实战篇：基于single-spa的分布式微前端项目实战1

1. 微前端流行框架single-spa介绍

Single-spa 是一个将多个单页面应用聚合为一个整体应用的 JavaScript 微前端框架。 使用 single-spa 进行前端架构设计可以带来很多好处，例如:

- 在同一页面上使用多个前端框架而不用刷新页面(React, AngularJS, Angular, Ember等你正在使用的框架)
- 独立部署每一个单页面应用
- 新功能使用新框架，旧的单页应用不用重写可以共存
- 改善初始加载时间，延迟加载代码

Single-spa 从现代框架组件生命周期中获得灵感，将生命周期应用于整个应用程序。 它脱胎于 Canopy 使用 React + React-router 替换 AngularJS + ui-router 的思考，避免应用程序被束缚。现在 single-spa 几乎支持任何框架。由于 JavaScript 因其许多框架的寿命短而臭名昭著，我们决定让它在任何您想要的框架都易于使用。

Single-spa 包括以下内容:

1. Applications，每个应用程序本身就是一个完整的 SPA (某种程度上)。 每个应用程序都可以响应 url 路由事件，并且必须知道如何从 DOM 中初始化、挂载和卸载自己。 传统 SPA 应用程序和 Single SPA 应用程序的主要区别在于，它们必须能够与其他应用程序共存，而且它们没有各自的 html 页面。

   例如，React 或 Angular spa 就是应用程序。 当激活时，它们监听 url 路由事件并将内容放在 DOM上。 当它们处于非活动状态时，它们不侦听 url 路由事件，并且完全从 DOM 中删除。

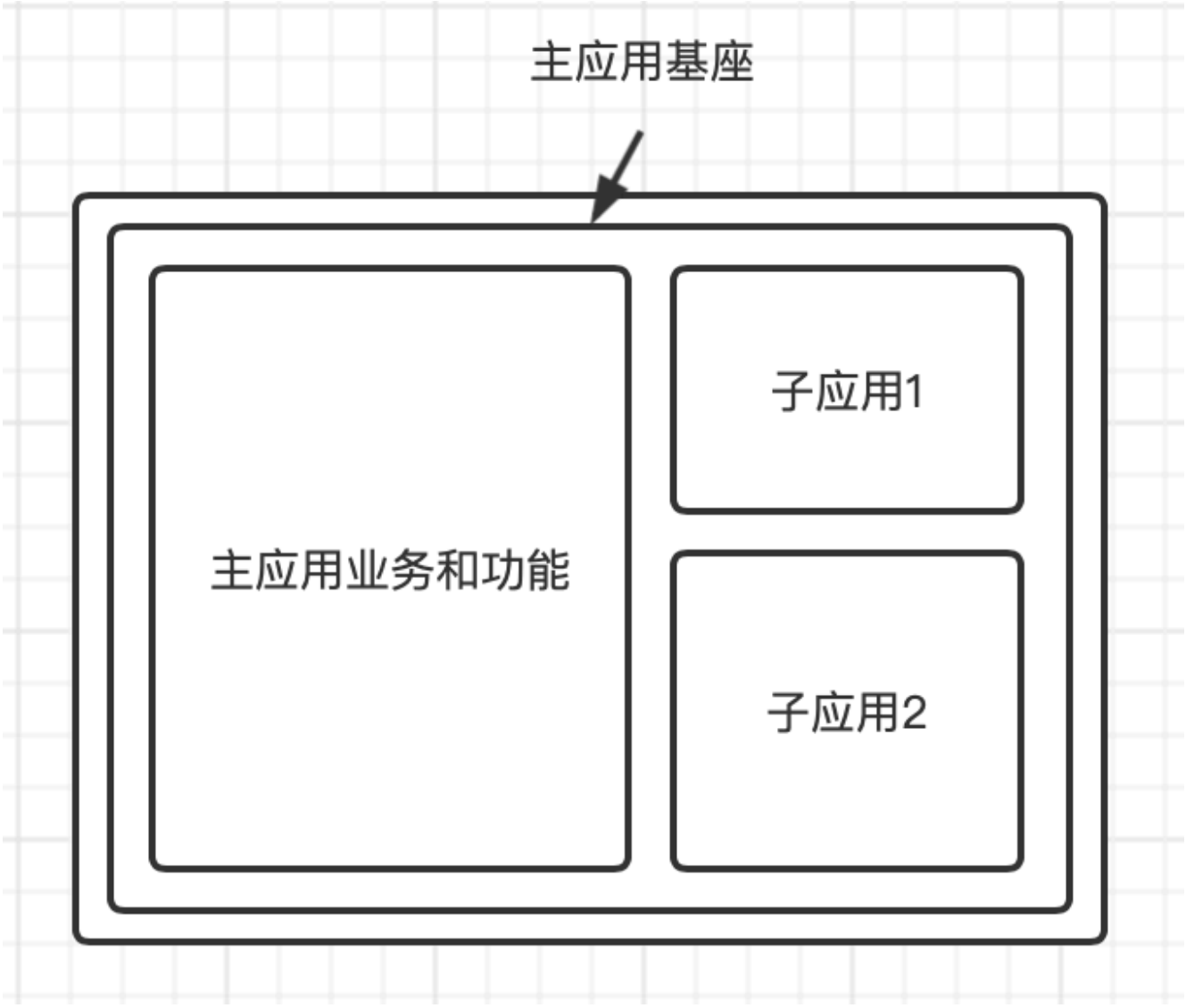
2. 一个 single-spa-config配置, 这是html页面和向Single SPA注册应用程序的JavaScript。每个应用程序都注册了三件东西
   - A name
   - A function (加载应用程序的代码)
   - A function (确定应用程序何时处于活动状态/非活动状态)

