SCP 插件化模块方案预研

概述

背景

`SCP` 云平台服务目录有大量的模块功能，内部的一些产品如 `DaaS`，需要在 `SCP` 上面加一些功能，这些功能是完全独立的一个模块，希望能够独立开发部署，做成可插拔式的集成进来。

需求目标

1. UI支持动态展示新增应用的入口
2. 支持动态插拔插件，提高系统灵活性
3. 隔离开发关注点，减少对原有代码的影响，避免前端代码成为一个大泥球，从而提高代码可维护性
4. 应用独立维护、测试

同类产品分析

当一个前端项目开发规模越来越大时，前端领域逐渐出现一种趋势，为了便于多个团队可以同时开发和维护一个大型且复杂的产品，可以将大型的前端项目分解成许多个小而易于管理的独立部署的应用，并实现应用级别的资源（UI组件/工具函数/业务模块）分享，就像后端领域的微服务一样，就是微前端。

实现微前端的方式目前主流的有多种，从浏览器自带的 `iframe` ，以及一些开源的微前端框架 `qiankun`，`microApp` 等等，这些框架为了解决集成一些现有的第三方应用、或是技术栈不同的应用，而引入了如样式隔离、js 沙箱等机制。带来的复杂度上升，隔离不充分导致的一些疑难问题以及一些兼容性（比如 沙箱所依赖的 proxy 特性是不支持 IE 的）。

从被集成方的角度来看的，被集成的子应用分为两种：

1. 受控，在平台内部开发一些完整的模块功能，看起来就跟平台其他模块一致，如 `DaaS`
2. 非受控，完全第三方应用，有着自己的技术栈与完整应用，如 `DMP`

非受控方的集成

由于无法控制集成的子应用所使用的技术方案，需要做到对子应用的隔离，那么有以下两种方案：

- iframe嵌入
- 单点登录

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、js 隔离这类问题统统都能被完美解决，但它的最大问题也在于他的隔离性无法被突破，导致应用间上下文无法被共享，随之带来开发体验、产品体验的问题。

1. 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。
2. 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。
3. 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。
4. 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 集成的方式， 之前已经有过预研，上述的部分问题已经有过讨论，见 应用集成

对于受控方的集成

由于这部分是内部可控的，我们可以限制所使用的技术栈，这样的话我们能够有复用现有平台模块的能力。

由于技术栈同构，每次加载子应用的时候不会重新去加载公共依赖，达到与平台原生模块一致的体验，不会有性能相关问题。

不过这种方式的缺陷也比较明显，那就是必须使用同一个框架，同一套技术栈，主应用使用什么，子应用也必须保持一致，对外的扩展性相对较差。

由于目前我们需要集成的主要是内部子系统（受控），为了提供更好的用户体验，我们将尽可能以受控的方式集成到平台上，故当前方案的研究主要在受控式集成方案上。

受控式集成方案

方向

在插件化集成方面，我们只需要关注的是如何独立开发，以及开发完成后的代码怎么去集成进我们现有的 SCP 应用中。

在构建步骤中，对应用进行一步的拆分，并重新组合，独立构建应用，生成 chunk 文件，在需要的时候，才加载对应的应用。

但是，首先它有一个严重的限制：**必须使用同一个框架。**对于现有场景来说，这并不是问题，采用微服务的团队里，也不会因为微服务这一个前端，来使用不同的语言和技术来开发。当然了，如果后续要使用别的框架，也不是问题，我们只需要结合之前的微前端框架去满足我们的需求。

