实战篇：基于single-spa的分布式微前端项目实战2

1. layout布局引擎的学习

上节课通过简单的案例认识了single-spa的微前端运行模式和基本使用， 本节课在微前端的基础上进一步提升项目能力和体验，首先介绍single-spa的布局引擎layout。layout是在路由加载的基础上进一步出现的微件化加载模式，他与纯微件的思想并不相同，但基于这个理论可以结合WebComponent实现真正的微件加载。

1.1 初始化布局引擎

布局引擎的构建方式，步骤如下：

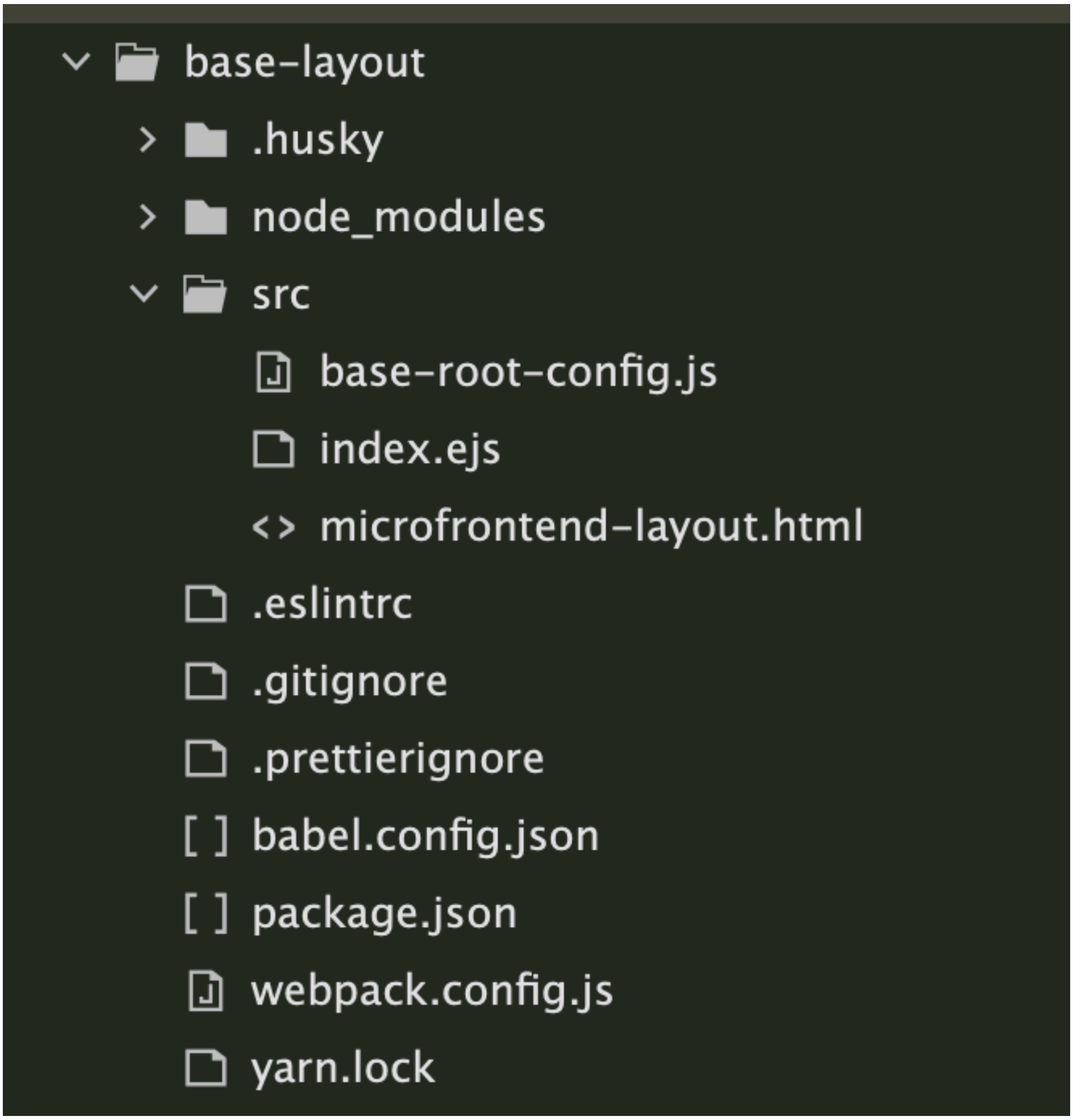
1. 在项目根目录下初始化single-spa基座项目，代码如下：

   ```
   npm init single-spa
   ```

2. 在项目配置选项中选择合适的参数，代码如下：

   ```
   zhangyunpeng@zhangyunpengdeMacBook-Pro root % npm init single-spa
   ? Directory for new project ./base-layout
   ? Select type to generate single-spa root config
   ? Which package manager do you want to use? yarn
   ? Will this project use Typescript? No
   ? Would you like to use single-spa Layout Engine Yes
   ? Organization name (can use letters, numbers, dash or underscore) base
   ```

3. 创建的项目结构，如图所示。



```
base-layout
  .husky
  node_modules
  src
    base-root-config.js
    index.ejs
    microfrontend-layout.html
  .eslintrc
  .gitignore
  .prettierignore
  babel.config.json
  package.json
  webpack.config.js
  yarn.lock
```

4. 本次创建的项目结构中多了一个microfrontend-layout.html文件。

1.2 学习layout项目

在项目启动时会发现该项目与默认的single-spa项目初始界面完全相同，接下来对生成的文件做进一步的学习。首先是index.ejs的内容，代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Root Config</title>
```

```html
  <!--
    Remove this if you only support browsers that support async/await.
    This is needed by babel to share largeish helper code for compiling async/await in
older
    browsers. More information at https://github.com/single-spa/create-single-
spa/issues/112
  -->
  <script src="https://cdn.jsdelivr.net/npm/regenerator-runtime@0.13.7/runtime.min.js">
</script>

