微前端课程资料

1. 课程目标

- **初级**：掌握微前端基础概念。
- **中级**：掌握目前前端微前端落地实践。
- **高级**：熟悉现在微前端的底层设计，熟悉多个框架的实际设计及对比。

2. 课程大纲

- 微前端介绍
- 微前端常见框架
- 微前端对比&总结
- 课后问题

---

3. 微前端介绍

微前端定义

微前端（Micro-Frontends）是一种类似于微服务的架构，它将微服务的理念应用于浏览器端，即将 Web 应用由单一的单体应用转变为多个小型前端应用聚合为一的应用。

简单的说：**微前端就是在一个Web应用中独立运行其他的Web应用**。

微前端特点

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，我们通常很难对一个已经存在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。

5. **环境隔离**：应用之间 JavaScript、CSS 隔离避免互相影响。
6. **消息通信**：统一的通信方式，降低使用通信的成本。
7. **依赖复用**：解决依赖、公共逻辑需要重复维护的问题。

微前端框架需了解哪些部分

- **整体隔离机制：**首先了解框架在全局层面是如何实现 **JS 隔离**（如沙箱、Proxy、快照沙箱、iframe 沙箱等）以及 **CSS 隔离**（如 Shadow DOM、Scoped CSS、样式前缀、动态隔离域）的方法与原理。
- **核心运行机制：**理解主应用如何调度子应用、如何加载与卸载资源、如何分发路由以及如何管理生命周期。
- **框架特点与侧重点**：在掌握隔离与运行机制后，再了解该框架的设计取舍——它更偏向轻量、易接入，还是偏向强隔离、安全、性能优化等方向。
- **通信体系**：了解其是否提供内置事件机制、全局状态管理或跨应用通信方案，以及通信是否与隔离机制冲突或互补。
- **依赖共享与工程化**：框架对公共依赖如何处理、如何避免重复加载、如何提升构建效率以及是否支持动态模块加载（如 Module Federation）。

---

4. 微前端常见框架

4.1 iframe

整体隔离机制：iframe 依托浏览器原生提供的最强隔离环境：每个 iframe 都拥有独立的 JS 上下文（独立的 window、document）、事件循环以及独立的 CSS 渲染树，父子页面互不影响，因此无需额外的沙箱或 CSS 隔离技术。

核心运行机制：主应用通过创建、销毁 iframe 元素来加载或卸载子应用，资源加载全由浏览器负责；iframe 内的路由与主应用 URL 不同步，无法统一路由调度，也不存在微前端所需的生命周期管理能力。

框架特点与侧重点：iframe 最大优势是隔离彻底、安全性高，接入成本极低；

缺点：

a. **url 不同步**：浏览器刷新时 iframe 的 url 状态会丢失，浏览器的前进、后退按钮也无法使用。
b. **UI 不同步，DOM 结构不共享**：例如，在 iframe 内的弹窗要实现浏览器全局居中会非常困难。

c. **全局上下文完全隔离**：内存变量不共享，主子应用间的通信、数据同步、cookie 透传等都存在问题。

d. **性能差**：每次子应用进入都是一次浏览器上下文的重建和资源的重新加载。

通信体系：不提供任何原生的跨应用通信机制，只能通过 postMessage、全局变量桥接等方式实现点对点通信；但由于上下文完全隔离，通信复杂、能力受限，不利于构建多应用协作体系。

依赖共享与工程化：每个 iframe 都运行在独立的上下文中，无法共享依赖和缓存，所有资源会重复加载；无法结合工程化体系实现依赖预加载、构建优化，也不支持 Module Federation 等动态依赖管理方案。

4.2 single-spa

整体隔离机制

single-spa 本身不提供严格的隔离环境。所有子应用在同一个 `document` 下运行，共享同一个 JS 主线程和全局 `window` 对象。因此，它需要依赖其他方案来解决 JS 和 CSS 的隔离问题：

- **JS 隔离**：通常需要手动管理或借助 `single-spa-leaked-globals` 等插件来清理子应用卸载后可能泄露的全局变量。但这种方式并不完美，尤其是当不同子应用依赖同一全局库的不同版本时，依然存在冲突风险。
- **CSS 隔离**：由于所有 DOM 都在一个文档中，样式会全局生效，容易产生冲突。需要开发者自行采用 BEM、CSS Modules、CSS-in-JS、Shadow DOM 或为样式规则添加特定前缀等方案来解决。

