# Essentials 基础

## Installation 安装

Compatibility Note 兼容性

Vue does not support IE8 and below, because it uses ECMAScript 5 features that are un-shimmable in IE8. However it supports all ECMAScript 5 compliant browsers.

Vue 不支持 IE8 及以下版本，因为 Vue 使用了 IE8 无法模拟的 ECMAScript 5 特性。但它支持所有兼容 ECMAScript 5 的浏览器。

Release Notes 更新日志

Latest stable version: 2.6.10 最新稳定版本：2.6.10

Detailed release notes for each version are available on GitHub.

每个版本的更新日志见 GitHub。

When using Vue, we recommend also installing the Vue Devtools in your browser, allowing you to inspect and debug your Vue applications in a more user-friendly interface.

在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

Direct <script> Include

直接用 <script> 引入

Simply download and include with a script tag. Vue will be registered as a global variable.

直接下载并用 <script> 标签引入，Vue 会被注册为一个全局变量。

Don’t use the minified version during development. You will miss out on all the nice warnings for common mistakes!

在开发环境下不要使用压缩版本，不然你就失去了所有常见错误相关的警告!

Development Version：With full warnings and debug mode

开发版本：包含完整的警告和调试模式  
  
Production Version：Warnings stripped, 33.30KB min+gzip

生产版本：删除了警告，33.30KB min+gzip

For prototyping or learning purposes, you can use the latest version with:

对于制作原型或学习，你可以这样使用最新版本：

For production, we recommend linking to a specific version number and build to avoid unexpected breakage from newer versions:

对于生产环境，我们推荐链接到一个明确的版本号和构建文件，以避免新版本造成的不可预期的破坏：

If you are using native ES Modules, there is also an ES Modules compatible build:

如果你使用原生 ES Modules，这里也有一个兼容 ES Module 的构建文件：

You can browse the source of the NPM package at cdn.jsdelivr.net/npm/vue.

你可以在 cdn.jsdelivr.net/npm/vue 浏览 NPM 包的源代码。

Vue is also available on unpkg and cdnjs (cdnjs takes some time to sync so the latest release may not be available yet).

Vue 也可以在 unpkg 和 cdnjs 上获取 (cdnjs 的版本更新可能略滞后)。

Make sure to read about the different builds of Vue and use the production  
version in your published site, replacing vue.js with vue.min.js. This is a smaller build optimized for speed instead of development experience.

请确认了解不同构建版本并在你发布的站点中使用生产环境版本，把 vue.js 换成 vue.min.js。这是一个更小的构建，可以带来比开发环境下更快的速度体验。

NPM is the recommended installation method when building large scale applications with Vue. It pairs nicely with module bundlers such as Webpack or Browserify. Vue also provides accompanying tools for authoring Single File Components.

在用 Vue 构建大型应用时推荐使用 NPM 安装[1]。NPM 能很好地和诸如 webpack 或 Browserify 模块打包器配合使用。同时 Vue 也提供配套工具来开发单文件组件。

CLI 命令行工具 (CLI)

Vue provides an official CLI for quickly scaffolding ambitious Single Page Applications. It provides batteries-included build setups for a modern frontend workflow. It takes only a few minutes to get up and running with hot-reload, lint-on-save, and production-ready builds. See the Vue CLI docs for more details.

Vue 提供了一个官方的 CLI，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。更多详情可查阅 Vue CLI 的文档。

The CLI assumes prior knowledge of Node.js and the associated build tools. If you are new to Vue or front-end build tools, we strongly suggest going through the guide without any build tools before using the CLI.

CLI 工具假定用户对 Node.js 和相关构建工具有一定程度的了解。如果你是新手，我们强烈建议先在不用构建工具的情况下通读指南，在熟悉 Vue 本身之后再使用 CLI。

Explanation of Different Builds 对不同构建版本的解释

In the dist/ directory of the NPM package you will find many different builds of Vue.js. Here’s an overview of the difference between them:

在 NPM 包的 dist/ 目录你将会找到很多不同的 Vue.js 构建版本。这里列出了它们之间的差别：

Terms 术语

Full: builds that contain both the compiler and the runtime.

完整版：同时包含编译器和运行时的版本。

Compiler: code that is responsible for compiling template strings into JavaScript render functions.

编译器：用来将模板字符串编译成为 JavaScript 渲染函数的代码。

Runtime: code that is responsible for creating Vue instances, rendering and patching virtual DOM, etc. Basically everything minus the compiler.

运行时：用来创建 Vue 实例、渲染并处理虚拟 DOM 等的代码。基本上就是除去编译器的其它一切。

UMD: UMD builds can be used directly in the browser via a <script> tag. The default file from jsDelivr CDN at https://cdn.jsdelivr.net/npm/vue is the Runtime + Compiler UMD build (vue.js).

UMD：UMD 版本可以通过 <script> 标签直接用在浏览器中。jsDelivr CDN 的 https://cdn.jsdelivr.net/npm/vue 默认文件就是运行时 + 编译器的 UMD 版本 (vue.js)。

CommonJS: CommonJS builds are intended for use with older bundlers like browserify or webpack 1. The default file for these bundlers (pkg.main) is the Runtime only CommonJS build (vue.runtime.common.js).

CommonJS：CommonJS 版本用来配合老的打包工具比如 Browserify 或 webpack 1。这些打包工具的默认文件 (pkg.main) 是只包含运行时的 CommonJS 版本 (vue.runtime.common.js)。

ES Module: starting in 2.6 Vue provides two ES Modules (ESM) builds:

ES Module：从 2.6 开始 Vue 会提供两个 ES Modules (ESM) 构建文件：

ESM for bundlers: intended for use with modern bundlers like webpack 2 or Rollup. ESM format is designed to be statically analyzable so the bundlers can take advantage of that to perform “tree-shaking” and eliminate unused code from your final bundle. The default file for these bundlers (pkg.module) is the Runtime only ES Module build (vue.runtime.esm.js).

为打包工具提供的 ESM：为诸如 webpack 2 或 Rollup 提供的现代打包工具。ESM 格式被设计为可以被静态分析，所以打包工具可以利用这一点来进行“tree-shaking”并将用不到的代码排除出最终的包。为这些打包工具提供的默认文件 (pkg.module) 是只有运行时的 ES Module 构建 (vue.runtime.esm.js)。

ESM for browsers (2.6+ only): intended for direct imports in modern browsers via <script type="module">.

为浏览器提供的 ESM (2.6+)：用于在现代浏览器中通过 <script type="module"> 直接导入。

Runtime + Compiler vs. Runtime-only 运行时 + 编译器 vs. 只包含运行时

If you need to compile templates on the client (e.g. passing a string to the template option, or mounting to an element using its in-DOM HTML as the template), you will need the compiler and thus the full build:

如果你需要在客户端编译模板 (比如传入一个字符串给 template 选项，或挂载到一个元素上并以其 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

When using vue-loader or vueify, templates inside \*.vue files are pre-compiled into JavaScript at build time. You don’t really need the compiler in the final bundle, and can therefore use the runtime-only build.

当使用 vue-loader 或 vueify 的时候，\*.vue 文件内部的模板会在构建时预编译成 JavaScript。你在最终打好的包里实际上是不需要编译器的，所以只用运行时版本即可。

Since the runtime-only builds are roughly 30% lighter-weight than their full-build counterparts, you should use it whenever you can. If you still wish to use the full build instead, you need to configure an alias in your bundler:

因为运行时版本相比完整版体积要小大约 30%，所以应该尽可能使用这个版本。如果你仍然希望使用完整版，则需要在打包工具里配置一个别名：

Development vs. Production Mode 开发环境 vs. 生产环境模式

Development/production modes are hard-coded for the UMD builds: the un-minified files are for development, and the minified files are for production.

对于 UMD 版本来说，开发环境/生产环境模式是硬编码好的：开发环境下用未压缩的代码，生产环境下使用压缩后的代码。

CommonJS and ES Module builds are intended for bundlers, therefore we don’t provide minified versions for them. You will be responsible for minifying the final bundle yourself.

CommonJS 和 ES Module 版本是用于打包工具的，因此我们不提供压缩后的版本。你需要自行将最终的包进行压缩。

CommonJS and ES Module builds also preserve raw checks for process.env.NODE\_ENV to determine the mode they should run in. You should use appropriate bundler configurations to replace these environment variables in order to control which mode Vue will run in. Replacing process.env.NODE\_ENV with string literals also allows minifiers like UglifyJS to completely drop the development-only code blocks, reducing final file size.

CommonJS 和 ES Module 版本同时保留原始的 process.env.NODE\_ENV 检测，以决定它们应该运行在什么模式下。你应该使用适当的打包工具配置来替换这些环境变量以便控制 Vue 所运行的模式。把 process.env.NODE\_ENV 替换为字符串字面量同时可以让 UglifyJS 之类的压缩工具完全丢掉仅供开发环境的代码块，以减少最终的文件尺寸。

CSP environments CSP 环境

Some environments, such as Google Chrome Apps, enforce Content Security Policy (CSP), which prohibits the use of new Function() for evaluating expressions. The full build depends on this feature to compile templates, so is unusable in these environments.

有些环境，如 Google Chrome Apps，会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。完整版本依赖于该功能来编译模板，所以无法在这些环境下使用。

On the other hand, the runtime-only build is fully CSP-compliant. When using the runtime-only build with Webpack + vue-loader or Browserify + vueify, your templates will be precompiled into renderfunctions which work perfectly in CSP environments.

另一方面，运行时版本则是完全兼容 CSP 的。当通过 webpack + vue-loader 或者 Browserify + vueify 构建时，模板将被预编译为 render 函数，可以在 CSP 环境中完美运行。

Dev Build 开发版本

Important: the built files in GitHub’s /dist folder are only checked-in during releases. To use Vue from the latest source code on GitHub, you will have to build it yourself!

重要: GitHub 仓库的 /dist 文件夹只有在新版本发布时才会提交。如果想要使用 GitHub 上 Vue 最新的源码，你需要自己构建！

AMD Module Loaders AMD 模块加载器

All UMD builds can be used directly as an AMD module.

所有 UMD 版本都可以直接用作 AMD 模块。

## Introduction介绍

What is Vue.js? Vue.js 是什么

Vue (pronounced /vjuː/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.

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。

If you’d like to learn more about Vue before diving in, we created a video walking through the core principles and a sample project.

如果你想在深入学习 Vue 之前对它有更多了解，我们制作了一个视频，带您了解其核心概念和一个示例工程。

If you are an experienced frontend developer and want to know how Vue compares to other libraries/frameworks, check out the Comparison with Other Frameworks.

如果你已经是有经验的前端开发者，想知道 Vue 与其它库/框架有哪些区别，请查看对比其它框架。

Getting Started 起步

The official guide assumes intermediate level knowledge of HTML, CSS, and JavaScript. If you are totally new to frontend development, it might not be the best idea to jump right into a framework as your first step - grasp the basics then come back! Prior experience with other frameworks helps, but is not required.

官方指南假设你已了解关于 HTML、CSS 和 JavaScript 的中级知识。如果你刚开始学习前端开发，将框架作为你的第一步可能不是最好的主意——掌握好基础知识再来吧！之前有其它框架的使用经验会有帮助，但这不是必需的。

The easiest way to try out Vue.js is using the JSFiddle Hello World example. Feel free to open it in another tab and follow along as we go through some basic examples. Or, you can create an index.html file and include Vue with:

尝试 Vue.js 最简单的方法是使用 JSFiddle 上的 Hello World 例子。你可以在浏览器新标签页中打开它，跟着例子学习一些基础用法。或者你也可以创建一个 .html 文件，然后通过如下方式引入 Vue：

The Installation page provides more options of installing Vue. Note: We do not recommend that beginners start with vue-cli, especially if you are not yet familiar with Node.js-based build tools.

安装教程给出了更多安装 Vue 的方式。请注意我们不推荐新手直接使用 vue-cli，尤其是在你还不熟悉基于 Node.js 的构建工具时。

If you prefer something more interactive, you can also check out this tutorial series on Scrimba, which gives you a mix of screencast and code playground that you can pause and play around with anytime.

如果你喜欢交互式的东西，你也可以查阅这个 Scrimba 上的系列教程，它揉合了录屏和代码试验田，并允许你随时暂停和播放。

Declarative Rendering 声明式渲染

At the core of Vue.js is a system that enables us to declaratively render data to the DOM using straightforward template syntax:

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统：

We have already created our very first Vue app! This looks pretty similar to rendering a string template, but Vue has done a lot of work under the hood. The data and the DOM are now linked, and everything is now reactive. How do we know? Open your browser’s JavaScript console (right now, on this page) and set app.message to a different value. You should see the rendered example above update accordingly.

我们已经成功创建了第一个 Vue 应用！看起来这跟渲染一个字符串模板非常类似，但是 Vue 在背后做了大量工作。现在数据和 DOM 已经被建立了关联，所有东西都是响应式的。我们要怎么确认呢？打开你的浏览器的 JavaScript 控制台 (就在这个页面打开)，并修改 app.message 的值，你将看到上例相应地更新。

In addition to text interpolation, we can also bind element attributes like this:

除了文本插值，我们还可以像这样来绑定元素特性：

Hover your mouse over me for a few seconds to see my dynamically bound title!  
鼠标悬停几秒钟查看此处动态绑定的提示信息！

Here we are encountering something new. The v-bind attribute you are seeing is called a directive. Directives are prefixed with v- to indicate that they are special attributes provided by Vue, and as you may have guessed, they apply special reactive behavior to the rendered DOM. Here, it is basically saying “keep this element’s title attribute up-to-date with the message property on the Vue instance.”

这里我们遇到了一点新东西。你看到的 v-bind 特性被称为指令。指令带有前缀 v-，以表示它们是 Vue 提供的特殊特性。可能你已经猜到了，它们会在渲染的 DOM 上应用特殊的响应式行为。在这里，该指令的意思是：“将这个元素节点的 title 特性和 Vue 实例的 message 属性保持一致”。

If you open up your JavaScript console again and enter app2.message = 'some new message', you’ll once again see that the bound HTML - in this case the title attribute - has been updated.

如果你再次打开浏览器的 JavaScript 控制台，输入 app2.message = '新消息'，就会再一次看到这个绑定了 title 特性的 HTML 已经进行了更新。

Conditionals and Loops 条件与循环

It’s easy to toggle the presence of an element, too:

控制切换一个元素是否显示也相当简单：

Now you see me 现在你看到我了

Go ahead and enter app3.seen = false in the console. You should see the message disappear.

继续在控制台输入 app3.seen = false，你会发现之前显示的消息消失了。

This example demonstrates that we can bind data to not only text and attributes, but also the structureof the DOM. Moreover, Vue also provides a powerful transition effect system that can automatically apply transition effects when elements are inserted/updated/removed by Vue.

这个例子演示了我们不仅可以把数据绑定到 DOM 文本或特性，还可以绑定到 DOM 结构。此外，Vue 也提供一个强大的过渡效果系统，可以在 Vue 插入/更新/移除元素时自动应用过渡效果。

There are quite a few other directives, each with its own special functionality. For example, the v-fordirective can be used for displaying a list of items using the data from an Array:

还有其它很多指令，每个都有特殊的功能。例如，v-for 指令可以绑定数组的数据来渲染一个项目列表：

In the console, enter app4.todos.push({ text: 'New item' }). You should see a new item appended to the list.

在控制台里，输入 app4.todos.push({ text: '新项目' })，你会发现列表最后添加了一个新项目。

Handling User Input 处理用户输入

To let users interact with your app, we can use the v-on directive to attach event listeners that invoke methods on our Vue instances:

为了让用户和你的应用进行交互，我们可以用 v-on 指令添加一个事件监听器，通过它调用在 Vue 实例中定义的方法：

Note that in this method we update the state of our app without touching the DOM - all DOM manipulations are handled by Vue, and the code you write is focused on the underlying logic.

注意在 reverseMessage 方法中，我们更新了应用的状态，但没有触碰 DOM——所有的 DOM 操作都由 Vue 来处理，你编写的代码只需要关注逻辑层面即可。

Vue also provides the v-model directive that makes two-way binding between form input and app state a breeze:

Vue 还提供了 v-model 指令，它能轻松实现表单输入和应用状态之间的双向绑定。

Composing with Components 组件化应用构建

The component system is another important concept in Vue, because it’s an abstraction that allows us to build large-scale applications composed of small, self-contained, and often reusable components. If we think about it, almost any type of application interface can be abstracted into a tree of components:

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：

In Vue, a component is essentially a Vue instance with pre-defined options. Registering a component in Vue is straightforward:

在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。在 Vue 中注册组件很简单：

Now you can compose it in another component’s template:

现在你可以用它构建另一个组件模板：

But this would render the same text for every todo, which is not super interesting. We should be able to pass data from the parent scope into child components. Let’s modify the component definition to make it accept a prop:

但是这样会为每个待办项渲染同样的文本，这看起来并不炫酷。我们应该能从父作用域将数据传到子组件才对。让我们来修改一下组件的定义，使之能够接受一个 prop：

Now we can pass the todo into each repeated component using v-bind:

现在，我们可以使用 v-bind 指令将待办项传到循环输出的每个组件中：

This is a contrived example, but we have managed to separate our app into two smaller units, and the child is reasonably well-decoupled from the parent via the props interface. We can now further improve our <todo-item> component with more complex template and logic without affecting the parent app.

尽管这只是一个刻意设计的例子，但是我们已经设法将应用分割成了两个更小的单元。子单元通过 prop 接口与父单元进行了良好的解耦。我们现在可以进一步改进 <todo-item> 组件，提供更为复杂的模板和逻辑，而不会影响到父单元。

In a large application, it is necessary to divide the whole app into components to make development manageable. We will talk a lot more about components later in the guide, but here’s an (imaginary) example of what an app’s template might look like with components:  
在一个大型应用中，有必要将整个应用程序划分为组件，以使开发更易管理。在后续教程中我们将详述组件，不过这里有一个 (假想的) 例子，以展示使用了组件的应用模板是什么样的：

Relation to Custom Elements 与自定义元素的关系

You may have noticed that Vue components are very similar to Custom Elements, which are part of the Web Components Spec. That’s because Vue’s component syntax is loosely modeled after the spec. For example, Vue components implement the Slot API and the is special attribute. However, there are a few key differences:

你可能已经注意到 Vue 组件非常类似于自定义元素——它是 Web 组件规范的一部分，这是因为 Vue 的组件语法部分参考了该规范。例如 Vue 组件实现了 Slot API 与 is 特性。但是，还是有几个关键差别：

The Web Components Spec has been finalized, but is not natively implemented in every browser. Safari 10.1+, Chrome 54+ and Firefox 63+ natively support web components. In comparison, Vue components don’t require any polyfills and work consistently in all supported browsers (IE9 and above). When needed, Vue components can also be wrapped inside a native custom element.

Web Components 规范已经完成并通过，但未被所有浏览器原生实现。目前 Safari 10.1+、Chrome 54+ 和 Firefox 63+ 原生支持 Web Components。相比之下，Vue 组件不需要任何 polyfill，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 组件也可以包装于原生自定义元素之内。

Vue components provide important features that are not available in plain custom elements, most notably cross-component data flow, custom event communication and build tool integrations.

Vue 组件提供了纯自定义元素所不具备的一些重要功能，最突出的是跨组件数据流、自定义事件通信以及构建工具集成。

Although Vue doesn’t use custom elements internally, it has great interoperability when it comes to consuming or distributing as custom elements. Vue CLI also supports building Vue components that register themselves as native custom elements.

虽然 Vue 内部没有使用自定义元素，不过在应用使用自定义元素、或以自定义元素形式发布时，依然有很好的互操作性。Vue CLI 也支持将 Vue 组件构建成为原生的自定义元素。

Ready for More? 准备好了吗？

We’ve briefly introduced the most basic features of Vue.js core - the rest of this guide will cover them and other advanced features with much finer details, so make sure to read through it all!

我们刚才简单介绍了 Vue 核心最基本的功能——本教程的其余部分将更加详细地涵盖这些功能以及其它高级功能，所以请务必读完整个教程！

## The Vue Instance Vue 实例

Creating a Vue Instance 创建一个 Vue 实例

Every Vue application starts by creating a new Vue instance with the Vue function:

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：

Although not strictly associated with the MVVM pattern, Vue’s design was partly inspired by it. As a convention, we often use the variable vm (short for ViewModel) to refer to our Vue instance.

虽然没有完全遵循 MVVM 模型，但是 Vue 的设计也受到了它的启发。因此在文档中经常会使用 vm (ViewModel 的缩写) 这个变量名表示 Vue 实例。

When you create a Vue instance, you pass in an options object. The majority of this guide describes how you can use these options to create your desired behavior. For reference, you can also browse the full list of options in the API reference.

当创建一个 Vue 实例时，你可以传入一个选项对象。这篇教程主要描述的就是如何使用这些选项来创建你想要的行为。作为参考，你也可以在 API 文档 中浏览完整的选项列表。

A Vue application consists of a root Vue instance created with new Vue, optionally organized into a tree of nested, reusable components. For example, a todo app’s component tree might look like this:

一个 Vue 应用由一个通过 new Vue 创建的根 Vue 实例，以及可选的嵌套的、可复用的组件树组成。举个例子，一个 todo 应用的组件树可以是这样的：

We’ll talk about the component system in detail later. For now, just know that all Vue components are also Vue instances, and so accept the same options object (except for a few root-specific options).

我们会在稍后的组件系统章节具体展开。不过现在，你只需要明白所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象 (一些根实例特有的选项除外)。

Data and Methods 数据与方法

When a Vue instance is created, it adds all the properties found in its data object to Vue’s reactivity system. When the values of those properties change, the view will “react”, updating to match the new values.

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。

When this data changes, the view will re-render. It should be noted that properties in data are only reactive if they existed when the instance was created. That means if you add a new property, like:

当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时 data 中存在的属性才是响应式的。也就是说如果你添加一个新的属性，比如：

Then changes to b will not trigger any view updates. If you know you’ll need a property later, but it starts out empty or non-existent, you’ll need to set some initial value. For example:

那么对 b 的改动将不会触发任何视图的更新。如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。比如：

The only exception to this being the use of Object.freeze(), which prevents existing properties from being changed, which also means the reactivity system can’t track changes.

这里唯一的例外是使用 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。

In addition to data properties, Vue instances expose a number of useful instance properties and methods. These are prefixed with $ to differentiate them from user-defined properties. For example:

除了数据属性，Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来。例如：

In the future, you can consult the API reference for a full list of instance properties and methods.

以后你可以在 API 参考中查阅到完整的实例属性和方法的列表。

Instance Lifecycle Hooks 实例生命周期钩子

Each Vue instance goes through a series of initialization steps when it’s created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。

For example, the created hook can be used to run code after an instance is created:

比如 created 钩子可以用来在一个实例被创建之后执行代码：

There are also other hooks which will be called at different stages of the instance’s lifecycle, such as mounted, updated, and destroyed. All lifecycle hooks are called with their this context pointing to the Vue instance invoking it.

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 mounted、updated和 destroyed。生命周期钩子的 this 上下文指向调用它的 Vue 实例。

Don’t use arrow functions on an options property or callback, such as created: () => console.log(this.a) or vm.$watch('a', newValue => this.myMethod()). Since an arrow function doesn’t have a this, this will be treated as any other variable and lexically looked up through parent scopes until found, often resulting in errors such as Uncaught TypeError: Cannot read property of undefined or Uncaught TypeError: this.myMethod is not a function.

不要在选项属性或回调上使用箭头函数，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数并没有 this，this 会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 Uncaught TypeError: Cannot read property of undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误。

**Lifecycle Diagram 生命周期图示**

Below is a diagram for the instance lifecycle. You don’t need to fully understand everything going on right now, but as you learn and build more, it will be a useful reference.

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。

## Template Syntax模板语法

Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance’s data. All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.

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。

Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

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。

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also directly write render functions instead of templates, with optional JSX support.

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，直接写渲染 (render) 函数，使用可选的 JSX 语法。

Interpolations 插值

Text 文本

The most basic form of data binding is text interpolation using the “Mustache” syntax (double curly braces):

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

The mustache tag will be replaced with the value of the msg property on the corresponding data object. It will also be updated whenever the data object’s msg property changes.

Mustache 标签将会被替代为对应数据对象上 msg 属性的值。无论何时，绑定的数据对象上 msg 属性发生了改变，插值处的内容都会更新。

You can also perform one-time interpolations that do not update on data change by using the v-once directive, but keep in mind this will also affect any other bindings on the same node:

通过使用 v-once 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

Raw HTML 原始 HTML

The double mustaches interprets the data as plain text, not HTML. In order to output real HTML, you will need to use the v-html directive:

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 v-html 指令：

The contents of the span will be replaced with the value of the rawHtml property, interpreted as plain HTML - data bindings are ignored. Note that you cannot use v-html to compose template partials, because Vue is not a string-based templating engine. Instead, components are preferred as the fundamental unit for UI reuse and composition.

这个 span 的内容将会被替换成为属性值 rawHtml，直接作为 HTML——会忽略解析属性值中的数据绑定。注意，你不能使用 v-html 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to XSS vulnerabilities. Only use HTML interpolation on trusted content and neveron user-provided content.

你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致 XSS 攻击。请只对可信内容使用 HTML 插值，绝不要对用户提供的内容使用插值。

Attributes 特性

Mustaches cannot be used inside HTML attributes. Instead, use a v-bind directive:

Mustache 语法不能作用在 HTML 特性上，遇到这种情况应该使用 v-bind 指令：

In the case of boolean attributes, where their mere existence implies true, v-bind works a little differently. In this example:

对于布尔特性 (它们只要存在就意味着值为 true)，v-bind 工作起来略有不同，在这个例子中：

If isButtonDisabled has the value of null, undefined, or false, the disabled attribute will not even be included in the rendered <button> element.

