2021-11-15 Richard技术分享

微前端简介

背景

随着这些年互联网的飞速发展，很多企业的 web 应用在持续迭代中功能越来越复杂，参与的人员、团队不断增多，导致项目出现难以维护的问题，这种情况 PC 端尤其常见，许多研发团队也在找寻一种高效管理复杂应用的方案，于是微前端被提及的越来越频繁。

微前端并不是一项新的技术，而是一种**微服务架构**理念，它将单一的 web 应用拆解成多个可以独立开发、独立运行、独立部署的小型应用，并将它们整合为一个应用。

微服务架构带来了哪些好处?

假设服务边界已经被正确地定义为可独立运行的业务领域，并确保在微服务设计中遵循诸多最佳实践。那么至少会以下几个方面获得显而易见的好处：

- **复杂性**：服务可以更好地分离，每一个服务都足够小，完成完整的定义清晰的职责；
- **扩展性**：每一个服务可以独立横向扩展以满足业务伸缩性，并减少资源的不必要消耗；
- **灵活性**：每一个服务可以独立失败，允许每个团队决定最适合他们的技术和基础架构；
- **敏捷性**：每一个服务都可以独立开发，测试和部署，并允许团队扩展独立部署和维护服务的交付。

那么前端的现状呢?

在前端，往往由一个前端团队创建并维护一个 Web 应用程序，使用 REST API 从后端服务获取数据。这种方式如果做得好的话，它能够提供优秀的用户体验，但最主要的缺点就是单页面应用（SPA）不能很好地扩展和部署。在一个大公司里，单前端团队可能成为一个发展瓶颈。随着时间的推移，往往由一个独立团队所开发的前端层越来越难以维护。

特别是这么一个特性丰富、功能强大的前端 Web 应用程序，却位于后端微服务架构之上。并且随着业务的发展，前端变得越来越**臃肿**，一个项目可能会有 90% 的前端代码，却只有非常薄的后端，甚至这种情况在 **Serverless** 架构的背景下还会愈演愈烈。

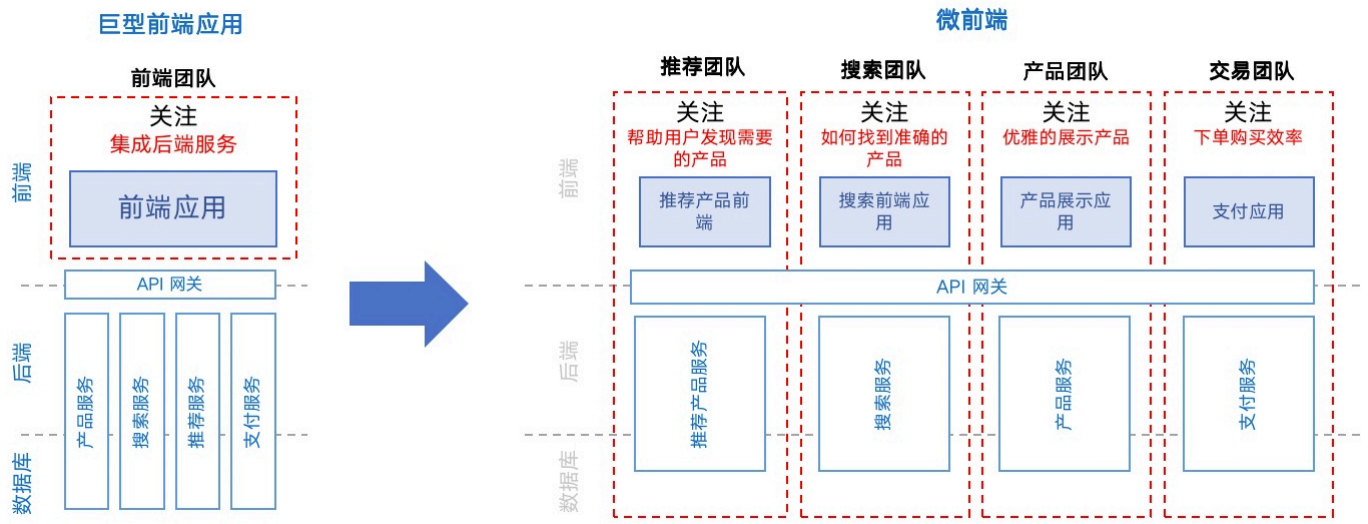
什么是微前端?

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简单。

微前端（Micro Frontend）这个术语其实就是微服务的衍生物。将微服务理念扩展到前端开发，同时构建多个完全自治的和松耦合的 App 模块（服务），其中每个 App 模块只负责特定的 UI 元素和功能。

