服务端渲染SSR

资源

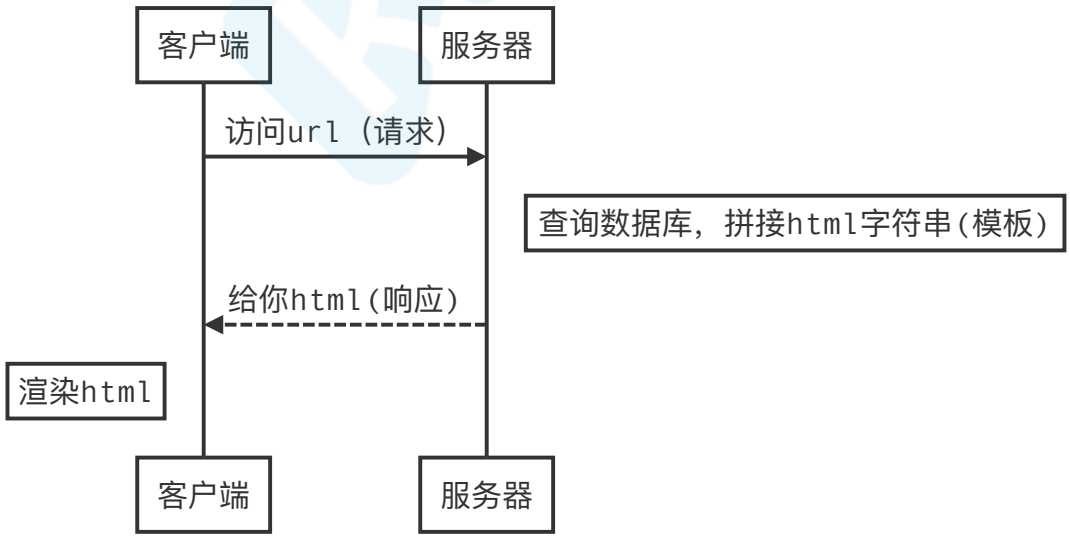
1. vue ssr
2. nuxt.js

知识点

理解ssr

传统web开发

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。



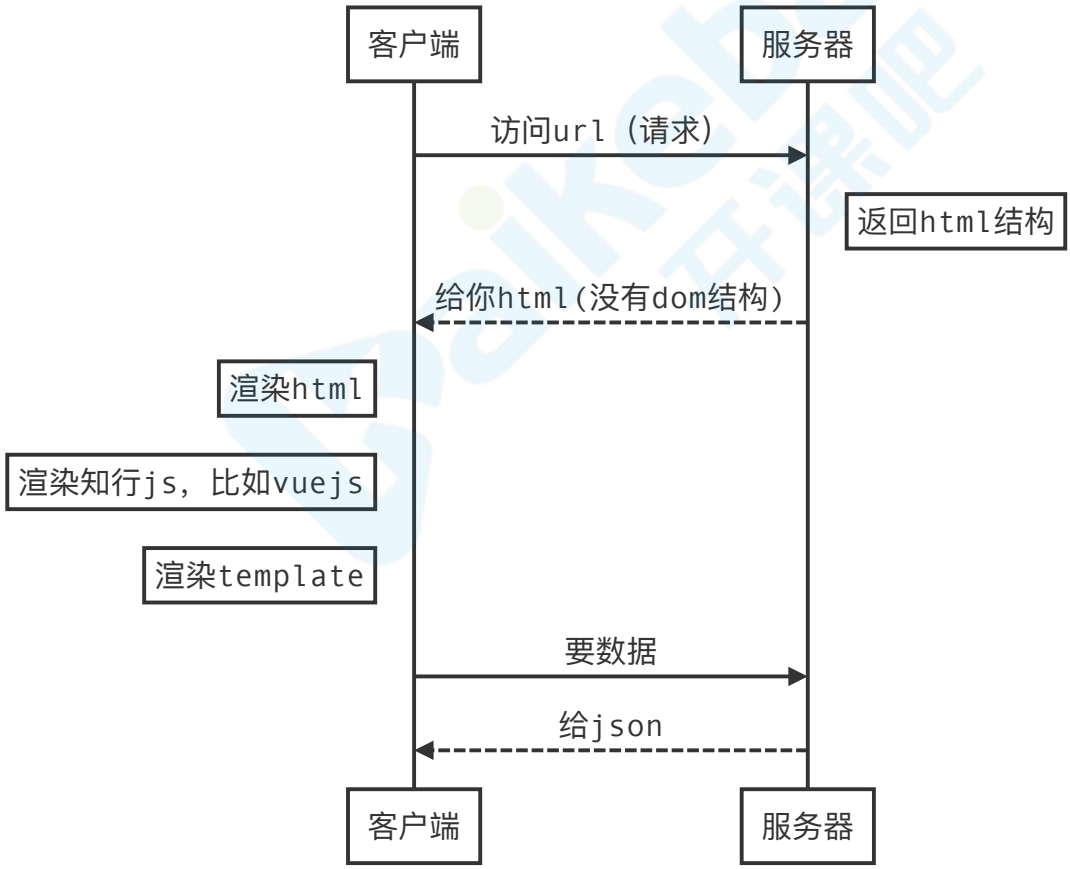
测试代码，server\01-express.js

打开页面查看源码，浏览器拿到的是全部的dom结构

```
        <html>
            <div>
                <div id="app">
                    <h1>开课吧</h1>
                    <p class="demo">开课吧还不错</p>
                </div>
            </body>
        </html>
```

单页应用 Single Page App

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。



测试：`npm run serve`

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。

```
<!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
    <link rel="icon" href="/favicon.ico">
    <title>study-vue</title>
  <link href="/js/app.js" rel="preload" as="script"><link href="/js/cl
  <body>
    <noscript>
      <strong>We're sorry but study-vue doesn't work properly without
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  <script type="text/javascript" src="/js/chunk-vendors.js"></script>
</html>
```

单页应用缺点：

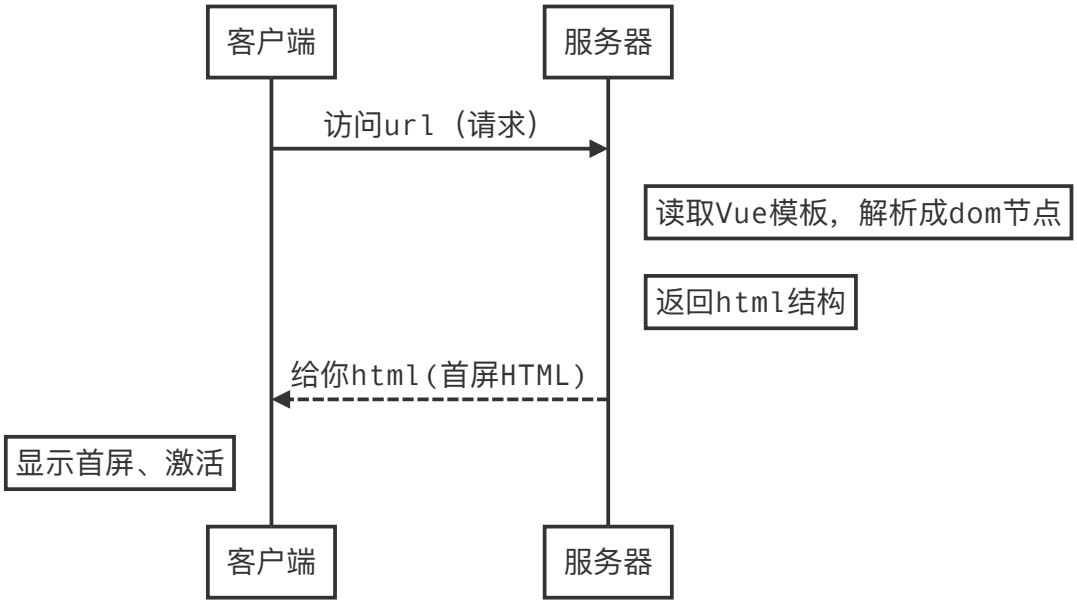
