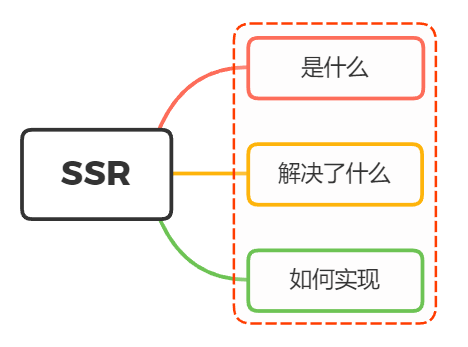
# 面试官：SSR解决了什么问题？有做过SSR吗？你是怎么做的？



## 一、是什么

Server-Side Rendering 我们称其为SSR，意为服务端渲染

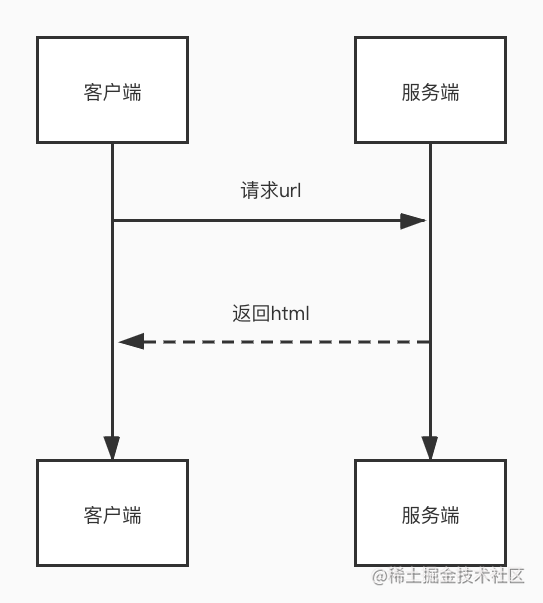
指由服务侧完成页面的 HTML 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程

