**Vue.js 服务器端渲染指南**

**注意： 本指南需要最低为如下版本的 Vue，以及以下 library 支持：**

* vue & vue-server-renderer 2.3.0+
* vue-router 2.5.0+
* vue-loader 12.0.0+ & vue-style-loader 3.0.0+

**如果先前已经使用过 Vue 2.2 的服务器端渲染(SSR)，你应该注意到，推荐的代码结构现在**[**略有不同**](https://ssr.vuejs.org/zh/structure.html)**（使用新的**[**runInNewContext**](https://ssr.vuejs.org/zh/api.html#runinnewcontext)**选项，并设置为 false）。现有的应用程序可以继续运行，但建议你迁移到新的推荐规范。**

**什么是服务器端渲染(SSR)？**

Vue.js 是构建客户端应用程序的框架。默认情况下，可以在浏览器中输出 Vue 组件，进行生成 DOM 和操作 DOM。然而，也可以将同一个组件渲染为服务器端的 HTML 字符串，将它们直接发送到浏览器，最后将静态标记"混合"为客户端上完全交互的应用程序。

服务器渲染的 Vue.js 应用程序也可以被认为是"同构"或"通用"，因为应用程序的大部分代码都可以在服务器**和**客户端上运行。

**为什么使用服务器端渲染(SSR)？**

与传统 SPA（Single-Page Application - 单页应用程序）相比，服务器端渲染(SSR)的优势主要在于：

* 更好的 SEO，由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面。

请注意，截至目前，Google 和 Bing 可以很好对同步 JavaScript 应用程序进行索引。在这里，同步是关键。如果你的应用程序初始展示 loading 菊花图，然后通过 Ajax 获取内容，抓取工具并不会等待异步完成后再行抓取页面内容。也就是说，如果 SEO 对你的站点至关重要，而你的页面又是异步获取内容，则你可能需要服务器端渲染(SSR)解决此问题。

* 更快的内容到达时间(time-to-content)，特别是对于缓慢的网络情况或运行缓慢的设备。无需等待所有的 JavaScript 都完成下载并执行，才显示服务器渲染的标记，所以你的用户将会更快速地看到完整渲染的页面。通常可以产生更好的用户体验，并且对于那些「内容到达时间(time-to-content)与转化率直接相关」的应用程序而言，服务器端渲染(SSR)至关重要。

使用服务器端渲染(SSR)时还需要有一些权衡之处：

* 开发条件所限。浏览器特定的代码，只能在某些生命周期钩子函数(lifecycle hook)中使用；一些外部扩展库(external library)可能需要特殊处理，才能在服务器渲染应用程序中运行。
* 涉及构建设置和部署的更多要求。与可以部署在任何静态文件服务器上的完全静态单页面应用程序(SPA)不同，服务器渲染应用程序，需要处于 Node.js server 运行环境。
* 更多的服务器端负载。在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用 CPU 资源(CPU-intensive - CPU 密集)，因此如果你预料在高流量环境(high traffic)下使用，请准备相应的服务器负载，并明智地采用缓存策略。

在对你的应用程序使用服务器端渲染(SSR)之前，你应该问第一个问题是否真的需要它。这主要取决于内容到达时间(time-to-content)对应用程序的重要程度。例如，如果你正在构建一个内部仪表盘，初始加载时的额外几百毫秒并不重要，这种情况下去使用服务器端渲染(SSR)将是一个小题大作之举。然而，内容到达时间(time-to-content)要求是绝对关键的指标，在这种情况下，服务器端渲染(SSR)可以帮助你实现最佳的初始加载性能。

**服务器端渲染 vs 预渲染(SSR vs Prerendering)**

如果你调研服务器端渲染(SSR)只是用来改善少数营销页面（例如 /, /about, /contact 等）的 SEO，那么你可能需要**预渲染**。无需使用 web 服务器实时动态编译 HTML，而是使用预渲染方式，在构建时(build time)简单地生成针对特定路由的静态 HTML 文件。优点是设置预渲染更简单，并可以将你的前端作为一个完全静态的站点。

如果你使用 webpack，你可以使用 [**prerender-spa-plugin**](https://github.com/chrisvfritz/prerender-spa-plugin) 轻松地添加预渲染。它已经被 Vue 应用程序广泛测试 - 事实上，[**作者**](https://github.com/chrisvfritz)是 Vue 核心团队的成员。

**关于此指南**

本指南专注于，使用 Node.js server 的服务器端单页面应用程序渲染。将 Vue 服务器端渲染(SSR)与其他后端设置进行混合使用，是其它后端自身的一个主题，本指南不包括在内。

本指南将会非常深入，并且假设你已经熟悉 Vue.js 本身，并且具有 Node.js 和 webpack 的相当不错的应用经验。如果你倾向于使用提供了平滑开箱即用体验的更高层次解决方案，你应该去尝试使用 [**Nuxt.js**](http://nuxtjs.org/)。它建立在同等的 Vue 技术栈之上，但抽象出很多模板，并提供了一些额外的功能，例如静态站点生成。但是，如果你需要更直接地控制应用程序的结构，Nuxt.js 并不适合这种使用场景。无论如何，阅读本指南将更有助于更好地了解一切如何运行。

当你阅读时，参考官方 [**HackerNews Demo**](https://github.com/vuejs/vue-hackernews-2.0/) 将会有所帮助，此示例使用了本指南涵盖的大部分技术。

最后，请注意，本指南中的解决方案不是限定的 - 我们发现它们对我们来说很好，但这并不意味着无法继续改进。可能会在未来持续改进，欢迎通过随意提交 pull request 作出贡献！

# 基本用法

## 安装

npm install vue vue-server-renderer --save

我们将在整个指南中使用 NPM，但你也可以使用 [**Yarn**](https://yarnpkg.com/en/)。

#### 注意

* 推荐使用 Node.js 版本 6+。
* vue-server-renderer 和 vue 必须匹配版本。
* vue-server-renderer 依赖一些 Node.js 原生模块，因此只能在 Node.js 中使用。我们可能会提供一个更简单的构建，可以在将来在其他「JavaScript 运行时(runtime)」运行。

## 渲染一个 Vue 实例

// 第 1 步：创建一个 Vue 实例

const Vue = require('vue')

const app = new Vue({

template: `<div>Hello World</div>`

})

// 第 2 步：创建一个 renderer

const renderer = require('vue-server-renderer').createRenderer()

// 第 3 步：将 Vue 实例渲染为 HTML

renderer.renderToString(app, (err, html) => {

if (err) throw err

console.log(html)

// => <div data-server-rendered="true">Hello World</div>

})

## 与服务器集成

在 Node.js 服务器中使用时相当简单直接，例如 [**Express**](https://expressjs.com/)：

npm install express --save

const Vue = require('vue')

const server = require('express')()

const renderer = require('vue-server-renderer').createRenderer()

server.get('\*', (req, res) => {

const app = new Vue({

data: {

url: req.url

},

template: `<div>访问的 URL 是： {{ url }}</div>`

})

renderer.renderToString(app, (err, html) => {

if (err) {

res.status(500).end('Internal Server Error')

return

}

res.end(`

<!DOCTYPE html>

<html lang="en">

<head><title>Hello</title></head>

<body>${html}</body>

</html>

`)

})

})

server.listen(8080)

## 使用一个页面模板

当你在渲染 Vue 应用程序时，renderer 只从应用程序生成 HTML 标记(markup)。在这个示例中，我们必须用一个额外的 HTML 页面包裹容器，来包裹生成的 HTML 标记。

为了简化这些，你可以直接在创建 renderer 时提供一个页面模板。多数时候，我们会将页面模板放在特有的文件中，例如 index.template.html：

<!DOCTYPE html>

<html lang="en">

<head><title>Hello</title></head>

<body>

<!--vue-ssr-outlet-->

</body>

</html>

注意 <!--vue-ssr-outlet--> 注释 -- 这里将是应用程序 HTML 标记注入的地方。

然后，我们可以读取和传输文件到 Vue renderer 中：

const renderer = createRenderer({

template: require('fs').readFileSync('./index.template.html', 'utf-8')

})

renderer.renderToString(app, (err, html) => {

console.log(html) // will be the full page with app content injected.