  <!--
    This CSP allows any SSL-enabled host and for arbitrary eval(), but you should limit
these directives further to increase your app's security.
    Learn more about CSP policies at https://content-security-policy.com/#directive
  -->
  <meta http-equiv="Content-Security-Policy" content="default-src 'self' https:
localhost:*; script-src 'unsafe-inline' 'unsafe-eval' https: localhost:*; connect-src
https: localhost:* ws://localhost:*; style-src 'unsafe-inline' https:; object-src
'none';">
  <meta name="importmap-type" content="systemjs-importmap" />
  <!-- If you wish to turn off import-map-overrides for specific environments (prod),
uncomment the line below -->
  <!-- More info at https://github.com/joeldenning/import-map-
overrides/blob/master/docs/configuration.md#domain-list -->
  <!-- <meta name="import-map-overrides-domains" content="denylist:prod.example.com" />
-->

  <!-- Shared dependencies go into this import map. Your shared dependencies must be of
one of the following formats:

    1. System.register (preferred when possible) -
https://github.com/systemjs/systemjs/blob/master/docs/system-register.md
    2. UMD - https://github.com/umdjs/umd
    3. Global variable

    More information about shared dependencies can be found at https://single-
spa.js.org/docs/recommended-setup#sharing-with-import-maps.
  -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-
spa.min.js"
      }
    }
  </script>
  <link rel="preload" href="https://cdn.jsdelivr.net/npm/single-
spa@5.9.0/lib/system/single-spa.min.js" as="script">
```

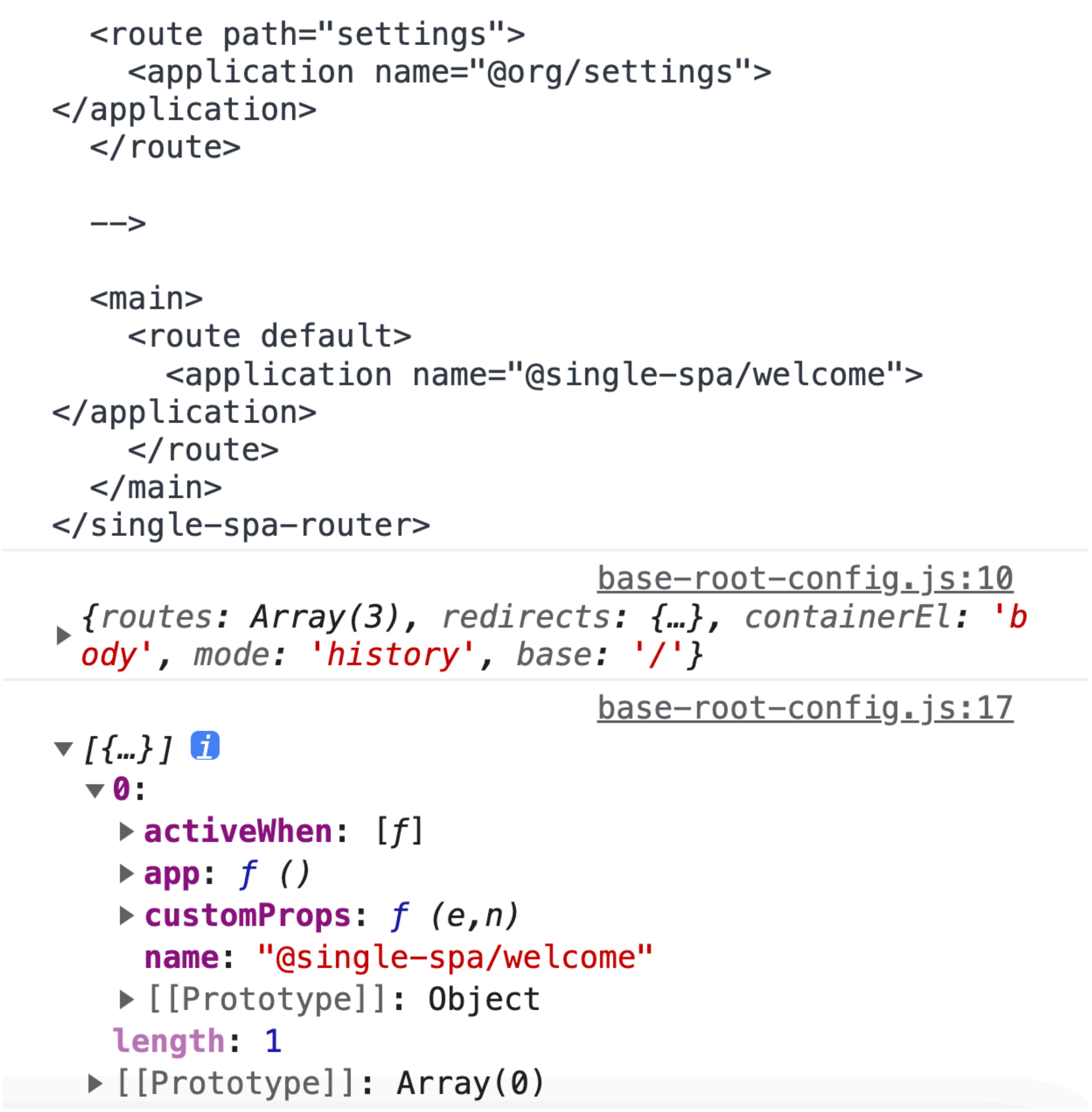
```
  <!-- Add your organization's prod import map URL to this script's src  -->
  <!-- <script type="systemjs-importmap" src="/importmap.json"></script> -->

