# 微前端架构基础知识

## 链接：

微前端架构设计：<https://juejin.im/post/6854573214430380039>

## 微前端架构存在的价值

我们先来了解一下微前端出现的原因，由于大部分开发同学开发的都是单体应用。先举几个实际工作中开发单体应用常见的问题吧：

 同学A跟我吐槽：

* 场景：
  + 最近要接受一个遗留后台系统，里面的技术老旧不说，连框架都是自己造的轮子，文档也没有
* 问题：
  + 近期可能有一些新功能要增加到旧系统内，但是在原有系统上改造难度高
  + 后续可能会对旧系统内的现有功能做一些改造，但是原有的一套技术体系老旧，想要重构到ts+react+webpack，但是里面有些祖传页面大概率不会迭代，又必须保持线上

 同学B团队近期也遇到了问题：

* 场景：
  + 他们所在的团队要负责一个运营后台系统，目前系统内的功能已经非常庞大，涉及的成员也非常多，里面每一个子菜单的功能都非常庞大，每个子菜单可能都是由于不同的团队在负责。并且目前所有功能全部聚合在一个项目内
* 问题：
  + 项目内的代码规范和技术体系选择，都会造成不同团队的争议
  + 某个子菜单内的功能进行了变更整个项目都需要重新打包部署，增加了不稳定性和分支混乱，部署时间长
  + 不同子菜单内同时有需求迭代，两者的功能需要进行合并才能部署到预发、线下，这就造成了不同需求之间的互相挤兑，影响测试发布效率，而且还需要保证相互之间没有影响

在前端应用日益复杂化，框架技术更新迭代快的场景下，现有的单体工程化方案在多团队协作、解决历史遗留代码力不从心，微前端工程化方案很好的解决了上述问题：技术栈无关、拆解巨石应用。

## 微前端的核心价值

* 拆解巨石应用为多个子应用
  + 子应用独立运行和发布
  + 增量升级
* 技术栈无关
* 沙盒隔离子应用

通过微前端的核心价值我们可以很好的解决上述问题：历史遗留代码、跨多个团队协作、发布效率问题。跨空间和时间的产品很可能会导致应用变成一个巨石应用、技术栈老旧，导致项目的维护成本变高，微前端就是为了解决这些问题。

## 微前端的技术架构

### 为什么不采用iframe

其实从浏览器原生的方案来说，iframe不从体验角度上来看几乎是最可靠的微前端方案了，主应用通过iframe来加载子应用，iframe自带的样式、环境隔离机制使得它具备天然的沙盒机制，但也是由于它的隔离性对用于体验带来了一些副作用：

* 视窗大小不同步（例如我们在iframe内的弹窗想要居中展示）
* 登录态cookie同步问题
* 子应用间通信问题
* 大量组件重复

### SPA微前端架构

从iframe的用户体验我们很难将其作为微前端的标准方案，那我们自己要做一套微前端框架具体要做哪些事情呢，从iframe功能我们大致能够了解到，微服务框架需要具备以下几个功能：