核心运行机制

single-spa 是一个微前端路由和生命周期调度器。它通过劫持 `popstate`、`hashchange` 事件和 `history.pushState` 等路由相关 API，来监听 URL 的变化。当 URL 变更时，它会匹配预先注册的子应用，并依次调用其 `bootstrap`（引导）、`mount`（挂载）、`unmount`（卸载）等生命周期钩子，从而实现对子应用的加载和管理。

框架特点与侧重点

single-spa 的核心优势是提供了一套与框架无关的生命周期管理标准，实现了统一的路由调度机制。

- **优点**：
  a. **统一路由**：主子应用共享 URL，浏览器刷新状态不丢失，前进、后退功能正常。
  b. **DOM 共享**：所有应用在同一 DOM 树下，便于实现全局弹窗、布局等复杂 UI 交互。
  c. **上下文共享**：共享 `window` 对象，使得数据共享和通信相对直接。
- **缺点**：
  a. **隔离性弱**：CSS 和 JS 默认不隔离，需要开发者额外处理以避免冲突。

b. **接入成本高**：子应用需要进行改造，以符合 single-spa 的生命周期规范（导出 `bootstrap`，`mount`，`unmount` 函数）。

c. **技术选型限制**：虽然框架无关，但若主应用和多个子应用技术栈差异巨大（如 React 和 Vue 混合），可能导致包体积增大和潜在的性能问题。

通信体系

框架本身不提供通信机制。由于共享同一个全局上下文（`window`），应用间通信相对灵活，可以通过以下方式实现：

- 使用 `window` 对象挂载全局变量或事件总线（EventBus）。
- 通过 `CustomEvent` API 进行事件派发和监听。
- 借助 RxJS 等响应式库来管理跨应用状态。
- 通过 `props` 在主应用中向子应用传递数据和回调。

依赖共享：

single-spa 推荐使用 **SystemJS** 和 **Import Maps** 方案来管理公共依赖。通过在主应用的 `index.html` 中定义 Import Map，可以将 React、Vue 等公共库或自定义的工具函数模块映射到指定的 URL。这样，所有子应用都可以通过模块名（`System.import('@org/react')`）加载同一个外部依赖，从而实现单例加载，避免资源冗余。这种方式与 Webpack 的 Module Federation 类似，能够很好地融入现代前端工程化体系。

代码示例：

关于具体的实现，可以参考 `single-spa` 的官方文档，它详细介绍了如何配置 `root-config`（主应用）、注册子应用、定义生命周期以及使用 `single-spa-layout` 进行布局。

官方文档链接: single-sp.js.org

核心生命周期函数

在一个 `single-spa` 页面中，注册的应用程序会经历加载、初始化 (bootstrap)、挂载 (mount)、卸载 (unmount) 和移除 (unload) 等阶段。`single-spa` 通过生命周期钩子提供了对每个阶段的访问。 [^1]

每个生命周期函数都必须返回一个 `Promise` 或是一个 `async` 函数。它们都会接收到一个 `props` 对象作为参数，其中包含应用名称、 `single-spa` 实例和自定义属性等信息。 [^1]

以下是必须实现的三个核心生命周期函数：

- **`bootstrap`** :
  - **调用时机**: 这个函数在应用第一次被激活前，只会被调用一次。
  - **用途**: 用于执行应用的初始化逻辑，例如设置全局变量、初始化资源或在应用挂载前运行一次性代码。
- **`mount`** :
  - **调用时机**: 当应用的 `activity function` 返回真值（即应用需要被激活显示）时调用。
  - **用途**: 创建并挂载 DOM 元素，渲染 UI，添加 DOM 事件监听器等。这是将你的微前端内容展示给用户的地方。
- **`unmount`** :
  - **调用时机**: 当应用的 `activity function` 从真值变为假值（即应用需要被隐藏）时调用。
  - **用途**: 清理 `mount` 阶段创建的所有资源，例如移除 DOM 元素、清除事件监听器，以释放内存并避免应用间的干扰。