  <% if (isLocal) { %>
  <script type="systemjs-importmap">
    {
      "imports": {
        "@single-spa/welcome": "https://unpkg.com/single-spa-welcome/dist/single-spa-
welcome.js",
        //这里会发现本项目将@base/root-confi本身也作为子应用进行了管理
        "@base/root-config": "//localhost:9000/base-root-config.js"
      }
    }
  </script>
  <% } %>

  <!--
    If you need to support Angular applications, uncomment the script tag below to
ensure only one instance of ZoneJS is loaded
    Learn more about why at https://single-spa.js.org/docs/ecosystem-angular/#zonejs
  -->
  <!-- <script src="https://cdn.jsdelivr.net/npm/zone.js@0.11.3/dist/zone.min.js">
</script> -->

  <script src="https://cdn.jsdelivr.net/npm/import-map-overrides@2.2.0/dist/import-map-
overrides.js"></script>
  <% if (isLocal) { %>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/extras/amd.js">
</script>
  <% } else { %>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.min.js">
</script>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/extras/amd.min.js">
</script>
  <% } %>

</head>
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <script>
    System.import('@base/root-config');
  </script>
  <import-map-overrides-full show-when-local-storage="devtools" dev-libs></import-map-
overrides-full>
</body>
```

```
</html>
```

观察项目后会发现项目的index.ejs并没有任何变化，只是将@base/root-config也作为子应用进行了管理。接下来查看base-root-config.js文件，代码如下。

```
//引用注册应用和启动服务模块
import { registerApplication, start } from "single-spa";
//引用布局引擎工具
//constructApplications是根据路由对象生成应用信息的工具函数
//constructRoutes能根据html代码生成路由对象
//constructLayoutEngine可以将路由对象和应用对象形成布局对象
import {
  constructApplications,
  constructRoutes,
  constructLayoutEngine,
} from "single-spa-layout";
//读取microfrontend-layout.html文件
import microfrontendLayout from "./microfrontend-layout.html";
//将布局内容转换成路由数组
const routes = constructRoutes(microfrontendLayout);
//将路由信息中包含的应用初始化
const applications = constructApplications({
  routes,
  loadApp({ name }) {
    return System.import(name);
  },
});
//将应用信息融入布局引擎
const layoutEngine = constructLayoutEngine({ routes, applications });
//遍历应用数据注册应用
applications.forEach(registerApplication);
//激活引擎
layoutEngine.activate();
//开启服务
start();
```

阅读完代码后会发现使用了layout引擎后整体的初始化代码完全变了，不需要在JavaScript中手动注册应用，便可以直接实现应用注册，可以将文件中的变量按顺序输出到控制台，如图所示。



```
  <route path="settings">
    <application name="@org/settings">
</application>
  </route>

  -->

  <main>
    <route default>
      <application name="@single-spa/welcome">
</application>
    </route>
  </main>
</single-spa-router>
```

```
                                   base-root-config.js:10
▶ {routes: Array(3), redirects: {…}, containerEl: 'body', mode: 'history', base: '/'}
                                   base-root-config.js:17
▼ [{…}] ℹ
  ▼ 0:
    ▶ activeWhen: [ƒ]
    ▶ app: ƒ ()
    ▶ customProps: ƒ (e,n)
      name: "@single-spa/welcome"
    ▶ [[Prototype]]: Object
    length: 1
  ▶ [[Prototype]]: Array(0)
```

最后查看microfrontend-layout.html内容（详细内容可以参考文档：https://zh-hans.single-spa.js.org/docs/layout-definition），代码如下：

```
<single-spa-router>
  <!-- Example layouts you might find helpful:
  layout支持基座应用同时加载多个子应用作为运行容器
  可以使用<application>标签直接加载子应用名称
  还可以通过path为子应用设置路由
  <nav>
    <application name="@org/navbar"></application>
  </nav>
  <route path="settings">
    <application name="@org/settings"></application>
  </route>
  -->

  <main>
    <!-- route标签相当于路由容器，default代表默认加载的微应用 -->
    <route default>
      <application name="@single-spa/welcome"></application>
    </route>
  </main>
</single-spa-router>
```

2. 老项目改造微前端子应用

2.1 原生webpack项目的创建

上节课学习了Vue项目的创建，本节课学习如何将原生的HTML静态页面项目改造成single-spa的子应用，步骤如下：

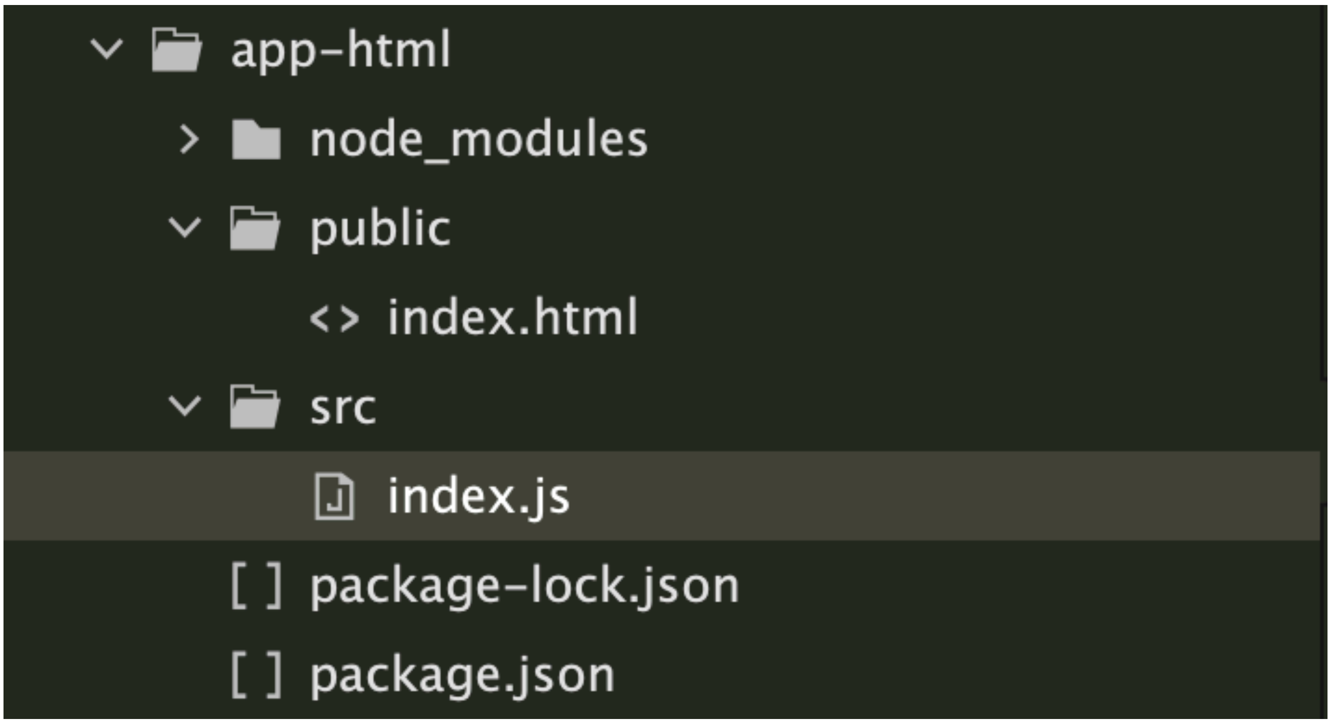
1. 在项目中新建子应用app-html空文件夹
2. 在应用中初始化package.json文件，代码如下：

   ```
   npm init -y
   ```

3. 在应用中安装依赖，代码如下：

   ```
   npm i webpack webpack-cli webpack-dev-server -D
   ```

4. 在项目中新建public，src文件夹并在文件夹内部初始化相应文件，如图所示。