其次，采用这种方式还有一个限制，那就是：**规范。**在采用这种方案时，我们需要：

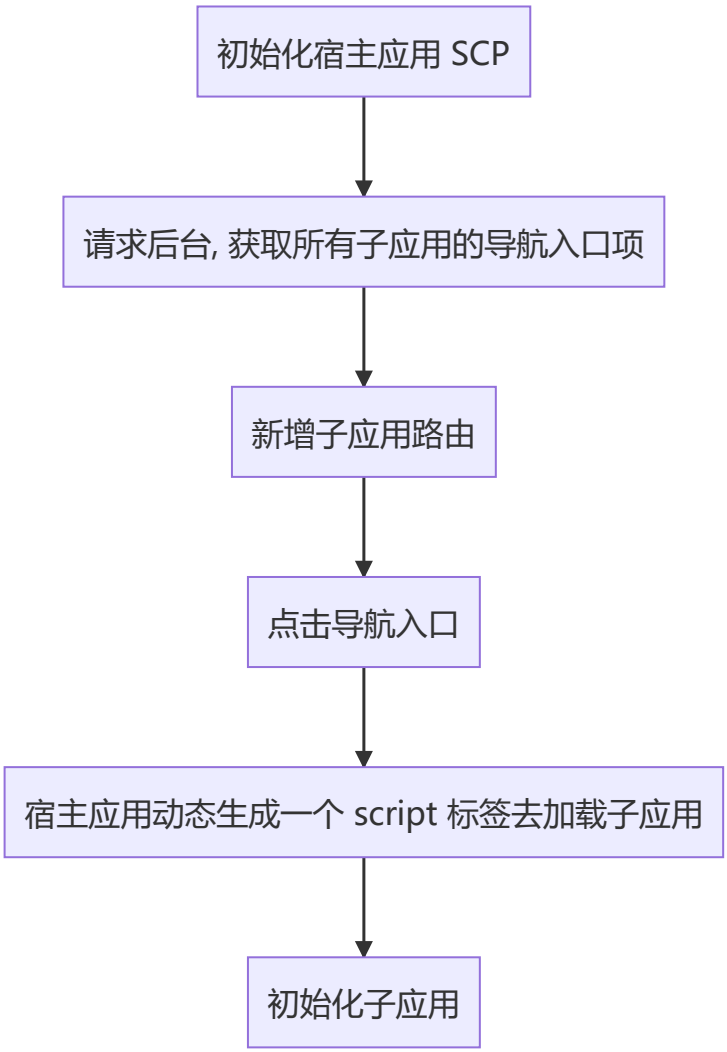
- 统一依赖版本
- 规范应用路由与名称，避免不同的应用之间，因为这些组件名称发生冲突
- 构建复杂度会变高，需要修改构建系统
- 共享通用代码
- 制定代码规范

方案概览

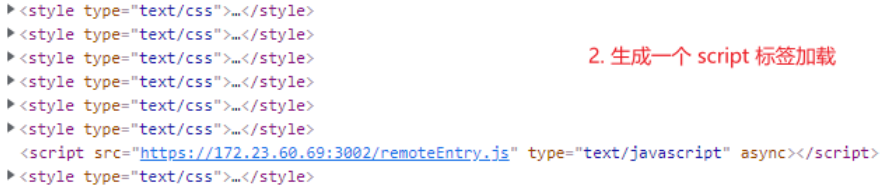
原 SCP 项目作为宿主应用，然后使用 `webpack5` 的 `module-federation` 特性，子模块打包成 chunk 作为一个独立的子应用集成进来按需加载。

1. 子应用使用 `ModuleFederationPlugin` 插件在 `exposes` 导出模块初始化入口，宿主应用异步创建 `script` 去动态集成子应用。
2. 宿主应用拿到子应用的入口，通过入口导出的内容去动态添加路由，然后在导航上新增一项子模块的链接，这里可以是可以通过后端请求返回。
3. 打包方面，子应用会打包成单独一个 chunk 文件，公共依赖与宿主应用共享，子应用可以在公共依赖版本没变更的时候，单独打包升级发布。

