

中台项目-微前端qiankun

学习视频

基础： [黑马前端基于qiankun搭建微前端项目实战教程_哔哩哔哩_bilibili](#)

路由、部署配置注意： [qiankun+vite微前端上线注意事项，base公共路径设置_哔哩哔哩_bilibili](#)

微前端

什么是微前端？

微前端是将前端应用分解成一系列更小、更易管理的独立部分的架构方案。类似于微服务，每个微应用可以由不同团队独立开发、测试、部署。

微前端的好处？

1. 技术栈无关

主应用和子应用可以使用不同的技术栈

允许渐进式技术栈升级

降低全局技术升级的风险

2. 独立开发部署

各团队可以独立开发、测试、部署

有哪些微前端方案？

下面介绍有哪些主流的微前端方案。

1. qiankun（蚂蚁金服，后续也采用这种）

优点：

- 基于 single-spa 封装

- 完善的沙箱机制
- 开箱即用的 API
- 中文社区活跃

适用场景：

- 大型中台系统
- 需要多团队协作的项目

2. single-spa

优点：

- 最早的微前端框架
- 灵活性高
- 社区成熟

缺点：

- 配置较复杂
- 需要自己实现样式隔离

3. Module Federation（来自Webpack 5，中文：模块联邦，可以用来做远程组件）

优点：

- Webpack 原生支持
- 真正的运行时模块共享
- 构建时优化

适用场景：

- 新项目
- 需要精细化控制模块共享

4. micro-app（京东）

优点：

- 使用简单
- 基于 Web Components
- 性能好

适用场景：

- 对性能要求高的项目
- 喜欢简单配置的团队

5. Iframe

优点：

- 具有天然的隔离属性，js沙箱、样式隔离等都很好。

缺点：

- UI不同步，比如在iframe中添加蒙层弹框，只会在iframe中显示，不是全屏的。
- 慢，每次进入会重新加载，多个iframe时浏览器容易卡死。

基础改造



基座改造

一、Antd Pro基座改造 (umi系)

接下来对中台(antd Pro 创建的项目，作为基座)进行改造。

1、安装@umijs/plugin-qiankun

【umimax已内置】

```
1 pnpm i @umijs/plugin-qiankun
```

2、注册子应用

config/config.ts配置：

在config的**qiankun.master.apps**数组中 注册子应用

```
1 export default defineConfig({  
2   qiankun: {  
3     master: {
```

```

4      apps: [
5        {
6          name: 'sub-umi', //子应用的名称
7          entry: '///localhost:5175', //子应用的入口地址
8          activeRule: '/qiankun/umi', //子应用的激活规则, 指路由
9          sandbox: {
10             strictStyleIsolation: true, //严格样式隔离
11           },
12         },
13       ],
14     },
15   },
16   //其他配置
17 })

```

3、配置访问子应用的路由

config/router.tsx：在基座的路由中添加能访问子应用的路由。

方式1：用**microApp**属性指定要渲染的子应用的name。（本次改造采用这种）

```

1    {
2      path: '/qiankun',
3      name: 'qiankun',
4      routes: [
5        {
6          path: '/qiankun/umi',
7          name: 'sub-umi',
8          microApp: 'sub-umi', //和注册时的name一致
9          microAppProps: {
10             // 子应用自动设置loading
11             // autoSetLoading: true, //可以用autoSetLoading, 需要子应用引入antd
12             loader: (loading: boolean) => <Spin spinning={loading} />,
13           },
14         },
15       ],
16     },

```

方式2：使用component属性指定组件，在组件中使用qiankun提供的 MicoApp组件

```

1  // 1、路由

```

```

2    {
3      path: '/app1',
4      name: 'sub-app',
5      element:<SubApp/>
6    }
7
8    // 2、 SubApp组件, 用MicroApp组件占位, 需要指定name(和注册时间同名)
9    import React from 'react';
10   import { MicroApp } from '@umijs/max';
11   type Props = {};
12
13   const MicroApp1 = (props: Props) => {
14     return <MicroApp name="sub-app" />;
15   };
16
17   export default MicroApp1;
18
19

```

二、非umi系基座改造

1、安装qiankun

```
1  npm i qiankun // 或者 yarn add qiankun
```

2、修改入口文件

在src/index.tsx中增加如下代码：从qiankun中引入注册和启动的函数，注册子应用并调用start启动。

```

1
2  import { start, registerMicroApps } from 'qiankun';
3
4  // 1. 要加载的子应用列表
5  const apps = [
6    {

