



下载APP



## 38 | 服务端渲染原理：Vue 3中的SSR是如何实现的？

2022-01-17 大圣

《玩转Vue 3全家桶》

课程介绍 &gt;



讲述：大圣

时长 10:07 大小 9.28M



你好，我是大圣，上一讲我们学完 vue-router 源码，Vue 全家桶的生态就基本介绍完了，包括 Vue 的响应式、运行时、编译器，以及全家桶的 vuex 和 vue-router。

今天我来给你介绍 Vue 中优化的一个进阶知识点：SSR（Server Side Rendering），也就是服务端渲染。

### SSR 是什么

要想搞清楚 SSR 是什么？我们需要先理解这个方案是为解决什么问题而产生的。



在现在 MVVM 盛行的时代，无论是 Vue 还是 React 的全家桶，都有路由框架的身影，所以，页面的渲染流程也全部都是浏览器加载完 JavaScript 文件后，由 JavaScript 获取当前

的路由地址，再决定渲染哪个页面。

这种架构下，**所有的路由和页面都是在客户端进行解析和渲染的，我们称之为 Client Side Rendering，简称为 CSR，也就是客户端渲染。**

交互体验确实提升了，但同时也带来了两个小问题。

首先，如果采用 CSR，我们在 ailemente 项目中执行 `npm run build` 命令后，可以在项目根目录下看到多了一个 `dist` 文件夹，打开其中的 `index.html` 文件，看到下面的代码：

 复制代码

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <link rel="icon" href="/favicon.ico" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Vite App</title>
8     <script type="module" crossorigin src="/assets/index.c305634d.js"></script>
9     <link rel="modulepreload" href="/assets/vendor.9419ee42.js">
10    <link rel="stylesheet" href="/assets/index.1826a359.css">
11  </head>
12  <body>
13    <div id="app"></div>
14
15  </body>
16 </html>
17
```

这就是项目部署上线之后的入口文件，`body` 内部就是一个空的 `div` 标签，用户访问这个页面后，页面的首屏需要等待 JavaScript 加载和执行完毕才能看到，这样白屏时间肯定比 `body` 内部写页面标签的要长一些，尤其在客户端网络环境差的情况下，等待 JavaScript 下载和执行的白屏时间是很伤害用户体验的。

其次，搜索引擎的爬虫抓取到你的页面数据后，发现 `body` 是空的，也会认为你这个页面是空的，这对于 SEO 是很不利的。即使现在基于 Google 的搜索引擎爬虫已经能够支持 JavaScript 的执行，但是爬虫不会等待页面的网络数据请求，何况国内主要的搜索引擎还是百度。

所以如果你的项目对白屏时间和搜索引擎有要求，我们就需要在用户访问页面的时候，能够把首屏渲染的 HTML 内容写入到 body 内部，也就是说我们需要在服务器端实现组件的渲染，这就是 SSR 的用武之地。

## 怎么做 SSR

那怎么在服务器端实现组件渲染呢？Vue 提供了 @vue/server-renderer 这个专门做服务端解析的库，我们来尝试使用一下。

首先创建一个新的文件夹 vue-ssr，执行下面命令来安装 server-renderer、vue 和 express：

[复制代码](#)

```
1 npm init -y
2 npm install @vue/server-renderer vue@next express --save
```

然后新建 server.js，核心就是要实现在服务器端解析 Vue 的组件，直接把渲染结果返回给浏览器。

下面的代码中我们使用 express 启动了一个服务器，监听 9093 端口，在用户访问首页的时候，通过 createSSRApp 创建一个 Vue 的实例，并且通过 @vue/compiler-ssr 对模板的 template 进行编译，返回的函数配置在 vueapp 的 ssrRender 属性上，最后通过 @vue/server-renderer 的 renderToString 方法渲染 Vue 的实例，把 renderToString 返回的字符串通过 res.send 返回给客户端。

[复制代码](#)