2. single-spa快速上手

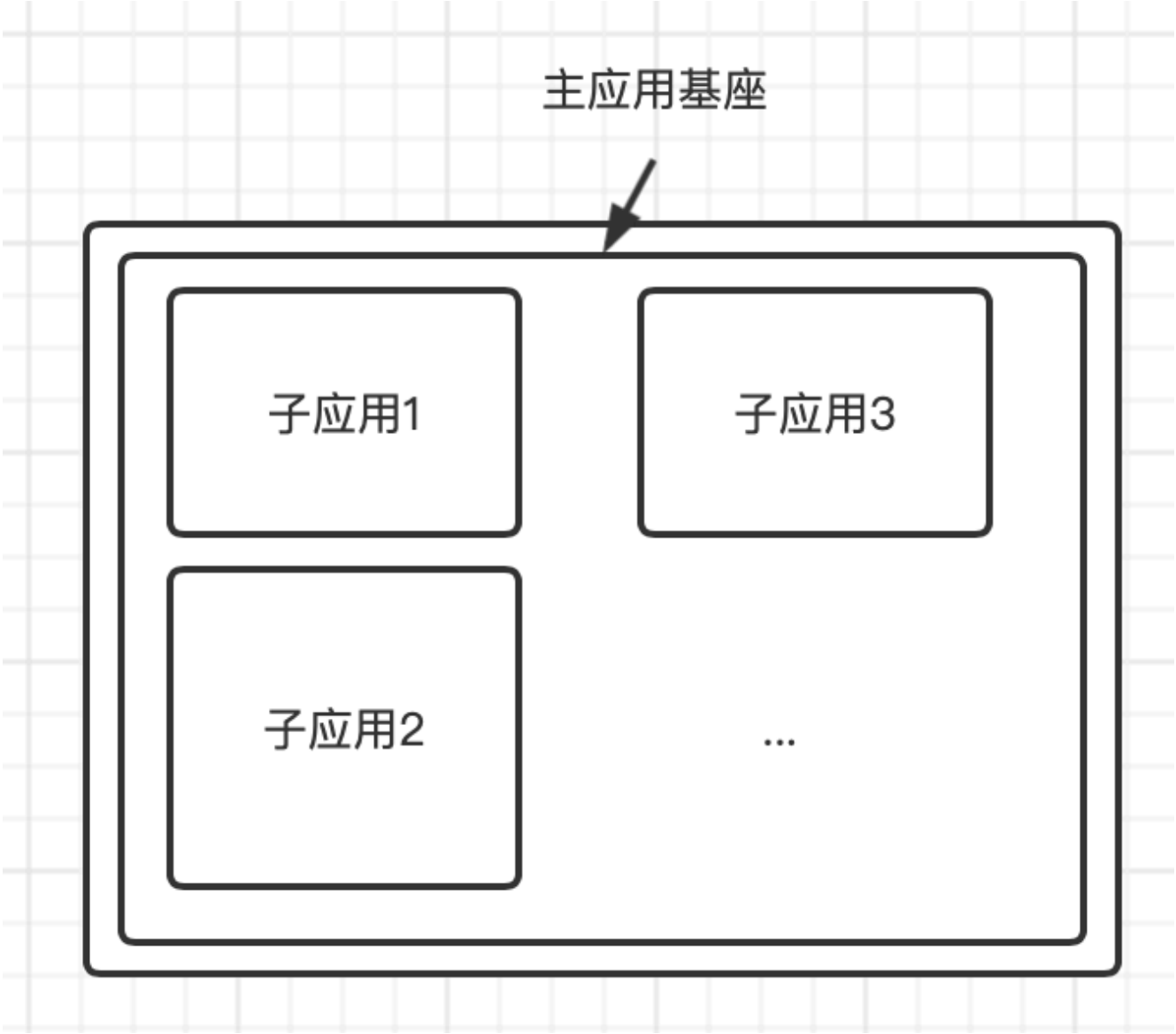
2.1 主应用和子应用介绍

在single-spa的世界开启前，需要了解主应用和子应用的概念，在上节课的学习中相信大家已经初步了解了微前端的几种架构模式，single-spa的基本组成就是一个管理应用注册的主应用和N个用于拆分的子应用。通常，主应用和子应用的组合方式有几种：

1. 包含具体业务的主应用伴随子应用模式，如图所示。



2. 纯基座式主应用伴随子应用模式，如图所示。



无论使用哪种模式，主应用都具备基座能力，用于管理子应用的注册和其生命周期。

2.2 single-spa的快速上手

想要使用single-spa构建微前端项目是非常简单的，首先确保电脑上已经安装node运行环境，这样才能保证初步构建项目结构。

快速上手的步骤如下：

1. 在代码编辑器中创建一个空项目，使用命令行工具打开空项目并输入创建基座应用指令，代码如下：

```
npm init single-spa
```

2. 控制台上会出现日志提示，此步骤需要输入当前项目创建的目录地址，默认是文件夹根目录，如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro root % npm init single-spa
? Directory for new project (.)
```

3. 由于微前端项目中可能会包含多个子应用，本次选择将基座应用命名为base，代码如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro root % npm init single-spa
? Directory for new project ./base
```