1. seo
2. 首屏加载速度

优点：

1. 渲染计算放到客户端
2. 省流量

服务端渲染 Server Side Render

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，这种页面渲染方式被称为服务端渲染 (server side render)



Vue SSR实战

新建工程

vue-cli创建工程即可

```
vue create ssr
```

> 演示项目使用vue-cli 4.x创建

安装依赖

```
npm install vue-server-renderer@2.6.10 -S
```

> 要确保vue、vue-server-renderer版本一致

启动脚本

创建一个express服务器，将vue ssr集成进来，./server/02-simple-ssr.js

```
const express = require("express");
const app = express();

// 导入Vue构造函数
const Vue = require("vue");
```

```
// createRenderer用于获取渲染器
const { createRenderer } = require("vue-server-renderer");

// 获取渲染器
const renderer = createRenderer();

app.get("/", async (req, res) => {
  // 创建一个待渲染vue实例
  const vm = new Vue({
    data: { name: "村长真棒" },
    template: `
      <div >
          <h1>{{name}}</h1>
      </div>
    `
  });

  try {
    // renderToString将vue实例渲染为html字符串，它返回一个Promise
    const html = await renderer.renderToString(vm);
    // 返回html给客户端
    res.send(html);
  } catch (error) {
    // 渲染出错返回500错误
    res.status(500).send("Internal Server Error");
  }
});

app.listen(3000);
```

