# 第1章 安装

## 1.1 独立版本

直接下载并用 <script> 标签引入，Vue 会被注册为一个全局变量。重要提示：在开发时请用开发版本，遇到常见错误它会给出友好的警告。

### 开发版本

https://v1-cn.vuejs.org/js/vue.js

### 生产版本

<https://v1-cn.vuejs.org/js/vue.min.js>

### CDN

可以从 jsdelivr 或 cdnjs 获取（版本更新可能略滞后）。

也可以使用 npmcdn，这个链接指向发布到 npm 上的最新稳定版本。 可以在 npmcdn.com/vue/ 上查看包的源码。

### CSP兼容版本

有些环境，如 Google Chrome Apps，强制应用内容安全策略 (CSP) ，不能使用 new Function() 对表达式求值。这时可以用 CSP 兼容版本。

## 1.2 NPM

在用 Vue.js 构建大型应用时推荐使用 NPM 安装，NPM 能很好地和诸如 Webpack 或 Browserify 的 CommonJS 模块打包器配合使用。Vue.js 也提供配套工具来开发单文件组件。

# 最新稳定版本

$ npm install vue

# 最新稳定 CSP 兼容版本

$ npm install vue@csp

## 1.3 命令行工具

Vue.js 提供一个官方命令行工具，可用于快速搭建大型单页应用。该工具提供开箱即用的构建工具配置，带来现代化的前端开发流程。只需一分钟即可启动带热重载、保存时静态检查以及可用于生产环境的构建配置的项目：

# 全局安装 vue-cli

$ npm install -g vue-cli

# 创建一个基于 "webpack" 模板的新项目

$ vue init webpack my-project

# 安装依赖，走你

$ cd my-project

$ npm install

$ npm run dev

## 1.4 开发版本

重要：发布到 NPM 上的 CommonJS 包 (vue.common.js) 只在发布新版本时签入 master 分支，所以这些文件在 dev 分支下跟稳定版本是一样的。想使用 GitHub 上最新的源码，需要自己编译：

git clone https://github.com/vuejs/vue.git node\_modules/vue

cd node\_modules/vue

npm install

npm run build

## 1.5 Brower

# 最新稳定版本

$ bower install vue

## 1.6 AMD 模块加载器

独立版本或通过 Bower 安装的版本已用 UMD 包装，因此它们可以直接用作 AMD 模块。

# 第2章 起步

我们以 Vue 数据绑定的快速导览开始。如果你对高级概述更感兴趣，可查看这篇博文。

尝试 Vue.js 最简单的方法是使用 JSFiddle Hello World 例子。在浏览器新标签页中打开它，跟着我们查看一些基础示例。如果你喜欢用包管理器下载/安装，查看安装（https://v1-cn.vuejs.org/guide/installation.html）教程。

## 2.1 HelloWord

|  |
| --- |
| <div id="app">  {{ message }}  </div>  new Vue({  el: '#app',  data: {  message: 'Hello Vue.js!'  }  } |

## 2.2 双向绑定

<div id="app">

<p>{{ message }}</p>

<input v-model="message">

</div>

new Vue({

el: '#app',

data: {

message: 'Hello Vue.js!'

}

})

## 2.3 渲染列表

<div id="app">

<ul>

<li v-for="todo in todos">

{{ todo.text }}

</li>

</ul>

</div>

new Vue({

el: '#app',

data: {

todos: [

{ text: 'Learn JavaScript' },

{ text: 'Learn Vue.js' },

{ text: 'Build Something Awesome' }

]

}

})

## 2.4 处理用户输入

<div id="app">

<p>{{ message }}</p>

<button v-on:click="reverseMessage">Reverse Message</button>

</div>

new Vue({

el: '#app',

data: {

message: 'Hello Vue.js!'