加载流程


初始化宿主应用 SCP
请求后台, 获取所有子应用的导航入口项
新增子应用路由
点击导航入口
宿主应用动态生成一个 script 标签去加载子应用
初始化子应用

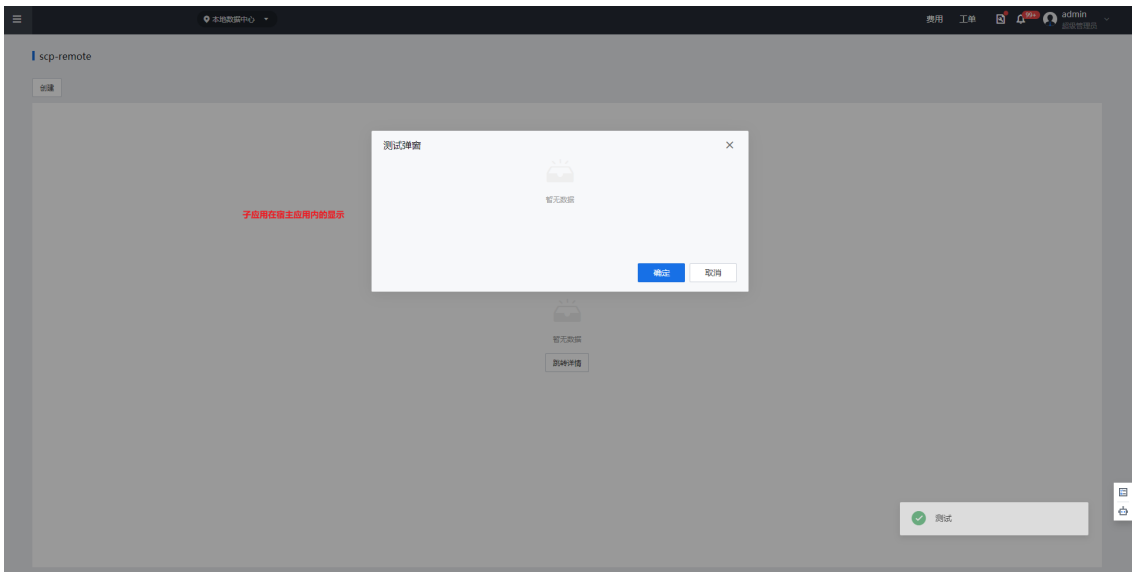
1. 
Name
remote-chunk
Headers
Preview
Response
Initiator
Timing
Cookies
{success: 1, status: 200,…}
data: {chunk: "https://172.23.60.69:3002/remoteEntry.js", name: "_scp_remote"}
chunk: "https://172.23.60.69:3002/remoteEntry.js"
name: "_scp_remote"
status: 200
success: 1
1. 请求后台，获取子应用的入口文件等信息

2. 
<style type="text/css">…</style>
<style type="text/css">…</style>
<style type="text/css">…</style>
<style type="text/css">…</style>
<style type="text/css">…</style>
<style type="text/css">…</style>
<script src="https://172.23.60.69:3002/remoteEntry.js" type="text/javascript" async></script>
<style type="text/css">…</style>
2. 生成一个 script 标签加载

3.

Name	Status	Type	Initiator	Size	Time	Waterfall
src_app-common_mod-init_remote_js.dd5b261448f7b4d924b2.js	200	script	entry.5ac6b4d....js:2924	3.4 kB	4 ms	
src_app-acmp_mod-common_storage_util_js-src_app-co...mon_v...	200	script	entry.5ac6b4d....js:2924	19.1 kB	28 ms	
node_modules_virtual_common_util_timer-node_module...d-src_...	200	script	entry.5ac6b4d....js:2924	25.8 kB	30 ms	
src_app-common_chart_datetime_js-src_app-common_common_...	200	script	entry.5ac6b4d....js:2924	19.9 kB	51 ms	
src_app-acmp_mod-common_unit-progress_index_vue.5c1c1ffc1...	200	script	entry.5ac6b4d....js:2924	11.8 kB	50 ms	
src_app-common_mod-overview_statistics-card_index_vue.fb1aa...	200	script	entry.5ac6b4d....js:2924	13.7 kB	50 ms	
src_app-acmp_mod-launch_fire_js.3cfa209a666b872ec064.js	200	script	entry.5ac6b4d....js:2924	348 kB	71 ms	
remoteEntry.js	200	script	remote.js?f8d8:27	175 kB	42 ms	
vendors-node_modules_vue-hot-reload-api_dist_index...node_m...	200	script	remoteEntry.js:527	9.6 kB	12 ms	
vendors-node_modules_core-js_modules_es_array_conc...s-node...	200	script	remoteEntry.js:527	120 kB	812 ms	
src_exports_js.js	200	script	remoteEntry.js:527	10.5 kB	18 ms	

3. 加载子应用的入口 remoteEntry

4. 

子应用加载方式

在运行时通过接口获取配置，然后动态的创建异步 `script` 标签加载，去初始化子应用的导航、路由等。

> 以下配置宿主应用名称为 `_host_`， 子应用名称为 `_remote_`

1. `webpack` 配置

```
// _host_ webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: '_host_',
    }),
  ],
};

// _remote_ webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: '_remote_',
      filename: 'remoteEntry.js',
      exposes: {
          './exports': './src/exports.js',
      },
    }),
  ],
```

```
};

// _remote_ ./src/exports.js
const getRoutes = () => {
    return [
        {
            path: '/remote',
            component: () => import('./entry.vue'),
        },
        {
            path: '/remote/detail',
            component: () => import('./views/detail.vue'),
        },
    ];
};

export const bootstrap = async ({ router }) => {
    router.addRoutes(getRoutes());
};
```