```

```

7     name: "sub-react", // 子应用的名称
8     entry: '///localhost:8080', // 默认会加载这个路径下的html, 解析里面的js
9     activeRule: "/sub-react", // 匹配的路由
10    container: "#sub-app" // 加载的容器
11  },
12 ]
13
14 // 2. 注册子应用
15 registerMicroApps(apps, { //下面的配置对象可以不写
16   beforeLoad: [async app => console.log('before load', app.name)],
17   beforeMount: [async app => console.log('before mount', app.name)],
18   afterMount: [async app => console.log('after mount', app.name)],
19 })
20
21 start() // 3. 启动微服务
22 /*
23 // 配置qiankun启动参数
24 start({
25   // prefetch: true, // 预加载 默认是true, 即在主应用加载的时候, 加载子应用
26   sandbox: {
27     //沙箱
28     // experimentalStyleIsolation: true, // 实验性样式隔离, 好像没用哦
29     strictStyleIsolation: true, // 严格样式隔离
30   },
31 })
32 */

```

3、注册子应用路由, 提供容器dom

router注册一下子应用的路由, element设置为null, 在跳转到子应用的路由时, 展示id为subApp的div。

```

1 //router/index.jsx
2 const router = [
3   {
4     path: "/",
5     element: <Home />,
6   },
7   {
8     path: "/app1/*",
9     element: null,
10  },
11 ]

```

```

12
13 //App.jsx
14 function App() {
15     const element = useRoutes(router)
16
17     return (
18         <div className="main-layout">
19             <nav>
20                 <Link to="/">首页</Link>
21                 <Link to="/app1">子应用1</Link>
22             </nav>
23
24             <div className="test-aa">123</div>
25             <div className="main-content">
26                 <Suspense fallback={<div>Loading...</div>}>
27                     {element}
28                     {/* 需要子应用的容器 */}
29                     <div id="subApp"></div>
30                 </Suspense>
31             </div>
32         </div>
33     )
34 }

```

一旦浏览器的 url 发生变化，便会自动触发 qiankun 的匹配逻辑。所有 activeRule 规则匹配上的微应用就会被插入到指定的 container 中，同时依次调用微应用暴露出的生命周期钩子。

- registerMicroApps(apps, lifeCycles?)

注册所有子应用，qiankun会根据activeRule去匹配对应的子应用并加载

- start(options?)

启动 qiankun，可以进行预加载和沙箱设置，**更多options:**

<https://qiankun.umijs.org/zh/api#startopts>

至此基座基本就改造完成，接下来改造子应用。

子应用改造 🤖

子应用改造主要需要注意：

1. 用umd格式打包，当然，像umi，vite这些插件已经设置了。

一、umi子应用改造

中台项目的子应用是采用umi进行构建。

1. 安装插件

在子应用目录安装@umijs/plugins插件，才能在umirc中用qiankun字段。

```
1 pnpm i @umijs/plugins
```

2. 使用插件

在.umirc.ts 中使用上面的插件，这样就在基座中通过子应用的地址来访问这个子应用了。

```
1
2 export default defineConfig({
3   base: '/', // 用qiankun插件后默认base为包名，所以这里重置一下
4   qiankun: { //告诉umi这个项目需要用到qiankun
5     slave: {},
6   },
7   plugins: ['@umijs/plugins/dist/qiankun', '@umijs/plugins/dist/model',
8     '@umijs/plugins/dist/mf'], //plugins使用@umijs/plugins插件，功能分别是支持
    qiankun、允许useModel、模块联邦mf
  })
```

3. 生命周期

如果需要在生命周期中做一些事情，可以在入口文件app.tsx中导出qiankun对象，在对象中的方法写代码，qiankun会执行这些生命周期函数。

```
1
2 export const qiankun = {
3   async mount(props: any) {
4     console.log(props)
5   },
6   async bootstrap() {
7     console.log('umi app bootstrapped');
```



```
8    },
9    async afterMount(props: any) {
10      console.log('umi app afterMount', props);
11    },
12  };
```

二、vue3+vite改造

创建子应用

创建子应用，选择vue3+vite

```
1  npm create vite@latest
```

改造子应用

1. 安装 `vite-plugin-qiankun` 包，因为qiankun和vite有些问题，需要这个包解决。

```
1  pnpm i vite-plugin-qiankun
```