路由

路由支持仍然使用vue-router

安装

若未引入vue-router则需要安装

```
npm i vue-router -s
```

创建路由实例

每次请求的url委托给vue-router处理，02-simple-ssr.js

```
// 引入vue-router
```

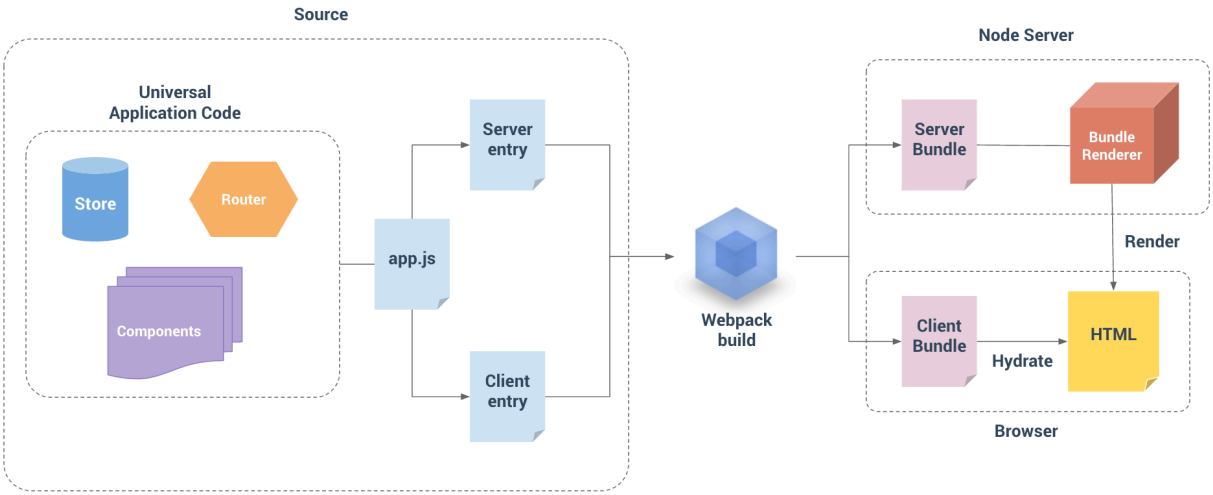
```
const Router = require('vue-router')
Vue.use(Router)

// path修改为通配符
app.get('*', async function (req, res) {
  // 每次创建一个路由实例
  const router = new Router({
    mode: 'history',
    routes: [
      { path: "/", component: {template:'<div>index page</div>'} },
      { path: "/detail", component: {template:'<div>detail page</div>'} }
    ]
  });

  const vm = new Vue({
    data: { msg: '村长真棒' },
    // 添加router-view显示内容
    template: `
      <div>
        <router-link to="/">index</router-link>
        <router-link to="/detail">detail</router-link>
        <router-view></router-view>
      </div>`,
    router, // 挂载
  })

  try {
    // 跳转至对应路由
    router.push(req.url);
    const html = await renderer.renderToString(vm)
    res.send(html)
  } catch (error) {
    res.status(500).send('渲染出错')
  }
})
```

同构开发SSR应用

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**。

构建流程

目标是生成一个「服务器 bundle」用于服务端首屏渲染，和一个「客户端bundle」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

创建@/router/index.js

```
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: [
      { path: "/", component: {template:'<div>index page</div>'} },
      { path: "/detail", component: {template:'<div>detail page</div>'} }
    ]
  });
}
```

主文件

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。创建main.js：

```
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 };
}
```

服务端入口

上面的bundle就是webpack打包的服务端bundle，我们需要编写服务端入口文件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);
  });
};
```

客户端入口

客户端入口只需创建vue实例并执行挂载，这一步称为激活。创建entry-client.js：

```
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");
});
```

webpack配置

安装依赖

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

具体配置，vue.config.js

```
// 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
        });
      });
  }
};
```

脚本配置

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

宿主文件

最后需要定义宿主文件，修改./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>
```

服务器启动文件

修改服务器启动文件，现在需要处理所有路由，./server/04-ssr.js

```
// 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"), "utf-8"), // 宿主文件
  clientManifest: require(resolve("../dist/client/vue-ssr-client-manifest.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)
})
```

整合Vuex

安装vuex

```
npm install -S vuex
```

store/index.js

```
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;
        }
    }
  })
}
```

挂载store，main.js

```
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 }
}
```

使用，.src/components/Index.vue

```
<h2 @click="$store.commit('add')">{{$store.state.count}}</h2>
```

> 注意事项：注意打包和重启服务

数据预取

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**。

异步数据获取，store/index.js

```
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，Index.vue

```
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,
      });
    }
  },
});
```