它主要解决了两个问题：

1. 随着项目迭代应用越来越庞大，难以维护。
2. 跨团队或跨部门协作开发项目导致效率低下的问题。



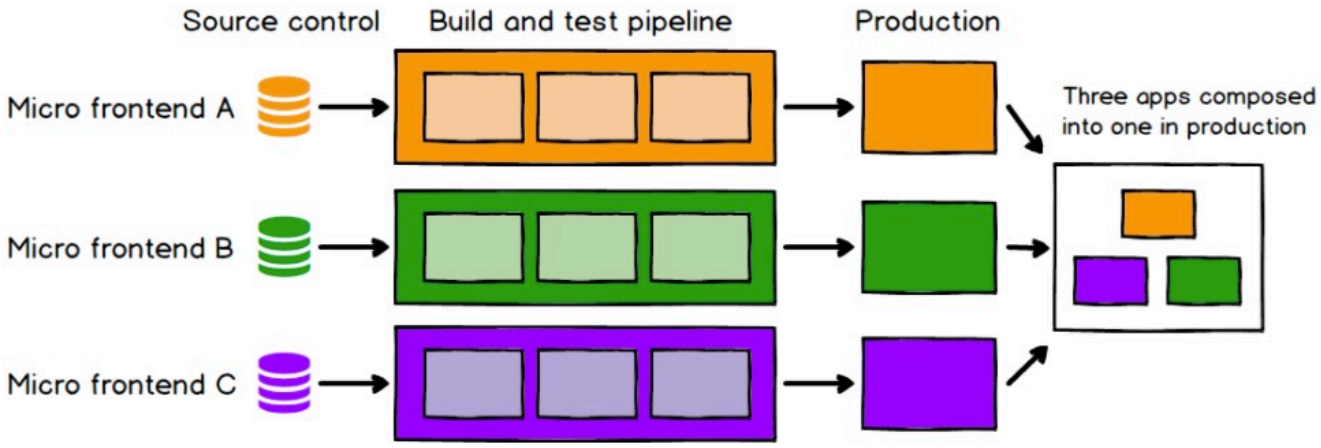
巨型前端应用

- 按技能划分团队，**不理解业务**
- 所有服务集成在单一前端，**稳定性低**
- 应用不断增长成为巨型前端，**维护成本高**
- 已开发的功能无法重用，**开发成本高**

微前端

- 微应用可快速整合**缩短开发时间**
- 更小的代码仓库**易于维护**
- 更小的功能集合易于测试，**提高稳定性**
- 按业务划分团队，更**专注业务领域开发**
- 微应用可独立开发构建，可**独立运维**

研发效率



- 不同团队独立的发布运维；
- 不同团队间独立迭代和自主创新；
- 可根据业务线形成从前到后的研发团队。

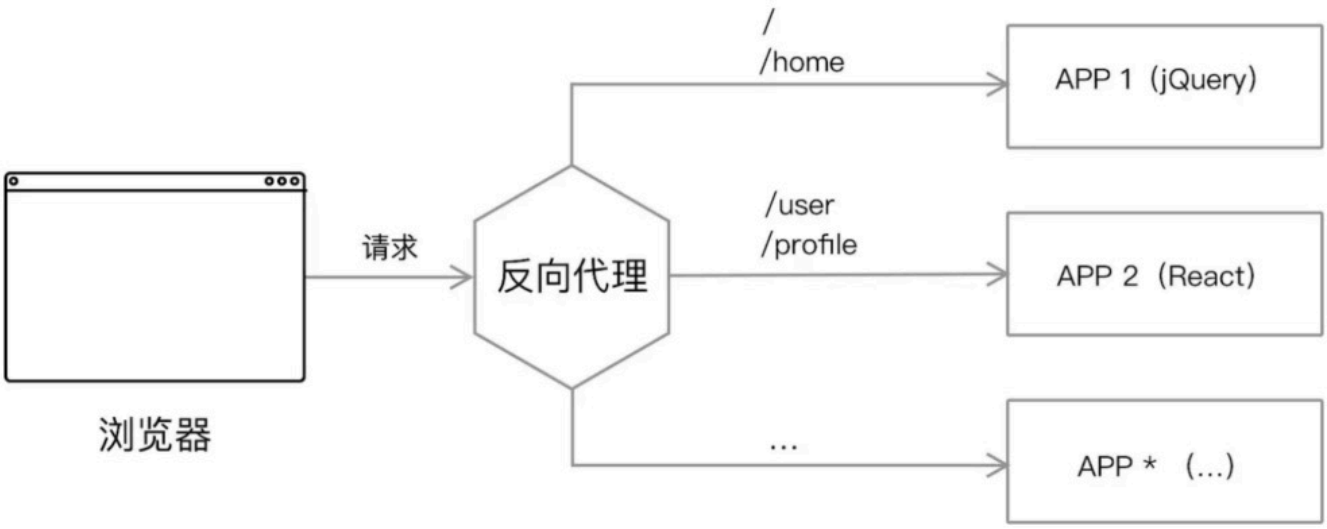
微服务实现方式

从技术实践上，微前端架构可以采用以下的几种方式进行：

- **路由分发式**：通过 HTTP 服务器的反向代理功能，来将请求路由到对应的应用上；
- **前端微服务化**：在不同的框架之上设计通讯、加载机制，以在一个页面内加载对应的应用；
- **微应用**：通过软件工程的方式，在部署构建环境中，组合多个独立应用成一个单体应用；
- **前端容器化**：通过将 iFrame 作为容器，来容纳其它前端应用；
- **应用组件化**：借助于 Web Components 技术，来构建跨框架的前端应用。