可选的生命周期函数

- **`unload`** :
  - **调用时机**: 仅当手动调用 `unloadApplication` API 时触发。
  - **用途**: 在应用被完全移除（状态变回 `NOT_LOADED` ，下次激活会重新 `bootstrap` ）之前执行清理逻辑。这对于实现热重载等高级功能非常有用。

这些生命周期函数是 `single-spa` 微前端架构的核心。通过正确地实现它们，你可以确保各个独立的微前端应用能够和谐地共存在同一个页面上，并由 `single-spa` 进行统一的路由和状态管理。对于不同的前端框架（如 React, Angular, Vue）， `single-spa` 社区提供了相应的辅助库（如 `single-spa-react` ），这些库会帮助你自动生成符合 `single-spa` 要求的生命周期函数。

4.3 qiankun

整体隔离机制：

qiankun 提供了生产级的沙箱隔离方案，这是其相较于 single-spa 的核心增强。

- **JS 隔离**：基于 ES6 的 `Proxy` 实现。它为每个子应用创建一个代理的 `window` 对象（`fakeWindow`），劫持对全局 `window` 的所有读写操作。这使得子应用间的全局变量、事件监听等都限制在各自的沙箱内部，避免了全局命名冲突和环境污染。它提供了三种沙箱模式：
  a. **ProxySandbox**：支持多实例，每个应用拥有独立的状态，互不影响。
  b. **LegacySandbox**：单实例模式，通过快照对比 `window` 属性的变更来实现隔离。
  c. **SnapshotSandbox**：在不支持 `Proxy` 的旧版浏览器中的降级方案，原理与 LegacySandbox 类似。
- **CSS 隔离**：默认启用基于 `Shadow DOM` 的隔离方案。当浏览器支持时，`qiankun` 会将子应用包裹在一个 `Shadow Root` 中渲染，从而使其样式与主应用及其他子应用完全隔离，无需开发者手动处理。

核心运行机制：

`qiankun` 采用“HTML Entry”的方式加载子应用，极大地简化了接入流程。其核心步骤如下：首先通过 `fetch` 获取子应用的 `index.html` 文件；然后解析该 HTML，提取出所有的 JS 和 CSS 资源；接着，它会再次 `fetch` 所有外链的样式表并将其转换为内联 `<style>` 标签；最后，将 JS 代码在前面提到的沙箱环境中执行，最终完成子应用的渲染。整个过程由框架自动完成。

框架特点与侧重点：

作为 `single-spa` 的封装，`qiankun` 致力于提供开箱即用的生产级体验，重点解决隔离和资源加载两大痛点。

- **优点**：
  a. **强隔离性**：内置了可靠的 JS 和 CSS 沙箱，开发者无需过多关注隔离问题。
  b. **接入极简**：基于 HTML Entry，子应用几乎无需改造，像开发普通 SPA 一样即可接入。
  c. **资源预加载**：支持配置 `prefetch`，利用浏览器空闲时间预先加载子应用资源，提升切换速度和用户体验。
  d. 继承 `single-spa` 的优点，如路由统一、DOM 共享等。
- **缺点/挑战**：
  a. **全局弹窗问题**：对于需要挂载到 `document.body` 的全局组件（如 Modal、Drawer），`Shadow DOM` 会破坏其原有布局，需要特殊处理，目前没有完美解决方案。
  b. **沙箱有性能开销**：`Proxy` 代理和快照机制会带来一定的性能损耗，尽管在现代浏览器中通常可接受。

通信体系：

`qiankun` 提供了一套官方的、基于发布订阅模式的全局状态通信机制。通过 `initGlobalState` API 可以创建一个全局共享的状态对象。主应用和子应用都可以订阅该状态的变化、监听更新，并通过 `setGlobalState` 方法进行修改，从而实现简单、可靠的跨应用通信。

依赖共享与工程化：

`qiankun` 官方推荐使用 Webpack 的 `externals` 配置来实现公共依赖的共享。具体做法是将 React、Vue 等公共库从子应用的构建包中排除，然后在主应用的 `index.html` 中通过 CDN 或本地服务器统一引入。这种方式简单直接，但官方也提示，不建议在技术栈或依赖版本差异较大的子应用之间强行共享依赖，以避免潜在的兼容性问题。