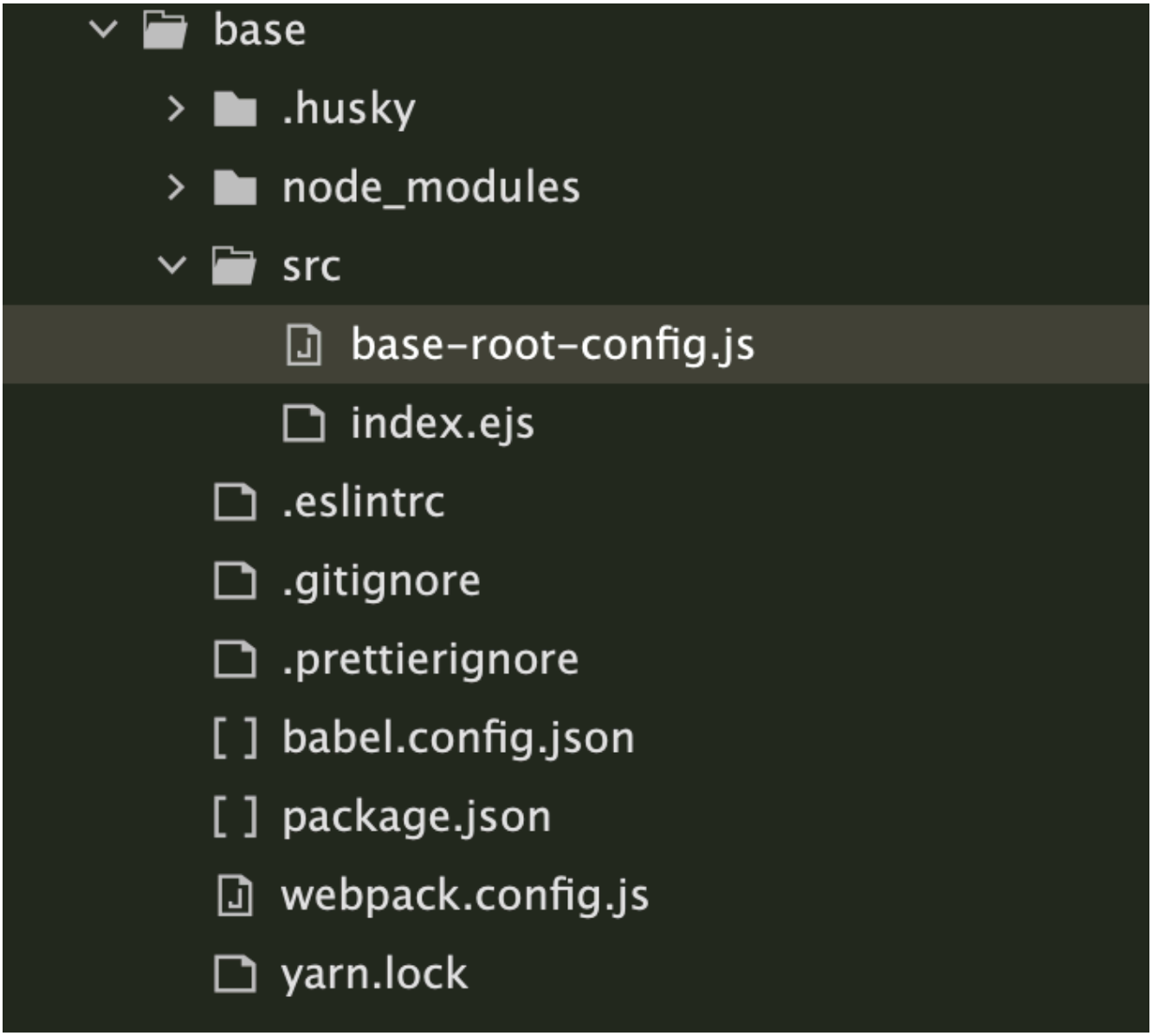
4. 控制台上会进一步提示，用于选择基座应用的模式，这里选择无业务基座模式，代码如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro root % npm init single-spa
? Directory for new project ./base
? Select type to generate
  single-spa application / parcel
  in-browser utility module (styleguide, api cache, etc)
❯ single-spa root config
```

5. 接下来会提示选择包管理器、开发语言、是否使用路由模式和配置项名称，代码如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro root % npm init single-spa
? Directory for new project ./base
? 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 No
? Organization name (can use letters, numbers, dash or underscore) base
```

6. 选择完毕后会进入自动安装依赖和创建项目阶段，创建完成后编辑器中会生成项目基本结构，如图所示。



7. 接下来进入base项目目录，在命令行工具中输入项目启动命令，代码如下：

```
npm start
```

8. 命令行会提示如下信息，代码如下：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro base % npm start

> start
> webpack serve --port 9000 --env isLocal

<i> [webpack-dev-server] Project is running at:
<i> [webpack-dev-server] Loopback: http://localhost:9000/
<i> [webpack-dev-server] On Your Network (IPv4): http://172.95.167.174:9000/
<i> [webpack-dev-server] On Your Network (IPv6): http://[fe80::1]:9000/
<i> [webpack-dev-server] Content not from webpack is served from
'/Users/zhangyunpeng/Desktop/root/base/public' directory
<i> [webpack-dev-server] 404s will fallback to '/index.html'
asset base-root-config.js 224 KiB [emitted] (name: main) 1 related asset
asset index.html 3.38 KiB [emitted]
```

9. 出现如此信息代表项目启动成功，此时访问http://localhost:9000即可打开基座应用首页，如图所示。

Welcome

to your single-spa root config! 🥳

This page is being rendered by an example single-spa application that is being imported by your root config.

Next steps

1. Add shared dependencies

- Locate the import map in `src/index.ejs`
- Add an entry for modules that will be shared across your dependencies. For example, a React application generated with create-single-spa will need to add React and ReactDOM to the import map.

