React SSR 原理解析和同构实践

目录

CSR & SSR

`csr`（Client Side Rendering）就是在浏览器从服务器中获取到的只是一个带有空标签的html文件，然后执行js文件生成dom和操作dom，日常中开发的后台管理类的系统大多都是csr的模式。

`ssr`（Server Side Rendering）是在服务端已经完成渲染工作，浏览器从服务器获得的是完整的html的dom字符串。不同于以前通过后端模板等方案生成页面，现在的React、Vue、Svelte等优秀框架都有SSR的解决方案。

为什么要使用SSR?

关于这个问题尤大在Vue SSR 指南中也给出了答案。

SSR对比CSR的优点：

- 更好的SEO：对SSR的应用搜索引擎可以直接获取完全渲染的页面，但是在CSR的应用中搜索引擎获取到的只是一个空标签。
- 更快的内容到达时间。

SSR对比CSR的缺点：

- 引用成本高：ssr还需要使用node作为服务器，对于代码构建和部署也要求更高，加大了开发成本
- 更多的服务器负载。

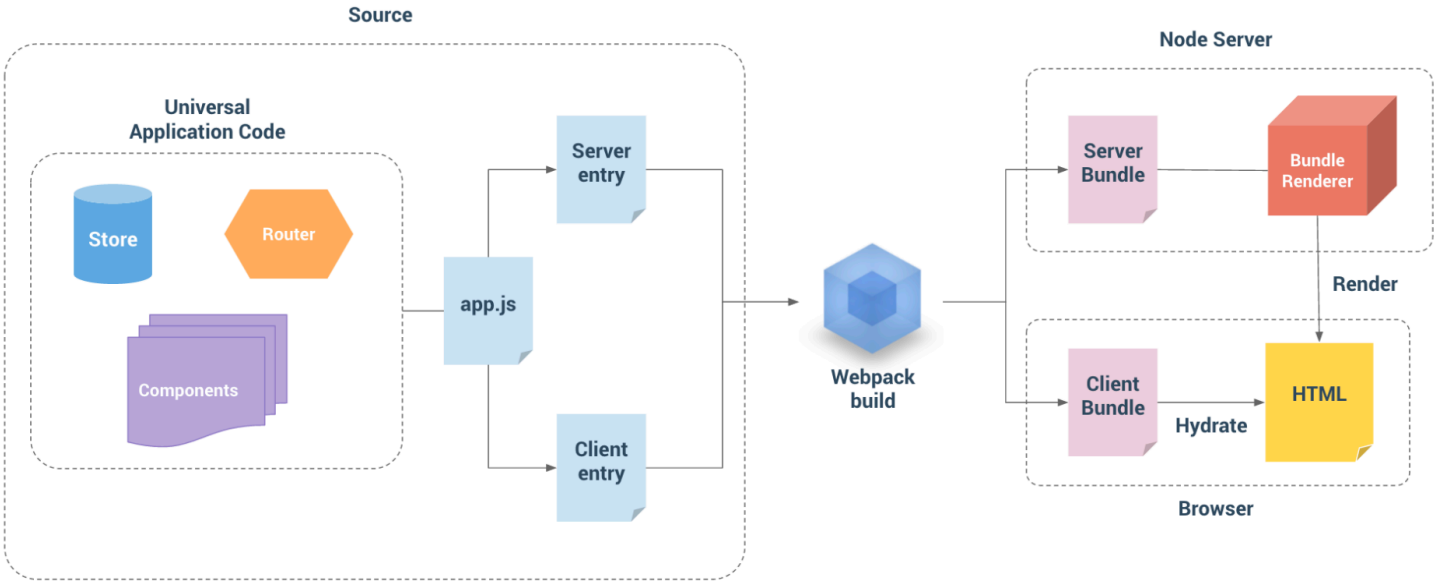
- 传统开发思路受限：在开发的时候要区分是node环境还是浏览器环境，部分生命周期在服务器端也不会生效。

more：

如果单纯的想对个别数据变动不频繁的页面做SEO，可以考虑预渲染的方案，毕竟SSR的成本是相对较高的。

- 预渲染：预渲染就是把页面生成静态的html解构，然后进行静态部署。实现方案一般是通过一个无头浏览器打开系统，等到js执行完毕，dom渲染完毕后通过 `XMLSerializer` API 进行处理最后生成静态html字符串。

SSR构建流程



上图来自Vue官网，详细展示了ssr项目的构建流程，虽然React和Vue的实现方式略有不同，但是整体思路也是如此：

- Store、Router、Compoents、app.js这些模块（一下简称web模块）是公共的，在SSR的情况下这些模块既要在客户端使用，也要在服务端使用。
- web模块被Client-entry引用通过webpack打包作为静态资源使用。
- web模块被Server-entry引用通过webpack打包被nodejs调用，用于请求页面的时候，进行组件渲染返回html字符串。
- 最终node Server返回的html字符串和js加载生成的html在浏览器端进行同构。