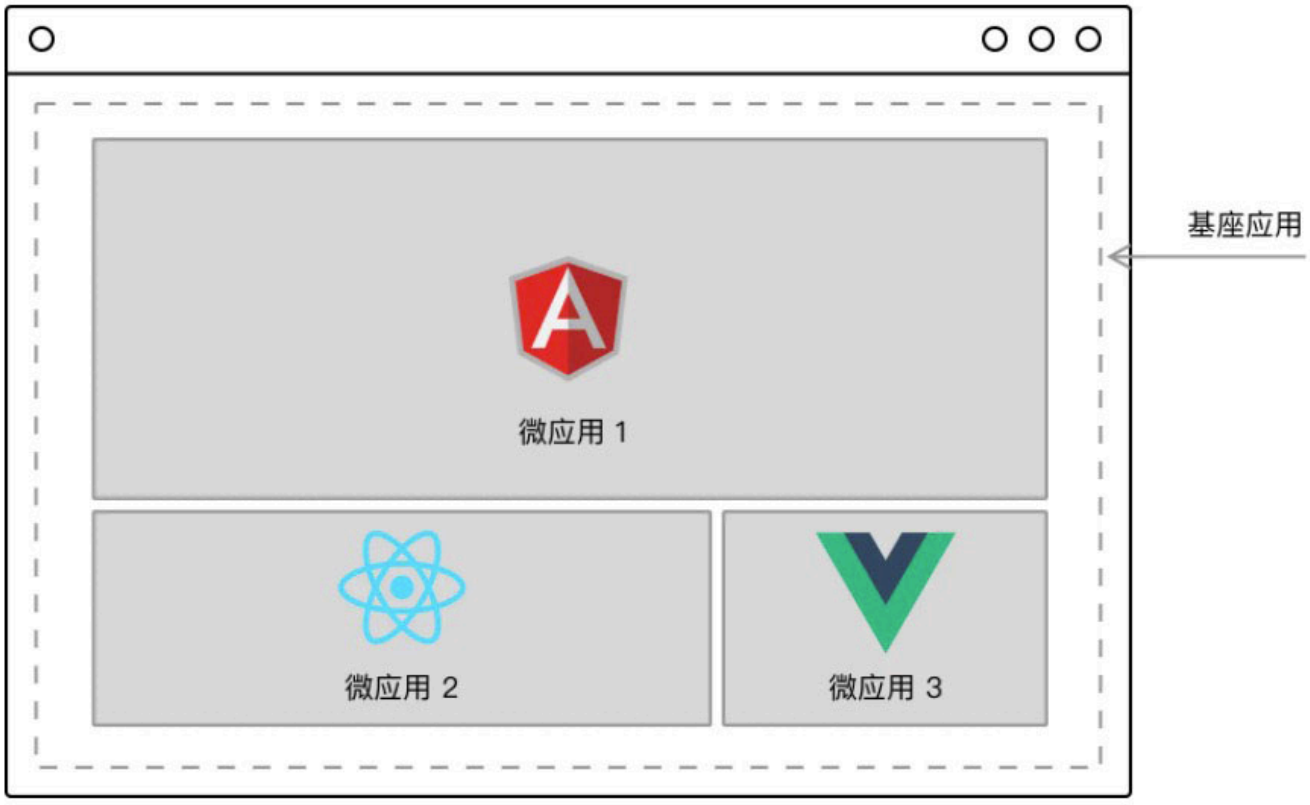
路由分发式

路由分发式微前端，即通过路由将不同的业务分发到不同的、独立前端应用上。 其通常可以通过 HTTP 服务器的反向代理来实现，又或者是应用框架自带的路由来解决。如下图所示：



前端微服务化

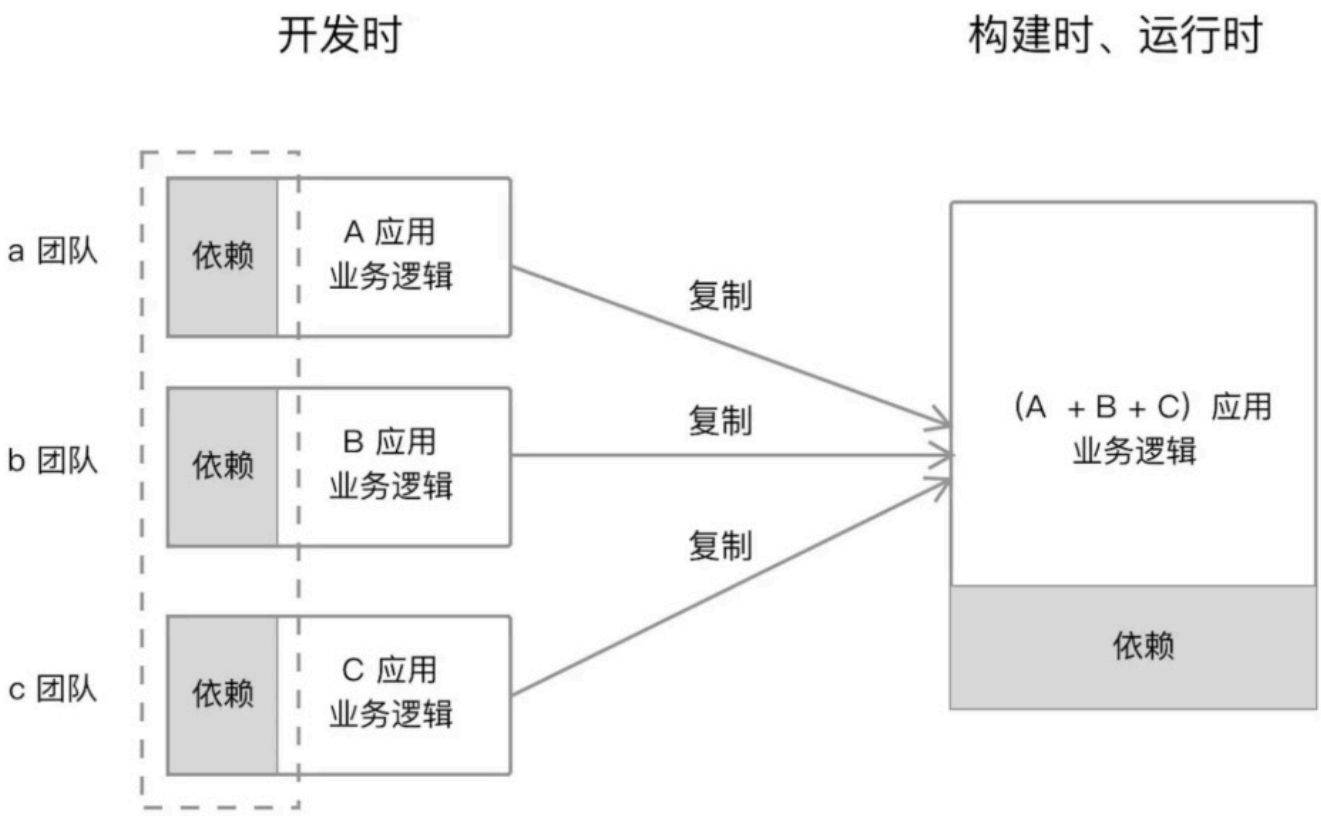
前端微服务化，是微服务架构在前端的实施，每个前端应用都是完全独立（技术栈、开发、部署、构建独立）、自主运行的，最后通过模块化的方式组合出完整的前端应用。 其架构如下图所示：