4.4. EMP (Module Federation)

整体隔离机制：

EMP/Module Federation 方案在处理 **不同技术栈**（如混合使用 React, Vue, Angular）时，隔离策略与处理相同技术栈时有本质区别。它不再依赖共享的全局上下文，而是通过将 **Web Components** 作为“隔离容器”来实现。

- **JS 隔离**: 通过 Web Components 的 **Shadow DOM** 实现。每个不同技术栈的子应用（如一个 Vue 应用）被完整地封装在一个自定义的 HTML 元素中（例如 <vue-app></vue-app>）。这个元素内部拥有自己独立的 DOM 树和作用域。从外部宿主应用（Host）来看，它只是一个普通的 HTML 标签，从而避免了不同框架（如 React 和 Vue）的运行时因操作同一个 window 或 document 对象而产生的冲突。
- **CSS 隔离**: 同样由 **Shadow DOM** 提供天然的强隔离。封装在 Web Component 内部的样式表只会作用于其内部的 DOM，完全不会泄露到外部，也几乎不受外部全局样式的影响。

核心运行机制：

当用于多技术栈场景时，Module Federation 的角色转变为一个**动态加载器**，负责在运行时获取封装了其他框架应用的 Web Component 代码。

核心流程：首先，将每个子应用通过各自框架的工具（如 @angular/elements 或 vue-custom-element）打包成一个标准的 Web Component。然后，在 Webpack 配置中，通过 ModuleFederationPlugin 的 exposes 选项将这个打包好的 Web Component 引导文件（bootstrap.js 或 main.js）暴露出去。

宿主应用（Host）则通过 remotes 配置引用这些远程模块。当路由触发或需要加载该组件时，宿主应用会动态地创建一个对应的自定义 HTML 标签（如 document.createElement('vue-app')），并将其插入到页面的指定位置。浏览器一旦检测到这个自定义标签，就会执行其内部已经封装好的、用于启动对应框架（如 Vue）的逻辑。

框架特点与侧重点：

这种模式的重点是 **“用标准技术（Web Components）抹平框架差异，再用模块联邦（Module Federation）实现动态加载”**。

优点：

1. **真正的技术栈无关**：允许在同一个应用中和平共存、独立部署和升级由 React, Vue, Angular 等不同框架甚至不同版本构建的部分。
2. **强隔离性**：得益于 Shadow DOM，JS 和 CSS 隔离效果非常好，解决了不同框架共存时的最大难题。
3. **渐进式迁移**：是大型项目从旧技术栈（如 AngularJS）向新技术栈迁移的理想方案，可以一次只替换一小部分功能。

缺点/挑战：

1. **包体积显著增大**：这被认为是 **反模式** 的主要原因。由于无法安全地共享核心框架（如不能让 Vue 和 React 共享同一个虚拟 DOM 机制），每个子应用都需要加载其完整的框架运行时。最终用户需要下载多个框架的库，导致初始加载性能变差。
2. **实现复杂度高**：每个子应用都需要额外进行 Web Component 的封装打包，宿主应用也需要编写加载和挂载这些自定义元素的逻辑。
3. **不适用于细粒度共享**：此模式适用于集成整个页面或大型功能区块，而不适合共享零散的小组件。

通信体系：

由于 Shadow DOM 的强隔离，跨框架通信变得相对困难。不能再依赖共享 window 对象。主要通过标准的 Web API 进行：

属性（Properties）和特性（Attributes）：宿主应用可以通过设置 Web Component 标签的 Attributes来向其传递简单的字符串类型数据。更复杂的数据（对象、数组）则通过 JavaScript 的属性（Properties）来传递。

自定义事件：子应用可以通过 new CustomEvent() 创建并向外派发事件，宿主应用或其他组件可以监听这些事件以接收数据或状态变更通知。这是一种标准的、跨框架的“自下而上”通信方式。

依赖共享与工程化：

在多技术栈场景下，依赖共享的策略变得非常保守。

- **框架库不共享**：React, Vue, Angular 等核心框架库 **绝对不能共享**，必须各自打包加载，这是保证稳定运行的前提。Module Federation 的 shared 配置在这种情况下意义不大。
- **纯业务逻辑或工具函数可共享**：对于不依赖特定UI框架的纯 JavaScript 工具函数库（例如 axios, lodash, date-fns），理论上可以共享，但通常为了降低复杂度和风险，更倾向于让每个应用独立打包自己的依赖。
- 官方不推荐在不同技术栈间使用此方案，它更适合作为解决历史遗留问题或公司合并后技术栈整合的 **无奈之举**，而非首选架构。