由上图也可以初步新建如下的文件目录：

```Plain Text
.
├── build
|   ├── base.config.js
|   ├── client.config.js
```

```
|   └── server.config.js
├── entry
|   ├── server-entry.jsx
|   └── client-entry.jsx
├── server
|   └── app.js
└── web
    ├── components
    |   └── Test.jsx
    ├── index.jsx
    └── pages
        └── Index.jsx
```

- build：webpack构建的目录
- entry：入口文件的目录
- server：node服务器的目录
- web：前端代码和资源目录

webpack构建服务端bundle

webpack构建主要是把web目录下的文件分别通过client和server的配置打包成两份代码，这里不再介绍客户端代码的打包，主要介绍一下服务端代码的打包。

</> JavaScript

```
const WebpackChain = require('webpack-chain');
const nodeExternals = require('webpack-node-externals');
const {
    resolvePath,
    isDev
} = require('./utils');

module.exports = {
    getServerConfig: function() {
        const chain = new WebpackChain();
        chain
            .entry('server')
                .add(resolvePath('entry/server-entry.js'))
                .end()
            .output
                .path(resolvePath('dist/server'))
                .filename('[name].js')
                .libraryTarget('commonjs2')
                .end()
            .when(isDev, function(chain) {
```

```
                chain.watch(true);
            })
            .target('node')
            .externals(nodeExternals({
                allowlist: [/\.(css|less|sass|scss)$/]
            }));

        return chain.toConfig();
    }
}
```

可以看到，服务器端打包和客户端的打包有一点区别：

- target: 'node'target 设置为 `node`，webpack 将在类 Node.js 环境编译代码。(使用 Node.js 的 `require` 加载 chunk，而不加载任何内置模块，如 `fs` 或 `path`)。每个target都包含各种deployment（部署）/environment（环境）特定的附加项，以满足其需求。
- output的libraryTarget设置为 'commonjs2'打包输出的代码将在node环境下运行。
- nodeExternals打包后的代码会被服务器代码（server目录）调用，所以不用把node_modules的依赖打包。

React服务端渲染 API

在开始编写代码之前，我们需要了解react实现服务端渲染必须的几个API，参考ReactDOMServer

- renderToString把React元素渲染html字符串。

</> JavaScript

```
function Com () {
  return (
    <div>123</div>
  )
}

renderToString(<Com />); // 返回<div>123</div>
```

- renderToNodeStream把React元素渲染成html，和renderToString不同的是，该方法返回一个可输出HTML字符串的 `可读流`
- hydrate如果您调用 `ReactDOM.hydrate()` 已经具有此服务器渲染标记的节点，**React将保留它并仅附加事件处理程序**，从而使您获得非常出色的首次加载体验。在SSR应用中使用 `hrdrate` 替代 `render`

</> JavaScript

```
reactDom.hydrate(
  <App></App>,
  document.getElementById('root')
```

```
)
```

简单实现SSR

- 根组件

```JavaScript
import React from 'react';

export const Index = () => {
  return (
    <div>123</div>
  )
}
```

- client-entry.jsx

客户端入口实际上就是把组件挂载到dom中：

```JavaScript
import React from 'react';
import ReactDom from 'react-dom';
import {Index} from 'web/index'

ReactDom.hydrate(
    <Index />,
  document.getElementById('app')
)
```

- server-entry.jsx

服务端入口应该导出一个函数，该函数返回解析后的html字符串：

```JavaScript
import {renderToString} from 'react-dom/server';
import {Index} from 'web/index';
import React from 'react';

export function serverRender() {
  return renderToString(
    <Index />
  )
}
```

- start.js

在启动node服务器之前，我们需要先把客户端和服务端先打包好，然后供node服务调用。start.js暴露两个方法分别是客户端执行打包和服务端打包。在启动node服务之前调用这两个函数。

</> JavaScript

```
const webpack = require('webpack');
const {getClientConfig} = require('./client.config');
const {getServerConfig} = require('./server.config');

exports.startClientServer = () => {
    return new Promise((resolve, reject) => {
        const config = getClientConfig();
        const compile = webpack(config);
        compile.run((err, stats) => {
            if (err || stats.hasErrors()) {
                console.log(err || stats.toString());
                reject();
            } else {
                resolve();
            }
        })
    })
}

exports.startServerBuild = () => {
    return new Promise((resolve, reject) => {
        webpack(getServerConfig(), (err, stats) => {
            if (err || stats.hasErrors()) {
                console.log(err || stats.toString());
                reject();
            } else {
                resolve();
            }
        })
    })
}
```

- app.js

启动一个node服务，服务端使用express框架

</> JavaScript

```
const express = require('express');
const path = require('path');
const {startClientServer, startServerBuild} = require('../build/start')
```

```
4
5  const app = express();
6  const PORT = 3000;
7
8  app.get('*', (req, res) => {
9      const {serverRender} = require(path.join(__dirname,
   '../dist/server/server.js'));
10     res.send(serverRender());
11 })
12
13 async function bootstrap () {
14     // 等待webpack打包完毕后启动服务
15     await Promise.all([startClientServer(), startServerBuild()]);
16     app.listen(PORT, () => {
17         console.log('server running~~')
18     })
19 }
20
21 bootstrap();
```

