# qiankun

## [微前端] 路由加载流程是如何的？【热度: 971】

**关键词：微前端路由加载**

微前端是一种架构模式，旨在将大型前端应用程序拆分为更小、更容易维护的独立部分。微前端的路由原理可以通过以下步骤概括：

1. 主应用加载：用户访问主应用时，主应用负责加载，并决定加载哪些微前端应用。
2. 路由分发：主应用根据当前URL路径，将请求分发给相应的微前端应用。
3. 微前端应用加载：被分发的微前端应用根据接收到的请求加载自己的代码和资源。
4. 渲染内容：微前端应用接收到请求后，将自己的内容渲染到主应用的容器中。
5. 子应用间通信：如果不同微前端应用之间需要进行通信，可以使用共享的状态管理工具或事件总线。
6. 事件处理：主应用和微前端应用都可以处理路由变化事件，以便更新页面内容。

## [微前端] 微前端架构一般是如何做 JavaScript隔离【热度: 127】

**关键词：JS 隔离**

在微前端架构中，JavaScript 隔离是核心之一，用以确保各个子应用间代码运行时不互相干扰、变量不冲突，以及能够安全地卸载应用。为了实现这一目标，主要采用以下几种方法：

### 1. 使用沙箱技术：

iframe：最直接的隔离方式是将子应用运行在iframe中。这种方式提供了良好的隔离性，因为iframe内部有自己独立的执行环境，包括 JavaScript 运行环境和 DOM 环境。但iframe的使用可能会导致性能问题，且父子通信复杂。

JavaScript Sandboxing：通过创建一个独立的 JavaScript 执行环境，比如使用 Web Workers，或者更高级的沙箱库（如 Google 的 Caja），以在主页环境隔离执行 JavaScript 代码。

### 2. 命名空间和模块化：

命名空间：通过命名空间（Namespace）封装每个子应用的代码，确保全局变量和函数不会与其他应用冲突。

模块化：利用 ES Modules 或 CommonJS 等模块化标准，使代码封装在模块中运行，通过 import/export 管理依赖，减少全局变量的使用，从而实现隔离。

### 3. 状态管理隔离：

虽然主要关注 JavaScript 代码的隔离，但在单页应用中，子应用间状态管理（如使用 Redux、Vuex 等状态管理库）也可能导致隔离问题。可以为每个子应用创建独立的状态树，只通过明确定义的接口来共享必要的状态信息。

### 4. 使用微前端框架或库：

模块联邦（Module Federation）：Webpack 的模块联邦功能允许不同的前端应用共享 JavaScript 模块，同时保持应用间的隔离。它可以动态地加载另一个应用导出的模块，而不需要将它们打包进单个文件里。

专门的微前端框架：如 Single-SPA、Qiankun 等，这些框架提供了一套完整的解决方案，用于管理微前端应用的加载、卸载以及相互隔离，部分内部采用了类似沙箱的技术实现隔离。

### 5. 服务端渲染（SSR）隔离：

通过服务端渲染各个微前端应用，再将渲染好的静态 HTML 集成到主应用中。这样，每个子应用的 JavaScript 在客户端激活（Hydration）之前是隔离的。SSR 可以减少初次加载时间，同时具备部分隔离性，尤其是在初次加载阶段。

实施 JavaScript 隔离时，需要根据具体项目需求、技术栈和团队的熟练度来选取合适的隔离策略，以确保子应用之间的高度独立性和可维护性。

## 如果让你来设计实现一个微前端框架，你会怎么设计？

* 主应用和子应用之间的通信：可以通过自定义事件系统来实现，主应用和子应用之间通过自定义事件来进行通信。
* 子应用的加载和卸载：可以通过监听路由变化，动态加载和卸载微应用。
* 子应用之间的通信：可以通过自定义事件系统来实现，子应用之间通过自定义事件来进行通信。
* 样式隔离：可以通过将子应用的样式进行隔离，避免样式冲突。
* 共享状态：可以通过共享状态来实现，主应用和子应用之间通过共享状态来进行通信。
* 路由处理：可以通过监听路由变化，动态加载和卸载微应用。
* 生命周期管理：可以通过监听路由变化，动态加载和卸载微应用。
* 安全性：可以通过对子应用进行沙箱隔离，避免子应用对主应用的影响。
* 性能优化：可以通过对子应用进行懒加载，避免一次性加载所有子应用。
* 部署：可以通过将子应用进行打包，然后通过 CDN 进行部署。

