微前端

1. 背景

随着技术的发展，前端应用承载的内容也日益复杂，基于此而产生的各种问题也应运而生，从MPA（Multi-Page Application，多页应用）到SPA（Single-Page Application，单页应用），虽然解决了切换体验的延迟问题，但也带来了首次加载时间长，以及工程爆炸增长后带来的巨石应用（Monolithic）问题；对于MPA来说，其部署简单，各应用之间天然硬隔离，并且具备技术栈无关、独立开发、独立部署等特点。要是能够将这两方的特点结合起来，会不会给用户和开发带来更好的用户体验？至此，在借鉴了微服务理念下，微前端便应运而生。

2. 什么是微前端

微前端的概念是由ThoughtWorks在2016年提出的，它借鉴了微服务的架构理念，核心在于将一个庞大的前端应用拆分成多个独立灵活的小型应用，每个应用都可以独立开发、独立运行、独立部署，再将这些小型应用融合为一个完整的应用，或者将原本运行已久、没有关联的几个应用融合为一个应用。微前端既可以将多个项目融合为一，又可以减少项目之间的耦合，提升项目扩展性，相比一整块的前端仓库，微前端架构下的前端仓库倾向于更小更灵活。

3. 微前端的价值

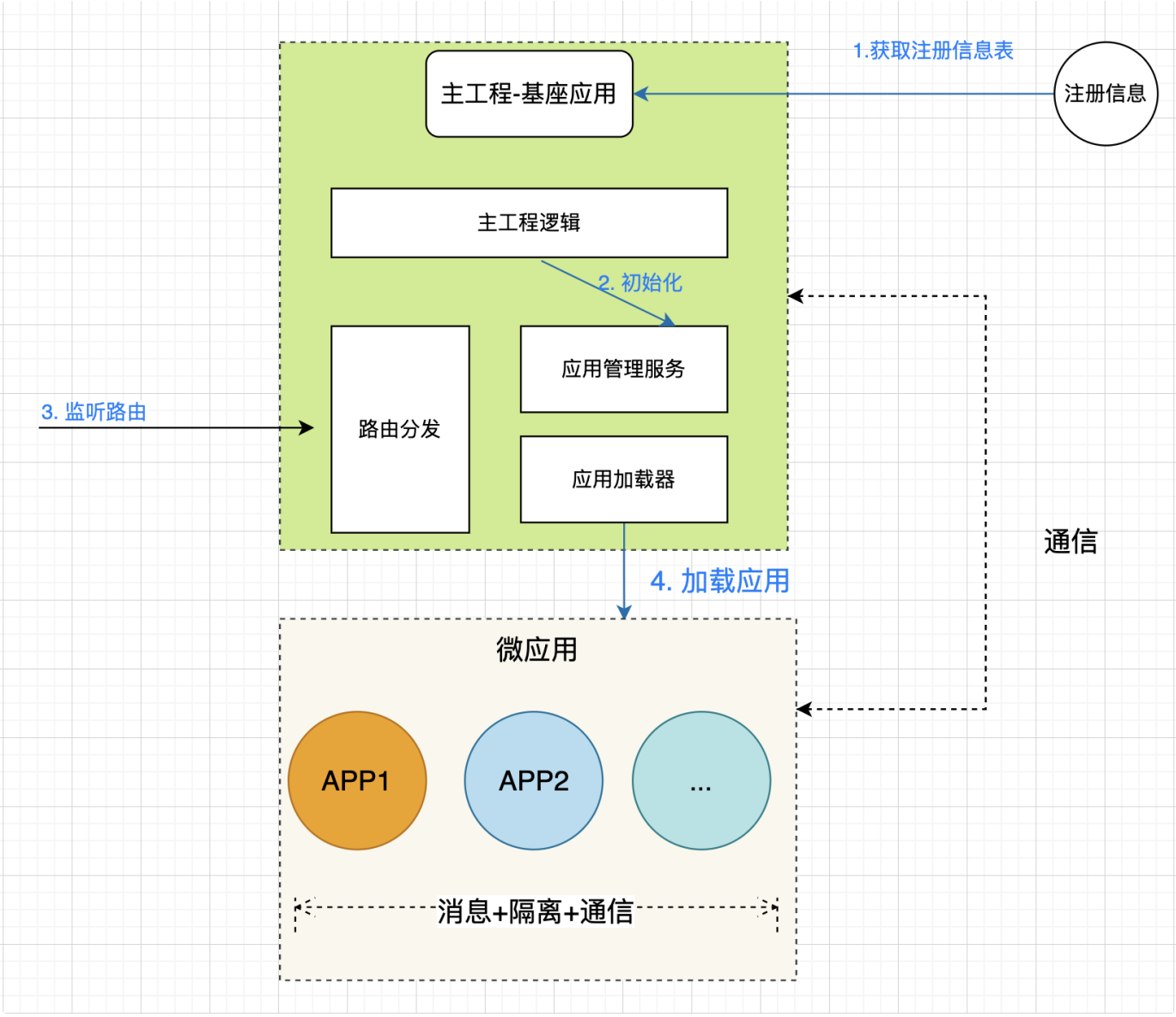
微前端架构具备以下几个核心价值：

- 技术栈无关 主框架不限制接入应用的技术栈，子应用具备完全自主权
- 独立开发、独立部署 子应用仓库独立，前后端可独立开发，部署完成后主框架自动完成同步更新
- 独立运行时 每个子应用之间状态隔离，运行时状态不共享