},

methods: {

reverseMessage: function () {

this.message = this.message.split('').reverse().join('')

}

}

})

## 2.5 综合

<div id="app">

<input v-model="newTodo" v-on:keyup.enter="addTodo">

<ul>

<li v-for="todo in todos">

<span>{{ todo.text }}</span>

<button v-on:click="removeTodo($index)">X</button>

</li>

</ul>

</div>

new Vue({

el: '#app',

data: {

newTodo: '',

todos: [

{ text: 'Add some todos' }

]

},

methods: {

addTodo: function () {

var text = this.newTodo.trim()

if (text) {

this.todos.push({ text: text })

this.newTodo = ''

}

},

removeTodo: function (index) {

this.todos.splice(index, 1)

}

}

})

# 第3 章 概述

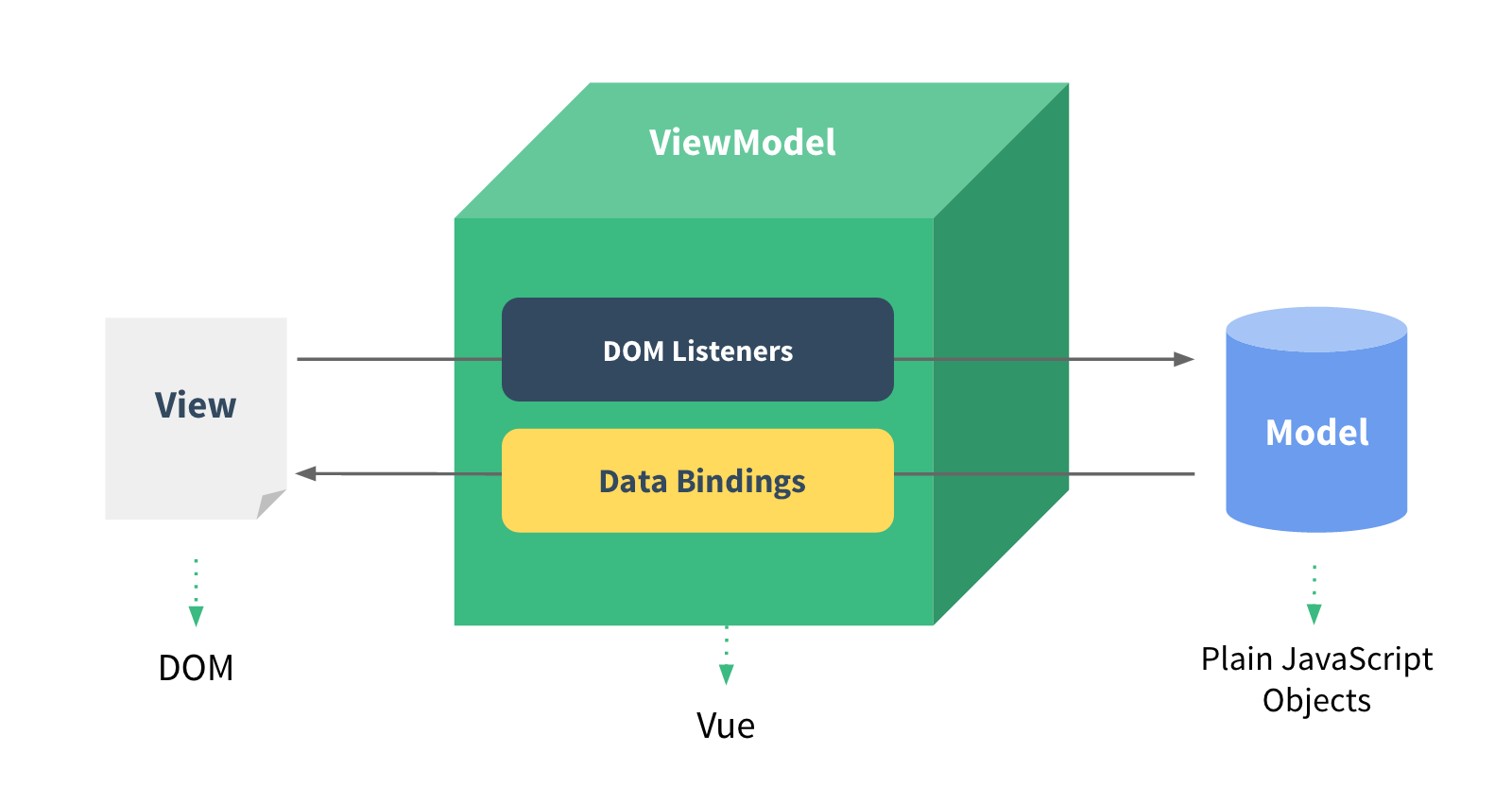
Vue.js（读音 /vjuː/, 类似于 view）是一个构建数据驱动的 web 界面的库。Vue.js 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。

