcher 都需要被重新求值。Vue 则根本没有这个问题，因为它采用的是

基于依赖追踪的观察系统，所以所有的数据变更触发都是独立的，除非它们之间有明确的依赖关系。

4. Vue.js 中指令和组件的概念区分得更为清晰。指令只负责封装 DOM 操作，而组件代表一个自给自足的独立单元——它拥有自己的视图和数据逻辑。在 Angular 中它们两者间有不少概念上的混淆。

当然，需要指出的是 Vue.js 还是一个相对年轻的项目，而 Angular 经受过大量的实战检验，并且背后有谷歌支持，还有一个更大的社区。

### • Vue.js 和 React.js 有什么区别？

React.js 和 Vue.js 确实有一些相似——它们都提供数据驱动、可组合搭建的视图组件。然而，它们的内部实现是完全不同的。React 是基于 Virtual DOM——一种在内存中描述 DOM 树状态的数据结构。React 中的数据通常被看作是不可变的，而 DOM 操作则是通过 Virtual DOM 的 diff 来计算的。与之相比，Vue.js 中的数据默认是可变的，而数据的变更会直接出发对应的 DOM 更新。相比于 Virtual DOM，Vue.js 使用实际的 DOM 作为模板，并且保持对真实节点的引用来进行数据绑定。

Virtual DOM 提供了一个函数式的描述视图的方法，这很 cool。因为它不使用数据观察机制，每次更新都会重新渲染整个应用，因此从定义上保证了视图与数据的同步。它也开辟了 JavaScript 同构应用的可能性。

实话实说，我自己对 React 的设计理念也是十分欣赏的。但 React 有一个问题就是组件的逻辑和视图结合得非常紧密。对于部分开发者来说，他们可能觉得这是个优点，但对那些像我一样兼顾设计和开发的人来说，还是更偏好模板，因为模板能让我们更好地在视觉上思考设计和 CSS。JSX 和 JavaScript 逻辑的混合干扰了我将

代码映射到设计的思维过程。相反，Vue.js 通过在模板中加入一个轻量级的 DSL (指令系统)，换来一个依旧直观的模板，且能够将逻辑封装进指令和过滤器中。

React 的另一个问题是：由于 DOM 更新完全交由 Virtual DOM 管理，当你真的想要自己控制 DOM 是就有点棘手了（虽然理论上你可以，但这样做时你本质上在对抗 React 的设计思想）。对于需要复杂时间控制的动画来说这就变成了一项很讨人厌的限制。在这方面，Vue.js 允许更多的灵活性，并且有不少用 Vue.js 构建的富交互实例。**参见一些用 Vue.js 制作的 FWA/Awwwards 获奖站点**

- **Vue.js 和 Polymer 有什么区别?**

Polymer 是另一个由 Google 支持的项目，实际上也是 Vue.js 的灵感来源之一。Vue.js 的组件可以类比为 Polymer 中的自定义元素，它们都提供类似的开发体验。最大的不同在于，Polymer 依赖最新的 Web Components 特性，在不支持的浏览器中，需要加载笨重的 polyfill，性能也会受到影响。相对的，Vue.js 无需任何依赖，最低兼容到 IE9。

- **Vue.js 和 KnockoutJS 有什么区别?**

首先，Vue 提供了一个更清晰的语法来获取/设置 VM 属性。

在更高层面上，Vue 与 Knockout 的区别是，Vue 的组件系统鼓励你使用自上而下、结构优先、声明式的设计策略，而不是从下而上命令式的创建 ViewModel。Vue 的源数据是一个纯对象，不包含逻辑，可以直接 JSON.stringify 并传给 post 请求。ViewModel 代理了数据对象。Vue 实例始终用原生数据绑定相应的 DOM 元素。Knockout 的 ViewModel 本质上是数据，Model 与

ViewModel 的界限非常模糊，很可能导致过于复杂的  
ViewModel 逻辑。

# 极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/vue-js/>