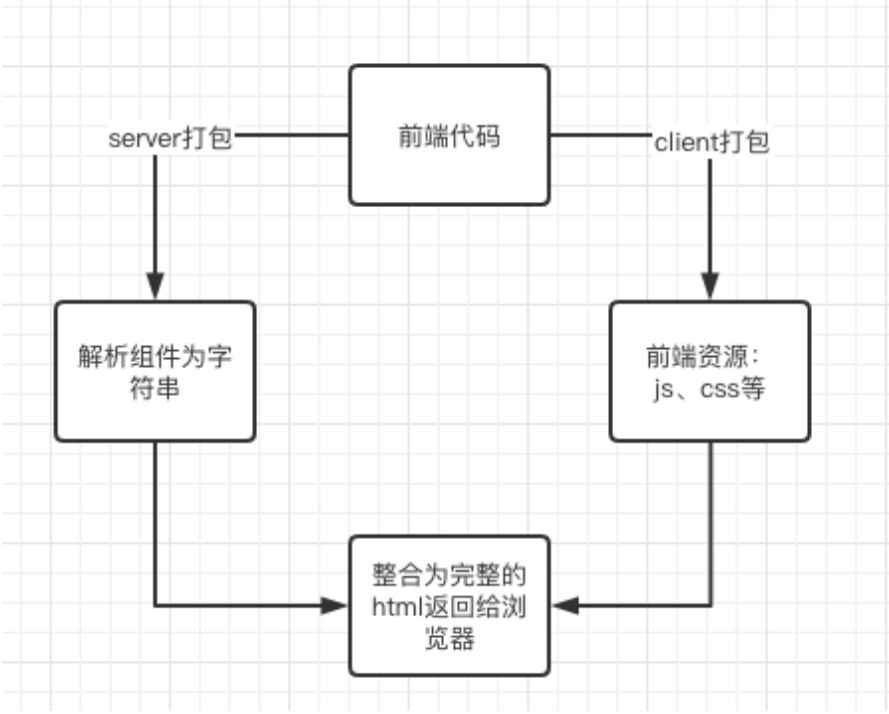
在浏览器访问 `http://localhost:3000` 就可以看到返回结果，但是还有一些问题等待解决：

1. 访问服务器的时候实际上仅仅返回的是服务端返回的字符串，并没有同构的过程
2. 开发环境的热更新
3. 路由的同构
4. 数据预取

下面我们将一一处理这些问题。

处理HTML

在上一节中提到 `访问服务器的时候实际上仅仅返回的是服务端返回的字符串，并没有同构的过程` ，实际上就是因为我们只处理了服务端渲染的字符串，并未把客户端打包的资源和浏览器整合到一起。



HTML

```
<!-- 现在访问localhost:3000返回给浏览器的结果为 -->
<div>
  123
</div>
```

HTML

```
<!-- 期望返回给浏览器的结果为 -->
<!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">
    <link rel="stylesheet" href="style.css">
    <title>Document</title>
</head>
<body>
    <div>123</div>
</body>
<script src="script.js"></script>
</html>
```

对于静态文件我们可以可以用node服务直接访问文件，但是每次打包完成后js和css的文件都是带有hash的，如何把这些文件注入到html中？在这里提供两个解决方案，一个是通过 `webpack-manifest-`

`plugin` 插件，生成打包清单；另一种是通过 `html-webpack-plugin` 把资源注入到html中。下面详细介绍这两种方案：

- webpack-manifest-plugin方案在webpack中使用该插件后，打包会额外生成一个清单文件

```JSON
{
  "dist/batman.js": "dist/batman.1234567890.js",
  "dist/joker.js": "dist/joker.0987654321.js"
}
```

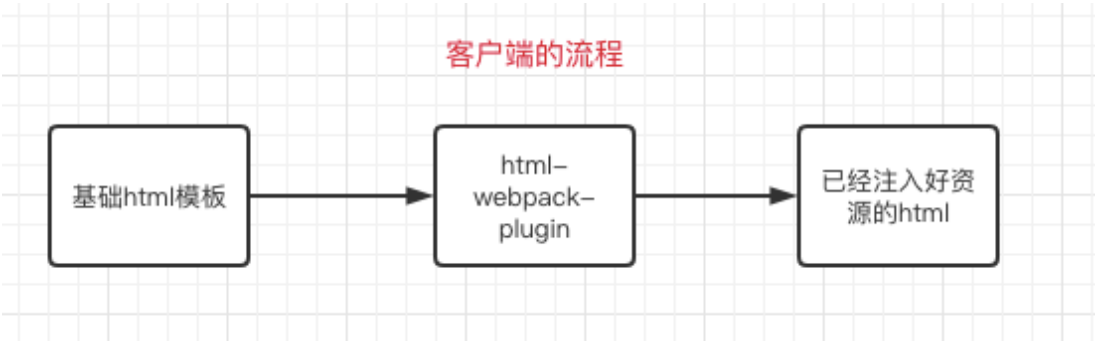
所以我们可以通过该清单直接获取打包后的js和css

```JavaScript
app.get('/', (req, res) => {
    const {serverRender} = require(serverBundlePath);
    // 引入客户端清单文件
    const clientManifest = require(clientManifestPath);
    const 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>
            <div id="app">${serverRender()}</div>
        </body>
        <script src="${clientManifest['client-entry.js']}"></script>
        </html>
    `
    res.send(html);
})
```