Vue.js 自身不是一个全能框架——它只聚焦于视图层。因此它非常容易学习，非常容易与其它库或已有项目整合。另一方面，在与相关工具和支持库一起使用时，Vue.js 也能完美地驱动复杂的单页应用。

如果你是有经验的前端开发者，想知道 Vue.js 与其它库/框架的区别，查看对比其它框架；如果你对使用 Vue.js 开发大型应用更感兴趣，查看构建大型应用。

## 3.1 响应的数据绑定

Vue. js 的核心是一个响应的数据绑定系统，它让数据与 DOM 保持同步非常简单。在使用 jQuery 手工操作 DOM 时，我们的代码常常是命令式的、重复的与易错的。Vue.js 拥抱数据驱动的视图概念。通俗地讲，它意味着我们在普通 HTML 模板中使用特殊的语法将 DOM “绑定”到底层数据。一旦创建了绑定，DOM 将与数据保持同步。每当修改了数据，DOM 便相应地更新。这样我们应用中的逻辑就几乎都是直接修改数据了，不必与 DOM 更新搅在一起。这让我们的代码更容易撰写、理解与维护。



可能是最简单的例子：

<!-- 这是我们的 View -->

<div id="example-1">

Hello {{ name }}!

</div>

// 这是我们的 Model

var exampleData = {

name: 'Vue.js'

}

// 创建一个 Vue 实例或 "ViewModel"

// 它连接 View 与 Model

var exampleVM = new Vue({

el: '#example-1',

data: exampleData

})

看起来这跟单单渲染一个模板非常类似，但是 Vue.js 在背后做了大量工作。并且 DOM 会自动响应数据的变化。我们如何知道？打开你的浏览器的控制台，修改 exampleData.name，你将看到上例相应地更新。

注意我们不需要撰写任何 DOM 操作代码：被绑定增强的 HTML 模板是底层数据状态的声明式的映射，数据不过是普通 JavaScript 对象。我们的视图完全由数据驱动。

让我们来看第二个例子：

<div id="example-2">

<p v-if="greeting">Hello!</p>

</div>

var exampleVM2 = new Vue({

el: '#example-2',

data: {

greeting: true

}

})

