```
server {
    listen  80;
    server_name  www.xxx.com;

    location / {
      index  /data/dist/index.html;
    }
}
```

可以根据 `nginx` 配置得出，当我们在地址栏输入 `www.xxx.com` 时，这时会打开我们 `dist` 目录下的 `index.html` 文件，然后我们在跳转路由进入到 `www.xxx.com/login`

关键在这里，当我们在 `website.com/login` 页执行刷新操作， `nginx location` 是没有相关配置的，所以就会出现 404 的情况

27.2.2. 为什么hash模式下没有问题

`router hash` 模式我们都知道是用符号#表示的，如 `website.com/#/login`，`hash` 的值为 `#/login`

它的特点在于： `hash` 虽然出现在 `URL` 中，但不会被包括在 `HTTP` 请求中，对服务端完全没有影响，因此改变 `hash` 不会重新加载页面

`hash` 模式下，仅 `hash` 符号之前的内容会被包含在请求中，如 `website.com/#/login` 只有 `website.com` 会被包含在请求中，因此对于服务端来说，即使没有配置 `location`，也不会返回404错误

27.3. 解决方案

看到这里我相信大部分同学都能想到怎么解决问题了，

产生问题的本质是因为我们的路由是通过JS来执行视图切换的，

当我们进入到子路由时刷新页面， `web` 容器没有相对应的页面此时会出现404

所以我们只需要配置将任意页面都重定向到 `index.html`，把路由交由前端处理

对 `nginx` 配置文件 `.conf` 修改，添加 `try_files $uri $uri/ /index.html;`

```bash
server {
  listen  80;
  server_name  www.xxx.com;

  location / {
    index  /data/dist/index.html;
    try_files $uri $uri/ /index.html;
  }
}
```

修改完配置文件后记得配置的更新

```bash
nginx -s reload
```

这么做以后，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 `index.html` 文件

为了避免这种情况，你应该在 `Vue` 应用里面覆盖所有的路由情况，然后在给出一个 404 页面

```javascript
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '*', component: NotFoundComponent }
  ]
})
```

28. SSR解决了什么问题？有做过SSR吗？你是怎么做的？

28.1. 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 `Web` 3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

28.1.1. 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

打开页面查看源码，浏览器拿到的是全部的 `dom` 结构

28.1.2. 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

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

28.1.3. 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

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`

28.2. 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

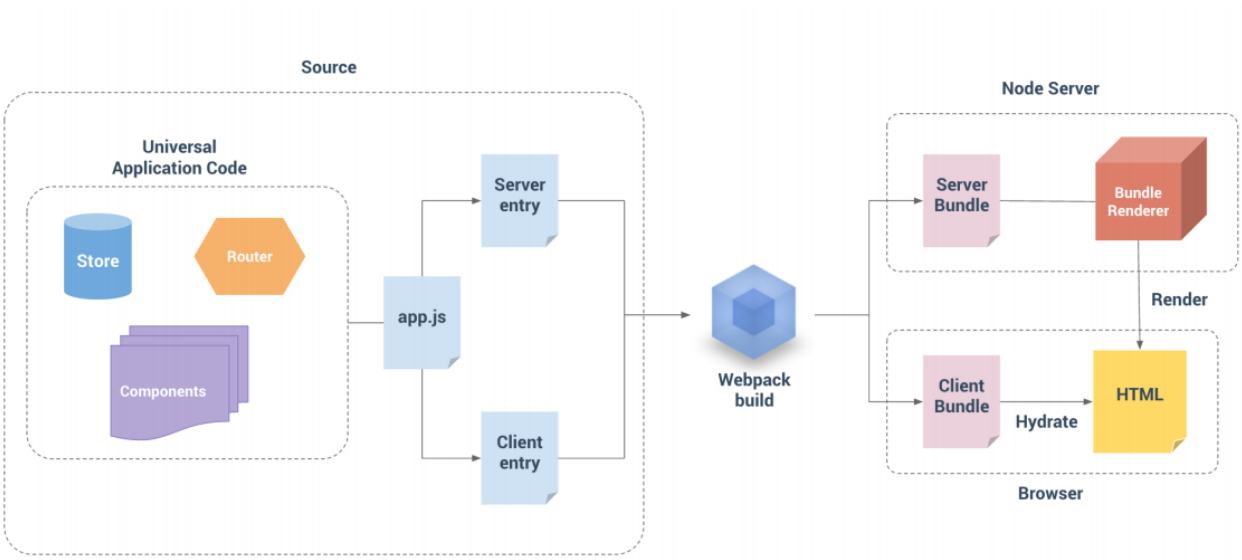
所以在我们选择是否使用 `SSR` 前，我们需要慎重问问自己这些问题：

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

28.3. 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

JavaScript 复制代码

```
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

```JavaScript
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

```JavaScript
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

```JavaScript
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

```JavaScript
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

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

对 `vue.config.js` 进行配置

JavaScript | 复制代码