```
1 // 引入express
2 const express = require('express')
3 const app = express()
4 const Vue = require('vue') // vue@next
5 const renderer3 = require('@vue/server-renderer')
6 const vue3Compile= require('@vue/compiler-ssr')
7
8 // 一个vue的组件
9 const vueapp = {
10   template: `<div>
11     <h1 @click="add">{{num}}</h1>
12     <ul >
13       <li v-for="(todo,n) in todos" >{{n+1}}--{{todo}}</li>
```

```
14     </ul>
15   </div>`,
16   data(){
17     return {
18       num:1,
19       todos:['吃饭','睡觉','学习Vue']
20     }
21   },
22   methods:{
23     add(){
24       this.num++
25     }
26   }
27 }
28 // 使用@vue/compiler-ssr解析template
29 vueapp.ssrRender = new Function('require',vue3Compile.compile(vueapp.template)
30 // 路由首页返回结果
31 app.get('/',async function(req,res){
32   let vapp = Vue.createSSRApp(vueapp)
33   let html = await renderer3.renderToString(vapp)
34   const title = "Vue SSR"
35   let ret = `
36   <!DOCTYPE html>
37   <html lang="en">
38     <head>
39       <meta charset="UTF-8" />
40       <link rel="icon" href="/favicon.ico" />
41       <meta name="viewport" content="width=device-width, initial-scale=1.0" />
42       <title>${title}</title>
43     </head>
44     <body>
45       <div id="app">
46         ${html}
47       </div>
48     </body>
49   </html>`
50   res.send(ret)
51 })
52
53 app.listen(9093,()=>{
54   console.log('listen 9093')
55 })
```

现在我们访问页面后，点击右键查看网页源代码，会出现下图所示的页面：

```
1
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5   <meta charset="UTF-8" />
6   <link rel="icon" href="/favicon.ico" />
7   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8   <title>Vue SSR</title>
9 </head>
10 <body>
11   <div id="app">
12     <div><h1>1</h1><ul><!--[--><li>1--吃饭</li><li>2--睡觉</li><li>3--学习Vue</li><!--[--></ul></div>
13   </div>
14 </body>
15 </html>
```

可以看到，首屏的 body 标签内部就出现了 vue 组件中 v-for 渲染后的标签结果，我们的第一步就完成了。

但具体 SSR 是怎么实现的呢？我们一起来看源码。

## Vue SSR 源码剖析

在 CSR 环境下，template 解析的 render 函数用来返回组件的虚拟 DOM，而 SSR 环境下 template 解析的 ssrRender 函数，函数内部是通过 \_push 对字符串进行拼接，最终生成组件渲染的结果的。你可以在官方的 [🔗 模板渲染演示页面](#) 选择 ssr 设置后，看到渲染的结果：

📋 复制代码

```
1 const { mergeProps: _mergeProps } = require("vue")
2 const { ssrRenderAttrs: _ssrRenderAttrs, ssrInterpolate: _ssrInterpolate, ssrR
3
4 return function ssrRender(_ctx, _push, _parent, _attrs, $props, $setup, $data,
5   const _cssVars = { style: { color: _ctx.color }}
6   _push(`<div${_ssrRenderAttrs(_mergeProps(_attrs, _cssVars))}><ul><!--[-->`)
7   _ssrRenderList(_ctx.todos, (todo, n) => {
8     _push(`<li>${
9       _ssrInterpolate(n+1)
10     }--${
11       _ssrInterpolate(todo)
12     }</li>`)
13   })
14   _push(`<!--[--></ul></div>`)
15 }
```

可以看到 ssrRender 函数内部通过传递的 \_push 函数拼接组件渲染的结果后，直接返回 renderToString 函数的执行结果。



那 `renderToString` 是如何工作的呢？

现在你已经拥有了源码阅读的技巧，我们进入到 `vue-next/packages/server-renderer` 文件中，打开 **`renderToString` 文件**：

[复制代码](#)

