微前端介绍

作为前端的开发者，相信大家在耳边经常会听到微前端的概念。大家可能到之后会感觉到一脸懵逼，听着好像是很高大上的技术，对于初级的前端开发者来说这种技术在实际的业务中可能用不上，或者是根本就没有用这种技术方案的必要，所以，不知道这种技术方案其实也正常。

起源

微前端的出现是为了解决大型复杂应用程序的维护和开发难题。传统的单体式前端架构虽然简单易用，但在开发和维护大型应用程序时会产生臃肿和难以维护的问题，若单体应用程序越大，这些问题就会暴露的很明显。正因为如此，微前端才渐渐的出现在人们的视线中。

什么是微前端

官方的描述：微前端是一种多个团队通过独立发布功能的方式来共同构建现代化 web 应用的技术手段及方法策略。其实，简单点来说就是：微前端是一种前端架构设计模式，其思想是将一个大型的前端应用拆分成多个独立的小型子应用，每个子应用都可以独立开发、构建、测试和部署，并且可以被组合成一个完整的前端应用。

微前端是一种设计架构，并不是技术。是借鉴于微服务思想的设计架构，是一种为了解决庞大且难以维护的项目的方案。

微前端解决了什么问题

1、大型应用程序的维护困难：传统的前端应用程序通常是由一个团队设计、开发、维护。在应用规模越来越大的时候，维护开销的成本也就会越高，并且可能会引起依赖管理和资源冲突等问题。

2、大型应用程序的可扩展性问题：当应用程序越来越大，其扩展性会变得困难，这可能会导致性能的下降以及代码质量的下降等问题。

3、多团队协同开发问题：在传统的前端应用程序中，由于团队之间的代码复用和协同开发是困难的，因此这也可能导致冲突和代码混乱等问题。

微前端具备的核心价值

1、技术栈无关

主框架不限制接入应用的技术栈，微应用具备完全自主权。

2、独立开发、独立部署

微应用仓库独立、前后端可独立开发、部署完成后主框架自动完成同步更新。

3、增量升级

在面对各种复杂场景时，通常很难对一个已经存在的庞大的系统做全量的技术栈升级或重构，而微前端是一种非常好的实施渐进式重构的手段和策略。

4、独立运行时

每个微应用之间状态隔离，运行时状态不共享

微前端的优点

1、更好的代码可维护性：微前端拆分应用程序成为小型可重用的程序，降低了应用程序的复杂性，提高了代码的可维护性。

2、降低了开发的成本以及复杂性：不同的团队可以使用不同的技术栈来开发各应用程序的不同部分，从而降低了复杂性、提高了效率。

3、更好的可重用性和可扩展性：微前端使得不同的应用程序更加独立并重用，可以在开发周期中独立升级不同的应用程序，并且主框架中自动完成更新。

4、更高的可靠性和安全性：微前端使得各个应用程序在运行和部署的过程中模块之间的解耦，减少了模块之间的互相冲突和影响，提高程序的可靠性和安全性。

微前端的缺点

1、技术复杂度更高：涉及到跨团队协作和跨技术栈的模块集成，需要涉及到框架、通讯等技术问题。

2、存在技术风险：需要开发额外的网络层和路由层来解决跨域问题。

3、项目总体运行效率可能会变慢：微前端需要多个应用程序之间进行通信，需要时间成本和通信成本，这可能会对主程序效率造成一定的损失。

微前端解决跨域的方案

Proxy

通过配置Web服务器反向代理，将多个前端应用程序转发到同一个主机的端口上。需要Nginx、Apache等服务器和一个负载均衡模块，允许所有的请求通过同一个域名和端口号对外访问。

iframe

每个微前端的应用程序会以iframe形式嵌套在页面中，采用iframe的方式就避免了跨域问题。这种情况下，主应用程序负责加载嵌套的子应用程序，并且控制子应用程序的样式和事件通信。

实现过程

1、在主应用容器（如基座）中，准备好一个可以容纳不同微应用的容器（如 div）。

2、在主应用中创建一个 iframe 标签。

```
1 <div id="app"></div>
2 <script>
```

```
  const appEl = document.querySelector('#app')
  const microApp = document.createElement('iframe')
  microApp.setAttribute('src', 'http://localhost:8080')
  microApp.setAttribute('frameborder', '0')
  microApp.style.width = '100%'
  appEl.appendChild(microApp)
</script>
```

3、子应用中默认运行的端口为 8080。iframe 中的 src 属性指向子应用的对应页面。

4、如果 iframe 标签的源和主应用的源不同，则子应用需要向主应用发送消息，以允许主应用将数据发送回来。可以通过window.parent.postMessage()来实现。

```
window.parent.postMessage({data: 'message from micro app'}, '*')
```

5、在主应用中，我们监听 message 事件而不是 load 事件，以便在子应用向主应用发送一条 “解锁” 消息时，我们知道在哪个时间点上允许访问来自子应用的数据。

```
window.addEventListener('message', event => {
  if (event.origin !== 'http://localhost:8000') {
    return
  }
  const data = event.data
  console.log(data)
})
```

综上所述，我们可以通过iframe标签来解决主应用与子应用的跨域问题，主应用程序通过iframe标签嵌入子应用，并且使用postMessage来允许不同iframe元素之间进行通信。

CORS

允许来自其他域或子域的异步http请求。当子应用使用Ajax请求数据时，主应用程序必须设置Access-Control-Allow-Origin头，以便子应用程序可以使用ajax请求并接收数据。需要服务端进行设置。

```
  devServer: {
    headers: {
      "Access-Control-Allow-Origin": "*",
    },
  },
```