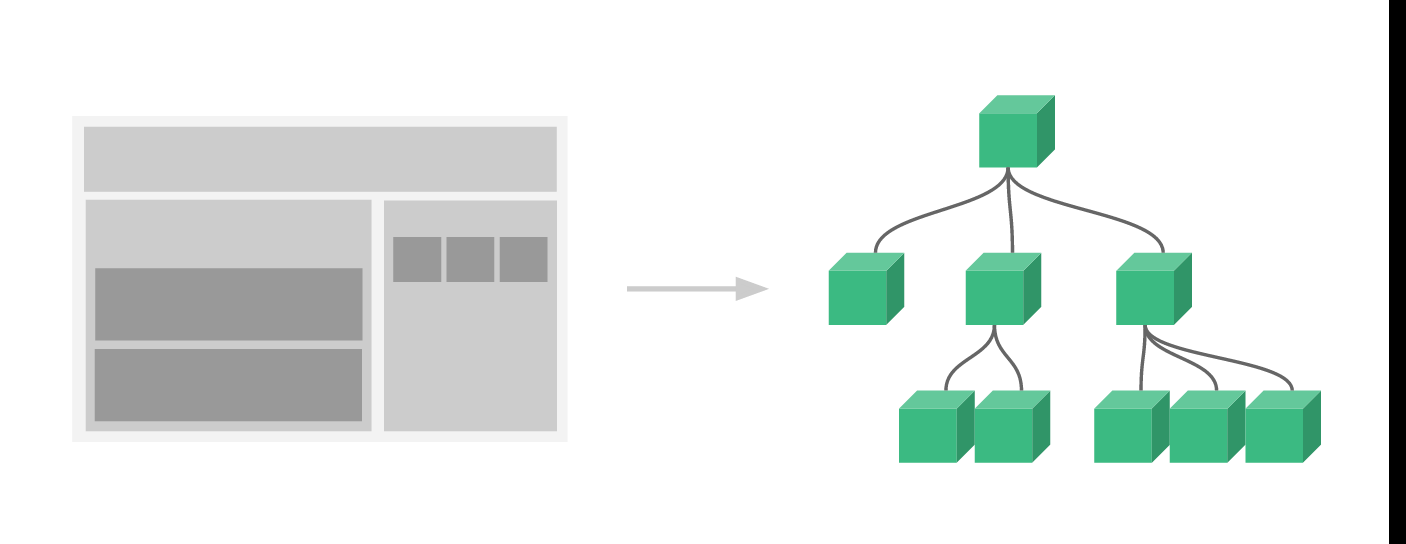
这里我们遇到新东西。你看到的 v-if 特性被称为指令。指令带有前缀 v-，以指示它们是 Vue.js 提供的特殊特性。并且如你所想象的，它们会对绑定的目标元素添加响应式的特殊行为。继续在控制台设置 exampleVM2.greeting 为 false，你会发现 “Hello!” 消失了。

第二个例子演示了我们不仅可以绑定 DOM 文本到数据，也可以绑定 DOM 结构 到数据。而且，Vue.js 也提供一个强大的过渡效果系统，可以在 Vue 插入/删除元素时自动应用过渡效果。

也有一些其它指令，每个都有特殊的功能。例如 v-for 指令用于显示数组元素，v-bind 指令用于绑定 HTML 特性。我们将在后面详细讨论全部的数据绑定语法。

## 3.2 组件系统

组件系统是 Vue.js 另一个重要概念，因为它提供了一种抽象，让我们可以用独立可复用的小组件来构建大型应用。如果我们考虑到这点，几乎任意类型的应用的界面都可以抽象为一个组件树：



实际上，一个典型的用 Vue.js 构建的大型应用将形成一个组件树。在后面的教程中我们将详述组件，不过这里有一个假想的例子，看看使用了组件的应用模板是什么样的：

<div id="app">

<app-nav></app-nav>

<app-view>

<app-sidebar></app-sidebar>

<app-content></app-content>

</app-view>

</div>

你可能已经注意到 Vue.js 组件非常类似于自定义元素——它是 Web 组件规范的一部分。实际上 Vue.js 的组件语法参考了该规范。例如 Vue 组件实现了 Slot API 与 is 特性。但是，有几个关键的不同：

Web 组件规范仍然远未完成，并且没有浏览器实现。相比之下，Vue.js 组件不需要任何补丁，并且在所有支持的浏览器（IE9 及更高版本）之下表现一致。必要时，Vue.js 组件也可以放在原生自定义元素之内。

Vue.js 组件提供了原生自定义元素所不具备的一些重要功能，比如组件间的数据流，自定义事件系统，以及动态的、带特效的组件替换。

组件系统是用 Vue.js 构建大型应用的基础。另外，Vue.js 生态系统也提供了高级工具与多种支持库，它们和 Vue.js 一起构成了一个更加“框架”性的系统。

# 第4 章 Vue示例

## 4.1 构造器

每个 Vue.js 应用的起步都是通过构造函数 Vue 创建一个 Vue 的根实例：

var vm = new Vue({

// 选项

})