采用这种方式意味着，一个页面上同时存在二个及以上的前端应用在运行。而路由分发式方案，则是一个页面只有唯一一个应用。

组合式集成：微应用化

微应用化，即在开发时，应用都是以单一、微小应用的形式存在，而在运行时，则通过构建系统合并这些应用，组合成一个新的应用。 其架构如下图所示：



微应用化更多的是以软件工程的方式，来完成前端应用的开发，因此又可以称之为组合式集成。对于一个大型的前端应用来说，采用的架构方式，往往会是通过业务作为主目录，而后在业务目录中放置相关的组件，同时拥有一些通用的共享模板。

前端容器：iframe

iframe 作为一个非常 “古老” 的，人人都觉得普通的技术，却一直很管用。它能有效地将另一个网页 / 单页面应用嵌入到当前页面中，两个页面间的 CSS 和 JavaScript 是相互隔离的 —— 除去 iframe 父子通信部分的代码，它们之间的代码是完全不相互干扰的。iframe 便相当于是创建了一个全新的独立的宿主环境，类似于沙箱隔离，它意味着前端应用之间可以相互独立运行。

结合 Web Components 构建

Web Components 是一套不同的技术，允许开发者创建可重用的定制元素（它们的功能封装在代码之外），并且在 web 应用中使用它们。

为什么不用 iframe?

如果从业务拆分、应用隔离的视角来看，那 iframe 就是最简单的微前端基石方案。天然隔离、天然沙箱，几乎所有微前端方案都会被挑战这样一个问题：为什么不用 iframe？iframe 最大的特性就是提供了浏览器原生的硬隔离方案，不论是样式隔离，还是 js 隔离，这类问题通通都能被完美解决。

但是最终所有现代化的微前端方案又不约而同地放弃了 iframe。iframe 最大的问题也正来自于它优异的隔离性，彻底隔绝了应用间上下文，导致内存变量无法被共享，又产生了一些开发体验、产品体验的问题。

iframe 优点：

- **框架无关**：iframe 只加载部署完应用链接；
- **独立沙箱**：iframe 拥有浏览器的独立沙箱，子应用和主应用完全不用考虑 js & css 污染问题；
- **开发简单**：仅仅一个 iframe 标签。

iframe 的一些痛点：

- **UI 体验不好**：父子之间的 UI 难以同步，内嵌的 iframe 并不会自动调节宽高，一旦需要通信交互就难受了；iframe 仅在制定的渲染块内渲染，而子应用的一些类似于带遮罩层的 Modal 等全局的 UI 组件只会在 iframe 内呈现；
- **通信困难**：iframe 强大的沙箱机制带来的副作用就是父子应用通信困难，全局上下文完全隔离，内存变量不共享，仅仅一个 postmessage API，跨域 cookie、Promise 等等都难以实现；
- **路由丢失**：子应用的 url 改变无法及时同步父应用，随着页面的刷新，子应用的路由状态丢失，后退、前进按钮无法使用；
- **加载时间长**：每次子应用进入都是一次浏览器上下文重建、资源重新加载的过程。

qiankun 和 emp 为啥受欢迎?

由于业务场景更为复杂，业务规模更为庞大，所以国内在微前端的完整解决方案上处于绝对的领先优势。通常用 qiankun 和 emp 这两个方案。

从现代化前端角度去思考 “微前端”，可以想到下图这样几个方向。

方案一：qiankun

在微前端界，`qiankun` 算得上是最早成型且知名度最广的框架了，它是真正意义上的单页微前端框架，那么 `qiankun` 到底有哪些特点呢，在其官网中找到了如下概括：

- **基于**：`single-spa` 封装，提供了更加开箱即用的 API；
- **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/JQuery 还是其他等框架；
- **HTML Entry 接入方式**：让你接入微应用像使用 iframe 一样简单；
- **样式隔离**：确保微应用之间样式互相不干扰；
- **JS 沙箱**：确保微应用之间 全局变量 / 事件 不冲突；
- **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，加速微应用打开速度；
- **umi 插件**：提供了 @umijs/plugin-qiankun 供 umi 应用一键切换成微前端架构系统。

如果用一个词概括 qiankun 的话，那就是 “基座”。所有应用都注册在基座上，通过基座应用来监听路由，并按照路由规则来加载不同的应用，以实现应用间解耦。

接下来从应用加载、沙箱隔离与内部通信三个方向解析 qiankun。

应用加载

qiankun 通过 html 文件为应用入口，然后通过一个 html 解析器从文件中提取 js 和 css 依赖，并通过 fetch 下载依赖，于是在 qiankun 中你可以这样配置入口：

```
const MicroApps = [{
  entry: '',
  container: '#root',
  activeRule: '/app_1'
}]
```

qiankun 会通过 import-html-entry 请求 http://example.com，得到对应的 html 文件，解析内部的所有 script 和 style 标签，依次下载和执行它们，这使得应用加载变得更易用，同时能确保在隔离的沙箱环境中执行。

沙箱隔离

在沙箱中要保证 JavaScript 与 CSS 的隔离。针对 JavaScript，qiankun 使用了 proxy 与快照两个模式来处理隔离。

首先来看 Proxy。把解析所得的 script 脚本，用 with (window){} 包裹起来，然后把 window.proxy 作为函数的第一个参数传进来，所以 with 语法内的 window 实际上是 window.proxy。