微前端架构方案

自由组织模式

不同的团队可以自由地开发和部署独立的微前端应用程序，并通过协调机制，将这些应用程序组合在一起形成完整的应用系统。适合团队功能分离比较明显、团队规模比较大的大型企业级应用开发。

基座模式

将多个子应用程序作为模块加载到一个主应用程序中。这些模块是独立的小型应用程序。每个子应用程序都可以独立开发、测试、部署，而主应用程序主要就是将子应用进行集成和协调，子应用程序发生变化，主应用程序也会自动的完成更新。

基座模式需要使用一些基础设施来支持子应用程序加载、路由和通信。这些基础设施可以是自定义，也可以使用现有的框架来实现。比如：

Single-SPA：一个支持多框架、多技术栈的JavaScript微前端框架，用于构建大型单页应用程序。

qiankun：一个基于Single-SPA封装的微前端框架，支持React、Vue、Angular等技术栈。

去中心化模式

是在多个子应用程序之间创建对等的关系，每个应用程序对应整个应用程序系统来说都是平等的。也就是说每一个应用程序都可以作为容器或者是子应用程序，这种模式可以确保多个应用程序之间的一致性和数据同步性。例如：webpack5中的模块联邦就现实了去中心化的模式，它允许不同的团队和应用程序独立开发和部署自己的代码，并且将其组合在一起以创建复杂的应用程序。

适用于：团队规模不大、领域和业务比较统一的较为分散的应用系统开发。

Single-SPA

在single-spa框架中有三种类型的微前端应用：

1、single-spa-application/parcel：微前端架构中的子应用程序，可以使用Vue、React、Angular框架，可以利用根应用提供的共享工具和服务进行通信。

2、single-spa root config：创建微前端容器应用，根应用就是主应用，负责加载其他子应用，并作为单页应用（SPA）的容器。将不同的子应用集成在一个页面中，并为每个子应用创建一个独立的上下文。

3、utillty modules：公共模块应用，非渲染组件，可以在不同应用之间共享JavaScript模块和组件。

创建主应用

`npx create-single-spa`

```
1 ? Directory for new project spa-test
2 ? Select type to generate single-spa root config
3 ? Which package manager do you want to use? pnpm
```

```
? Will this project use Typescript? Yes
? Would you like to use single-spa Layout Engine No
? Organization name (can use letters, numbers, dash or underscore) test

```

创建子应用

react项目

`npx create-single-spa`

```
? Directory for new project todos
? Select type to generate single-spa application / parcel
? Which framework do you want to use? react
? Which package manager do you want to use? pnpm
? Will this project use Typescript? Yes
? Organization name (can use letters, numbers, dash or underscore) test
? Project name (can use letters, numbers, dash or underscore) todos
```

创建完会有对应的提示

```
Project setup complete!
Steps to test your React single-spa application:
1. Run 'pnpm start -- --port 8500'
2. Go to http://single-spa-playground.org/playground/instant-test?name=@test/todos&url=8500 to see it working!
```

创建完成之后，使用 `pnpm start` 启动应用，并访问 http://localhost:8080/，可以看到提示

Your Microfrontend is not here

The @test/todos microfrontend is running in "integrated" mode, since standalone-single-spa-webpack-plugin is disabled. This means that it does not work as a standalone application without changing configuration.

How do I develop this microfrontend?

To develop this microfrontend, try the following steps:

1. Copy the following URL to your clipboard: http://localhost:8500/test-todos.js
2. In a new browser tab, go to the your single-spa web app. This is where your "root config" is running. You do not have to run the root config locally if it is already running on a deployed environment - go to the deployed environment directly.
3. In the browser console, run `localStorage.setItem('devtools', true);` Refresh the page.
4. A yellowish rectangle should appear at the bottom right of your screen. Click on it. Find the name @test/todos and click on it. If it is not present, click on Add New Module.
5. Paste the URL above into the input that appears. Refresh the page.
6. Congrats, your local code is now being used!

For further information about "integrated" mode, see the following links:

- Local Development Overview
- Import Map Overrides Documentation

If you prefer Standalone mode

To run this microfrontend in "standalone" mode, the standalone-single-spa-webpack-plugin must not be disabled. In some cases, this is done by running `npm run start:standalone`. Alternatively, you can add `--env standalone` to your package.json start script if you are using webpack-config-single-spa.

If neither of those work for you, see more details about enabling standalone mode at Standalone Plugin Documentation.

启动会报你的微应用不在这里的错误，此时需要到去主应用中对子应用进行注册。

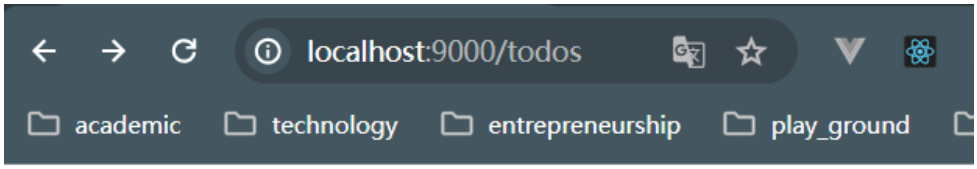
```
// 主应用中的study-root-config.ts文件
registerApplication({
  name: "@test/todos",
  app: () =>
    System.import<LifeCycles>(
      "@test/todos"
    ),
  activeWhen: ["/todos"],
});

```

```
  <-- 主应用中的index.ejs文件 -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js",
        "react": "https://unpkg.com/react@17/umd/react.production.min.js",
        "react-dom": "https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"
      }
    }
  </script>

  <script type="systemjs-importmap">
    {
      "imports": {
        "@test/root-config": "//localhost:9000/test-root-config.js",
        "@test/todos": "//localhost:8080/test-todos.js"
      }
    }
  </script>
```

启动主应用，在地址栏上访问http://localhost:9000/todos即可。

当访问react-demo的时候，我们希望就展示react-demo微应用就可以了，此时，我们可以将根应用的访问触发进行精准的匹配，如下：

```
registerApplication(
  "@single-spa/welcome",
  () =>
    System.import<LifeCycles>(
      "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js"
    ),
  (location) => location.pathname === "/"
);
```



@test/todos is mounted!