})

### 模板插值

模板还支持简单插值。给定如下模板：

<html>

<head>

<!-- 使用双花括号(double-mustache)进行 HTML 转义插值(HTML-escaped interpolation) -->

<title>{{ title }}</title>

<!-- 使用三花括号(triple-mustache)进行 HTML 不转义插值(non-HTML-escaped interpolation) -->

{{{ meta }}}

</head>

<body>

<!--vue-ssr-outlet-->

</body>

</html>

我们可以通过传入一个"渲染上下文对象"，作为 renderToString 函数的第二个参数，来提供插值数据：

const context = {

title: 'hello',

meta: `

<meta ...>

<meta ...>

`

}

renderer.renderToString(app, context, (err, html) => {

// page title will be "Hello"

// with meta tags injected

})

也可以与 Vue 应用程序实例共享 context 对象，允许模板插值中的组件动态地注册数据。

此外，模板支持一些高级特性，例如：

* 在使用 \*.vue 组件时，自动注入「关键的 CSS(critical CSS)」；
* 在使用 clientManifest 时，自动注入「资源链接(asset links)和资源预加载提示(resource hints)」；
* 在嵌入 Vuex 状态进行客户端融合(client-side hydration)时，自动注入以及 XSS 防御。

在之后的指南中介绍相关概念时，我们将详细讨论这些。

**编写通用代码**

在进一步介绍之前，让我们花点时间来讨论编写"通用"代码时的约束条件 - 即运行在服务器和客户端的代码。由于用例和平台 API 的差异，当运行在不同环境中时，我们的代码将不会完全相同。所以这里我们将会阐述你需要理解的关键事项。

**服务器上的数据响应**

在纯客户端应用程序(client-only app)中，每个用户会在他们各自的浏览器中使用新的应用程序实例。对于服务器端渲染，我们也希望如此：每个请求应该都是全新的、独立的应用程序实例，以便不会有交叉请求造成的状态污染(cross-request state pollution)。

因为实际的渲染过程需要确定性，所以我们也将在服务器上“预取”数据("pre-fetching" data) - 这意味着在我们开始渲染时，我们的应用程序就已经解析完成其状态。也就是说，将数据进行响应式的过程在服务器上是多余的，所以默认情况下禁用。禁用响应式数据，还可以避免将「数据」转换为「响应式对象」的性能开销。

**组件生命周期钩子函数**

由于没有动态更新，所有的生命周期钩子函数中，只有 beforeCreate 和 created 会在服务器端渲染(SSR)过程中被调用。这就是说任何其他生命周期钩子函数中的代码（例如 beforeMount 或 mounted），只会在客户端执行。

此外还需要注意的是，你应该避免在 beforeCreate 和 created 生命周期时产生全局副作用的代码，例如在其中使用 setInterval 设置 timer。在纯客户端(client-side only)的代码中，我们可以设置一个 timer，然后在 beforeDestroy或 destroyed 生命周期时将其销毁。但是，由于在 SSR 期间并不会调用销毁钩子函数，所以 timer 将永远保留下来。为了避免这种情况，请将副作用代码移动到 beforeMount 或 mounted 生命周期中。

**访问特定平台(Platform-Specific) API**

通用代码不可接受特定平台的 API，因此如果你的代码中，直接使用了像 window 或 document，这种仅浏览器可用的全局变量，则会在 Node.js 中执行时抛出错误，反之也是如此。

对于共享于服务器和客户端，但用于不同平台 API 的任务(task)，建议将平台特定实现包含在通用 API 中，或者使用为你执行此操作的 library。例如，[**axios**](https://github.com/mzabriskie/axios) 是一个 HTTP 客户端，可以向服务器和客户端都暴露相同的 API。

对于仅浏览器可用的 API，通常方式是，在「纯客户端(client-only)」的生命周期钩子函数中惰性访问(lazily access)它们。

请注意，考虑到如果第三方 library 不是以上面的通用用法编写，则将其集成到服务器渲染的应用程序中，可能会很棘手。你*可能*要通过模拟(mock)一些全局变量来使其正常运行，但这只是 hack 的做法，并且可能会干扰到其他 library 的环境检测代码。

**自定义指令**

大多数自定义指令直接操作 DOM，因此会在服务器端渲染(SSR)过程中导致错误。有两种方法可以解决这个问题：

1. 推荐使用组件作为抽象机制，并运行在「虚拟 DOM 层级(Virtual-DOM level)」（例如，使用渲染函数(render function)）。
2. 如果你有一个自定义指令，但是不是很容易替换为组件，则可以在创建服务器 renderer 时，使用 [**directives**](https://ssr.vuejs.org/zh/api.html#directives) 选项所提供"服务器端版本(server-side version)"。

# 源码结构

## 避免状态单例

当编写纯客户端(client-only)代码时，我们习惯于每次在新的上下文中对代码进行取值。但是，Node.js 服务器是一个长期运行的进程。当我们的代码进入该进程时，它将进行一次取值并留存在内存中。这意味着如果创建一个单例对象，它将在每个传入的请求之间共享。

如基本示例所示，我们**为每个请求创建一个新的根 Vue 实例**。这与每个用户在自己的浏览器中使用新应用程序的实例类似。如果我们在多个请求之间使用一个共享的实例，很容易导致交叉请求状态污染(cross-request state pollution)。

因此，我们不应该直接创建一个应用程序实例，而是应该暴露一个可以重复执行的工厂函数，为每个请求创建新的应用程序实例：

// app.js

const Vue = require('vue')

module.exports = function createApp (context) {

return new Vue({

data: {

url: context.url

},

template: `<div>访问的 URL 是： {{ url }}</div>`

})

}

并且我们的服务器代码现在变为：

// server.js

const createApp = require('./app')

server.get('\*', (req, res) => {

const context = { url: req.url }

const app = createApp(context)

renderer.renderToString(app, (err, html) => {

// 处理错误……

res.end(html)

})

})

同样的规则也适用于 router、store 和 event bus 实例。你不应该直接从模块导出并将其导入到应用程序中，而是需要在 createApp 中创建一个新的实例，并从根 Vue 实例注入。

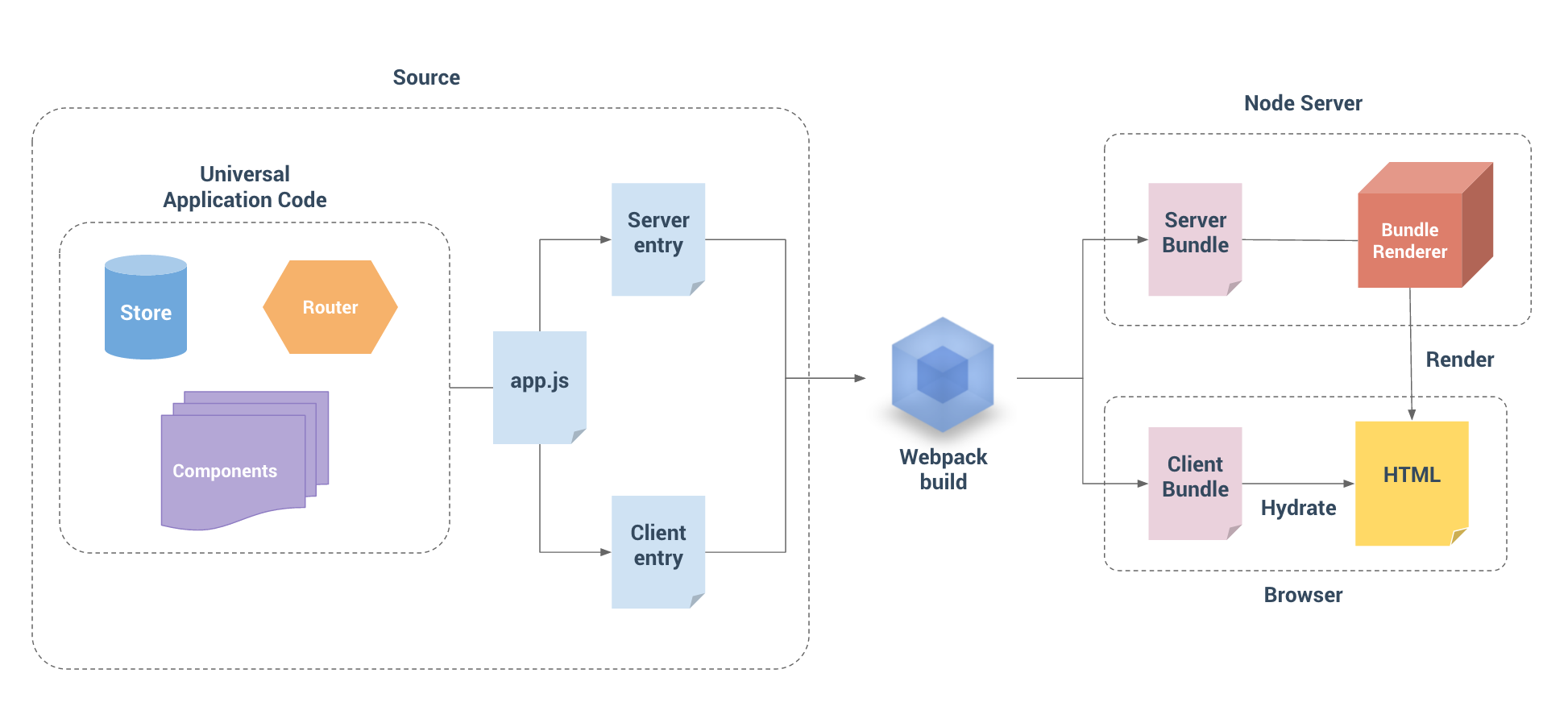
**在使用带有 { runInNewContext: true } 的 bundle renderer 时，可以消除此约束，但是由于需要为每个请求创建一个新的 vm 上下文，因此伴随有一些显著性能开销。**