```
app-html
  node_modules
  public
    index.html
  src
    index.js
  package-lock.json
  package.json
```

5. 安装项目所需的loader和插件，代码如下：

```
npm i single-spa-html -s
npm i html-loader  -D
```

6. 在创建webpack.config.js并初始化配置信息，代码如下：

```
const path = require('path')
module.exports = {
  //设置运行模式
  mode:'development',
  //配置入口文件
  entry:{
    index:'./src/index.js'
  },
  //配置源代码映射
  devtool:'source-map',
 //配置devServer
  devServer:{
    //授权服务允许跨域，否则基座应用无法加载该文件
    headers: {'Access-Control-Allow-Origin': '*'},
    //配置端口号
    port:8086,
    //配置域名
    host:'localhost',
    //配置静态资源路径
    static:[path.resolve(__dirname,'dist'),path.resolve(__dirname,'public'),]
  },
  module:{
    //配置html-loader用以让js支持html翻译
    rules:[
      {
        test:/\.html$/,
        use:{loader:'html-loader'}
      }
    ]
  },
  //配置输出数据
  output:{
    //生成的文件放到js/app.js路径下
    filename:'js/app.js',
    //生成的文件保存到dist文件夹中
    path:path.resolve(__dirname,'dist'),
    //必须配置明确的publicPath否则基座应用无法识别该项目
    publicPath:'/',
    //配置包名和导出模式，这里使用umd模式以暴露bootstrap、mount和unmount，默认模式无法被基座应
用加载
    library:{
      name:'@app/html',
      type:'umd'
    }
  }
}
```

7. 接下来在index.js文件中实现视图内容的输出，代码如下：

```
import singleSpaHtml from 'single-spa-html';
import html from '../public/index.html'
const htmlLifecycles = singleSpaHtml({
  template: html,
})

export const bootstrap = htmlLifecycles.bootstrap;
export const mount = htmlLifecycles.mount;
export const unmount = htmlLifecycles.unmount;
```

8. 接下来改造index.html文件，代码如下：

```
<!-- 这里最好使用html代码片段，这种加载模式相当于单独组件的子应用 -->
<div>
  <button id="btn">点我</button>
  <script type="text/javascript">
    document.querySelector('#btn').onclick = function(){
      console.log('点击事件')
    }
  </script>
</div>
```

9. 接下来在layout引擎项目中改造index.ejs将子应用注册到imports-map中，代码如下：

```
<script type="systemjs-importmap">
    {
      "imports": {
        "@base/root-config": "//localhost:9000/base-root-config.js",
        "@app/html":"//localhost:8086/js/app.js"
      }
    }
  </script>
```

10. 在layout引擎项目中的microfrontend-layout.html文件中做如下修改：

```
<route default>
  <application name="@app/html"></application>
</route>
```

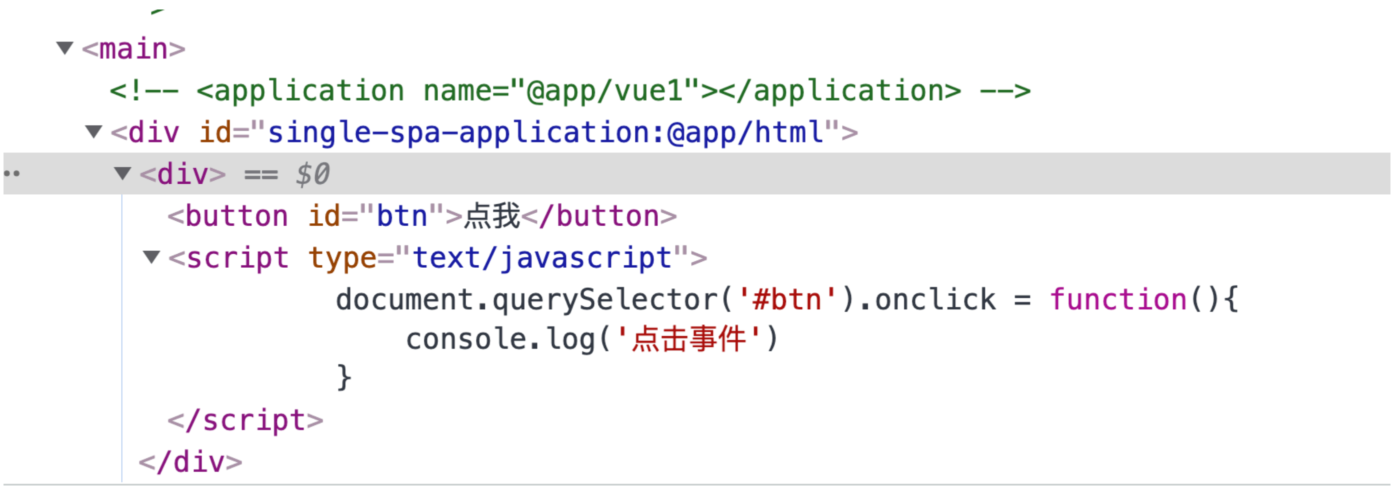
11. 在webpack项目中创建启动命令，代码如下：