如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled 特性甚至不会被包含在渲染出来的 <button> 元素中。

Using JavaScript Expressions 使用 JavaScript 表达式

So far we’ve only been binding to simple property keys in our templates. But Vue.js actually supports the full power of JavaScript expressions inside all data bindings:

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

These expressions will be evaluated as JavaScript in the data scope of the owner Vue instance. One restriction is that each binding can only contain one single expression, so the following will NOT work:

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

Template expressions are sandboxed and only have access to a whitelist of globals such as Math and Date. You should not attempt to access user defined globals in template expressions.

模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 Math 和 Date 。你不应该在模板表达式中试图访问用户定义的全局变量。

Directives 指令

Directives are special attributes with the v- prefix. Directive attribute values are expected to be a single JavaScript expression (with the exception of v-for, which will be discussed later). A directive’s job is to reactively apply side effects to the DOM when the value of its expression changes. Let’s review the example we saw in the introduction:

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。回顾我们在介绍中看到的例子：

Here, the v-if directive would remove/insert the <p> element based on the truthiness of the value of the expression seen.

这里，v-if 指令将根据表达式 seen 的值的真假来插入/移除 <p> 元素。

Arguments 参数

Some directives can take an “argument”, denoted by a colon after the directive name. For example, the v-bind directive is used to reactively update an HTML attribute:

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，v-bind 指令可以用于响应式地更新 HTML 特性：

Here href is the argument, which tells the v-bind directive to bind the element’s href attribute to the value of the expression url.

在这里 href 是参数，告知 v-bind 指令将该元素的 href 特性与表达式 url 的值绑定。

Another example is the v-on directive, which listens to DOM events:

另一个例子是 v-on 指令，它用于监听 DOM 事件：

Here the argument is the event name to listen to. We will talk about event handling in more detail too.

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

Dynamic Arguments 动态参数

New in 2.6.0+ 2.6.0 新增

Starting in version 2.6.0, it is also possible to use a JavaScript expression in a directive argument by wrapping it with square brackets:

从 2.6.0 开始，可以用方括号括起来的 JavaScript 表达式作为一个指令的参数：

Here attributeName will be dynamically evaluated as a JavaScript expression, and its evaluated value will be used as the final value for the argument. For example, if your Vue instance has a data property, attributeName, whose value is "href", then this binding will be equivalent to v-bind:href.

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。

Similarly, you can use dynamic arguments to bind a handler to a dynamic event name:

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：

Similarly, when eventName‘s value is "focus", for example, v-on:[eventName] will be equivalent to v-on:focus.

同样地，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。

Dynamic Argument Value Constraints 对动态参数的值的约束

Dynamic arguments are expected to evaluate to a string, with the exception of null. The special value null can be used to explicitly remove the binding. Any other non-string value will trigger a warning.

动态参数预期会求出一个字符串，异常情况下值为 null。这个特殊的 null 值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

Dynamic Argument Expression Constraints 对动态参数表达式的约束

Dynamic argument expressions have some syntax constraints because certain characters are invalid inside HTML attribute names, such as spaces and quotes. You also need to avoid uppercase keys when using in-DOM templates.

动态参数表达式有一些语法约束，因为某些字符，例如空格和引号，放在 HTML 特性名里是无效的。同样，在 DOM 中使用模板时你需要回避大写键名。

For example, the following is invalid:

例如，下面的代码是无效的：

The workaround is to either use expressions without spaces or quotes, or replace the complex expression with a computed property.

变通的办法是使用没有空格或引号的表达式，或用计算属性替代这种复杂表达式。

In addition, if you are using in-DOM templates (templates directly written in an HTML file), you have to be aware that browsers will coerce attribute names into lowercase:

另外，如果你在 DOM 中使用模板 (直接在一个 HTML 文件里撰写模板)，需要留意浏览器会把特性名全部强制转为小写：

Modifiers 修饰符

Modifiers are special postfixes denoted by a dot, which indicate that a directive should be bound in some special way. For example, the .prevent modifier tells the v-on directive to call event.preventDefault() on the triggered event:

修饰符 (modifier) 是以半角句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()：

You’ll see other examples of modifiers later, for v-on and for v-model, when we explore those features.

在接下来对 v-on 和 v-for 等功能的探索中，你会看到修饰符的其它例子。

Shorthands 缩写

The v- prefix serves as a visual cue for identifying Vue-specific attributes in your templates. This is useful when you are using Vue.js to apply dynamic behavior to some existing markup, but can feel verbose for some frequently used directives. At the same time, the need for the v- prefix becomes less important when you are building a SPA, where Vue manages every template. Therefore, Vue provides special shorthands for two of the most often used directives, v-bind and v-on:

v- 前缀作为一种视觉提示，用来识别模板中 Vue 特定的特性。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时，v- 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue 管理所有模板的单页面应用程序 (SPA - single page application) 时，v- 前缀也变得没那么重要了。因此，Vue 为 v-bind 和 v-on 这两个最常用的指令，提供了特定简写：

They may look a bit different from normal HTML, but : and @ are valid characters for attribute names and all Vue-supported browsers can parse it correctly. In addition, they do not appear in the final rendered markup. The shorthand syntax is totally optional, but you will likely appreciate it when you learn more about its usage later.

它们看起来可能与普通的 HTML 略有不同，但 : 与 @ 对于特性名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

## Computed Properties and Watchers 计算属性和侦听器

Computed Properties 计算属性

In-template expressions are very convenient, but they are meant for simple operations. Putting too much logic in your templates can make them bloated and hard to maintain. For example:

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。例如：

At this point, the template is no longer simple and declarative. You have to look at it for a second before realizing that it displays message in reverse. The problem is made worse when you want to include the reversed message in your template more than once.

在这个地方，模板不再是简单的声明式逻辑。你必须看一段时间才能意识到，这里是想要显示变量 message 的翻转字符串。当你想要在模板中多次引用此处的翻转字符串时，就会更加难以处理。

That’s why for any complex logic, you should use a computed property.

所以，对于任何复杂逻辑，你都应当使用计算属性。

Basic Example 基础例子

Here we have declared a computed property reversedMessage. The function we provided will be used as the getter function for the property vm.reversedMessage:

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

You can open the console and play with the example vm yourself. The value of vm.reversedMessageis always dependent on the value of vm.message.

你可以打开浏览器的控制台，自行修改例子中的 vm。vm.reversedMessage 的值始终取决于 vm.message 的值。

You can data-bind to computed properties in templates just like a normal property. Vue is aware that vm.reversedMessage depends on vm.message, so it will update any bindings that depend on vm.reversedMessage when vm.message changes. And the best part is that we’ve created this dependency relationship declaratively: the computed getter function has no side effects, which makes it easier to test and understand.

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.reversedMessage依赖于 vm.message，因此当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新。而且最妙的是我们已经以声明的方式创建了这种依赖关系：计算属性的 getter 函数是没有副作用 (side effect) 的，这使它更易于测试和理解。

Computed Caching vs Methods 算属性缓存 vs 方法

You may have noticed we can achieve the same result by invoking a method in the expression:

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

Instead of a computed property, we can define the same function as a method instead. For the end result, the two approaches are indeed exactly the same. However, the difference is that computed properties are cached based on their reactive dependencies. A computed property will only re-evaluate when some of its reactive dependencies have changed. This means as long as message has not changed, multiple access to the reversedMessage computed property will immediately return the previously computed result without having to run the function again.

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是基于它们的响应式依赖进行缓存的。只在相关响应式依赖发生改变时它们才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

This also means the following computed property will never update, because Date.now() is not a reactive dependency:

这也同样意味着下面的计算属性将不再更新，因为 Date.now() 不是响应式依赖：

In comparison, a method invocation will always run the function whenever a re-render happens.

相比之下，每当触发重新渲染时，调用方法将总会再次执行函数。

Why do we need caching? Imagine we have an expensive computed property A, which requires looping through a huge Array and doing a lot of computations. Then we may have other computed properties that in turn depend on A. Without caching, we would be executing A’s getter many more times than necessary! In cases where you do not want caching, use a method instead.

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 A，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 A 。如果没有缓存，我们将不可避免的多次执行 A 的 getter！如果你不希望有缓存，请用方法来替代。

Computed vs Watched Property 计算属性 vs 侦听属性

Vue does provide a more generic way to observe and react to data changes on a Vue instance: watch properties. When you have some data that needs to change based on some other data, it is tempting to overuse watch - especially if you are coming from an AngularJS background. However, it is often a better idea to use a computed property rather than an imperative watch callback. Consider this example:

Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：侦听属性。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用计算属性而不是命令式的 watch 回调。细想一下这个例子：

The above code is imperative and repetitive. Compare it with a computed property version:

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

Much better, isn’t it?

好得多了，不是吗？

Computed Setter 计算属性的 setter

Computed properties are by default getter-only, but you can also provide a setter when you need it:

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

Now when you run vm.fullName = 'John Doe', the setter will be invoked and vm.firstName and vm.lastName will be updated accordingly.

现在再运行 vm.fullName = 'John Doe' 时，setter 会被调用，vm.firstName 和 vm.lastName 也会相应地被更新。

Watchers侦听器

While computed properties are more appropriate in most cases, there are times when a custom watcher is necessary. That’s why Vue provides a more generic way to react to data changes through the watch option. This is most useful when you want to perform asynchronous or expensive operations in response to changing data.

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 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。

For example: 例如：

In this case, using the watch option allows us to perform an asynchronous operation (accessing an API), limit how often we perform that operation, and set intermediary states until we get a final answer. None of that would be possible with a computed property.

在这个示例中，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

In addition to the watch option, you can also use the imperative vm.$watch API.

除了 watch 选项之外，您还可以使用命令式的 vm.$watch API。

## Class and Style BindingsClass 与 Style 绑定

A common need for data binding is manipulating an element’s class list and its inline styles. Since they are both attributes, we can use v-bind to handle them: we only need to calculate a final string with our expressions. However, meddling with string concatenation is annoying and error-prone. For this reason, Vue provides special enhancements when v-bind is used with class and style. In addition to strings, the expressions can also evaluate to objects or arrays.

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 v-bind 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 v-bind 用于 class 和 style 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

Binding HTML Classes绑定 HTML Class

Object Syntax对象语法

We can pass an object to v-bind:class to dynamically toggle classes:

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

The above syntax means the presence of the active class will be determined by the truthiness of the data property isActive.

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。

You can have multiple classes toggled by having more fields in the object. In addition, the v-bind:class directive can also co-exist with the plain class attribute. So given the following template:

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。当有如下模板:

And the following data: 和如下 data：

It will render: 结果渲染为：

When isActive or hasError changes, the class list will be updated accordingly. For example, if hasError becomes true, the class list will become "static active text-danger".

当 isActive 或者 hasError 变化时，class 列表将相应地更新。例如，如果 hasError 的值为 true，class 列表将变为 "static active text-danger"。

The bound object doesn’t have to be inline:

绑定的数据对象不必内联定义在模板里：

This will render the same result. We can also bind to a computed property that returns an object. This is a common and powerful pattern:

渲染的结果和上面一样。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：

Array Syntax数组语法

We can pass an array to v-bind:class to apply a list of classes:

我们可以把一个数组传给 v-bind:class，以应用一个 class 列表：

Which will render: 渲染为：

If you would like to also toggle a class in the list conditionally, you can do it with a ternary expression:

如果你也想根据条件切换列表中的 class，可以用三元表达式：

This will always apply errorClass, but will only apply activeClass when isActive is truthy.

这样写将始终添加 errorClass，但是只有在 isActive 是 truthy[1] 时才添加 activeClass。

However, this can be a bit verbose if you have multiple conditional classes. That’s why it’s also possible to use the object syntax inside array syntax:

不过，当有多个条件 class 时这样写有些繁琐。所以在数组语法中也可以使用对象语法：

With Components用在组件上

This section assumes knowledge of Vue Components. Feel free to skip it and come back later.

这个章节假设你已经对 Vue 组件有一定的了解。当然你也可以先跳过这里，稍后再回过头来看。

When you use the class attribute on a custom component, those classes will be added to the component’s root element. Existing classes on this element will not be overwritten.

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。

For example, if you declare this component:

例如，如果你声明了这个组件：

Then add some classes when using it:

然后在使用它的时候添加一些 class：

The rendered HTML will be:

HTML 将被渲染为:

The same is true for class bindings:

对于带数据绑定 class 也同样适用：

When isActive is truthy, the rendered HTML will be:

当 isActive 为 truthy[1] 时，HTML 将被渲染成为：

Binding Inline Styles绑定内联样式

Object Syntax对象语法

The object syntax for v-bind:style is pretty straightforward - it looks almost like CSS, except it’s a JavaScript object. You can use either camelCase or kebab-case (use quotes with kebab-case) for the CSS property names:

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，记得用引号括起来) 来命名：

It is often a good idea to bind to a style object directly so that the template is cleaner:

直接绑定到一个样式对象通常更好，这会让模板更清晰：

Again, the object syntax is often used in conjunction with computed properties that return objects.

同样的，对象语法常常结合返回对象的计算属性使用。

Array Syntax数组语法

The array syntax for v-bind:style allows you to apply multiple style objects to the same element:

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：

Auto-prefixing自动添加前缀

When you use a CSS property that requires vendor prefixes in v-bind:style, for example transform, Vue will automatically detect and add appropriate prefixes to the applied styles.

当 v-bind:style 使用需要添加浏览器引擎前缀的 CSS 属性时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

Multiple Values多重值

Starting in 2.3.0+ you can provide an array of multiple (prefixed) values to a style property, for example:

从 2.3.0 起你可以为 style 绑定中的属性提供一个包含多个值的数组，常用于提供多个带前缀的值，例如：

This will only render the last value in the array which the browser supports. In this example, it will render display: flex for browsers that support the unprefixed version of flexbox.

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 display: flex。

## Conditional Rendering条件渲染

v-if

The directive v-if is used to conditionally render a block. The block will only be rendered if the directive’s expression returns a truthy value.

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。

It is also possible to add an “else block” with v-else:

也可以用 v-else 添加一个“else 块”：

Conditional Groups with v-if on <template>

在 <template> 元素上使用 v-if 条件渲染分组

Because v-if is a directive, it has to be attached to a single element. But what if we want to toggle more than one element? In this case we can use v-if on a <template> element, which serves as an invisible wrapper. The final rendered result will not include the <template> element.

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。

You can use the v-else directive to indicate an “else block” for v-if:

你可以使用 v-else 指令来表示 v-if 的“else 块”：

A v-else element must immediately follow a v-if or a v-else-if element - otherwise it will not be recognized.

v-else 元素必须紧跟在带 v-if 或者 v-else-if 的元素的后面，否则它将不会被识别。

The v-else-if, as the name suggests, serves as an “else if block” for v-if. It can also be chained multiple times:

v-else-if，顾名思义，充当 v-if 的“else-if 块”，可以连续使用：

Similar to v-else, a v-else-if element must immediately follow a v-if or a v-else-ifelement.

类似于 v-else，v-else-if 也必须紧跟在带 v-if 或者 v-else-if 的元素之后。

Controlling Reusable Elements with key

用 key 管理可复用的元素

Vue tries to render elements as efficiently as possible, often re-using them instead of rendering from scratch. Beyond helping make Vue very fast, this can have some useful advantages. For example, if you allow users to toggle between multiple login types:

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：

Then switching the loginType in the code above will not erase what the user has already entered. Since both templates use the same elements, the <input> is not replaced - just its placeholder.

那么在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，<input> 不会被替换掉——仅仅是替换了它的 placeholder。

Check it out for yourself by entering some text in the input, then pressing the toggle button:

自己动手试一试，在输入框中输入一些文本，然后按下切换按钮：

This isn’t always desirable though, so Vue offers a way for you to say, “These two elements are completely separate - don’t re-use them.” Add a key attribute with unique values:

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：

Now those inputs will be rendered from scratch each time you toggle. See for yourself:

现在，每次切换时，输入框都将被重新渲染。请看：

Note that the <label> elements are still efficiently re-used, because they don’t have key attributes.

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。

Another option for conditionally displaying an element is the v-show directive. The usage is largely the same:

另一个用于根据条件展示元素的选项是 v-show 指令。用法大致一样：

The difference is that an element with v-show will always be rendered and remain in the DOM; v-show only toggles the display CSS property of the element.

不同的是带有 v-show 的元素始终会被渲染并保留在 DOM 中。v-show 只是简单地切换元素的 CSS 属性 display。

Note that v-show doesn’t support the <template> element, nor does it work with v-else.

注意，v-show 不支持 <template> 元素，也不支持 v-else。

v-if is “real” conditional rendering because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.

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 is also lazy: if the condition is false on initial render, it will not do anything - the conditional block won’t be rendered until the condition becomes true for the first time.

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。

In comparison, v-show is much simpler - the element is always rendered regardless of initial condition, with CSS-based toggling.

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。

Generally speaking, v-if has higher toggle costs while v-show has higher initial render costs. So prefer v-show if you need to toggle something very often, and prefer v-if if the condition is unlikely to change at runtime.

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。

v-if with v-for v-if 与 v-for 一起使用

Using v-if and v-for together is not recommended. See the style guide for further information.

不推荐同时使用 v-if 和 v-for。请查阅风格指南以获取更多信息。

When used together with v-if, v-for has a higher priority than v-if. See the list rendering guide for details.

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。请查阅列表渲染指南 以获取详细信息。

## List Rendering列表渲染

Mapping an Array to Elements with v-for

用 v-for 把一个数组对应为一组元素

We can use the v-for directive to render a list of items based on an array. The v-for directive requires a special syntax in the form of item in items, where items is the source data array and item is an alias for the array element being iterated on:

我们用 v-for 指令根据一组数组的选项列表进行渲染。v-for 指令需要使用 item in items 形式的特殊语法，items 是源数据数组并且 item 是数组元素迭代的别名。

Inside v-for blocks we have full access to parent scope properties. v-for also supports an optional second argument for the index of the current item.

在 v-for 块中，我们拥有对父作用域属性的完全访问权限。v-for 还支持一个可选的第二个参数为当前项的索引。

You can also use of as the delimiter instead of in, so that it is closer to JavaScript’s syntax for iterators:

你也可以用 of 替代 in 作为分隔符，因为它是最接近 JavaScript 迭代器的语法：

v-for with an Object 一个对象的 v-for

You can also use v-for to iterate through the properties of an object.

你也可以用 v-for 通过一个对象的属性来迭代。

You can also provide a second argument for the property’s name (a.k.a. key):

你也可以提供第二个的参数为 property 名称 (也就是键名)：

And another for the index:

第三个参数为索引：

When iterating over an object, the order is based on the enumeration order of Object.keys(), which is not guaranteed to be consistent across JavaScript engine implementations.

在遍历对象时，是按 Object.keys() 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的。

Maintaining State 维护状态

When Vue is updating a list of elements rendered with v-for, by default it uses an “in-place patch” strategy. If the order of the data items has changed, instead of moving the DOM elements to match the order of the items, Vue will patch each element in-place and make sure it reflects what should be rendered at that particular index. This is similar to the behavior of track-by="$index" in Vue 1.x.

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。这个类似 Vue 1.x 的 track-by="$index" 。

This default mode is efficient, but only suitable when your list render output does not rely on child component state or temporary DOM state (e.g. form input values).

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出。

To give Vue a hint so that it can track each node’s identity, and thus reuse and reorder existing elements, you need to provide a unique key attribute for each item:

为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性：

It is recommended to provide a key attribute with v-for whenever possible, unless the iterated DOM content is simple, or you are intentionally relying on the default behavior for performance gains.

建议尽可能在使用 v-for 时提供 key attribute，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。

Since it’s a generic mechanism for Vue to identify nodes, the key also has other uses that are not specifically tied to v-for, as we will see later in the guide.

因为它是 Vue 识别节点的一个通用机制，key 并不与 v-for 特别关联，key 还具有其他用途，我们将在后面的指南中看到其他用途。

Don’t use non-primitive values like objects and arrays as v-for keys. Use string or numeric values instead.

不要使用对象或数组之类的非原始类型值作为 v-for 的 key。用字符串或数类型的值取而代之。

For detailed usage of the key attribute, please see the key API documentation.

更多 key attribute 的细节用法请移步至 key 的 API 文档。

Array Change Detection 数组更新检测

Mutation Methods 变异方法

Vue wraps an observed array’s mutation methods so they will also trigger view updates. The wrapped methods are:

Vue 包含一组观察数组的变异方法，所以它们也将会触发视图更新。这些方法如下：

You can open the console and play with the previous examples’ items array by calling their mutation methods. For example: example1.items.push({ message: 'Baz' }).

你打开控制台，然后用前面例子的 items 数组调用变异方法：example1.items.push({ message: 'Baz' }) 。

Replacing an Array 替换数组

Mutation methods, as the name suggests, mutate the original array they are called on. In comparison, there are also non-mutating methods, e.g. filter(), concat() and slice(), which do not mutate the original array but always return a new array. When working with non-mutating methods, you can replace the old array with the new one:

变异方法 (mutation method)，顾名思义，会改变被这些方法调用的原始数组。相比之下，也有非变异 (non-mutating method) 方法，例如：filter(), concat() 和 slice() 。这些不会改变原始数组，但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：

You might think this will cause Vue to throw away the existing DOM and re-render the entire list - luckily, that is not the case. Vue implements some smart heuristics to maximize DOM element reuse, so replacing an array with another array containing overlapping objects is a very efficient operation.

你可能认为这将导致 Vue 丢弃现有 DOM 并重新渲染整个列表。幸运的是，事实并非如此。Vue 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的、启发式的方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

Caveats 注意事项

Due to limitations in JavaScript, Vue cannot detect the following changes to an array:

由于 JavaScript 的限制，Vue 不能检测以下变动的数组：

When you directly set an item with the index, e.g. vm.items[indexOfItem] = newValue

当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue

When you modify the length of the array, e.g. vm.items.length = newLength

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

For example: 举个例子：

To overcome caveat 1, both of the following will accomplish the same as vm.items[indexOfItem] = newValue, but will also trigger state updates in the reactivity system:

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，同时也将触发状态更新：

You can also use the vm.$set instance method, which is an alias for the global Vue.set:

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：

To deal with caveat 2, you can use splice:

为了解决第二类问题，你可以使用 splice：

Object Change Detection Caveats 对象更改检测注意事项

Again due to limitations of modern JavaScript, Vue cannot detect property addition or deletion. For example:

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

Vue does not allow dynamically adding new root-level reactive properties to an already created instance. However, it’s possible to add reactive properties to a nested object using the Vue.set(object, propertyName, value) method. For example, given:

对于已经创建的实例，Vue 不能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：

You could add a new age property to the nested userProfile object with:

你可以添加一个新的 age 属性到嵌套的 userProfile 对象：

You can also use the vm.$set instance method, which is an alias for the global Vue.set:

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

Sometimes you may want to assign a number of new properties to an existing object, for example using Object.assign() or \_.extend(). In such cases, you should create a fresh object with properties from both objects. So instead of:

有时你可能需要为已有对象赋予多个新属性，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：

You would add new, reactive properties with:

你应该这样做：

Displaying Filtered/Sorted Results 显示过滤/排序结果

Sometimes we want to display a filtered or sorted version of an array without actually mutating or resetting the original data. In this case, you can create a computed property that returns the filtered or sorted array.

有时，我们想要显示一个数组的过滤或排序副本，而不实际改变或重置原始数据。在这种情况下，可以创建返回过滤或排序数组的计算属性。

For example: 例如：

In situations where computed properties are not feasible (e.g. inside nested v-for loops), you can use a method:

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 with a Range 一段取值范围的 v-for

v-for can also take an integer. In this case it will repeat the template that many times.

v-for 也可以取整数。在这种情况下，它将重复多次模板。

Similar to template v-if, you can also use a <template> tag with v-for to render a block of multiple elements. For example:

类似于 v-if，你也可以利用带有 v-for 的 <template> 渲染多个元素。比如：

Note that it’s not recommended to use v-if and v-for together. Refer to style guide for details.

注意我们不推荐同时使用 v-if 和 v-for。更多细节可查阅风格指南。

When they exist on the same node, v-for has a higher priority than v-if. That means the v-ifwill be run on each iteration of the loop separately. This can be useful when you want to render nodes for only some items, like below:

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，如下：

The above only renders the todos that are not complete.

上面的代码只传递了未完成的 todos。

If instead, your intent is to conditionally skip execution of the loop, you can place the v-if on a wrapper element (or <template>). For example:

而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>)上。如：

v-for with a Component 一个组件的 v-for

This section assumes knowledge of Components. Feel free to skip it and come back later.

了解组件相关知识，查看 组件。完全可以先跳过它，以后再回来查看。

You can directly use v-for on a custom component, like any normal element:

在自定义组件里，你可以像任何普通元素一样用 v-for 。

In 2.2.0+, when using v-for with a component, a key is now required.

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

However, this won’t automatically pass any data to the component, because components have isolated scopes of their own. In order to pass the iterated data into the component, we should also use props:

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 ：

The reason for not automatically injecting item into the component is because that makes the component tightly coupled to how v-for works. Being explicit about where its data comes from makes the component reusable in other situations.

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。

Here’s a complete example of a simple todo list:

下面是一个简单的 todo list 的完整例子：

Note the is="todo-item" attribute. This is necessary in DOM templates, because only an <li> element is valid inside a <ul>. It does the same thing as <todo-item>, but works around a potential browser parsing error. See DOM Template Parsing Caveats to learn more.

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

## Event Handling事件处理

Listening to Events 监听事件

We can use the v-on directive to listen to DOM events and run some JavaScript when they’re triggered.

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

For example: 示例：

Method Event Handlers 事件处理方法

The logic for many event handlers will be more complex though, so keeping your JavaScript in the value of the v-on attribute isn’t feasible. That’s why v-on can also accept the name of a method you’d like to call.

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。

For example:示例：

Methods in Inline Handlers 内联处理器中的方法

Instead of binding directly to a method name, we can also use methods in an inline JavaScript statement:

除了直接绑定到一个方法，也可以在内联 JavaScript 语句中调用方法：

Sometimes we also need to access the original DOM event in an inline statement handler. You can pass it into a method using the special $event variable:

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：

Event Modifiers 事件修饰符

It is a very common need to call event.preventDefault() or event.stopPropagation() inside event handlers. Although we can do this easily inside methods, it would be better if the methods can be purely about data logic rather than having to deal with DOM event details.