## 介绍构建步骤

到目前为止，我们还没有讨论过如何将相同的 Vue 应用程序提供给客户端。为了做到这一点，我们需要使用 webpack 来打包我们的 Vue 应用程序。事实上，我们可能需要在服务器上使用 webpack 打包 Vue 应用程序，因为：

* 通常 Vue 应用程序是由 webpack 和 vue-loader 构建，并且许多 webpack 特定功能不能直接在 Node.js 中运行（例如通过 file-loader 导入文件，通过 css-loader 导入 CSS）。
* 尽管 Node.js 最新版本能够完全支持 ES2015 特性，我们还是需要转译客户端代码以适应老版浏览器。这也会涉及到构建步骤。

所以基本看法是，对于客户端应用程序和服务器应用程序，我们都要使用 webpack 打包 - 服务器需要「服务器 bundle」然后用于服务器端渲染(SSR)，而「客户端 bundle」会发送给浏览器，用于混合静态标记。



我们将在后面的章节讨论规划结构的细节 - 现在，先假设我们已经将构建过程的规划都弄清楚了，我们可以在启用 webpack 的情况下编写我们的 Vue 应用程序代码。

## 使用 webpack 的源码结构

现在我们正在使用 webpack 来处理服务器和客户端的应用程序，大部分源码可以使用通用方式编写，可以使用 webpack 支持的所有功能。同时，在编写通用代码时，有一些[**事项**](https://ssr.vuejs.org/zh/universal.html)要牢记在心。

一个基本项目可能像是这样：

src

├── components

│ ├── Foo.vue

│ ├── Bar.vue

│ └── Baz.vue

├── App.vue

├── app.js # universal entry

├── entry-client.js # 仅运行于浏览器

└── entry-server.js # 仅运行于服务器

### app.js

app.js 是我们应用程序的「通用 entry」。在纯客户端应用程序中，我们将在此文件中创建根 Vue 实例，并直接挂载到 DOM。但是，对于服务器端渲染(SSR)，责任转移到纯客户端 entry 文件。app.js 简单地使用 export 导出一个 createApp 函数：

import Vue from 'vue'

import App from './App.vue'

// 导出一个工厂函数，用于创建新的

// 应用程序、router 和 store 实例

export function createApp () {

const app = new Vue({

// 根实例简单的渲染应用程序组件。

render: h => h(App)

})

return { app }

}

### entry-client.js:

客户端 entry 只需创建应用程序，并且将其挂载到 DOM 中：

import { createApp } from './app'

// 客户端特定引导逻辑……

const { app } = createApp()

// 这里假定 App.vue 模板中根元素具有 `id="app"`

app.$mount('#app')

### entry-server.js:

服务器 entry 使用 default export 导出函数，并在每次渲染中重复调用此函数。此时，除了创建和返回应用程序实例之外，它不会做太多事情 - 但是稍后我们将在此执行服务器端路由匹配(server-side route matching)和数据预取逻辑(data pre-fetching logic)。

import { createApp } from './app'

export default context => {

const { app } = createApp()

return app

}

# 路由和代码分割

## 使用 vue-router 的路由

你可能已经注意到，我们的服务器代码使用了一个 \* 处理程序，它接受任意 URL。这允许我们将访问的 URL 传递到我们的 Vue 应用程序中，然后对客户端和服务器复用相同的路由配置！

为此，建议使用官方提供的 vue-router。我们首先创建一个文件，在其中创建 router。注意，类似于 createApp，我们也需要给每个请求一个新的 router 实例，所以文件导出一个 createRouter 函数：

// router.js

import Vue from 'vue'

import Router from 'vue-router'

Vue.use(Router)

export function createRouter () {

return new Router({

mode: 'history',

routes: [

// ...

]

})

}

然后更新 app.js:

// app.js

import Vue from 'vue'

import App from './App.vue'

import { createRouter } from './router'

export function createApp () {

// 创建 router 实例

const router = createRouter()

const app = new Vue({

// 注入 router 到根 Vue 实例

router,

render: h => h(App)

})

// 返回 app 和 router

return { app, router }

}

现在我们需要在 entry-server.js 中实现服务器端路由逻辑(server-side routing logic)：

// entry-server.js

import { createApp } from './app'

export default context => {

// 因为有可能会是异步路由钩子函数或组件，所以我们将返回一个 Promise，

// 以便服务器能够等待所有的内容在渲染前，

// 就已经准备就绪。

return new Promise((resolve, reject) => {

const { app, router } = createApp()

// 设置服务器端 router 的位置

router.push(context.url)

// 等到 router 将可能的异步组件和钩子函数解析完

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

// 匹配不到的路由，执行 reject 函数，并返回 404

if (!matchedComponents.length) {

return reject({ code: 404 })

}

// Promise 应该 resolve 应用程序实例，以便它可以渲染

resolve(app)

}, reject)

})

}

假设服务器 bundle 已经完成构建（请再次忽略现在的构建设置），服务器用法看起来如下：

// server.js

const createApp = require('/path/to/built-server-bundle.js')

server.get('\*', (req, res) => {

const context = { url: req.url }

createApp(context).then(app => {

renderer.renderToString(app, (err, html) => {

if (err) {

if (err.code === 404) {

res.status(404).end('Page not found')

} else {

res.status(500).end('Internal Server Error')

}

} else {

res.end(html)

}

})

})

})

## 代码分割

应用程序的代码分割或惰性加载，有助于减少浏览器在初始渲染中下载的资源体积，可以极大地改善大体积 bundle 的可交互时间(TTI - time-to-interactive)。这里的关键在于，对初始首屏而言，"只加载所需"。

Vue 提供异步组件作为第一类的概念，将其与 [**webpack 2 所支持的使用动态导入作为代码分割点**](https://webpack.js.org/guides/code-splitting-async/)相结合，你需要做的是：

// 这里进行修改……

import Foo from './Foo.vue'

// 改为这样：

const Foo = () => import('./Foo.vue')

如果你正在构建纯客户端 Vue 应用程序，这将在任何情况下运行。但是，在 SSR 中使用时有一些限制。首先，你需要在启动渲染之前先在服务器上解析所有的异步组件，否则你将在标记中获得一个空的占位符。在客户端，你还需要在开始混合之前执行此操作，否则客户端将遇到内容不匹配错误。

这使得在应用程序中的任意位置使用异步组件变得有点棘手（我们将来可能会改进这一点）。但是，**如果你在路由级别完成以上，可以实现无缝运行**（即在路由配置中使用异步组件），因为在解析路由时，vue-router 将自动解析匹配的异步组件。你需要做的是确保在服务器和客户端上都使用 router.onReady。我们已经在我们的服务器入口(server entry)中实现过了，现在我们只需要更新客户端入口(client entry)：

// entry-client.js

import { createApp } from './app'

const { app, router } = createApp()

router.onReady(() => {

app.$mount('#app')

})

异步路由组件的路由配置示例：

// router.js

import Vue from 'vue'

import Router from 'vue-router'

Vue.use(Router)

export function createRouter () {

return new Router({

mode: 'history',

routes: [

{ path: '/', component: () => import('./components/Home.vue') },

{ path: '/item/:id', component: () => import('./components/Item.vue') }

]

})

}

**数据预取和状态**

**数据预取存储容器(Data Store)**

在服务器端渲染(SSR)期间，我们本质上是在渲染我们应用程序的"快照"，所以如果应用程序依赖于一些异步数据，**那么在开始渲染过程之前，需要先预取和解析好这些数据**。

另一个需要关注的问题是在客户端，在挂载(mount)到客户端应用程序之前，需要获取到与服务器端应用程序完全相同的数据 - 否则，客户端应用程序会因为使用与服务器端应用程序不同的状态，然后导致混合失败。

为了解决这个问题，获取的数据需要位于视图组件之外，即放置在专门的数据预取存储容器(data store)或"状态容器(state container)）"中。首先，在服务器端，我们可以在渲染之前预取数据，并将数据填充到 store 中。此外，我们将在 HTML 中序列化(serialize)和内联预置(inline)状态。这样，在挂载(mount)到客户端应用程序之前，可以直接从 store 获取到内联预置(inline)状态。