此时就可以根据不同的url展示对应的子应用了。 此时默认会将子应用加载到main标签中，你也可以将子应用放在指定的dom节点上面，通过以下配置即可：

```
<body>
  <h1 id="react-todos"></h1>
</body>
```

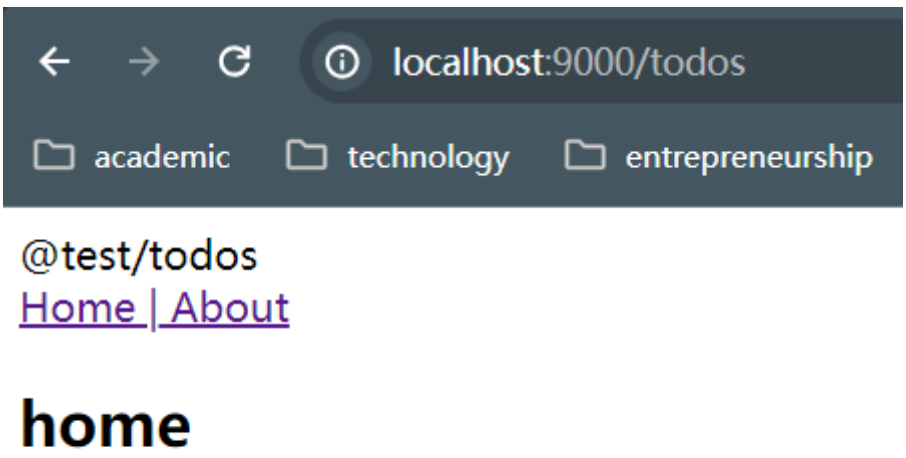
```
// 子应用的test-todos文件
const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: Root,
  errorBoundary(err, info, props) {
    // Customize the root error boundary for your microfrontend here.
    return null;
  },
  // 插入到指定的dom节点
  domElementGetter: () => document.getElementById("react-todos"),
});
```

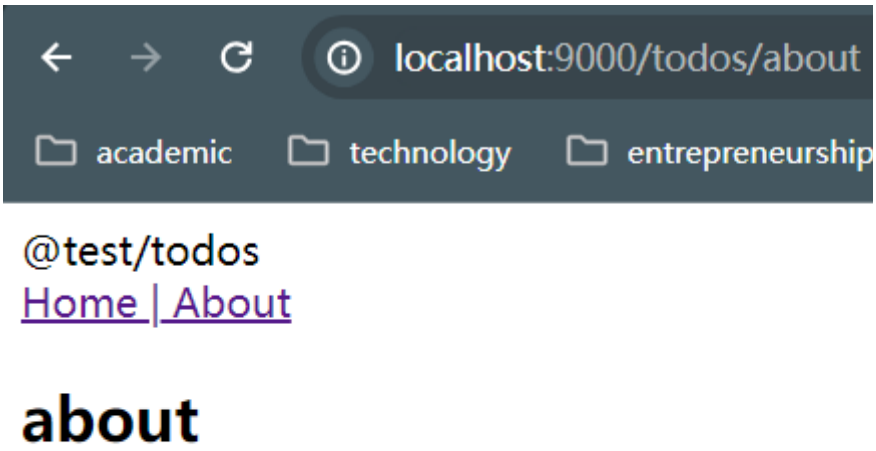
同样的，我们可以使用react-router-dom根据路由加载对应的组件，相关代码如下：

```tsx
// 子应用的 root.component.tsx
import React from "react";
import { BrowserRouter, Route, Link, useRoutes } from "react-router-dom";
// 在src目录创建两个组件
import Home from "./home";
import About from "./about";

// 创建路由
const routes = [
  {
    path: "/",
    element: (<Home></Home>)
  },
  {
    path: "/home",
    element: (<Home></Home>)
  },
  {
    path: "/about",
    element: (<About ></About >)
  },
];
function RouterView() {
  const elem = useRoutes(routes)
  return elem
}
export default function Root(props) {
  // return <section>{props.name} is mounted!</section>;

  return (
    <BrowserRouter basename="/todos">
      <div>{props.name}</div>
      <div>
        <Link to="/home">Home |</Link>
        <Link to="/about"> About</Link>
      </div>
      <RouterView />
    </BrowserRouter>
  );
}

```

```jsx
import React, { Component } from "react";
export class home extends Component {
  render() {
    return (
      <div>
        <h2>home</h2>
      </div>
    );
  }
}

export default home;
```

```jsx
import React from "react";

export default function about() {
  return (
    <div>
      <h2>about</h2>
    </div>
  );
}