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。

To address this problem, Vue provides event modifiers for v-on. Recall that modifiers are directive postfixes denoted by a dot.

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。

Order matters when using modifiers because the relevant code is generated in the same order. Therefore using v-on:click.prevent.self will prevent all clicks while v-on:click.self.prevent will only prevent clicks on the element itself.

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 v-on:click.prevent.self 会阻止所有的点击，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

Unlike the other modifiers, which are exclusive to native DOM events, the .once modifier can also be used on component events. If you haven’t read about components yet, don’t worry about this for now.

不像其它只能对原生的 DOM 事件起作用的修饰符，.once 修饰符还能被用到自定义的组件事件上。如果你还没有阅读关于组件的文档，现在大可不必担心。

Vue also offers the .passive modifier, corresponding

to addEventListener‘s passive option.

Vue 还对应 addEventListener 中的 passive 选项提供了 .passive 修饰符。

The .passive modifier is especially useful for improving performance on mobile devices.

这个 .passive 修饰符尤其能够提升移动端的性能。

Don’t use .passive and .prevent together, because .prevent will be ignored and your browser will probably show you a warning. Remember, .passive communicates to the browser that you don’t want to prevent the event’s default behavior.

不要把 .passive 和 .prevent 一起使用，因为 .prevent 将会被忽略，同时浏览器可能会向你展示一个警告。请记住，.passive 会告诉浏览器你不想阻止事件的默认行为

Key Modifiers 按键修饰符

When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers for v-on when listening for key events:

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：

You can directly use any valid key names exposed via KeyboardEvent.key as modifiers by converting them to kebab-case.

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。

In the above example, the handler will only be called if $event.key is equal to 'PageDown'.

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

Key Codes 按键码

The use of keyCode events is deprecated and may not be supported in new browsers.

Using keyCode attributes is also permitted:

keyCode 的事件用法已经被废弃了并可能不会被最新的浏览器支持。

Vue provides aliases for the most commonly used key codes when necessary for legacy browser support:

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

A few keys (.esc and all arrow keys) have inconsistent key values in IE9, so these built-in aliases should be preferred if you need to support IE9.

有一些按键 (.esc 以及所有的方向键) 在 IE9 中有不同的 key 值, 如果你想支持 IE9，这些内置的别名应该是首选。

You can also define custom key modifier aliases via the global config.keyCodes object:

你还可以通过全局 config.keyCodes 对象自定义按键修饰符别名：

System Modifier Keys 系统修饰键

You can use the following modifiers to trigger mouse or keyboard event listeners only when the corresponding modifier key is pressed:

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

Note: On Macintosh keyboards, meta is the command key (⌘). On Windows keyboards, meta is the Windows key (⊞). On Sun Microsystems keyboards, meta is marked as a solid diamond (◆). On certain keyboards, specifically MIT and Lisp machine keyboards and successors, such as the Knight keyboard, space-cadet keyboard, meta is labeled “META”. On Symbolics keyboards, meta is labeled “META” or “Meta”.

注意：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”。

For example:例如：

Note that modifier keys are different from regular keys and when used with keyup events, they have to be pressed when the event is emitted. In other words, keyup.ctrl will only trigger if you release a key while holding down ctrl. It won’t trigger if you release the ctrl key alone. If you do want such behaviour, use the keyCode for ctrl instead: keyup.17.

请注意修饰键与常规按键不同，在和 keyup 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 ctrl 的情况下释放其它按键，才能触发 keyup.ctrl。而单单释放 ctrl 也不会触发事件。如果你想要这样的行为，请为 ctrl 换用 keyCode：keyup.17。

.exact Modifier .exact 修饰符

The .exact modifier allows control of the exact combination of system modifiers needed to trigger an event.

.exact 修饰符允许你控制由精确的系统修饰符组合触发的事件。

Mouse Button Modifiers 鼠标按钮修饰符

These modifiers restrict the handler to events triggered by a specific mouse button.

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

Why Listeners in HTML? 为什么在 HTML 中监听事件?

You might be concerned that this whole event listening approach violates the good old rules about “separation of concerns”. Rest assured - since all Vue handler functions and expressions are strictly bound to the ViewModel that’s handling the current view, it won’t cause any maintenance difficulty. In fact, there are several benefits in using v-on:

你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 Vue.js 事件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上的困难。实际上，使用 v-on 有几个好处：

It’s easier to locate the handler function implementations within your JS code by skimming the HTML template.

扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。

Since you don’t have to manually attach event listeners in JS, your ViewModel code can be pure logic and DOM-free. This makes it easier to test.

因为你无须在 JavaScript 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，更易于测试。

When a ViewModel is destroyed, all event listeners are automatically removed. You don’t need to worry about cleaning it up yourself.

当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

## Form Input Bindings表单输入绑定

Basic Usage 基础用法

You can use the v-model directive to create two-way data bindings on form input, textarea, and select elements. It automatically picks the correct way to update the element based on the input type. Although a bit magical, v-model is essentially syntax sugar for updating data on user input events, plus special care for some edge cases.

你可以用 v-model 指令在表单 <input>、<textarea> 及 <select> 元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇，但 v-model 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

v-model will ignore the initial value, checked or selected attributes found on any form elements. It will always treat the Vue instance data as the source of truth. You should declare the initial value on the JavaScript side, inside the data option of your component.

v-model 会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。

v-model internally uses different properties and emits different events for different input elements:

v-model 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

text and textarea elements use value property and input event;

text 和 textarea 元素使用 value 属性和 input 事件；

checkboxes and radiobuttons use checked property and change event;

checkbox 和 radio 使用 checked 属性和 change 事件；

select fields use value as a prop and change as an event.

select 字段将 value 作为 prop 并将 change 作为事件。

For languages that require an IME (Chinese, Japanese, Korean etc.), you’ll notice that v-model doesn’t get updated during IME composition. If you want to cater for these updates as well, use input event instead.

对于需要使用输入法 (如中文、日文、韩文等) 的语言，你会发现 v-model 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 input 事件。

Interpolation on textareas (<textarea>{{text}}</textarea>) won't work. Use v-modelinstead.

在文本区域插值 (<textarea>{{text}}</textarea>) 并不会生效，应用 v-model 来代替。

Checkbox 复选框

Single checkbox, boolean value:

单个复选框，绑定到布尔值：

Multiple checkboxes, bound to the same Array:

多个复选框，绑定到同一个数组：

If the initial value of your v-model expression does not match any of the options, the <select> element will render in an “unselected” state. On iOS this will cause the user not being able to select the first item because iOS does not fire a change event in this case. It is therefore recommended to provide a disabled option with an empty value, as demonstrated in the example above.

如果 v-model 表达式的初始值未能匹配任何选项，<select> 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 change 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

Value Bindings 值绑定

For radio, checkbox and select options, the v-model binding values are usually static strings (or booleans for checkbox):

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)：

But sometimes we may want to bind the value to a dynamic property on the Vue instance. We can use v-bind to achieve that. In addition, using v-bind allows us to bind the input value to non-string values.

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。

Checkbox 复选框

The true-value and false-value attributes don’t affect the input’s value attribute, because browsers don’t include unchecked boxes in form submissions. To guarantee that one of two values is submitted in a form (e.g. “yes” or “no”), use radio inputs instead.

这里的 true-value 和 false-value 特性并不会影响输入控件的 value 特性，因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交，(比如“yes”或“no”)，请换用单选按钮

Modifiers 修饰符

By default, v-model syncs the input with the data after each input event (with the exception of IME composition as stated above). You can add the lazy modifier to instead sync after change events:

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：

If you want user input to be automatically typecast as a number, you can add the number modifier to your v-model managed inputs:

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：

This is often useful, because even with type="number", the value of HTML input elements always returns a string. If the value cannot be parsed with parseFloat(), then the original value is returned.

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。

If you want whitespace from user input to be trimmed automatically, you can add the trim modifier to your v-model-managed inputs:

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：

v-model with Components 在组件上使用 v-model

If you’re not yet familiar with Vue’s components, you can skip this for now.

如果你还不熟悉 Vue 的组件，可以暂且跳过这里。

HTML’s built-in input types won’t always meet your needs. Fortunately, Vue components allow you to build reusable inputs with completely customized behavior. These inputs even work with v-model! To learn more, read about custom inputs in the Components guide.

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue 的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 v-model 一起使用！要了解更多，请参阅组件指南中的自定义输入组件。

## Components Basics组件基础

Base Example 基本示例

Here’s an example of a Vue component:

这里有一个 Vue 组件的示例：

Components are reusable Vue instances with a name: in this case, <button-counter>. We can use this component as a custom element inside a root Vue instance created with new Vue:

组件是可复用的 Vue 实例，且带有一个名字：在这个例子中是 <button-counter>。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：

Since components are reusable Vue instances, they accept the same options as new Vue, such as data, computed, watch, methods, and lifecycle hooks. The only exceptions are a few root-specific options like el.

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。

Reusing Components 组件的复用

Components can be reused as many times as you want:

你可以将组件进行任意次数的复用：

Notice that when clicking on the buttons, each one maintains its own, separate count. That’s because each time you use a component, a new instance of it is created.

注意当点击按钮时，每个组件都会各自独立维护它的 count。因为你每用一次组件，就会有一个它的新实例被创建。

data Must Be a Function data 必须是一个函数

When we defined the <button-counter> component, you may have noticed that data wasn’t directly provided an object, like this:

当我们定义这个 <button-counter> 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：

Instead, a component’s data option must be a function, so that each instance can maintain an independent copy of the returned data object:

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：

If Vue didn’t have this rule, clicking on one button would affect the data of all other instances, like below:

如果 Vue 没有这条规则，点击一个按钮就可能会像如下代码一样影响到其它所有实例：

Organizing Components 组件的组织

It’s common for an app to be organized into a tree of nested components:

通常一个应用会以一棵嵌套的组件树的形式来组织：

For example, you might have components for a header, sidebar, and content area, each typically containing other components for navigation links, blog posts, etc.

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。

To use these components in templates, they must be registered so that Vue knows about them. There are two types of component registration: global and local. So far, we’ve only registered components globally, using Vue.component:

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：

Globally registered components can be used in the template of any root Vue instance (new Vue) created afterwards – and even inside all subcomponents of that Vue instance’s component tree.

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。

That’s all you need to know about registration for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Component Registration.

到目前为止，关于组件注册你需要了解的就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把组件注册读完。

Passing Data to Child Components with Props 通过 Prop 向子组件传递数据

Earlier, we mentioned creating a component for blog posts. The problem is, that component won’t be useful unless you can pass data to it, such as the title and content of the specific post we want to display. That’s where props come in.

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。

Props are custom attributes you can register on a component. When a value is passed to a prop attribute, it becomes a property on that component instance. To pass a title to our blog post component, we can include it in the list of props this component accepts, using a props option:

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：

A component can have as many props as you’d like and by default, any value can be passed to any prop. In the template above, you’ll see that we can access this value on the component instance, just like with data.

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 data 中的值一样。

Once a prop is registered, you can pass data to it as a custom attribute, like this:

一个 prop 被注册之后，你就可以像这样把数据作为一个自定义特性传递进来：

In a typical app, however, you’ll likely have an array of posts in data:

然而在一个典型的应用中，你可能在 data 里有一个博文的数组：

Then want to render a component for each one:

并想要为每篇博文渲染一个组件：

Above, you’ll see that we can use v-bind to dynamically pass props. This is especially useful when you don’t know the exact content you’re going to render ahead of time, like when fetching posts from an API.

如上所示，你会发现我们可以使用 v-bind 来动态传递 prop。这在你一开始不清楚要渲染的具体内容，比如从一个 API 获取博文列表的时候，是非常有用的。

That’s all you need to know about props for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Props.

到目前为止，关于 prop 你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把 prop 读完。

A Single Root Element 单个根元素

When building out a <blog-post> component, your template will eventually contain more than just the title:

当构建一个 <blog-post> 组件时，你的模板最终会包含的东西远不止一个标题：

At the very least, you’ll want to include the post’s content:

最最起码，你会包含这篇博文的正文：

If you try this in your template however, Vue will show an error, explaining that every component must have a single root element. You can fix this error by wrapping the template in a parent element, such as:

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)。你可以将模板的内容包裹在一个父元素内，来修复这个问题，例如：

As our component grows, it’s likely we’ll not only need the title and content of a post, but also the published date, comments, and more. Defining a prop for each related piece of information could become very annoying:

看起来当组件变得越来越复杂的时候，我们的博文不只需要标题和内容，还需要发布日期、评论等等。为每个相关的信息定义一个 prop 会变得很麻烦：

So this might be a good time to refactor the <blog-post> component to accept a single post prop instead:

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

The above example and some future ones use JavaScript’s template literal to make multi-line templates more readable. These are not supported by Internet Explorer (IE), so if you must support IE and are not transpiling (e.g. with Babel or TypeScript), use newline escapes instead.

上述的这个和一些接下来的示例使用了 JavaScript 的模板字符串来让多行的模板更易读。它们在 IE 下并没有被支持，所以如果你需要在不 (经过 Babel 或 TypeScript 之类的工具) 编译的情况下支持 IE，请使用折行转义字符取而代之。

Now, whenever a new property is added to post objects, it will automatically be available inside <blog-post>.

现在，不论何时为 post 对象添加一个新的属性，它都会自动地在 <blog-post> 内可用。

Listening to Child Components Events 监听子组件事件

As we develop our <blog-post> component, some features may require communicating back up to the parent. For example, we may decide to include an accessibility feature to enlarge the text of blog posts, while leaving the rest of the page its default size:

在我们开发 <blog-post> 组件时，它的一些功能可能要求我们和父级组件进行沟通。例如我们可能会引入一个可访问性的功能来放大博文的字号，同时让页面的其它部分保持默认的字号。

In the parent, we can support this feature by adding a postFontSize data property:

在其父组件中，我们可以通过添加一个 postFontSize 数据属性来支持这个功能：

Which can be used in the template to control the font size of all blog posts:

它可以在模板中用来控制所有博文的字号：

Now let’s add a button to enlarge the text right before the content of every post:

现在我们在每篇博文正文之前添加一个按钮来放大字号：

The problem is, this button doesn’t do anything:

问题是这个按钮不会做任何事：

When we click on the button, we need to communicate to the parent that it should enlarge the text of all posts. Fortunately, Vue instances provide a custom events system to solve this problem. The parent can choose to listen to any event on the child component instance with v-on, just as we would with a native DOM event:

当点击这个按钮时，我们需要告诉父级组件放大所有博文的文本。幸好 Vue 实例提供了一个自定义事件的系统来解决这个问题。父级组件可以像处理 native DOM 事件一样通过 v-on 监听子组件实例的任意事件：

Then the child component can emit an event on itself by calling the built-in $emit method, passing the name of the event:

同时子组件可以通过调用内建的 $emit 方法 并传入事件名称来触发一个事件：

Thanks to the v-on:enlarge-text="postFontSize += 0.1" listener, the parent will receive the event and update postFontSize value.

有了这个 v-on:enlarge-text="postFontSize += 0.1" 监听器，父级组件就会接收该事件并更新 postFontSize 的值。

Emitting a Value With an Event 使用事件抛出一个值

It’s sometimes useful to emit a specific value with an event. For example, we may want the <blog-post> component to be in charge of how much to enlarge the text by. In those cases, we can use $emit‘s 2nd parameter to provide this value:

有的时候用一个事件来抛出一个特定的值是非常有用的。例如我们可能想让 <blog-post> 组件决定它的文本要放大多少。这时可以使用 $emit 的第二个参数来提供这个值：

Then when we listen to the event in the parent, we can access the emitted event’s value with $event:

然后当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：

Or, if the event handler is a method:

或者，如果这个事件处理函数是一个方法：

Then the value will be passed as the first parameter of that method:

那么这个值将会作为第一个参数传入这个方法：

Using v-model on Components 在组件上使用 v-model

Custom events can also be used to create custom inputs that work with v-model. Remember that:

自定义事件也可以用于创建支持 v-model 的自定义输入组件。记住：

does the same thing as: 等价于：

When used on a component, v-model instead does this:

当用在组件上时，v-model 则会这样：

For this to actually work though, the <input> inside the component must:

为了让它正常工作，这个组件内的 <input> 必须：

Bind the value attribute to a value prop

将其 value 特性绑定到一个名叫 value 的 prop 上

On input, emit its own custom input event with the new value

在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

Here’s that in action:

写成代码之后是这样的：

Now v-model should work perfectly with this component:

现在 v-model 就应该可以在这个组件上完美地工作起来了：

That’s all you need to know about custom component events for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Custom Events.

到目前为止，关于组件自定义事件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把自定义事件读完。

Content Distribution with Slots 通过插槽分发内容

Just like with HTML elements, it’s often useful to be able to pass content to a component, like this:

和 HTML 元素一样，我们经常需要向一个组件传递内容，像这样：

Which might render something like:

可能会渲染出这样的东西：

Fortunately, this task is made very simple by Vue’s custom <slot> element:

幸好，Vue 自定义的 <slot> 元素让这变得非常简单：

As you’ll see above, we just add the slot where we want it to go – and that’s it. We’re done!

如你所见，我们只要在需要的地方加入插槽就行了——就这么简单！

That’s all you need to know about slots for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Slots.

到目前为止，关于插槽你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把插槽读完。

Dynamic Components 动态组件

Sometimes, it’s useful to dynamically switch between components, like in a tabbed interface:

有的时候，在不同组件之间进行动态切换是非常有用的，比如在一个多标签的界面里：

The above is made possible by Vue’s <component> element with the is special attribute:

上述内容可以通过 Vue 的 <component> 元素加一个特殊的 is 特性来实现：

In the example above, currentTabComponent can contain either:

在上述示例中，currentTabComponent 可以包括

the name of a registered component, or

已注册组件的名字，或

a component’s options object

一个组件的选项对象

See this fiddle to experiment with the full code, or this version for an example binding to a component’s options object, instead of its registered name.

你可以在这里查阅并体验完整的代码，或在这个版本了解绑定组件选项对象，而不是已注册组件名的示例。

That’s all you need to know about dynamic components for now, but once you’ve finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Dynamic & Async Components.

到目前为止，关于动态组件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把动态和异步组件读完。

DOM Template Parsing Caveats 解析 DOM 模板时的注意事项

Some HTML elements, such as <ul>, <ol>, <table> and <select> have restrictions on what elements can appear inside them, and some elements such as <li>, <tr>, and <option> can only appear inside certain other elements.

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。

This will lead to issues when using components with elements that have such restrictions. For example:

这会导致我们使用这些有约束条件的元素时遇到一些问题。例如：

The custom component <blog-post-row> will be hoisted out as invalid content, causing errors in the eventual rendered output. Fortunately, the is special attribute offers a workaround:

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

It should be noted that this limitation does not apply if you are using string templates from one of the following sources:

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：

String templates (e.g. template: '...') 字符串 (例如：template: '...')

Single-file (.vue) components 单文件组件 (.vue)

That’s all you need to know about DOM template parsing caveats for now – and actually, the end of Vue’s Essentials. Congratulations! There’s still more to learn, but first, we recommend taking a break to play with Vue yourself and build something fun.

到这里，你需要了解的解析 DOM 模板时的注意事项——实际上也是 Vue 的全部必要内容，大概就是这些了。恭喜你！接下来还有很多东西要去学习，不过首先，我们推荐你先休息一下，试用一下 Vue，自己随意做些好玩的东西。

Once you feel comfortable with the knowledge you’ve just digested, we recommend coming back to read the full guide on Dynamic & Async Components, as well as the other pages in the Components In-Depth section of the sidebar.

如果你感觉已经掌握了这些知识，我们推荐你再回来把完整的组件指南，包括侧边栏中组件深入章节的所有页面读完。

# Components In-Depth深入了解组件

## Component Registration组件注册

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

Component Names 组件名

When registering a component, it will always be given a name. For example, in the global registration we’ve seen so far:

在注册一个组件的时候，我们始终需要给它一个名字。比如在全局注册的时候我们已经看到了：

The component’s name is the first argument of Vue.component.

该组件名就是 Vue.component 的第一个参数。

The name you give a component may depend on where you intend to use it. When using a component directly in the DOM (as opposed to in a string template or single-file component), we strongly recommend following the W3C rules for custom tag names (all-lowercase, must contain a hyphen). This helps you avoid conflicts with current and future HTML elements.

你给予组件的名字可能依赖于你打算拿它来做什么。当直接在 DOM 中使用一个组件 (而不是在字符串模板或单文件组件) 的时候，我们强烈推荐遵循 W3C 规范中的自定义组件名 (字母全小写且必须包含一个连字符)。这会帮助你避免和当前以及未来的 HTML 元素相冲突。

You can see other recommendations for component names in the Style Guide.

你可以在风格指南中查阅到关于组件名的其它建议。

Name Casing 组件名大小写

You have two options when defining component names:

定义组件名的方式有两种：

With kebab-case 使用 kebab-case

When defining a component with kebab-case, you must also use kebab-case when referencing its custom element, such as in <my-component-name>.

当使用 kebab-case (短横线分隔命名) 定义一个组件时，你也必须在引用这个自定义元素时使用 kebab-case，例如 <my-component-name>。

With PascalCase使用 PascalCase

When defining a component with PascalCase, you can use either case when referencing its custom element. That means both <my-component-name> and <MyComponentName> are acceptable. Note, however, that only kebab-case names are valid directly in the DOM (i.e. non-string templates).

当使用 PascalCase (首字母大写命名) 定义一个组件时，你在引用这个自定义元素时两种命名法都可以使用。也就是说 <my-component-name> 和 <MyComponentName> 都是可接受的。注意，尽管如此，直接在 DOM (即非字符串的模板) 中使用时只有 kebab-case 是有效的。

Global Registration 全局注册

So far, we’ve only created components using Vue.component:

到目前为止，我们只用过 Vue.component 来创建组件：

These components are globally registered. That means they can be used in the template of any root Vue instance (new Vue) created after registration. For example:

这些组件是全局注册的。也就是说它们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。比如：

This even applies to all subcomponents, meaning all three of these components will also be available inside each other.

在所有子组件中也是如此，也就是说这三个组件在各自内部也都可以相互使用。

Local Registration 局部注册

Global registration often isn’t ideal. For example, if you’re using a build system like Webpack, globally registering all components means that even if you stop using a component, it could still be included in your final build. This unnecessarily increases the amount of JavaScript your users have to download.

全局注册往往是不够理想的。比如，如果你使用一个像 webpack 这样的构建系统，全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

In these cases, you can define your components as plain JavaScript objects:

在这些情况下，你可以通过一个普通的 JavaScript 对象来定义组件：

Then define the components you’d like to use in a components option:

然后在 components 选项中定义你想要使用的组件：

For each property in the components object, the key will be the name of the custom element, while the value will contain the options object for the component.

对于 components 对象中的每个属性来说，其属性名就是自定义元素的名字，其属性值就是这个组件的选项对象。

Note that locally registered components are not also available in subcomponents. For example, if you wanted ComponentA to be available in ComponentB, you’d have to use:

注意局部注册的组件在其子组件中不可用。例如，如果你希望 ComponentA 在 ComponentB 中可用，则你需要这样写：

Or if you’re using ES2015 modules, such as through Babel and Webpack, that might look more like:

或者如果你通过 Babel 和 webpack 使用 ES2015 模块，那么代码看起来更像：

Note that in ES2015+, placing a variable name like ComponentA inside an object is shorthand for ComponentA: ComponentA, meaning the name of the variable is both:

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：

the custom element name to use in the template, and

用在模板中的自定义元素的名称

the name of the variable containing the component options

包含了这个组件选项的变量名

Module Systems 模块系统

If you’re not using a module system with import/require, you can probably skip this section for now. If you are, we have some special instructions and tips just for you.

如果你没有通过 import/require 使用一个模块系统，也许可以暂且跳过这个章节。如果你使用了，那么我们会为你提供一些特殊的使用说明和注意事项。

Local Registration in a Module System 在模块系统中局部注册

If you’re still here, then it’s likely you’re using a module system, such as with Babel and Webpack. In these cases, we recommend creating a components directory, with each component in its own file.

如果你还在阅读，说明你使用了诸如 Babel 和 webpack 的模块系统。在这些情况下，我们推荐创建一个 components 目录，并将每个组件放置在其各自的文件中。

Then you’ll need to import each component you’d like to use, before you locally register it. For example, in a hypothetical ComponentB.js or ComponentB.vue file:

然后你需要在局部注册之前导入每个你想使用的组件。例如，在一个假设的 ComponentB.js 或 ComponentB.vue 文件中：

Now both ComponentA and ComponentC can be used inside ComponentB‘s template.

现在 ComponentA 和 ComponentC 都可以在 ComponentB 的模板中使用了。

Automatic Global Registration of Base Components 基础组件的自动化全局注册

Many of your components will be relatively generic, possibly only wrapping an element like an input or a button. We sometimes refer to these as base components and they tend to be used very frequently across your components.

可能你的许多组件只是包裹了一个输入框或按钮之类的元素，是相对通用的。我们有时候会把它们称为基础组件，它们会在各个组件中被频繁的用到。

The result is that many components may include long lists of base components:

所以会导致很多组件里都会有一个包含基础组件的长列表：

Just to support relatively little markup in a template:

而只是用于模板中的一小部分：

Fortunately, if you’re using Webpack (or Vue CLI 3+, which uses Webpack internally), you can use require.context to globally register only these very common base components. Here’s an example of the code you might use to globally import base components in your app’s entry file (e.g. src/main.js):

幸好如果你使用了 webpack (或在内部使用了 webpack 的 Vue CLI 3+)，那么就可以使用 require.context 只全局注册这些非常通用的基础组件。这里有一份可以让你在应用入口文件 (比如 src/main.js) 中全局导入基础组件的示例代码：

Remember that global registration must take place before the root Vue instance is created (with new Vue). Here’s an example of this pattern in a real project context.