```
{
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "webpack serve --config webpack.config.js --color --progress --hot"
  }
}
```

12. 启动webpack项目和基座应用并访问http://localhost:9000，查看是否可以展示静态页面的内容，若视图中出现按钮，代表成功。

2.2 静态页面的JavaScript失效问题

在成功加载自定义子应用时，会发现html文件的JavaScript代码并没有执行。查看Web控制台会发现，该代码已经被成功的加载但是并没有被执行，如图所示。

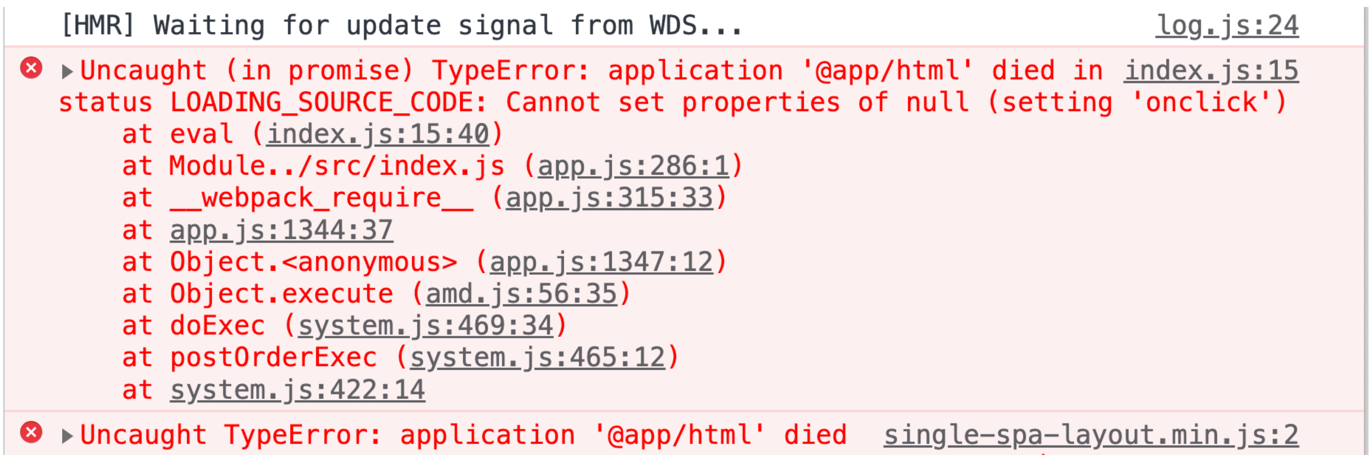


```
▼<main>
   <!-- <application name="@app/vue1"></application> -->
  ▼<div id="single-spa-application:@app/html">
    ▼<div> == $0
       <button id="btn">点我</button>
      ▼<script type="text/javascript">
                 document.querySelector('#btn').onclick = function(){
                     console.log('点击事件')
                 }
       </script>
     </div>
```

若想通过JavaScript为静态页面中的DOM节点绑定事件，则需要在项目的index.js中操作，遂在index.js文件中编写如下代码：

```
import singleSpaHtml from 'single-spa-html';
import html from '../public/index.html'
const htmlLifecycles = singleSpaHtml({
  template: html,
})
document.querySelector('#btn').onclick = function(){
  console.log(123)
}
export const bootstrap = htmlLifecycles.bootstrap;
export const mount = htmlLifecycles.mount;
export const unmount = htmlLifecycles.unmount;
```

代码编写后会出现视图无法显示的问题，控制台会出现如下错误，如图所示。



```
[HMR] Waiting for update signal from WDS...                                log.js:24
▶Uncaught (in promise) TypeError: application '@app/html' died in index.js:15
status LOADING_SOURCE_CODE: Cannot set properties of null (setting 'onclick')
    at eval (index.js:15:40)
    at Module../src/index.js (app.js:286:1)
    at __webpack_require__ (app.js:315:33)
    at app.js:1344:37
    at Object.<anonymous> (app.js:1347:12)
    at Object.execute (amd.js:56:35)
    at doExec (system.js:469:34)
    at postOrderExec (system.js:465:12)
    at system.js:422:14
▶Uncaught TypeError: application '@app/html' died   single-spa-layout.min.js:2
```

这是因为在index.js执行阶段为load子应用阶段，此时基座应用并没有渲染DOM节点，导致了此时无法获取子应用的真实DOM对象，想要为这种应用绑定交互事件必须等待子应用mount执行完毕，但该代码不可以直接编写在mount中，因为mount函数在执行后，需要等待基座应用识别到Promise状态变更后才能真正渲染，所以需要改造事件绑定代码，代码如下：