这样，在执行这段代码时，所有类似 var name = '张三' 这样的语句添加的全局变量 name，实际上是被挂载到了 window.proxy 上，而不是真正的全局 window 上。当应用被卸载时，对应的 proxy 会被清除，因此不会导致 js 污染。

而在 IE 11 等不支持 Proxy 特性的浏览器上，启用快照模式来保证兼容。它的大致思路是，在加载应用前，将 window 上的所有属性保存起来（即拍摄快照）；当应用被卸载时，再恢复 window 上的所有属性，这样也可以防止全局污染。但是当页面同时存在多个应用实例时，qiankun 无法将其隔离开，所以 IE 下的快照策略无法支持多实例模式。

CSS 同样有两套方案，一套是基于 Shadow DOM 完成，另一套类似 Vue 的 Scope 属性。在 Shadow DOM 启用 strictStyleIsolation 时，qiankun 将采用 shadowDom 的方式进行样式隔离，即为子应用的根节点创建一个 shadow root，最终整个应用的所有 DOM 将形成一棵 shadow tree。

我们知道，Shadow DOM 的特点是，它内部所有节点的样式对树外面的节点无效，因此自然就实现了样式隔离。但是这种方案是存在缺陷的，因为某些 UI 框架可能会生成一些弹出框直接挂载到 document.body 下，此时由于脱离了 shadow tree，所以它的样式仍然会对全局造成污染。

另一套 scoped 的特性也并非完美，为子应用的根节点添加一个特定的随机属性，开启后就像这样：

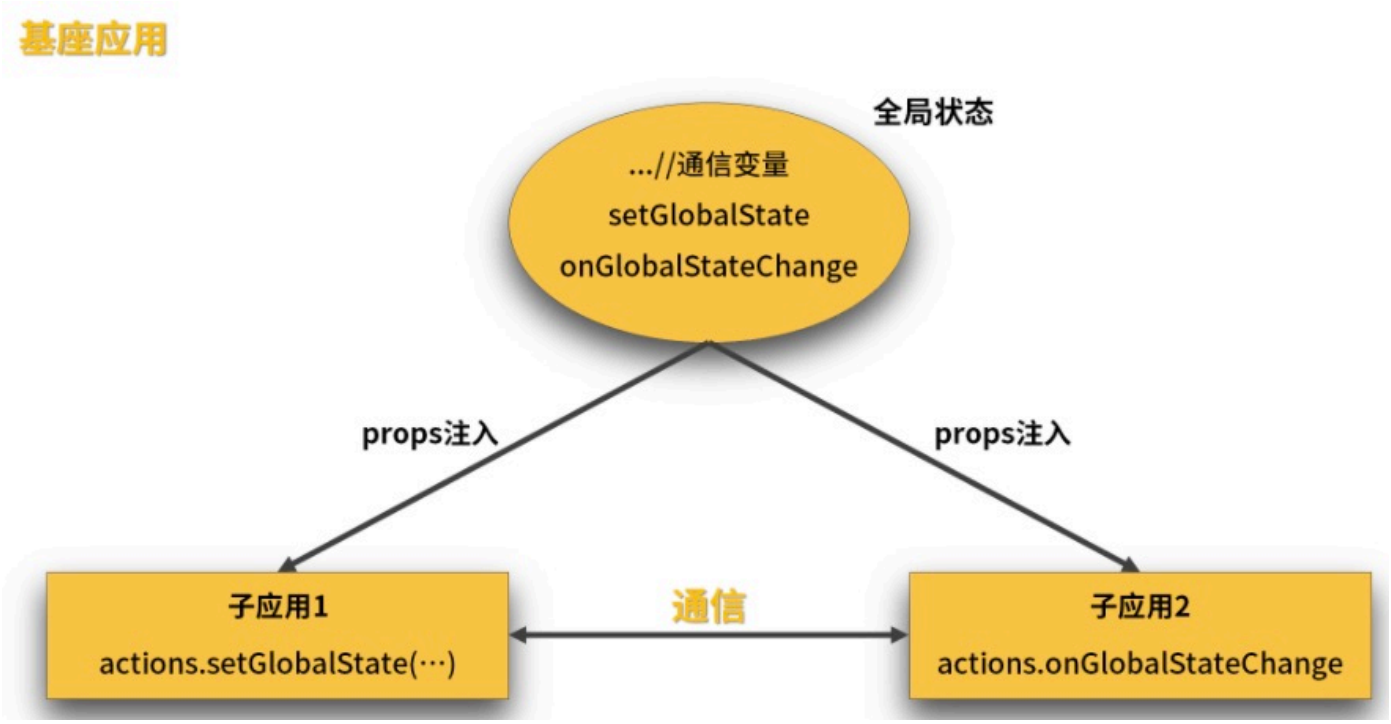
```
// 假设应用名是 react16
.app-main {
  font-size: 14px;
}

div[data-qiankun-react16] .app-main {
  font-size: 14px;
}
```

虽然有用，但对 `@keyframes`、`@font-face`、`@import`、`@page` 的支持还存在适配问题。

内部通信

qiankun 提供了一个简要的方案，思路是基于一个全局的 globalState 对象。这个对象由基座应用负责创建，内部包含一组用于通信的变量，以及两个分别用于修改变量值和监听变量变化的方法：setGlobalState 和 onGlobalStateChange。



