预渲染：如何借助 Vite 搭建高可用的服务端渲染(SSR)工程?

发布于 2022-05-09

在非常早期的 Web 技术中，大家还是使用 JSP 这种古老的模板语法来编写前端的页面，然后直接将 JSP 文件放到服务端，在服务端填入数据并渲染出完整的页面内容，可以说那个时代的做法是天然的服务端渲染。但随着 AJAX 技术的成熟以及各种前端框架(如 Vue、React)的兴起，前后端分离的开发模式逐渐成为常态，前端只负责页面 UI 及逻辑的开发，而服务端只负责提供数据接口，这种开发方式下的页面渲染也叫 客户端渲染 (Client Side Render，简称 CSR)。

但客户端渲染也存在着一定的问题，例如首屏加载比较慢、对 SEO 不太友好，因此 SSR (Server Side Render)即服务端渲染技术应运而生，它在保留 CSR 技术栈的同时，也能解决 CSR 的各种问题。

这一节，我们一起来聊聊SSR，说说它的基本原理，以及如何借助 Vite 的各项构建能力实现 SSR，让你体会到 SSR 的各个生命周期是如何串联起来的。

> 需要说明的是，本节最终的目的并不是让大家学习某个现有的 SSR 解决方案(如 Nuxt)，或者简单了解一下实现原理，而是教大家如何从底层开始手动搭建一个可以深度定制的 SSR 项目，并且解决各种 SSR 中的工程化问题。

SSR 基本概念

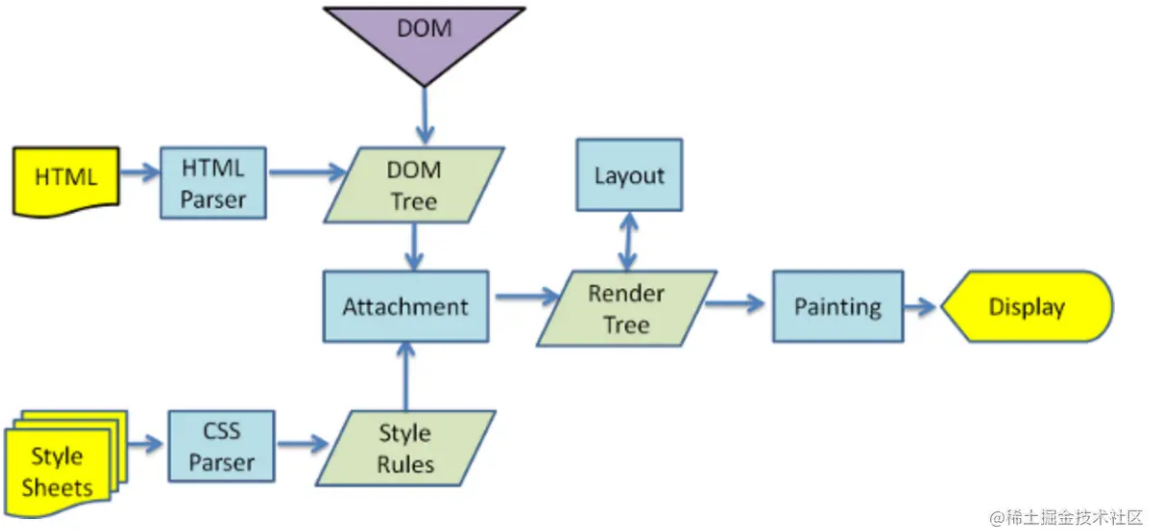
首先我们来分析一下 CSR 的问题，它的 HTML 产物一般是如下的结构:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <link rel="stylesheet" href="xxx.css" />
```