```





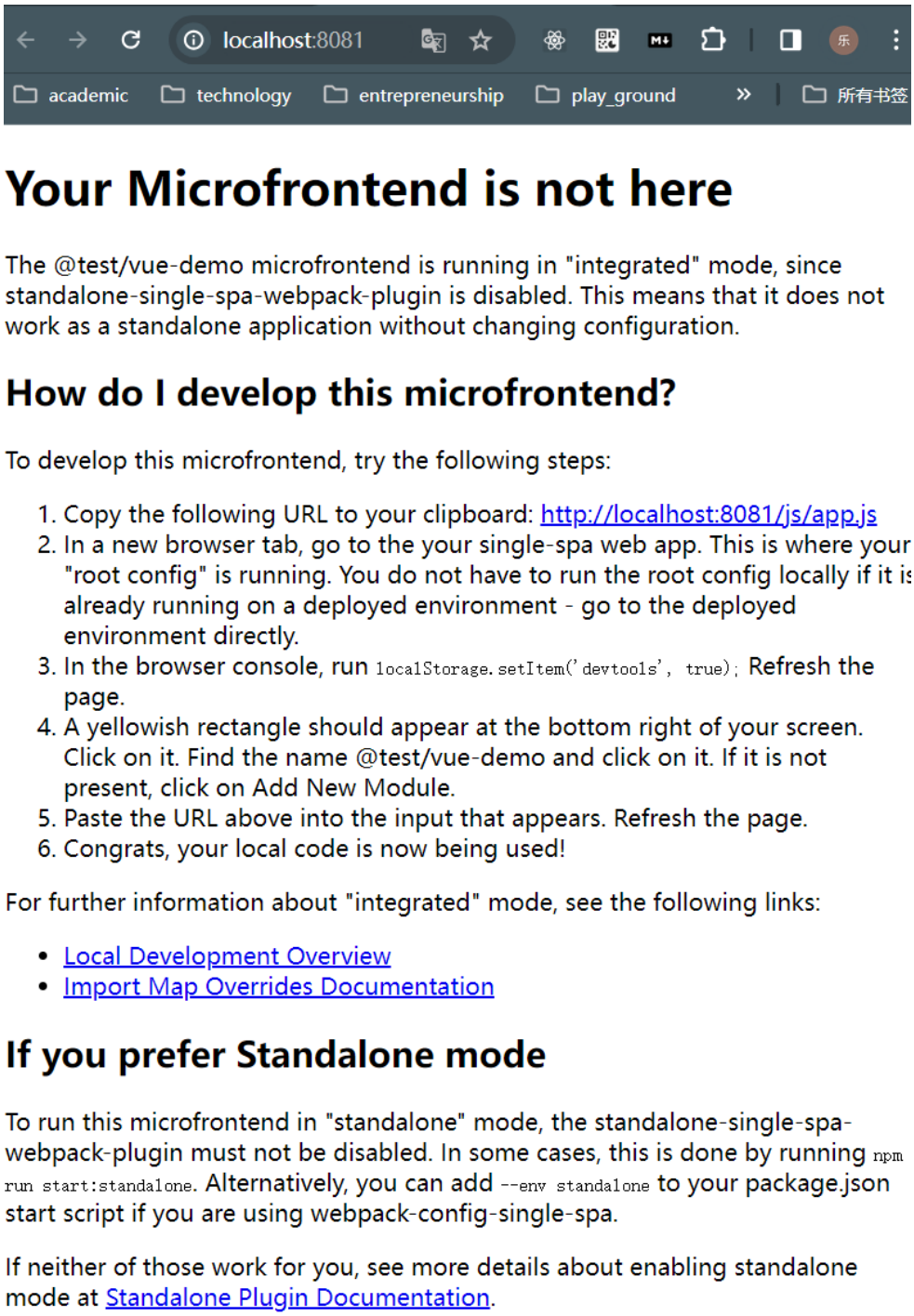
vue项目

```
$ npx create-single-spa
? Directory for new project vue-demo
? Select type to generate single-spa application / parcel
? Which framework do you want to use? vue
? Organization name (can use letters, numbers, dash or underscore) test
```

需要排除公共的依赖包。

```
const { defineConfig } = require("@vue/cli-service");
module.exports = defineConfig({
  transpileDependencies: true,
  configureWebpack: {
    output: {
      // 注意这里一定要写system 否则会报错
      libraryTarget: "system",
    },
    externals: ["vue", "vue-router"],
  },
});

```

同样的，启动npm run serve也会报错 Your Microfrontend is not here



这里的配置跟前面的配置一致，先注册子应用，然后再去主应用中将公共的依赖包引入，并且指定子应用的地址。

```
registerApplication({
  name: "@test/vue-demo",
  app: () => System.import<LifeCycles>("@test/vue-demo"),
  activeWhen: ["/vue-demo"],
});
```

```
<!-- 引入公共依赖 -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.min.js",
        "react": "https://unpkg.com/react@17/umd/react.production.min.js",
        "react-dom": "https://unpkg.com/react-dom@17/umd/react-dom.production.min.js",
        "vue": "https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js",
        "vue-router": "https://cdn.jsdelivr.net/npm/vue-router@3.0.7/dist/vue-router.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">

<!-- 应用地址 -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "@test/root-config": "//localhost:9000/test-root-config.js",
        "@test/todos": "//localhost:8080/test-todos.js",
        "@test/vue-demo": "//localhost:8081/js/app.js"
      }
    }
  </script>