为此，我们将使用官方状态管理库 [**Vuex**](https://github.com/vuejs/vuex/)。我们先创建一个 store.js 文件，里面会模拟一些根据 id 获取 item 的逻辑：

// store.js

import Vue from 'vue'

import Vuex from 'vuex'

Vue.use(Vuex)

// 假定我们有一个可以返回 Promise 的

// 通用 API（请忽略此 API 具体实现细节）

import { fetchItem } from './api'

export function createStore () {

return new Vuex.Store({

state: {

items: {}

},

actions: {

fetchItem ({ commit }, id) {

// `store.dispatch()` 会返回 Promise，

// 以便我们能够知道数据在何时更新

return fetchItem(id).then(item => {

commit('setItem', { id, item })

})

}

},

mutations: {

setItem (state, { id, item }) {

Vue.set(state.items, id, item)

}

}

})

}

然后修改 app.js：

// app.js

import Vue from 'vue'

import App from './App.vue'

import { createRouter } from './router'

import { createStore } from './store'

import { sync } from 'vuex-router-sync'

export function createApp () {

// 创建 router 和 store 实例

const router = createRouter()

const store = createStore()

// 同步路由状态(route state)到 store

sync(store, router)

// 创建应用程序实例，将 router 和 store 注入

const app = new Vue({

router,

store,

render: h => h(App)

})

// 暴露 app, router 和 store。

return { app, router, store }

}

**带有逻辑配置的组件(Logic Collocation with Components)**

那么，我们在哪里放置「dispatch 数据预取 action」的代码？

我们需要通过访问路由，来决定获取哪部分数据 - 这也决定了哪些组件需要渲染。事实上，给定路由所需的数据，也是在该路由上渲染组件时所需的数据。所以在路由组件中放置数据预取逻辑，是很自然的事情。

我们将在路由组件上暴露出一个自定义静态函数 asyncData。注意，由于此函数会在组件实例化之前调用，所以它无法访问 this。需要将 store 和路由信息作为参数传递进去：

<!-- Item.vue -->

<template>

<div>{{ item.title }}</div>

</template>

<script>

export default {

asyncData ({ store, route }) {

// 触发 action 后，会返回 Promise

return store.dispatch('fetchItem', route.params.id)

},

computed: {

// 从 store 的 state 对象中的获取 item。

item () {

return this.$store.state.items[this.$route.params.id]

}

}

}

</script>

**服务器端数据预取(Server Data Fetching)**

在 entry-server.js 中，我们可以通过路由获得与 router.getMatchedComponents() 相匹配的组件，如果组件暴露出 asyncData，我们就调用这个方法。然后我们需要将解析完成的状态，附加到渲染上下文(render context)中。

// entry-server.js

import { createApp } from './app'

export default context => {

return new Promise((resolve, reject) => {

const { app, router, store } = createApp()

router.push(context.url)

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

if (!matchedComponents.length) {

return reject({ code: 404 })

}

// 对所有匹配的路由组件调用 `asyncData()`

Promise.all(matchedComponents.map(Component => {

if (Component.asyncData) {

return Component.asyncData({

store,

route: router.currentRoute

})

}

})).then(() => {

// 在所有预取钩子(preFetch hook) resolve 后，

// 我们的 store 现在已经填充入渲染应用程序所需的状态。

// 当我们将状态附加到上下文，

// 并且 `template` 选项用于 renderer 时，

// 状态将自动序列化为 `window.\_\_INITIAL\_STATE\_\_`，并注入 HTML。

context.state = store.state

resolve(app)

}).catch(reject)

}, reject)

})

}

当使用 template 时，context.state 将作为 window.\_\_INITIAL\_STATE\_\_ 状态，自动嵌入到最终的 HTML 中。而在客户端，在挂载到应用程序之前，store 就应该获取到状态：

// entry-client.js

const { app, router, store } = createApp()

if (window.\_\_INITIAL\_STATE\_\_) {

store.replaceState(window.\_\_INITIAL\_STATE\_\_)

}

**客户端数据预取(Client Data Fetching)**

在客户端，处理数据预取有两种不同方式：

1. **在路由导航之前解析数据：**

使用此策略，应用程序会等待视图所需数据全部解析之后，再传入数据并处理当前视图。好处在于，可以直接在数据准备就绪时，传入视图渲染完整内容，但是如果数据预取需要很长时间，用户在当前视图会感受到"明显卡顿"。因此，如果使用此策略，建议提供一个数据加载指示器(data loading indicator)。

我们可以通过检查匹配的组件，并在全局路由钩子函数中执行 asyncData 函数，来在客户端实现此策略。注意，在初始路由准备就绪之后，我们应该注册此钩子，这样我们就不必再次获取服务器提取的数据。

// entry-client.js

// ...忽略无关代码

router.onReady(() => {

// 添加路由钩子函数，用于处理 asyncData.

// 在初始路由 resolve 后执行，

// 以便我们不会二次预取(double-fetch)已有的数据。

// 使用 `router.beforeResolve()`，以便确保所有异步组件都 resolve。

router.beforeResolve((to, from, next) => {

const matched = router.getMatchedComponents(to)

const prevMatched = router.getMatchedComponents(from)

// 我们只关心之前没有渲染的组件

// 所以我们对比它们，找出两个匹配列表的差异组件

let diffed = false

const activated = matched.filter((c, i) => {

return diffed || (diffed = (prevMatched[i] !== c))

})

if (!activated.length) {

return next()

}

// 这里如果有加载指示器(loading indicator)，就触发

Promise.all(activated.map(c => {

if (c.asyncData) {

return c.asyncData({ store, route: to })

}

})).then(() => {

// 停止加载指示器(loading indicator)

next()

}).catch(next)

})

app.$mount('#app')

})

1. **匹配要渲染的视图后，再获取数据：**

此策略将客户端数据预取逻辑，放在视图组件的 beforeMount 函数中。当路由导航被触发时，可以立即切换视图，因此应用程序具有更快的响应速度。然而，传入视图在渲染时不会有完整的可用数据。因此，对于使用此策略的每个视图组件，都需要具有条件加载状态。

这可以通过纯客户端(client-only)的全局 mixin 来实现：

Vue.mixin({

beforeMount () {

const { asyncData } = this.$options

if (asyncData) {

// 将获取数据操作分配给 promise

// 以便在组件中，我们可以在数据准备就绪后

// 通过运行 `this.dataPromise.then(...)` 来执行其他任务

this.dataPromise = asyncData({

store: this.$store,

route: this.$route

})

}

}

})

这两种策略是根本上不同的用户体验决策，应该根据你创建的应用程序的实际使用场景进行挑选。但是无论你选择哪种策略，当路由组件重用（同一路由，但是 params 或 query 已更改，例如，从 user/1 到 user/2）时，也应该调用 asyncData 函数。我们也可以通过纯客户端(client-only)的全局 mixin 来处理这个问题：

Vue.mixin({

beforeRouteUpdate (to, from, next) {

const { asyncData } = this.$options

if (asyncData) {

asyncData({

store: this.$store,

route: to

}).then(next).catch(next)

} else {

next()

}

}

})

**Store 代码拆分(Store Code Splitting)**

在大型应用程序中，我们的 Vuex store 可能会分为多个模块。当然，也可以将这些模块代码，分割到相应的路由组件 chunk 中。假设我们有以下 store 模块：

// store/modules/foo.js

export default {

namespaced: true,

// 重要信息：state 必须是一个函数，

// 因此可以创建多个实例化该模块

state: () => ({

count: 0

}),

actions: {

inc: ({ commit }) => commit('inc')

},

mutations: {

inc: state => state.count++

}

}

我们可以在路由组件的 asyncData 钩子函数中，使用 store.registerModule 惰性注册(lazy-register)这个模块：

// 在路由组件内

<template>

<div>{{ fooCount }}</div>

</template>

<script>

// 在这里导入模块，而不是在 `store/index.js` 中

import fooStoreModule from '../store/modules/foo'

export default {

asyncData ({ store }) {

store.registerModule('foo', fooStoreModule)

return store.dispatch('foo/inc')

},

// 重要信息：当多次访问路由时，

// 避免在客户端重复注册模块。

destroyed () {

this.$store.unregisterModule('foo')

},

computed: {

fooCount () {

return this.$store.state.foo.count

}

}

}

</script>

由于模块现在是路由组件的依赖，所以它将被 webpack 移动到路由组件的异步 chunk 中。

# 客户端混合

在 entry-client.js 中，我们用下面这行挂载(mount)应用程序：

// 这里假定 App.vue template 根元素的 `id="app"`

app.$mount('#app')