2. 修改vite.config.js，使用上面的插件

```
1  import qiankun from 'vite-plugin-qiankun';
2
3  defineConfig({
4    base: '/sub-vue', // 和基座中配置的activeRule一致
5    server: {
6      port: 3002,
7      cors: true,
8      origin: 'http://localhost:3002'
9    },
10   plugins: [
11     vue(),
12     qiankun('sub-vue', { // 配置qiankun插件
13       useDevMode: true
```

```
14     })
15   ]
16 })
```

3. 修改main.ts

我们需要提供三个必须的生命周期函数，即：bootstrap（只在第一次进入的时候执行）、mount（挂载）、onmount（卸载）

```
1  import { createApp } from 'vue'
2  import './style.css'
3  import App from './App.vue'
4  import { renderWithQiankun, qiankunWindow } from 'vite-plugin-qiankun/dist/helper';
5
6  let app: any;
7  if (!qiankunWindow.__POWERED_BY_QIANKUN__) {
8    createApp(App).mount('#app');
9  } else {
10    renderWithQiankun({
11      // 子应用挂载
12      mount(props) {
13        app = createApp(App);
14        app.mount(props.container.querySelector('#app'));
15      },
16      // 只有子应用第一次加载会触发
17      bootstrap() {
18        console.log('vue app bootstrap');
19      },
20      // 更新
21      update() {
22        console.log('vue app update');
23      },
24      // 卸载
25      unmount() {
26        console.log('vue app unmount');
27        app?.unmount();
28      }
29    });
30  }
```

三、create-react-app改造

1、改造入口文件

代码如下

- 导出三个必须的生命周期函数，供qiankun使用。
- 根据window.__POWERED_BY_QIANKUN__来决定render逻辑

```
1  let root: Root
2
3  // 将render方法用函数包裹，供后续主应用与独立运行调用
4  function render(props: any) {
5    const { container } = props
6    const dom = container ? container.querySelector('#root') :
    document.getElementById('root')
7    root = createRoot(dom)
8    root.render(
9      // 可以根据需要指定basename
10     //<BrowserRouter basename={window.__POWERED_BY_QIANKUN__ ? "/sub-react" :
    ""}>
11     <BrowserRouter>
12       <App/>
13     </BrowserRouter>
14   )
15 }
16
17 // 判断是否在qiankun环境下，非qiankun环境下独立运行
18 if (!(window as any).__POWERED_BY_QIANKUN__) {
19   render({});
20 }
21
22 // 各个生命周期
23 // bootstrap 只有在微应用初始化的时候调用一次，下次微应用重新进入时会直接调用 mount 钩
    子，不会再重复触发 bootstrap。
24 export async function bootstrap() {
25   console.log('react app bootstrapped');
26 }
27
28 // 应用每次进入都会调用 mount 方法，通常我们在这里触发应用的渲染方法
29 export async function mount(props: any) {
30   render(props);
31 }
```

```
32
33 // 应用每次 切出/卸载 会调用的方法，通常在这里我们会卸载微应用的应用实例
34 export async function unmount(props: any) {
35   root.unmount();
36 }
```

2、新增public-path.js

动态设置 webpack publicPath，防止资源加载出错

```
1  if (window.__POWERED_BY_QIANKUN__) {
2    // 动态设置 webpack publicPath，防止资源加载出错
3    // eslint-disable-next-line no-undef
4    webpack_public_path = window.__INJECTED_PUBLIC_PATH_BY_QIANKUN__
5  }
```

3、修改webpack配置文件，用umd格式打包

这里用craco修改webpack的配置，所以应该先下载craco。在根目录下新增craco.config.js文件并新增如下配置

```
1  const { name } = require("../package");
2
3  module.exports = {
4    webpack: (webpackConfig) => {
5      webpackConfig.output = {
6        ...webpackConfig.output,
7        library: `${packageName}-${name}`,
8        libraryTarget: "umd", //主要是这个配置，用umd打包这个项目
9        chunkLoadingGlobal: `webpackJsonp_${packageName}`,
10       filename: "static/js/[name].umd.js",
11       chunkFilename: "static/js/[name].umd.chunk.js",
12     }
13     return webpackConfig
14   }
15   };
```

什么是umd?