简而言之，就是通过订阅全局变量的修改状态实现通信，是标准的订阅 - 发布模式。

qiankun 在 2.0 版本的时候提供了最简单的基于 props 通信的 API，主要解决父子应用强耦合时的通信。

所以说，**qiankun 总体上属于轻量级微前端框架，它的接口设计简约，功能丰富**。

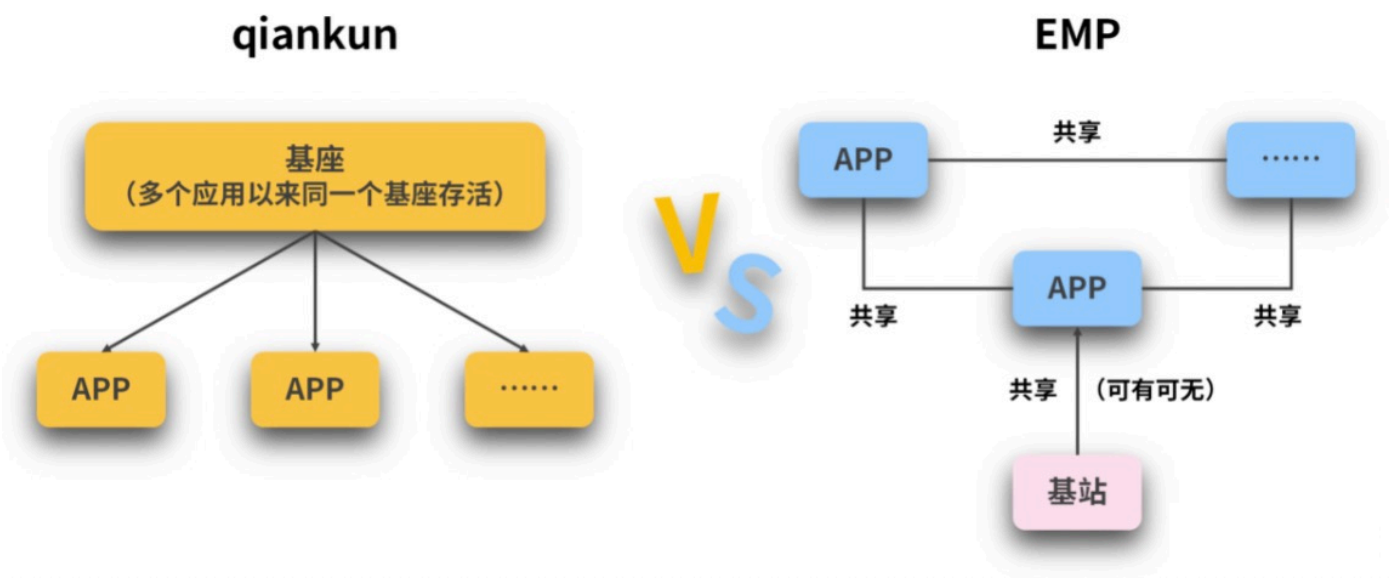
方案二：EMP

EMP 是由欢聚时代业务中台自主研发的最年轻的**单页微前端解决方案**。EMP 基于 Webpack 5 的 Module Federation 实现，用一个词概括，就是 **“去中心化”**。

那么，他有哪些特性呢，下面我们一起看看：

- **基于**`Webpack5`的新特性`Module Federation`实现，达到**第三方依赖共享，减少不必要的代码引入**的目的；
- **每个微应用独立部署运行**：并通过 cdn 的方式引入主程序中，因此只需要部署一次，便可以提供给任何基于`Module Federation`的应用使用。并且此部分代码是远程引入，无需参与应用的打包；
- **动态更新微应用**：`EMP`是通过`cdn`加载微应用，因此每个微应用中的代码有变动时，无需重新打包发布新的整合应用便能加载到最新的微应用；
- **去中心化**：每个微应用间都可以引入其他的微应用，无中心应用的概念；
- **跨技术栈组件式调用**：提供了在主应用框架中可以调用其他框架组件的能力；
- **按需加载**：开发者可以选择只加载微应用中需要的部分，而不是强制只能将整个应用全部加载；
- **应用间通信**：每一个应用都可以进行状态共享，就像在使用 npm 模块进行开发一样便捷；
- **生成对应技术栈模板**：它能像`create-react-app`一样，也能像`create-vue-app`一样，通过指令一键搭建好开发环境，减少开发者的负担；
- **远程拉取 ts 声明文件**：`emp-cli`中内置了拉取远程应用中代码声明文件的能力，让使用 ts 开发的开发者不再为代码报错而烦恼。

在 qiankun 的模式下，通过中心基座集成各微应用。而在 emp 的方案中不需要中心化的基座，每一个微前端应用都可以通过远程调用的方式引入共享模块。那它这里是怎么实现的呢？让我们来看一下 Webpack 5 中最重要的特性 Module Federation。