由于服务器已经渲染好标记(markup)，我们显然无需将其丢弃，然后重新创建所有的 DOM 元素。相反，我们需要"混合"静态标记，然后将它们变为响应式。

如果你检查服务器渲染的输出结果，你会注意到应用程序的根元素有一个特殊的属性：

<div id="app" data-server-rendered="true">

data-server-rendered 特殊属性，让客户端 Vue 知道标记是由服务器渲染，并且应该以混合模式进行挂载。

在开发模式下，Vue 将推断客户端生成的虚拟 DOM 树(virtual DOM tree)，是否与从服务器渲染的 DOM 结构(DOM structure)匹配。如果无法匹配，它将退出混合模式，丢弃现有的 DOM 并从头开始渲染。**在生产模式下，可以禁用此推断，以获得最佳性能。**

### 混合说明(Hydration Caveats)

使用「SSR + 客户端混合」时，需要了解的一件事是，浏览器可能会更改的一些特殊的 HTML 结构。例如，当你在 Vue 模板中写入：

<table>

<tr><td>hi</td></tr>

</table>

浏览器会在 <table> 内部自动注入 <tbody>，然而，由于 Vue 生成的虚拟 DOM(virtual DOM) 不包含 <tbody>，所以会导致无法匹配。为能够正确匹配，请确保在模板中写入有效的 HTML。

**Bundle Renderer 指引**

**使用基本 SSR 的问题**

到目前为止，我们假设打包的服务器端代码，将由服务器通过 require 直接使用：

const createApp = require('/path/to/built-server-bundle.js')

这是理所应当的，然而在每次编辑过应用程序源代码之后，都必须停止并重启服务。这在开发过程中会影响开发效率。此外，Node.js 本身不支持 source map。

**传入 BundleRenderer**

vue-server-renderer 提供一个名为 createBundleRenderer 的 API，用于处理此问题，通过使用 webpack 的自定义插件，server bundle 将生成为可传递到 bundle renderer 的特殊 JSON 文件。所创建的 bundle renderer，用法和普通 renderer 相同，但是 bundle renderer 提供以下优点：

* 内置的 source map 支持（在 webpack 配置中使用 devtool: 'source-map'）
* 在开发环境甚至部署过程中热重载（通过读取更新后的 bundle，然后重新创建 renderer 实例）
* 关键 CSS(critical CSS) 注入（在使用 \*.vue 文件时）：自动内联在渲染过程中用到的组件所需的CSS。更多细节请查看 [**CSS**](https://ssr.vuejs.org/zh/css.html) 章节。
* 使用 [**clientManifest**](https://ssr.vuejs.org/zh/api.html#clientmanifest) 进行资源注入：自动推断出最佳的预加载(preload)和预取(prefetch)指令，以及初始渲染所需的代码分割 chunk。

在下一章节中，我们将讨论如何配置 webpack，以生成 bundle renderer 所需的构建工件(build artifact)，但现在假设我们已经有了这些需要的构建工件，以下就是创建和使用 bundle renderer 的方法：

const { createBundleRenderer } = require('vue-server-renderer')

const renderer = createBundleRenderer(serverBundle, {

runInNewContext: false, // 推荐

template, // （可选）页面模板

clientManifest // （可选）客户端构建 manifest

})

// 在服务器处理函数中……

server.get('\*', (req, res) => {

const context = { url: req.url }

// 这里无需传入一个应用程序，因为在执行 bundle 时已经自动创建过。

// 现在我们的服务器与应用程序已经解耦！

renderer.renderToString(context, (err, html) => {

// 处理异常……

res.end(html)

})

})

bundle renderer 在调用 renderToString 时，它将自动执行「由 bundle 创建的应用程序实例」所导出的函数（传入上下文作为参数），然后渲染它。

# 构建配置

我们假设你已经知道，如何为纯客户端(client-only)项目配置 webpack。服务器端渲染(SSR)项目的配置大体上与纯客户端项目类似，但是我们建议将配置分为三个文件：base, client 和 server。基本配置(base config)包含在两个环境共享的配置，例如，输出路径(output path)，别名(alias)和 loader。服务器配置(server config)和客户端配置(client config)，可以通过使用 [**webpack-merge**](https://github.com/survivejs/webpack-merge) 来简单地扩展基本配置。

## 服务器配置(Server Config)

服务器配置，是用于生成传递给 createBundleRenderer 的 server bundle。它应该是这样的：

const merge = require('webpack-merge')

const nodeExternals = require('webpack-node-externals')

const baseConfig = require('./webpack.base.config.js')

const VueSSRServerPlugin = require('vue-server-renderer/server-plugin')

module.exports = merge(baseConfig, {

// 将 entry 指向应用程序的 server entry 文件

entry: '/path/to/entry-server.js',

// 这允许 webpack 以 Node 适用方式(Node-appropriate fashion)处理动态导入(dynamic import)，

// 并且还会在编译 Vue 组件时，

// 告知 `vue-loader` 输送面向服务器代码(server-oriented code)。

target: 'node',

// 对 bundle renderer 提供 source map 支持

devtool: 'source-map',

// 此处告知 server bundle 使用 Node 风格导出模块(Node-style exports)

output: {

libraryTarget: 'commonjs2'

},

// https://webpack.js.org/configuration/externals/#function

// https://github.com/liady/webpack-node-externals

// 外置化应用程序依赖模块。可以使服务器构建速度更快，

// 并生成较小的 bundle 文件。

externals: nodeExternals({

// 不要外置化 webpack 需要处理的依赖模块。

// 你可以在这里添加更多的文件类型。例如，未处理 \*.vue 原始文件，

// 你还应该将修改 `global`（例如 polyfill）的依赖模块列入白名单

whitelist: /\.css$/

}),

// 这是将服务器的整个输出

// 构建为单个 JSON 文件的插件。

// 默认文件名为 `vue-ssr-server-bundle.json`

plugins: [

new VueSSRServerPlugin()

]

})

在生成 vue-ssr-server-bundle.json 之后，只需将文件路径传递给 createBundleRenderer：

const { createBundleRenderer } = require('vue-server-renderer')

const renderer = createBundleRenderer('/path/to/vue-ssr-server-bundle.json', {

// ……renderer 的其他选项

})

又或者，你还可以将 bundle 作为对象传递给 createBundleRenderer。这对开发过程中的热重新是很有用的 - 具体请查看 HackerNews demo 的[**参考设置**](https://github.com/vuejs/vue-hackernews-2.0/blob/master/build/setup-dev-server.js)。

### 扩展说明(Externals Caveats)

请注意，在 externals 选项中，我们将 CSS 文件列入白名单。这是因为从依赖模块导入的 CSS 还应该由 webpack 处理。如果你导入依赖于 webpack 的任何其他类型的文件（例如 \*.vue, \*.sass），那么你也应该将它们添加到白名单中。

如果你使用 runInNewContext: 'once' 或 runInNewContext: true，那么你还应该将修改 global 的 polyfill 列入白名单，例如 babel-polyfill。这是因为当使用新的上下文模式时，**server bundle 中的代码具有自己的 global 对象。**由于在使用 Node 7.6+ 时，在服务器并不真正需要它，所以实际上只需在客户端 entry 导入它。

## 客户端配置(Client Config)

客户端配置(client config)和基本配置(base config)大体上相同。显然你需要把 entry 指向你的客户端入口文件。除此之外，如果你使用 CommonsChunkPlugin，请确保仅在客户端配置(client config)中使用，因为服务器包需要单独的入口 chunk。

### 生成 clientManifest

**需要版本 2.3.0+**

除了 server bundle 之外，我们还可以生成客户端构建清单(client build manifest)。使用客户端清单(client manifest)和服务器 bundle(server bundle)，renderer 现在具有了服务器和客户端的构建信息，因此它可以自动推断和注入[**资源预加载 / 数据预取指令(preload / prefetch directive)**](https://css-tricks.com/prefetching-preloading-prebrowsing/)，以及 css 链接 / script 标签到所渲染的 HTML。

好处是双重的：

1. 在生成的文件名中有哈希时，可以取代 html-webpack-plugin 来注入正确的资源 URL。
2. 在通过 webpack 的按需代码分割特性渲染 bundle 时，我们可以确保对 chunk 进行最优化的资源预加载/数据预取，并且还可以将所需的异步 chunk 智能地注入为 <script> 标签，以避免客户端的瀑布式请求(waterfall request)，以及改善可交互时间(TTI - time-to-interactive)。