 <https://blog.csdn.net/badbaby52906/article/details/144950661?spm=1001.2014.3001.5501>

umd格式-CSDN博客

文章浏览阅读705次，点赞6次，收藏5次。介绍umd的用途和配置

在基座能访问子应用，即说明配置成功。

说明

一、样式隔离

使用

子应用之间的样式隔离qiankun已经实现了，但是基座和子应用之间的样式隔离没有实现。

我们可以在基座注册子应用时 设置 **strictStyleIsolation为true**，这样设置主要是对直接设置class时进行隔离（className='test'），在项目中我们一般是 **css module 和strictStyleIsolation 一起使用**，子应用能加自己的前缀是更好的。


```
1  export default defineConfig({
2    qiankun: {
3      master: {
4        apps: [
5          {
6            name: 'sub-umi',
7            entry: '://localhost:5175',
8            activeRule: '/qiankun/umi',
9            sandbox: {
10             strictStyleIsolation: true, //严格样式隔离
11           },
12         },
13       ],
14     },
15   },
```

```
16 //其他配置
17 })
```

strictStyleIsolation原理

【基于shadow dom来做的样式隔离】

shadowDOM 的MDN地址如下：



https://developer.mozilla.org/zh-CN/docs/Web/API/Web_components/Using_sh...

使用影子 DOM - Web API | MDN

自定义元素的一个重要方面是封装，因为自定义元素从定义上来说是一种可重用功能：它可以被放置在任何网页中，并且期望它能够正常工作。因此，很重要的一点是，...

`strictStyleIsolation` 的原理是基于 Web Components 中的 **Shadow DOM** 技术。让我详细解释一下：

1. 基本实现原理：

```
1 function createShadowContainer(container, appName) {
2   // 创建 Shadow DOM
3   const shadow = container.attachShadow({ mode: 'open' });
4
5   // 子应用的所有内容都会被放入这个 Shadow DOM 中
6   return shadow;
7 }
```

实际效果：

```
1 <!-- 普通 DOM 结构 -->
2 <div id="main">
3   <div id="app1">
4     #shadow-root (open)
5     <!-- 子应用的所有内容都在 Shadow DOM 中 -->
6     <style>
7       .title { color: red; }
8     </style>
```

```
9      <div class="title">我是app1的标题</div>
10    </div>
11  </div>
```

2. Shadow DOM 的特性：

- 独立的 DOM 树
- 样式完全隔离
- JavaScript 访问限制
- 事件局部化

3. 样式隔离效果：

```
1  /* Shadow DOM 内部的样式 */
2  .title { color: red; }
3
4  /* 外部的样式无法影响到 Shadow DOM 内部 */
5  #main .title { color: blue; } /* 这个样式不会影响 Shadow DOM 内的 .title */
```

主要优点：

- 完全的样式隔离
- 不需要额外的样式转换
- 原生的隔离方案

主要缺点：

- 一些第三方库可能无法正常工作
- 弹窗类组件可能会被限制在 Shadow DOM 内
- 浏览器兼容性问题

二、js沙箱

1. 在基座中修改window会共享到各个子应用。
2. 在子应用A修改window不会影响到子应用B。

qiankun中js沙箱的原理

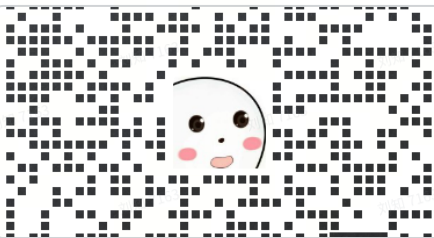
qiankun中的js沙箱是对window进行隔离，主要解决全局变量冲突和全局状态相互影响的问题。

qiankun提供了三种沙箱模式：

- 1、在不支持proxy的浏览器，提供【快照沙箱】，在进入子应用之前
- 2、在支持proxy的浏览器，用proxy代理window，子应用修改代理后的window。单例就采用legacySandbox沙箱，多例就采用proxySandbox沙箱。

类似代码：

```
1  const proxy = new Proxy(window)
2  (function(window){
3    //子应用的代码
4  })(proxy)
```



👉 <https://juejin.cn/post/6920110573418086413#heading-12>

15分钟快速理解qiankun的js沙箱原理及其实现qiankun框架为了
实现js隔离，提供了三种不同场景使用的沙箱，分 - 掘金
qiankun框架为了实现js隔离，提供了三种不同场景使用的沙箱，分别是…