```

启动运行主应用，url改为子应用对应的路由。此时发现会报错

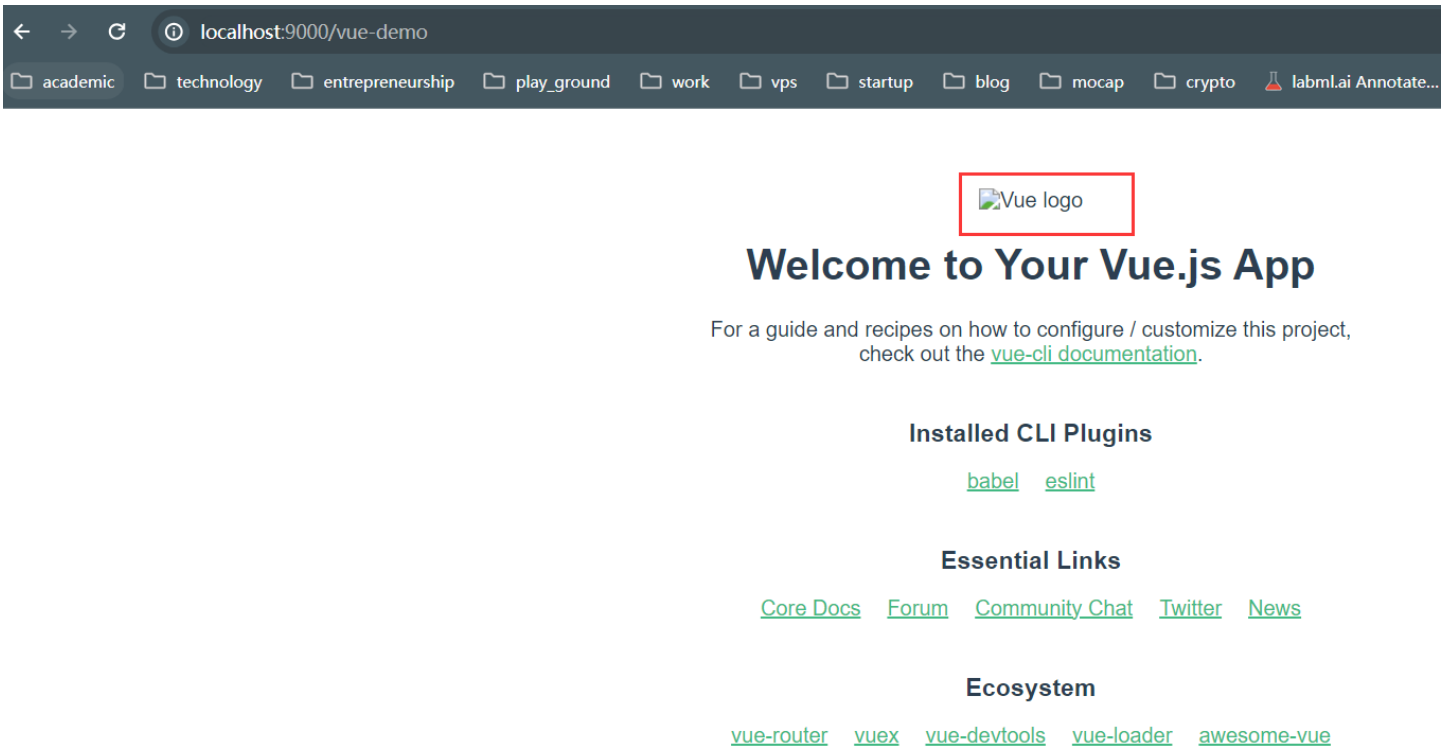
```
Uncaught runtime errors:
ERROR
application '@test/vue-demo' died in status LOADING_SOURCE_CODE: Cannot read properties of undefined (reading 'meta')
TypeError: application '@test/vue-demo' died in status LOADING_SOURCE_CODE: Cannot read properties of undefined (reading 'meta')
    at autoPublicPath (http://localhost:8081/js/app.js:5449:32)
    at ./node_modules/.pnpm/systemjs-webpack-interop@2.3.7_webpack@5.89.0/node_modules/systemjs-webpack-interop/auto-public-
```

```
path/2.js (http://localhost:8081/js/app.js:5432:1)
    at __webpack_require__ (http://localhost:8081/js/app.js:21038:33)
    at http://localhost:8081/js/app.js:22097:11
    at Object.<anonymous> (http://localhost:8081/js/app.js:22101:12)
    at Object.execute
(https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/extras/amd.js:56:35)
    at doExec
(https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.js:469:34)
    at postOrderExec
(https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.js:465:12)
    at https://cdn.jsdelivr.net/npm/systemjs@6.8.3/dist/system.js:422:14
```

这里只需要将vue.config.js文件中的output：libraryTarget改为system就可以解决了。

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,
  configureWebpack: {
    output: {
      libraryTarget: "system"
    },
    externals: ["vue", "vue-router"]
  }
})

```

但是此时的子应用的图片加载不出来。



因为受到CSP限制，所以静态资源加载不出来。Content Security Policy（CSP）策略问题： CSP是一种安全策略，机制会防止获取资源。如果您已经启用了CSP头，那可能您需要检查您的策略中是否确实允许加载来自该址的图片。

在主应用中的`<meta http-equiv="Content-Security-Policy"></meta> content中加上img-src 'self' data:`就可以解决了。

```
<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'; img-src 'self' data:;" />
```

完成上面注册子应用之后，就可以继续使用子应用注册路由了，

```
import Vue from "vue";
import singleSpaVue from "single-spa-vue";
import VueRouter from "vue-router";
```

```
import App from "./App.vue";
import "./public-path.js";
Vue.config.productionTip = false;
Vue.use(VueRouter);

const About = { template: "<h1>About</h1>" };
const Home = { template: "<h1>Home</h1>" };
const routes = [
  {
    path: "/about",
    component: About,
  },
  {
    path: "/home",
    component: Home,
  },
];
const router = new VueRouter({
  routes,
  mode: "history",
  base: "vue-demo",
});
const vueLifecycles = singleSpaVue({
  Vue,
  appOptions: {
    // 注册路由
    router,
    render(h) {
      return h(App, {
        props: {
          // 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,
          */
        },
      });
    },
  },
});

```

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

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

    <router-view />
    <!-- <img alt="Vue logo" src="./assets/logo.png" />
    <HelloWorld msg="Welcome to Your Vue.js App" /> -->
  </div>
</template>

<script>
// import HelloWorld from "./components/HelloWorld.vue";

export default {
  name: "App",
  components: {
    // HelloWorld,
  },
};
</script>
```