```
"react": "https://cdn.jsdelivr.net/npm/react@16.13.1/umd/react.pr
"react-dom": "https://cdn.jsdelivr.net/npm/react-dom@16.13.1/umd/
```

Refer to the corresponding **single-spa framework helpers** for more specific information.

2.3 项目结构介绍

2.3.1 项目结构介绍

```
├── babel.config.json #babel配置文件
├── package.json #包描述文件
├── src #源代码文件
│   ├── base-root-config.js #基座应用的配置文件
│   └── index.ejs #视图容器的ejs模版
├── webpack.config.js #webpack的配置文件
└── yarn.lock #yarn的版本锁定文件
```

2.3.2 base-root-config.js介绍

```
//registerApplication注册子应用模块，启动项目模块
import { registerApplication, start } from "single-spa";
//注册一个名为@single-spa/welcome的子应用
registerApplication({
  name: "@single-spa/welcome", //应用名称
  //应用的导入函数，支持使用System.import()函数，本实例为线上子应用地址
  app: () =>
    System.import(
      "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js"
    ),
  //为应用绑定的路由地址，默认的single-spa为路由分发模式
  //路由匹配规则为模糊匹配，符合/**规则时该子应用均会被渲染到视图中
  activeWhen: ["/"],
});
//增加子应用示例
// registerApplication({
//   name: "@base/navbar",
//   app: () => System.import("@base/navbar"),
//   activeWhen: ["/"]
// });
//启动基座服务
start({
  urlRerouteOnly: true,
});
```

2.3.3 index.ejs介绍