```
1 export async function renderToString(  
2   input: App | VNode,  
3   context: SSRContext = {}  
4 ): Promise<string> {  
5   if (isVNode(input)) {  
6     // raw vnode, wrap with app (for context)  
7     return renderToString(createApp({ render: () => input }), context)  
8   }  
9   const vnode = createVNode(input._component, input._props)  
10  vnode.appContext = input._context  
11  // provide the ssr context to the tree  
12  input.provide(ssrContextKey, context)  
13  const buffer = await renderComponentVNode(vnode)  
14  
15  await resolveTeleports(context)  
16  
17  return unrollBuffer(buffer as SSRBuffer)  
18 }
```

这段代码可以看到，我们通过 `renderComponentVNode` 函数对创建的 `Vnode` 进行渲染，生成一个 `buffer` 变量，最后通过 `unrollBuffer` 返回字符串。


我们先继续看 **`renderComponentVNode` 函数**，它内部通过 `renderComponentSubTree` 进行虚拟 DOM 的子树渲染，而 `renderComponentSubTree` 内部调用组件内部的 `ssrRender` 函数，这个函数就是我们代码中通过 `@vue/compiler-ssr` 解析之后的 `ssrRender` 函数，传递的 `push` 参数是通过 `createBuffer` 传递的：

[复制代码](#)

```
1 export function renderComponentVNode(  
2   vnode: VNode,  
3   parentComponent: ComponentInternalInstance | null = null,  
4   slotScopeId?: string  
5 ): SSRBuffer | Promise<SSRBuffer> {  
6   const instance = createComponentInstance(vnode, parentComponent, null)  
7   const res = setupComponent(instance, true /* isSSR */)
```

```
8   if (hasAsyncSetup || prefetches) {
9     ....
10    return p.then(() => renderComponentSubTree(instance, slotScopeId))
11  } else {
12    return renderComponentSubTree(instance, slotScopeId)
13  }
14 }
15 function renderComponentSubTree(instance, slotScopeId) {
16   const { getBuffer, push } = createBuffer()
17   const ssrRender = instance.ssrRender || comp.ssrRender
18   if (ssrRender) {
19     ssrRender(
20       instance.proxy,
21       push,
22       instance,
23       attrs,
24       // compiler-optimized bindings
25       instance.props,
26       instance.setupState,
27       instance.data,
28       instance.ctx
29     )
30   }
31 }
```

**createBuffer 的实现**也很简单，buffer 是一个数组，push 函数就是不停地在数组最后新增数据，如果 item 是字符串，就在数组最后一个数据上直接拼接字符串，否则就在数组尾部新增一个元素，这种提前合并字符串的做法，也算是一个小优化。

 复制代码

```
1 export function createBuffer() {
2   let appendable = false
3   const buffer: SSRBuffer = []
4   return {
5     getBuffer(): SSRBuffer {
6       // Return static buffer and await on items during unroll stage
7       return buffer
8     },
9     push(item: SSRBufferItem) {
10      const isStringItem = isString(item)
11      if (appendable && isStringItem) {
12        buffer[buffer.length - 1] += item as string
13      } else {
14        buffer.push(item)
15      }
16      appendable = isStringItem
17      if (isPromise(item) || (isArray(item) && item.hasAsync)) {
18        // promise, or child buffer with async, mark as async.
```

```
19      // this allows skipping unnecessary await ticks during unroll stage
20      buffer.hasAsync = true
21    }
22  }
23 }
24 }
```

最后我们看下返回字符串的 **unrollBuffer 函数**，由于 buffer 数组中可能会有异步的组件，服务器返回渲染内容之前，我们要把组件依赖的异步任务使用 await，等待执行完毕后，进行字符串的拼接，最后返回给浏览器。

[复制代码](#)