先来看看Web3个阶段的发展史：

* 传统服务端渲染SSR
* 单页面应用SPA
* 服务端渲染SSR

### **传统web开发**

网页内容在服务端渲染完成，⼀次性传输到浏览器

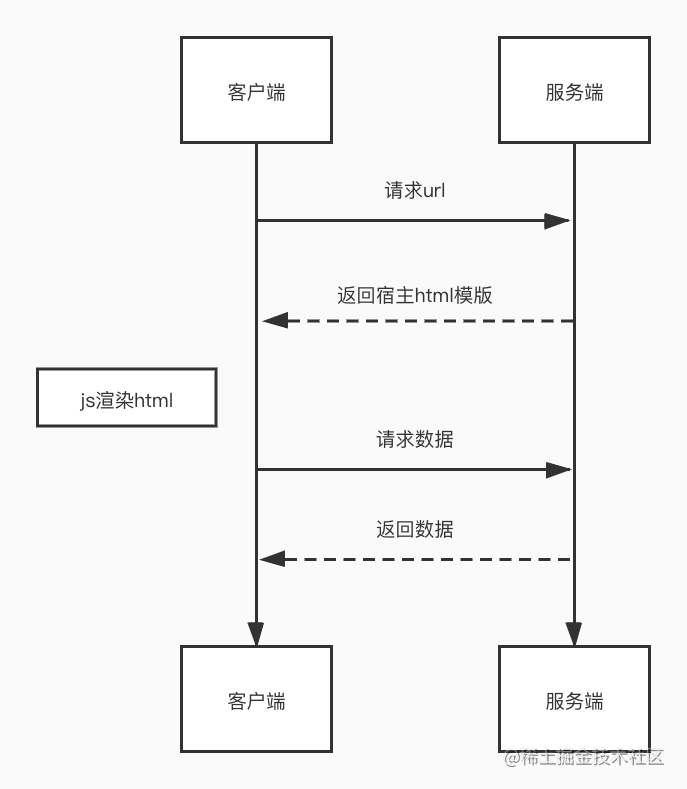


img

打开页面查看源码，浏览器拿到的是全部的dom结构

### **单页应用SPA**

单页应用优秀的用户体验，使其逐渐成为主流，页面内容由JS渲染出来，这种方式称为客户端渲染

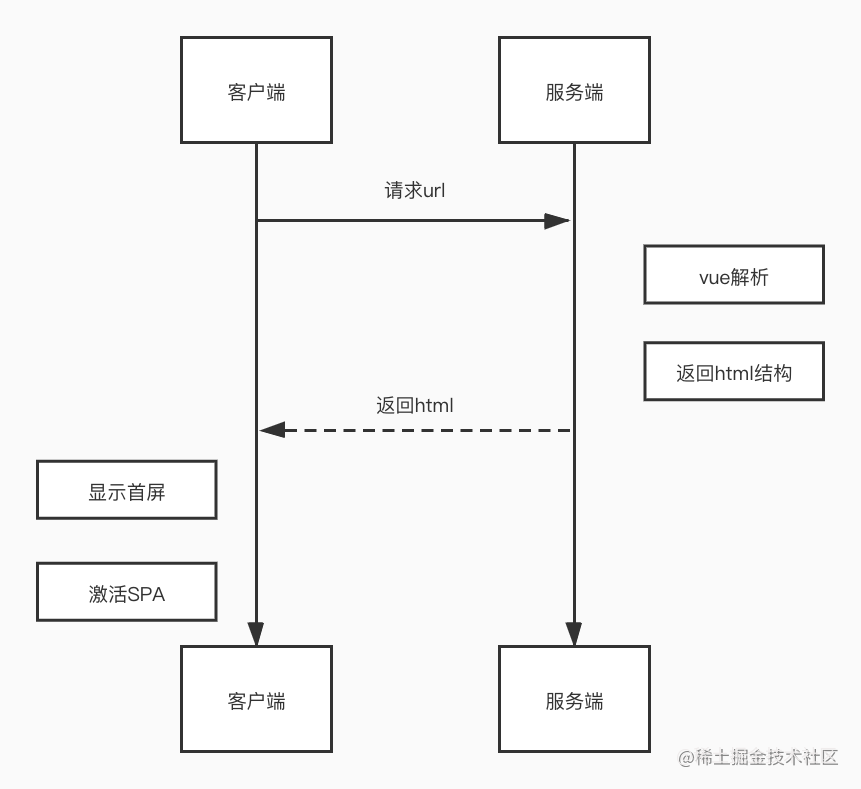


img

打开页面查看源码，浏览器拿到的仅有宿主元素#app，并没有内容

### 服务端渲染SSR

SSR解决方案，后端渲染出完整的首屏的dom结构返回，前端拿到的内容包括首屏及完整spa结构，应用激活后依然按照spa方式运行



img

看完前端发展，我们再看看Vue官方对SSR的解释：

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，最后将这些静态标记"激活"为客户端上完全可交互的应用程序

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

我们从上门解释得到以下结论：

* Vue SSR是一个在SPA上进行改良的服务端渲染
* 通过Vue SSR渲染的页面，需要在客户端激活才能实现交互
* Vue SSR将包含两部分：服务端渲染的首屏，包含交互的SPA

## 二、解决了什么

SSR主要解决了以下两种问题：

* seo：搜索引擎优先爬取页面HTML结构，使用ssr时，服务端已经生成了和业务想关联的HTML，有利于seo
* 首屏呈现渲染：用户无需等待页面所有js加载完成就可以看到页面视图（压力来到了服务器，所以需要权衡哪些用服务端渲染，哪些交给客户端）

但是使用SSR同样存在以下的缺点：

* 复杂度：整个项目的复杂度
* 库的支持性，代码兼容
* 性能问题
* 每个请求都是n个实例的创建，不然会污染，消耗会变得很大
* 缓存 node serve、 nginx判断当前用户有没有过期，如果没过期的话就缓存，用刚刚的结果。
* 降级：监控cpu、内存占用过多，就spa，返回单个的壳
* 服务器负载变大，相对于前后端分离服务器只需要提供静态资源来说，服务器负载更大，所以要慎重使用