```
// 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-${target}.js`,
        // 对 bundle renderer 提供 source map 支持
        devtool: 'source-map',
        // target设置为node使webpack以Node适用的方式处理动态导入，
        // 并且还会在编译Vue组件时告知`vue-loader`输出面向服务器代码。
        target: TARGET_NODE ? "node" : "web",
        // 是否模拟node全局变量
        node: TARGET_NODE ? undefined : false,
        output: {
            // 此处使用Node风格导出模块
            libraryTarget: TARGET_NODE ? "commonjs2" : undefined
        },
        // https://webpack.js.org/configuration/externals/#function
        // https://github.com/liady/webpack-node-externals
        // 外置化应用程序依赖模块。可以使服务器构建速度更快，并生成较小的打包文件。
        externals: TARGET_NODE
        ? nodeExternals({
            // 不要外置化webpack需要处理的依赖模块。
            // 可以在这里添加更多的文件类型。例如，未处理 *.vue 原始文件，
            // 还应该将修改`global`（例如polyfill）的依赖模块列入白名单
            whitelist: [/\.css$/]
        })
        : 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

JavaScript | 复制代码

```
npm i cross-env -D
```

定义创建脚本 `package.json`

JavaScript | 复制代码

```
"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`

```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`

```JavaScript
npm install -S vuex
```

创建 `vuex` 工厂函数

```JavaScript
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`

JavaScript 复制代码

```
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

JavaScript 复制代码

```
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

JavaScript | 复制代码

```javascript
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`

JavaScript | 复制代码

```
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`

JavaScript | 复制代码

```
// 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`

JavaScript | 复制代码

```
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

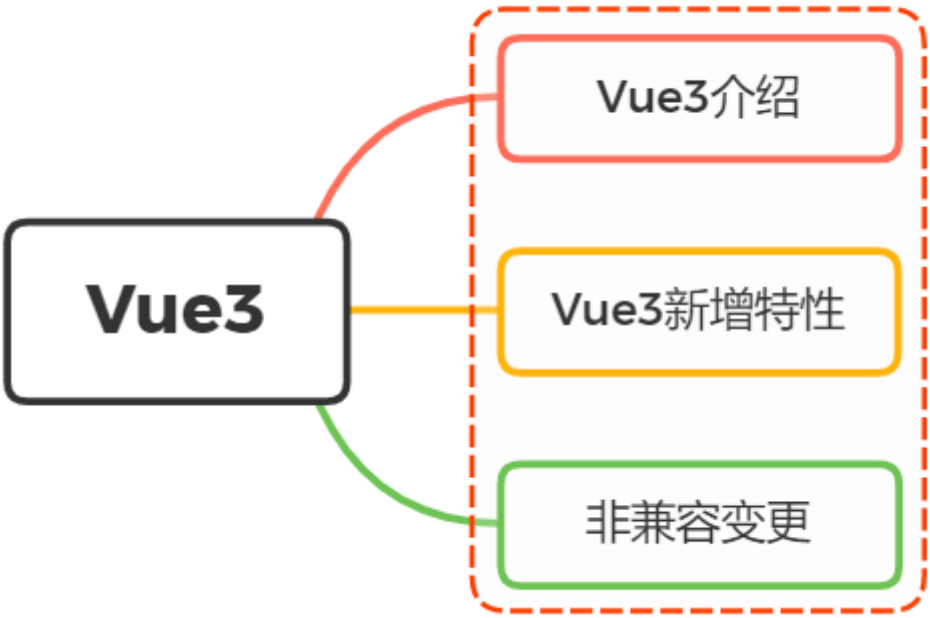
JavaScript 复制代码

```
// 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页的选项，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)
})
```

28.4. 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

29. vue3有了解过吗？能说说跟vue2的区别吗？



29.1. Vue3介绍

关于 `vue3` 的重构背景，尤大是这样说的：

「Vue 新版本的理念成型于 2018 年末，当时 Vue 2 的代码库已经有两岁半了。比起通用软件的生命周期来这好像也没那么久，但在这段时期，前端世界已经今昔非比了

在我们更新（和重写）Vue 的主要版本时，主要考虑两点因素：首先是新的 JavaScript 语言特性在主流浏览器中的受支持水平；其次是当前代码库中随时间推移而逐渐暴露出来的一些设计和架构问题」

简要就是：

- 利用新的语言特性(es6)
- 解决架构问题

29.2. 哪些变化

从上图中，我们可以概览 `Vue3` 的新特性，如下：

- 速度更快
- 体积减少
- 更易维护
- 更接近原生
- 更易使用

29.2.1. 速度更快

`vue3` 相比 `vue2`

- 重写了虚拟 `Dom` 实现
- 编译模板的优化
- 更高效的组件初始化
- `undate` 性能提高1.3~2倍
- `SSR` 速度提高了2~3倍

29.2.2. 体积更小

通过 `webpack` 的 `tree-shaking` 功能，可以将无用模块“剪辑”，仅打包需要的

能够 `tree-shaking` ，有两大好处：

- 对开发人员，能够对 `vue` 实现更多其他的功能，而不必担忧整体体积过大
- 对使用者，打包出来的包体积变小了

`vue` 可以开发出更多其他的功能，而不必担忧 `vue` 打包出来的整体体积过多

29.2.3. 更易维护