2. 异步加载代码

```
// _host_
// 创建一个 script 标签加载 remoteEntry
async function loadScript(url) {
    return new Promise((resolve, reject) => {
        const element = document.createElement('script');
        element.src = url;
        element.type = 'text/javascript';
        element.async = true;

        element.onload = () => {
            console.log(`Dynamic Script Loaded: ${url}`);
            resolve();
        };

        element.onerror = () => {
            console.error(`Dynamic Script Error: ${url}`);
            reject();
        };

        document.head.appendChild(element);
    });
}

async function loadModule(scope, module) {
    // 初始化共享作用域（shared scope）
    await __webpack_init_sharing__('default');

    // scope 为 remoteEntry 加载时挂载的全局变量，如 `_remote_`
    const container = window[scope];

    // 初始化远程容器，拿到共享的模块
    await container.init(__webpack_share_scopes__.default);
    const factory = await window[scope].get(module);
    const Module = factory();
    return Module;
```

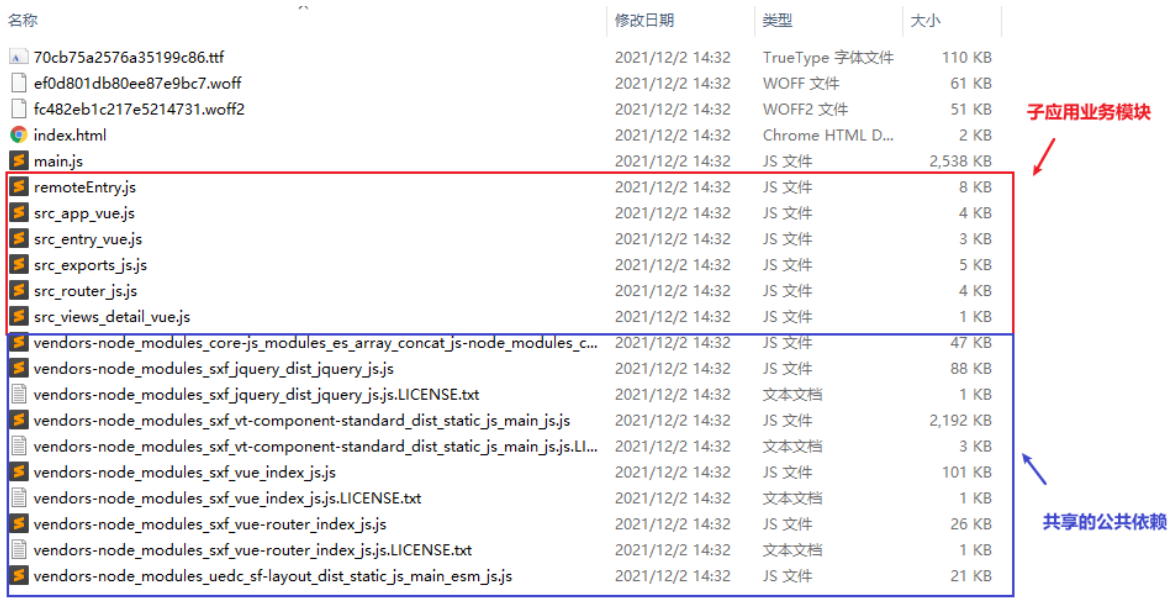
```
}

async function initChunks(remoteList) {
    for (const { chunk, name } of remoteList) {
        await loadScript(chunk);
        const { bootstrap } = await loadComponent(name, './exports');

        if (bootstrap) {
            await bootstrap();
        } else {
            console.warn(`${name} chunk 需暴露 bootstrap 方法 `);
        }
    }
}

export async function loadRemote() {
    // await new Promise(resolve => {
    //     fetchRemoteConfig().then(async ({ data }) => {
    //         const remoteList = data;
    //         await initChunks(remoteList);
    //         resolve();
    //     });
    // });

    await initChunks(chunks);
}

// chunks 如下格式
// const chunks = [{ chunk: 'https://172.23.60.69:3001/remoteEntry.js', name:
'_remote_' }];
```