一个 Vue 实例其实正是一个 MVVM 模式中所描述的 ViewModel - 因此在文档中经常会使用 vm 这个变量名。

在实例化 Vue 时，需要传入一个选项对象，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 API 文档中查看。

可以扩展 Vue 构造器，从而用预定义选项创建可复用的组件构造器：

**var MyComponent = Vue.extend({**

**// 扩展选项**

**})**

**// 所有的 `MyComponent` 实例都将以预定义的扩展选项被创建**

**var myComponentInstance = new MyComponent()**

尽管可以命令式地创建扩展实例，不过在多数情况下将组件构造器注册为一个自定义元素，然后声明式地用在模板中。我们将在后面详细说明组件系统。现在你只需知道所有的 Vue.js 组件其实都是被扩展的 Vue 实例。

## 4.2 属性与方法

每个 Vue 实例都会代理其 data 对象里所有的属性：

var data = { a: 1 }

var vm = new Vue({

data: data

})

vm.a === data.a // -> true

// 设置属性也会影响到原始数据

vm.a = 2

data.a // -> 2

// ... 反之亦然

data.a = 3

vm.a // -> 3

注意只有这些被代理的属性是响应的。如果在实例创建之后添加新的属性到实例上，它不会触发视图更新。我们将在后面详细讨论响应系统。

除了这些数据属性，Vue 实例暴露了一些有用的实例属性与方法。这些属性与方法都有前缀 $，以便与代理的数据属性区分。例如：

var data = { a: 1 }

var vm = new Vue({

el: '#example',

data: data

})

vm.$data === data // -> true

vm.$el === document.getElementById('example') // -> true

// $watch 是一个实例方法

vm.$watch('a', function (newVal, oldVal) {

// 这个回调将在 `vm.a` 改变后调用

})

参考 API 文档查看全部的实例属性与方法。

## 4.3 实例生命周期

Vue 实例在创建时有一系列初始化步骤——例如，它需要建立数据观察，编译模板，创建必要的数据绑定。在此过程中，它也将调用一些生命周期钩子，给自定义逻辑提供运行机会。例如 created 钩子在实例创建后调用：

var vm = new Vue({

data: {

a: 1

},

created: function () {

// `this` 指向 vm 实例

console.log('a is: ' + this.a)

}

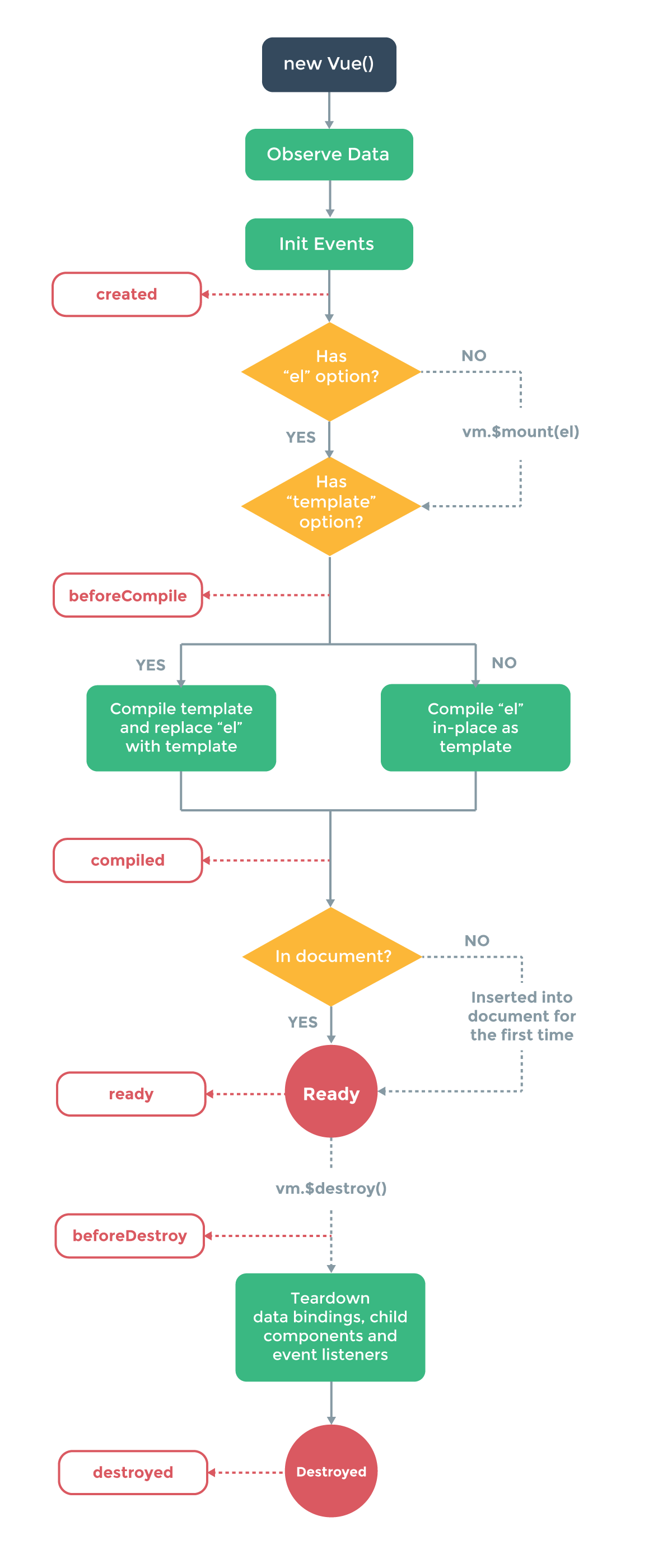
})