三、剔除重复依赖

如果基座和子应用使用了项目的库，可以考虑子应用使用基座的包，从而减少重复加载

有两种方式：

1. externals，基座用cdn引入包，子应用相同的cdn设置为ignore。（**更推荐用externals**）
2. 模块联邦，基座将重复包打包至remote.js，子应用不打包重复的包，而是在运行时请求基座的remote.js

externals

流程：

- 基座：将所有公共依赖配置 `webpack` 的 `externals`，并且在 `index.html` 使用外链引入这些公共依赖
- 子应用：和主应用一样配置 `webpack` 的 `externals`，并且在 `index.html` 使用外链引入这些公共依赖，注意，还需要给子应用的公共依赖的加上 `ignore` 属性（这是自定义的属性，非标准属性），`qiankun`在解析时如果发现 `ignore` 属性就会自动忽略

以`lodash`为例：

基座：

修改`config/config.ts`文件，在`externals`中添加`lodash`，之后在`headScripts`数组中添加`lodash`的`cdn`地址。

```
1 // 修改config/config.js
2 export default defineConfig({
3   /**
4    * @name <head> 中额外的 script
5    * @description 配置 <head> 中额外的 script
6    */
7   headScripts: [
8     // 解决首次加载时白屏的问题
9     //{ src: '/scripts/loading.js', async: true },
10    //lodash 的cdn
11    { src: 'https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js',
12      async: false },
13    ],
14   externals: {
15     lodash: '_', //externals 指定lodash和他的全局变量名
16   },
17 })
```

umi子应用：

子应用同样需要在自己的配置文件中添加`cdn`的`lodash`，并且需要添加`ignore`忽略`lodash`。

<!-- 注意：这里的公共依赖的版本，基座和子应用需要一致 -->

```
1
2 export default defineConfig({
3   // 剔除重复包
```

```
4     externals: {
5       lodash: '_',
6     },
7     headScripts: [
8       {
9         src: 'https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js',
10        async: false,
11        ignore: true,
12        // 子应用需要添加ignore, 忽略lodash, 使用基座挂载window上的lodash, 这样只需要
        请求一次
13      },
14    ],
15  })
```

四、公共组件

我们后续会用pnpm的workspace来做monorepo，这样在单仓下基座和子应用就能共享组件了，但是这样还有一个问题，当公共组件变化，子应用就需要重新打包部署才能得到公共组件的变化，所以可以采用模块联邦的方式，让子应用使用基座的远程组件，这样就只需要对基座进行打包部署。

接下来先介绍monorepo改造基座，并举例在子应用中如何使用workspce的公共组件，最后介绍将公共组件转为远程组件使用。

1、monorepo改造

1. 在基座的根目录新建 `pnpm-workspace.yaml` 文件，文件内容：

意思是：

- 会将这三个文件夹下的目录添加到workspace工作空间中，他们可以相互通过workspce访问到。
- 其中app文件夹存放子应用的项目代码，src/expoese/components文件下存放各种公共组件的代码。
- 在umi中要导出远程组件，需要将组件写到src/exposes文件夹下，umi自动处理exposes文件夹下的组件。

```
1  packages:
2    - 'apps/*'
3    - 'src/exposes/components/*'
4    - 'src/exposes/*'
```

2. 新建yaml中涉及的文件夹，将子应用放入apps文件夹中。

至此简单的monorepo改造完毕

2、添加公共组件

下面演示 `PureButton` 这个公共组件的创建和使用。

1. 前置步骤

在`src/exposes/components`（看上面，这是一个workspace目录）目录下执行 `npm init -y`，我们会将components目录作为组件库的目录，后续的公共组件都写在这个目录下。其package.json类似步骤2（ps：也可以不这么做，也可以直接所有公共组件写在exposes下，只要组件是个npm包就行。）

```
1  {
2    "name": "components",
3    "version": "1.0.1",
4    "description": "",
5    "main": "index.tsx",
6    "keywords": [],
7    "author": "",
8    "license": "ISC"
9  }
10
```

2. 在workspce中创建组件包

在`src/exposes/components`下新建一个 `PureButton` 文件夹，在这个文件夹中执行 `npm init -y` 生成package.json 文件，主要修改文件中以下字段：