注意点：

1. 由于路由需要在初始化的时候就加载，所以 `loadRemote` 的这部分逻辑要尽量在 `VueRouter` 初始化完成而 `App` 主应用未创建的时候去执行，这样就不会出现在子应用页面刷新后，找不到路由的问题。
2. 子应用导出的内容即 `exports` 文件内，使用到的路由组件等尽量以异步的方式延迟加载，这样的话在打包的时候 `webpack` 会进行代码切割， 输出的入口 `remoteEntry.js` 部分就只有 `webpack` 自行生成的 `runtime` 胶水代码，这部分内容很小，而真正的业务代码会在进入具体业务模块后才开始加载。

编译打包结果

子应用打包出来，需要最终加载的有一个 `remoteEntry.js` 入口以及一些业务模块，公共依赖在页面不会进行加载。



依赖共享

宿主应用与子应用在 `ModuleFederationPlugin` 插件中配置 `shared` 字段，两者都要配置

```
// _host_ webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: '_host_',
        shared: {
            vue: { packageName: '@sxf/vue', requiredVersion: '1.0.2', singleton:
true },
            jquery: { packageName: '@sxf/jquery', requiredVersion: '1.0.4',
singleton: true },
            $: { packageName: '@sxf/jquery', requiredVersion: '1.0.4',
singleton: true },
            '@sxf/vt-component-standard': { singleton: true },
        },
    }),
  ],
};
```

这样配置之后，对于在 `shared` 内声明的第三方包会被分离，做到子应用与宿主应用共享的效果

其他问题

宿主应用

1. 与子应用的通信
    - 解耦后独立开发，不需要通信
    - 如后续扩展实在需要通信的时候，提供一个事件通信机制（常用的发布订阅模式都行）
2. 依赖共享，如 `vue`、`VTComponent`、`jQuery` 等公共依赖

   见前面提到的依赖共享，通过配置 `shared` 声明依赖的名称、版本等，另外：

    - 组件库不能源码引用，不然会造成共享公共方法，实例不一致的问题，所以需要优先处理 `VTComponent` 独立打包，不能使用源码。

- 共享依赖的时候 `alias` 会有影响，比如 `shared` 里直接配置 `@sxf/vue` ，使用的时候因为 `alias` 的关系，代码中是 `import Vue from 'vue'` 这样的话不会共享同个包。 需要在 `shared` 中把 `key` 配置为 `vue`， `packageName` 为 `@sxf/vue` 才会生效，如下：

```
shared: {
    vue: { packageName: '@sxf/vue', requiredVersion: '1.0.2', singleton: true },
}
```

3. 子应用动态路由以及导航栏的添加
   - 导航目前是在前端写死的，需要进行改造，否则只能硬编码到项目中，不太合理
   - 子应用的路由写在子应用内部，在加载的时候，通过 `router.addRoutes` 去动态加到主应用路由表里面去。
     - 由于需要向后台请求一次获取路由表去用 `addRoutes` 添加，动态添加时应该在什么时机?
       - 不使用 `addRoutes`，在整个应用初始化之前就要获取的子应用的路由表，然后初始化给设置进去，不然如果当前页面是子应用的页面，页面刷新时会找不到路由而跳转到首页去。
       - router 找不到对应路径而重定向的时候，会有个 `router.currentRoute.redirectedFrom` 字段标识从哪重定向过来的，可以在添加完成后再跳转回来。这种情况还是会出现一次跳首页的问题

子应用

子应用需要支持脱离宿主应用独立开发，且能够随时集成到宿主应用。

1. 初始化开发脚手架提供，能够直接开发业务页面，不用额外配置过多东西
   - 提供一个初始化有 `@sxf/vt-component-standard`、`@uedc/sf-layout` 等组件且内置了各种配置，以及初始化代码的模板仓库
   - 将上述内容封装成一个 `CLI`，提供统一的打包配置，这种方式优点是可以统一对公共配置升级，子应用开发时不需要额外的一些配置，但是缺点也是一样，比较难做定制化的配置。
2. 打包出来的 chunk 的导出规范，如文件名、格式、版本信息、缓存等
3. 独立开发时，一些登录状态等处理
   - 本地代理联调，如果本地启动了 `SCP`，远程代理登录了，那么不同端口的子应用服务也能直接共享到 `cookies`，麻烦的是必须要本地启动 `SCP`。
   - 给子应用专门封装一个登录页
4. 国际化
   - 子应用单独处理国际化相关内容

业务公共组件

子应用开发的时候，涉及到原 SCP 平台的一些业务组件时

1. 将这部分业务组件独立出去单独发一个 `npm` 包
2. 将这部分业务组件独立出去同样当做一个项目，在这个项目中，将这些组件通过 `exposes` 共享（远程组件库）
3. 子应用自行开发