```
</head>

<body>
  <!-- 一开始没有页面内容 -->
  <div id="root"></div>
  <!-- 通过 JS 执行来渲染页面 -->
  <script src="xxx.chunk.js"></script>
</body>
</html>
```

顺便我们也回顾一下浏览器的渲染流程，如下图所示:



当浏览器拿到如上的 HTML 内容之后，其实并不能渲染完整的页面内容，因为此时的 body 中基本只有一个空的 div 节点，并没有填入真正的页面内容。而接下来浏览器开始下载并执行 JS 代码，经历了框架初始化、数据请求、DOM 插入等操作之后才能渲染出完整的页面。也就是说，在 CSR 中完整的页面内容本质上通过 JS 代码执行之后才能够渲染。这主要会导致两个方面的问题:

- **首屏加载速度比较慢。**首屏加载需要依赖 JS 的执行，下载和执行 JS 都可能是非常耗时的操作，尤其是在一些网络不佳的场景，或者性能敏感的低端机下。
- **对 SEO(搜索引擎优化) 不友好。**页面 HTML 没有具体的页面内容，导致搜索引擎爬虫无法获取关键词信息，导致网站排名受到影响。

那么 SSR 是如何解决这些问题的呢?

在 SSR 的场景下，服务端生成好**完整的 HTML 内容**，直接返回给浏览器，浏览器能够根据 HTML 渲染出完整的首屏内容，而不需要依赖 JS 的加载，这样一方面能够降低首屏渲染的时间，另一方面也能将完整的页面内容展现给搜索引擎的爬虫，利于 SEO。

当然，SSR 中只能生成页面的内容和结构，并不能完成事件绑定，因此需要在浏览器中执行 CSR 的 JS 脚本，完成事件绑定，让页面拥有交互的能力，这个过程被称作

执行 CSR 的 JS 脚本，完成事件绑定，让页面拥有交互的能力，这个过程被称作

`hydrate` (翻译为 注水 或者 激活)。同时，像这样服务端渲染 + 客户端 hydrate 的应用也被称为 同构应用 。

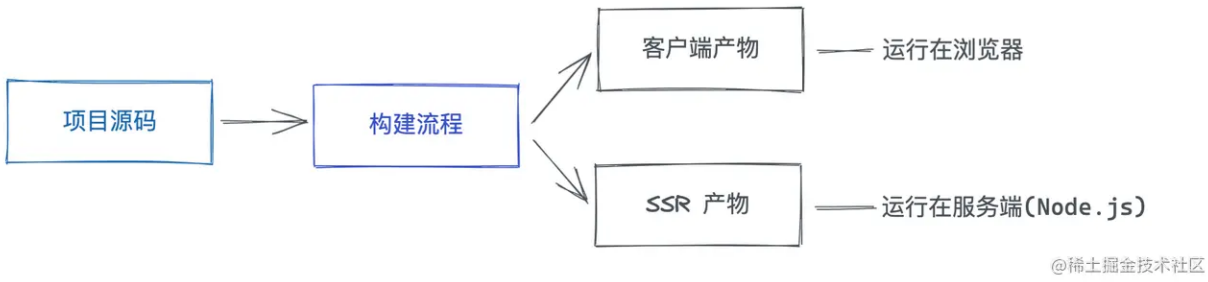
SSR 生命周期分析

前面我们提到了 SSR 会在服务端(这里主要指 Node.js 端)提前渲染出完整的 HTML 内容，那这是如何做到的呢?

首先需要保证前端的代码经过编译后放到服务端中能够正常执行，其次在服务端渲染前端组件，生成并组装应用的 HTML。这就涉及到 SSR 应用的两大生命周期: 构建时 和 运行时 ，我们不妨来仔细梳理一下。

我们先来看看 构建时 需要做哪些事情:

> **解决模块加载问题。**在原有的构建过程之外，需要加入 `SSR 构建` 的过程 ，具体来说，我们需要另外生成一份 `CommonJS` 格式的产物，使之能在 Node.js 正常加载。当然，随着 Node.js 本身对 ESM 的支持越来越成熟，我们也可以复用前端 ESM 格式的代码，Vite 在开发阶段进行 SSR 构建也是这样的思路。



> **移除样式代码的引入。**直接引入一行 css 在服务端其实是无法执行的，因为 Node.js 并不能解析 CSS 的内容。但 `CSS Modules` 的情况除外，如下所示:

```
import styles from './index.module.css'

// 这里的 styles 是一个对象，如{ "container": "xxx" }，而不是 CSS 代码
console.log(styles)
```

> **依赖外部化(external)。**对于某些第三方依赖我们并不需要使用构建后的版本，而是直接从 `node_modules` 中读取，比如 `react-dom` ，这样在 `SSR 构建` 的过程中将不会构建这些依赖，从而极大程度上加速 SSR 的构建。

对于 SSR 的运行时，一般可以拆分为比较固定的生命周期阶段，简单来讲可以整理为以下几个核心的阶段:

- - **加载 SSR 入口模块**。在这个阶段，我们需要确定 SSR 构建产物的入口，即组件的入口在哪里，并加载对应的模块。
- - **进行数据预取**。这时候 Node 侧会通过查询数据库或者网络请求来获取应用所需的数据。
- - **渲染组件**。这个阶段为 SSR 的核心，主要将第 `1` 步中加载的组件渲染成 HTML 字符串或者 Stream 流。
- - **HTML 拼接**。在组件渲染完成之后，我们需要拼接完整的 HTML 字符串，并将其作为响应返回给浏览器。

从上面的分析中你可以发现，SSR 其实是 构建 和 运行时 互相配合才能实现的，也就是说，仅靠构建工具是不够的，写一个 Vite 插件严格意义上无法实现 SSR 的能力，我们需要对 Vite 的构建流程做一些整体的调整，并且加入一些服务端运行时的逻辑才能实现。那么接下来，我们就以具体的代码示例来讲解如何在 Vite 中完成 SSR 工程的搭建。

基于 Vite 搭建 SSR 项目

1. SSR 构建 API

首先 Vite 作为一个构建工具，是如何支持 SSR 构建的呢？换句话说，它是如何让前端的代码也能顺利在 Node.js 中成功跑起来的呢？

可以分为两种情况，在开发环境下，Vite 依然秉承 ESM 模块按需加载即 `no-bundle` 的理念，对外提供了 `ssrLoadModule` API，你可以无需打包项目，将入口文件的路径传入 `ssrLoadModule` 即可：

```
// 加载服务端入口模块
const xxx = await vite.ssrLoadModule('/src/entry-server.tsx')
```

而在生产环境下，Vite 会默认进行打包，对于 SSR 构建输出 CommonJS 格式的产物。我们可以在 `package.json` 中加入这样类似的构建指令：