完成以上操作之后就可以实现路由的正常切换了。

跨应用通信

```
$ npx create-single-spa
? Directory for new project utils
? Select type to generate in-browser utility
module (styleguide, api cache, etc)
? Which framework do you want to use? none
? Which package manager do you want to use? pnpm
? Will this project use Typescript? Yes
? Organization name (can use letters, numbers,
```

```
dash or underscore) test
? Project name (can use letters, numbers, dash or
underscore) utils
```

需要在主应用中写入应用地址：（注意：此时是不需要进行注册的）

```
<!-- 应用地址 -->
  <script type="systemjs-importmap">
    {
      "imports": {
        "@test/root-config": "//localhost:9000/test-root-config.js",
        "@test/todos": "//localhost:8080/test-todos.js",
        "@test/vue-demo": "//localhost:8081/js/app.js",
        "@test/utils": "//localhost:8082/study-utils.js"
      }
    }
  </script>
```

工具库中的study-utils.js文件

```
// Anything exported from this file is importable by other in-browser modules.
export function publicApiFunction(test) {
    console.log(test)
    return test
}

```

在react项目中使用

在react项目中使用utils中定义的方法，在公共的方法封装成一个hooks，创建一个hooks文件夹。

```

```

```
import React, { useEffect, useState } from "react";
function useUtilsModule() {
  const [utilsModule, setUtilModule] = useState();
  useEffect(() => {
    // 导入
    System.import("@test/utils").then(setUtilModule);
  }, []);
  return utilsModule;
}
export default useUtilsModule;
```

在组件中使用

```
import useUtilsModule from './hooks';
export default function about() {
  const utilsModule = useUtilsModule();
  let result = "";
  if (utilsModule) {
    result = utilsModule.publicApiFunction("react");
  }
  return (
    <div>
      <div>about---{result}</div>
    </div>
  );
}
```

在vue项目中使用

```
<template>
  <div>
    About {{ msg }}

    <button @click="getUtils">跨应用通信</button>
  </div>
</template>

```

```vue
<script>
  export default {
    data() {
      return {
        msg: "",
      };
    },
    methods: {
      async getUtils() {
        // 异步获取
        const utilsModules = await window.System.import("@test/utils");
        this.msg = utilsModules.publicApiFunction("vue --about");
      },
    },
  };
</script>

<style lang="scss" scoped></style>
```

以上就完成了微前端中Single-SPA的基本使用。

qiankun

https://qiankun.umijs.org/zh/guide

qiankun是一个基于Single-SPA的微前端解决方案，它可以帮我们将多个独立的前端应用整合到一个整体，并实现这些应用的共享和协同。

特性

1、基于single-spa封装，提供了更加开箱即用的API。

2、与技术栈无关，任意技术栈的应用均可使用/接入，不论是react、vue、angular还是其他框架。

3、html entry接入方式，让你接入微应用像使用iframe一样简单。

4、样式隔离，确保微应用之间样式互相不干扰。

5、js沙箱，确保微应用之间全局变量/事件不冲突。

6、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，加快微应用打开速度。

实战

创建react三个项目，一个是base、一个micro-app1、一个micro-app2。 应用安装：`npm install qiankun`

子应用中安装：`npm install react-app-rewired -D` 并且改package.json文件

```
"scripts": {
  "start": "react-app-rewired start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},

```

分别在各自的 src 目录新增 public-path.js：（用来处理子应用在主应用中静态资源加载不出来的问题）

```
if (window.__POWERED_BY_QIANKUN__) {
  __webpack_public_path__ = window.__INJECTED_PUBLIC_PATH_BY_QIANKUN__;
}
```

分别在子应用的index.js中添加qiankun的生命周期

```
function render(props) {
  const { container } = props;

  ReactDOM.render(
    <App />,
    container
    ? container.querySelector("#root")
    : document.querySelector("#root")
  );
}

if (!window.__POWERED_BY_QIANKUN__) {
```

```
  render({});
}

export async function bootstrap() {
  console.log("[react16] react app bootstraped");
}

export async function mount(props) {
  console.log("[react16] props from main framework", props);
  render(props);
}

export async function unmount(props) {
  const { container } = props;
  ReactDOM.unmountComponentAtNode(
    container
    ? container.querySelector("#root")
    : document.querySelector("#root")
  );
}
```

然后在主应用中index.js中进行注册即可导入微应用。

```
registerMicroApps([
  {
    // 组织名称
    name: "reactApp",
    // 入口
    entry: "//localhost:3011",
    // 挂载点
    container: "#micro-app1",
    // 访问对应的路由 触发
    activeRule: "/micro-app1",
  },
  {
    name: "reactApp1",
    entry: "//localhost:3012",
    container: "#micro-app2",
    activeRule: "/micro-app2",
  },
]);
```

启动主应用访问对应的路由地址即可访问到微应用了。

主应用与子应用之间进行通信

在主应用中进行注册的时候可以传递props参数

```
registerMicroApps([
  {
    name: "reactApp",
    entry: "//localhost:3011",
    container: "#micro-app1",
    activeRule: "/micro-app1",
    props: {
      name: "青峰1",
    },
  },
  {
    name: "vueApp",
    entry: "//localhost:3012",
    container: "#micro-app2",
    activeRule: "/micro-app2",
    props: {
      name: "青峰2",
    },
  },
]);
```