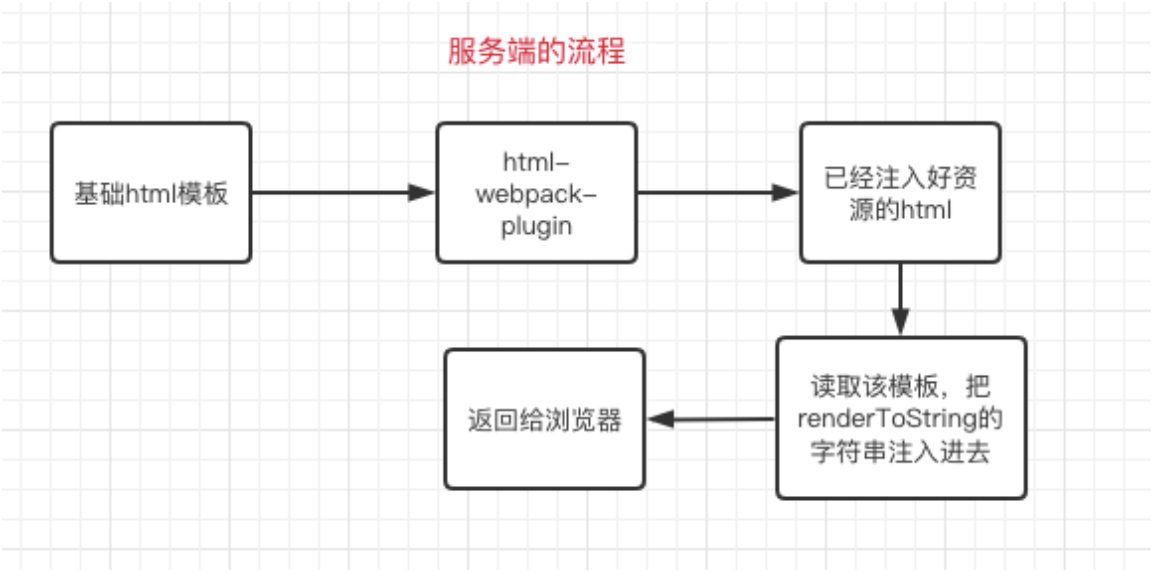
通过这种方式就可以把js和css注入到html中。

- html-webpack-plugin方案

在客户端的打包过程中，html-webpack-plugin插件会自动把js和css等注入到html中。



但是在服务端渲染中还需要对该模板进行处理把html加载进去。




```JavaScript
app.get('/', async (req, res) => {
    const {serverRender} = require(serverBundlePath);
    // 读取打包后已注入资源的html文件
    const html = (await fs.readFile(htmlPath)).toString().replace(
        'ssr-placeholder',
      serverRender()
    )
    res.send(html);
})
```

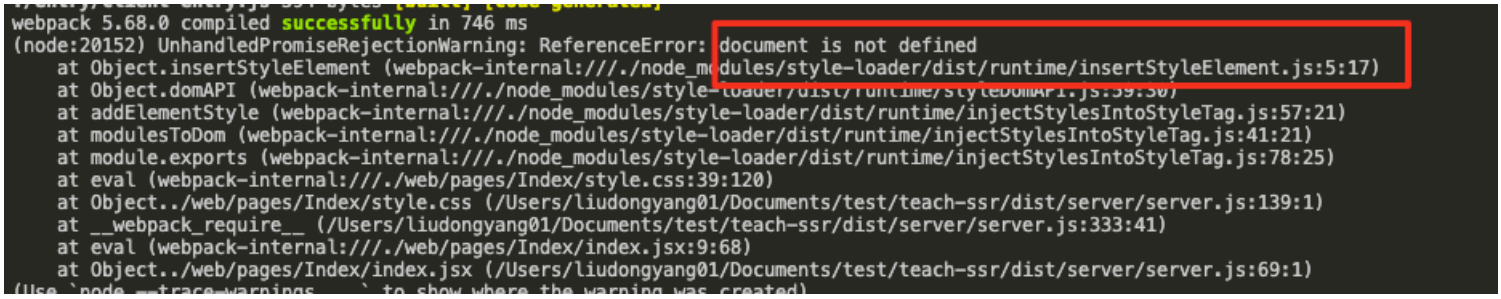
处理CSS

CSS也是开发中必不可少的，在开发的时候我们一般使用css-loader和style-loader处理css。

```JavaScript
chain
    .module
    .rule('css')
    .test(/.css$/)
```

```
    .use('style')
    .loader('style-loader')
    .end()
    .use('css')
    .loader('css-loader')
    .end()
```

但是在启动时候却出现错误



让我们看一下style-loader的insertStyleElement.js文件中做了什么?