```
1 async function unrollBuffer(buffer: SSRBuffer): Promise<string> {
2   if (buffer.hasAsync) {
3     let ret = ''
4     for (let i = 0; i < buffer.length; i++) {
5       let item = buffer[i]
6       if (isPromise(item)) {
7         item = await item
8       }
9       if (isString(item)) {
10        ret += item
11      } else {
12        ret += await unrollBuffer(item)
13      }
14    }
15    return ret
16  } else {
17    // sync buffer can be more efficiently unrolled without unnecessary await
18    // ticks
19    return unrollBufferSync(buffer)
20  }
21 }
```

至此我们就把 Vue 中 SSR 的渲染流程梳理完毕了，通过 compiler-ssr 模块把 template 解析成 ssrRender 函数后，整个组件通过 renderToString 把组件渲染成字符串返回给浏览器。

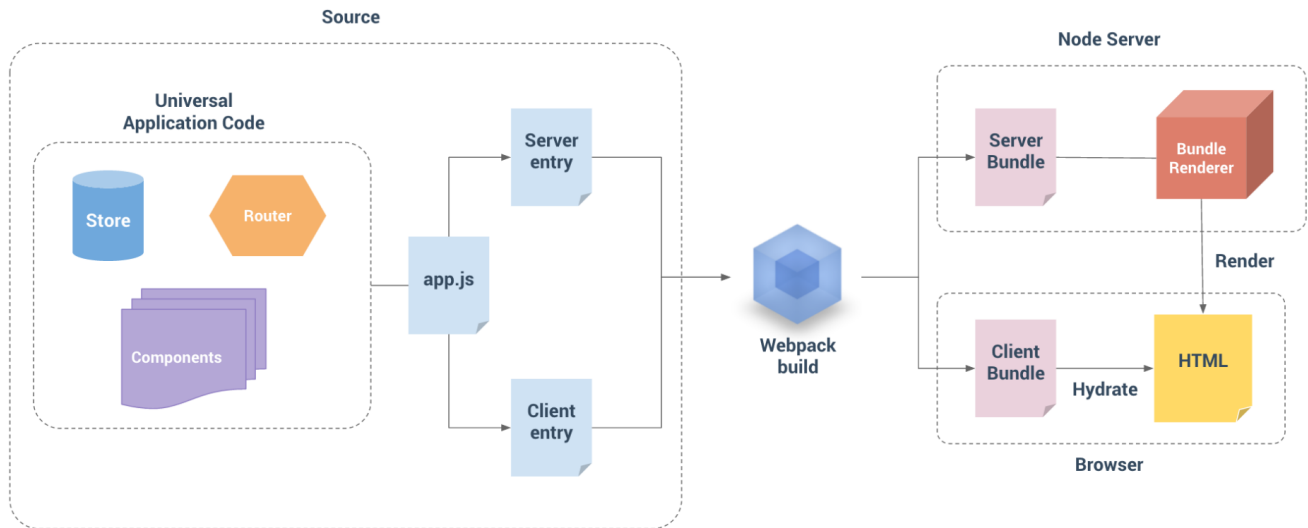
SSR 最终实现了通过服务器端解析 Vue 组件的方式，提高首屏的响应时间和页面的 SEO 友好度。

## 同构应用和其他渲染方式



现在服务器渲染 SSR 的逻辑我们已经掌握了，但是现在页面中没有 JavaScript 的加入，我们既需要提供服务器渲染的首屏内容，又需要 CSR 带来的优秀交互体验，这个时候我们就需要使用同构的方式来构建 Vue 的应用。

什么是同构应用呢？看来自于 Vue 官网的同构应用的经典架构图：



左边是我们的源码，无论项目有多么复杂，都可以拆分为 component + store + router 三大模块。这一部分的源码，设置了两个入口，分别是客户端入口 client entry 和服务端入口 server entry。打包的过程中也有两个打包的配置文件，分别客户端的配置和服务端端的配置。

最终在服务端实现用户首次访问页面的时候通过服务器端入口进入，显示服务器渲染的结果，然后用户在后续的操作中由客户端接管，通过 vue-router 来提高页面跳转的交互体验，这就是**同构应用**的概念。

## SSR+ 同构的问题

当然，没有任何一个技术架构是完美的，SSR 和同构带来了很好的首屏速度和 SEO 友好度，但是也让我们的项目多了一个 Node 服务器模块。

首先，我们部署的难度会提高。之前的静态资源直接上传到服务器的 Nginx 目录下，做好版本管理即可，现在还需要在服务器上部署一个 Node 环境，额外带来了部署和监控的成本，工作量提升了。

其次，SSR 和同构的架构，实际上，是把客户端渲染组件的计算逻辑移到了服务器端执行，在并发量大的场景中，会加大服务器的负载。所以，所有的同构应用下还需要有降级渲染的逻辑，在服务器负载过高或者服务器有异常报错的情况下，让页面恢复为客户端渲染。

总的来说，同构解决问题的同时，也带来了额外的系统复杂度。**每个技术架构的出现都是为了解决一些特定的问题，但是它们的出现也必然会带来新的问题。**

针对同构出现的问题目前也有一些解决方案来应对。