在子应用index.js文件中的mount生命周期中获取props参数

```
export async function mount(props) {
  console.log("[react16] props from main framework", props);
  console.log(props);
  render(props);
}

```

也可以通过initGlobalState(state)进行通信

```
// 子应用入口文件
export async function mount(props) {
  console.log("[react16] props from main framework", props);
  // 监听主应用传递的数据
  props.onGlobalStateChange((state, prev) => {
    // state: 变更后的状态; prev 变更前的状态
    console.log(state, prev);
  });

  // 向主应用传递数据
  // props.setGlobalState(state);
  render(props);
}
```

```
// 主应用index.js
import { initGlobalState } from "qiankun";
const state = {
  name: "青峰",
};
// 初始化 state
const actions = initGlobalState(state);

actions.onGlobalStateChange((state, prev) => {
  // state: 变更后的状态; prev 变更前的状态
  console.log(state, prev);
});
setTimeout(() => {
  // 向子应用传递数据
  actions.setGlobalState({ ...state, age: 18 });
```

```
}, 2000);
actions.offGlobalStateChange();
```

vue项目进行通信：

```
// vue.config.js文件
const { defineConfig } = require("@vue/cli-service");
const { name } = require("./package");

module.exports = defineConfig({
  transpileDependencies: true,
  devServer: {
    headers: {
      "Access-Control-Allow-Origin": "*",
    },
  },
  configureWebpack: {
    output: {
      library: `${name}-[name]`,
      libraryTarget: "umd", // 把微应用打包成 umd 库格式
      // jsonpFunction: `webpackJsonp_${name}`,
    },
  },
});
```

修改main.js文件

```
import { createApp } from "vue";
import App from "./App.vue";


let instance = null;
function render(props = {}) {
  const { container } = props;


  instance = createApp(App).mount(
    container ? container.querySelector("#app") : "#app"
  );
}


// 独立运行时
```

```
if (!window.__POWERED_BY_QIANKUN__) {
  render();
}


export async function bootstrap() {
  console.log("[vue] vue app bootstraped");
}
export async function mount(props) {
  console.log("[vue] props from main framework", props);
  render(props);
}
export async function unmount() {
  instance.$destroy();
  instance.$el.innerHTML = "";
  instance = null;
}
```

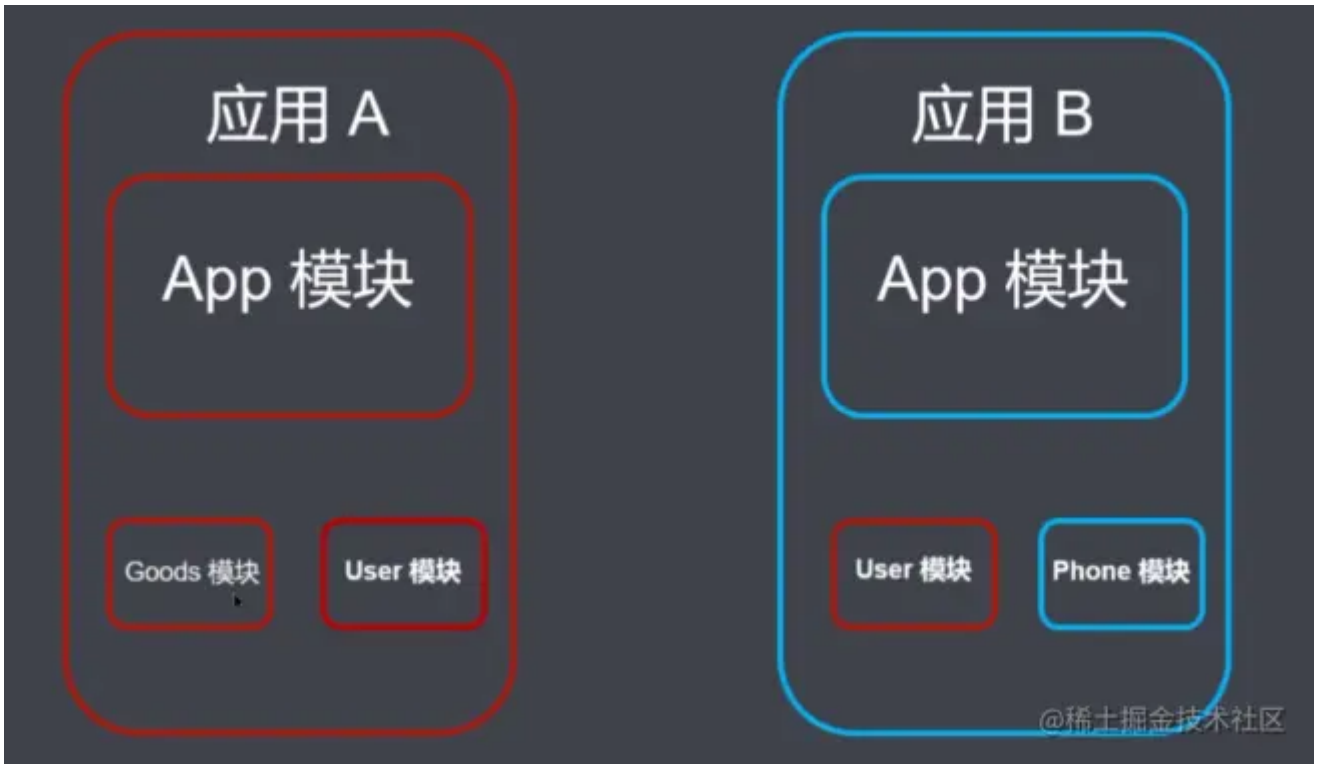
同样也是需要在主应用中进行注册。

```
registerMicroApps([
  {
    name: "vueApp",
    entry: "//localhost:3013",
    container: "#micro-vue",
    activeRule: "/micro-vue",
    props: {
      qfname: "青峰3",
    },
  },
]);
```