</> JavaScript

```
/* istanbul ignore next  */
function insertStyleElement(options) {
  const element = document.createElement("style");

  options.setAttributes(element, options.attributes);
  options.insert(element, options.options);

  return element;
}

module.exports = insertStyleElement;
```

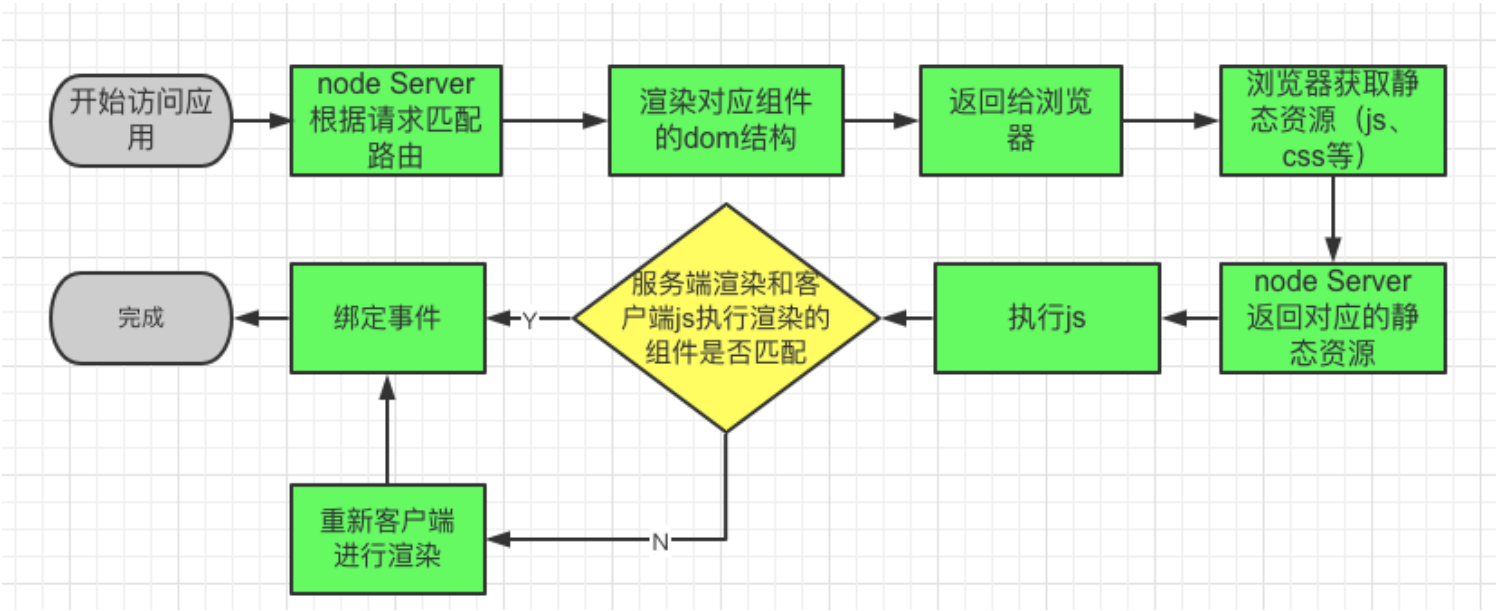
style-loader是在js运行的时候动态把代码注入到html中，因为在node环境下是没有document的，所以抛了异常，这一点在服务端渲染中需要特别注意。既然不能动态加载，我们可以把css打包成单独的文件，然后在html引入即可。在这里需要借助MiniCssExtractPlugin。

</> JavaScript

```
chain
    .module
    .rule('css')
    .test(/.css$/)
    .use('MiniCss')
    .loader(MiniCssExtractPlugin.loader)
    .end()
    .use('css')
```

```
    .loader('css-loader')
    .end()
chain
    .plugin('mini-css-extract-plugin')
    .use(MiniCssExtractPlugin, [
        {
            filename: '[name].[contenthash:6].css',
            chunkFilename: '[name].[contenthash:6].chunk.css'
        }
    ])
```

静态资源处理



上图是从访问浏览器到渲染的过程，上一节处理完成html完成后还不能算完成了一个完整的ssr，因为浏览器无法获取到js、css等资源，也无法完成react的hydrate过程，所以我们需要把js资源也返回给浏览器。

```JavaScript
app.use(express.static(path.join(__dirname, '../dist/client')));
```