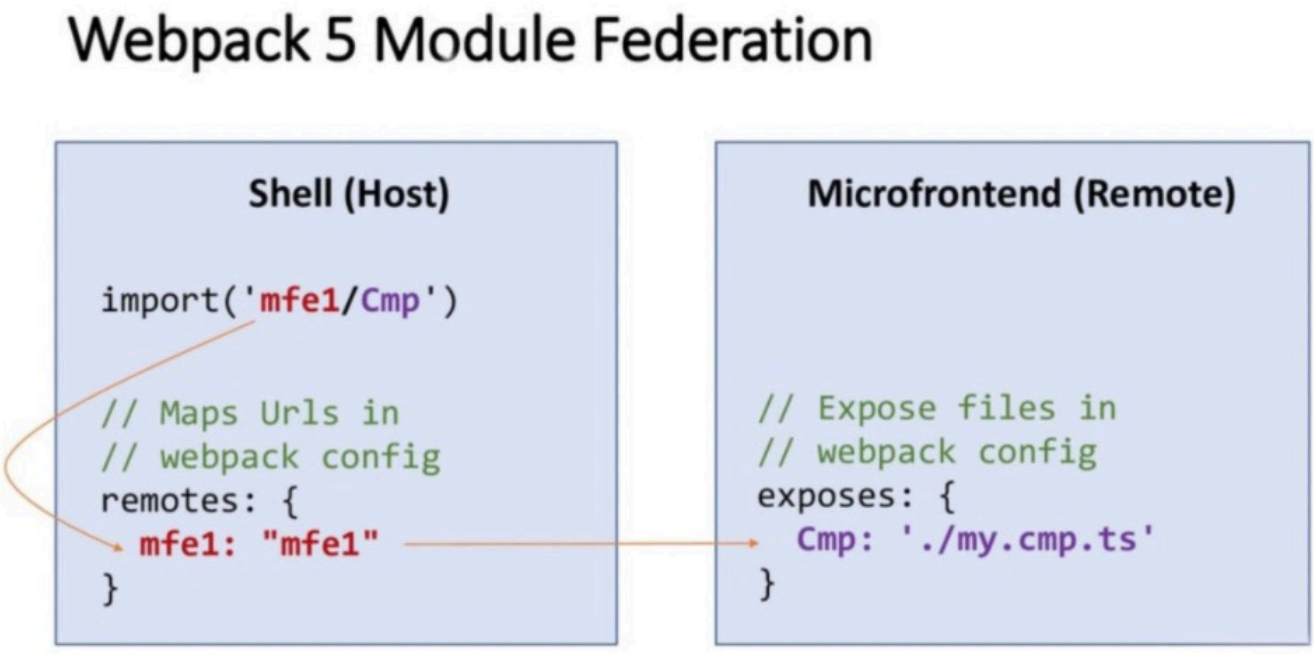
Module Federation 模块联邦

它的目的是将多个独立构建的应用组成一个应用程序。由此，Module Federation 提供了在当前应用中远程加载其他服务器上应用的能力。这就非常接近后端 “微服务” 的概念了。

这里它又可以细化出两个概念：

- **host**：引用了其他应用的应用；
- **remote**：被其他应用所使用的应用。

举一个实际的例子可能更好理解，如下图所示。我们在项目 B 的 Webpack 的 exposes（host） 字段中暴露一个模块叫 Cmp，而在项目 A 的 Webpack 的 remotes 字段中注册远程的模块，然后在使用时，通过 import 完成导入。



集成

对于新项目，直接使用 emp 开发是毫无压力的，有 emp cli 工具直接可以完成开发。对老项目则需要迁移升级开发工具链，例如 Webpack 至少需要升级到 Webpack 5，然后再与 emp cli 相适应调教，整体过程相较于 qiankun 要烦琐许多。

方案对比

在了解了 iframe、qiankun 及 emp 后，三者直接的特点与缺点，如下表所示。

解决方案	相对特点	缺点
iframe	天生隔离样式与脚本、多页	不是单页应用，会导致浏览器刷新 iframe url 状态丢失、后退前进按钮无法使用
弹框类的功能无法应用到整个大应用中，只能在对应的窗口内展示		
由于可能应用间不是在相同的域内，主应用的 cookie 要透传到根域名都不同的子应用中才能实现免登录效果		
每次子应用进入都是一次浏览器上下文重建、资源重新加载的过程，占用大量资源的同时也在极大地消耗资源		
iframe 的特性导致搜索引擎无法获取到其中的内容，进而无法实现应用的 seo		
qiankun	HTML Entry 接入方式	css 隔离方案并不完美
资源预加载		
EMP	目前无法涵盖所有框架	动态更新微应用
去中心化		

跨技术栈组件式调用		
按需加载		
应用间通信		
生成对应技术栈模板		
远程拉取 ts 声明文件		

微前端架构落地

落地路线图

微前端拆的不只是代码，也不只是工程，而是业务。这一块儿需要与 Leader 有充分沟通，如何完成团队内的业务划分，依照业务边界确定拆分职责。最后团队划分成什么样，业务就划分成什么样，微前端也就划分成什么样。

业务拆分

与微服务类似，要划分不同的前端边界，不是一件容易的事。就当前而言，以下几种方式是常见的划分微前端的方式：

- **按照业务拆分**。
- **按照权限拆分**。
- **按照变更的频率拆分**。
- **按照组织结构拆分**。利用康威定律来进一步拆分前端应用。
- **跟随后端微服务划分**。实践证明， DDD 与事件风暴是一种颇为有效的后端微前端拆分模式，对于前端来说，

它也颇有有效 —— 直接跟踪后端服务。

每个项目都有自己特殊的背景，切分微前端的方式便不一样。即使项目的类型相似，也存在一些细微的差异。

代码侧改造

如果使用 qiankun，那么代码侧并不需要改造太多。

首先需要构建主应用基座，注册相关应用。

主应用

1. 安装 qiankun

```
yarn add qiankun # 或者 npm i qiankun -S
```

1. 在主应用中注册微应用

```
import { registerMicroApps, start } from 'qiankun';

registerMicroApps([
  {
    name: 'react app', // app name registered
    entry: '//localhost:7100',
    container: '#yourContainer',
    activeRule: '/yourActiveRule',
  },
  {
    name: 'vue app',
    entry: { scripts: ['//localhost:7100/main.js'] },
    container: '#yourContainer2',
    activeRule: '/yourActiveRule2',
  },
]);

start();
// start({ strictStyleIsolation: true });
// start({strictStyleIsolation: true, experimentalStyleIsolation: true});
```

当微应用信息注册完之后，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。

如果微应用不是直接跟路由关联的时候，你也可以选择手动加载微应用的方式：

```
import { loadMicroApp } from 'qiankun';

loadMicroApp({
  name: 'app',
  entry: '//localhost:7100',
  container: '#yourContainer',
});
```