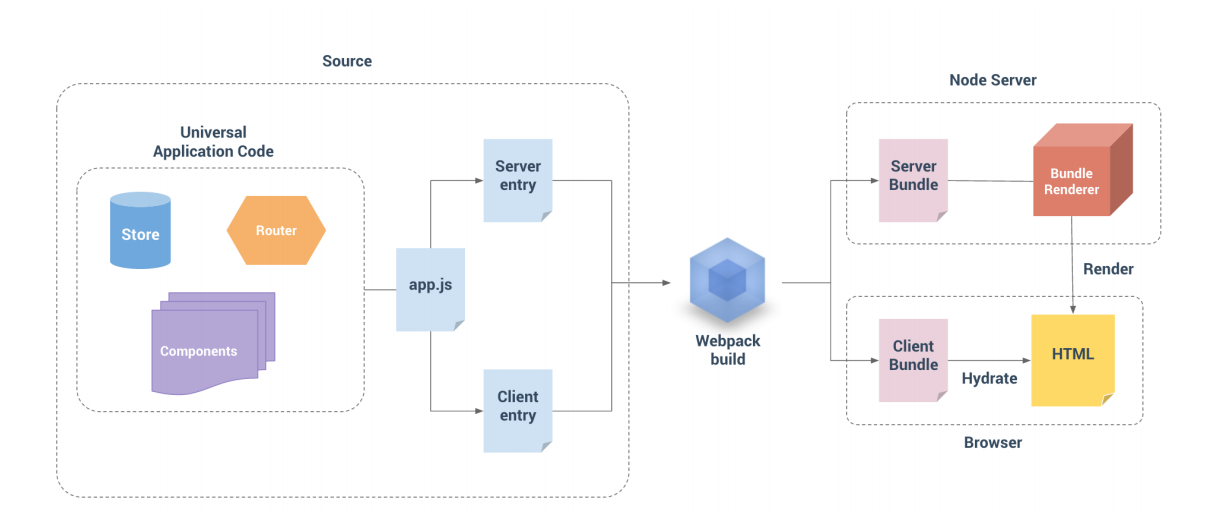
所以在我们选择是否使用SSR前，我们需要慎重问问自己这些问题：

1. 需要SEO的页面是否只是少数几个，这些是否可以使用预渲染（Prerender SPA Plugin）实现
2. 首屏的请求响应逻辑是否复杂，数据返回是否大量且缓慢

## 三、如何实现

对于同构开发，我们依然使用webpack打包，我们要解决两个问题：服务端首屏渲染和客户端激活

这里需要生成一个服务器bundle文件用于服务端首屏渲染和一个客户端bundle文件用于客户端激活



代码结构 除了两个不同入口之外，其他结构和之前vue应用完全相同

src  
├── router  
├────── index.js # 路由声明  
├── store  
├────── index.js # 全局状态  
├── main.js # ⽤于创建vue实例  
├── entry-client.js # 客户端⼊⼝，⽤于静态内容“激活”  
└── entry-server.js # 服务端⼊⼝，⽤于⾸屏内容渲染

路由配置

import Vue from "vue";  
import Router from "vue-router";  
  
Vue.use(Router);  
//导出⼯⼚函数  
  
export function createRouter() {  
 return new Router({  
 mode: 'history',  
 routes: [  
 // 客户端没有编译器，这⾥要写成渲染函数  
 { path: "/", component: { render: h => h('div', 'index page') } },  
 { path: "/detail", component: { render: h => h('div', 'detail page') } }  
 ]  
 });  
}

主文件main.js

跟之前不同，主文件是负责创建vue实例的工厂，每次请求均会有独立的vue实例创建

import Vue from "vue";  
import App from "./App.vue";  
import { createRouter } from "./router";  
// 导出Vue实例⼯⼚函数，为每次请求创建独⽴实例  
// 上下⽂⽤于给vue实例传递参数  
export function createApp(context) {  
 const router = createRouter();  
 const app = new Vue({  
 router,  
 context,  
 render: h => h(App)  
 });  
 return { app, router };  
}

编写服务端入口src/entry-server.js

它的任务是创建Vue实例并根据传入url指定首屏

import { createApp } from "./main";  
// 返回⼀个函数，接收请求上下⽂，返回创建的vue实例  
export default context => {  
 // 这⾥返回⼀个Promise，确保路由或组件准备就绪  
 return new Promise((resolve, reject) => {  
 const { app, router } = createApp(context);  
 // 跳转到⾸屏的地址  
 router.push(context.url);  
 // 路由就绪，返回结果  
 router.onReady(() => {  
 resolve(app);  
 }, reject);  
 });  
};