// -> "a is: 1"

也有一些其它的钩子，在实例生命周期的不同阶段调用，如 compiled、 ready 、destroyed。钩子的 this 指向调用它的 Vue 实例。一些用户可能会问 Vue.js 是否有“控制器”的概念？答案是，没有。组件的自定义逻辑可以分割在这些钩子中。

## 4.4 生命周期图示

下图说明了实例的生命周期。你不需要立马弄明白所有的东西，不过以后它会有帮助。



# 第 5 章 数据绑定语法

Vue.js 的模板是基于 DOM 实现的。这意味着所有的 Vue.js 模板都是可解析的有效的 HTML，且通过一些特殊的特性做了增强。Vue 模板因而从根本上不同于基于字符串的模板，请记住这点。

## 5.1 插值

### 5.1.1 文本

数据绑定最基础的形式是文本插值，使用 “Mustache” 语法（双大括号）：

<span>Message: {{ msg }}</span>

Mustache 标签会被相应数据对象的 msg 属性的值替换。每当这个属性变化时它也会更新。

你也可以只处理单次插值，今后的数据变化就不会再引起插值更新了：

<span>This will never change: {{\* msg }}</span>

### 5.1.2 原始的HTML

双 Mustache 标签将数据解析为纯文本而不是 HTML。为了输出真的 HTML 字符串，需要用三 Mustache 标签：

<div>{{{ raw\_html }}}</div>

内容以 HTML 字符串插入——数据绑定将被忽略。如果需要复用模板片断，应当使用 partials。

在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。记住，只对可信内容使用 HTML 插值，永不用于用户提交的内容。