只需要把客户端打包的路径设置为静态资源路径即可。

至此，我们已经完成了一个完整的ssr流程！

开发环境热更新

webpack开发环境的热更新可以直接借助DevServer实现，但是我们使用的是node服务做了资源返回，所以需要借助**webpack-dev-middleware + webpack-hot-middleware**实现热更新。

```JavaScript
```

```
const path = require('path');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');
const webpackHotMiddleware = require('webpack-hot-middleware');
const {getClientConfig} = require('./client.config');
const {getServerConfig} = require('./server.config');
const {cleanDist} = require('./utils');

exports.startClientServer = async (app) => {
    // 启动之前删除之前编译的代码
    await cleanDist(path.join(__dirname, '../dist'));
    const config = getClientConfig();
    const compile = webpack(config);
    // 使用webpackDevMiddleware
    app.use(webpackDevMiddleware(
        compile, {
            publicPath: config.output.publicPath,
            writeToDisk: true
        }
    ))
    // 热重载
    .use(webpackHotMiddleware(compile))
}
```

webpackDevMiddleware 的作用是把webpack打包的资源让node server使用

webpackHotMiddleware 的作用是热重载

同时需要修改webpack的配置

</> JavaScript

```
exports.getClientConfig = () => {
    const chain = new WebpackChain();
    // some code...

    chain.entry('client-entry')
        .when(isDev, entry => {
            // 在开发环境下需要在入口出添加
            entry.add('webpack-hot-middleware/client')
                .add(resolvePath('../entry/client-entry.tsx'))
        }, entry => {
            entry
                .add(resolvePath('../entry/client-entry'))
        })
```

```
        .end()
    // some code...
    chain
        .plugin('HotModuleReplacementPlugin')
        .use(webpack.HotModuleReplacementPlugin)
        .end()
}

exports.getServerConfig = () => {
    // ...
    isDev && chain.watch(true)
    // ...
}
```

修改入口文件

```
// client-entry.jsx
// some code ...

if (module.hot) {
    module.hot.accept.();
}
```

修改服务器代码

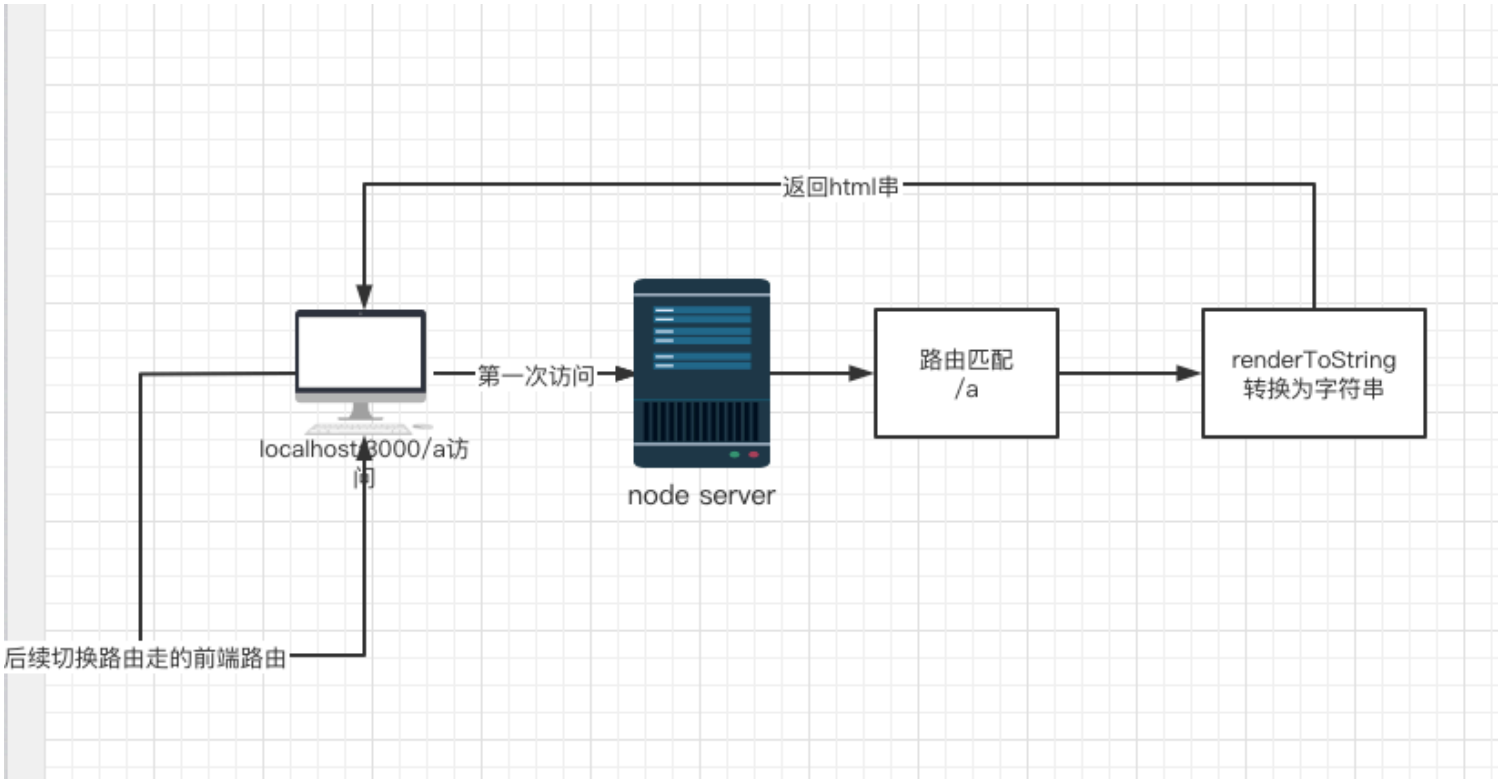
```
// app.js
// 因为webpackDevMiddleware已经对资源做了处理，所以不需要express再处理
// app.use(express.static(path.join(__dirname, '../dist/client')));

async function bootstrap () {
    // 把app传入到startClientServer中
    await Promise.all([startClientServer(app), startServerBuild()]);
    app.listen(PORT, () => {
        console.log('server running~~')
    })
}
```