```
{
  "build:ssr": "vite build --ssr 服务端入口路径"
}
```

这样 Vite 会专门为 SSR 打包出一份构建产物。因此你可以看到，大部分 SSR 构建时的事情，Vite 已经帮我们提供了开箱即用的方案，我们后续直接使用即可。

2. 项目骨架搭建

接下来我们正式开始 SSR 项目的搭建，你可以通过脚手架初始化一个 `react+ts` 的项目:

```
npm init vite
pnpm i
```

删除项目自带的 `src/main.ts` ，然后在 src 目录下新建 `entry-client.tsx` 和 `entry-server.tsx` 两个入口文件:

```
// entry-client.ts
// 客户端入口文件
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

ReactDOM.hydrate(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)

// entry-server.ts
// 导出 SSR 组件入口
import App from "./App";
import './index.css'

function ServerEntry(props: any) {
  return (
    <App/>
  );
}

export { ServerEntry };
```

我们以 Express 框架为例来实现 Node 后端服务，后续的 SSR 逻辑会接入到这个服务中。当然你需要安装以下的依赖:

```
pnpm i express -S
pnpm i @types/express -D
```

接着新建 src/ssr-server/index.ts：

```
// src/ssr-server/index.ts
// 后端服务
import express from 'express';

async function createServer() {
  const app = express();

  app.listen(3000, () => {
    console.log('Node 服务器已启动~')
    console.log('http://localhost:3000');
  });
}

createServer();
```

然后你可以在 package.json 中添加如下的 scripts：

```
{
  "scripts": {
    // 开发阶段启动 SSR 的后端服务
    "dev": "nodemon --watch src/ssr-server --exec 'esno src/ssr-server/index.ts'",
    // 打包客户端产物和 SSR 产物
    "build": "npm run build:client && npm run build:server",
    "build:client": "vite build --outDir dist/client",
    "build:server": "vite build --ssr src/entry-server.tsx --outDir dist/server",
    // 生产环境预览 SSR 效果
    "preview": "NODE_ENV=production esno src/ssr-server/index.ts"
  },
}
```

其中设计到两个额外的工具，给大家解释一下:

- nodemon：一个监听文件变化自动重启 Node 服务的工具。
- esno：类似 ts-node 的工具，用来执行 ts 文件，底层基于 Esbuild 实现。

同时你也需要安装这两个依赖:

```
pnpm i esno nodemon -D
```

OK，现在我们就基本上搭建好了基本的项目骨架，接下里我们专注于 SSR 运行时的实现逻辑即可。

3. SSR 运行时实现

3. SSR 逻辑的实现

SSR 作为一种特殊的后端服务，我们可以将其封装成一个中间件的形式，如以下的代码所示:

```
import express, { RequestHandler, Express } from 'express';
import { ViteDevServer } from 'vite';

const isProd = process.env.NODE_ENV === 'production';
const cwd = process.cwd();

async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  let vite: ViteDevServer | null = null;
  if (!isProd) {
    vite = await (await import('vite')).createServer({
      root: process.cwd(),
      server: {
        middlewareMode: 'ssr',
      }
    })
    // 注册 Vite Middlewares
    // 主要用来处理客户端资源
    app.use(vite.middlewares);
  }
  return async (req, res, next) => {
    // SSR 的逻辑
    // 1. 加载服务端入口模块
    // 2. 数据预取
    // 3. 「核心」渲染组件
    // 4. 拼接 HTML，返回响应
  };
}

async function createServer() {
  const app = express();
  // 加入 Vite SSR 中间件
  app.use(await createSsrMiddleware(app));

  app.listen(3000, () => {
    console.log('Node 服务器已启动~')
    console.log('http://localhost:3000');
  });
}

createServer();
```

接下来我们把焦点放在中间件内 SSR 的逻辑实现上，首先实现第一步即 加载服务端入口模块：

```
async function loadSsrEntryModule(vite: ViteDevServer | null) {
  // 生产模式下直接 require 打包后的产物
  if (isProd) {
    const entryPath = path.join(cwd, 'dist/server/entry-server.js');
```

```
    return require(entryPath);
  }
  // 开发环境下通过 no-bundle 方式加载
  else {
    const entryPath = path.join(cwd, 'src/entry-server.tsx');
    return vite!.ssrLoadModule(entryPath);
  }
}
```

中间件内的逻辑如下:

```
async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  // 省略前面的代码
  return async (req, res, next) => {
    const url = req.originalUrl;
    // 1. 服务端入口加载
    const { ServerEntry } = await loadSsrEntryModule(vite);
    // ...
  }
}
```

接下来我们来实现服务端的数据预取操作，你可以在 `entry-server.tsx` 中添加一个简单的获取数据的函数:

```
export async function fetchData() {
  return { user: 'xxx' }
}
```

然后在 SSR 中间件中完成数据预取的操作:

```
// src/ssr-server/index.ts
async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  // 省略前面的代码
  return async (req, res, next) => {
    const url = req.originalUrl;
    // 1. 服务端入口加载
    const { ServerEntry, fetchData } = await loadSsrEntryModule(vite);
    // 2. 预取数据
    const data = await fetchData();
  }
}
```

接着我们进入到核心的组件渲染阶段:

```
// src/ssr-server/index.ts
import { renderToString } from 'react-dom/server';
import React from 'react';
```

```
async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  // 省略前面的代码
  return async (req, res, next) => {
    const url = req.originalUrl;
    // 1. 服务端入口加载
    const { ServerEntry, fetchData } = await loadSsrEntryModule(vite);
    // 2. 预取数据
    const data = await fetchData();
    // 3. 组件渲染 -> 字符串
    const appHtml = renderToString(React.createElement(ServerEntry, { data }));
  }
}
```

由于在第一步之后我们拿到了入口组件，现在可以调用前端框架的 `renderToString` API 将组件渲染为字符串，组件的具体内容便就此生成了。

OK，目前我们已经拿到了组件的 HTML 以及预取的数据，接下来我们在根目录下的 HTML 中提供相应的插槽，方便内容的替换:

```
// index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/src/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="root"><!-- SSR_APP --></div>
    <script type="module" src="/src/entry-client.tsx"></script>
    <!-- SSR_DATA -->
  </body>
</html>
```

紧接着我们在 SSR 中间件中补充 HTML 拼接的逻辑:

```
// src/ssr-server/index.ts
function resolveTemplatePath() {
  return isProd ?
    path.join(cwd, 'dist/client/index.html') :
    path.join(cwd, 'index.html');
}

async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  // 省略之前的代码
  return async (req, res, next) => {
    const url = req.originalUrl;
    // 省略前面的步骤
    // 4. 拼接完整 HTML 字符串，返回客户端
    const templatePath = resolveTemplatePath();
```

```
    let template = await fs.readFileSync(templatePath, 'utf-8');
    // 开发模式下需要注入 HMR、环境变量相关的代码，因此需要调用 vite.transformIndexHtml
    if (!isProd && vite) {
      template = await vite.transformIndexHtml(url, template);
    }
    const html = template
      .replace('<!-- SSR_APP -->', appHtml)
      // 注入数据标签，用于客户端 hydrate
      .replace(
        '<!-- SSR_DATA -->',
        `<script>window.__SSR_DATA__=${JSON.stringify(data)}</script>`
      );
    res.status(200).setHeader('Content-Type', 'text/html').end(html);
  }
}
```

在拼接 HTML 的逻辑中，除了添加页面的具体内容，同时我们也注入了一个挂载全局数据的 `script` 标签，这是用来干什么的呢?

在 SSR 的基本概念中我们就提到过，为了激活页面的交互功能，我们需要执行 CSR 的 JavaScript 代码来进行 hydrate 操作，而客户端 hydrate 的时候需要和服务端 同步预取后的数据 ，保证页面渲染的结果和服务端渲染一致，因此，我们刚刚注入的数据 script 标签便派上用场了。由于全局的 window 上挂载服务端预取的数据，我们可以在 `entry-client.tsx` 也就是客户端渲染入口中拿到这份数据，并进行 hydrate:

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

// @ts-ignore
const data = window.__SSR_DATA__;

ReactDOM.hydrate(
  <React.StrictMode>
    <App data={data}/>
  </React.StrictMode>,
  document.getElementById('root')
)
```

现在，我们基本开发完了 SSR 核心的逻辑，执行 `npm run dev` 启动项目:

```
→  ssr git:(main) ✗ npm run dev

> vite-ssr-app@0.0.0 dev
> nodemon --watch src/ssr-server --exec 'esno src/ssr-server/index.ts'

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): src/ssr-server/**/*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `esno src/ssr-server/index.ts`
Node 服务器已启动~
http://localhost:3000
```


打开浏览器后查看页面源码，你可以发现 SSR 生成的 HTML 已经顺利返回了:

4. 生产环境的 CSR 资源处理

如果你现在执行 `npm run build` 及 `npm run preview` 进行生产环境的预览，会发现 SSR 可以正常返回内容，但所有的静态资源及 CSR 的代码都失效了:

不过开发阶段并没有这个问题，这是因为对于开发阶段的静态资源 Vite Dev Server 的中间件已经帮我们处理了，而生产环境所有的资源都已经打包完成，我们需要启用单独的静态资源服务来承载这些资源。这里你可以 `serve-static` 中间件来完成这个服务，首先安装对应第三方包:

```
pnpm i serve-static -S
```

接着我们到 server 端注册:

```
// 过滤出页面请求
function matchPageUrl(url: string) {
  if (url === '/') {
    return true;
  }
  return false;
}

async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  return async (req, res, next) => {
    try {
      const url = req.originalUrl;
      if (!matchPageUrl(url)) {
        // 走静态资源的处理
        return await next();
      }
      // SSR 的逻辑省略
    } catch(e: any) {
      vite?.ssrFixStacktrace(e);
      console.error(e);
      res.status(500).end(e.message);
    }
  }
}

async function createServer() {
  const app = express();
  // 加入 Vite SSR 中间件
  app.use(await createSsrMiddleware(app));

  // 注册中间件，生产环境端处理客户端资源
  if (isProd) {
    app.use(serve(path.join(cwd, 'dist/client')))
  }
  // 省略其它代码
}
```