* 子应用加载器（Loader）
* 路由控制（Router）
* 沙盒隔离（Sandbox）
* 子应用通信（Store）

### 主工程基座

微前端架构必须有一个主工程基座，基座主要是做为承载子应用的容器，子应用通过导出对应的格式，主应用在进入到对应路由时加载对应的子应用。

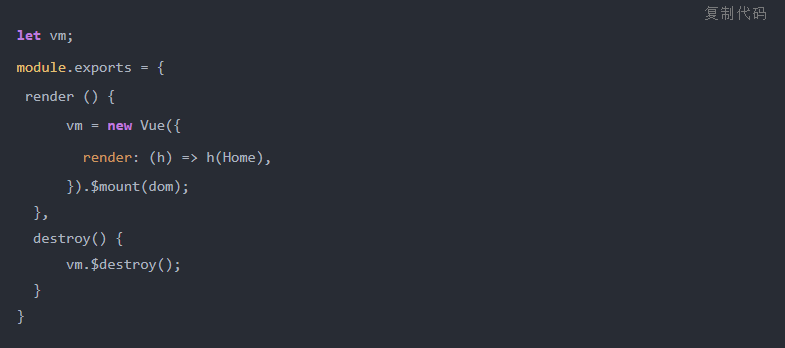
## 微前端核心功能

### Loader

loader是微前端核心模块的加载器，可以通过loader来进行子应用的加载，目前的微前端方案设计里面一般有两种模式。

第一种是非侵入式，通过加载对应子应用的 index.html 文件，再通过对首页html文件进行解析，获取到子应用的js文件和css文件，进行加载。

另一种是子应用打包成一个js文件，按照规范的导出格式,主应用只加载 index.js 文件。获取到对应的render和destroy方法。



作为模块加载器它通常需要提供以下几种能力：

* 下载子应用源码
* 编译解析子应用导出内容
* 将公共依赖注入到子应用中

### External

在微前端中有一个需要解决的问题就是，子应用间的公共依赖，我们如何抽离项目间的公共依赖呢，由于我们将一个应用拆分成了多个子应用，那子应用之间的依赖如何复用。如果了解commonJS的同学应该知道，commonJS具备加载模块缓存能力，加载过的模块会将其缓存起来，那么是不是我们可以将子模块以commonJS的规范进行打包。在加载子模块时，提供全局的exports和require方法，将子应用导出的exports进行收集,在require时加载我们配置的external资源。

### Common JS加载实现

代码：

Module.\_catcheModule = {}

function req(moduleId){

if(Module.\_catcheModule[p]){

//模块已存在

return Module.\_catcheModule[p].exports

}

//没有缓存就生成一个

let module = new Module(p);

Module.\_catcheModule[p] = module;

//加载模块

module.exports = module.load(p);

return module.exports;

}

function Module(p){

this.id = p

this.exports = {}

this.load = function(filepath){

return Module.\_extensions(this)

}

}

Module.\_wrapper = ['(function(exports,require,module,\_\_dirname,\_\_filename){','\n})']

Module.\_extensions = function(module){

let fn = Module.\_wrapper[0] + fs.readFileSync(module.id,'utf8') + Module.\_wrapper[1]

vm.runInThisContext(fn).call(module.exports,module.exports,req,module)

return module.exports

}

通过上面commonJS的源码模式实现，我们只需要将exports中增加公共依赖，并且子应用通过webpack构建工具，提供external配置同样的公共依赖即可

### Sanbox

在微前端框架中另一个核心的模块就是沙盒，由于多个子应用会反复的展示在同一个容器内，子应用中难免会造成对当前环境的副作用，例如：全局样式、全局变量、监听事件、定时器等。沙盒在这里主要是为运行中的程序提供隔离环境，避免应用之间相互影响

#### 在web端设计沙盒我们需要考虑哪些因素

* 全局环境污染
* 事件污染
* style污染
* 定时器污染
* localstorage污染

为解决全局环境污染和style污染，通常采用，快照模式和代理劫持模式。

#### 快照模式

* 沙盒启动时
  + 遍历存储当前全局环境变量，将其缓存
  + 遍历head所有标签，将其缓存
* 沙盒运行时
  + 执行副作用程序
* 沙盒关闭
  + 遍历缓存遍历将其与全局环境对比，发现差异进行还原
  + 将当前head标签与缓存标签进行对比，发现差异进行还原

#### 代理模式

 沙盒启动时

* 缓存原始节点添加、删除、在节点前插入等原始方法
* 缓存添加监听事件、移除监听事件方法
* 缓存添加定时器事件、移除定时器事件
* 将缓存的事件更改为代理事件，在代理事件内部执行真实事件前，收集变更

 沙盒运行时

* 执行副作用程序

 沙盒关闭

* 恢复沙盒运行期间产生的变更
* 移除代理事件