要使用客户端清单(client manifest)，客户端配置(client config)将如下所示：

const webpack = require('webpack')

const merge = require('webpack-merge')

const baseConfig = require('./webpack.base.config.js')

const VueSSRClientPlugin = require('vue-server-renderer/client-plugin')

module.exports = merge(baseConfig, {

entry: '/path/to/entry-client.js',

plugins: [

// 重要信息：这将 webpack 运行时分离到一个引导 chunk 中，

// 以便可以在之后正确注入异步 chunk。

// 这也为你的 应用程序/vendor 代码提供了更好的缓存。

new webpack.optimize.CommonsChunkPlugin({

name: "manifest",

minChunks: Infinity

}),

// 此插件在输出目录中

// 生成 `vue-ssr-client-manifest.json`。

new VueSSRClientPlugin()

]

})

然后，你就可以使用生成的客户端清单(client manifest)以及页面模板：

const { createBundleRenderer } = require('vue-server-renderer')

const template = require('fs').readFileSync('/path/to/template.html', 'utf-8')

const serverBundle = require('/path/to/vue-ssr-server-bundle.json')

const clientManifest = require('/path/to/vue-ssr-client-manifest.json')

const renderer = createBundleRenderer(serverBundle, {

template,

clientManifest

})

通过以上设置，使用代码分割特性构建后的服务器渲染的 HTML 代码，将看起来如下（所有都是自动注入）：

<html>

<head>

<!-- 用于当前渲染的 chunk 会被资源预加载(preload) -->

<link rel="preload" href="/manifest.js" as="script">

<link rel="preload" href="/main.js" as="script">

<link rel="preload" href="/0.js" as="script">

<!-- 未用到的异步 chunk 会被数据预取(preload)（次要优先级） -->

<link rel="prefetch" href="/1.js" as="script">

</head>

<body>

<!-- 应用程序内容 -->

<div data-server-rendered="true"><div>async</div></div>

<!-- manifest chunk 优先 -->

<script src="/manifest.js"></script>

<!-- 在主 chunk 之前注入异步 chunk -->

<script src="/0.js"></script>

<script src="/main.js"></script>

</body>

</html>`

### 手动资源注入(Manual Asset Injection)

默认情况下，当提供 template 渲染选项时，资源注入是自动执行的。但是有时候，你可能需要对资源注入的模板进行更细粒度(finer-grained)的控制，或者你根本不使用模板。在这种情况下，你可以在创建 renderer 并手动执行资源注入时，传入 inject: false。

在 renderToString 回调函数中，你传入的 context 对象会暴露以下方法：

* context.renderStyles()

这将返回内联 <style> 标签包含所有关键 CSS(critical CSS) ，其中关键 CSS 是在要用到的 \*.vue 组件的渲染过程中收集的。有关更多详细信息，请查看 [**CSS 管理**](https://ssr.vuejs.org/zh/css.html)。

如果提供了 clientManifest，返回的字符串中，也将包含着 <link rel="stylesheet"> 标签内由 webpack 输出(webpack-emitted)的 CSS 文件（例如，使用 extract-text-webpack-plugin 提取的 CSS，或使用 file-loader 导入的 CSS）

* context.renderState(options?: Object)

此方法序列化 context.state 并返回一个内联的 script，其中状态被嵌入在 window.\_\_INITIAL\_STATE\_\_ 中。

上下文状态键(context state key)和 window 状态键(window state key)，都可以通过传递选项对象进行自定义：

context.renderState({

contextKey: 'myCustomState',

windowKey: '\_\_MY\_STATE\_\_'

})

// -> <script>window.\_\_MY\_STATE\_\_={...}</script>

* context.renderScripts()
  + 需要 clientManifest

此方法返回引导客户端应用程序所需的 <script> 标签。当在应用程序代码中使用异步代码分割(async code-splitting)时，此方法将智能地正确的推断需要引入的那些异步 chunk。

* context.renderResourceHints()
  + 需要 clientManifest

此方法返回当前要渲染的页面，所需的 <link rel="preload/prefetch"> 资源提示(resource hint)。默认情况下会：

* 预加载页面所需的 JavaScript 和 CSS 文件
* 预取异步 JavaScript chunk，之后可能会用于渲染

使用 [**shouldPreload**](https://ssr.vuejs.org/zh/api.html#shouldpreload) 选项可以进一步自定义要预加载的文件。

* context.getPreloadFiles()
  + 需要 clientManifest

此方法不返回字符串 - 相反，它返回一个数组，此数组是由要预加载的资源文件对象所组成。这可以用在以编程方式(programmatically)执行 HTTP/2 服务器推送(HTTP/2 server push)。

由于传递给 createBundleRenderer 的 template 将会使用 context 对象进行插值，你可以（通过传入 inject: false）在模板中使用这些方法：

<html>

<head>

<!-- 使用三花括号(triple-mustache)进行 HTML 不转义插值(non-HTML-escaped interpolation) -->

{{{ renderResourceHints() }}}

{{{ renderStyles() }}}

</head>

<body>

<!--vue-ssr-outlet-->

{{{ renderState() }}}

{{{ renderScripts() }}}

</body>

</html>

如果你根本没有使用 template，你可以自己拼接字符串。

**CSS 管理**

管理 CSS 的推荐方法是简单地使用 \*.vue 单个文件组件内的 <style>，它提供：

* 与 HTML 并列同级，组件作用域 CSS
* 能够使用预处理器(pre-processor)或 PostCSS
* 开发过程中热重载(hot-reload)

更重要的是，vue-style-loader（vue-loader 内部使用的 loader），具备一些服务器端渲染的特殊功能：

* 客户端和服务器端的通用编程体验。
* 在使用 bundleRenderer 时，自动注入关键 CSS(critical CSS)。

如果在服务器端渲染期间使用，可以在 HTML 中收集和内联（使用 template 选项时自动处理）组件的 CSS。在客户端，当第一次使用该组件时，vue-style-loader 会检查这个组件是否已经具有服务器内联(server-inlined)的 CSS - 如果没有，CSS 将通过 <style> 标签动态注入。

* 通用 CSS 提取。

此设置支持使用 [**extract-text-webpack-plugin**](https://github.com/webpack-contrib/extract-text-webpack-plugin) 将主 chunk(main chunk) 中的 CSS 提取到单独的 CSS 文件中（使用 template 自动注入），这样可以将文件分开缓存。建议用于存在很多公用 CSS 时。

内部异步组件中的 CSS 将内联为 JavaScript 字符串，并由 vue-style-loader 处理。

**启用 CSS 提取**

要从 \*.vue 文件中提取 CSS，可以使用 vue-loader 的 extractCSS 选项（需要 vue-loader 12.0.0+）

// webpack.config.js

const ExtractTextPlugin = require('extract-text-webpack-plugin')

// CSS 提取应该只用于生产环境

// 这样我们在开发过程中仍然可以热重载

const isProduction = process.env.NODE\_ENV === 'production'

module.exports = {

// ...

module: {

rules: [

{

test: /\.vue$/,

loader: 'vue-loader',

options: {

// enable CSS extraction

extractCSS: isProduction

}

},

// ...

]

},

plugins: isProduction

// make sure to add the plugin!

? [new ExtractTextPlugin({ filename: 'common.[chunkhash].css' })]

: []

}

请注意，上述配置仅适用于 \*.vue 文件中的样式，然而你也可以使用 <style src="./foo.css"> 将外部 CSS 导入 Vue 组件。

如果你想从 JavaScript 中导入 CSS，例如，import 'foo.css'，你需要配置合适的 loader：

module.exports = {

// ...

module: {

rules: [

{

test: /\.css$/,

// 重要：使用 vue-style-loader 替代 style-loader

use: isProduction

? ExtractTextPlugin.extract({

use: 'css-loader',

fallback: 'vue-style-loader'

})

: ['vue-style-loader', 'css-loader']

}

]

},

// ...

}

**从依赖模块导入样式**

从 NPM 依赖模块导入 CSS 时需要注意的几点：

1. 在服务器端构建过程中，不应该外置化提取。
2. 如果使用 CSS 提取 + 使用 CommonsChunkPlugin 插件提取 vendor，在 extract-text-webpack-plugin 提取 CSS 到 vendor chunk 时将遇到问题。为了应对这个问题，请避免在 vendor chunk 中包含 CSS 文件。客户端 webpack 配置示例如下：

module.exports = {

// ...

plugins: [

// 将依赖模块提取到 vendor chunk 以获得更好的缓存，是很常见的做法。

new webpack.optimize.CommonsChunkPlugin({

name: 'vendor',

minChunks: function (module) {

// 一个模块被提取到 vendor chunk 时……

return (

// 如果它在 node\_modules 中

/node\_modules/.test(module.context) &&

// 如果 request 是一个 CSS 文件，则无需外置化提取

!/\.css$/.test(module.request)

)

}

}),

// 提取 webpack 运行时和 manifest

new webpack.optimize.CommonsChunkPlugin({

name: 'manifest'

}),

// ...