5. Wujie

Wujie 是一个基于 WebComponent 和 iframe 的微前端框架，它巧妙地结合了两者的优点，旨在解决传统 iframe 方案的痛点，提供更接近原生应用的集成体验。

整体隔离机制

- **JS 隔离**：Wujie 沿用了 iframe 作为天然的 JavaScript "沙箱"。每个子应用都运行在一个与主应用同域的、不可见的 iframe 中，这确保了子应用拥有独立的 window、document 和事件循环，实现了完美的 JS 运行时隔离。
- **CSS 隔离**：利用 WebComponent 的 Shadow DOM 技术，Wujie 为每个子应用创建了一个独立的 DOM 树。所有子应用的 DOM 元素都被渲染到这个 Shadow DOM 内部，从而天然地实现了 CSS 样式的隔离，避免了主子应用间的样式污染。

核心运行机制

Wujie 的核心在于其创新的“跨界”渲染机制。它通过 Proxy 劫持子应用 iframe 内的 document 对象，将子应用对 DOM 的所有操作（如创建、修改、删除节点）都代理到主应用中的 WebComponent (Shadow DOM) 上。这意味着，子应用的 JS 在 iframe 的里运行，但其视图却在主应用的里渲染，从而解决了 iframe 在 UI 层面无法融合的根本问题。

框架特点与侧重点

Wujie 的最大特点是**在享受 iframe 极致隔离优势的同时，解决了其所有交互体验上的弊端**。

优点：

1. **URL 同步**：通过将子应用的路由信息（如 hash 或 path）同步到主应用的 URL 上，实现了路由状态的统一管理。浏览器刷新时，Wujie 可以根据主应用 URL 恢复子应用的路由和页面状态，解决了 iframe 状态丢失的问题，浏览器的前进、后退功能也能正常工作。
2. **UI 融合，DOM 共享**：由于 DOM 实际渲染在主应用的 Shadow DOM 中，子应用的弹窗、抽屉等绝对定位元素都是相对于主应用视窗进行定位的，完美解决了 iframe 中 UI 元素无法全局居中或覆盖全局的问题。
3. **内存变量共享**：主子应用在同一个域下，子应用可以通过 window.parent 直接访问主应用的全局上下文，方便进行变量共享和方法调用。
4. **高性能体验**：支持应用预加载和应用保活（keep-alive）。应用切换时，本质上只是对 Shadow DOM 的隐藏/显示或插拔，无需重新加载资源和重建浏览器上下文，切换速度极快，体验流畅，有效解决了白屏问题。

通信体系

得益于同域和共享的 UI 层，Wujie 提供了立体化的通信方案：

- **Props 注入**：主应用可以像 Vue/React 组件一样，通过属性向子应用传递数据。
- **window.parent**：子应用可以直接通过 window.parent 访问主应用的 window 对象，实现双向通信。
- **EventBus**：内置了基于发布订阅模式的 EventBus，为主子应用、子子应用之间提供了跨应用通信的能力，便于构建协作体系。

依赖共享与工程化

Wujie 支持公共依赖的共享，以优化性能和减少资源冗余。主应用可以预先加载如 Vue、React 等公共库，子应用在运行时可以声明并“借用”这些已加载的依赖，而无需重复请求和执行。这种机制可以与现代化的工程体系（如 Webpack）结合，实现依赖预加载和构建优化。

---

5. 微前端对比&总结

5.1. 方案对比

特性	single-spa	qiankun	wujie	Module Federation (EMP)	iframe
核心原理	路由劫持 + 生命周期	single-spa 封装	iframe + WebComponent	Webpack 运行时模块共享	浏览器原生
JS隔离	无内置，需自行实现	Proxy / Snapshot 沙箱	iframe 沙箱 + Proxy	依赖共享，无沙箱	浏览器原生隔离
CSS隔离	无内置，需自行实现	shadow DOM / 动态样式表	shadow DOM	CSS Modules / 命名约定	浏览器原生隔离
通信方式	props / 自定义事件	全局 State / props	props / window.parent / EventBus	JS 模块导入/导出	postMessage
易用性	较低，需自行封装	高，开箱即用	高，开箱即用	中，需 Webpack5	极高，但体验差
主要优点	灵活，定制性强	功能完善，社区成熟	隔离彻底，体验好	依赖共享，性能好	隔离最彻底
主要缺点	需做大量封装	全局弹窗处理不完美	兼容性（依赖 Proxy）	强依赖 Webpack5	URL 不同步，体验差