缺陷预防

无法做到插件化的内容

插件化是尽量以路由级别的完全独立模块页面，如果是对原有页面进行一些修改，那么这种细粒度的定制只能用硬编码的方式写入进去，所以在交互设计层面需要关注这些问题。

目前识别到涉及硬编码的模块

根据交互稿识别后，目前 `SCP` 只能进行硬编码的模块

导航

导航目前的所有文本、跳转URL等都是写死在前端业务代码内，如果要在服务目录加一个入口，是直接在代码中添加一项。 现有方式做动态展示的话，需要把内容写死，然后通过条件判断去显示。

如果要做到子应用的入口代码跟宿主应用分离，需要进行改动，有以下几种方式：

- 导航服务目录存放在后台，包括文本，跳转 URL 等，使用这种方式会导致前端路由跟后端代码耦合起来，如果前端路由有改变，那么后端也必须跟着改。
- 提供一个 hook 机制，在原来导航项不变的情况下，子应用注册 hook，主应用使用 `Object.defineProperty` 拦截导航对象，然后在 `getter` 中执行这些 hook 去动态插入新的项。
- 导航服务目录存放在前端的一个配置文件，后端读取配置，把主应用跟所有子应用的导航整合起来通过接口返回给前端。

DaaS

1. 系统管理 -> 服务管理 启禁用 DaaS 服务的入口。
2. 申请配额
3. 工单审批
4. 角色权限
5. 云主机的操作项

业务访问可视化

1. 云主机控制台

针对这部分模块的演进

1. 探索针对页面的某一小块区域的插件化，组件级别的粒度
2. 与后端配合，这部分内容做到后端返回配置，前端渲染

基础公共依赖升级

如果有一方公共依赖升级了，导致依赖不一致的处理

- 主应用跟子应用要做到同时升级公共依赖，否则会出现依赖版本不一致的部分兼容问题

打包方案升级

目前整套方案是基于 `webpack5` 的 `module-federation` 特性，后续如果要迁移 `vite` 的兜底方案

- 在 `vite` 中有对标的技术方案（vite-plugin-federation），需要验证
- 宿主应用跟子应用要使用一致的打包方案，否则需要做一层中间层去兼容差异，比如 `webpack` 与 `vite`

后端配合项

1. 提供一个获取子应用信息的接口，返回类似以下格式，标识子应用的入口 chunk 地址以及应用名称，前端动态的去加载。

```
{
    "data": [
        {
            "chunk": "https://172.23.60.67:3001/remoteEntry.js",
            "name": "_scp_remote"
        }
    ]
}
```

2. 提供一个获取所有导航项的接口
    - `SCP` 宿主应用和各个子应用，在打包后都会有一个导航项的 `JSON` 配置，对应展开的服务目录入口
    - 后端需要扫描上述配置整合起来，这个整合的过程需要把权限等方面给考虑进去
    - 前端初始化的时候请求接口去获取导航项
3. 联调问题，子应用是必须要登录 `SCP` 才能调试？还是有什么其他能独立联调的方式

演进计划

长期演进目标

想要做到开放生态，让任意第三方都能接入到我们平台，需要提供一个**解决方案**，这个解决方案要包括

- 受控方式的接入
    - 提供项目的开发框架，其中有统一的构建打包工具
    - 提供符合设计规范的基础组件及业务组件
    - 提供平台内置已有的基础模块，比如用户体系，工单等，这部分要能够自由组合
- 非受控方式的接入（用户已有的第三方应用集成的能力）
    - iframe 式接入
        - 基于通讯的 SDK 调用
        - ...
    - 多页跳转
        - 单点登录授权
        - ...

短期改进计划

当前业务中存在的问题

1. **封装业务公共组件，提供 SDK**
    - 问题：没有基础业务组件
    - 影响：容易造成子应用的重复开发，影响质量、效率
    - 解决方案：提前识别并做分装（当前封装的：选择资源池、选择租户、租户列表）
    - 风险点：现有的业务组件比较分散且跟实际业务耦合过深
2. **对现有业务模块进行拆分解耦**
    - 问题：现有业务模块互相耦合过深
    - 影响：无法做到自由组合去生成一个新的子系统
    - 解决方案：进行拆分解耦，能够做到除公共依赖外完全独立

- 风险点：历史包袱重，改动影响面大，可能产生改动引发，质量风险比较高