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。这里有一个真实项目情景下的示例。

## Props

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

Prop Casing (camelCase vs kebab-case) Prop 的大小写 (camelCase vs kebab-case)

HTML attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you’re using in-DOM templates, camelCased prop names need to use their kebab-cased (hyphen-delimited) equivalents:

HTML 中的特性名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。这意味着当你使用 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：

Again, if you’re using string templates, this limitation does not apply.

重申一次，如果你使用字符串模板，那么这个限制就不存在了。

Prop Types Prop 类型

So far, we’ve only seen props listed as an array of strings:

到这里，我们只看到了以字符串数组形式列出的 prop：

Usually though, you’ll want every prop to be a specific type of value. In these cases, you can list props as an object, where the properties’ names and values contain the prop names and types, respectively:

但是，通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些属性的名称和值分别是 prop 各自的名称和类型：

This not only documents your component, but will also warn users in the browser’s JavaScript console if they pass the wrong type. You’ll learn much more about type checks and other prop validationsfurther down this page.

这不仅为你的组件提供了文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。你会在这个页面接下来的部分看到类型检查和其它 prop 验证。

Passing Static or Dynamic Props 传递静态或动态 Prop

So far, you’ve seen props passed a static value, like in:

像这样，你已经知道了可以像这样给 prop 传入一个静态的值：

You’ve also seen props assigned dynamically with v-bind, such as in:

你也知道 prop 可以通过 v-bind 动态赋值，例如：

In the two examples above, we happen to pass string values, but any type of value can actually be passed to a prop.

在上述两个示例中，我们传入的值都是字符串类型的，但实际上任何类型的值都可以传给一个 prop。

Passing the Properties of an Object 传入一个对象的所有属性

If you want to pass all the properties of an object as props, you can use v-bind without an argument (v-bind instead of v-bind:prop-name). For example, given a post object:

如果你想要将一个对象的所有属性都作为 prop 传入，你可以使用不带参数的 v-bind(取代 v-bind:prop-name)。例如，对于一个给定的对象 post：

The following template: 下面的模板：

Will be equivalent to：等价于：

One-Way Data Flow 单向数据流

All props form a one-way-down binding between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent’s state, which can make your app’s data flow harder to understand.

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

In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should not attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console.

额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。

There are usually two cases where it’s tempting to mutate a prop:

这里有两种常见的试图改变一个 prop 的情形：

The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards. In this case, it’s best to define a local data property that uses the prop as its initial value:

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：

The prop is passed in as a raw value that needs to be transformed. In this case, it’s best to define a computed property using the prop’s value:

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：

Note that objects and arrays in JavaScript are passed by reference, so if the prop is an array or object, mutating the object or array itself inside the child component will affect parent state.

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 Validation Prop 验证

Components can specify requirements for its props, such as the types you’ve already seen. If a requirement isn’t met, Vue will warn you in the browser’s JavaScript console. This is especially useful when developing a component that’s intended to be used by others.

我们可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

To specify prop validations, you can provide an object with validation requirements to the value of props, instead of an array of strings. For example:

为了定制 prop 的验证方式，你可以为 props 中的值提供一个带有验证需求的对象，而不是一个字符串数组。例如：

When prop validation fails, Vue will produce a console warning (if using the development build).

当 prop 验证失败的时候，(开发环境构建版本的) Vue 将会产生一个控制台的警告。

Note that props are validated before a component instance is created, so instance properties (e.g. data, computed, etc) will not be available inside default or validator functions.

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。

Type Checks 类型检查

The type can be one of the following native constructors:

type 可以是下列原生构造函数中的一个：

In addition, type can also be a custom constructor function and the assertion will be made with an instanceof check. For example, given the following constructor function exists:

额外的，type 还可以是一个自定义的构造函数，并且通过 instanceof 来进行检查确认。例如，给定下列现成的构造函数：

You could use: 你可以使用：

to validate that the value of the author prop was created with new Person.

来验证 author prop 的值是否是通过 new Person 创建的。

Non-Prop Attributes 非 Prop 的特性

A non-prop attribute is an attribute that is passed to a component, but does not have a corresponding prop defined.

一个非 prop 特性是指传向一个组件，但是该组件并没有相应 prop 定义的特性。

While explicitly defined props are preferred for passing information to a child component, authors of component libraries can’t always foresee the contexts in which their components might be used. That’s why components can accept arbitrary attributes, which are added to the component’s root element.

因为显式定义的 prop 适用于向一个子组件传入信息，然而组件库的作者并不总能预见组件会被用于怎样的场景。这也是为什么组件可以接受任意的特性，而这些特性会被添加到这个组件的根元素上。

For example, imagine we’re using a 3rd-party bootstrap-date-input component with a Bootstrap plugin that requires a data-date-picker attribute on the input. We can add this attribute to our component instance:

例如，想象一下你通过一个 Bootstrap 插件使用了一个第三方的 <bootstrap-date-input> 组件，这个插件需要在其 <input> 上用到一个 data-date-picker 特性。我们可以将这个特性添加到你的组件实例上：

And the data-date-picker="activated" attribute will automatically be added to the root element of bootstrap-date-input.

然后这个 data-date-picker="activated" 特性就会自动添加到 <bootstrap-date-input> 的根元素上。

Replacing/Merging with Existing Attributes 替换/合并已有的特性

Imagine this is the template for bootstrap-date-input:

想象一下 <bootstrap-date-input> 的模板是这样的：

To specify a theme for our date picker plugin, we might need to add a specific class, like this:

为了给我们的日期选择器插件定制一个主题，我们可能需要像这样添加一个特别的类名：

In this case, two different values for class are defined:

在这种情况下，我们定义了两个不同的 class 的值：

form-control, which is set by the component in its template

form-control，这是在组件的模板内设置好的

date-picker-theme-dark, which is passed to the component by its parent

date-picker-theme-dark，这是从组件的父级传入的

For most attributes, the value provided to the component will replace the value set by the component. So for example, passing type="text" will replace type="date" and probably break it! Fortunately, the class and style attributes are a little smarter, so both values are merged, making the final value: form-control date-picker-theme-dark.

对于绝大多数特性来说，从外部提供给组件的值会替换掉组件内部设置好的值。所以如果传入 type="text" 就会替换掉 type="date" 并把它破坏！庆幸的是，class和 style 特性会稍微智能一些，即两边的值会被合并起来，从而得到最终的值：form-control date-picker-theme-dark。

Disabling Attribute Inheritance 禁用特性继承

If you do not want the root element of a component to inherit attributes, you can set inheritAttrs: false in the component’s options. For example:

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。例如：

This can be especially useful in combination with the $attrs instance property, which contains the attribute names and values passed to a component, such as:

这尤其适合配合实例的 $attrs 属性使用，该属性包含了传递给一个组件的特性名和特性值，例如：

With inheritAttrs: false and $attrs, you can manually decide which element you want to forward attributes to, which is often desirable for base components:

有了 inheritAttrs: false 和 $attrs，你就可以手动决定这些特性会被赋予哪个元素。在撰写基础组件的时候是常会用到的：

Note that inheritAttrs: false option does not affect style and class bindings.

注意 inheritAttrs: false 选项不会影响 style 和 class 的绑定。

This pattern allows you to use base components more like raw HTML elements, without having to care about which element is actually at its root:

这个模式允许你在使用基础组件的时候更像是使用原始的 HTML 元素，而不会担心哪个元素是真正的根元素：

## Custom Events自定义事件

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

Event Names 事件名

Unlike components and props, event names don’t provide any automatic case transformation. Instead, the name of an emitted event must exactly match the name used to listen to that event. For example, if emitting a camelCased event name:

不同于组件和 prop，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。举个例子，如果触发一个 camelCase 名字的事件：

Listening to the kebab-cased version will have no effect:

则监听这个名字的 kebab-case 版本是不会有任何效果的：

Unlike components and props, event names will never be used as variable or property names in JavaScript, so there’s no reason to use camelCase or PascalCase. Additionally, v-on event listeners inside DOM templates will be automatically transformed to lowercase (due to HTML’s case-insensitivity), so v-on:myEvent would become v-on:myevent – making myEvent impossible to listen to.

不同于组件和 prop，事件名不会被用作一个 JavaScript 变量名或属性名，所以就没有理由使用 camelCase 或 PascalCase 了。并且 v-on 事件监听器在 DOM 模板中会被自动转换为全小写 (因为 HTML 是大小写不敏感的)，所以 v-on:myEvent 将会变成 v-on:myevent——导致 myEvent 不可能被监听到。

For these reasons, we recommend you always use kebab-case for event names.

因此，我们推荐你始终使用 kebab-case 的事件名。

Customizing Component v-model 自定义组件的 v-model

By default, v-model on a component uses value as the prop and input as the event, but some input types such as checkboxes and radio buttons may want to use the value attribute for a different purpose. Using the model option can avoid a conflict in such cases:

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。model 选项可以用来避免这样的冲突：

Now when using v-model on this component:

现在在这个组件上使用 v-model 的时候：

the value of lovingVue will be passed to the checked prop. The lovingVue property will then be updated when <base-checkbox> emits a change event with a new value.

这里的 lovingVue 的值将会传入这个名为 checked 的 prop。同时当 <base-checkbox> 触发一个 change 事件并附带一个新的值的时候，这个 lovingVue 的属性将会被更新。

Note that you still have to declare the checked prop in component’s props option.

注意你仍然需要在组件的 props 选项里声明 checked 这个 prop。

Binding Native Events to Components 将原生事件绑定到组件

There may be times when you want to listen directly to a native event on the root element of a component. In these cases, you can use the .native modifier for v-on:

你可能有很多次想要在一个组件的根元素上直接监听一个原生事件。这时，你可以使用 v-on 的 .native 修饰符：

This can be useful sometimes, but it’s not a good idea when you’re trying to listen on a very specific element, like an <input>. For example, the <base-input> component above might refactor so that the root element is actually a <label> element:

在有的时候这是很有用的，不过在你尝试监听一个类似 <input> 的非常特定的元素时，这并不是个好主意。比如上述 <base-input> 组件可能做了如下重构，所以根元素实际上是一个 <label> 元素：

In that case, the .native listener in the parent would silently break. There would be no errors, but the onFocus handler wouldn’t be called when we expected it to.

这时，父级的 .native 监听器将静默失败。它不会产生任何报错，但是 onFocus 处理函数不会如你预期地被调用。

To solve this problem, Vue provides a $listeners property containing an object of listeners being used on the component. For example:

为了解决这个问题，Vue 提供了一个 $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。例如：

Using the $listeners property, you can forward all event listeners on the component to a specific child element with v-on="$listeners". For elements like <input>, that you also want to work with v-model, it’s often useful to create a new computed property for listeners, like inputListeners below:

有了这个 $listeners 属性，你就可以配合 v-on="$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。对于类似 <input> 的你希望它也可以配合 v-model 工作的组件来说，为这些监听器创建一个类似下述 inputListeners 的计算属性通常是非常有用的：

Now the <base-input> component is a fully transparent wrapper, meaning it can be used exactly like a normal <input> element: all the same attributes and listeners will work, without the .nativemodifier.

现在 <base-input> 组件是一个完全透明的包裹器了，也就是说它可以完全像一个普通的 <input> 元素一样使用了：所有跟它相同的特性和监听器的都可以工作。

.sync Modifier  修饰符

In some cases, we may need “two-way binding” for a prop. Unfortunately, true two-way binding can create maintenance issues, because child components can mutate the parent without the source of that mutation being obvious in both the parent and the child.

在有些情况下，我们可能需要对一个 prop 进行“双向绑定”。不幸的是，真正的双向绑定会带来维护上的问题，因为子组件可以修改父组件，且在父组件和子组件都没有明显的改动来源。

That’s why instead, we recommend emitting events in the pattern of update:myPropName. For example, in a hypothetical component with a title prop, we could communicate the intent of assigning a new value with:

这也是为什么我们推荐以 update:myPropName 的模式触发事件取而代之。举个例子，在一个包含 title prop 的假设的组件中，我们可以用以下方法表达对其赋新值的意图：

Then the parent can listen to that event and update a local data property, if it wants to. For example:

然后父组件可以监听那个事件并根据需要更新一个本地的数据属性。例如：

For convenience, we offer a shorthand for this pattern with the .sync modifier:

为了方便起见，我们为这种模式提供一个缩写，即 .sync 修饰符：

Note that v-bind with the .sync modifier does not work with expressions (e.g. v-bind:title.sync=”doc.title + ‘!’” is invalid). Instead, you must only provide the name of the property you want to bind, similar to v-model.

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。

The .sync modifier can also be used with v-bind when using an object to set multiple props at once:

当我们用一个对象同时设置多个 prop 的时候，也可以将这个 .sync 修饰符和 v-bind 配合使用：

This passes each property in the doc object (e.g. title) as an individual prop, then adds v-onupdate listeners for each one.

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。

Using v-bind.sync with a literal object, such as in v-bind.sync=”{ title: doc.title }”, will not work, because there are too many edge cases to consider in parsing a complex expression like this.

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，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

## Slots插槽

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

In 2.6.0, we introduced a new unified syntax (the v-slot directive) for named and scoped slots. It replaces the slot and slot-scope attributes, which are now deprecated, but have notbeen removed and are still documented here. The rationale for introducing the new syntax is described in this RFC.

在 2.6.0 中，我们为具名插槽和作用域插槽引入了一个新的统一的语法 (即 v-slot 指令)。它取代了 slot 和 slot-scope 这两个目前已被废弃但未被移除且仍在文档中的特性。新语法的由来可查阅这份 RFC。

Slot Content 插槽内容

Vue implements a content distribution API inspired by the Web Components spec draft, using the <slot> element to serve as distribution outlets for content.

Vue 实现了一套内容分发的 API，这套 API 的设计灵感源自 Web Components 规范草案，将 <slot> 元素作为承载分发内容的出口。

This allows you to compose components like this:

它允许你像这样合成组件：

Then in the template for <navigation-link>, you might have:

然后你在 <navigation-link> 的模板中可能会写为：

When the component renders, <slot></slot> will be replaced by “Your Profile”. Slots can contain any template code, including HTML:

当组件渲染的时候，<slot></slot> 将会被替换为“Your Profile”。插槽内可以包含任何模板代码，包括 HTML：

Or even other components: 甚至其它的组件：

If <navigation-link>‘s template did not contain a <slot> element, any content provided between its opening and closing tag would be discarded.

如果 <navigation-link> 没有包含一个 <slot> 元素，则该组件起始标签和结束标签之间的任何内容都会被抛弃。

Compilation Scope 编译作用域

When you want to use data inside a slot, such as in:

当你想在一个插槽中使用数据时，例如：

That slot has access to the same instance properties (i.e. the same “scope”) as the rest of the template. The slot does not have access to <navigation-link>‘s scope. For example, trying to access urlwould not work:

该插槽跟模板的其它地方一样可以访问相同的实例属性 (也就是相同的“作用域”)，而不能访问 <navigation-link> 的作用域。例如 url 是访问不到的：

As a rule, remember that:

作为一条规则，请记住：

Everything in the parent template is compiled in parent scope; everything in the child template is compiled in the child scope.

父级模板里的所有内容都是在父级作用域中编译的；子模板里的所有内容都是在子作用域中编译的。

Fallback Content 后备内容

There are cases when it’s useful to specify fallback (i.e. default) content for a slot, to be rendered only when no content is provided. For example, in a <submit-button> component:

有时为一个插槽设置具体的后备 (也就是默认的) 内容是很有用的，它只会在没有提供内容的时候被渲染。例如在一个 <submit-button> 组件中：

We might want the text “Submit” to be rendered inside the <button> most of the time. To make “Submit” the fallback content, we can place it in between the <slot> tags:

我们可能希望这个 <button> 内绝大多数情况下都渲染文本“Submit”。为了将“Submit”作为后备内容，我们可以将它放在 <slot> 标签内：

Now when we use <submit-button> in a parent component, providing no content for the slot:

现在当我在一个父级组件中使用 <submit-button> 并且不提供任何插槽内容时：

will render the fallback content, “Submit”:

后备内容“Submit”将会被渲染：

But if we provide content:

但是如果我们提供内容：

Then the provided content will be rendered instead:

则这个提供的内容将会被渲染从而取代后备内容：

Named Slots 具名插槽

Updated in 2.6.0+. See here for the deprecated syntax using the slot attribute.

自 2.6.0 起有所更新。已废弃的使用 slot 特性的语法在这里。

There are times when it’s useful to have multiple slots. For example, in a <base-layout> component with the following template:

有时我们需要多个插槽。例如对于一个带有如下模板的 <base-layout> 组件：

For these cases, the <slot> element has a special attribute, name, which can be used to define additional slots:

对于这样的情况，<slot> 元素有一个特殊的特性：name。这个特性可以用来定义额外的插槽：

A <slot> outlet without name implicitly has the name “default”.

一个不带 name 的 <slot> 出口会带有隐含的名字“default”。

To provide content to named slots, we can use the v-slot directive on a <template>, providing the name of the slot as v-slot‘s argument:

在向具名插槽提供内容的时候，我们可以在一个 <template> 元素上使用 v-slot 指令，并以 v-slot 的参数的形式提供其名称：

Now everything inside the <template> elements will be passed to the corresponding slots. Any content not wrapped in a <template> using v-slot is assumed to be for the default slot.

现在 <template> 元素中的所有内容都将会被传入相应的插槽。任何没有被包裹在带有 v-slot 的 <template> 中的内容都会被视为默认插槽的内容。

However, you can still wrap default slot content in a <template> if you wish to be explicit:

然而，如果你希望更明确一些，仍然可以在一个 <template> 中包裹默认插槽的内容：

Either way, the rendered HTML will be:

任何一种写法都会渲染出：

Note that v-slot can only be added to a <template> (with one exception), unlike the deprecated slot attribute.

注意 v-slot 只能添加在一个 <template> 上 (只有一种例外情况)，这一点和已经废弃的 slot 特性不同。

Scoped Slots 作用域插槽

Updated in 2.6.0+. See here for the deprecated syntax using the slot-scope attribute.

自 2.6.0 起有所更新。已废弃的使用 slot-scope 特性的语法在这里。

Sometimes, it’s useful for slot content to have access to data only available in the child component. For example, imagine a <current-user> component with the following template:、

有时让插槽内容能够访问子组件中才有的数据是很有用的。例如，设想一个带有如下模板的 <current-user> 组件：

We might want to replace this fallback content to display the user’s first name, instead of last, like this:

我们想让它的后备内容显示用户的名，以取代正常情况下用户的姓，如下：

That won’t work, however, because only the <current-user> component has access to the userand the content we’re providing is rendered in the parent.

然而上述代码不会正常工作，因为只有 <current-user> 组件可以访问到 user 而我们提供的内容是在父级渲染的。

To make user available to the slot content in the parent, we can bind user as an attribute to the <slot> element:

为了让 user 在父级的插槽内容可用，我们可以将 user 作为 <slot> 元素的一个特性绑定上去：

Attributes bound to a <slot> element are called slot props. Now, in the parent scope, we can use v-slot with a value to define a name for the slot props we’ve been provided:

绑定在 <slot> 元素上的特性被称为插槽 prop。现在在父级作用域中，我们可以给 v-slot 带一个值来定义我们提供的插槽 prop 的名字：

In this example, we’ve chosen to name the object containing all our slot props slotProps, but you can use any name you like.

在这个例子中，我们选择将包含所有插槽 prop 的对象命名为 slotProps，但你也可以使用任意你喜欢的名字。

Abbreviated Syntax for Lone Default Slots 独占默认插槽的缩写语法

In cases like above, when only the default slot is provided content, the component’s tags can be used as the slot’s template. This allows us to use v-slot directly on the component:

在上述情况下，当被提供的内容只有默认插槽时，组件的标签才可以被当作插槽的模板来使用。这样我们就可以把 v-slot 直接用在组件上：

This can be shortened even further. Just as non-specified content is assumed to be for the default slot, v-slot without an argument is assumed to refer to the default slot:

这种写法还可以更简单。就像假定未指明的内容对应默认插槽一样，不带参数的 v-slot 被假定对应默认插槽：

Note that the abbreviated syntax for default slot cannot be mixed with named slots, as it would lead to scope ambiguity:

注意默认插槽的缩写语法不能和具名插槽混用，因为它会导致作用域不明确：

Whenever there are multiple slots, use the full <template> based syntax for all slots:

只要出现多个插槽，请始终为所有的插槽使用完整的基于 <template> 的语法：

Destructuring Slot Props 解构插槽 Prop

Internally, scoped slots work by wrapping your slot content in a function passed a single argument:

作用域插槽的内部工作原理是将你的插槽内容包括在一个传入单个参数的函数里：

That means the value of v-slot can actually accept any valid JavaScript expression that can appear in the argument position of a function definition. So in supported environments (single-file components or modern browsers), you can also use ES2015 destructuring to pull out specific slot props, like so:

这意味着 v-slot 的值实际上可以是任何能够作为函数定义中的参数的 JavaScript 表达式。所以在支持的环境下 (单文件组件或现代浏览器)，你也可以使用 ES2015 解构来传入具体的插槽 prop，如下：

This can make the template much cleaner, especially when the slot provides many props. It also opens other possibilities, such as renaming props, e.g. user to person:

这样可以使模板更简洁，尤其是在该插槽提供了多个 prop 的时候。它同样开启了 prop 重命名等其它可能，例如将 user 重命名为 person：

You can even define fallbacks, to be used in case a slot prop is undefined:

你甚至可以定义后备内容，用于插槽 prop 是 undefined 的情形：

Dynamic Slot Names 动态插槽名

Dynamic directive arguments also work on v-slot, allowing the definition of dynamic slot names:

动态指令参数也可以用在 v-slot 上，来定义动态的插槽名：

Named Slots Shorthand 具名插槽的缩写

Similar to v-on and v-bind, v-slot also has a shorthand, replacing everything before the argument (v-slot:) with the special symbol #. For example, v-slot:header can be rewritten as #header:

跟 v-on 和 v-bind 一样，v-slot 也有缩写，即把参数之前的所有内容 (v-slot:) 替换为字符 #。例如 v-slot:header 可以被重写为 #header：

However, just as with other directives, the shorthand is only available when an argument is provided. That means the following syntax is invalid:

然而，和其它指令一样，该缩写只在其有参数的时候才可用。这意味着以下语法是无效的：

Instead, you must always specify the name of the slot if you wish to use the shorthand:

如果你希望使用缩写的话，你必须始终以明确插槽名取而代之：

Other Examples 其它示例

Slot props allow us to turn slots into reusable templates that can render different content based on input props. This is most useful when you are designing a reusable component that encapsulates data logic while allowing the consuming parent component to customize part of its layout.

插槽 prop 允许我们将插槽转换为可复用的模板，这些模板可以基于输入的 prop 渲染出不同的内容。这在设计封装数据逻辑同时允许父级组件自定义部分布局的可复用组件时是最有用的。

For example, we are implementing a <todo-list> component that contains the layout and filtering logic for a list:

例如，我们要实现一个 <todo-list> 组件，它是一个列表且包含布局和过滤逻辑：

Instead of hard-coding the content for each todo, we can let the parent component take control by making every todo a slot, then binding todo as a slot prop:

我们可以将每个 todo 作为父级组件的插槽，以此通过父级组件对其进行控制，然后将 todo 作为一个插槽 prop 进行绑定：

Now when we use the <todo-list> component, we can optionally define an alternative <template> for todo items, but with access to data from the child:

现在当我们使用 <todo-list> 组件的时候，我们可以选择为 todo 定义一个不一样的 <template> 作为替代方案，并且可以从子组件获取数据：

However, even this barely scratches the surface of what scoped slots are capable of. For real-life, powerful examples of scoped slot usage, we recommend browsing libraries such as Vue Virtual Scroller, Vue Promised, and Portal Vue.

这只是作用域插槽用武之地的冰山一角。想了解更多现实生活中的作用域插槽的用法，我们推荐浏览诸如 Vue Virtual Scroller、Vue Promised 和 Portal Vue 等库。

Deprecated Syntax 废弃了的语法

The v-slot directive was introduced in Vue 2.6.0, offering an improved, alternative API to the still-supported slot and slot-scope attributes. The full rationale for introducing v-slot is described in this RFC. The slot and slot-scope attributes will continue to be supported in all future 2.x releases, but are officially deprecated and will eventually be removed in Vue 3.

v-slot 指令自 Vue 2.6.0 起被引入，提供更好的支持 slot 和 slot-scope 特性的 API 替代方案。v-slot 完整的由来参见这份 RFC。在接下来所有的 2.x 版本中 slot 和 slot-scope 特性仍会被支持，但已经被官方废弃且不会出现在 Vue 3 中。

Named Slots with the slot Attribute 带有 slot 特性的具名插槽

Deprecated in 2.6.0+. See here for the new, recommended syntax.

自 2.6.0 起被废弃。新推荐的语法请查阅这里。

To pass content to named slots from the parent, use the special slot attribute on <template>(using the <base-layout> component described here as example):

在 <template> 上使用特殊的 slot 特性，可以将内容从父级传给具名插槽 (把这里提到过的 <base-layout> 组件作为示例)：

Or, the slot attribute can also be used directly on a normal element:

或者直接把 slot 特性用在一个普通元素上：

There can still be one unnamed slot, which is the default slot that serves as a catch-all for any unmatched content. In both examples above, the rendered HTML would be:

这里其实还有一个未命名插槽，也就是默认插槽，捕获所有未被匹配的内容。上述两个示例的 HTML 渲染结果均为：

Scoped Slots with the slot-scope Attribute 带有 slot-scope 特性的作用域插槽

Deprecated in 2.6.0+. See here for the new, recommended syntax.

自 2.6.0 起被废弃。新推荐的语法请查阅这里。

To receive props passed to a slot, the parent component can use <template> with the slot-scopeattribute (using the <slot-example> described here as example):

在 <template> 上使用特殊的 slot-scope 特性，可以接收传递给插槽的 prop (把这里提到过的 <slot-example> 组件作为示例)：

Here, slot-scope declares the received props object as the slotProps variable, and makes it available inside the <template> scope. You can name slotProps anything you like similar to naming function arguments in JavaScript.