- name（我们可以用@loctek这个前缀，包名用横线分割，不要用驼峰。）
- version
- main(main是这个包的入口)

```
1  {
2    "name": "@loctek/pure-button",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.tsx",
6    "types": "index.d.ts",
7    "keywords": [],
8    "author": "",
```

```
9     "license": "ISC",
10    "private": true
11  }
12
```

3. 书写组件代码

新建index.tsx，书写组件的代码

```
1  import styles from './style.module.less';
2  import type { Props } from './index.d';
3
4  const PureButton = ({ btnStr = '' }: Props) => {
5    return (
6      <>
7        <div className={styles['btn-container']}>{btnStr}</div>
8      </>
9    );
10 };
11
12 export default PureButton;
13
```

4. 测验组件是否生效

(这里单纯测试workspace是否生效，实际项目我们采用模块联邦的方式。)

在本地开发中，我们可以在apps的子应用中，在其package.json中添加上面的包，然后在子项目的目录执行 `pnpm i` 或者 `pnpm i @loctek/pure-button` 命令，这样就会将 `@loctek/pure-button` 组件的软链接添加到子项目的node_modules中供子应用使用。

```
1  "dependencies": {
2    //其他包...
3    "@loctek/pure-button": "workspace:*",
4  },
```

3、利用模块联邦使用公共组件

官网地址：



<https://umijs.org/docs/max/mf>

Module Federation 插件

在 Umi 项目使用 Module Federation 功能。

1. 改造基座，导出远程组件

在config/config.ts中使用 mf 字段，这样最终会在打包文件中多出一个remote.js文件。用name起一个名字，remoteHash用于取消打包的hash，library指定打包的文件的模块

```
1 export default defineConfig({
2   // 模块联邦
3   mf: {
4     name: 'master',
5     //关闭remote.js的hash
6     remoteHash: false,
7     // qiankun时必须要有，window是挂载到window上，默认是var
8     library: { type: 'window', name: 'master' },
9   },
10  //其他配置
11 })
```

2. 子应用注册远程组件

子应用的.umirc.ts中，1、添加mf插件、2、添加mf字段

mf中的remotes指定访问的远程组件地址和他的name，基座最终会被部署到80端口，所以我们的entry就是 `//localhost:80/remote.js`，remote.js就是会将基座中src/exposes下的文件打包进去。(ps:在本地开发阶段可以写基座项目启动的地址，比如 `//localhost:8080/remote.js`)

shared字段填这个子应用用到的远程包，因为我们会将所有的组件写入components，所以直接这样写好就行了。

```
1   plugins: ['@umijs/plugins/dist/qiankun', '@umijs/plugins/dist/model',
2     '@umijs/plugins/dist/mf'], //添加 @umijs/plugins/dist/mf
3   mf: {
4     remotes: [
```

```

4      {
5          aliasName: 'masterAppXXX',//一个别名
6          name: 'master', // 对应基座应用的 name
7          entry: '///localhost:80/remote.js',// 基座应用中导出的共享包的入口
8      },
9  ],
10 // 声明共享依赖
11 shared: {
12     'components': {
13         singleton: true,//单例，整个应用只存在一个，防止一个库加载多个版本。
14         eager: false,//控制共享模块的加载时机，默认为false：异步加载，实际用的时候加载；为true时指：同步加载，应用启动就加载，使用于本地开发的时候。
15         requiredVersion: '^1.0.1',
16     },
17 },
18 },
19

```

3. 使用远程组件

随便在子应用中找个文件用远程的PureButton组件

```

1 //impoer MasterApp from '别名/包名'
2 import MasterApp from 'masterAppXXX/components';
3 //因为是默认导出，所以需要拿到默认导出的东西后解构
4 const { PureButton } = MasterApp;
5 export default function Foo(){
6
7     return (<>
8         <PureButton btnStr="555" />
9     </>)
10 }

```

五、应用之间的通信

官网地址：



<https://umijs.org/docs/max/micro-frontend#%E7%88%B6%E5%AD%90%E5%B...>

微前端

@umi/max 内置了 Qiankun 微前端插件，它可以一键启用 Qiankun 微前端开发模式，帮助您轻松地在 Umi 项目中集成 Qiankun 微应用，构建出一个生产可用的微前端架...