```
<!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>
```

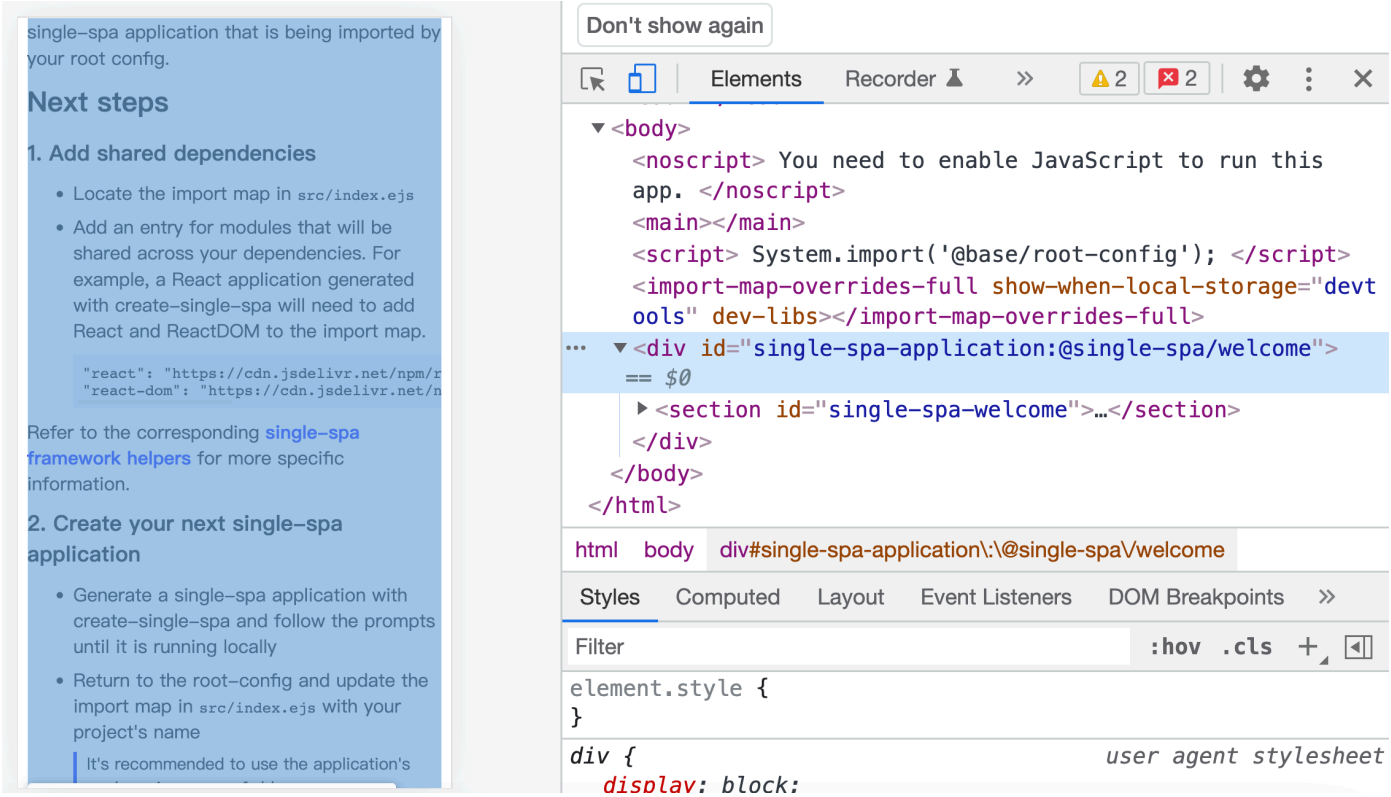
```
  <!--
    Remove this if you only support browsers that support async/await.
    This is needed by babel to share largeish helper code for compiling async/await in
older
    browsers. More information at https://github.com/single-spa/create-single-
spa/issues/112
  -->
  <script src="https://cdn.jsdelivr.net/npm/regenerator-runtime@0.13.7/runtime.min.js">
</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.
  -->
  <!-- 公共依赖模块的导入配置，single-spa推荐使用cdn模式架子啊single-spa的公共依赖 -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-
spa.min.js"
      }
    }
  </script>
  <!-- single-spa的预加载 -->
  <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> -->
  <!-- 本地运行时加载的应用模块，该部分用于注册子应用
    内部需要使用严格的JSON格式字符串使用双引号，最后一项不可以加逗号
    这个配置主要解决的事将single-spa的子应用命名管理（类似服务注册中心）
  -->
  <% if (isLocal) { %>
  <script type="systemjs-importmap">
    {
      "imports": {
        "@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>
  <!-- 加载System包用以支持System.import() -->
  <% 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>
  <main></main>
  <script>
    //该导入项用于触发base-root-config.js文件的加载
    //他会寻找imports对象中注册的@base/root-config名称所对应的js文件
    System.import('@base/root-config');
```

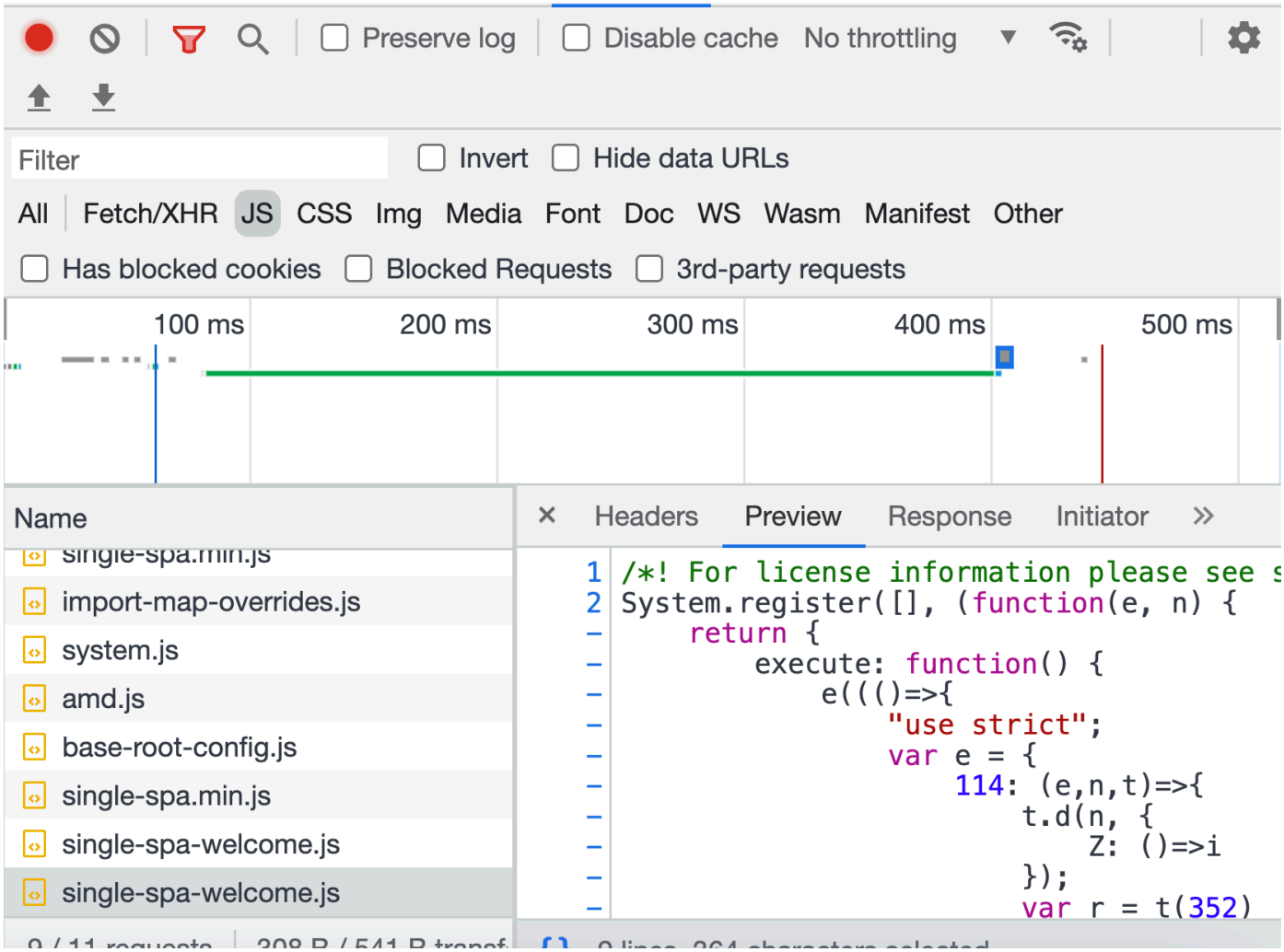
```
  </script>
  <import-map-overrides-full show-when-local-storage="devtools" dev-libs></import-map-overrides-full>
</body>
</html>
```

2.3.4 默认子应用是如何加载的

在启动项目过程中，基座应用会优先触发base-root-config.js中的代码执行，这个过程中会通过registerApplication()函数实现子应用的注册，再最后触发start()函数时，基座应用便会直接启动，启动的过程中子应用就会被加载并追加到body的最后，如图所示。



子应用的加载方式是采用动态加载JavaScript代码的方式实现应用加载的，所以并不是采用iframe或AJAX来实现的应用渲染，如图所示。



该加载方式使用了createElement('script')进行动态js加载，但是在实际检查elments查看器时却无法找到动态加载js的标签，这是因为其内部加载的规则是一旦数据加载成功便删除创建的script标签，其源码逻辑可以在控制台中检出，如图所示。