### 5.1.3 HTML特性

Mustache 标签也可以用在 HTML 特性 (Attributes) 内：

<div id="item-{{ id }}"></div>

注意在 Vue.js 指令和特殊特性内不能用插值。不必担心，如果 Mustache 标签用错了地方 Vue.js 会给出警告。

## 5.2 绑定表达式

放在 Mustache 标签内的文本称为绑定表达式。在 Vue.js 中，一段绑定表达式由一个简单的 JavaScript 表达式和可选的一个或多个过滤器构成。

### 5.2.1 JavaScript表达式

到目前为止，我们的模板只绑定到简单的属性键。不过实际上 Vue.js 在数据绑定内支持全功能的 JavaScript 表达式：

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

这些表达式将在所属的 Vue 实例的作用域内计算。一个限制是每个绑定只能包含单个表达式，因此下面的语句是无效的：

<!-- 这是一个语句，不是一个表达式： -->

{{ var a = 1 }}

<!-- 流程控制也不可以，可改用三元表达式 -->

{{ if (ok) { return message } }}

### 5.2.2 过滤器

Vue.js 允许在表达式后添加可选的“过滤器 (Filter) ”，以“管道符”指示：

{{ message | capitalize }}

这里我们将表达式 message 的值“管输（pipe）”到内置的 capitalize 过滤器，这个过滤器其实只是一个 JavaScript 函数，返回大写化的值。Vue.js 提供数个内置过滤器，在后面我们会谈到如何开发自己的过滤器。

注意管道语法不是 JavaScript 语法，因此不能在表达式内使用过滤器，只能添加到表达式的后面。

过滤器可以串联：

{{ message | filterA | filterB }}

过滤器也可以接受参数：

{{ message | filterA 'arg1' arg2 }}

过滤器函数始终以表达式的值作为第一个参数。带引号的参数视为字符串，而不带引号的参数按表达式计算。这里，字符串 'arg1' 将传给过滤器作为第二个参数，表达式 arg2 的值在计算出来之后作为第三个参数。

## 5.3 指令

指令 (Directives) 是特殊的带有前缀 v- 的特性。指令的值限定为绑定表达式，因此上面提到的 JavaScript 表达式及过滤器规则在这里也适用。指令的职责就是当其表达式的值改变时把某些特殊的行为应用到 DOM 上。我们来回头看下“概述”里的例子：

<p v-if="greeting">Hello!</p>

这里 v-if 指令将根据表达式 greeting 值的真假删除/插入 <p> 元素。

### 5.3.1 参数

有些指令可以在其名称后面带一个“参数” (Argument)，中间放一个冒号隔开。例如，v-bind 指令用于响应地更新 HTML 特性：

<a v-bind:href="url"></a>

这里 href 是参数，它告诉 v-bind 指令将元素的 href 特性跟表达式 url 的值绑定。可能你已注意到可以用特性插值 href="{{url}}" 获得同样的结果：这样没错，并且实际上在内部特性插值会转为 v-bind 绑定。

另一个例子是 v-on 指令，它用于监听 DOM 事件：

<a v-on:click="doSomething">

这里参数是被监听的事件的名字。我们也会详细说明事件绑定。

### 5.3.2 修饰符

修饰符 (Modifiers) 是以半角句号 . 开始的特殊后缀，用于表示指令应当以特殊方式绑定。例如 .literal 修饰符告诉指令将它的值解析为一个字面字符串而不是一个表达式：

<a v-bind:href.literal="/a/b/c"></a>

当然，这似乎没有意义，因为我们只需要使用 href="/a/b/c" 而不必使用一个指令。这个例子只是为了演示语法。后面我们将看到修饰符更多的实践用法。

## 5.4 缩写

v- 前缀是一种标识模板中特定的 Vue 特性的视觉暗示。当你需要在一些现有的 HTML 代码中添加动态行为时，这些前缀可以起到很好的区分效果。但你在使用一些常用指令的时候，你会感觉一直这么写实在是啰嗦。而且在构建单页应用（SPA ）时，Vue.js 会管理所有的模板，此时 v- 前缀也没那么重要了。因此Vue.js 为两个最常用的指令 v-bind 和 v-on 提供特别的缩写：

### 5.4.1 v-bind 缩写

<!-- 完整语法 -->

<a v-bind:href="url"></a>

<!-- 缩写 -->

<a :href="url"></a>

<!-- 完整语法 -->

<button v-bind:disabled="someDynamicCondition">Button</button>

<!-- 缩写 -->

<button :disabled="someDynamicCondition">Button</button>

### 5.4.2 v-on 缩写

<!-- 完整语法 -->

<a v-on:click="doSomething"></a>

<!-- 缩写 -->

<a @click="doSomething"></a>

它们看起来跟“合法”的 HTML 有点不同，但是它们在所有 Vue.js 支持的浏览器中都能被正确地解析，并且不会出现在最终渲染的标记中。缩写语法完全是可选的，不过随着一步步学习的深入，你会庆幸拥有它们。