]

}

**Head 管理**

类似于资源注入，Head 管理遵循相同的理念：我们可以在组件的生命周期中，将数据动态地追加到渲染上下文(render context)，然后在模板中的占位符替换为这些数据。

**在 2.3.2+ 的版本，你可以通过 this.$ssrContext 来直接访问组件中的服务器端渲染上下文(SSR context)。在旧版本中，你必须通过将其传递给 createApp() 并将其暴露于根实例的 $options 上，才能手动注入服务器端渲染上下文(SSR context) - 然后子组件可以通过 this.$root.$options.ssrContext 来访问它。**

我们可以编写一个简单的 mixin 来完成标题管理：

// title-mixin.js

function getTitle (vm) {

// 组件可以提供一个 `title` 选项

// 此选项可以是一个字符串或函数

const { title } = vm.$options

if (title) {

return typeof title === 'function'

? title.call(vm)

: title

}

}

const serverTitleMixin = {

created () {

const title = getTitle(this)

if (title) {

this.$ssrContext.title = title

}

}

}

const clientTitleMixin = {

mounted () {

const title = getTitle(this)

if (title) {

document.title = title

}

}

}

// 可以通过 `webpack.DefinePlugin` 注入 `VUE\_ENV`

export default process.env.VUE\_ENV === 'server'

? serverTitleMixin

: clientTitleMixin

现在，路由组件可以利用以上 mixin，来控制文档标题(document title)：

// Item.vue

export default {

mixins: [titleMixin],

title () {

return this.item.title

}

asyncData ({ store, route }) {

return store.dispatch('fetchItem', route.params.id)

},

computed: {

item () {

return this.$store.state.items[this.$route.params.id]

}

}

}

然后 template 的内容将会传递给 bundle renderer：

<html>

<head>

<title>{{ title }}</title>

</head>

<body>

...

</body>

</html>

**注意：**

* 使用双花括号(double-mustache)进行 HTML 转义插值(HTML-escaped interpolation)，以避免 XSS 攻击。
* 你应该在创建 context 对象时提供一个默认标题，以防在渲染过程中组件没有设置标题。

使用相同的策略，你可以轻松地将此 mixin 扩展为通用的头部管理工具(generic head management utility)。

# 缓存

虽然 Vue 的服务器端渲染(SSR)相当快速，但是由于创建组件实例和虚拟 DOM 节点的开销，无法与纯基于字符串拼接(pure string-based)的模板的性能相当。在 SSR 性能至关重要的情况下，明智地利用缓存策略，可以极大改善响应时间并减少服务器负载。

## 页面级别缓存(Page-level Caching)

在大多数情况下，服务器渲染的应用程序依赖于外部数据，因此本质上页面内容是动态的，不能持续长时间缓存。然而，如果内容不是用户特定(user-specific)（即对于相同的 URL，总是为所有用户渲染相同的内容），我们可以利用名为 [**micro-caching**](https://www.nginx.com/blog/benefits-of-microcaching-nginx/) 的缓存策略，来大幅度提高应用程序处理高流量的能力。

这通常在 Nginx 层完成，但是我们也可以在 Node.js 中实现它：

const microCache = LRU({

max: 100,

maxAge: 1000 // 重要提示：条目在 1 秒后过期。

})

const isCacheable = req => {

// 实现逻辑为，检查请求是否是用户特定(user-specific)。

// 只有非用户特定(non-user-specific)页面才会缓存

}

server.get('\*', (req, res) => {

const cacheable = isCacheable(req)

if (cacheable) {

const hit = microCache.get(req.url)

if (hit) {

return res.end(hit)

}

}

renderer.renderToString((err, html) => {

res.end(html)

if (cacheable) {

microCache.set(req.url, html)

}

})

})

由于内容缓存只有一秒钟，用户将无法查看过期的内容。然而，这意味着，对于每个要缓存的页面，服务器最多只能每秒执行一次完整渲染。

## 组件级别缓存(Component-level Caching)

vue-server-renderer 内置支持组件级别缓存(component-level caching)。要启用组件级别缓存，你需要在创建 renderer 时提供[**具体缓存实现方式(cache implementation)**](https://ssr.vuejs.org/zh/api.html#cache)。典型做法是传入 [**lru-cache**](https://github.com/isaacs/node-lru-cache)：

const LRU = require('lru-cache')

const renderer = createRenderer({

cache: LRU({

max: 10000,

maxAge: ...

})

})

然后，你可以通过实现 serverCacheKey 函数来缓存组件。

export default {

name: 'item', // 必填选项

props: ['item'],

serverCacheKey: props => props.item.id,

render (h) {

return h('div', this.item.id)

}

}

请注意，可缓存组件**还必须定义一个唯一的 name 选项**。通过使用唯一的名称，每个缓存键(cache key)对应一个组件：你无需担心两个组件返回同一个 key。

serverCacheKey 返回的 key 应该包含足够的信息，来表示渲染结果的具体情况。如果渲染结果仅由 props.item.id决定，则上述是一个很好的实现。但是，如果具有相同 id 的 item 可能会随时间而变化，或者如果渲染结果依赖于其他 prop，则需要修改 getCacheKey 的实现，以考虑其他变量。

返回常量将导致组件始终被缓存，这对纯静态组件是有好处的。

### 何时使用组件缓存

如果 renderer 在组件渲染过程中进行缓存命中，那么它将直接重新使用整个子树的缓存结果。这意味着在以下情况，你**不**应该缓存组件：

* 它具有可能依赖于全局状态的子组件。
* 它具有对渲染上下文产生副作用(side effect)的子组件。

因此，应该小心使用组件缓存来解决性能瓶颈。在大多数情况下，你不应该也不需要缓存单一实例组件。适用于缓存的最常见类型的组件，是在大的 v-for 列表中重复出现的组件。由于这些组件通常由数据库集合(database collection)中的对象驱动，它们可以使用简单的缓存策略：使用其唯一 id，再加上最后更新的时间戳，来生成其缓存键(cache key)：

# Streaming

对于 vue-server-renderer 的基本 renderer 和 bundle renderer 都提供开箱即用的流式渲染功能。所有你需要做的就是，用 renderToStream 替代 renderToString：

const stream = renderer.renderToStream(context)

返回的值是 [**Node.js stream**](https://nodejs.org/api/stream.html)：

let html = ''

stream.on('data', data => {

html += data.toString()

})

stream.on('end', () => {

console.log(html) // 渲染完成

})

stream.on('error', err => {

// handle error...

})

## 流式传输说明(Streaming Caveats)

在流式渲染模式下，当 renderer 遍历虚拟 DOM 树(virtual DOM tree)时，会尽快发送数据。这意味着我们可以尽快获得"第一个 chunk"，并开始更快地将其发送给客户端。

然而，当第一个数据 chunk 被发出时，子组件甚至可能不被实例化，它们的生命周期钩子也不会被调用。这意味着，如果子组件需要在其生命周期钩子函数中，将数据附加到渲染上下文(render context)，当流(stream)启动时，这些数据将不可用。这是因为，大量上下文信息(context information)（如头信息(head information)或内联关键 CSS(inline critical CSS)）需要在应用程序标记(markup)之前出现，我们基本上必须等待流(stream)完成后，才能开始使用这些上下文数据。

因此，如果你依赖由组件生命周期钩子函数填充的上下文数据，则**不建议**使用流式传输模式。

# API 参考

## createRenderer([options])

使用（可选的）[**选项**](https://ssr.vuejs.org/zh/api.html#renderer-options)创建一个 [**Renderer**](https://ssr.vuejs.org/zh/api.html#class-renderer) 实例。

const { createRenderer } = require('vue-server-renderer')

const renderer = createRenderer({ ... })

## createBundleRenderer(bundle[, options])

使用 server bundle 和（可选的）[**选项**](https://ssr.vuejs.org/zh/api.html#renderer-options)创建一个 [**BundleRenderer**](https://ssr.vuejs.org/zh/api.html#class-bundlerenderer) 实例。

const { createBundleRenderer } = require('vue-server-renderer')

const renderer = createBundleRenderer(serverBundle, { ... })

serverBundle 参数可以是以下之一：

* 绝对路径，指向一个已经构建好的 bundle 文件（.js 或 .json）。必须以 / 开头才会被识别为文件路径。
* 由 webpack + vue-server-renderer/server-plugin 生成的 bundle 对象。
* JavaScript 代码字符串（不推荐）。

更多细节请查看 [**Server Bundle 指引**](https://ssr.vuejs.org/zh/bundle-renderer.html) 和 [**构建配置**](https://ssr.vuejs.org/zh/build-config.html)。