```
    var loader = this;
    return new Promise(function (resolve, reject) {
      var script = systemJSPrototype.createScript(url);
      script.addEventListener('error', function () {
        reject(Error(errMsg(3,  'Error loading ' + url + (firstPar
      });
      script.addEventListener('load', function () {
        document.head.removeChild(script);
        // Note that if an error occurs that isn't caught by this
        // that getRegister will return null and a "did not instan
        if (lastWindowErrorUrl === url) {
          reject(lastWindowError);
        }
        else {
          var register = loader.getRegister();
          // Clear any auto import registration for dynamic import
          if (register && register[0] === lastAutoImportDeps)
            clearTimeout(lastAutoImportTimeout);
          resolve(register);
        }
      });
      document.head.appendChild(script);
    });
```

3 微前端项目构建

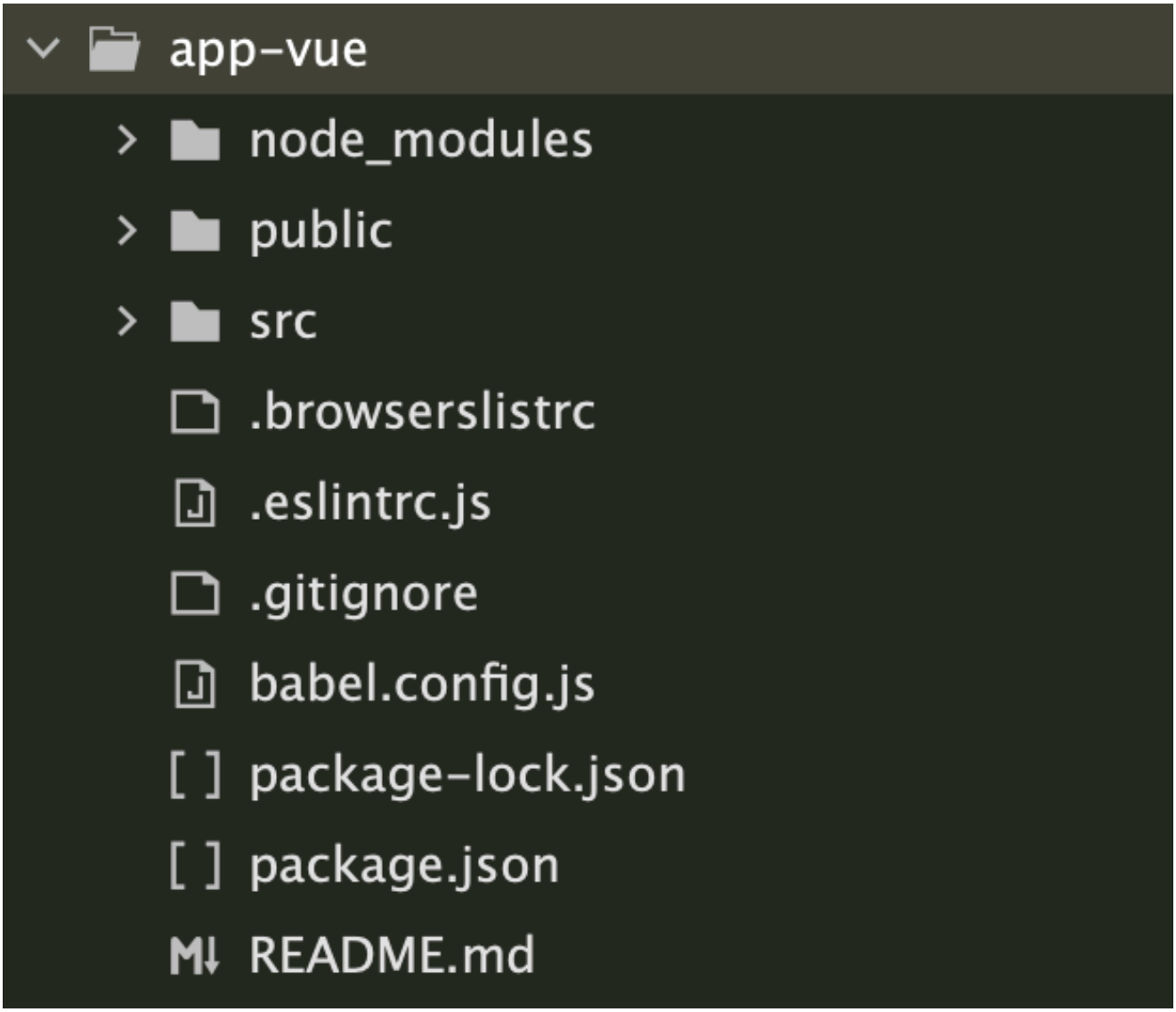
3.1 构建第一个子应用

认识主应用和主应用的基本构成后，下一步需要实现的就是一个可以与主应用结合运行的子应用。single-spa提供了大量可以无缝对接的应用框架，在不需要做大量破坏性操作的前提下，single-spa可以与几乎任何亲端框架整合形成微前端架构应用。以Vue框架为例，构建子应用的步骤如下：

1. 使用vue命令初始化vue项目，代码如下：