这样一来，我们就解决了生产环境下静态资源失效的问题。不过，一般情况下，我们会将静态资源部上传到 CDN 上，并且将 Vite 的 `base` 配置为域名前缀，这样我们可以通过 CDN 直接访问到静态资源，而不需要加上服务端的处理。不过作为本地的生产环境预览而言， `serve-static` 还是一个不错的静态资源处理手段。

工程化问题

以上我们基本实现了 SSR 核心的 构建 和 运行时 功能，可以初步运行一个基于 Vite 的 SSR 项目，但在实际的场景中仍然是有不少的工程化问题需要我们注意。下面我就和你

一起梳理一下到底需要考虑哪些问题，以及相应的解决思路是如何的，同时我也会推荐一些比较成熟的解决方案。

1. 路由管理

在 SPA 场景下，对于不同的前端框架，一般会有不同的路由管理方案，如 Vue 中的 `vue-router` 、React 的 `react-router` 。不过归根结底，路由方案在 SSR 过程中所完成的功能都是差不多的:

> 告诉框架现在渲染哪个路由。在 Vue 中我们可以通过 `router.push` 确定即将渲染的路由，React 中则通过 `StaticRouter` 配合 `location` 参数来完成。

> 设置 `base` 前缀。规定路径的前缀，如 `vue-router` 中 base 参数、 `react-router` 中 `StaticRouter` 组件的 basename。

2. 全局状态管理

对于全局的状态管理而言，对于不同的框架也有不同的生态和方案，比如 Vue 中的 Vuex、Pinia，React 中的 Redux、Recoil。各个状态管理工具的用法并不是本文的重点，接入 SSR 的思路也比较简单，在 `预取数据` 阶段初始化服务端的 `store` ，将异步获取的数据存入 `store` 中，然后在 `拼接 HTML` 阶段将数据从 store 中取出放到数据 script 标签中，最后在客户端 hydrate 的时候通过 window 即可访问到预取数据。

> 需要注意的服务端处理许多不同的请求，对于每个请求都需要**分别**初始化 store，即一个请求一个 store，不然会造成全局状态污染的问题。

3. CSR 降级

在某些比较极端的情况下，我们需要降级到 CSR，也就是客户端渲染。一般而言包括如下的降级场景:

- • 服务器端**预取数据**失败，需要降级到客户端获取数据。
- • 服务器出现异常，需要返回**兜底的 CSR 模板**，完全降级为 CSR。

- - 本地**开发调试**，有时需要跳过 SSR，仅进行 CSR。

对于第一种情况，在客户端入口文件中需要有重新获取数据的逻辑，我们可以进行这样的补充:

```
// entry-client.tsx
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

async function fetchData() {
  // 客户端获取数据
}


async fucntion hydrate() {
  let data;
  if (window.__SSR_DATA__) {
    data = window.__SSR_DATA__;
  } else {
    // 降级逻辑
    data = await fetchData();
  }
  // 也可简化为 const data = window.__SSR_DATA__ ?? await fetchData();
  ReactDOM.hydrate(
    <React.StrictMode>
      <App data={data}/>
    </React.StrictMode>,
    document.getElementById('root')
  )
}
```

对于第二种场景，即 服务器执行出错 ，我们可以在之前的 SSR 中间件逻辑追加 catch 逻辑:

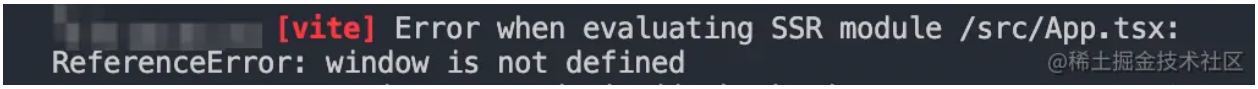
```
async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  return async (req, res, next) => {
    try {
      // SSR 的逻辑省略
    } catch(e: any) {
      vite?.ssrFixStacktrace(e);
      console.error(e);
      // 在这里返回浏览器 CSR 模板内容
    }
  }
}
```

对于第三种情况，我们可以通过通过 ?csr 的 url query 参数来强制跳过 SSR，在 SSR 中间件添加如下逻辑:

```
async function createSsrMiddleware(app: Express): Promise<RequestHandler> {
  return async (req, res, next) => {
    try {
      if (req.query?.csr) {
        // 响应 CSR 模板内容
        return;
      }
      // SSR 的逻辑省略
    } catch(e: any) {
      vite?.ssrFixStacktrace(e);
      console.error(e);
    }
  }
}
```

4. 浏览器 API 兼容

由于 Node.js 中不能使用浏览器里面诸如 `window` 、 `document` 之类的 API，因此一旦在服务端执行到这样的 API 会报如下的错误：



那么如何来解决这个问题呢？

首先我们可以通过 `import.meta.env.SSR` 这个 Vite 内置的环境变量来判断是否处于 SSR 环境，以此来规避业务代码在服务端出现浏览器的 API：

```
if (import.meta.env.SSR) {
  // 服务端执行的逻辑
} else {
  // 在此可以访问浏览器的 API
}
```