编写客户端入口entry-client.js

客户端入口只需创建vue实例并执行挂载，这⼀步称为激活

import { createApp } from "./main";  
// 创建vue、router实例  
const { app, router } = createApp();  
// 路由就绪，执⾏挂载  
router.onReady(() => {  
 app.$mount("#app");  
});

对webpack进行配置

安装依赖

npm install webpack-node-externals lodash.merge -D

对vue.config.js进行配置

// 两个插件分别负责打包客户端和服务端  
const VueSSRServerPlugin = require("vue-server-renderer/server-plugin");  
const VueSSRClientPlugin = require("vue-server-renderer/client-plugin");  
const nodeExternals = require("webpack-node-externals");  
const merge = require("lodash.merge");  
// 根据传⼊环境变量决定⼊⼝⽂件和相应配置项  
const TARGET\_NODE = process.env.WEBPACK\_TARGET === "node";  
const target = TARGET\_NODE ? "server" : "client";  
module.exports = {  
 css: {  
 extract: false  
 },  
 outputDir: './dist/'+target,  
 configureWebpack: () => ({  
 // 将 entry 指向应⽤程序的 server / client ⽂件  
 entry: `./src/entry-${target}.js`,  
 // 对 bundle renderer 提供 source map ⽀持  
 devtool: 'source-map',  
 // target设置为node使webpack以Node适⽤的⽅式处理动态导⼊，  
 // 并且还会在编译Vue组件时告知`vue-loader`输出⾯向服务器代码。  
 target: TARGET\_NODE ? "node" : "web",  
 // 是否模拟node全局变量  
 node: TARGET\_NODE ? undefined : false,  
 output: {  
 // 此处使⽤Node⻛格导出模块  
 libraryTarget: TARGET\_NODE ? "commonjs2" : undefined  
 },  
 // https://webpack.js.org/configuration/externals/#function  
 // https://github.com/liady/webpack-node-externals  
 // 外置化应⽤程序依赖模块。可以使服务器构建速度更快，并⽣成较⼩的打包⽂件。  
 externals: TARGET\_NODE  
 ? nodeExternals({  
 // 不要外置化webpack需要处理的依赖模块。  
 // 可以在这⾥添加更多的⽂件类型。例如，未处理 \*.vue 原始⽂件，  
 // 还应该将修改`global`（例如polyfill）的依赖模块列⼊⽩名单  
 whitelist: [/\.css$/]  
 })  
 : undefined,  
 optimization: {  
 splitChunks: undefined  
 },  
 // 这是将服务器的整个输出构建为单个 JSON ⽂件的插件。  
 // 服务端默认⽂件名为 `vue-ssr-server-bundle.json`  
 // 客户端默认⽂件名为 `vue-ssr-client-manifest.json`。  
 plugins: [TARGET\_NODE ? new VueSSRServerPlugin() : new  
 VueSSRClientPlugin()]  
 }),  
 chainWebpack: config => {  
 // cli4项⽬添加  
 if (TARGET\_NODE) {  
 config.optimization.delete('splitChunks')  
 }  
  
 config.module  
 .rule("vue")  
 .use("vue-loader")  
 .tap(options => {  
 merge(options, {  
 optimizeSSR: false  
 });  
 });  
 }  
};

对脚本进行配置，安装依赖

npm i cross-env -D

定义创建脚本package.json

"scripts": {  
 "build:client": "vue-cli-service build",  
 "build:server": "cross-env WEBPACK\_TARGET=node vue-cli-service build",  
 "build": "npm run build:server && npm run build:client"  
}

执行打包：npm run build

最后修改宿主文件/public/index.html

<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <meta charset="utf-8">  
 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
 <meta name="viewport" content="width=device-width,initial-scale=1.0">  
 <title>Document</title>  
 </head>  
 <body>  
 <!--vue-ssr-outlet-->  
 </body>  
</html>

是服务端渲染入口位置，注意不能为了好看而在前后加空格

安装vuex

npm install -S vuex

创建vuex工厂函数

import Vue from 'vue'  
import Vuex from 'vuex'  
Vue.use(Vuex)  
export function createStore () {  
 return new Vuex.Store({  
 state: {  
 count:108  
 },  
 mutations: {  
 add(state){  
 state.count += 1;  
 }  
 }  
 })  
}

在main.js文件中挂载store

import { createStore } from './store'  
export function createApp (context) {  
 // 创建实例  
 const store = createStore()  
 const app = new Vue({  
 store, // 挂载  
 render: h => h(App)  
 })  
 return { app, router, store }  
}