这里的 slot-scope 声明了被接收的 prop 对象会作为 slotProps 变量存在于 <template> 作用域中。你可以像命名 JavaScript 函数参数一样随意命名 slotProps。

Here slot="default" can be omitted as it is implied:

这里的 slot="default" 可以被忽略为隐性写法：

The slot-scope attribute can also be used directly on a non-<template> element (including components):

slot-scope 特性也可以直接用于非 <template> 元素 (包括组件)：

The value of slot-scope can accept any valid JavaScript expression that can appear in the argument position of a function definition. This means in supported environments (single-file components or modern browsers) you can also use ES2015 destructuring in the expression, like so:

slot-scope 的值可以接收任何有效的可以出现在函数定义的参数位置上的 JavaScript 表达式。这意味着在支持的环境下 (单文件组件或现代浏览器)，你也可以在表达式中使用 ES2015 解构，如下：

Using the <todo-list> described here as an example, here’s the equivalent usage using slot-scope:

使用这里描述过的 <todo-list> 作为示例，与它等价的使用 slot-scope 的代码是：

## Dynamic & Async Components动态组件 & 异步组件

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

keep-alive with Dynamic Components 在动态组件上使用 keep-alive

Earlier, we used the is attribute to switch between components in a tabbed interface:

我们之前曾经在一个多标签的界面中使用 is 特性来切换不同的组件：

When switching between these components though, you’ll sometimes want to maintain their state or avoid re-rendering for performance reasons. For example, when expanding our tabbed interface a little:

当在这些组件之间切换的时候，你有时会想保持这些组件的状态，以避免反复重渲染导致的性能问题。例如我们来展开说一说这个多标签界面：

You’ll notice that if you select a post, switch to the Archive tab, then switch back to Posts, it’s no longer showing the post you selected. That’s because each time you switch to a new tab, Vue creates a new instance of the currentTabComponent.

你会注意到，如果你选择了一篇文章，切换到 Archive 标签，然后再切换回 Posts，是不会继续展示你之前选择的文章的。这是因为你每次切换新标签的时候，Vue 都创建了一个新的 currentTabComponent 实例。

Recreating dynamic components is normally useful behavior, but in this case, we’d really like those tab component instances to be cached once they’re created for the first time. To solve this problem, we can wrap our dynamic component with a <keep-alive> element:

重新创建动态组件的行为通常是非常有用的，但是在这个案例中，我们更希望那些标签的组件实例能够被在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 <keep-alive> 元素将其动态组件包裹起来。

Now the Posts tab maintains its state (the selected post) even when it’s not rendered. See this fiddle for the complete code.

现在这个 Posts 标签保持了它的状态 (被选中的文章) 甚至当它未被渲染时也是如此。你可以在这个 fiddle 查阅到完整的代码。

Note that <keep-alive> requires the components being switched between to all have names, either using the name option on a component, or through local/global registration.

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。

Check out more details on <keep-alive> in the API reference.

你可以在 API 参考文档 查阅更多关于 <keep-alive> 的细节。

Async Components 异步组件

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it’s needed. To make that easier, Vue allows you to define your component as a factory function that asynchronously resolves your component definition. Vue will only trigger the factory function when the component needs to be rendered and will cache the result for future re-renders. For example:

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，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。例如：

As you can see, the factory function receives a resolve callback, which should be called when you have retrieved your component definition from the server. You can also call reject(reason) to indicate the load has failed. The setTimeout here is for demonstration; how to retrieve the component is up to you. One recommended approach is to use async components together with Webpack’s code-splitting feature:

如你所见，这个工厂函数会收到一个 resolve 回调，这个回调函数会在你从服务器得到组件定义的时候被调用。你也可以调用 reject(reason) 来表示加载失败。这里的 setTimeout 是为了演示用的，如何获取组件取决于你自己。一个推荐的做法是将异步组件和 webpack 的 code-splitting 功能一起配合使用：

You can also return a Promise in the factory function, so with Webpack 2 and ES2015 syntax you can do:

你也可以在工厂函数中返回一个 Promise，所以把 webpack 2 和 ES2015 语法加在一起，我们可以写成这样：

When using local registration, you can also directly provide a function that returns a Promise:

当使用局部注册的时候，你也可以直接提供一个返回 Promise 的函数：

If you’re a Browserify user that would like to use async components, its creator has unfortunately made it clear that async loading “is not something that Browserify will ever support.” Officially, at least. The Browserify community has found some workarounds, which may be helpful for existing and complex applications. For all other scenarios, we recommend using Webpack for built-in, first-class async support.

如果你是一个 Browserify 用户同时喜欢使用异步组件，很不幸这个工具的作者明确表示异步加载“并不会被 Browserify 支持”，至少官方不会。Browserify 社区已经找到了一些变通方案，这些方案可能会对已存在的复杂应用有帮助。对于其它的场景，我们推荐直接使用 webpack，以拥有内置的头等异步支持。

Handling Loading State 处理加载状态

The async component factory can also return an object of the following format:

这里的异步组件工厂函数也可以返回一个如下格式的对象：

Note that you must use Vue Router 2.4.0+ if you wish to use the above syntax for route components.

注意如果你希望在 Vue Router 的路由组件中使用上述语法的话，你必须使用 Vue Router 2.4.0+ 版本。

## Handling Edge Cases处理边界情况

This page assumes you’ve already read the Components Basics. Read that first if you are new to components.

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

All the features on this page document the handling of edge cases, meaning unusual situations that sometimes require bending Vue’s rules a little. Note however, that they all have disadvantages or situations where they could be dangerous. These are noted in each case, so keep them in mind when deciding to use each feature.

这里记录的都是和处理边界情况有关的功能，即一些需要对 Vue 的规则做一些小调整的特殊情况。不过注意这些功能都是有劣势或危险的场景的。我们会在每个案例中注明，所以当你使用每个功能的时候请稍加留意。

Element & Component Access 访问元素 & 组件

In most cases, it’s best to avoid reaching into other component instances or manually manipulating DOM elements. There are cases, however, when it can be appropriate.

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。

Accessing the Root Instance 访问根实例

In every subcomponent of a new Vue instance, this root instance can be accessed with the $rootproperty. For example, in this root instance:

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。例如，在这个根实例中：

All subcomponents will now be able to access this instance and use it as a global store:

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。

This can be convenient for demos or very small apps with a handful of components. However, the pattern does not scale well to medium or large-scale applications, so we strongly recommend using Vuex to manage state in most cases.

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 来管理应用的状态。

Accessing the Parent Component Instance 访问父级组件实例

Similar to $root, the $parent property can be used to access the parent instance from a child. This can be tempting to reach for as a lazy alternative to passing data with a prop.

和 $root 类似，$parent 属性可以用来从一个子组件访问父组件的实例。它提供了一种机会，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。

In most cases, reaching into the parent makes your application more difficult to debug and understand, especially if you mutate data in the parent. When looking at that component later, it will be very difficult to figure out where that mutation came from.

在绝大多数情况下，触达父级组件会使得你的应用更难调试和理解，尤其是当你变更了父级组件的数据的时候。当我们稍后回看那个组件的时候，很难找出那个变更是从哪里发起的。

There are cases however, particularly shared component libraries, when this might be appropriate. For example, in abstract components that interact with JavaScript APIs instead of rendering HTML, like these hypothetical Google Maps components:

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。举个例子，在和 JavaScript API 进行交互而不渲染 HTML 的抽象组件内，诸如这些假设性的 Google 地图组件一样：

The <google-map> component might define a map property that all subcomponents need access to. In this case <google-map-markers> might want to access that map with something like this.$parent.getMap, in order to add a set of markers to it. You can see this pattern in action here.

这个 <google-map> 组件可以定义一个 map 属性，所有的子组件都需要访问它。在这种情况下 <google-map-markers> 可能想要通过类似 this.$parent.getMap 的方式访问那个地图，以便为其添加一组标记。你可以在这里查阅这种模式。

Keep in mind, however, that components built with this pattern are still inherently fragile. For example, imagine we add a new <google-map-region> component and when <google-map-markers>appears within that, it should only render markers that fall within that region:

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。比如，设想一下我们添加一个新的 <google-map-region> 组件，当 <google-map-markers> 在其内部出现的时候，只会渲染那个区域内的标记：

Then inside <google-map-markers> you might find yourself reaching for a hack like this:

那么在 <google-map-markers> 内部你可能发现自己需要一些类似这样的 hack：

This has quickly gotten out of hand. That’s why to provide context information to descendent components arbitrarily deep, we instead recommend dependency injection.

很快它就会失控。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的原因。

Accessing Child Component Instances & Child Elements 访问子组件实例或子元素

Despite the existence of props and events, sometimes you might still need to directly access a child component in JavaScript. To achieve this you can assign a reference ID to the child component using the ref attribute. For example:

尽管存在 prop 和事件，有的时候你仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，你可以通过 ref 特性为这个子组件赋予一个 ID 引用。例如：

Now in the component where you’ve defined this ref, you can use:

现在在你已经定义了这个 ref 的组件里，你可以使用：

to access the <base-input> instance. This may be useful when you want to, for example, programmatically focus this input from a parent. In that case, the <base-input> component may similarly use a ref to provide access to specific elements inside it, such as:

来访问这个 <base-input> 实例，以便不时之需。比如程序化地从一个父级组件聚焦这个输入框。在刚才那个例子中，该 <base-input> 组件也可以使用一个类似的 ref 提供对内部这个指定元素的访问，例如：

And even define methods for use by the parent:

甚至可以通过其父级组件定义方法：

Thus allowing the parent component to focus the input inside <base-input> with:

这样就允许父级组件通过下面的代码聚焦 <base-input> 里的输入框：

When ref is used together with v-for, the ref you get will be an array containing the child components mirroring the data source.

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 are only populated after the component has been rendered, and they are not reactive. It is only meant as an escape hatch for direct child manipulation - you should avoid accessing $refs from within templates or computed properties.

$refs 只会在组件渲染完成之后生效，并且它们不是响应式的。这仅作为一个用于直接操作子组件的“逃生舱”——你应该避免在模板或计算属性中访问 $refs。

Dependency Injection 依赖注入

Earlier, when we described Accessing the Parent Component Instance, we showed an example like this:

在此之前，在我们描述访问父级组件实例的时候，展示过一个类似这样的例子：

In this component, all descendants of <google-map> needed access to a getMap method, in order to know which map to interact with. Unfortunately, using the $parent property didn’t scale well to more deeply nested components. That’s where dependency injection can be useful, using two new instance options: provide and inject.

在这个组件里，所有 <google-map> 的后代都需要访问一个 getMap 方法，以便知道要跟哪个地图进行交互。不幸的是，使用 $parent 属性无法很好的扩展到更深层级的嵌套组件上。这也是依赖注入的用武之地，它用到了两个新的实例选项：provide和 inject。

The provide options allows us to specify the data/methods we want to provide to descendent components. In this case, that’s the getMap method inside <google-map>:

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。在这个例子中，就是 <google-map> 内部的 getMap 方法：

Then in any descendants, we can use the inject option to receive specific properties we’d like to add to that instance:

然后在任何后代组件里，我们都可以使用 inject 选项来接收指定的我们想要添加在这个实例上的属性：

You can see the full example here. The advantage over using $parent is that we can access getMapin any descendant component, without exposing the entire instance of <google-map>. This allows us to more safely keep developing that component, without fear that we might change/remove something that a child component is relying on. The interface between these components remains clearly defined, just as with props.

你可以在这里看到完整的示例。相比 $parent 来说，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。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。同时这些组件之间的接口是始终明确定义的，就和 props 一样。

In fact, you can think of dependency injection as sort of “long-range props”, except:

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：

ancestor components don’t need to know which descendants use the properties it provides

祖先组件不需要知道哪些后代组件使用它提供的属性

descendant components don’t need to know where injected properties are coming from

后代组件不需要知道被注入的属性来自哪里

However, there are downsides to dependency injection. It couples components in your application to the way they’re currently organized, making refactoring more difficult. Provided properties are also not reactive. This is by design, because using them to create a central data store scales just as poorly as using $root for the same purpose. If the properties you want to share are specific to your app, rather than generic, or if you ever want to update provided data inside ancestors, then that’s a good sign that you probably need a real state management solution like Vuex instead.

然而，依赖注入还是有负面影响的。它将你应用程序中的组件与它们当前的组织方式耦合起来，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 这样真正的状态管理方案了。

Learn more about dependency injection in the API doc.

你可以在 API 参考文档学习更多关于依赖注入的知识。

Programmatic Event Listeners 程序化的事件侦听器

So far, you’ve seen uses of $emit, listened to with v-on, but Vue instances also offer other methods in its events interface. We can:

现在，你已经知道了 $emit 的用法，它可以被 v-on 侦听，但是 Vue 实例同时在其事件接口中提供了其它的方法。我们可以：

Listen for an event with $on(eventName, eventHandler)

通过 $on(eventName, eventHandler) 侦听一个事件

Listen for an event only once with $once(eventName, eventHandler)

通过 $once(eventName, eventHandler) 一次性侦听一个事件

Stop listening for an event with $off(eventName, eventHandler)

通过 $off(eventName, eventHandler) 停止侦听一个事件

You normally won’t have to use these, but they’re available for cases when you need to manually listen for events on a component instance. They can also be useful as a code organization tool. For example, you may often see this pattern for integrating a 3rd-party library:

你通常不会用到这些，但是当你需要在一个组件实例上手动侦听事件时，它们是派得上用场的。它们也可以用于代码组织工具。例如，你可能经常看到这种集成一个第三方库的模式：

This has two potential issues:

这里有两个潜在的问题：

It requires saving the picker to the component instance, when it’s possible that only lifecycle hooks need access to it. This isn’t terrible, but it could be considered clutter.

它需要在这个组件实例中保存这个 picker，如果可以的话最好只有生命周期钩子可以访问到它。这并不算严重的问题，但是它可以被视为杂物。

Our setup code is kept separate from our cleanup code, making it more difficult to programmatically clean up anything we set up.

我们的建立代码独立于我们的清理代码，这使得我们比较难于程序化地清理我们建立的所有东西。

You could resolve both issues with a programmatic listener:

你应该通过一个程序化的侦听器解决这两个问题：

Using this strategy, we could even use Pikaday with several input elements, with each new instance automatically cleaning up after itself:

使用了这个策略，我甚至可以让多个输入框元素同时使用不同的 Pikaday，每个新的实例都程序化地在后期清理它自己：

See this fiddle for the full code. Note, however, that if you find yourself having to do a lot of setup and cleanup within a single component, the best solution will usually be to create more modular components. In this case, we’d recommend creating a reusable <input-datepicker> component.

查阅这个 fiddle 可以了解到完整的代码。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模块化组件。在这个例子中，我们推荐创建一个可复用的 <input-datepicker> 组件。

To learn more about programmatic listeners, check out the API for Events Instance Methods.

想了解更多程序化侦听器的内容，请查阅实例方法 / 事件相关的 API。

Note that Vue’s event system is different from the browser’s EventTarget API. Though they work similarly, $emit, $on, and $off are not aliases for dispatchEvent, addEventListener, and removeEventListener.

注意 Vue 的事件系统不同于浏览器的 EventTarget API。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。

Circular References 循环引用

Recursive Components 递归组件

Components can recursively invoke themselves in their own template. However, they can only do so with the name option:

组件是可以在它们自己的模板中调用自身的。不过它们只能通过 name 选项来做这件事：

When you register a component globally using Vue.component, the global ID is automatically set as the component’s name option.

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。

If you’re not careful, recursive components can also lead to infinite loops:

稍有不慎，递归组件就可能导致无限循环：

A component like the above will result in a “max stack size exceeded” error, so make sure recursive invocation is conditional (i.e. uses a v-if that will eventually be false).

类似上述的组件将会导致“max stack size exceeded”错误，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)。

Circular References Between Components 组件之间的循环引用

Let’s say you’re building a file directory tree, like in Finder or File Explorer. You might have a tree-folder component with this template:

假设你需要构建一个文件目录树，像访达或资源管理器那样的。你可能有一个 <tree-folder> 组件，模板是这样的：

When you look closely, you’ll see that these components will actually be each other’s descendent andancestor in the render tree - a paradox! When registering components globally with Vue.component, this paradox is resolved for you automatically. If that’s you, you can stop reading here.

当你仔细观察的时候，你会发现这些组件在渲染树中互为对方的后代和祖先——一个悖论！当通过 Vue.component 全局注册组件的时候，这个悖论会被自动解开。如果你是这样做的，那么你可以跳过这里。

However, if you’re requiring/importing components using a module system, e.g. via Webpack or Browserify, you’ll get an error:

然而，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：

To explain what’s happening, let’s call our components A and B. The module system sees that it needs A, but first A needs B, but B needs A, but A needs B, etc. It’s stuck in a loop, not knowing how to fully resolve either component without first resolving the other. To fix this, we need to give the module system a point at which it can say, “A needs B eventually, but there’s no need to resolve B first.”

为了解释这里发生了什么，我们先把两个组件称为 A 和 B。模块系统发现它需要 A，但是首先 A 依赖 B，但是 B 又依赖 A，但是 A 又依赖 B，如此往复。这变成了一个循环，不知道如何不经过其中一个组件而完全解析出另一个组件。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。”

In our case, let’s make that point the tree-folder component. We know the child that creates the paradox is the tree-folder-contents component, so we’ll wait until the beforeCreate lifecycle hook to register it:

在我们的例子中，把 <tree-folder> 组件设为了那个点。我们知道那个产生悖论的子组件是 <tree-folder-contents> 组件，所以我们会等到生命周期钩子 beforeCreate 时去注册它：

Or alternatively, you could use Webpack’s asynchronous import when you register the component locally:

或者，在本地注册组件的时候，你可以使用 webpack 的异步 import：

Problem solved! 这样问题就解决了！

Alternate Template Definitions 模板定义的替代品

Inline Templates 内联模板

When the inline-template special attribute is present on a child component, the component will use its inner content as its template, rather than treating it as distributed content. This allows more flexible template-authoring.

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。

Your inline template needs to be defined inside the DOM element to which Vue is attached.

内联模板需要定义在 Vue 所属的 DOM 元素内。

However, inline-template makes the scope of your templates harder to reason about. As a best practice, prefer defining templates inside the component using the template option or in a <template> element in a .vue file.

不过，inline-template 会让模板的作用域变得更加难以理解。所以作为最佳实践，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。

Another way to define templates is inside of a script element with the type text/x-template, then referencing the template by an id. For example:

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：

Your x-template needs to be defined outside the DOM element to which Vue is attached.

x-template 需要定义在 Vue 所属的 DOM 元素外。

These can be useful for demos with large templates or in extremely small applications, but should otherwise be avoided, because they separate templates from the rest of the component definition.

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。

Controlling Updates 控制更新

Thanks to Vue’s Reactivity system, it always knows when to update (if you use it correctly). There are edge cases, however, when you might want to force an update, despite the fact that no reactive data has changed. Then there are other cases when you might want to prevent unnecessary updates.

感谢 Vue 的响应式系统，它始终知道何时进行更新 (如果你用对了的话)。不过还是有一些边界情况，你想要强制更新，尽管表面上看响应式的数据没有发生改变。也有一些情况是你想阻止不必要的更新。

Forcing an Update 强制更新

If you find yourself needing to force an update in Vue, in 99.99% of cases, you’ve made a

mistake somewhere.

如果你发现你自己需要在 Vue 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事。

You may not have accounted for change detection caveats with arrays or objects, or you may be relying on state that isn’t tracked by Vue’s reactivity system, e.g. with data.

你可能还没有留意到数组或对象的变更检测注意事项，或者你可能依赖了一个未被 Vue 的响应式系统追踪的状态。

However, if you’ve ruled out the above and find yourself in this extremely rare situation of having to manually force an update, you can do so with $forceUpdate.

然而，如果你已经做到了上述的事项仍然发现在极少数的情况下需要手动强制更新，那么你可以通过 $forceUpdate 来做这件事。

Cheap Static Components with v-once 通过 v-once 创建低开销的静态组件

Rendering plain HTML elements is very fast in Vue, but sometimes you might have a component that contains a lot of static content. In these cases, you can ensure that it’s only evaluated once and then cached by adding the v-once directive to the root element, like this: 渲染普通的 HTML 元素在 Vue 中是非常快速的，但有的时候你可能有一个组件，这个组件包含了大量静态内容。在这种情况下，你可以在根元素上添加 v-once 特性以确保这些内容只计算一次然后缓存起来，就像这样：

Once again, try not to overuse this pattern. While convenient in those rare cases when you have to render a lot of static content, it’s simply not necessary unless you actually notice slow rendering – plus, it could cause a lot of confusion later. For example, imagine another developer who’s not familiar with v-once or simply misses it in the template. They might spend hours trying to figure out why the template isn’t updating correctly.

再说一次，试着不要过度使用这个模式。当你需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非你非常留意渲染变慢了，不然它完全是没有必要的——再加上它在后期会带来很多困惑。例如，设想另一个开发者并不熟悉 v-once 或漏看了它在模板中，他们可能会花很多个小时去找出模板为什么无法正确更新。

# Transitions & Animation

## Enter/Leave & List Transitions进入/离开 & 列表过渡

Overview 概述

Vue provides a variety of ways to apply transition effects when items are inserted, updated, or removed from the DOM. This includes tools to:

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。  
包括以下工具：

automatically apply classes for CSS transitions and animations

在 CSS 过渡和动画中自动应用 class

integrate 3rd-party CSS animation libraries, such as Animate.css

可以配合使用第三方 CSS 动画库，如 Animate.css

use JavaScript to directly manipulate the DOM during transition hooks

在过渡钩子函数中使用 JavaScript 直接操作 DOM

integrate 3rd-party JavaScript animation libraries, such as Velocity.js

可以配合使用第三方 JavaScript 动画库，如 Velocity.js

On this page, we’ll only cover entering, leaving, and list transitions, but you can see the next section for managing state transitions.

在这里，我们只会讲到进入、离开和列表的过渡，你也可以看下一节的 管理过渡状态。

Transitioning Single Elements/Components 单元素/组件的过渡

Vue provides a transition wrapper component, allowing you to add entering/leaving transitions for any element or component in the following contexts:

Vue 提供了 transition 的封装组件，在下列情形中，可以给任何元素和组件添加进入/离开过渡

Conditional rendering (using v-if) 条件渲染 (使用 v-if)

Conditional display (using v-show) 条件展示 (使用 v-show)

Dynamic components动态组件

Component root nodes组件根节点

This is what an example looks like in action:

这里是一个典型的例子：

When an element wrapped in a transition component is inserted or removed, this is what happens:

当插入或删除包含在 transition 组件中的元素时，Vue 将会做以下处理：

Vue will automatically sniff whether the target element has CSS transitions or animations applied. If it does, CSS transition classes will be added/removed at appropriate timings.

自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。

If the transition component provided JavaScript hooks, these hooks will be called at appropriate timings.

如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。

If no CSS transitions/animations are detected and no JavaScript hooks are provided, the DOM operations for insertion and/or removal will be executed immediately on next frame (Note: this is a browser animation frame, different from Vue’s concept of nextTick).

如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作 (插入/删除) 在下一帧中立即执行。(注意：此指浏览器逐帧动画机制，和 Vue 的 nextTick 概念不同)

Transition Classes 过渡的类名

There are six classes applied for enter/leave transitions.

在进入/离开的过渡中，会有 6 个 class 切换。

v-enter: Starting state for enter. Added before element is inserted, removed one frame after element is inserted.

v-enter：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。

v-enter-active: Active state for enter. Applied during the entire entering phase. Added before element is inserted, removed when transition/animation finishes. This class can be used to define the duration, delay and easing curve for the entering transition.

v-enter-active：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。

v-enter-to: Only available in versions 2.1.8+. Ending state for enter. Added one frame after element is inserted (at the same time v-enter is removed), removed when transition/animation finishes.

v-enter-to: 2.1.8版及以上 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除)，在过渡/动画完成之后移除。

v-leave: Starting state for leave. Added immediately when a leaving transition is triggered, removed after one frame.

v-leave: 定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。

v-leave-active: Active state for leave. Applied during the entire leaving phase. Added immediately when leave transition is triggered, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the leaving transition.

v-leave-active：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。

v-leave-to: Only available in versions 2.1.8+. Ending state for leave. Added one frame after a leaving transition is triggered (at the same time v-leave is removed), removed when the transition/animation finishes.

v-leave-to: 2.1.8版及以上 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除)，在过渡/动画完成之后移除。

Each of these classes will be prefixed with the name of the transition. Here the v- prefix is the default when you use a <transition> element with no name. If you use <transition name="my-transition"> for example, then the v-enter class would instead be my-transition-enter.

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 <transition>，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。

v-enter-active and v-leave-active give you the ability to specify different easing curves for enter/leave transitions, which you’ll see an example of in the following section.

v-enter-active 和 v-leave-active 可以控制进入/离开过渡的不同的缓和曲线，在下面章节会有个示例说明。

CSS Transitions CSS 过渡

One of the most common transition types uses CSS transitions. Here’s an example:

常用的过渡都是使用 CSS 过渡。

CSS Animations CSS 动画

CSS animations are applied in the same way as CSS transitions, the difference being that v-enter is not removed immediately after the element is inserted, but on an animationend event.

CSS 动画用法同 CSS 过渡，区别是在动画中 v-enter 类名在节点插入 DOM 后不会立即删除，而是在 animationend 事件触发时删除。

Here’s an example, omitting prefixed CSS rules for the sake of brevity:

示例：(省略了兼容性前缀)

Custom Transition Classes 自定义过渡的类名

You can also specify custom transition classes by providing the following attributes:

我们可以通过以下特性来自定义过渡类名：

These will override the conventional class names. This is especially useful when you want to combine Vue’s transition system with an existing CSS animation library, such as Animate.css.

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如 Animate.css 结合使用十分有用。

Here’s an example: 示例：

Using Transitions and Animations Together 同时使用过渡和动画

Vue needs to attach event listeners in order to know when a transition has ended. It can either be transitionend or animationend, depending on the type of CSS rules applied. If you are only using one or the other, Vue can automatically detect the correct type.