5.2. 核心问题解决方案总结

样式隔离

- **Shadow DOM**: 最常用的方案，天然隔离。但子应用中动态挂载到 `document.body` 的元素（如 Modal）样式会失效。
- **CSS Modules / BEM**: 在编译时生成唯一类名或通过命名规范避免冲突。对旧项目改造不友好。
- **CSS-in-JS**: 将 CSS 写入 JS，组件级别隔离。
- **PostCSS**: 编译时为所有 class 添加统一前缀。

JS 隔离

- **基于 Proxy 的沙箱**: 如 `qiankun`，通过代理 `window` 对象，拦截 `get/set` 操作，将全局变量的修改限制在沙箱内部。
- **基于 iframe 的沙箱**: 如 `wujie`，利用 `iframe` 提供的独立运行环境，并通过 `Proxy` 劫持 DOM 操作来改善体验。

公共依赖

- **Webpack Module Federation**: 运行时共享，是目前最优的方案之一，但有技术栈限制。
- **Webpack externals**: 将公共库从 bundle 中排除，通过 CDN 引入。需要依赖是 UMD 格式。
- **npm / lerna**: 将公共部分抽成 npm 包进行管理。更新和版本管理较为繁琐。

---

6. 面试常见问题

1. **常见的微前端解决方案有哪些?**
    - **Single-SPA**: 核心是路由管理，需要自行处理资源加载与隔离。
    - **Qiankun**: 基于 Single-SPA，内置沙箱隔离和资源预加载。
    - **Webpack Module Federation**: 模块联邦，用于依赖共享，减少冗余代码。

- **wujie**: 基于 iframe 和 WebComponent，隔离性好。
- **iframe**: 简单但存在通信和体验问题。

2. **有哪些样式隔离方案?**
   - **Shadow DOM**: 天然隔离，但可能影响第三方组件样式。
   - **CSS Modules**: 编译时生成唯一类名，适合 React/Vue 项目。
   - **BEM 命名空间**: 通过命名约定手动管理，维护成本高。
   - **动态样式表**: 子应用卸载时移除其样式表（Qiankun 采用）。
   - **CSS-in-JS**: 生成唯一类名，避免全局污染（如 styled-components）。

3. **子应用通信机制有哪些?**
   - **URL 参数传递**: 通过路由同步状态，适合简单数据。
   - **Props 传递**: 主应用向子应用传递数据和方法。
   - **自定义事件 (EventBus)**: 发布/订阅模式，实现应用间解耦。
   - **全局状态管理**: 如 Redux/Vuex 共享 store，需考虑版本兼容。
   - **Window.postMessage**: 用于跨窗口通信，需注意安全性。

4. **JS 沙箱机制如何实现?**
   - **Proxy 代理**: 创建一个虚拟的全局对象，拦截和隔离子应用的全局操作（如 `qiankun` 的 `ProxySandbox`）。
   - **快照（Snapshot）**: 在应用挂载前记录全局状态，卸载时恢复（如 `qiankun` 的 `SnapshotSandbox`）。
   - **iframe 沙箱**: 利用 `iframe` 的独立全局对象，但通信成本较高。

5. **如何优化微前端性能?**
   - **按需加载**: 路由触发时才动态加载子应用资源。
   - **共享公共依赖**: 使用 Module Federation 或 `externals` 共享 React、Vue 等基础库。
   - **预加载策略**: 在浏览器空闲时预加载子应用资源（`qiankun` 和 `wujie` 均支持）。
   - **缓存优化**: 利用 HTTP 缓存策略减少重复下载。
   - **应用保活**: 切换应用时不销毁，而是隐藏，下次进入时直接恢复（`wujie` 支持）。