```
vue create app-vue
```

2. 通过VueCLI脚手架初始化一个vue3.0并且带有路由组件的项目，如图所示：



3. 创建完成后不需要启动项目，接下来在命令行工具中进入app-vue文件夹并安装single-spa-vue插件，代码如下：

```
vue add single-spa
```

4. 安装完成后，将项目中的package.json文件中的name改成@app/vue
5. 下一步集成webpack的set-public-path插件，在命令行工具中输入：

```
npm install systemjs-webpack-interop -S
```

6. 依赖安装完成后在项目的src文件夹下创建文件set-public-path.js，在其中初始化配置信息，代码如下：

```
import { setPublicPath } from 'systemjs-webpack-interop';
if(process.env.STANDALONE_SINGLE_SPA){
  setPublicPath('@app/vue');
}
```

7. 在main.js中追加set-public-path文件的引用，并阅读main.js，代码如下：

```
//引入Vue3的初始化对象
import { h, createApp } from 'vue';
//引入singleSpaVue插件
import singleSpaVue from 'single-spa-vue';
//初始化publicPath
import './set-public-path'
//引入根组件
import App from './App.vue';
//引入路由对象
import router from './router';
//得到子应用生命周期对象
const vueLifecycles = singleSpaVue({
  createApp,
  appOptions: {
    render() {
      return h(App, {
        // single-spa props are available on the "this" object. Forward them to
your component as needed.
        // https://single-spa.js.org/docs/building-applications#lifecycle-props
        // if you uncomment these, remember to add matching prop definitions for
them in your App.vue file.
        /*
        name: this.name,
        mountParcel: this.mountParcel,
        singleSpa: this.singleSpa,
        */
      });
    },
  },
  //实例化成功后触发的函数
  handleInstance(app) {
    app.use(router);
  },
});
//导出主应用加载子应用时所需要的函数对象
export const bootstrap = vueLifecycles.bootstrap;
export const mount = vueLifecycles.mount;
export const unmount = vueLifecycles.unmount;
```

8. 改造src/router/index.js文件，将路由的根路径修改为/app，代码如下：

```
const router = createRouter({
  history: createWebHistory('/app'),
  routes
})
```

9. 完成后在控制台输入单独启动（作为独立应用启动）命令，代码如下：

```
npm run serve:standalone
```

10. 访问http://localhost:8080/app若展示正确的欢迎页面，代表独立子应用已运行成功。

3.2 将子应用和主应用整合

目前子应用已经可以独立运行，独立运行的目的是脱离主应用单独调试。由于微前端项目为保证应用管理，最终需要将子应用的渲染结果加载到主应用的运行容器中，所以子应用在运行时若脱离宿主环境会无法单独运行。

接下来在主应用中注册并加载子应用，实现微前端的第一个应用整合，步骤如下：

1. 在主应用的index.ejs文件中，将子应用的应用名称和访问路径关联，代码如下：

```
<script type="systemjs-importmap">
    {
      "imports": {
        "@base/root-config": "//localhost:9000/base-root-config.js",
        //将@app/vue作为导入名称方面在应用开发时使用名称导入依赖
        "@app/vue":"//localhost:8080/js/app.js"
      }
    }
  </script>
```

2. 在base-root-config.js文件中注册子应用，代码如下：

```
import { registerApplication, start } from "single-spa";

registerApplication({
  name: "@single-spa/welcome",
  app: () =>
    System.import(
      "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js"
    ),
  //actionWhen是函数时可以通过location.pathname来自定义访问规则
  //默认使用"/"代表“/**”规则全部符合
  activeWhen:["/"]
});
//注册子应用
registerApplication({
  //配置应用名称
  name: "@app/vue",
  //远程导入应用
  app: () => System.import("@app/vue"),
  //当访问/app/**时激活子应用
  activeWhen: ["/app"],
});

// registerApplication({
//   name: "@base/navbar",
```

```
//   app: () => System.import("@base/navbar"),
//   activeWhen: ["/"]
// });

start({
  urlRerouteOnly: true,
});
```

3. 将子应用停止，使用子应用模式启动，代码如下：

```
npm run serve
```

4. 在主应用的命令行工具中启动主应用，代码如下：

```
npm start
```

5. 在浏览器中访问主应用地址http://localhost:9000/app并观察网页运行结果，会发现可能仍然展示的是默认远程子应用的欢迎页面。若发现这种情况可以将网页向下拉，就会看见其实子应用已经正确加载。若正上方显示的是vue项目，则向下拉还可以看见默认子应用在同屏展示，如图所示。

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

Welcome

to your single-spa root config! 🥳

This page is being rendered by an example single-spa application that is being imported by your root config.

Next steps

1. Add shared dependencies

- Locate the import map in `src/index.ejs`
- Add an entry for modules that will be shared across your dependencies. For example, a React application generated with create-single-spa will need to add React and ReactDOM to the

registerApplication中的activeWhen属性除了可以设置为数组外，还可以使用函数的形式进行配置。数组设置的方式可以支持同一个子应用匹配N个路由，该属性在使用时是模糊匹配的，若activeWhen:['/app']，则路由符合/app/**的情况时，应用都会被加载到视图中。函数要求返回一个boolean类型的数据，当数据为true的时候路由才能正确加载。函数的参数是location对象，可以通过pathname来识别url路径是否匹配，详细内容可以参考官方文档。下面演示一下activeWhen的改造方式。

3.3 加载子应用的过程

首先，single-spa并不是基于完整沙箱概念实现的，所以其JavaScript和CSS样式都并非在沙箱内执行的，基座应用的window和document对象也会暴漏给子应用。

一个简单的例子

CSS样式逃逸可以参考浏览器的查看器，会发现Vue项目中的样式最终会被挂在到基座应用的head标签中，至于window和document对象可以简单的做一个实验：

1. 在主应用的index.ejs文件中script标签部分增加如下内容：