当然，我们也可以通过 polyfill 的方式，在 Node 中注入浏览器的 API，使这些 API 能够正常运行起来，解决如上的问题。我推荐使用一个比较成熟的 polyfill 库 `jsdom` ，使用方式如下：

```
const jsdom = require('jsdom');
const { window } = new JSDOM(`<!DOCTYPE html><p>Hello world</p>`);
const { document } = window;
// 挂载到 node 全局
global.window = window;
global.document = document;
```

5. 自定义 Head

在 SSR 的过程中，我们虽然可以在决定组件的内容，即 `<div id="root"></div>` 这个容器 div 中的内容，但对于 HTML 中 `head` 的内容我们无法根据**组件的内部状态**来决定，比如对于一个直播间的页面，我们需要在服务端渲染出 title 标签，title 的内容是不同主播的直播间名称，不能在代码中写死，这种情况怎么办？

React 生态中的 react-helmet 以及 Vue 生态中的 vue-meta 库就是为了解决这样的问题，让我们可以直接在组件中写一些 Head 标签，然后在服务端能够拿到组件内部的状态。这里我以一个 `react-helmet` 例子来说明:

```
// 前端组件逻辑
import { Helmet } from "react-helmet";

function App(props) {
  const { data } = props;
  return {
    <div>
       <Helmet>
        <title>{ data.user }的页面</title>
        <link rel="canonical" href="http://mysite.com/example" />
      </Helmet>
    </div>
  }
}
// 服务端逻辑
import Helmet from 'react-helmet';

// renderToString 执行之后
const helmet = Helmet.renderStatic();
console.log("title 内容: ", helmet.title.toString());
console.log("link 内容: ", helmet.link.toString())
```

启动服务后访问页面，可以发现终端能打印出如下的信息:

```
title 内容:  <title data-react-helmet="true">xxx的页面</title>
link 内容:  <link data-react-helmet="true" rel="canonical" href="http://mysite.com/example"/>
```


如此一来，我们就能根据组件的状态确定 Head 内容，然后在 拼接 HTML 阶段将这些内容插入到模板中。

6. 流式渲染

在不同前端框架的底层都实现了流式渲染的能力，即边渲染边响应，而不是等整个组件树

渲染完毕之后再响应，这么做可以让响应提前到达浏览器，提升首屏的加载性能。Vue 中的 renderToNodeStream 和 React 中的 renderToNodeStream 都实现了流式渲染的能力, 大致的使用方式如下:

```
import { renderToNodeStream } from 'react-dom/server';

// 返回一个 Nodejs 的 Stream 对象
const stream = renderToNodeStream(element);
let html = ''

stream.on('data', data => {
  html += data.toString()
  // 发送响应
})

stream.on('end', () => {
  console.log(html) // 渲染完成
  // 发送响应
})

stream.on('error', err => {
  // 错误处理
})
```

不过，流式渲染在我们带来首屏性能提升的同时，也给我们带来了一些限制: **如果我们需要在 HTML 中填入一些与组件状态相关的内容，则不能使用流式渲染。**比如 `react-helmet` 中自定义的 head 内容，即便在渲染组件的时候收集到了 head 信息，但在流式渲染中，此时 HTML 的 head 部分已经发送给浏览器了，而这部分响应内容已经无法更改，因此 `react-helmet` 在 SSR 过程中将会失效。

7. SSR 缓存

SSR 是一种典型的 CPU 密集型操作，为了尽可能降低线上机器的负载，设置缓存是一个非常重要的环节。在 SSR 运行时，缓存的内容可以分为这么几个部分:

- 文件读取缓存 。尽可能避免多次重复读磁盘的操作，每次磁盘 IO 尽可能地复用缓存结果。如下代码所示:

```
function createMemoryFsRead() {
  const fileContentMap = new Map();
  return async (filePath) => {
    const cacheResult = fileContentMap.get(filePath);
    if (cacheResult) {
      return cacheResult;
    }
    const fileContent = await fs.readFile(filePath);
    fileContentMap.set(filePath, fileContent);
```

```
    fileContentMap.set(filePath, fileContent);
    return fileContent;
  }
}

const memoryFsRead = createMemoryFsRead();
memoryFsRead('file1');
// 直接复用缓存
memoryFsRead('file1');
```

- 预取数据缓存。对于某些实时性不高的接口数据，我们可以采取缓存的策略，在下次相同的请求进来时复用之前预取数据的结果，这样预取数据过程的各种 IO 消耗，也可以一定程度上减少首屏时间。

- HTML 渲染缓存。拼接完成的 HTML 内容是缓存的重点，如果能将这部分进行缓存，那么下次命中缓存之后，将可以节省 renderToString、HTML 拼接等一系列的消耗，服务端的性能收益会比较明显。

对于以上的缓存内容，具体的缓存位置可以是：

- - 服务器内存。如果是放到内存中，需要考虑缓存淘汰机制，防止内存过大导致服务宕机，一个典型的缓存淘汰方案是 lru-cache (基于 LRU 算法)。
- - Redis 数据库，相当于以传统后端服务器的设计思路来处理缓存。
- - CDN 服务。我们可以将页面内容缓存到 CDN 服务上，在下一次相同的请求进来时，使用 CDN 上的缓存内容，而不用消费源服务器的资源。对于 CDN 上的 SSR 缓存，大家可以通过阅读这篇文章深入了解。