Vue 为了知道过渡的完成，必须设置相应的事件监听器。它可以是 transitionend 或 animationend ，这取决于给元素应用的 CSS 规则。如果你使用其中任何一种，Vue 能自动识别类型并设置监听。

However, in some cases you may want to have both on the same element, for example having a CSS animation triggered by Vue, along with a CSS transition effect on hover. In these cases, you will have to explicitly declare the type you want Vue to care about in a type attribute, with a value of either animation or transition.

但是，在一些场景中，你需要给同一个元素同时设置两种过渡动效，比如 animation很快的被触发并完成了，而 transition 效果还没结束。在这种情况中，你就需要使用 type 特性并设置 animation 或 transition 来明确声明你需要 Vue 监听的类型。

Explicit Transition Durations 显性的过渡持续时间

In most cases, Vue can automatically figure out when the transition has finished. By default, Vue waits for the first transitionend or animationend event on the root transition element. However, this may not always be desired - for example, we may have a choreographed transition sequence where some nested inner elements have a delayed transition or a longer transition duration than the root transition element.

在很多情况下，Vue 可以自动得出过渡效果的完成时机。默认情况下，Vue 会等待其在过渡效果的根元素的第一个 transitionend 或 animationend 事件。然而也可以不这样设定——比如，我们可以拥有一个精心编排的一系列过渡效果，其中一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

In such cases you can specify an explicit transition duration (in milliseconds) using the duration prop on the <transition> component:

在这种情况下你可以用 <transition> 组件上的 duration 属性定制一个显性的过渡持续时间 (以毫秒计)：

You can also specify separate values for enter and leave durations:

你也可以定制进入和移出的持续时间：

You can also define JavaScript hooks in attributes:

可以在属性中声明 JavaScript 钩子

These hooks can be used in combination with CSS transitions/animations or on their own.

这些钩子函数可以结合 CSS transitions/animations 使用，也可以单独使用。

When using JavaScript-only transitions, the done callbacks are required for the enter and leave hooks. Otherwise, the hooks will be called synchronously and the transition will finish immediately.

当只用 JavaScript 过渡的时候，在 enter 和 leave 中必须使用 done 进行回调。否则，它们将被同步调用，过渡会立即完成。

It’s also a good idea to explicitly add v-bind:css="false" for JavaScript-only transitions so that Vue can skip the CSS detection. This also prevents CSS rules from accidentally interfering with the transition.

推荐对于仅使用 JavaScript 过渡的元素添加 v-bind:css="false"，Vue 会跳过 CSS 的检测。这也可以避免过渡过程中 CSS 的影响。

Now let’s dive into an example. Here’s a JavaScript transition using Velocity.js:

一个使用 Velocity.js 的简单例子：

Transitions on Initial Render 初始渲染的过渡

If you also want to apply a transition on the initial render of a node, you can add the appearattribute:

可以通过 appear 特性设置节点在初始渲染的过渡

By default, this will use the transitions specified for entering and leaving. If you’d like however, you can also specify custom CSS classes:

这里默认和进入/离开过渡一样，同样也可以自定义 CSS 类名。

In the example above, either appear attribute or v-on:appear hook will cause an appear transition.

在上面的例子中，无论是 appear 特性还是 v-on:appear 钩子都会生成初始渲染过渡。

Transitioning Between Elements 多个元素的过渡

We discuss transitioning between components later, but you can also transition between raw elements using v-if/v-else. One of the most common two-element transitions is between a list container and a message describing an empty list:

我们之后讨论多个组件的过渡，对于原生标签可以使用 v-if/v-else 。最常见的多标签过渡是一个列表和描述这个列表为空消息的元素：

This works well, but there’s one caveat to be aware of:

可以这样使用，但是有一点需要注意：

When toggling between elements that have the same tag name, you must tell Vue that they are distinct elements by giving them unique key attributes. Otherwise, Vue’s compiler will only replace the content of the element for efficiency. Even when technically unnecessary though, it’s considered good practice to always key multiple items within a <transition>component.

当有相同标签名的元素切换时，需要通过 key 特性设置唯一的值来标记以让 Vue 区分它们，否则 Vue 为了效率只会替换相同标签内部的内容。即使在技术上没有必要，给在 <transition> 组件中的多个元素设置 key 是一个更好的实践。

In these cases, you can also use the key attribute to transition between different states of the same element. Instead of using v-if and v-else, the above example could be rewritten as:

在一些场景中，也可以通过给同一个元素的 key 特性设置不同的状态来代替 v-if和 v-else，上面的例子可以重写为：

It’s actually possible to transition between any number of elements, either by using multiple v-ifs or binding a single element to a dynamic property. For example:

使用多个 v-if 的多个元素的过渡可以重写为绑定了动态属性的单个元素过渡。例如：

Which could also be written as: 可以重写为：

Transition Modes 过渡模式

There’s still one problem though. Try clicking the button below:

这里还有一个问题，试着点击下面的按钮：

As it’s transitioning between the “on” button and the “off” button, both buttons are rendered - one transitioning out while the other transitions in. This is the default behavior of <transition> - entering and leaving happens simultaneously.

在 “on” 按钮和 “off” 按钮的过渡中，两个按钮都被重绘了，一个离开过渡的时候另一个开始进入过渡。这是 <transition> 的默认行为 - 进入和离开同时发生。

Sometimes this works great, like when transitioning items are absolutely positioned on top of each other:

在元素绝对定位在彼此之上的时候运行正常：

And then maybe also translated so that they look like slide transitions:

然后，我们加上 translate 让它们运动像滑动过渡：

Simultaneous entering and leaving transitions aren’t always desirable though, so Vue offers some alternative transition modes:

同时生效的进入和离开的过渡不能满足所有要求，所以 Vue 提供了 过渡模式

in-out: New element transitions in first, then when complete, the current element transitions out.

in-out：新元素先进行过渡，完成之后当前元素过渡离开。

out-in: Current element transitions out first, then when complete, the new element transitions in.

out-in：当前元素先进行过渡，完成之后新元素过渡进入。

Now let’s update the transition for our on/off buttons with out-in:

用 out-in 重写之前的开关按钮过渡：

With one attribute addition, we’ve fixed that original transition without having to add any special styling.

只用添加一个简单的特性，就解决了之前的过渡问题而无需任何额外的代码。

The in-out mode isn’t used as often, but can sometimes be useful for a slightly different transition effect. Let’s try combining it with the slide-fade transition we worked on earlier:

in-out 模式不是经常用到，但对于一些稍微不同的过渡效果还是有用的。  
将之前滑动淡出的例子结合：

Pretty cool, right? 很酷吧？

Transitioning Between Components 多个组件的过渡

Transitioning between components is even simpler - we don’t even need the key attribute. Instead, we wrap a dynamic component:

多个组件的过渡简单很多 - 我们不需要使用 key 特性。相反，我们只需要使用动态组件：

List Transitions 列表过渡

So far, we’ve managed transitions for:

目前为止，关于过渡我们已经讲到：

Individual nodes单个节点

Multiple nodes where only 1 is rendered at a time同一时间渲染多个节点中的一个

So what about for when we have a whole list of items we want to render simultaneously, for example with v-for? In this case, we’ll use the <transition-group> component. Before we dive into an example though, there are a few things that are important to know about this component:

那么怎么同时渲染整个列表，比如使用 v-for ？在这种场景中，使用 <transition-group> 组件。在我们深入例子之前，先了解关于这个组件的几个特点：

Unlike <transition>, it renders an actual element: a <span> by default. You can change the element that’s rendered with the tag attribute.

不同于 <transition>，它会以一个真实元素呈现：默认为一个 <span>。你也可以通过 tag 特性更换为其他元素。

Transition modes are not available, because we are no longer alternating between mutually exclusive elements.

过渡模式不可用，因为我们不再相互切换特有的元素。

Elements inside are always required to have a unique key attribute.

内部元素 总是需要 提供唯一的 key 属性值。

List Entering/Leaving Transitions 列表的进入/离开过渡

Now let’s dive into an example, transitioning entering and leaving using the same CSS classes we’ve used previously:

现在让我们由一个简单的例子深入，进入和离开的过渡使用之前一样的 CSS 类名。

There’s one problem with this example. When you add or remove an item, the ones around it instantly snap into their new place instead of smoothly transitioning. We’ll fix that later.

这个例子有个问题，当添加和移除元素的时候，周围的元素会瞬间移动到他们的新布局的位置，而不是平滑的过渡，我们下面会解决这个问题。

List Move Transitions 列表的排序过渡

The <transition-group> component has another trick up its sleeve. It can not only animate entering and leaving, but also changes in position. The only new concept you need to know to use this feature is the addition of the v-move class, which is added when items are changing positions. Like the other classes, its prefix will match the value of a provided name attribute and you can also manually specify a class with the move-class attribute.

<transition-group> 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 v-move 特性，它会在元素的改变定位的过程中应用。像之前的类名一样，可以通过 name 属性来自定义前缀，也可以通过 move-class 属性手动设置。

This class is mostly useful for specifying the transition timing and easing curve, as you’ll see below:

v-move 对于设置过渡的切换时机和过渡曲线非常有用，你会看到如下的例子：

This might seem like magic, but under the hood, Vue is using an animation technique called FLIP to smoothly transition elements from their old position to their new position using transforms.

这个看起来很神奇，内部的实现，Vue 使用了一个叫 FLIP 简单的动画队列  
使用 transforms 将元素从之前的位置平滑过渡新的位置。

We can combine this technique with our previous implementation to animate every possible change to our list!

我们将之前实现的例子和这个技术结合，使我们列表的一切变动都会有动画过渡。

One important note is that these FLIP transitions do not work with elements set to display: inline. As an alternative, you can use display: inline-block or place elements in a flex context.

需要注意的是使用 FLIP 过渡的元素不能设置为 display: inline 。作为替代方案，可以设置为 display: inline-block 或者放置于 flex 中

These FLIP animations are also not limited to a single axis. Items in a multidimensional grid can be transitioned too:

FLIP 动画不仅可以实现单列过渡，多维网格也同样可以过渡：

Staggering List Transitions 列表的交错过渡

By communicating with JavaScript transitions through data attributes, it’s also possible to stagger transitions in a list:

通过 data 属性与 JavaScript 通信 ，就可以实现列表的交错过渡：

Reusable Transitions 可复用的过渡

Transitions can be reused through Vue’s component system. To create a reusable transition, all you have to do is place a <transition> or <transition-group> component at the root, then pass any children into the transition component.

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将 <transition> 或者 <transition-group> 作为根组件，然后将任何子组件放置在其中就可以了。

Here’s an example using a template component:

使用 template 的简单例子：

And functional components are especially well-suited to this task:

函数式组件更适合完成这个任务：

Dynamic Transitions 动态过渡

Yes, even transitions in Vue are data-driven! The most basic example of a dynamic transition binds the name attribute to a dynamic property.

在 Vue 中即使是过渡也是数据驱动的！动态过渡最基本的例子是通过 name 特性来绑定动态值。

This can be useful when you’ve defined CSS transitions/animations using Vue’s transition class conventions and want to switch between them.

当你想用 Vue 的过渡系统来定义的 CSS 过渡/动画 在不同过渡间切换会非常有用。

Really though, any transition attribute can be dynamically bound. And it’s not only attributes. Since event hooks are methods, they have access to any data in the context. That means depending on the state of your component, your JavaScript transitions can behave differently.

所有过渡特性都可以动态绑定，但我们不仅仅只有特性可以利用，还可以通过事件钩子获取上下文中的所有数据，因为事件钩子都是方法。这意味着，根据组件的状态不同，你的 JavaScript 过渡会有不同的表现。

Finally, the ultimate way of creating dynamic transitions is through components that accept props to change the nature of the transition(s) to be used. It may sound cheesy, but the only limit really is your imagination.

最后，创建动态过渡的最终方案是组件通过接受 props 来动态修改之前的过渡。一句老话，唯一的限制是你的想象力。

## State Transitions状态过渡

Vue’s transition system offers many simple ways to animate entering, leaving, and lists, but what about animating your data itself? For example:

Vue 的过渡系统提供了非常多简单的方法设置进入、离开和列表的动效。那么对于数据元素本身的动效呢，比如：

numbers and calculations数字和运算

colors displayed颜色的显示

the positions of SVG nodes SVG 节点的位置

the sizes and other properties of elements元素的大小和其他的属性

All of these are either already stored as raw numbers or can be converted into numbers. Once we do that, we can animate these state changes using 3rd-party libraries to tween state, in combination with Vue’s reactivity and component systems.

这些数据要么本身就以数值形式存储，要么可以转换为数值。有了这些数值后，我们就可以结合 Vue 的响应式和组件系统，使用第三方库来实现切换元素的过渡状态。

Animating State with Watchers 状态动画与侦听器

Watchers allow us to animate changes of any numerical property into another property. That may sound complicated in the abstract, so let’s dive into an example using GreenSock:

通过侦听器我们能监听到任何数值属性的数值更新。可能听起来很抽象，所以让我们先来看看使用 GreenSock 一个例子：

When you update the number, the change is animated below the input. This makes for a nice demo, but what about something that isn’t directly stored as a number, like any valid CSS color for example? Here’s how we could accomplish this with Tween.js and Color.js:

当你把数值更新时，就会触发动画。这个是一个不错的演示，但是对于不能直接像数字一样存储的值，比如 CSS 中的 color 的值，通过下面的例子我们来通过 Tween.js 和 Color.js 实现一个例子：

Dynamic State Transitions 动态状态过渡

As with Vue’s transition components, the data backing state transitions can be updated in real time, which is especially useful for prototyping! Even using a simple SVG polygon, you can achieve many effects that would be difficult to conceive of until you’ve played with the variables a little.

就像 Vue 的过渡组件一样，数据背后状态过渡会实时更新，这对于原型设计十分有用。当你修改一些变量，即使是一个简单的 SVG 多边形也可实现很多难以想象的效果。

Organizing Transitions into Components 把过渡放到组件里

Managing many state transitions can quickly increase the complexity of a Vue instance or component. Fortunately, many animations can be extracted out into dedicated child components. Let’s do this with the animated integer from our earlier example:

管理太多的状态过渡会很快的增加 Vue 实例或者组件的复杂性，幸好很多的动画可以提取到专用的子组件。

Within child components, we can use any combination of transition strategies that have been covered on this page, along with those offered by Vue’s built-in transition system. Together, there are very few limits to what can be accomplished.

我们能在组件中结合使用这一节讲到各种过渡策略和 Vue 内建的过渡系统。总之，对于完成各种过渡动效几乎没有阻碍。

Bringing Designs to Life 赋予设计以生命

To animate, by one definition, means to bring to life. Unfortunately, when designers create icons, logos, and mascots, they’re usually delivered as images or static SVGs. So although GitHub’s octocat, Twitter’s bird, and many other logos resemble living creatures, they don’t really seem alive.

只要一个动画，就可以带来生命。不幸的是，当设计师创建图标、logo 和吉祥物的时候，他们交付的通常都是图片或静态的 SVG。所以，虽然 GitHub 的章鱼猫、Twitter 的小鸟以及其它许多 logo 类似于生灵，它们看上去实际上并不是活着的。

Vue can help. Since SVGs are just data, we only need examples of what these creatures look like when excited, thinking, or alarmed. Then Vue can help transition between these states, making your welcome pages, loading indicators, and notifications more emotionally compelling.

Vue 可以帮到你。因为 SVG 的本质是数据，我们只需要这些动物兴奋、思考或警戒的样例。然后 Vue 就可以辅助完成这几种状态之间的过渡动画，来制作你的欢迎页面、加载指示、以及更加带有情感的提示。

Sarah Drasner demonstrates this in the demo below, using a combination of timed and interactivity-driven state changes:

Sarah Drasner 展示了下面这个 demo，这个 demo 结合了时间和交互相关的状态改变：

# Reusability & Composition

## Mixins混入

Basics 基础

Mixins are a flexible way to distribute reusable functionalities for Vue components. A mixin object can contain any component options. When a component uses a mixin, all options in the mixin will be “mixed” into the component’s own options.

混入 (mixin) 提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

Option Merging 选项合并

When a mixin and the component itself contain overlapping options, they will be “merged” using appropriate strategies.

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进行“合并”。

For example, data objects undergo a recursive merge, with the component’s data taking priority in cases of conflicts.

比如，数据对象在内部会进行递归合并，并在发生冲突时以组件数据优先。

Hook functions with the same name are merged into an array so that all of them will be called. Mixin hooks will be called before the component’s own hooks.

同名钩子函数将合并为一个数组，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。

Options that expect object values, for example methods, components and directives, will be merged into the same object. The component’s options will take priority when there are conflicting keys in these objects:

值为对象的选项，例如 methods、components 和 directives，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对。

Note that the same merge strategies are used in Vue.extend().

注意：Vue.extend() 也使用同样的策略进行合并。

Global Mixin 全局混入

You can also apply a mixin globally. Use with caution! Once you apply a mixin globally, it will affect every Vue instance created afterwards. When used properly, this can be used to inject processing logic for custom options:

混入也可以进行全局注册。使用时格外小心！一旦使用全局混入，它将影响每一个之后创建的 Vue 实例。使用恰当时，这可以用来为自定义选项注入处理逻辑。

Use global mixins sparsely and carefully, because it affects every single Vue instance created, including third party components. In most cases, you should only use it for custom option handling like demonstrated in the example above. It’s also a good idea to ship them as Pluginsto avoid duplicate application.

请谨慎使用全局混入，因为它会影响每个单独创建的 Vue 实例 (包括第三方组件)。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。推荐将其作为插件发布，以避免重复应用混入。

Custom Option Merge Strategies 自定义选项合并策略

When custom options are merged, they use the default strategy which overwrites the existing value. If you want a custom option to be merged using custom logic, you need to attach a function to Vue.config.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：

For most object-based options, you can use the same strategy used by methods:

对于多数值为对象的选项，可以使用与 methods 相同的合并策略：

A more advanced example can be found on Vuex‘s 1.x merging strategy:

可以在 Vuex 1.x 的混入策略里找到一个更高级的例子：

## Custom Directives自定义指令

Intro 简介

In addition to the default set of directives shipped in core (v-model and v-show), Vue also allows you to register your own custom directives. Note that in Vue 2.0, the primary form of code reuse and abstraction is components - however there may be cases where you need some low-level DOM access on plain elements, and this is where custom directives would still be useful. An example would be focusing on an input element, like this one:

除了核心功能默认内置的指令 (v-model 和 v-show)，Vue 也允许注册自定义指令。注意，在 Vue2.0 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通 DOM 元素进行底层操作，这时候就会用到自定义指令。举个聚焦输入框的例子，如下：

When the page loads, that element gains focus (note: autofocus doesn’t work on mobile Safari). In fact, if you haven’t clicked on anything else since visiting this page, the input above should be focused now. Now let’s build the directive that accomplishes this:

当页面加载时，该元素将获得焦点 (注意：autofocus 在移动版 Safari 上不工作)。事实上，只要你在打开这个页面后还没点击过任何内容，这个输入框就应当还是处于聚焦状态。现在让我们用指令来实现这个功能：

If you want to register a directive locally instead, components also accept a directives option:

如果想注册局部指令，组件中也接受一个 directives 的选项：

Then in a template, you can use the new v-focus attribute on any element, like this:

然后你可以在模板中任何元素上使用新的 v-focus 属性，如下：

Hook Functions 钩子函数

A directive definition object can provide several hook functions (all optional):

一个指令定义对象可以提供如下几个钩子函数 (均为可选)：

bind: called only once, when the directive is first bound to the element. This is where you can do one-time setup work.

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: called when the bound element has been inserted into its parent node (this only guarantees parent node presence, not necessarily in-document).

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: called after the containing component’s VNode has updated, but possibly before its children have updated. The directive’s value may or may not have changed, but you can skip unnecessary updates by comparing the binding’s current and old values (see below on hook arguments).

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)。

We’ll cover VNodes in more detail later, when we discuss render functions.

我们会在稍后讨论渲染函数时介绍更多 VNodes 的细节。

componentUpdated: called after the containing component’s VNode and the VNodes of its children have updated.

componentUpdated：指令所在组件的 VNode 及其子 VNode 全部更新后调用。

unbind: called only once, when the directive is unbound from the element.

unbind：只调用一次，指令与元素解绑时调用。

We’ll explore the arguments passed into these hooks (i.e. el, binding, vnode, and oldVnode) in the next section.

接下来我们来看一下钩子函数的参数 (即 el、binding、vnode 和 oldVnode)。

Directive Hook Arguments 钩子函数参数

Directive hooks are passed these arguments:

指令钩子函数会被传入以下参数：

el: The element the directive is bound to. This can be used to directly manipulate the DOM. 指令所绑定的元素，可以用来直接操作 DOM 。

binding: An object containing the following properties. 一个对象，包含以下属性：

name: The name of the directive, without the v- prefix. 指令名，不包括 v- 前缀。

value: The value passed to the directive. For example in v-my-directive="1 + 1", the value would be 2. 指令的绑定值，例如：v-my-directive="1 + 1" 中，绑定值为 2。

oldValue: The previous value, only available in update and componentUpdated. It is available whether or not the value has changed. 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: The expression of the binding as a string. For example in v-my-directive="1 + 1", the expression would be "1 + 1".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: The argument passed to the directive, if any. For example in v-my-directive:foo, the arg would be "foo".传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。

modifiers: An object containing modifiers, if any. For example in v-my-directive.foo.bar, the modifiers object would be { foo: true, bar: true }.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: The virtual node produced by Vue’s compiler. See the VNode API for full details. Vue 编译生成的虚拟节点。移步 VNode API 来了解更多详情。

oldVnode: The previous virtual node, only available in the update and componentUpdatedhooks. 上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

Apart from el, you should treat these arguments as read-only and never modify them. If you need to share information across hooks, it is recommended to do so through element’s dataset.

除了 el 之外，其它参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 dataset 来进行。

An example of a custom directive using some of these properties:

这是一个使用了这些属性的自定义钩子样例：

Directive arguments can be dynamic. For example, in v-mydirective:argument=[dataproperty], argument is the string value assigned to the arg property in your directive hook binding parameter and dataproperty is a reference to a data property on your component instance assigned to the value property in the same binding parameter. As directive hooks are invoked, the value property of the binding parameter will dynamically change based on the value of dataproperty.

指令的参数可以是动态的。例如，在 v-mydirective:argument=[dataproperty]中，argument 是一个赋值给这个指令钩子 binding 参数中的 arg property 的字符串，同时 dataproperty 是一个引用到组件实例上并赋值给同一个 binding 参数中的 value property 的 data property。当指令钩子被调用的时候，binding 参数中的 valueproperty 会基于 dataproperty 的值动态改变。

An example of a custom directive using a dynamic argument:

一个使用了动态参数的自定义指令的例子如下：

Function Shorthand 函数简写

In many cases, you may want the same behavior on bind and update, but don’t care about the other hooks. For example:

在很多时候，你可能想在 bind 和 update 时触发相同行为，而不关心其它的钩子。比如这样写:

Object Literals 对象字面量

If your directive needs multiple values, you can also pass in a JavaScript object literal. Remember, directives can take any valid JavaScript expression.

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法的 JavaScript 表达式。

## Render Functions & JSX渲染函数 & JSX

Basics 基础

Vue recommends using templates to build your HTML in the vast majority of cases. There are situations however, where you really need the full programmatic power of JavaScript. That’s where you can use the render function, a closer-to-the-compiler alternative to templates.

Vue 推荐在绝大多数情况下使用模板来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力。这时你可以用渲染函数，它比模板更接近编译器。

Let’s dive into a simple example where a render function would be practical. Say you want to generate anchored headings:

让我们深入一个简单的例子，这个例子里 render 函数很实用。假设我们要生成一些带锚点的标题：

For the HTML above, you decide you want this component interface:

对于上面的 HTML，你决定这样定义组件接口：

When you get started with a component that only generates a heading based on the level prop, you quickly arrive at this:

当开始写一个只能通过 level prop 动态生成标题 (heading) 的组件时，你可能很快想到这样实现：

That template doesn’t feel great. It’s not only verbose, but we’re duplicating <slot></slot> for every heading level and will have to do the same when we add the anchor element.

这里用模板并不是最好的选择：不但代码冗长，而且在每一个级别的标题中重复书写了 <slot></slot>，在要插入锚点元素时还要再次重复。

While templates work great for most components, it’s clear that this isn’t one of them. So let’s try rewriting it with a render function:

虽然模板在大多数组件中都非常好用，但是显然在这里它就不合适了。那么，我们来尝试使用 render 函数重写上面的例子：

Much simpler! Sort of. The code is shorter, but also requires greater familiarity with Vue instance properties. In this case, you have to know that when you pass children without a v-slot directive into a component, like the Hello world! inside of anchored-heading, those children are stored on the component instance at $slots.default. If you haven’t already, it’s recommended to read through the instance properties API before diving into render functions.

看起来简单多了！这样代码精简很多，但是需要非常熟悉 Vue 的实例属性。在这个例子中，你需要知道，向组件中传递不带 v-slot 指令的子节点时，比如 anchored-heading 中的 Hello world!，这些子节点被存储在组件实例中的 $slots.default 中。如果你还不了解，在深入渲染函数之前推荐阅读实例属性 API。

Nodes, Trees, and the Virtual DOM 节点、树以及虚拟 DOM

Before we dive into render functions, it’s important to know a little about how browsers work. Take this HTML for example:

在深入渲染函数之前，了解一些浏览器的工作原理是很重要的。以下面这段 HTML 为例：

When a browser reads this code, it builds a tree of “DOM nodes” to help it keep track of everything, just as you might build a family tree to keep track of your extended family.

当浏览器读到这些代码时，它会建立一个“DOM 节点”树来保持追踪所有内容，如同你会画一张家谱树来追踪家庭成员的发展一样。

The tree of DOM nodes for the HTML above looks like this:

上述 HTML 对应的 DOM 节点树如下图所示：

Every element is a node. Every piece of text is a node. Even comments are nodes! A node is just a piece of the page. And as in a family tree, each node can have children (i.e. each piece can contain other pieces).