## [微前端] 为何通常在 微前端 应用隔离， 不选择 iframe 方案【热度: 280】

**关键词：iframe 隔离方案弊端**

在微前端架构中，虽然iframe能提供很好的应用隔离（包括 JavaScript 和 CSS 隔离），确保微前端应用之间不会相互干扰，但一般不把它作为首选方案，原因包括：

### 1. 性能开销

iframe会创建一个全新的浏览器上下文环境，每个iframe都有自己的文档对象模型（DOM）树、全局执行环境等。如果一个页面中嵌入了多个iframe，就会导致额外的内存和 CPU 资源消耗，特别是在性能有限的设备上更为显著。

### 2. 应用集成和交互问题

iframe自然隔离了父子页面的环境，这虽然提供了隔离，但同时也使得主应用与子应用之间的交云难度增加。虽然可以通过postMessage等 API 实现跨iframe通信，但这种方式相比于直接 JavaScript 调用来说，更为复杂，交互效率也较低。

### 3. UI 体验一致性

在iframe中运行的应用在视觉上可能与主应用难以实现无缝集成。iframe内外的样式、字体等一致性需要额外的处理。此外，iframe可能带来额外的滚动条，影响用户体验。

### 4. SEO 问题

如果微前端的某些内容是通过iframe呈现的，那么这部分内容对于搜索引擎是不可见的，这可能会对应用的 SEO 产生负面影响。

### 5. 安全问题

虽然iframe可以提供一定程度的隔离，但它也可能引入点击劫持等安全风险。此外，过多地使用iframe也可能增加网站被恶意脚本攻击的表面。

因此，虽然iframe是一种可行的应用隔离方法，它的这些局限性使得开发者在选择微前端技术方案时，往往会考虑其他提供更轻量级隔离、更好集成与交互体验的方案，如使用 JavaScript 沙箱、CSS 隔离技术、Web Components 等。这些方法虽然隔离性可能不如iframe彻底，但在整体的应用性能、用户体验和开发效率上通常会有更好的表现。

## 简述qiankun微前端的原理

qiankun 是一个基于 single-spa 的微前端实现库，它能够帮助我们将多个独立的前端应用聚合成一个整体的应用。它的实现原理是通过监听路由变化，动态加载和卸载微应用，同时通过自定义事件系统来实现微应用之间的通信

## qiankun 微前端的优势是什么？

* 可以将多个独立的前端应用整合成一个整体，提高了代码复用性和可维护性。
* 可以独立开发、独立部署、独立运行，提高了开发效率和部署灵活性。
* 可以实现子应用之间的通信和数据共享，提高了应用之间的协作能力。
* 可以实现子应用的按需加载和动态卸载，提高了应用的性能和用户体验。

## qiankun 微前端的基本使用流程是什么？

* 在主应用中安装 qiankun 库，并注册需要加载的子应用。
* 在子应用中导出一个生命周期对象，并在主应用中注册该子应用。
* 在主应用中启动 qiankun 应用，并指定需要加载的子应用。
* 在主应用中渲染子应用的容器，并在容器中加载子应用。
* 在主应用和子应用中实现应用间通信和数据共享。

## qiankun主应用如何开启样式隔离？

qiankun 主应用可以通过设置 sandbox 属性来开启样式隔离，sandbox 属性可以设置为true 或者一个对象，当设置为true 时，会开启默认的样式隔离，当设置为一个对象时，可以自定义样式隔离的配置。示例

loadMicroApp({

name: 'sub-app'