# 第6章 计算属性

在模板中绑定表达式是非常便利的，但是它们实际上只用于简单的操作。模板是为了描述视图的结构。在模板中放入太多的逻辑会让模板过重且难以维护。这就是为什么 Vue.js 将绑定表达式限制为一个表达式。如果需要多于一个表达式的逻辑，应当使用计算属性。

## 6.1 基础例子

<div id="example">

a={{ a }}, b={{ b }}

</div>

var vm = new Vue({

el: '#example',

data: {

a: 1

},

computed: {

// 一个计算属性的 getter

b: function () {

// `this` 指向 vm 实例

return this.a + 1

}

}

})

结果：

a = 1 ,b = 2

这里我们声明了一个计算属性 b。我们提供的函数将用作属性 vm.b的 getter。

console.log(vm.b) // -> 2

vm.a = 2

console.log(vm.b) // -> 3

你可以打开浏览器的控制台，修改 vm。vm.b 的值始终取决于 vm.a 的值。

你可以像绑定普通属性一样在模板中绑定计算属性。Vue 知道 vm.b 依赖于 vm.a，因此当 vm.a 发生改变时，依赖于 vm.b 的绑定也会更新。而且最妙的是我们是声明式地创建这种依赖关系：计算属性的 getter 是干净无副作用的，因此也是易于测试和理解的。

## 6.2 计算属性 vs.$watch

Vue.js 提供了一个方法 $watch，它用于观察 Vue 实例上的数据变动。当一些数据需要根据其它数据变化时， $watch 很诱人 —— 特别是如果你来自 AngularJS。不过，通常更好的办法是使用计算属性而不是一个命令式的 $watch 回调。考虑下面例子：

<div id="demo">{{fullName}}</div>

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar',

fullName: 'Foo Bar'

}

})

vm.$watch('firstName', function (val) {

this.fullName = val + ' ' + this.lastName

})

vm.$watch('lastName', function (val) {

this.fullName = this.firstName + ' ' + val

})

上面代码是命令式的重复的。跟计算属性对比：

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar'

},

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

})

6.3 计算 setter

计算属性默认只是 getter，不过在需要时你也可以提供一个 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]

}

}

}

// ...

现在在调用 vm.fullName = 'John Doe' 时，setter 会被调用，vm.firstName 和 vm.lastName 也会有相应更新。

关于计算属性背后的原理和技术细节详见计算属性的奥秘。

# 第 7 章 Class 与 style 绑定

数据绑定一个常见需求是操作元素的 class 列表和它的内联样式。因为它们都是 attribute，我们可以用 v-bind 处理它们：只需要计算出表达式最终的字符串。不过，字符串拼接麻烦又易错。因此，在 v-bind 用于 class 和 style 时，Vue.js 专门增强了它。表达式的结果类型除了字符串之外，还可以是对象或数组。

## 7.1 绑定HTML Class

尽管可以用 Mustache 标签绑定 class，比如 class="{{ className }}"，但是我们不推荐这种写法和 v-bind:class 混用。两者只能选其一！

### 7.1.1 对象语法

我们可以传给 v-bind:class 一个对象，以动态地切换 class。注意 v-bind:class 指令可以与普通的 class 特性共存：

<div class="static" v-bind:class="{ 'class-a': isA, 'class-b': isB }"></div>

data: {

isA: true,

isB: false

}

渲染为：

<div class="static class-a"></div>

当 isA 和 isB 变化时，class 列表将相应地更新。例如，如果 isB 变为 true，class 列表将变为 "static class-a class-b"。

你也可以直接绑定数据里的一个对象：

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

data: {

classObject: {

'class-a': true,

'class-b': false

}

}

我们也可以在这里绑定一个返回对象的计算属性。这是一个常用且强大的模式。

7.1.2 数组语法