与react中一样，也是在mount生命周期中进行监听数据传递就可以了。

webpack5模块联邦

模块联邦其实就是去中心化模式，它没有容器、主应用的概念，任何的应用都可以导出和导入，所以，每一个应用都可以当作为是一个主应用来使用。



React项目

先搭建两个react项目分别是root和user。

```
// user中的webpack.config.js文件
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const Mfp = require("webpack").container.ModuleFederationPlugin;
module.exports = {
...
  plugins: [
    new HtmlWebpackPlugin({
      template: "./src/index.html",
    }),

    new Mfp({
      // 对外提供打包后的文件名，打包出去的包的名称
      filename: "myuser.js",
      // 导出的应用名称 ；类似single-spa组织的名字
      name: "study",
      // 导出的文件 精细到每个文件
      exposes: {
        // 具体到哪个文件
```

```
        "./userexposes": "./src/User.js",
      },
      remotes: {
        // 给导入的文件命名：组织名称@地址/导出的包名称
        root: "study@http://localhost:3001/myroot.js",
      },
    }),
  ],
...
};
```

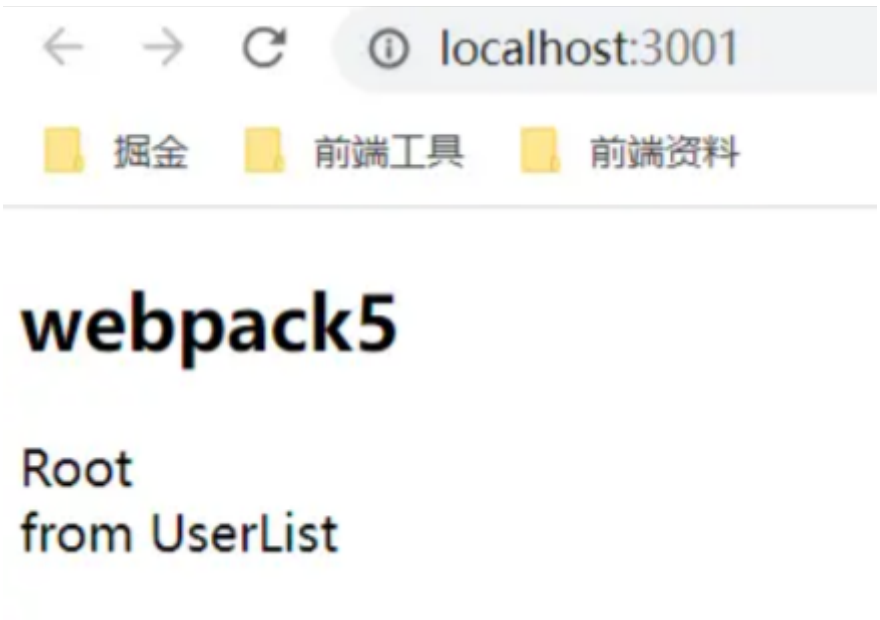
在root应用进行导入

```
// root中的webpack.config.js文件
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const Mfp = require("webpack").container.ModuleFederationPlugin;
module.exports = {
...
  plugins: [
    new HtmlWebpackPlugin({
      template: "./src/index.html",
    }),

    new Mfp({
      filename: "myroot.js",
      remotes: {
        // 给导入的文件命名：组织名称@地址/导出的包名称
        user: "study@http://localhost:3002/myuser.js",
      },
      name: "study",
      exposes: {
        "./rootexposes": "./src/Root.js",
      },
    }),
  ],
  ....
};
```

在root中的App.js文件使用user应用传递过来的组件

```
import React from "react";
import User from "./User";
// 异步加载 import 导入的文件命名/导出的具体组件
const Us = React.lazy(() => import("user/userexposes"));
export default function App() {
  return (
    <div>
      <h2>webpack5</h2>
      <User />
      <React.Suspense fallback="loading...">
        <Us />
      </React.Suspense>
    </div>
  );
}
```

root应用的端口号为3001，user应用的端口号为3002。



以上就完成了react项目中的模块联邦了。

Vue项目

在vue项目中的wepack.config.js中也是同样的配置，这里就不一一展开了，只是在使用中会有如下的差别：

```
import {defineAsyncComponent, createApp} from 'vue';
import App from './App'
const app = createApp(App)
// 导入包的命名/具体导出的组件
const Content = defineAsyncComponent(()=> import('home/Content'))
const Button = defineAsyncComponent(()=> import('home/Button'))
app.component(Content)
app.component(Button)
app.mount("#app")

```

总结

微前端是指将前端应用程序拆分成更小的独立部分，然后将其组合成一个整体应用程序。这种架构方式可以使不同团队独立开发、部署和维护各自的功能模块，从而提高应用程序的可维护性、灵活性和可扩展性。

未来，微前端将成为前端开发的趋势，因为它可以满足快速迭代的需求，使各团队独立开发，从而提高产品质量和研发效率。除此之外，微前端还提供了更好的技术栈灵活性，即团队可以选择最适合应用场景的技术栈，避免了一棵树上开花的问题。

微前端未来的发展趋势将会是更加成熟化和标准化。目前还有一些痛点，例如多语言支持、子应用间通信、路由同步等问题需要解决。因此，微前端框架和相关工具的发展将会更加完善，提供更好的解决方案来解决这些问题。同时，由于微前端的高度灵活性，未来很有可能出现类似于微服务的组织模式，即更多的团队会提供独立的子应用来实现一些业务需求。