## 解决方案

针对 SSR 架构的问题，我们也可以使用**静态网站生成 ( Static Site Generation , SSG )**的方式来解决，针对页面中变动频率不高的页面，直接渲染成静态页面来展示。

比如极客时间的首页变化频率比较高，每次我们都需要对每个课程的销量和评分进行排序，这部分的每次访问都需要从后端读取数据；但是每个课程内部的页面，比如文章详情页，变化频率其实是很低的，虽然课程的文本是存储在数据库里，但是每次上线前，我们可以把课程详情页生成静态的 HTML 页面再上线。

Vue 的 SSR 框架 nuxt 就提供了很好的 SSG 功能，由于这一部分页面变化频率低，我们静态化之后还可以通过部署到 CDN 来进行页面加速，每次新文章发布或者修改的时候，重新生成一遍即可。

当然 SSG 也不是完全没有问题，比如极客时间如果有一万门课了，每门课几十篇文章，每次部署都全量静态生成一遍，耗时是非常惊人的，所以也不断有新的解决方案出现。

如果你的页面是内嵌在客户端内部的，可以借助客户端的运算能力，把 SSR 的逻辑移动到客户端进行，使用**客户端渲染 ( Native Side Rendering , NSR )**的方式降低服务端的负载，同时也能提高首屏的响应时间。

针对 SSG 全量生成的性能问题，我们可以采用**增量渲染 ( Incremental Site Rendering , ISR )**的方式，每次只生成核心重点的页面，比如每个课程的开篇词，其他的页面访问的时候先通过 CSR 的方式渲染，然后把渲染结果存储在 CDN 中。

现在还有解决方案**边缘渲染 ( Edge Side Rendering , ESR )**，把静态内容和动态的内容都以流的方式返回给用户，在 CDN 节点上返回给用户缓存静态资源，同时在 CDN 上负责发起动态内容的请求。

今年还出现了在浏览器里跑 node 的 [webcontainer](#) 技术，如果这个技术成熟后，我们甚至可以把 Express、Egg.js 等后端应用也部署到 CDN 节点上，在浏览器端实现服务器应用的 ESR，一起期待 webcontainer 技术的发展。

## 总结

我们要聊的内容就讲完了，来回顾一下。

今天我们学习了 Vue 中服务器渲染的原理，Vue 通过 @vue/compiler-ssr 库把 template 解析成 ssrRender 函数，并且用 @vue/server-renderer 库提供了在服务器端渲染组件的能力，让用户访问首屏页面的时候，能够有更快的首屏渲染结果，并且对 SEO 也是友好的，server-renderer 通过提供 renderToString 函数，内部通过管理 buffer 数组实现组件的渲染。

然后我们学习了 SSR 之后的同构、静态网站生成 SSG、增量渲染 ISR 和边缘渲染 ESR 等内容，Vue 中的最成熟的 SSR 框架就是 nuxt 了，最新的 nuxt3 还没有正式发版，内部对于 SSG 和 ESR 都支持，等 nuxt3 发版后你可以自行学习。

每一个技术选型都是为了解决问题存在的，无论学习什么技术，我们都不要单纯地把它当做八股文，这样才能真正掌握好一个技术。

## 思考题

最后留个思考题，你现在负责的项目，是出于什么目的考虑使用 SSR 的呢？欢迎在评论区分享你的思考，我们下一讲再见。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独订阅本课程，你将得 20 元

 生成海报并分享

 赞 2     提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇    37 | 前端路由原理：vue-router源码剖析

下一篇    结束语 | Vue 3生态源码到底给我们带来了什么？

### 精选留言 (1)

 写留言



james

2022-01-19

厉害了，很全面