useModel

基座

在基座中定义了一些model，并且在app.jsx这个入口文件中导出子应用中需要使用的model

入口文件：app.tsx中导出useQiankunStateForSlave函数。

```
1 // 子应用(需要子应用是umi项目)获取主应用的全局状态，需要在app.tsx导出
  useQiankunStateForSlave供umi使用
2 interface QiankunState {
3   site: string;
4 }
5 export function useQiankunStateForSlave(): QiankunState {
6   const { site } = useModel('site');
7
8   return {
9     site, // 导出site供子应用使用
10  };
11 }
```

子应用

在子应用中用 `useModel('@@qiankunStateFromMaster')` 即可拿到导出的数据

```
1
2 import { useModel } from 'umi';
3
4 export default function HomePage() {
5   // 获取主应用的actions
6
7   const model = useModel('@@qiankunStateFromMaster');
```

```
8 console.log(model?.site)
9 return <>...<>
10 }
```

nginx部署

官网地址：

<https://qiankun.umijs.org/zh/cookbook#%E5%A6%82%E4%BD%95%E9%83%A8%E7%BD%B2>

入门教程 - qiankun

qiankun 指南 API 常见问题 入门教程 版本公告 发布日志 升级指南 1.x 版本 GitHub English qiankun 指南 API 常见问题 入门教程 版本公告 发布日志 升级指南 1.x 版本 GitHub English 入门教程 微应用的路由模式如何选择 activeRule 使用 location.pathnam...

- 可以将子应用和基座部署在同一个server，也可以部署在不同的server下。

一、部署在同一个server

部署在同一个server下，

- 子应用的路由base需要和基座的activeRule保持一致。
- 子应用的entry则需要与其nginx路径一致

基座注册：

```
1 qiankun: {
2   master: {
3     apps: [
4       {
5         name: 'sub-umi-1', //子应用的名称
6         entry: '/sub/umi1/', //子应用的入口地址,nginx配置的目录
7         activeRule: '/app-umi1-history', //子应用的激活规则,指路由
8         sandbox: {
```



```

9      strictStyleIsolation: true, //严格样式隔离
10    },
11  },
12 ],
13 },
14 },

```

子应用.umirc.ts

```

1  base: '/app-umi2-history', //供基座访问的路由前缀，会和activeRule一样
2  // 用qiankun插件后默认base为包名
3  publicPath: '/sub/umi2/', //资源的前缀，会和nginx中存放的目录保持一次

```

二、部署在不同的server

部署在不同的server需要为子应用的server添加跨域

nginx.conf内容：

```

1  # main主应用
2  server {
3      listen      80;
4      server_name localhost;
5
6
7      # CORS 配置
8      add_header Access-Control-Allow-Origin '*' always;
9      add_header Access-Control-Allow-Methods 'GET, POST, OPTIONS' always;
10     add_header Access-Control-Allow-Headers '*' always;
11     add_header Access-Control-Allow-Credentials 'true' always;
12
13     if ($request_method = 'OPTIONS') {
14         return 204;
15     }
16
17     location / {
18         root    html/main;
19         index  index.html index.htm;
20         try_files $uri $uri/ /index.html;
21     }
22
23     # API 代理配置

```

```
24     location ^~ /auth {
25         proxy_pass http://mall-center.dev.springbeetle.top;
26     }
27
28     location ^~ /perm {
29         proxy_pass http://mall-center.dev.springbeetle.top;
30     }
31
32     # 处理子应用的代理
33     # location ^~ /qiankun/react {
34     #     proxy_pass http://localhost:5173;
35     # }
36
37     error_page 500 502 503 504 /50x.html;
38     location = /50x.html {
39         root html;
40     }
41 }
42 # umi子应用
43 server {
44     listen 5175;
45     server_name localhost;
46
47     # # 添加5174全局 CORS 配置
48     add_header Access-Control-Allow-Origin '*' always;
49     add_header Access-Control-Allow-Methods 'GET, POST, OPTIONS' always;
50     add_header Access-Control-Allow-Headers '*' always;
51     add_header Access-Control-Allow-Credentials 'true' always;
52
53     if ($request_method = 'OPTIONS') {
54         return 204;
55     }
56
57     location / {
58         root html/qiankun/umi;
59         index index.html index.htm;
60         try_files $uri $uri/ /index.html;
61     }
62
63     # API 代理配置
64     # location ^~ /auth {
65     #     proxy_pass http://mall-center.dev.springbeetle.top;
66     # }
67
68     error_page 500 502 503 504 /50x.html;
69     location = /50x.html {
70         root html;
```

71	}					
72	}					