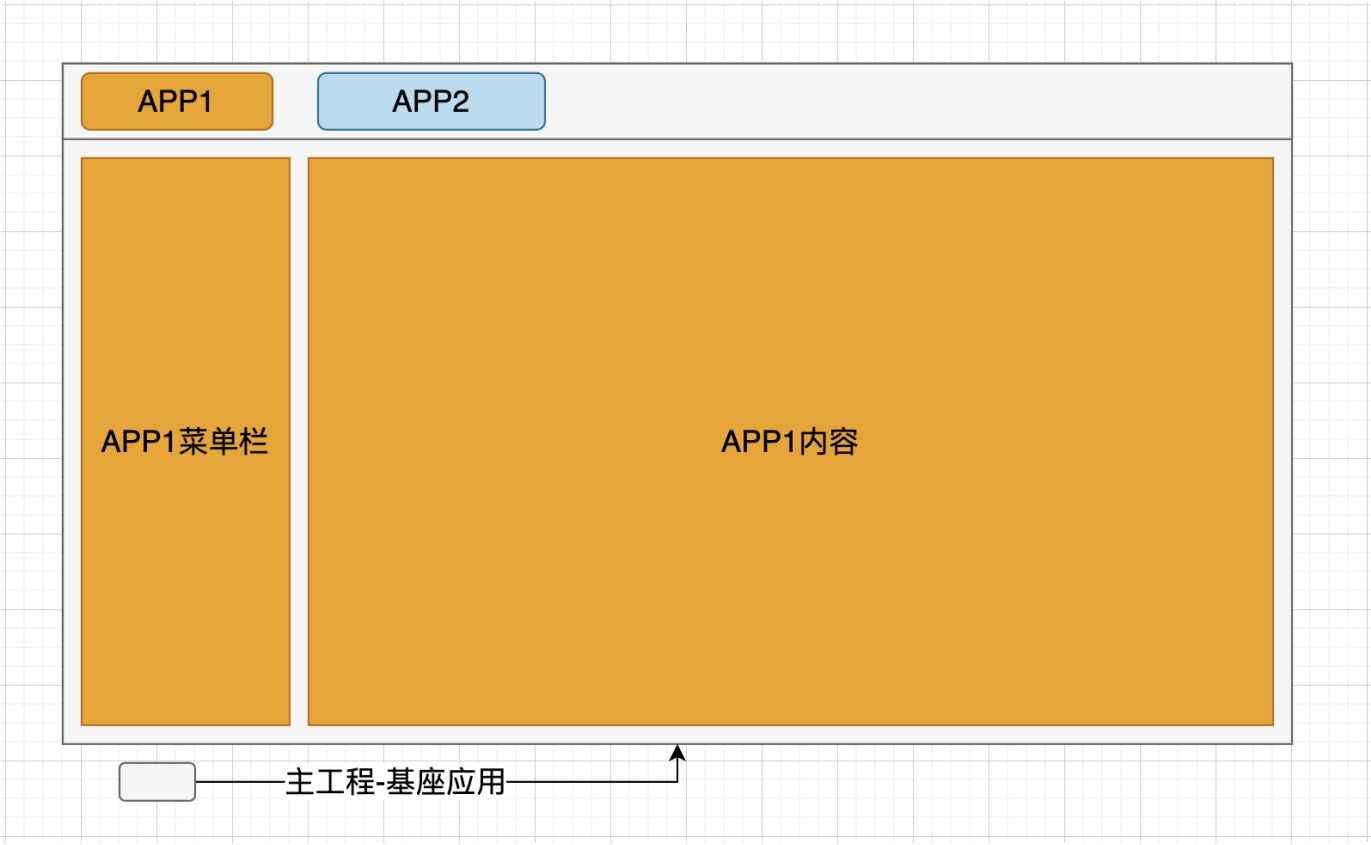
微前端架构旨在解决单体应用在一个相对长的时间跨度下，由于参与的人员、团队的增多、变迁，从一个普通应用演变成一个巨石应用后，随之而来的应用不可维护的问题。这类问题在企业级 Web 应用中尤其常见。