```
<script>
  window.abc = '全局的window对象'
  System.import('@base/root-config');
</script>
```

2. 在Vue子应用中的main.js中增加如下输出：

```
console.log(document.body)
console.log(window.abc)
```

3. 访问主应用的http://localhost:9000/app查看控制台，会发现控制台中能直接输出abc的内容，并且document的body指向的是基座应用的body对象，这就意味着single-spa的子应用并不是完全沙箱模式。

子应用的加载

single-spa加载子应用的过程为：

1. 下载子应用
2. 执行子应用的bootstrap并传入应用信息
3. 执行子应用的mount并传入应用信息

可以在Vue项目的main.js中改造代码，代码如下：

```
import { h, createApp } from 'vue';
import singleSpaVue from 'single-spa-vue';
import './set-public-path'
import App from './App.vue';
import router from './router';
console.log(document.body)
console.log(window.abc)
const vueLifecycles = singleSpaVue({
  createApp,
```

```
  appOptions: {
    render() {
      return h(App, {
        // single-spa props are available on the "this" object. Forward them to your
component as needed.
        // https://single-spa.js.org/docs/building-applications#lifecycle-props
        // if you uncomment these, remember to add matching prop definitions for them
in your App.vue file.
        /*
        name: this.name,
        mountParcel: this.mountParcel,
        singleSpa: this.singleSpa,
        */
      });
    },
  },
  handleInstance(app) {
    app.use(router);
  },
});
console.log('下载vue应用')
export const bootstrap = function(props){
  console.log(props)
  console.log('vue应用执行bootstrap')
  return vueLifecycles.bootstrap(props);
}

export const mount = function(props){
  console.log('vue应用执行mount')
  return vueLifecycles.mount(props);
}
export const unmount = function(){
  console.log('vue应用执行unmount')
  return vueLifecycles.unmount()
}
```

改造子应用后，访问基座程序加载子应用时会打印子应用的加载顺序，如图所示。

```
下载vue应用                                    main.js?56d7:28
                                               main.js?56d7:30
▶ {name: '@app/vue', singleSpa: {…}, mountParcel: f}
vue应用执行bootstrap                           main.js?56d7:31
vue应用执行mount                               main.js?56d7:36
```

在加载过程中，主应用可以为子应用传入自定义属性，通过registerApplication()函数进行自定义属性的传递。改造主应用的base-root-config文件中的子应用注册部分，加入自定义属性，代码如下：

```
registerApplication({
  //配置应用名称
  name: "@app/vue",
  //远程导入应用
  app: () => System.import("@app/vue"),
  //当访问/app/**时激活子应用
  activeWhen: ["/app"],
  //传入自定义属性
  customProps:{
    token:'abcdefg'
  }
});
```

在基座应用中访问控制台，会发现子应用生命周期执行时可以直接通过props对象获取自定义属性，如图所示。

4. 多个子应用跳转

多个子应用跳转需要在当前系统中创建一个新的子应用，本次依然创建一个与@app/vue项目相同的子应用，创建过程参考上方的应用创建步骤，创建app-vue1项目，接下来在@app/vue1项目中改造路由配置，步骤如下：

1. 在新的项目中去掉App.vue中的router-link标签
2. 在新项目中将路由配置改造成如下内容：

```
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'

const routes = [
  {
    path: '/vue1/',
    name: 'Home1',
    component: Home
  },
  {
    path: '/vue1/about',
    name: 'About1',
```

```
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this route
    // which is lazy-loaded when the route is visited.
    component: () => import(/* webpackChunkName: "about" */ '../views/About.vue')
  }
]

const router = createRouter({
  history: createWebHistory('/app'),
  routes
})

export default router
```

3. 在index.ejs中的imports中添加@app/vue1，然后在基座应用中注册新创建的子项目，代码如下：

```
registerApplication({
  //配置应用名称
  name: "@app/vue1",
  //远程导入应用
  app: () => System.import("@app/vue1"),
  //当访问/app/**时激活子应用
  activeWhen: ["/app"],
  //传入自定义属性
  customProps:{
    token:'abcdefg'
  }
});
```

4. 在@app/vue项目中将App.vue文件中的内容改造为如下：

```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link> |
    <router-link to="/vue1/">vue1的首页</router-link> |
    <router-link to="/vue1/about">vue1的关于</router-link>
  </div>
  <router-view/>
</template>
```

5. 访问主应用http://localhost:9000/app，通过菜单跳转会发现两个应用可以实现在同一个视图中切换页面。

5. 基于路由和微件化的区别

通过本节的学习简单的认识了一下基于single-spa的微前端构建方式，已经掌握了微前端项目中子应用和主应用的通信策略和子应用的工作流程。在这个过程中会发现，本节内容所演示的微前端加载方式是完全按照路由匹配的方式动态加载应用的，这个过程中虽然可以通过模糊匹配的概念在同一段URL路径中加载多个子应用，但是加载的子应用无法自定义布局，导致样式很难维护，所以本节所介绍的内容就是基于路由分发实现的微前端处理。

微件化是指同一个界面通过应用名称动态加载子应用，无需匹配URL路径即可实现子应用的布局和加载。