```
import singleSpaHtml from 'single-spa-html';
import html from '../public/index.html'
const htmlLifecycles = singleSpaHtml({
  template: html,
})

export const bootstrap = htmlLifecycles.bootstrap;
export const mount = props => {
  return htmlLifecycles.mount(props).then(() => {
    // 在此位置才能保证DOM渲染完成
    document.querySelector('#btn').onclick = function(){
      console.log(123)
    }
  })
};
export const unmount = htmlLifecycles.unmount;
```

完成编码改造后会发现此时静态页面中的按钮事件可以正常工作了。

3. 增加子应用实现视图布局和通信

3.1 迁移现有的Vue子应用

上节课介绍了如何使用自带插件的Vue框架实现子应用的构建，本节课以创建好的Vue项目改造为核心介绍如何改造现有的Vue项目实现子应用的创建，步骤如下：

1. 通过@vue/cli初始化一个带路由的Vue3项目（确保@vue/cli为最新版本，这样比较好迁移）。
2. 初始化完成后在项目中安装single-spa的依赖，代码如下：

```
npm i single-spa-vue -s
```

3. 在vue.config.js文件中输入改造现有项目的配置文件，代码如下：

```
//导入定义对象
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  //关闭lint检测
  lintOnSave:false,
  transpileDependencies: true,
  //配置公共路径这里需要带上域名防止子应用以来加载丢失
  publicPath:'http://localhost:8085/',
  //配置服务为固定地址并允许跨域
  devServer:{
    headers: {'Access-Control-Allow-Origin': '*'},
    port:8085,
    host:'localhost'
  },
  chainWebpack(webpackConfig){
    //关闭代码拆分，这里极为重要否则子应用无法正确识别
    webpackConfig.optimization.delete("splitChunks");
  },
  configureWebpack:{
    //开启源代码映射
    devtool:'source-map',
    output:{
      //设置导出为umd模式
      library:{
        name:'@app/vue',
        type:'umd'
      }
    }
  }
})
```

4. 改造main.js文件，代码如下：

```
import { h, createApp } from 'vue';
import singleSpaVue from 'single-spa-vue';
import App from './App.vue';
import router from './router'

const vueLifecycles = singleSpaVue({
  createApp,
  appOptions: {
    render() {
      return h(App, {

      });
    }
  },
  handleInstance(app) {
    app.use(router);
  }
});
export const bootstrap = vueLifecycles.bootstrap;
export const mount = vueLifecycles.mount;
export const unmount = vueLifecycles.unmount;
```

5. 在App.vue外层嵌套id为app的div防止样式丢失，代码如下：

```
<template>
  <div id="app">
    <nav>
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </nav>
    <router-view/>
  </div>
</template>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}

nav {
  padding: 30px;
}

nav a {
  font-weight: bold;
  color: #2c3e50;
}

nav a.router-link-exact-active {
  color: #42b983;
}
</style>
```

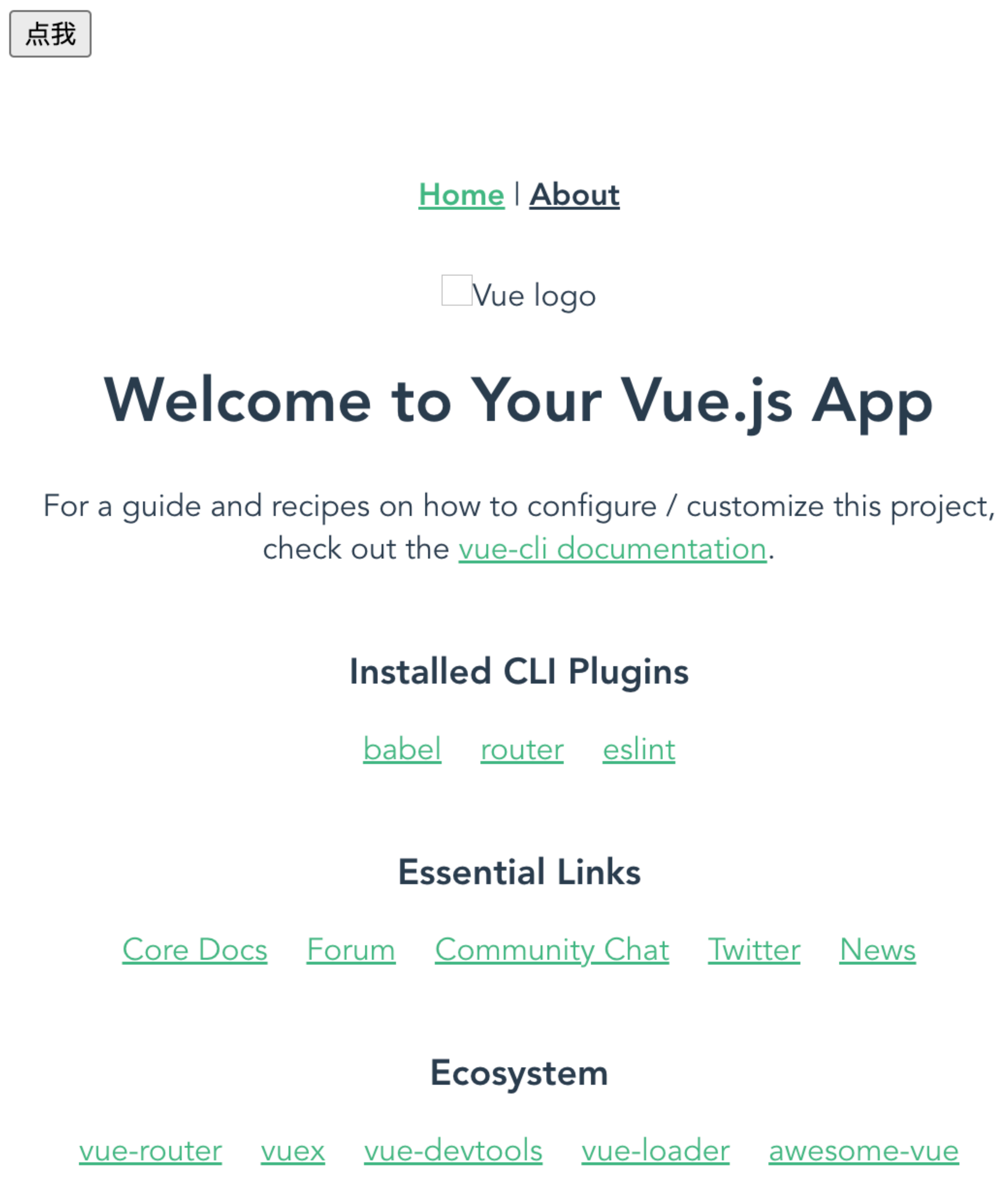
6. 在基座应用中追加子应用，代码如下：