> 需要补充的是，Vue 中另外实现了组件级别的缓存，这部分缓存一般放在内存中，可以实现更细粒度的 SSR 缓存。

8. 性能监控

在实际的 SSR 项目中，我们时常会遇到一些 SSR 线上性能问题，如果没有一个完整的性能监控机制，那么将很难发现和排查问题。对于 SSR 性能数据，有一些比较通用的指标：

- SSR 产物加载时间
- 数据预取的时间
- 组件渲染的时间
- 服务端接受请求到响应的完整时间

- SSR 缓存命中情况
- SSR 成功率、错误日志

我们可以通过 perf_hooks 来完成数据的采集，如下代码所示:

```
import { performance, PerformanceObserver } from 'perf_hooks';

// 初始化监听器逻辑
const perfObserver = new PerformanceObserver((items) => {
  items.getEntries().forEach(entry => {
    console.log('[performance]', entry.name, entry.duration.toFixed(2), 'ms');
  });
  performance.clearMarks();
});

perfObserver.observe({ entryTypes: ["measure"] })

// 接下来我们在 SSR 进行打点
// 以 renderToString  为例
performance.mark('render-start');
// renderToString 代码省略
performance.mark('render-end');
performance.measure('renderToString', 'render-start', 'render-end');
```

接着我们启动服务后访问，可以看到如下的打点日志信息:

```
[performance] renderToString 3.27 ms
[performance] renderToString 1.43 ms
[performance] renderToString 0.91 ms
```


同样的，我们可以将其它阶段的指标通过上述的方式收集起来，作为性能日志；另一方面，在生产环境下，我们一般需要结合具体的性能监控平台，对上述的各项指标进行打点上报，完成线上的 SSR 性能监控服务。

9. SSG/ISR/SPR

有时候对于一些静态站点(如博客、文档)，不涉及到动态变化的数据，因此我们并不需要用上服务端渲染。此时只需要在构建阶段产出完整的 HTML 进行部署即可，这种构建阶段生成 HTML 的做法也叫 SSG (Static Site Generation，静态站点生成)。

SSG 与 SSR 最大的区别就是产出 HTML 的时间点从 SSR 运行时 变成了 构建时 ，但核心的生命周期流程并没有发生变化:

这里给一段简单的实现代码:

```
// scripts/ssg.ts
// 以下的工具函数均可以从 SSR 流程复用
async function ssg() {
  // 1. 加载服务端入口
  const { ServerEntry, fetchData } = await loadSsrEntryModule(null);
  // 2. 数据预取
  const data = await fetchData();
  // 3. 组件渲染
  const appHtml = renderToString(React.createElement(ServerEntry, { data }));
  // 4. HTML 拼接
  const template = await resolveTemplatePath();
  const templateHtml = await fs.readFileSync(template, 'utf-8');
  const html = templateHtml
  .replace('<!-- SSR_APP -->', appHtml)
  .replace(
    '<!-- SSR_DATA -->',
    `<script>window.__SSR_DATA__=${JSON.stringify(data)}</script>`
  );
  // 最后，我们需要将 HTML 的内容写到磁盘中，将其作为构建产物
  fs.mkdirSync('./dist/client', { recursive: true });
  fs.writeFileSync('./dist/client/index.html', html);
}

ssg();
```

接着你可以在 `package.json` 中加入这样一段 npm scripts:

```
{
  "scripts": {
    "build:ssg": "npm run build && NODE_ENV=production esno scripts/ssg.ts"
  }
}
```

这样我们便初步实现了 SSG 的逻辑。当然，除了 SSG，业界还流传着一些其它的渲染模式，诸如 `SPR` 、 `ISR` ，听起来比较高大上，但实际上只是 SSR 和 SSG 所衍生出来的新功能罢了，这里简单给大家解释一下:

- `SPR` 即 `Serverless Pre Render` ，即把 SSR 的服务部署到 Serverless(FaaS) 环境中，实现服务器实例的自动扩缩容，降低服务器运维的成本。