服务器端渲染的是应用程序的"快照"，如果应用依赖于⼀些异步数据，那么在开始渲染之前，需要先预取和解析好这些数据

在store进行一步数据获取

export function createStore() {  
 return new Vuex.Store({  
 mutations: {  
 // 加⼀个初始化  
 init(state, count) {  
 state.count = count;  
 },  
 },  
 actions: {  
 // 加⼀个异步请求count的action  
 getCount({ commit }) {  
 return new Promise(resolve => {  
 setTimeout(() => {  
 commit("init", Math.random() \* 100);  
 resolve();  
 }, 1000);  
 });  
 },  
 },  
 });  
}

组件中的数据预取逻辑

export default {  
 asyncData({ store, route }) { // 约定预取逻辑编写在预取钩⼦asyncData中  
 // 触发 action 后，返回 Promise 以便确定请求结果  
 return store.dispatch("getCount");  
 }  
};

服务端数据预取，entry-server.js

import { createApp } from "./app";  
export default context => {  
 return new Promise((resolve, reject) => {  
 // 拿出store和router实例  
 const { app, router, store } = createApp(context);  
 router.push(context.url);  
 router.onReady(() => {  
 // 获取匹配的路由组件数组  
 const matchedComponents = router.getMatchedComponents();  
  
 // 若⽆匹配则抛出异常  
 if (!matchedComponents.length) {  
 return reject({ code: 404 });  
 }  
  
 // 对所有匹配的路由组件调⽤可能存在的`asyncData()`  
 Promise.all(  
 matchedComponents.map(Component => {  
 if (Component.asyncData) {  
 return Component.asyncData({  
 store,  
 route: router.currentRoute,  
 });  
 }  
 }),  
 )  
 .then(() => {  
 // 所有预取钩⼦ resolve 后，  
 // store 已经填充⼊渲染应⽤所需状态  
 // 将状态附加到上下⽂，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);  
 });  
};

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

// 导出store  
const { app, router, store } = createApp();  
// 当使⽤ template 时，context.state 将作为 window.\_\_INITIAL\_STATE\_\_ 状态⾃动嵌⼊到最终的 HTML   
// 在客户端挂载到应⽤程序之前，store 就应该获取到状态：  
if (window.\_\_INITIAL\_STATE\_\_) {  
 store.replaceState(window.\_\_INITIAL\_STATE\_\_);  
}

客户端数据预取处理，main.js

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,  
 });  
 }  
 },  
});

修改服务器启动文件

// 获取⽂件路径  
const resolve = dir => require('path').resolve(\_\_dirname, dir)  
// 第 1 步：开放dist/client⽬录，关闭默认下载index⻚的选项，不然到不了后⾯路由  
app.use(express.static(resolve('../dist/client'), {index: false}))  
// 第 2 步：获得⼀个createBundleRenderer  
const { createBundleRenderer } = require("vue-server-renderer");  
// 第 3 步：服务端打包⽂件地址  
const bundle = resolve("../dist/server/vue-ssr-server-bundle.json");  
// 第 4 步：创建渲染器  
const renderer = createBundleRenderer(bundle, {  
 runInNewContext: false, // https://ssr.vuejs.org/zh/api/#runinnewcontext  
 template: require('fs').readFileSync(resolve("../public/index.html"), "utf8"), // 宿主⽂件  
 clientManifest: require(resolve("../dist/client/vue-ssr-clientmanifest.json")) // 客户端清单  
});  
app.get('\*', async (req,res)=>{  
 // 设置url和title两个重要参数  
 const context = {  
 title:'ssr test',  
 url:req.url  
 }  
 const html = await renderer.renderToString(context);  
 res.send(html)  
})

### 小结

* 使用ssr不存在单例模式，每次用户请求都会创建一个新的vue实例
* 实现ssr需要实现服务端首屏渲染和客户端激活
* 服务端异步获取数据asyncData可以分为首屏异步获取和切换组件获取
* 首屏异步获取数据，在服务端预渲染的时候就应该已经完成
* 切换组件通过mixin混入，在beforeMount钩子完成数据获取

## 参考文献

* https://juejin.cn/post/6896007907050487816
* https://vue3js.cn/docs/zh