```
<script type="systemjs-importmap">
    {
      "imports": {
        "@base/root-config": "//localhost:9000/base-root-config.js",
        "@app/html":"//localhost:8086/js/app.js",
        "@app/vue":"//localhost:8085/js/app.js"
      }
    }
  </script>
```

7. 在布局视图中将Vue子应用展示到视图中，代码如下：

```
<!-- 模拟导航和视图 -->
<nav>
  <application name="@app/html"></application>
</nav>
<main>
  <application name="@app/vue"></application>
</main>
```

8. 配置到此后启动基座应用和两个子应用，会发现视图中可以正确渲染两个应用，Vue欢迎页面的图片会渲染失败，这里是由于配置不全导致的，如图所示。


点我

Home | About

Vue logo

Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.

Installed CLI Plugins

babel router eslint

Essential Links

Core Docs Forum Community Chat Twitter News

Ecosystem

vue-router vuex vue-devtools vue-loader awesome-vue

3.2 实现跨应用跳转视图

1. 改造@app/html中的视图代码为如下代码：

```
<div>
  <button class="btn" data-path="/">首页</button>
  <button class="btn" data-path="/about">关于</button>
</div>
```

2. 改造@app/html中的index.js文件，代码如下：

```
import singleSpaHtml from 'single-spa-html';
//导入singleSpa对象用于实现视图跳转
import singleSpa from 'single-spa'
// console.log(singleSpa)
import html from '../public/index.html'

const htmlLifecycles = singleSpaHtml({
  template: html,
})

export const bootstrap = htmlLifecycles.bootstrap;
export const mount = props => {
  return htmlLifecycles.mount(props).then(() => {
    // 为所有按钮绑定事件
    document.querySelectorAll('.btn').forEach(btn => {
      btn.onclick = (e) => {
        // 获取data-path的值
        let path = e.target.dataset.path
        console.log(path)
        // 通过公共路由组件实现视图跳转
        singleSpa.navigateToUrl(path)
      }
    })
  })
};
export const unmount = htmlLifecycles.unmount;
```

3. 配置完该文件后，网页会提示can not resolve single-spa的错误，这是因为本项目中并没有引用si ngle-spa库，这里就需要利用single-spa的公共依赖部分，在index.ejs文件的systemjs-importmap部分包含一个imports对象，这个对象就是用于注册全局公共依赖的部分，配置在这里的依赖可以在index.ejs中通过System.import()实现全局应用，所以接下来需要改造@app/html的webpack.config.js文件，通过externals属性将依赖设置为加载CDN模式，并在index.ejs最下方的script标签中追加System.import('single-spa')，代码如下：

```
externals:['single-spa']//配置single-spa不从webpack依赖中解析
```

4. 重启@app/html子应用，再次点击屏幕上的两个按钮会发现@app/vue子应用的视图可以实现跳转，如此便实现了跨应用的跳转方式。

3.3 加载公共依赖的方式

single-spa中的importmap可以实现公共依赖的加载，解下来以CDN的JQuery为例子，在两个子应用中共享全局依赖，步骤如下：

1. 在root中的index.ejs文件中定义JQuery全局依赖，代码如下：

```
<!DOCTYPE html>
<html lang="en">
```

```html
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Root Config</title>

  <!--
    Remove this if you only support browsers that support async/await.
    This is needed by babel to share largeish helper code for compiling async/await
in older
    browsers. More information at https://github.com/single-spa/create-single-
spa/issues/112
  -->
  <script src="https://cdn.jsdelivr.net/npm/regenerator-
runtime@0.13.7/runtime.min.js"></script>

  <!--
    This CSP allows any SSL-enabled host and for arbitrary eval(), but you should
limit these directives further to increase your app's security.
    Learn more about CSP policies at https://content-security-policy.com/#directive
  -->
  <meta http-equiv="Content-Security-Policy" content="default-src 'self' https:
localhost:*; script-src 'unsafe-inline' 'unsafe-eval' https: localhost:*; connect-
src https: localhost:* ws://localhost:*; style-src 'unsafe-inline' https:; object-
src 'none';">
  <meta name="importmap-type" content="systemjs-importmap" />
  <!-- If you wish to turn off import-map-overrides for specific environments
(prod), uncomment the line below -->
  <!-- More info at https://github.com/joeldenning/import-map-
overrides/blob/master/docs/configuration.md#domain-list -->
  <!-- <meta name="import-map-overrides-domains"
content="denylist:prod.example.com" /> -->

  <!-- Shared dependencies go into this import map. Your shared dependencies must
be of one of the following formats:

    1. System.register (preferred when possible) -
https://github.com/systemjs/systemjs/blob/master/docs/system-register.md
    2. UMD - https://github.com/umdjs/umd
    3. Global variable