entry: '[//localhost: 8080](notion://localhost/)',

container: '#sub-app-container' ,

sandbox: { strictStyleIsolation: true },

})；

## qiankun主子应用、子子应用之间的通信方式有哪些？

* 自定义事件：主子应用、子子应用之间可以通过自定义事件来进行通信。
* 全局变量：主子应用、子子应用之间可以通过全局变量来进行通信。
* window 对象：主子应用、子子应用之间可以通过 window 对象来进行通信。
* HTML5 通信 API：主子应用、子子应用之间可以通过 HTML5 通信 API来进行通信，如 postMessage、BroadcastChannel 等。
* 共享状态：主子应用、子子应用之间可以通过共享状态来进行通信。
* 跨域通信：主子应用、子子应用之间可以通过跨域通信来进行通信，如JSONP、CORS 等。
* 代理：主子应用、子子应用之间可以通过代理来进行通信，如 Nginx 代理、反向代理等。
* 中间件：主子应用、子子应用之间可以通过中间件来进行通信，如koa 中间件、express 中间件等。
* 事件总线：主子应用、子子应用之间可以通过事件总线来进行通信，如 EventBus、EventEmitter 等。
* WebSocket： 主子应用、子子应用之间可以通过 WebSocket 来进行通信。
* 服务端：主子应用、子子应用之间可以通过服务端来进行通信，如 RESTful API、GraphQL等。

## qiankun 微前端的子应用如何实现按需加载？

qiankun 微前端的子应用可以通过导出一个异步加载函数来实现按需加载，例如：

export async function bootstrap(){

// 子应用的启动逻辑

}

export async function mount) {

// 子应用的挂载逻辑

}

export async function unmount) {

// 子应用的卸载逻辑

}

在主应用中，可以通过 loadMicroApp 方法来动态加载子应用，例如：

import { loadMicroApp } from 'qiankun';

loadMicroApp({

name: 'sub-app',

entry: '//localhost:8080',

container: '#sub-app-container'

})

## qiankun 微前端的子应用如何实现路由跳转？

qiankun 微前端的子应用可以通过 history.pushState 方法来实现路由跳转，例如：

history-pushState(null, null, '/path');

在主应用中，可以通过 setMatchedPath 方法来设置当前子应用的路由路径，例如：

import ｛ setMatchedPath ｝from 'qiankun'；

setMatchedPath ('/path');

在主应用中，可以通过 onGlobalStateChange 方法来监听子应用的路由变化，例如：

import { onGlobalStateChange } from 'qiankun';

onGlobalStateChange((state, prev) = {

console.log（state.matchedPath）；// '/path'

｝）；

### [微前端] Qiankun 是如何做 JS 隔离的【热度: 228】

**关键词：JS 隔离**

Qiankun 是一个基于 Single-SPA 的微前端实现库，它提供了比较完善的 JS 隔离能力，确保微前端应用间的独立运行，避免了全局变量污染、样式冲突等问题。Qiankun 实现 JS 隔离的主要机制包括：

### 1. JS 沙箱

Qiankun 使用 JS 沙箱技术为每个子应用创建一个独立的运行环境。沙箱有以下两种类型：

快照沙箱（Snapshot Sandbox）：在子应用启动时，快照并记录当前全局环境的状态，然后在子应用卸载时，恢复全局环境到启动前的状态。这种方式不会对全局对象进行真正的隔离，而是通过记录和恢复的方式避免全局环境被污染。

Proxy 沙箱：通过 Proxy 对象创建一个全新的全局对象代理，子应用的所有全局变量修改操作都将在这个代理对象上进行，从而不会影响到真实的全局对象。这种方式提供了更为彻底的隔离效果，是 Qiankun 中推荐的沙箱隔离方式。

### 2. 动态执行 JS 代码

Qiankun 通过动态执行 JS 代码的方式加载子应用，避免了脚本直接在全局环境下执行可能导致的变量污染。具体来说，它可以动态获取子应用的 JS 资源，然后在沙箱环境中运行这些代码，确保代码执行的全局变量不会泄露到主应用的全局环境中。

### 3. 生命周期隔离

Qiankun 给每个子应用定义了一套生命周期钩子，如 bootstrap、mount、unmount 等，确保在应用加载、激活和卸载的过程中正确管理和隔离资源。通过在 unmount 生命周期钩子中正确清理子应用创建的全局监听器、定时器等，进一步保证了不同子应用间的独立性和隔离性。

### 4. 样式隔离

虽然主要针对 JS 隔离，Qiankun 也提供了样式隔离机制，通过动态添加和移除样式标签，保证子应用样式的独立性，避免不同子应用间的样式冲突。

通过以上机制，Qiankun 能够有效实现微前端架构中子应用的 JS 隔离，加强了应用间的独立性和安全性，使得不同子应用可以无缝集成在一起，同时又能够保持各自的运行环境独立不受影响。