每个元素都是一个节点。每段文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点 (也就是说每个部分可以包含其它的一些部分)。

Updating all these nodes efficiently can be difficult, but thankfully, you never have to do it manually. Instead, you tell Vue what HTML you want on the page, in a template:

高效地更新所有这些节点会是比较困难的，不过所幸你不必手动完成这个工作。你只需要告诉 Vue 你希望页面上的 HTML 是什么，这可以是在一个模板里：

Or a render function:

或者一个渲染函数里：

And in both cases, Vue automatically keeps the page updated, even when blogTitle changes.

在这两种情况下，Vue 都会自动保持页面的更新，即便 blogTitle 发生了改变。

The Virtual DOM 虚拟 DOM

Vue accomplishes this by building a virtual DOM to keep track of the changes it needs to make to the real DOM. Taking a closer look at this line:

Vue 通过建立一个虚拟 DOM 来追踪自己要如何改变真实 DOM。请仔细看这行代码：

What is createElement actually returning? It’s not exactly a real DOM element. It could perhaps more accurately be named createNodeDescription, as it contains information describing to Vue what kind of node it should render on the page, including descriptions of any child nodes. We call this node description a “virtual node”, usually abbreviated to VNode. “Virtual DOM” is what we call the entire tree of VNodes, built by a tree of Vue components.

createElement 到底会返回什么呢？其实不是一个实际的 DOM 元素。它更准确的名字可能是 createNodeDescription，因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点，包括及其子节点的描述信息。我们把这样的节点描述为“虚拟节点 (virtual node)”，也常简写它为“VNode”。“虚拟 DOM”是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼。

createElement Arguments createElement 参数

The next thing you’ll have to become familiar with is how to use template features in the createElement function. Here are the arguments that createElement accepts:

接下来你需要熟悉的是如何在 createElement 函数中使用模板中的那些功能。这里是 createElement 接受的参数：

The Data Object In-Depth 深入数据对象

One thing to note: similar to how v-bind:class and v-bind:style have special treatment in templates, they have their own top-level fields in VNode data objects. This object also allows you to bind normal HTML attributes as well as DOM properties such as innerHTML (this would replace the v-html directive):

有一点要注意：正如 v-bind:class 和 v-bind:style 在模板语法中会被特别对待一样，它们在 VNode 数据对象中也有对应的顶层字段。该对象也允许你绑定普通的 HTML 特性，也允许绑定如 innerHTML 这样的 DOM 属性 (这会覆盖 v-html 指令)。

Complete Example 完整示例

With this knowledge, we can now finish the component we started:

有了这些知识，我们现在可以完成我们最开始想实现的组件：

Constraints 约束

VNodes Must Be Unique VNode 必须唯一

All VNodes in the component tree must be unique. That means the following render function is invalid:

组件树中的所有 VNode 必须是唯一的。这意味着，下面的渲染函数是不合法的：

If you really want to duplicate the same element/component many times, you can do so with a factory function. For example, the following render function is a perfectly valid way of rendering 20 identical paragraphs:

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现。例如，下面这渲染函数用完全合法的方式渲染了 20 个相同的段落：

Replacing Template Features with Plain JavaScript 使用 JavaScript 代替模板功能

Wherever something can be easily accomplished in plain JavaScript, Vue render functions do not provide a proprietary alternative. For example, in a template using v-if and v-for:

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的渲染函数就不会提供专有的替代方法。比如，在模板中使用的 v-if 和 v-for：

This could be rewritten with JavaScript’s if/else and map in a render function:

这些都可以在渲染函数中用 JavaScript 的 if/else 和 map 来重写：

There is no direct v-model counterpart in render functions - you will have to implement the logic yourself:

渲染函数中没有与 v-model 的直接对应——你必须自己实现相应的逻辑：

This is the cost of going lower-level, but it also gives you much more control over the interaction details compared to v-model.

这就是深入底层的代价，但与 v-model 相比，这可以让你更好地控制交互细节。

Event & Key Modifiers 事件 & 按键修饰符

For the .passive, .capture and .once event modifiers, Vue offers prefixes that can be used with on:

对于 .passive、.capture 和 .once 这些事件修饰符, Vue 提供了相应的前缀可以用于 on：

For all other event and key modifiers, no proprietary prefix is necessary, because you can use event methods in the handler:

对于所有其它的修饰符，私有前缀都不是必须的，因为你可以在事件处理函数中使用事件方法：

Here’s an example with all of these modifiers used together:

这里是一个使用所有修饰符的例子：

Slots 插槽

You can access static slot contents as Arrays of VNodes from this.$slots:

你可以通过 this.$slots 访问静态插槽的内容，每个插槽都是一个 VNode 数组：

And access scoped slots as functions that return VNodes from this.$scopedSlots:

也可以通过 this.$scopedSlots 访问作用域插槽，每个作用域插槽都是一个返回若干 VNode 的函数：

To pass scoped slots to a child component using render functions, use the scopedSlots field in VNode data:

如果要用渲染函数向子组件中传递作用域插槽，可以利用 VNode 数据对象中的 scopedSlots 字段：

If you’re writing a lot of render functions, it might feel painful to write something like this:

如果你写了很多 render 函数，可能会觉得下面这样的代码写起来很痛苦：

Especially when the template version is so simple in comparison:

特别是对应的模板如此简单的情况下：

That’s why there’s a Babel plugin to use JSX with Vue, getting us back to a syntax that’s closer to templates:

这就是为什么会有一个 Babel 插件，用于在 Vue 中使用 JSX 语法，它可以让我们回到更接近于模板的语法上。

Aliasing createElement to h is a common convention you’ll see in the Vue ecosystem and is actually required for JSX. Starting with version 3.4.0 of the Babel plugin for Vue, we automatically inject const h = this.$createElement in any method and getter (not functions or arrow functions), declared in ES2015 syntax that has JSX, so you can drop the (h) parameter. With prior versions of the plugin, your app would throw an error if h was not available in the scope.

将 h 作为 createElement 的别名是 Vue 生态系统中的一个通用惯例，实际上也是 JSX 所要求的。从 Vue 的 Babel 插件的 3.4.0 版本开始，我们会在以 ES2015 语法声明的含有 JSX 的任何方法和 getter 中 (不是函数或箭头函数中) 自动注入 const h = this.$createElement，这样你就可以去掉 (h) 参数了。对于更早版本的插件，如果 h 在当前作用域中不可用，应用会抛错。

For more on how JSX maps to JavaScript, see the usage docs.

要了解更多关于 JSX 如何映射到 JavaScript，请阅读使用文档。

Functional Components 函数式组件

The anchored heading component we created earlier is relatively simple. It doesn’t manage any state, watch any state passed to it, and it has no lifecycle methods. Really, it’s only a function with some props.

之前创建的锚点标题组件是比较简单，没有管理任何状态，也没有监听任何传递给它的状态，也没有生命周期方法。实际上，它只是一个接受一些 prop 的函数。

In cases like this, we can mark components as functional, which means that they’re stateless (no reactive data) and instanceless (no this context).  
在这样的场景下，我们可以将组件标记为 functional，这意味它无状态 (没有响应式数据)，也没有实例 (没有 this 上下文)。

A functional component looks like this:

一个函数式组件就像这样：

Note: in versions before 2.3.0, the props option is required if you wish to accept props in a functional component. In 2.3.0+ you can omit the props option and all attributes found on the component node will be implicitly extracted as props.

注意：在 2.3.0 之前的版本中，如果一个函数式组件想要接收 prop，则 props 选项是必须的。在 2.3.0 或以上的版本中，你可以省略 props 选项，所有组件上的特性都会被自动隐式解析为 prop。

In 2.5.0+, if you are using single-file components, template-based functional components can be declared with:

在 2.5.0 及以上版本中，如果你使用了单文件组件，那么基于模板的函数式组件可以这样声明：

Everything the component needs is passed through context, which is an object containing:

组件需要的一切都是通过 context 参数传递，它是一个包括如下字段的对象：

props: An object of the provided props提供所有 prop 的对象

children: An array of the VNode children  VNode 子节点的数组

slots: A function returning a slots object一个函数，返回了包含所有插槽的对象

scopedSlots: (2.6.0+) An object that exposes passed-in scoped slots. Also exposes normal slots as functions. (2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。

data: The entire data object, passed to the component as the 2nd argument of createElement传递给组件的整个数据对象，作为 createElement 的第二个参数传入组件

parent: A reference to the parent component对父组件的引用

listeners: (2.3.0+) An object containing parent-registered event listeners. This is an alias to data.on  (2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器的对象。这是 data.on 的一个别名。

injections: (2.3.0+) if using the inject option, this will contain resolved injections. (2.3.0+) 如果使用了 inject 选项，则该对象包含了应当被注入的属性。

After adding functional: true, updating the render function of our anchored heading component would require adding the context argument, updating this.$slots.default to context.children, then updating this.level to context.props.level.

在添加 functional: true 之后，需要更新我们的锚点标题组件的渲染函数，为其增加 context 参数，并将 this.$slots.default 更新为 context.children，然后将 this.level 更新为 context.props.level。

Since functional components are just functions, they’re much cheaper to render.

因为函数式组件只是函数，所以渲染开销也低很多。

They’re also very useful as wrapper components. For example, when you need to:

在作为包装组件时它们也同样非常有用。比如，当你需要做这些时：

Programmatically choose one of several other components to delegate to

程序化地在多个组件中选择一个来代为渲染；

Manipulate children, props, or data before passing them on to a child component

在将 children、props、data 传递给子组件之前操作它们。

Here’s an example of a smart-list component that delegates to more specific components, depending on the props passed to it:

下面是一个 smart-list 组件的例子，它能根据传入 prop 的值来代为渲染更具体的组件：

Passing Attributes and Events to Child Elements/Components 向子元素或子组件传递特性和事件

On normal components, attributes not defined as props are automatically added to the root element of the component, replacing or intelligently merging with any existing attributes of the same name.

在普通组件中，没有被定义为 prop 的特性会自动添加到组件的根元素上，将已有的同名特性进行替换或与其进行智能合并。

Functional components, however, require you to explicitly define this behavior:

然而函数式组件要求你显式定义该行为：

By passing context.data as the second argument to createElement, we are passing down any attributes or event listeners used on my-functional-button. It’s so transparent, in fact, that events don’t even require the .native modifier.

通过向 createElement 传入 context.data 作为第二个参数，我们就把 my-functional-button 上面所有的特性和事件监听器都传递下去了。事实上这是非常透明的，以至于那些事件甚至并不要求 .native 修饰符。

If you are using template-based functional components, you will also have to manually add attributes and listeners. Since we have access to the individual context contents, we can use data.attrs to pass along any HTML attributes and listeners (the alias for data.on) to pass along any event listeners.

如果你使用基于模板的函数式组件，那么你还需要手动添加特性和监听器。因为我们可以访问到其独立的上下文内容，所以我们可以使用 data.attrs 传递任何 HTML 特性，也可以使用 listeners (即 data.on 的别名) 传递任何事件监听器。

slots() vs children slots() 和 children 对比

You may wonder why we need both slots() and children. Wouldn’t slots().default be the same as children? In some cases, yes - but what if you have a functional component with the following children?

你可能想知道为什么同时需要 slots() 和 children。slots().default 不是和 children 类似的吗？在一些场景中，是这样——但如果是如下的带有子节点的函数式组件呢？

For this component, children will give you both paragraphs, slots().default will give you only the second, and slots().foo will give you only the first. Having both children and slots()therefore allows you to choose whether this component knows about a slot system or perhaps delegates that responsibility to another component by passing along children.

对于这个组件，children 会给你两个段落标签，而 slots().default 只会传递第二个匿名段落标签，slots().foo 会传递第一个具名段落标签。同时拥有 children和 slots()，因此你可以选择让组件感知某个插槽机制，还是简单地通过传递 children，移交给其它组件去处理。

Template Compilation 模板编译

You may be interested to know that Vue’s templates actually compile to render functions. This is an implementation detail you usually don’t need to know about, but if you’d like to see how specific template features are compiled, you may find it interesting. Below is a little demo using Vue.compileto live-compile a template string: 你可能会有兴趣知道，Vue 的模板实际上被编译成了渲染函数。这是一个实现细节，通常不需要关心。但如果你想看看模板的功能具体是怎样被编译的，可能会发现会非常有意思。下面是一个使用 Vue.compile 来实时编译模板字符串的简单示例：

## Plugins插件

Plugins usually add global-level functionality to Vue. There is no strictly defined scope for a plugin - there are typically several types of plugins:

插件通常用来为 Vue 添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

Add some global methods or properties. e.g. vue-custom-element添加全局方法或者属性。如: vue-custom-element

Add one or more global assets: directives/filters/transitions etc. e.g. vue-touch添加全局资源：指令/过滤器/过渡等。如 vue-touch

Add some component options by global mixin. e.g. vue-router通过全局混入来添加一些组件选项。如 vue-router

Add some Vue instance methods by attaching them to Vue.prototype. 添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。

A library that provides an API of its own, while at the same time injecting some combination of the above. e.g. vue-router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

Using a Plugin 使用插件

Use plugins by calling the Vue.use() global method. This has to be done before you start your app by calling new Vue():

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：

You can optionally pass in some options:

也可以传入一个可选的选项对象：

Vue.use automatically prevents you from using the same plugin more than once, so calling it multiple times on the same plugin will install the plugin only once.

Vue.use 会自动阻止多次注册相同插件，届时即使多次调用也只会注册一次该插件。

Some plugins provided by Vue.js official plugins such as vue-router automatically calls Vue.use()if Vue is available as a global variable. However in a module environment such as CommonJS, you always need to call Vue.use() explicitly:

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()。然而在像 CommonJS 这样的模块环境中，你应该始终显式地调用 Vue.use()：

Checkout awesome-vue for a huge collection of community-contributed plugins and libraries.

awesome-vue 集合了大量由社区贡献的插件和库。

Writing a Plugin 开发插件

A Vue.js plugin should expose an install method. The method will be called with the Vueconstructor as the first argument, along with possible options:

Vue.js 的插件应该暴露一个 install 方法。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象：

## Filters过滤器

Vue.js allows you to define filters that can be used to apply common text formatting. Filters are usable in two places: mustache interpolations and v-bind expressions (the latter supported in 2.1.0+). Filters should be appended to the end of the JavaScript expression, denoted by the “pipe” symbol:

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 v-bind 表达式 (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示：

You can define local filters in a component’s options:

你可以在一个组件的选项中定义本地的过滤器：

or define a filter globally before creating the Vue instance:

或者在创建 Vue 实例之前全局定义过滤器：

Below is an example of our capitalize filter being used:

下面这个例子用到了 capitalize 过滤器：

The filter’s function always receives the expression’s value (the result of the former chain) as its first argument. In the above example, the capitalize filter function will receive the value of message as its argument.

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。在上述例子中，capitalize 过滤器函数将会收到 message 的值作为第一个参数。

Filters can be chained:

过滤器可以串联：

In this case, filterA, defined with a single argument, will receive the value of message, and then the filterB function will be called with the result of filterA passed into filterB‘s single argument.

在这个例子中，filterA 被定义为接收单个参数的过滤器函数，表达式 message 的值将作为参数传入到函数中。然后继续调用同样被定义为接收单个参数的过滤器函数 filterB，将 filterA 的结果传递到 filterB 中。

Filters are JavaScript functions, therefore they can take arguments:

过滤器是 JavaScript 函数，因此可以接收参数：

Here filterA is defined as a function taking three arguments. The value of message will be passed into the first argument. The plain string 'arg1' will be passed into the filterA as its second argument, and the value of expression arg2 will be evaluated and passed in as the third argument.

这里，filterA 被定义为接收三个参数的过滤器函数。其中 message 的值作为第一个参数，普通字符串 'arg1' 作为第二个参数，表达式 arg2 的值作为第三个参数。

# Tooling

## Single File Components单文件组件

Introduction 介绍

In many Vue projects, global components will be defined using Vue.component, followed by new Vue({ el: '#container' }) to target a container element in the body of every page.

在很多 Vue 项目中，我们使用 Vue.component 来定义全局组件，紧接着用 new Vue({ el: '#container '}) 在每个页面内指定一个容器元素。

This can work very well for small to medium-sized projects, where JavaScript is only used to enhance certain views. In more complex projects however, or when your frontend is entirely driven by JavaScript, these disadvantages become apparent:

这种方式在很多中小规模的项目中运作的很好，在这些项目里 JavaScript 只被用来加强特定的视图。但当在更复杂的项目中，或者你的前端完全由 JavaScript 驱动的时候，下面这些缺点将变得非常明显：

Global definitions force unique names for every component全局定义 (Global definitions) 强制要求每个 component 中的命名不得重复

String templates lack syntax highlighting and require ugly slashes for multiline HTML字符串模板 (String templates) 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 \

No CSS support means that while HTML and JavaScript are modularized into components, CSS is conspicuously left out不支持 CSS (No CSS support) 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏

No build step restricts us to HTML and ES5 JavaScript, rather than preprocessors like Pug (formerly Jade) and Babel没有构建步骤 (No build step) 限制只能使用 HTML 和 ES5 JavaScript, 而不能使用预处理器，如 Pug (formerly Jade) 和 Babel

All of these are solved by single-file components with a .vue extension, made possible with build tools such as Webpack or Browserify.

文件扩展名为 .vue 的 single-file components(单文件组件) 为以上所有问题提供了解决方法，并且还可以使用 webpack 或 Browserify 等构建工具。

Here’s an example of a file we’ll call Hello.vue:

这是一个文件名为 Hello.vue 的简单实例：

Now we get: 现在我们获得

Complete syntax highlighting 完整语法高亮

CommonJS modules CommonJS 模块

Component-scoped CSS 组件作用域的 CSS

As promised, we can also use preprocessors such as Pug, Babel (with ES2015 modules), and Stylus for cleaner and more feature-rich components.

正如我们说过的，我们可以使用预处理器来构建简洁和功能更丰富的组件，比如 Pug，Babel (with ES2015 modules)，和 Stylus。

These specific languages are only examples. You could as easily use Bublé, TypeScript, SCSS, PostCSS - or whatever other preprocessors that help you be productive. If using Webpack with vue-loader, it also has first-class support for CSS Modules.

这些特定的语言只是例子，你可以只是简单地使用 Babel，TypeScript，SCSS，PostCSS - 或者其他任何能够帮助你提高生产力的预处理器。如果搭配 vue-loader 使用 webpack，它也能为 CSS Modules 提供头等支持。

What About Separation of Concerns? 怎么看待关注点分离？

One important thing to note is that separation of concerns is not equal to separation of file types. In modern UI development, we have found that instead of dividing the codebase into three huge layers that interweave with one another, it makes much more sense to divide them into loosely-coupled components and compose them. Inside a component, its template, logic and styles are inherently coupled, and collocating them actually makes the component more cohesive and maintainable.

一个重要的事情值得注意，关注点分离不等于文件类型分离。在现代 UI 开发中，我们已经发现相比于把代码库分离成三个大的层次并将其相互交织起来，把它们划分为松散耦合的组件再将其组合起来更合理一些。在一个组件里，其模板、逻辑和样式是内部耦合的，并且把他们搭配在一起实际上使得组件更加内聚且更可维护。

Even if you don’t like the idea of Single-File Components, you can still leverage its hot-reloading and pre-compilation features by separating your JavaScript and CSS into separate files:

即便你不喜欢单文件组件，你仍然可以把 JavaScript、CSS 分离成独立的文件然后做到热重载和预编译。

Getting Started 起步

Example Sandbox 例子沙箱

If you want to dive right in and start playing with single-file components, check out this simple todo app on CodeSandbox.

如果你希望深入了解并开始使用单文件组件，请来 CodeSandbox 看看这个简单的 todo 应用。

For Users New to Module Build Systems in JavaScript

针对刚接触 JavaScript 模块开发系统的用户

With .vue components, we’re entering the realm of advanced JavaScript applications. That means learning to use a few additional tools if you haven’t already:

有了 .vue 组件，我们就进入了高级 JavaScript 应用领域。如果你没有准备好的话，意味着还需要学会使用一些附加的工具：

Node Package Manager (NPM): Read the Getting Started guide through section 10: Uninstalling global packages.

Node Package Manager (NPM)：阅读 Getting Started guide 直到 10: Uninstalling global packages章节。

Modern JavaScript with ES2015/16: Read through Babel’s Learn ES2015 guide. You don’t have to memorize every feature right now, but keep this page as a reference you can come back to.

Modern JavaScript with ES2015/16：阅读 Babel 的 Learn ES2015 guide。你不需要立刻记住每一个方法，但是你可以保留这个页面以便后期参考。

After you’ve taken a day to dive into these resources, we recommend checking out Vue CLI 3. Follow the instructions and you should have a Vue project with .vue components, ES2015, Webpack and hot-reloading in no time!

在你花一天时间了解这些资源之后，我们建议你参考 Vue CLI 3。只要遵循指示，你就能很快地运行一个带有 .vue 组件、ES2015、webpack 和热重载 (hot-reloading) 的 Vue 项目!

For Advanced Users 针对高级用户

The CLI takes care of most of the tooling configurations for you, but also allows fine-grained customization through its own config options.

CLI 会为你搞定大多数工具的配置问题，同时也支持细粒度自定义配置项。

In case you prefer setting up your own build setup from scratch, you will need to manually configure webpack with vue-loader. To learn more about webpack itself, check out their official docs and Webpack Academy.

有时你会想从零搭建你自己的构建工具，这时你需要通过 Vue Loader 手动配置 webpack。关于学习更多 webpack 的内容，请查阅其官方文档 和 Webpack Academy。

## Unit Testing单元测试

Vue CLI has built-in options for unit testing with Jest or Mocha that works out of the box. We also have the official Vue Test Utils which provides more detailed guidance for custom setups.

Vue CLI 拥有开箱即用的通过 Jest 或 Mocha 进行单元测试的内置选项。我们还有官方的 Vue Test Utils 提供更多详细的指引和自定义设置。

Simple Assertions 简单的断言

You don’t have to do anything special in your components to make them testable. Export the raw options:

你不必为了可测性在组件中做任何特殊的操作，导出原始设置就可以了：

Then import the component options along with Vue, and you can make many common assertions (here we are using Jasmine/Jest style expect assertions just as an example):

然后随着 Vue 导入组件的选项，你可以使用许多常见的断言 (这里我们使用的是 Jasmine/Jest 风格的 expect 断言作为示例)：

Writing Testable Components 编写可被测试的组件

A component’s render output is primarily determined by the props it receives. If a component’s render output solely depends on its props it becomes straightforward to test, similar to asserting the return value of a pure function with different arguments. Take a simplified example:

很多组件的渲染输出由它的 props 决定。事实上，如果一个组件的渲染输出完全取决于它的 props，那么它会让测试变得简单，就好像断言不同参数的纯函数的返回值。看下面这个例子：

You can assert its render output with different props using the propsData option:

你可以在不同的 props 中，通过 propsData 选项断言它的渲染输出：

Asserting Asynchronous Updates 断言异步更新

Since Vue performs DOM updates asynchronously, assertions on DOM updates resulting from state change will have to be made in a Vue.nextTick callback:

由于 Vue 进行 异步更新 DOM 的情况，一些依赖 DOM 更新结果的断言必须在 Vue.nextTick 回调中进行：

For more in-depth information on unit testing in Vue, check out Vue Test Utils and our cookbook entry about unit testing vue components.

关于更深入的 Vue 单元测试的内容，请移步 Vue Test Utils 以及我们关于 Vue 组件的单元测试的 cookbook 文章。

## TypeScript Support TypeScript 支持

Vue CLI provides built-in TypeScript tooling support. In our next major version of Vue (3.x), we are also planning to considerably improve our TypeScript support with built-in class based components API and TSX support.

Vue CLI 内置了 TypeScript 工具支持。在 Vue 的下一个大版本 (3.x) 中也计划了相当多的 TypeScript 支持改进，包括内置的基于 class 的组件 API 和 TSX 的支持。

Official Declaration in NPM Packages

发布为 NPM 包的官方声明文件

A static type system can help prevent many potential runtime errors, especially as applications grow. That’s why Vue ships with official type declarations for TypeScript - not only in Vue core, but also for vue-router and vuex as well.

静态类型系统能帮助你有效防止许多潜在的运行时错误，而且随着你的应用日渐丰满会更加显著。这就是为什么 Vue 不仅仅为 Vue core 提供了针对 TypeScript 的官方类型声明，还为 Vue Router 和 Vuex 也提供了相应的声明文件。

Since these are published on NPM, and the latest TypeScript knows how to resolve type declarations in NPM packages, this means when installed via NPM, you don’t need any additional tooling to use TypeScript with Vue.

而且，我们已经把它们发布到了 NPM，最新版本的 TypeScript 也知道该如何自己从 NPM 包里解析类型声明。这意味着只要你成功地通过 NPM 安装了，就不再需要任何额外的工具辅助，即可在 Vue 中使用 TypeScript 了。

Recommended Configuration 推荐配置

Note that you have to include strict: true (or at least noImplicitThis: true which is a part of strict flag) to leverage type checking of this in component methods otherwise it is always treated as any type.

注意你需要引入 strict: true (或者至少 noImplicitThis: true，这是 strict模式的一部分) 以利用组件方法中 this 的类型检查，否则它会始终被看作 any 类型。

See TypeScript compiler options docs for more details.

参阅 TypeScript 编译器选项文档 (英) 了解更多。

Development Tooling 开发工具链

Project Creation 工程创建

Vue CLI 3 can generate new projects that use TypeScript. To get started:

Vue CLI 3 可以使用 TypeScript 生成新工程。创建方式：

Editor Support 编辑器支持

For developing Vue applications with TypeScript, we strongly recommend using Visual Studio Code, which provides great out-of-the-box support for TypeScript. If you are using single-file components(SFCs), get the awesome Vetur extension, which provides TypeScript inference inside SFCs and many other great features.

要使用 TypeScript 开发 Vue 应用程序，我们强烈建议您使用 Visual Studio Code，它为 TypeScript 提供了极好的“开箱即用”支持。如果你正在使用单文件组件 (SFC), 可以安装提供 SFC 支持以及其他更多实用功能的 Vetur 插件。

WebStorm also provides out-of-the-box support for both TypeScript and Vue.