经过配置后，当修改前端代码的时候就可以热更新了。当修改服务端代码的时候可以使用nodemon等工具使服务重新启动，当服务重新启动的时候，热更新的websocket链接就会断开，需要重新刷新页面重新建立连接。

路由同构

在react项目中，路由一般使用的是react-router（本文使用的版本为5），react-router同样也支持服务端渲染，在进行同构之前我们需要了解类似于react-router路由框架在切换路由的时候，是不会向服务器发送请求的。服务端渲染的时候流程如下：



1. 当首次进入系统访问localhost:3000/a的时候会首先访问node服务器
2. node服务器收到请求后，会进行路由匹配，根据路由决定应该返回哪个页面（组件）的字符串给浏览器
3. html发送到浏览器后，执行js脚本，此时客户端的路由（react-router-dom）会再次执行，决定要渲染哪个页面（组件）
4. 如果服务器生成的字符串和前端js脚本的字符串匹配则hydrate完成，否则失败。
5. 后续的路由切换都是前端路由切换，不会产生请求。

- 添加前端路由

</> JavaScript

```
// client-entry.js
import React from 'react';
import ReactDom from 'react-dom';
import {BrowserRouter} from 'react-router-dom';
import {Index} from 'web/index';

ReactDom.hydrate(
  <BrowserRouter>
      <Index />
```

```
  </BrowserRouter>,
  document.getElementById('app')
)

// web/index.jsx
import React from 'react';
import {Switch, Route} from 'react-router-dom';
import {routerList} from './router';

export const Index = () => {
    return (
        <Switch>
            {
                routerList.map(Item => {
                    return (
                        <Route key={Item.path} path={Item.path} exact={Item.exact} component={Item.component}>
                        </Route>
                    )
                })
            }
        </Switch>
    )
}

// web/router
import {Index} from '../pages/Index';
import { About } from '../pages/About';

export const routerList = [
    {
        path: '/',
        component: Index,
        exact: true
    },
    {
        path: '/about',
        component: About
    }
]
```

添加前端路由和客户端渲染添加路由是一样的，因为最终在客户端运行的还是前端路由。

- server端路由的处理

在服务端需要根据请求判断请求的是哪个页面和组件，然后通过renderToString将其转换为字符串返回给浏览器。首先需要实现一个查找组件的方法，react-router恰好也提供了matchPath方法。

```JavaScript
import {routerList} from 'web/router'
import {matchPath} from 'react-router-dom'

export function findRoute(path) {
    return routerList.find(item => matchPath(path, item))
}
```

React-router在服务端渲染中也提供了StaticRouter用来替代BrowserRouter

```JavaScript
import {renderToString} from 'react-dom/server';
import {StaticRouter} from 'react-router-dom';
import {Index} from 'web/index';
import React from 'react';
import {findRoute} from 'web/router/findRoute';

export function serverRender(path) {
    const router = findRoute(path);
    if (router) {
        return renderToString(
            <StaticRouter location={router.path}>
                <Index />
            </StaticRouter>
        );
    } else {
        return '404';
    }
}
```

服务端渲染的路由同构相对其他模块比较简单，关键在于如何匹配路由渲染相对应的路由。

数据预取

目前实现的功能依然存在一个比较严重的问题，服务端返回浏览器的html并无请求的数据，只有静态的html解构，所以我们要在服务端提前获取到数据发送给浏览器。前面也提到过，服务端返回的结构要和客户端渲染的html一致，这样客户端只需要完成事件绑定，否则会在客户端再进行一次解析渲染，所以我们需要解决以下问题：

1. 如何在服务端预取数据
2. 取到数据之后如何保证服务端和客户端渲染一致

- 服务端预取数据

比较明确的是预取数据的这个接口在服务端和客户端都会调用，所以请求的时候既要可以在浏览器环境成功请求到数据，又可以在node环境请求到数据。这里推荐axios，axios对以上两个环境都有很好的支持。我们为每个页面都建一个fetch文件fetch文件就是一个数据请求预取的函数，下面用定时器模拟一下

```JavaScript
// fetch.js
export function fetch() {
    return new Promise(r => {
        setTimeout(() => {
            r({
                name: 'jack',
                age: 18
            })
        }, 3000)
    })
}
```

把数据预取的函数和路由进行绑定：

```JavaScript
import {Index} from '../pages/Index';
import { About } from '../pages/About';
import { fetch as IndexFetch } from '../pages/Index/fetch';
import { fetch as AboutFetch } from '../pages/About/fetch';

export const routerList = [
    {
        path: '/',
        component: Index,
        exact: true,
        // 把fetch方法和路由绑定
        fetch: IndexFetch
    },
    {
        path: '/about',
        component: About,
        fetch: AboutFetch
    }
]
```

