Vue服务端渲染

创建项目

依赖安装

```
npm install vue vue-server-renderer express -D
```

服务端处理

```
// 导入express作为渲染服务器
const express = require("express");
// 导入Vue用于声明待渲染实例
const Vue = require("vue");
// 导入createRenderer用于获取渲染器
const {
    createRenderer
} = require("vue-server-renderer");
// 创建express实例
const app = express();
// 获取渲染器
const renderer = createRenderer();
// 待渲染vue实例
const vm = new Vue({
    data: {
        name: "开课吧"
    },
    template: `
<div >
<h1>{{name}}</h1>
</div>
`
});
app.get("/", async function(req, res) {
    // renderToString可以将vue实例转换为html字符串
    // 若未传递回调函数,则返回Promise
    try {
        const html = await renderer.renderToString(vm);
        res.send(html);
    } catch (error) {
        res.status(500).send("Internal Server Error");
    }
});
app.listen(3000, () => {
    // eslint-disable-next-line no-console
    console.log("启动成功");
});
```

路由和状态管理

服务端依然支持vue-router和vuex

安装

```
npm i vue-router -s
npm install -S vuex
```

配置

vue-route

```
// router/index.js
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home";
import About from "@/views/About";
Vue.use(Router);
// 服务端渲染， 实例创建使用工厂函数
// 因为每一个接口访问都需要创建对应的实例去使用
export function createRouter() {
    return new Router({
        routes: [{
                path: "/",
                component: Home
            },
            {
                path: "/about",
                component: About
            }
        ]
    });
}
```

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;
            }
        }
    })
}
```

构建

项目中创建mian，entry-client和entry-sever

使用webpack打包生成SeverBundle和ClientBundle

SeverBundle在服务端用于处理页面渲染

ClientBundle在客户端用于挂载至静态页面处理页面交互

文件结构

main.js

创建vue实例

```
import Vue from "vue";
import App from "./App.vue";
import {
    createRouter
} from "./router";
import {
    createStore
} from './store'
// 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 store = createStore()
    const app = new Vue({
        router,
        store,
        render: h => h(App)
    })
    return {
        app,
        router,
        store
    }
}
```

entry-server.js

服务端入口，用于首屏渲染

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

entry-client.js

客户端入口，用于静态内容激活(创建vue实例并执行挂载)

```
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

webpack配置依赖安装

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

具体配置

详细配置地址

```
// 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
```

创建脚本

```
"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"
},
```

宿主文件

```
<!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>
```

启动文件

```
// 加载本地文件
const fs = require("fs");
// 处理url
const path = require("path");
const express = require('express')
const server = express()
// 获取绝对路径
const resolve = dir => {
    return 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
// /index.html
server.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:导入服务端打包文件
const bundle = require(resolve("../dist/server/vue-ssr-server-
bundle.json"));
// 第 4 步:创建渲染器
const template = fs.readFileSync(resolve("../public/index.html"), "utf-8");
const clientManifest = require(resolve("../dist/client/vue-ssr-client-
manifest.json "));
const renderer = createBundleRenderer(bundle, {
    runInNewContext: false, //
https://ssr.vuejs.org/zh/api/#runinnewcontext
    template, // 宿主文件
    clientManifest // 客户端清单
});
// 路由是通配符,表示所有url都接受
server.get('*', async (req, res) => {
    console.log(req.url);
    // 设置url和title两个重要参数
    const context = {
        title: 'ssr test',
        url: req.url // 首屏地址
    }
    const html = await renderer.renderToString(context);
    res.send(html)
}) server.listen(3000, function() {
    // eslint-disable-next-line no-console
    console.log(`server started at localhost:${port}`);
});
```

数据预取

当服务器渲染需要一些异步数据时，可以选择在开始渲染前，预先获取和解析数据，并在渲染时展示出来

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

vue文件中异步数据获取

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

服务端数据获取

```
// 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);
```

客户端挂载数据

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

此时，首屏没有问题, 其他页面出现状态异常, 客户端没有执行异步代码

在每个页面的beforeMount处理下它对应的asyncData

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