WebStorm 同样为 TypeScript 和 Vue 提供了“开箱即用”的支持。

Basic Usage 基本用法

To let TypeScript properly infer types inside Vue component options, you need to define components with Vue.component or Vue.extend:

要让 TypeScript 正确推断 Vue 组件选项中的类型，您需要使用 Vue.component 或 Vue.extend 定义组件：

Class-Style Vue Components 基于类的 Vue 组件

If you prefer a class-based API when declaring components, you can use the officially maintained vue-class-component decorator:

如果您在声明组件时更喜欢基于类的 API，则可以使用官方维护的 vue-class-component 装饰器：

Augmenting Types for Use with Plugins 增强类型以配合插件使用

Plugins may add to Vue’s global/instance properties and component options. In these cases, type declarations are needed to make plugins compile in TypeScript. Fortunately, there’s a TypeScript feature to augment existing types called module augmentation.

插件可以增加 Vue 的全局/实例属性和组件选项。在这些情况下，在 TypeScript 中制作插件需要类型声明。庆幸的是，TypeScript 有一个特性来补充现有的类型，叫做模块补充 (module augmentation)。

For example, to declare an instance property $myProperty with type string:

例如，声明一个 string 类型的实例属性 $myProperty：

After including the above code as a declaration file (like my-property.d.ts) in your project, you can use $myProperty on a Vue instance.

在你的项目中包含了上述作为声明文件的代码之后 (像 my-property.d.ts)，你就可以在 Vue 实例上使用 $myProperty 了。

You can also declare additional global properties and component options:

你也可以声明额外的属性和组件选项：

The above declarations allow the following code to be compiled:

上述的声明允许下面的代码顺利编译通过：

Annotating Return Types 标注返回值

Because of the circular nature of Vue’s declaration files, TypeScript may have difficulties inferring the types of certain methods. For this reason, you may need to annotate the return type on methods like render and those in computed.

因为 Vue 的声明文件天生就具有循环性，TypeScript 可能在推断某个方法的类型的时候存在困难。因此，你可能需要在 render 或 computed 里的方法上标注返回值。

If you find type inference or member completion isn’t working, annotating certain methods may help address these problems. Using the --noImplicitAny option will help find many of these unannotated methods.

如果你发现类型推导或成员补齐不工作了，标注某个方法也许可以帮助你解决这个问题。使用 --noImplicitAny 选项将会帮助你找到这些未标注的方法。

## Production Deployment生产环境部署

Most of the tips below are enabled by default if you are using Vue CLI. This section is only relevant if you are using a custom build setup.

以下大多数内容在你使用 Vue CLI 时都是默认开启的。该章节仅跟你自定义的构建设置有关。

Turn on Production Mode 开启生产环境模式

During development, Vue provides a lot of warnings to help you with common errors and pitfalls. However, these warning strings become useless in production and bloat your app’s payload size. In addition, some of these warning checks have small runtime costs that can be avoided in production mode.

开发环境下，Vue 会提供很多警告来帮你对付常见的错误与陷阱。而在生产环境下，这些警告语句却没有用，反而会增加应用的体积。此外，有些警告检查还有一些小的运行时开销，这在生产环境模式下是可以避免的。

Without Build Tools 不使用构建工具

If you are using the full build, i.e. directly including Vue via a script tag without a build tool, make sure to use the minified version (vue.min.js) for production. Both versions can be found in the Installation guide.

如果用 Vue 完整独立版本，即直接用 <script> 元素引入 Vue 而不提前进行构建，请记得在生产环境下使用压缩后的版本 (vue.min.js)。两种版本都可以在安装指导中找到。

With Build Tools 使用构建工具

When using a build tool like Webpack or Browserify, the production mode will be determined by process.env.NODE\_ENV inside Vue’s source code, and it will be in development mode by default. Both build tools provide ways to overwrite this variable to enable Vue’s production mode, and warnings will be stripped by minifiers during the build. All vue-cli templates have these pre-configured for you, but it would be beneficial to know how it is done:

当使用 webpack 或 Browserify 类似的构建工具时，Vue 源码会根据 process.env.NODE\_ENV 决定是否启用生产环境模式，默认情况为开发环境模式。在 webpack 与 Browserify 中都有方法来覆盖此变量，以启用 Vue 的生产环境模式，同时在构建过程中警告语句也会被压缩工具去除。所有这些在 vue-cli 模板中都预先配置好了，但了解一下怎样配置会更好。

In Webpack 4+, you can use the mode option:

在 webpack 4+ 中，你可以使用 mode 选项：

But in Webpack 3 and earlier, you’ll need to use DefinePlugin:

但是在 webpack 3 及其更低版本中，你需要使用 DefinePlugin：

Run your bundling command with the actual NODE\_ENV environment variable set to "production". This tells vueify to avoid including hot-reload and development related code.

在运行打包命令时将 NODE\_ENV 设置为 "production"。这等于告诉 vueify 避免引入热重载和开发相关的代码。

Apply a global envify transform to your bundle. This allows the minifier to strip out all the warnings in Vue’s source code wrapped in env variable conditional blocks. For example:

对打包后的文件进行一次全局的 envify 转换。这使得压缩工具能清除掉 Vue 源码中所有用环境变量条件包裹起来的警告语句。例如：

Or, using envify with Gulp:

或者在 Gulp 中使用 envify：

Or, using envify with Grunt and grunt-browserify:

或者配合 Grunt 和 grunt-browserify 使用 envify：

Pre-Compiling Templates 模板预编译

When using in-DOM templates or in-JavaScript template strings, the template-to-render-function compilation is performed on the fly. This is usually fast enough in most cases, but is best avoided if your application is performance-sensitive.

当使用 DOM 内模板或 JavaScript 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法。

The easiest way to pre-compile templates is using Single-File Components - the associated build setups automatically performs pre-compilation for you, so the built code contains the already compiled render functions instead of raw template strings.

预编译模板最简单的方式就是使用单文件组件——相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串。

If you are using Webpack, and prefer separating JavaScript and template files, you can use vue-template-loader, which also transforms the template files into JavaScript render functions during the build step.

如果你使用 webpack，并且喜欢分离 JavaScript 和模板文件，你可以使用 vue-template-loader，它也可以在构建过程中把模板文件转换成为 JavaScript 渲染函数。

Extracting Component CSS 提取组件的 CSS

When using Single-File Components, the CSS inside components are injected dynamically as <style>tags via JavaScript. This has a small runtime cost, and if you are using server-side rendering it will cause a “flash of unstyled content”. Extracting the CSS across all components into the same file will avoid these issues, and also result in better CSS minification and caching.

当使用单文件组件时，组件内的 CSS 会以 <style> 标签的方式通过 JavaScript 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式内容闪烁 (fouc)”。将所有组件的 CSS 提取到同一个文件可以避免这个问题，也会让 CSS 更好地进行压缩和缓存。

Refer to the respective build tool documentations to see how it’s done:

查阅这个构建工具各自的文档来了解更多：

Tracking Runtime Errors 跟踪运行时错误

If a runtime error occurs during a component’s render, it will be passed to the global Vue.config.errorHandler config function if it has been set. It might be a good idea to leverage this hook together with an error-tracking service like Sentry, which provides an official integration for Vue.

如果在组件渲染时出现运行错误，错误将会被传递至全局 Vue.config.errorHandler配置函数 (如果已设置)。利用这个钩子函数来配合错误跟踪服务是个不错的主意。比如 Sentry，它为 Vue 提供了官方集成。

# Scaling Up

## Routing路由

Official Router 官方路由

For most Single Page Applications, it’s recommended to use the officially-supported vue-router library. For more details, see vue-router’s documentation.

对于大多数单页面应用，都推荐使用官方支持的 vue-router 库。更多细节可以移步 vue-router 文档。

Simple Routing From Scratch 从零开始简单的路由

If you only need very simple routing and do not wish to involve a full-featured router library, you can do so by dynamically rendering a page-level component like this:

如果你只需要非常简单的路由而不想引入一个功能完整的路由库，可以像这样动态渲染一个页面级的组件：

Combined with the HTML5 History API, you can build a very basic but fully-functional client-side router. To see that in practice, check out this example app.

结合 HTML5 History API，你可以建立一个麻雀虽小五脏俱全的客户端路由器。可以直接看实例应用。

Integrating 3rd-Party Routers 整合第三方路由

If there’s a 3rd-party router you prefer to use, such as Page.js or Director, integration is similarly easy. Here’s a complete example using Page.js.

如果你有更偏爱的第三方路由，如 Page.js 或者 Director，整合起来也一样简单。这里有一个使用了 Page.js 的完整示例。

## State Management状态管理

Official Flux-Like Implementation 类 Flux 状态管理的官方实现

Large applications can often grow in complexity, due to multiple pieces of state scattered across many components and the interactions between them. To solve this problem, Vue offers vuex: our own Elm-inspired state management library. It even integrates into vue-devtools, providing zero-setup access to time travel debugging.

由于状态零散地分布在许多组件和组件之间的交互中，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 vuex：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 vue-devtools，无需配置即可进行时光旅行调试 (time travel debugging)。

Information for React Developers React 的开发者请参考以下信息

If you’re coming from React, you may be wondering how vuex compares to redux, the most popular Flux implementation in that ecosystem. Redux is actually view-layer agnostic, so it can easily be used with Vue via simple bindings. Vuex is different in that it knows it’s in a Vue app. This allows it to better integrate with Vue, offering a more intuitive API and improved development experience.

如果你是来自 React 的开发者，你可能会对 Vuex 和 Redux 间的差异表示关注，Redux 是 React 生态环境中最流行的 Flux 实现。Redux 事实上无法感知视图层，所以它能够轻松的通过一些简单绑定和 Vue 一起使用。Vuex 区别在于它是一个专门为 Vue 应用所设计。这使得它能够更好地和 Vue 进行整合，同时提供简洁的 API 和改善过的开发体验。

Simple State Management from Scratch 简单状态管理起步使用

It is often overlooked that the source of truth in Vue applications is the raw data object - a Vue instance only proxies access to it. Therefore, if you have a piece of state that should be shared by multiple instances, you can share it by identity:

经常被忽略的是，Vue 应用中原始数据对象的实际来源 - 当访问数据对象时，一个 Vue 实例只是简单的代理访问。所以，如果你有一处需要被多个实例间共享的状态，可以简单地通过维护一份数据来实现共享：

Now whenever sourceOfTruth is mutated, both vmA and vmB will update their views automatically. Subcomponents within each of these instances would also have access via this.$root.$data. We have a single source of truth now, but debugging would be a nightmare. Any piece of data could be changed by any part of our app at any time, without leaving a trace.

现在当 sourceOfTruth 发生变化，vmA 和 vmB 都将自动的更新引用它们的视图。子组件们的每个实例也会通过 this.$root.$data 去访问。现在我们有了唯一的数据来源，但是，调试将会变为噩梦。任何时间，我们应用中的任何部分，在任何数据改变后，都不会留下变更过的记录。

To help solve this problem, we can adopt a store pattern:

为了解决这个问题，我们采用一个简单的 store 模式：

Notice all actions that mutate the store’s state are put inside the store itself. This type of centralized state management makes it easier to understand what type of mutations could happen and how they are triggered. Now when something goes wrong, we’ll also have a log of what happened leading up to the bug.

需要注意，所有 store 中 state 的改变，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的 mutation 将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

In addition, each instance/component can still own and manage its own private state:

此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

It’s important to note that you should never replace the original state object in your actions - the components and the store need to share reference to the same object in order for mutations to be observed.

重要的是，注意你不应该在 action 中 替换原始的状态对象 - 组件和 store 需要引用同一个共享对象，mutation 才能够被观察

As we continue developing the convention where components are never allowed to directly mutate state that belongs to a store, but should instead dispatch events that notify the store to perform actions, we eventually arrive at the Flux architecture. The benefit of this convention is we can record all state mutations happening to the store and implement advanced debugging helpers such as mutation logs, snapshots, and history re-rolls / time travel.

接着我们继续延伸约定，组件不允许直接修改属于 store 实例的 state，而应执行 action 来分发 (dispatch) 事件通知 store 去改变，我们最终达成了 Flux 架构。这样约定的好处是，我们能够记录所有 store 中发生的 state 改变，同时实现能做到记录变更 (mutation)、保存状态快照、历史回滚/时光旅行的先进的调试工具。

This brings us full circle back to vuex, so if you’ve read this far it’s probably time to try it out!

说了一圈其实又回到了vuex，如果你已经读到这儿，或许可以去尝试一下！

## Server-Side Rendering服务端渲染

The Complete SSR Guide SSR 完全指南

We have created a standalone guide for creating server-rendered Vue applications. This is a very in-depth guide for those who are already familiar with client-side Vue development, server-side Node.js development and webpack. Check it out at ssr.vuejs.org.

在 2.3 发布后我们发布了一份完整的构建 Vue 服务端渲染应用的指南。这份指南非常深入，适合已经熟悉 Vue, webpack 和 Node.js 开发的开发者阅读。请移步 ssr.vuejs.org。

Properly configuring all the discussed aspects of a production-ready server-rendered app can be a daunting task. Luckily, there is an excellent community project that aims to make all of this easier: Nuxt.js. Nuxt.js is a higher-level framework built on top of the Vue ecosystem which provides an extremely streamlined development experience for writing universal Vue applications. Better yet, you can even use it as a static site generator (with pages authored as single-file Vue components)! We highly recommend giving it a try.

从头搭建一个服务端渲染的应用是相当复杂的。幸运的是，我们有一个优秀的社区项目 Nuxt.js 让这一切变得非常简单。Nuxt 是一个基于 Vue 生态的更高层的框架，为开发服务端渲染的 Vue 应用提供了极其便利的开发体验。更酷的是，你甚至可以用它来做为静态站生成器。推荐尝试。

Quasar Framework will generate an SSR app (with optional PWA handoff) that leverages its best-in-class build system, sensible configuration and developer extensibility to make designing and building your idea a breeze. With over one hundred specific “Material Design 2.0”-compliant components, you can decide which ones to execute on the server, which are available in the browser - and even manage the <meta> tags of your site. Quasar is a node.js and webpack based development environment that supercharges and streamlines rapid development of SPA, PWA, SSR, Electron and Cordova apps - all from one codebase.

Quasar Framework 可以通过其一流的构建系统、合理的配置和开发者扩展性生成 (可选地和 PWA 互通的) SSR 应用，让你的想法的设计和构建变得轻而易举。你可以在服务端挑选执行超过上百款遵循“Material Design 2.0”的组件，并在浏览器端可用。你甚至可以管理网站的 <meta> 标签。Quasar 是一个基于 Node.js 和 webpack 的开发环境，它可以通过一套代码完成 SPA、PWA、SSR、Electron 和 Cordova 应用的快速开发。

# Internals

## Reactivity in Depth深入响应式原理

Now it’s time to take a deep dive! One of Vue’s most distinct features is the unobtrusive reactivity system. Models are just plain JavaScript objects. When you modify them, the view updates. It makes state management simple and intuitive, but it’s also important to understand how it works to avoid some common gotchas. In this section, we are going to dig into some of the lower-level details of Vue’s reactivity system.

现在是时候深入一下了！Vue 最独特的特性之一，是其非侵入性的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以避开一些常见的问题。在这个章节，我们将研究一下 Vue 响应式系统的底层的细节。

How Changes Are Tracked 如何追踪变化

When you pass a plain JavaScript object to a Vue instance as its data option, Vue will walk through all of its properties and convert them to getter/setters using Object.defineProperty. This is an ES5-only and un-shimmable feature, which is why Vue doesn’t support IE8 and below.

当你把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的属性，并使用 Object.defineProperty 把这些属性全部转为 getter/setter。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

The getter/setters are invisible to the user, but under the hood they enable Vue to perform dependency-tracking and change-notification when properties are accessed or modified. One caveat is that browser consoles format getter/setters differently when converted data objects are logged, so you may want to install vue-devtools for a more inspection-friendly interface.

这些 getter/setter 对用户来说是不可见的，但是在内部它们让 Vue 能够追踪依赖，在属性被访问和修改时通知变更。这里需要注意的是不同浏览器在控制台打印数据对象时对 getter/setter 的格式化并不同，所以建议安装 vue-devtools 来获取对检查数据更加友好的用户界面。

Every component instance has a corresponding watcher instance, which records any properties “touched” during the component’s render as dependencies. Later on when a dependency’s setter is triggered, it notifies the watcher, which in turn causes the component to re-render.

每个组件实例都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据属性记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。

Change Detection Caveats 检测变化的注意事项

Due to the limitations of modern JavaScript (and the abandonment of Object.observe), Vue cannot detect property addition or deletion. Since Vue performs the getter/setter conversion process during instance initialization, a property must be present in the data object in order for Vue to convert it and make it reactive. For example:

受现代 JavaScript 的限制 (而且 Object.observe 也已经被废弃)，Vue 无法检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 Vue 将它转换为响应式的。例如：

Vue does not allow dynamically adding new root-level reactive properties to an already created instance. However, it’s possible to add reactive properties to a nested object using the Vue.set(object, propertyName, value) method:

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：

You can also use the vm.$set instance method, which is an alias to the global Vue.set:

您还可以使用 vm.$set 实例方法，这也是全局 Vue.set 方法的别名：

Sometimes you may want to assign a number of properties to an existing object, for example using Object.assign() or \_.extend(). However, new properties added to the object will not trigger changes. In such cases, create a fresh object with properties from both the original object and the mixin object:

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()。但是，这样添加到对象上的新属性不会触发更新。在这种情况下，你应该用原对象与要混合进去的对象的属性一起创建一个新的对象。

There are also a few array-related caveats, which were discussed earlier in the list rendering section.

也有一些数组相关的注意事项，之前已经在列表渲染中讲过。

Declaring Reactive Properties 声明响应式属性

Since Vue doesn’t allow dynamically adding root-level reactive properties, you have to initialize Vue instances by declaring all root-level reactive data properties upfront, even with an empty value:

由于 Vue 不允许动态添加根级响应式属性，所以你必须在初始化实例前声明所有根级响应式属性，哪怕只是一个空值：

If you don’t declare message in the data option, Vue will warn you that the render function is trying to access a property that doesn’t exist.

如果你未在 data 选项中声明 message，Vue 将警告你渲染函数正在试图访问不存在的属性。

There are technical reasons behind this restriction - it eliminates a class of edge cases in the dependency tracking system, and also makes Vue instances play nicer with type checking systems. But there is also an important consideration in terms of code maintainability: the data object is like the schema for your component’s state. Declaring all reactive properties upfront makes the component code easier to understand when revisited later or read by another developer.

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例能更好地配合类型检查系统工作。但与此同时在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的结构 (schema)。提前声明所有的响应式属性，可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

Async Update Queue 异步更新队列

In case you haven’t noticed yet, Vue performs DOM updates asynchronously. Whenever a data change is observed, it will open a queue and buffer all the data changes that happen in the same event loop. If the same watcher is triggered multiple times, it will be pushed into the queue only once. This buffered de-duplication is important in avoiding unnecessary calculations and DOM manipulations. Then, in the next event loop “tick”, Vue flushes the queue and performs the actual (already de-duped) work. Internally Vue tries native Promise.then, MutationObserver, and setImmediate for the asynchronous queuing and falls back to setTimeout(fn, 0).

可能你还没有注意到，Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替。

For example, when you set vm.someData = 'new value', the component will not re-render immediately. It will update in the next “tick”, when the queue is flushed. Most of the time we don’t need to care about this, but it can be tricky when you want to do something that depends on the post-update DOM state. Although Vue.js generally encourages developers to think in a “data-driven” fashion and avoid touching the DOM directly, sometimes it might be necessary to get your hands dirty. In order to wait until Vue.js has finished updating the DOM after a data change, you can use Vue.nextTick(callback) immediately after the data is changed. The callback will be called after the DOM has been updated. For example:

例如，当你设置 vm.someData = 'new value'，该组件不会立即重新渲染。当刷新队列时，组件会在下一个事件循环“tick”中更新。多数情况我们不需要关心这个过程，但是如果你想基于更新后的 DOM 状态来做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们必须要这么做。为了在数据变化之后等待 Vue 完成更新 DOM，可以在数据变化之后立即使用 Vue.nextTick(callback)。这样回调函数将在 DOM 更新完成后被调用。例如：

There is also the vm.$nextTick() instance method, which is especially handy inside components, because it doesn’t need global Vue and its callback’s this context will be automatically bound to the current Vue instance:

在组件内使用 vm.$nextTick() 实例方法特别方便，因为它不需要全局 Vue，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上：

Since $nextTick() returns a promise, you can achieve the same as the above using the new ES2016 async/await syntax:

因为 $nextTick() 返回一个 Promise 对象，所以你可以使用新的 ES2016 async/await 语法完成相同的事情：

# Migrating

## Migration from Vue 1.x从 Vue 1.x 迁移

Woah - this is a super long page! Does that mean 2.0 is completely different, I’ll have to learn the basics all over again, and migrating will be practically impossible?

哇，非常长的一页！是否意味着 Vue 2.0 已经完全不同了呢，是否需要从头学起呢，Vue 1.0 的项目是不是没法迁移了？

I’m glad you asked! The answer is no. About 90% of the API is the same and the core concepts haven’t changed. It’s long because we like to offer very detailed explanations and include a lot of examples. Rest assured, this is not something you have to read from top to bottom!

非常开心地告诉你，并不是！几乎 90% 的 API 和核心概念都没有变。因为本节包含了很多详尽的阐述以及许多迁移的例子，所以显得有点长。不用担心，你不必从头到尾把本节读一遍！

Where should I start in a migration? 我该从哪里开始项目迁移呢？

Start by running the migration helper on a current project. We’ve carefully minified and compressed a senior Vue dev into a simple command line interface. Whenever they recognize an obsolete feature, they’ll let you know, offer suggestions, and provide links to more info.

首先，在当前项目下运行迁移工具。我们非常谨慎地把高级 Vue 升级过程简化为使用一个简单的命令行工具。当工具识别出旧有的特性后，就会告知你并给出建议，同时附上关于详细信息的链接。

After that, browse through the table of contents for this page in the sidebar. If you see a topic you may be affected by, but the migration helper didn’t catch, check it out.

然后，浏览本页面的侧边栏列出的内容。如果发现有的标题对你的项目有影响，但是迁移工具没有给出提示，请检查自己的项目。

If you have any tests, run them and see what still fails. If you don’t have tests, just open the app in your browser and keep an eye out for warnings or errors as you navigate around.

如果你的项目有测试代码，运行并查看仍然失败的地方。如果没有测试代码，在浏览器中打开你的程序，通过导航环顾并留意那些报错或警告信息。

By now, your app should be fully migrated. If you’re still hungry for more though, you can read the rest of this page - or dive in to the new and improved guide from the beginning. Many parts will be skimmable, since you’re already familiar with the core concepts.

现在，你的应用程序应该已彻底完成迁移。如果你渴望了解更多，可以阅读本页面剩余部分 - 或者从介绍部分，从头开始深入新的文档和改进过的指南。由于你已经熟悉一些核心概念，所以许多部分已经被删除掉。

How long will it take to migrate a Vue 1.x app to 2.0? 将 Vue 1.x 版本的应用程序迁移到 2.0 要花多长时间？

It depends on a few factors:

这取决于几个因素：

The size of your app (small to medium-sized apps will probably be less than a day)

取决于你应用程序的规模 (中小型的基本上一天内就可以搞定)。

How many times you get distracted and start playing with a cool new feature. 😉  Not judging, it also happened to us while building 2.0!

取决于你分心和开始 2.0 最酷的新功能的次数。😉  无法判断时间，我们构建 2.0 应用的时候也经常发生这种事！

Which obsolete features you’re using. Most can be upgraded with find-and-replace, but others might take a few minutes. If you’re not currently following best practices, Vue 2.0 will also try harder to force you to. This is a good thing in the long run, but could also mean a significant (though possibly overdue) refactor.

取决于你使用了哪些旧有的特性。大部分可以通过查找和替换 (find-and-replace) 来实现升级，但有一些可能还是要花点时间。如果你没有遵循最佳实践，Vue 2.0 会尽力强迫你去遵循。这有利于项目的长期运行，但也可能意味着重大重构 (尽管有些需要重构的部分可能已经过时)。

If I upgrade to Vue 2, will I also have to upgrade Vuex and Vue Router?

如果我升级到到 Vue 2 ，我还必须同时升级 Vuex 和 Vue Router？

Only Vue Router 2 is compatible with Vue 2, so yes, you’ll have to follow the migration path for Vue Router as well. Fortunately, most applications don’t have a lot of router code, so this likely won’t take more than an hour.

只有 Vue Router 2 与 Vue 2 保持兼容，所以 Vue Router 是需要升级的，你必须遵循 Vue Router 迁移方式来处理。幸运的是，大多数应用没有很多 router 相关代码，所以迁移可能不会超过一个小时。

As for Vuex, even version 0.8 is compatible with Vue 2, so you’re not forced to upgrade. The only reason you may want to upgrade immediately is to take advantage of the new features in Vuex 2, such as modules and reduced boilerplate.

对于 Vuex ，版本 0.8+ 与 Vue 2 保持兼容，所以部分不必强制升级。可以促使你立即升级的唯一理由，是你想要使用那些 Vuex 2 中新的高级特性，比如模块 (modules) 和减少的样板文件 (reduced boilerplate)。

Templates 模板

Fragment Instances removed 片段实例 移除

Every component must have exactly one root element. Fragment instances are no longer allowed. If you have a template like this:

每个组件必须只有一个根元素。不再允许片段实例，如果你有这样的模板：

It’s recommended to wrap the entire contents in a new element, like this:

最好把整个内容都简单包裹到一个新的元素里，如下所示：

Upgrade Path升级方式

Run your end-to-end test suite or app after upgrading and look for console warnings about multiple root elements in a template.

升级后运行端到端测试套件 (end-to-end test suite) 或运行应用程序，并查看控制台警告 (console warnings) 来找出那些模板中有多个根元素的地方。