4. 微前端架构

当下微前端主要采用的是组合式应用路由方案，该方案的核心是“主从”思想，即包括一个基座（MainApp）应用和若干个微（MicroApp）应用，基座应用大多数是一个前端SPA项目，主要负责应用注册，路由映射，消息下发等，而微应用是独立前端项目，这些项目不限于采用React，Vue，Angular或者JQuery开发，每个微应用注册到基座应用中，由基座进行管理，但是如果脱离基座也是可以单独访问，基本的流程如下图所示：



当整个微前端框架运行之后，给用户的体验就是类似下图所示：



基座应用中有一些菜单项，点击每个菜单项可以展示对应的微应用，这些应用的切换是纯前端无感知的，很好的借鉴了SPA无刷新的特点

5. 微前端路由分发

作为微前端的基座应用，是整个应用的入口，负责承载当前微应用的展示和对其他路由微应用的转发，对于当前微应用的展示，一般是由以下几步构成：

1. 作为一个SPA的基座应用，本身是一套纯前端项目，要想展示微应用的页面除了采用iframe之外，要能先拉取到微应用的页面内容， 这就需要**远程拉取机制**。
2. 远程拉取机制通常会采用fetch API来首先获取到微应用的HTML内容，然后通过解析将微应用的JavaScript和CSS进行抽离，采用eval方法来运行JavaScript，并将CSS和HTML内容append到基座应用中留给微应用的展示区域，当微应用切换走时，同步卸载这些内容，这就构成的当前应用的展示流程。
3. 当然这个流程里会涉及到CSS样式的污染以及JavaScript对全局对象的污染，这个涉及到隔离问题会在后面讨论，而目前针对远程拉取机制这套流程，已有现成的库来实现，可以参考import-html-entry和system.js。

对于路由分发而言，以采用vue-router开发的基座SPA应用来举例，主要是下面这个流程：

1. 当浏览器的路径变化后，vue-router会监听hashchange或者popstate事件，从而获取到路由切换的时机。
2. 最先接收到这个变化的是基座的router，通过查询注册信息可以获取到转发到那个微应用，经过一些逻辑处理后，采用修改hash方法或者pushState方法来路由信息推送给微应用的路由，微应用可以是手动监听hashchange或者popstate事件接收，或者采用React-router，vue-router接管路由，后面的逻辑就由微应用自己控制。

6. 微前端的应用隔离

应用隔离问题主要分为主应用和微应用，微应用和微应用之间的JavaScript执行环境隔离，CSS样式隔离，我们先来说下CSS的隔离。

CSS隔离：当主应用和微应用同屏渲染时，就可能会有一些样式会相互污染，如果要彻底隔离CSS污染，可以采用CSS Module 或者命名空间的方式，给每个微应用模块以特定前缀，即可保证不会互相干扰，可以采用webpack的postcss插件，在打包时添加特定的前缀。

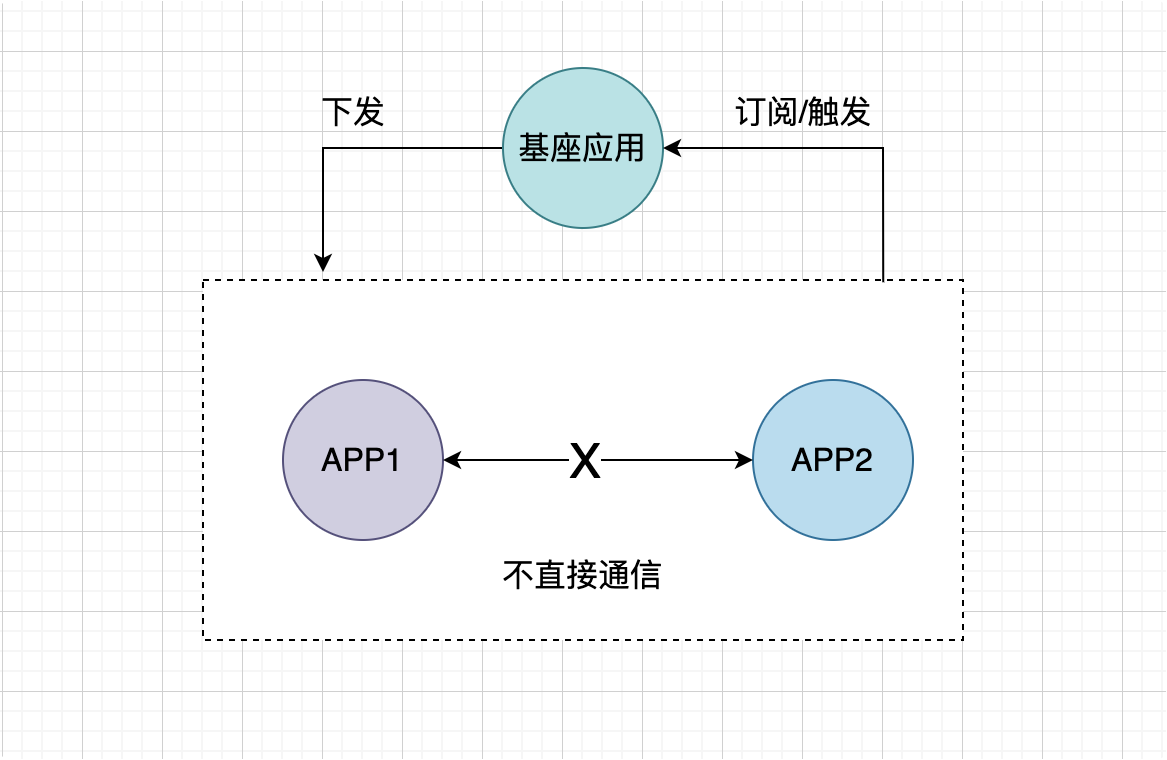
而对于微应用与微应用之间的CSS隔离就非常简单，在每次应用加载时，将该应用所有的link和style 内容进行标记。在应用卸载后，同步卸载页面上对应的link和style即可。