绑定完成后在服务端通过findRoute可以找到该方法调用并获取数据，在客户端可以直接把fetch传到组件上并

调用。

```JavaScript
// server-entry.js
import {renderToString} from 'react-dom/server';
import {StaticRouter} from 'react-router-dom';
import {Index} from 'web/index';
import React from 'react';
import {findRoute} from 'web/router/findRoute';

export async function serverRender(path) {
    const router = findRoute(path);
    const res = await router.fetch();
    if (router) {
        const content = renderToString(
            <StaticRouter location={router.path} context={{
                initData: res
            }}>
                <Index />
            </StaticRouter>
        );
        return {
            content,
            state: res
        }
    }
}
```

React-router中 `StaticRouter` 支持context属性，可以把数据传到props中。

在组件中 `props.staticContext.initData` 的值为context的值。

```JavaScript
export const Index = (props) => {
    return (
        <div>
            <div>
                {props.staticContext.initData.name} // jack
            </div>
        </div>
    )
}
```

- 数据同构

数据预取的时候，Index页面组件通过 `props.staticContext.initData` 获取预取到的数据，但是 `BrowserRouter` 并没有 `StaticRouter` 的context属性，所以要在客户端给Index组件的 `props` 添加 `staticContext.initData`

```JavaScript
routerList.map({path, exact, Component} => {
  return (
    <Route
      key={path}
      path={path}
      exact={exact}
      render={(props) => <Component {...props} staticContext={initData: data}  />}>
    </Route>
  )
})
```

通过这种方式使客户端和服务端有一致的数据，也就保证了双端渲染的一致性。

在服务端中通过每个页面的fetch获取到了数据，那么在浏览器如何获取到 `data` 数据。首先要排除在客户端再次调用fetch这种方法，不仅仅重复调用浪费资源还会在初始化渲染的时候使同构失败。比较理想的方法就是可以在浏览器端获取到服务端获取到的数据，我们可以通过对服务端数据序列化后传给前端去实现：

```JavaScript
// app.js
const {content, state} = await serverRender(url);
const 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>
  <div id="app">${content}</div>
  </body>
  <script>
  window.__INIT_STATE__=${JSON.stringify(state)}
  </script>
  <script src="${clientManifest['client-entry.js']}"></script>
  </html>
`
```

在客户端可以使用 `window.__INIT_STATE__` 可以获取服务端预取到的数据。

JavaScript

```
routerList.map({path, exact, Component} => {
  return (
    <Route
      key={path}
      path={path}
      exact={exact}
      render={(props) => <Component {...props} staticContext={initData: window.__INIT_STATE__}  />}>
    </Route>
  )
})
```

到这里便完成了数据预取和同构，但是在前面我们提到，ssr的模式只是初次请求的时候使用服务端渲染，后续切换页面都是客户端的路由切换行为，那么当前实现方式，在路由切换的时候就会出现问题，因为我们只有初次渲染的那个页面的数据，而且我们希望每次切换路由会请求新数据，所以在后续的切换的时候，需要先获取数据再传给页面组件，我们使用高阶组件实现。

JavaScript

```
// WrapperComponent
import React from "react";
import {withRouter} from 'react-router-dom';
// 标志是否是初次渲染
let hasRender = false;
export const WrapperComponent = (Component) => {
    return withRouter(class extends React.Component {
        constructor (props) {
            super(props);

            this.state = {
                // 服务端渲染取props.staticContext，客户端初次渲染取window.__INIT_STATE__
                staticContext: props.staticContext || {
                    initData: window && !hasRender && window.__INIT_STATE__
                }
            }
        }
        componentDidMount () {
          // 如果是初次渲染，直接取window.__INIT_STATE__，不用再请求
            if (!hasRender) {
                hasRender = true;
```

```
            } else {
              // 后续切换路由的时候，请求数据并更新
                this.props.fetch().then(res => {
                    console.log('fetch')
                    this.setState({
                        staticContext: {
                            initData: res
                        }
                    });
                })
            }
        }

        render () {
            return (
                <Component {...this.props} staticContext=
{this.state.staticContext}></Component>
            )
        }
    })
}
```

</> JavaScript

```
routerList.map(Item => {
  const NewComponent = WrapperComponent(Item.component);
  return (
    <Route
        key={Item.path}
      path={Item.path}
            exact={Item.exact}
            render={(props) => <NewComponent {...props} fetch={Item.fetch} />}
    >
    </Route>
    )
})
```

写在最后

在搭建ssr框架的时候不仅仅需要熟悉react、vue等前端框架，还需要对webpack、nodejs等有一定的了解，而且真实的线上环境会比文章描述要复杂的更多，所以在技术选型的时候请确定你真的需要ssr，并已经做好了要解决诸多问题的准备。建议大家选择一些比较稳定的服务端渲染框架: next.js、nuxt.js，也强烈推荐一个名叫ssr的框架。

代码

https://console.cloud.baidu-int.com/devops/icode/repos/baidu/personal-code/ssr-pro/tree/master