## Class: Renderer

#### renderer.renderToString(vm[, context], callback)

将 Vue 实例渲染为字符串。上下文对象 (context object) 可选。回调函数是典型的 Node.js 风格回调，其中第一个参数是可能抛出的错误，第二个参数是渲染完毕的字符串。

#### renderer.renderToStream(vm[, context])

将 Vue 实例渲染为一个 Node.js 流 (stream)。上下文对象 (context object) 可选。更多细节请查看[**流式渲染**](https://ssr.vuejs.org/zh/streaming.html)。

## Class: BundleRenderer

#### bundleRenderer.renderToString([context, ]callback)

将 bundle 渲染为字符串。上下文对象 (context object) 可选。回调是一个典型的 Node.js 风格回调，其中第一个参数是可能抛出的错误，第二个参数是渲染完毕的字符串。

#### bundleRenderer.renderToStream([context])

将 bundle 渲染为一个 Node.js 流 (stream). 上下文对象 (context object) 可选。更多细节请查看[**流式渲染**](https://ssr.vuejs.org/zh/streaming.html)。

## Renderer 选项

#### template

为整个页面的 HTML 提供一个模板。此模板应包含注释 <!--vue-ssr-outlet-->，作为渲染应用内容的占位符。

模板还支持使用渲染上下文 (render context) 进行基本插值：

* 使用双花括号 (double-mustache) 进行 HTML 转义插值 (HTML-escaped interpolation)；
* 使用三花括号 (triple-mustache) 进行 HTML 不转义插值 (non-HTML-escaped interpolation)。

当在渲染上下文 (render context) 上存在一些特定属性时，模板会自动注入对应的内容：

* context.head：（字符串）将会被作为 HTML 注入到页面的头部 (head) 里。
* context.styles：（字符串）内联 CSS，将以 style 标签的形式注入到页面头部。注意，如过你使用了 vue-loader + vue-style-loader 来处理组件 CSS，此属性会在构建过程中被自动生成。
* context.state：（对象）初始 Vuex store 状态，将以 window.\_\_INITIAL\_STATE\_\_ 的形式内联到页面。内联的 JSON 将使用 [**serialize-javascript**](https://github.com/yahoo/serialize-javascript) 自动清理，以防止 XSS 攻击。

此外，当提供 clientManifest 时，模板会自动注入以下内容：

* 渲染当前页面所需的最优客户端 JavaScript 和 CSS 资源（支持自动推导异步代码分割所需的文件）
* 为要渲染页面提供最佳的 <link rel="preload/prefetch"> 资源提示(resource hints)。

你也可以通过将 inject: false 传递给 renderer，来禁用所有自动注入。

具体查看：

* [**使用一个页面模板**](https://ssr.vuejs.org/zh/basic.html#using-a-page-template)
* [**手动资源注入(Manual Asset Injection)**](https://ssr.vuejs.org/zh/build-config.html#manual-asset-injection)

#### clientManifest

* 2.3.0+

通过此选项提供一个由 vue-server-renderer/client-plugin 生成的客户端构建 manifest 对象 (client build manifest object)。此对象包含了 webpack 整个构建过程的信息，从而可以让 bundle renderer 自动推导需要在 HTML 模板中注入的内容。更多详细信息，请查看[**生成 clientManifest**](https://ssr.vuejs.org/zh/build-config.html#generating-clientmanifest)。

#### inject

* + 2.3.0+

控制使用 template 时是否执行自动注入。默认是 true。

参考：[**手动资源注入(Manual Asset Injection)**](https://ssr.vuejs.org/zh/build-config.html#manual-asset-injection)。

#### shouldPreload

* + 2.3.0+

一个函数，用来控制什么文件应该生成 <link rel="preload"> 资源预加载提示 (resource hints)。

默认情况下，只有 JavaScript 和 CSS 文件会被预加载，因为它们是启动应用时所必需的。

对于其他类型的资源（如图像或字体），预加载过多可能会浪费带宽，甚至损害性能，因此预加载什么资源具体依赖于场景。你可以使用 shouldPreload 选项精确控制预加载资源：

const renderer = createBundleRenderer(bundle, {

template,

clientManifest,

shouldPreload: (file, type) => {

// 基于文件扩展名的类型推断。

// https://fetch.spec.whatwg.org/#concept-request-destination

if (type === 'script' || type === 'style') {

return true

}

if (type === 'font') {

// only preload woff2 fonts

return /\.woff2$/.test(file)

}

if (type === 'image') {

// only preload important images

return file === 'hero.jpg'

}

}

})

#### runInNewContext

* + 2.3.0+
  + 只用于 createBundleRenderer
  + Expects: boolean | 'once' ('once' 仅在 2.3.1+ 中支持)

默认情况下，对于每次渲染，bundle renderer 将创建一个新的 V8 上下文并重新执行整个 bundle。这具有一些好处 - 例如，应用程序代码与服务器进程隔离，我们无需担心文档中提到的[**状态单例问题**](https://ssr.vuejs.org/zh/structure.html#avoid-stateful-singletons)。然而，这种模式有一些相当大的性能开销，因为重新创建上下文并执行整个 bundle 还是相当昂贵的，特别是当应用很大的时候。

出于向后兼容的考虑，此选项默认为 true，但建议你尽可能使用 runInNewContext: false 或 runInNewContext: 'once'。

**在 2.3.0 中，此选项有一个 bug，其中 runInNewContext: false 仍然使用独立的全局上下文(separate global context)执行 bundle。以下信息假定版本为 2.3.1+。**

使用 runInNewContext: false，bundle 代码将与服务器进程在同一个 global 上下文中运行，所以请留意在应用程序代码中尽量避免修改 global。

使用 runInNewContext: 'once' (2.3.1+)，bundle 将在独立的全局上下文 (separate global context) 取值，然而只在启动时取值一次。这提供了一定程度的应用程序代码隔离，因为它能够防止 bundle 中的代码意外污染服务器进程的 global 对象。注意事项如下：

1. 在此模式下，修改 global（例如，polyfill）的依赖模块必须被打包进 bundle，不能被外部化 (externalize)；
2. 从 bundle 执行返回的值将使用不同的全局构造函数，例如，在服务器进程中捕获到 bundle 内部抛出的错误，使用的是 bundle 上下文中的 Error 构造函数，所以它不会是服务器进程中 Error 的一个实例。

参考：[**源码结构**](https://ssr.vuejs.org/zh/structure.html)

#### basedir

* + 2.2.0+
  + 只用于 createBundleRenderer

显式地声明 server bundle 的运行目录。运行时将会以此目录为基准来解析 node\_modules 中的依赖模块。只有在所生成的 bundle 文件与外部的 NPM 依赖模块放置在不同位置，或者 vue-server-renderer 是通过 NPM link 链接到当前项目中时，才需要配置此选项。

#### cache

提供[**组件缓存**](https://ssr.vuejs.org/zh/caching.html#component-level-caching)具体实现。缓存对象必须实现以下接口（使用 Flow 语法表示）：

type RenderCache = {

get: (key: string, cb?: Function) => string | void;

set: (key: string, val: string) => void;

has?: (key: string, cb?: Function) => boolean | void;

};

典型用法是传入 [**lru-cache**](https://github.com/isaacs/node-lru-cache)：

const LRU = require('lru-cache')

const renderer = createRenderer({

cache: LRU({

max: 10000

})

})

请注意，缓存对象应至少要实现 get 和 set。此外，如果 get 和 has 接收第二个参数作为回调，那 get 和 has 也可以是可选的异步函数。这允许缓存使用异步 API，例如，一个 Redis 客户端：

const renderer = createRenderer({

cache: {

get: (key, cb) => {

redisClient.get(key, (err, res) => {

// 处理任何错误

cb(res)

})

},

set: (key, val) => {

redisClient.set(key, val)

}

}

})

#### directives

对于自定义指令，允许提供服务器端实现：

const renderer = createRenderer({

directives: {

example (vnode, directiveMeta) {

// 基于指令绑定元数据(metadata)转换 vnode

}

}

})

例如，请查看 [**v-show 的服务器端实现**](https://github.com/vuejs/vue/blob/dev/src/platforms/web/server/directives/show.js)。

## webpack 插件

webpack 插件作为独立文件提供，并且应当直接 require：

const VueSSRServerPlugin = require('vue-server-renderer/server-plugin')

const VueSSRClientPlugin = require('vue-server-renderer/client-plugin')

生成的默认文件是：

* vue-ssr-server-bundle.json 用于服务器端插件；
* vue-ssr-client-manifest.json 用于客户端插件。

创建插件实例时可以自定义文件名：

const plugin = new VueSSRServerPlugin({

filename: 'my-server-bundle.json'

})

更多信息请查看[**构建配置**](https://ssr.vuejs.org/zh/build-config.html)。