JavaScript隔离：每当微应用的JavaScript被加载并运行时，它的核心实际上是对全局对象Window的修改以及一些全局事件的改变，例如jQuery这个js运行后，会在Window上挂载一个window.$对象，对于其他库React，Vue也不例外。为此，需要在加载和卸载每个微应用的同时，尽可能消除这种冲突和影响，最普遍的做法是采用沙箱机制（SandBox）。

沙箱机制的核心是让局部的JavaScript运行时，对外部对象的访问和修改处在可控的范围内，即无论内部怎么运行，都不会影响外部的对象。通常在Node.js端可以采用vm模块，而对于浏览器，则需要结合with关键字和window.Proxy对象来实现浏览器端的沙箱。

7. 微前端的消息通信

应用间通信有很多种方式，当然，要让多个分离的微应用之间要做到通信，本质上仍离不开中间媒介或者说全局对象。所以对于消息订阅（pub/sub）模式的通信机制是非常适用的，在基座应用中会定义事件中心Event，每个微应用分别来注册事件，当被触发事件时再有事件中心统一分发，这就构成了基本的通信机制，流程如下图：



当然，如果基座和微应用采用的是React或者是Vue，是可以结合Redux和Vuex来一起使用，实现应用之间的通信。

8. 微前端的主要优势

- 技术兼容性好，各个子应用可以基于不同的技术架构
- 代码库更小、内聚性更强
- 便于独立编译、测试和部署，可靠性更高
- 耦合性更低，各个团队可以独立开发，互不干扰
- 可维护性和扩展性更好，便于局部升级和增量升级

关于**技术兼容性**，由于在被基座应用加载前， 所有子应用已经编译成原生代码输出，所以基座应用可以加载各类技术栈编写的应用；由于拆分后应用体积明显变小，并且每个应用只实现一个业务模块，因此其内聚性更强；另外子应用本身也是完整的应用，所以它可以**独立编译、测试和部署**；关于**耦合性**，由于各个子应用只负责各自的业务模块，所以耦合性很低，非常便于独立开发；关于**可维护性和扩展性**，由于拆分出的应用都是完整的应用，因此专门升级某个功能模块就成为了可能，并且当需要增加模块时，只需要创建一个新应用，并修改基座应用的路由规则即可。

9. 微前端的缺点

- 子应用间的资源共享能力较差，使得项目总体积变大
- 需要对现有代码进行改造（指的是未按照微前端形式编写的旧工程

首先，子应用之间保持较高的独立性，反而使一些**公共资源不便于共享**。虽然大型第三方库可以通过externals的方式上传到cdn，但像一些工具函数，通用业务组件等则不易共享，这就使得项目整体体积反而变大。由于改造成本不高，代码改造通常算不上很严重的问题，但仍存在一定的代价。

10. 是否要用微前端

微前端帮助开发者解决了实际的问题，但是对于每个业务来说，是否适合使用微前端，以及是否正确的使用微前端，还是需要遵循以下一些原则：

1. 微前端最佳的使用场景是一些B端的管理系统，既能兼容集成历史系统，也可以将新的系统集成进来，并且不影响原先的交互体验。
2. 整体的微前端不仅仅是只将系统集成进来，而是整个微前端体系的完善，这其中就包括：
   - 基座应用和微应用的自动部署能力。
   - 微应用的配置管理能力。
   - 本地开发调试能力。
   - 线上监控和统计能力等等。只有将整个能力体系搭建完善，才能说是整个微前端体系流程的完善。
3. 当发现使用微前端反而使效率变低，简单的变更复杂那就说明微前端并不适用。