    More information about shared dependencies can be found at https://single-
spa.js.org/docs/recommended-setup#sharing-with-import-maps.
  加载线上的CDN依赖地址
-->
  <script type="systemjs-importmap">
    {
      "imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/single-
spa@5.9.0/lib/system/single-spa.min.js",
        "jquery":"https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.min.js"
      }
    }
  </script>
  <link rel="preload" href="https://cdn.jsdelivr.net/npm/single-
spa@5.9.0/lib/system/single-spa.min.js" as="script">

  <!-- Add your organization's prod import map URL to this script's src  -->
  <!-- <script type="systemjs-importmap" src="/importmap.json"></script> -->

  <% if (isLocal) { %>
  <script type="systemjs-importmap">
    {
      "imports": {
        "@base/root-config": "//localhost:9000/base-root-config.js",
        "@app/html":"//localhost:8086/js/app.js",
        "@app/vue":"//localhost:8085/js/app.js"
      }
    }
  </script>
  <% } %>

  <!--
    If you need to support Angular applications, uncomment the script tag below to
ensure only one instance of ZoneJS is loaded
    Learn more about why at https://single-spa.js.org/docs/ecosystem-
angular/#zonejs
  -->
  <!-- <script src="https://cdn.jsdelivr.net/npm/zone.js@0.11.3/dist/zone.min.js">
</script> -->

  <script src="https://cdn.jsdelivr.net/npm/import-map-overrides@2.2.0/dist/import-
map-overrides.js"></script>
  <% if (isLocal) { %>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.js">
</script>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/extras/amd.js">
</script>
  <% } else { %>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.min.js">
</script>
  <script src="https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/extras/amd.min.js">
</script>
  <% } %>

</head>
<body>
```

```
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <script>
    //加载全局的single-spa对象
    System.import('single-spa')
    //加载全局的jquery对象
    System.import('jquery')
    System.import('@base/root-config');
  </script>
  <import-map-overrides-full show-when-local-storage="devtools" dev-libs></import-
map-overrides-full>
</body>
</html>
```

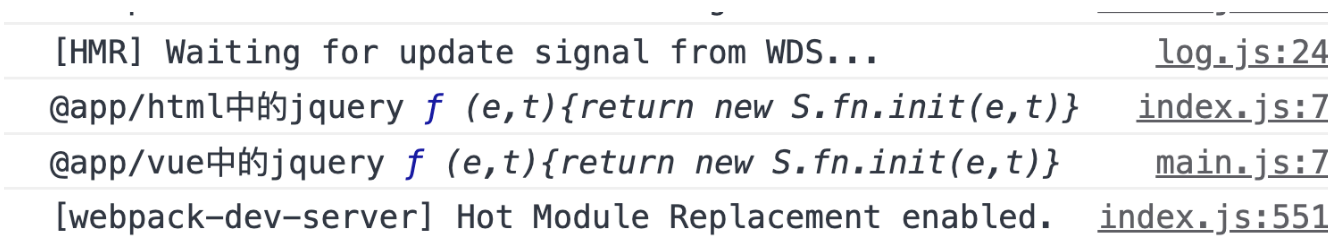
2. 在@app/html和@app/vue子应用中的webpack配置文件中通过externals属性排除jquery依赖，代码如下：

```
externals:['single-spa','jquery']//配置single-spa不从webpack依赖中解析
```

3. 分别在两个子应用的执行文件中加入jquery的ES导入方式，代码如下：

```
import $ from 'jquery'
//@app/vue的输出
console.log('@app/vue中的jquery',$)

//@app/html的输出
console.log('@app/html中的jquery',$)
```

4. 配置完成后重启子应用，并刷新基座应用，查看控制台输出，如图所示代表成功。



```
[HMR] Waiting for update signal from WDS...                    log.js:24
@app/html中的jquery ƒ (e,t){return new S.fn.init(e,t)}        index.js:7
@app/vue中的jquery ƒ (e,t){return new S.fn.init(e,t)}          main.js:7
[webpack-dev-server] Hot Module Replacement enabled.        index.js:551
```

5. 到此公共依赖处理便介绍完毕

4. QianKunJS介绍

qiankun 是一个基于 single-spa 的微前端实现库，旨在帮助大家能更简单、无痛的构建一个生产可用微前端架构系统。

QianKun的特性

- 📦 **基于 single-spa** 封装，提供了更加开箱即用的 API。
- 📱 **技术栈无关**，任意技术栈的应用均可 使用/接入，不论是 React/Vue/Angular/JQuery 还是其他等框架。
- 💪 **HTML Entry 接入方式**，让你接入微应用像使用 iframe 一样简单。
- 🛡 **样式隔离**，确保微应用之间样式互相不干扰。
- 🧳 **JS 沙箱**，确保微应用之间 全局变量/事件 不冲突。
- ⚡ **资源预加载**，在浏览器空闲时间预加载未打开的微应用资源，加速微应用打开速度。
- 🔌 **umi 插件**，提供了 @umijs/plugin-qiankun 供 umi 应用一键切换成微前端架构系统。

关于QianKun的重点

由于篇幅的关系，QianKun的介绍可以后续 参考官方文档https://qiankun.umijs.org/zh。QianKun是在single-spa的基础上进一步的简化封装而来的，目的是降低微前端架构的开发门槛，开发者可以通过QianKun提供的能力，很简单的编排微前端子应用。

QianKun的重点在于其借鉴了single-spa的优点，增加了完整的JS沙箱模式，确保子应用间的JavaScript可以做到完全隔离，在沙箱的基础上建立了完善的跨境通信机制，使得应用构建在single-spa的错综复杂上变得井井有条。当开发者对single-spa打下良好的基础后，再使用QianKun框架会使得开发者在微前端的路上走的更加顺利（本篇文章之所以以single-spa为主是因为他是微前端框架的始祖中相对成熟的，QianKun虽然封装完善但在高度定制化的需求下仍然有很多限制）