微应用

微应用不需要额外安装任何其他依赖即可接入 qiankun 主应用。

1. 导出相应的生命周期钩子

微应用需要在自己的入口 js (通常就是你配置的 webpack 的 entry js) 导出 `bootstrap`、`mount`、`unmount` 三个生命周期钩子，以供主应用在适当的时机调用。

```
/**
 * bootstrap 只会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。
 * 通常我们可以在这里做一些全局变量的初始化，比如不会在 unmount 阶段被销毁的应用级别的缓存等。
 */
export async function bootstrap() {
  console.log('react app bootstraped');
}

/**
 * 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
 */
export async function mount(props) {
  ReactDOM.render(<App />, props.container ? props.container.querySelector('#root') : document.getElementById('root'));
}

/**
 * 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
 */
export async function unmount(props) {
  ReactDOM.unmountComponentAtNode(
    props.container ? props.container.querySelector('#root') : document.getElementById('root'),
  );
}

/**
 * 可选生命周期钩子，仅使用 loadMicroApp 方式加载微应用时生效
 */
export async function update(props) {
  console.log('update props', props);
}
```

> qiankun 基于 single-spa，所以你可以在这里找到更多关于微应用生命周期相关的文档说明。无 webpack 等构建工具的应用接入方式请见这里

1. 配置微应用的打包工具

除了代码中暴露出相应的生命周期钩子之外，为了让主应用能正确识别微应用暴露出来的一些信息，微应用的打包工具需要增加如下配置：

webpack:

```
const packageName = require('./package.json').name;

module.exports = {
  output: {
    library: `${packageName}-[name]`,
    libraryTarget: 'umd',
    jsonpFunction: `webpackJsonp_${packageName}`,
  },
};
```

> 相关配置介绍可以查看 webpack 相关文档。

到这里微前端的改造工作基本就完成了，实际上接入微前端框架，只是微前端工程化工作的开始，完成代码交付是工作中的一部分，收集相关数据完成汇报工作，则是最重要的一部分。

例如，如何保障线上的基座稳定性与子应用稳定性保障？如何对线上子应用进行版本控制，完成质量管控？如何执行动态配置下发协助子应用提供 A/B 测试？

作为基座，需要在容器层提供丰富的数据监控及能力供给提升子应用的健壮性。

微前端的优缺点

微前端的优点

- 敏捷性 - 独立开发和更快的部署周期：
  - 开发团队可以选择自己的技术并及时更新技术栈。
  - 一旦完成其中一项就可以部署，而不必等待所有事情完毕。
- 降低错误和回归问题的风险，相互之间的依赖性急剧下降。
- 更简单快捷的测试，每一个小的变化不必再触碰整个应用程序。
- 更快交付客户价值，有助于持续集成、持续部署以及持续交付。
- 维护和 bugfix 非常简单，每个团队都熟悉所维护特定的区域。
- 应用自治。只需要遵循统一的接口规范或者框架，以便于系统集成到一起，相互之间是不存在依赖关系的。
- 单一职责。每个前端应用可以只关注于自己所需要完成的功能。
- 技术栈无关。你可以使用 Angular 的同时，又可以使用 React 和 Vue。

微前端的缺点

- 应用的拆分基础依赖于基础设施的构建，一旦大量应用依赖于同一基础设施，那么维护变成了一个挑战。
- 拆分的粒度越小，便意味着架构变得复杂、维护成本变高。
- 技术栈一旦多样化，便意味着技术栈混乱
- 开发与部署环境分离
  - 本地需要一个更为复杂的开发环境。
  - 每个 App 模块有一个孤立的部署周期。
  - 最终应用程序需要在**同一个**孤立的环境中运行。
- 复杂的集成

    - 需要考虑隔离 JS，避免 CSS 冲突，并考虑按需加载资源
    - 处理数据获取并考虑用户的初始化加载状态
    - 如何有效测试，微前端模块之间的 Contract Testing?
- 第三方模块重叠
    - 依赖冗余增加了管理的复杂性
    - 在团队之间共享公共资源的机制
- 影响最终用户的体验
    - 初始 Loading 时间可能会增加
    - HTML 会需要服务器端的渲染

你可能并不需要微前端

满足以下几点，你可能就不需要微前端

1. 你 / 你的团队 具备系统内所有架构组件的话语权
   简单来说就是，系统里的所有组件都是由一个小的团队开发的。
2. 你 / 你的团队 有足够动力去治理、改造这个系统中的所有组件
   直接改造存量系统的收益大于新老系统混杂带来的问题。
3. 系统及组织架构上，各部件之间本身就是强耦合、自洽、不可分离的
   系统本身就是一个最小单元的「架构量子」，拆分的成本高于治理的成本。
4. 极高的产品体验要求，对任何产品交互上的不一致零容忍
   不允许交互上不一致的情况出现，这基本上从产品上否决了渐进式升级的技术策略。

满足以下几点，你才确实可能需要微前端

1. 系统本身是需要集成和被集成的 一般有两种情况：
    1. 旧的系统不能下，新的需求还在来。
       没有一家商业公司会同意工程师以单纯的技术升级的理由，直接下线一个有着一定用户的存量系统的。而你大概又不能简单通过 iframe 这种「靠谱的」手段完成新功能的接入，因为产品说需要「弹个框弹到中间」
    2. 你的系统需要有一套支持动态插拔的机制。
       这个机制可以是一套精心设计的插件体系，但一旦出现接入应用或被接入应用年代够久远、改造成本过高的场景，可能后面还是会过渡到各种微前端的玩法。
2. 系统中的部件具备足够清晰的服务边界
   通过微前端手段划分服务边界，将复杂度隔离在不同的系统单元中，从而避免因熵增速度不一致带来的代码腐化的传染，以及研发节奏差异带来的工程协同上的问题。

参考

- 可能是你见过最完善的微前端解决方案
- 实施微前端的六种方式
- 微前端如何落地?
- qiankun 和 emp 在国内微前端中为啥这么受欢迎?
- https://qiankun.umijs.org/zh/guide/
- https://martinfowler.com/articles/micro-frontends.html
- https://developer.mozilla.org/zh-CN/docs/Web/Web_Components/Using_shadow_DOM