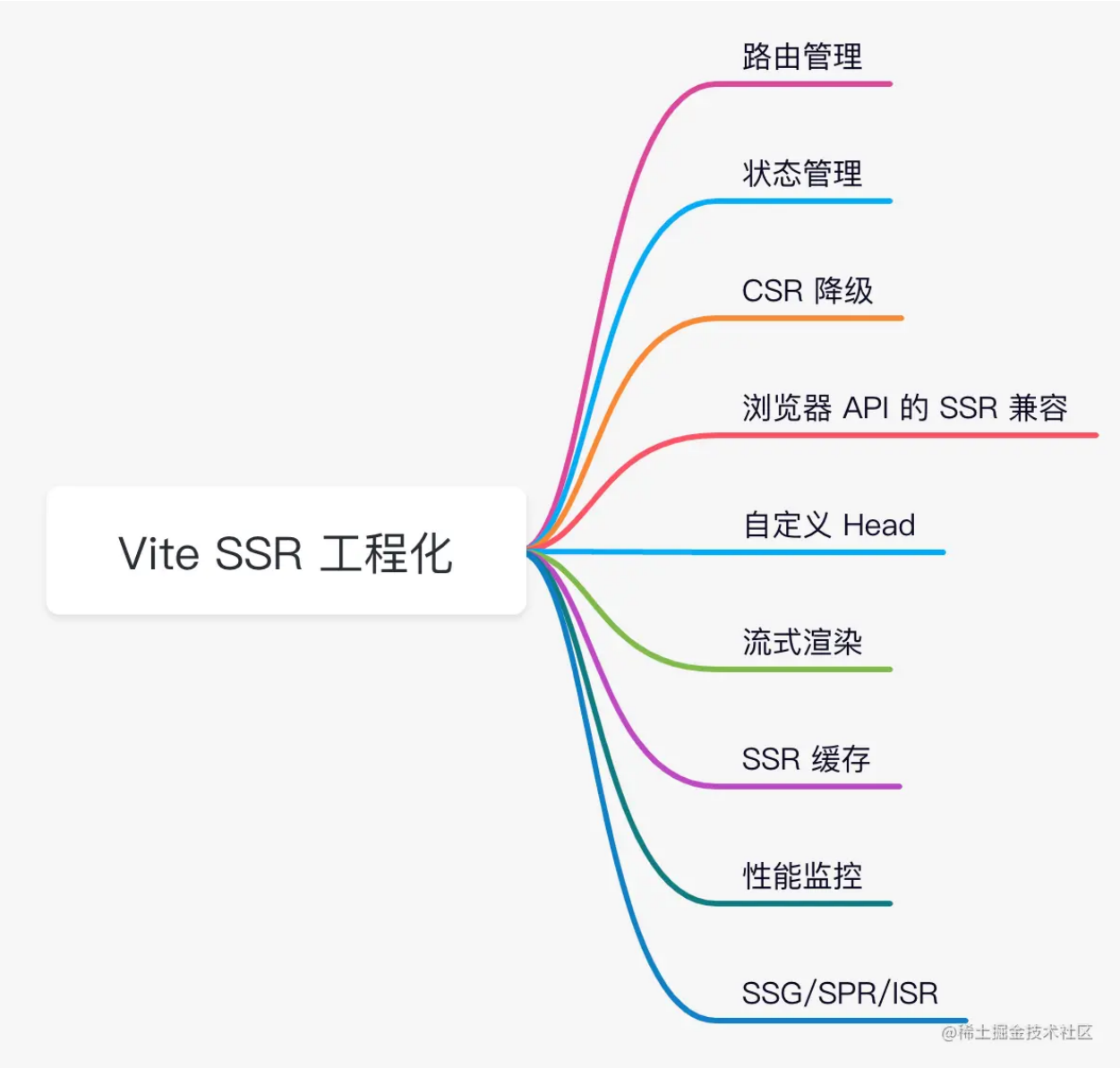
- `ISR` 即 `Incremental Site Rendering` ，即增量站点渲染，将一部分的 SSG 逻辑从构建时搬到了 `SSR` 运行时，解决的是大量页面 SSG 构建耗时长的问题。

小结

恭喜你，学习完了 `Vite SSR` 小节的内容。在本小节中，你需要重点掌握 **Vite 中 SSR 项目的搭建**和 **SSR 工程化问题及其对应的解决方案**。

首先，我给你介绍了 SSR 的基本概念，先分析传统的 CSR 到底存在哪些问题，然后介绍 SSR 是如何解决这些问题的。接着我给你分析了 SSR 应用的生命周期，主要包含 `构建时` 和 `运行时` 两个部分，带你从宏观上了解了要实现 SSR 究竟需要做哪些事情。当然，光介绍概念是远远不够的，在接下来的时间，我用具体的项目示例带你一步步搭建起了一个简单的 SSR 工程，借助 `Vite` 实现了 `客户端产物` 与 `SSR 产物` 的构建，然后通过 SSR 中间件的开发实现了 SSR 运行时的逻辑，包括 `加载服务端入口` 、 `数据预取` 、 `组件渲染` 和 `HTML 拼接` 这四个核心的步骤。

虽然说基于示例代码我们可以搭建 Vite 的 SSR 项目，但面对实际的开发场景，我们仍然需要考虑诸多的工程化问题，你可以参考下面这张图:



其中包括 路由管理 、 状态管理 、 CSR 降级 、 浏览器 API 的 SSR 兼容 、 自定义 Head 、 流式渲染 、 SSR 缓存 、 性能监控 以及更多的预渲染模式 SSG/SPR/ISR 。在后面的篇幅，我带你详细梳理了这些问题，分析各个问题的出现场景，并给出一些比较通用的解决思路和解决方案。相信通过这部分的学习，你也能驾驭更加复杂的 SSR 开发场景了。

OK，本文的内容到这里就结束了，希望能对你有所启发，也欢迎你把自己的学习心得和收获打在评论区，我们下一节再见👏。

上一篇：语法降级与Polyfill：联合前端编译工具链，消灭低版本浏览器兼容问题

下一篇：模块联邦: 如何实现优雅的跨应用代码共享?