**⚡️@keepfe/micro 微前端框架设计**

**一、 背景**

**相关概念:**

|  |
| --- |
| 什么是微前端？ |

微前端的理念就是用微服务的模型去重构前端工程

将Web 应用程序可被拆分成多个模块，独立开发，独立部署，独立测试（松耦合）

|  |
| --- |
| 微前端带来哪些优势： |

1. 工作效率提升：全量构建 到 增量 构建，分而治之对大型项目更容易拆分而治之。

2. 开发成本降低：提供跨端调用的能力，节省开发（这里的跨端是跨web应用端。

3. 便宜性：实现不同web项目的组合，解决业务遗留问题和用户组合使用的矛盾

**设计背景:**



|  |
| --- |
| 我们Keep前端的 微服务准备做什么？ |

* 阶段一：内网项目的可聚合可重组：
* 阶段二：外网功能组件下沉化
* 阶段三：实现 配置 => web 应用的 零开发

|  |
| --- |
| 补充 |

微服务是一种技术手段，可以在多种场景下 解决问题。

**现状分析：**

* 2B 内网 项目

- 部分系统搭建 立项较为零散，使用者使用往往需要多页面操作。

- 部分系统间 公用逻辑 变成 复用逻辑：

1. 人工Copy
2. Npm 化

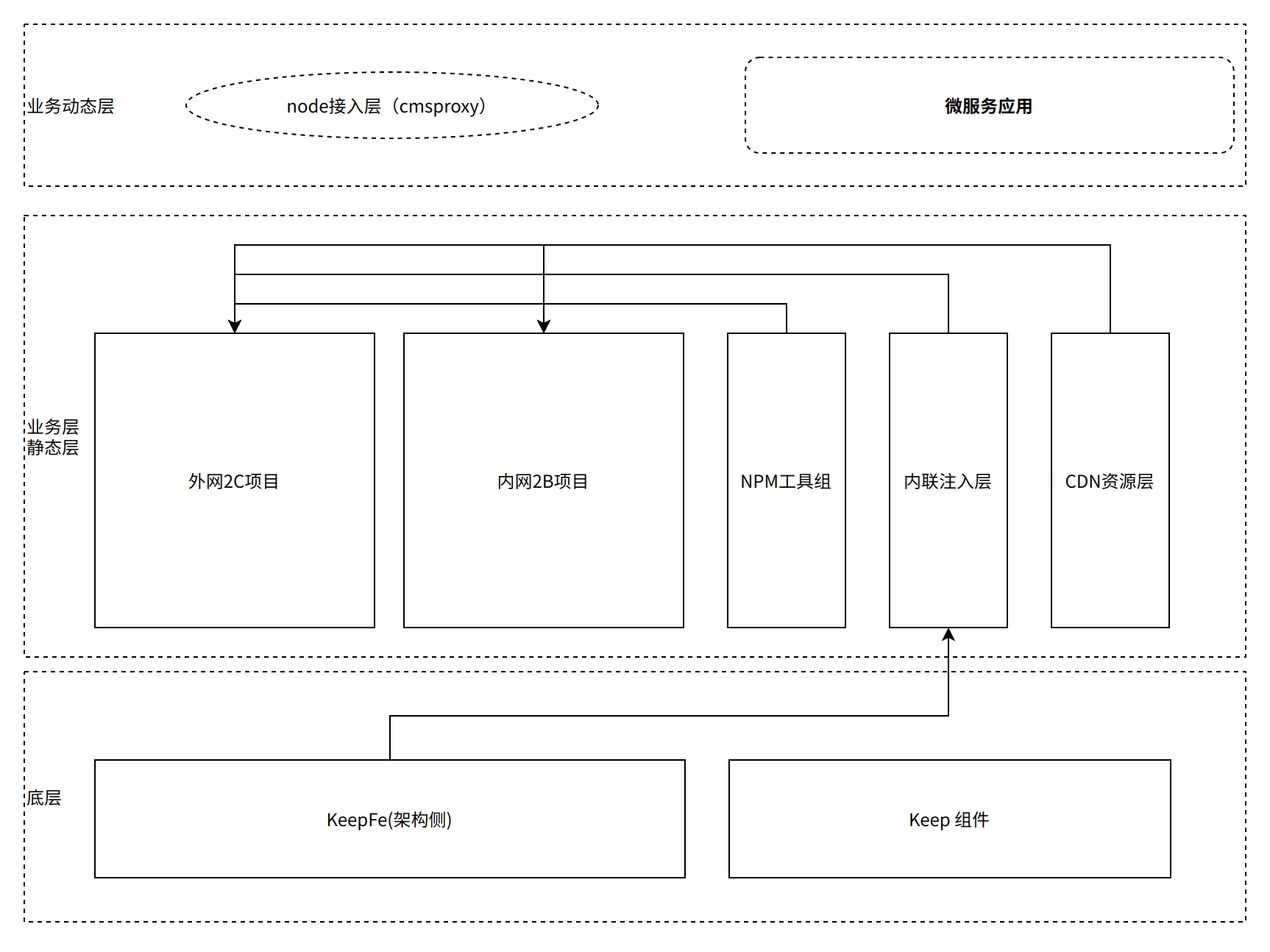
* 2C 外网 项目

- 不同web间 公用逻辑 变成复用逻辑

1. 人工Copy
2. Npm 化

**现有架构：**

|  |
| --- |
| 如图，Keep现有前端体系架构。 |



解释：我们看到 现阶段，Keep前端项目的公共组件 包括（Npm, 内联构建资源，cdn资源）类，公共组件均为静态，而微服务应用作为跨端能力，提供更上层的通用能力支持，并可以通过Node接入层管控，甚至可以看作 类似于 Faas, Baas功能

|  |
| --- |
| 浅谈npm化的弊端 |

1. npm 调用的逻辑需要我们在 使用端 执行 并实现，即需要 可能 文档协议的 调用逻辑
2. 多项目 npm 调用，会在 每个 项目中 代码重新生成一遍
3. npm 调用部分 工具 不能脱离构建，例如 我们 的 npm 化的 组件 是 sfcs 文件模型，代表 vue 组件的情况下。
4. npm 调用更多的其实是一种 方法，工具，而微服务 的 面向更加广阔，可以是一种成型应用能力

**二、 设计准备**

**1> 设计调研**

**A. 页面结构纬度：**

* iframe 嵌套
* Web Components (隔离性好)
* 组合式应用路由分发（需要设计和研发）

|  |
| --- |
| 评价：  1) Iframe 嵌套 不论从体验上，还是交互上，还是性能上，还是数据共享上都是隔离的，对于我们的场景十分不方便  2) WebComponents 模式技术，不适合 KEEP 的 以 Vue 项目为主；兼容性差；  [mdn链接]（https://developer.mozilla.org/zh-CN/docs/Web/Web\_Components）  3) 组合式路由分发  采用组合式路由分发： 请求的路由对应到对应的 应用上 |

**B. 设计模式纬度：**

* 基座模式：通过搭建基座、配置中心来管理子应用
* 统一框架的约定方案 如 [微前端在美团外卖的实践](https://juejin.cn/post/6844904073972432903)
* 多框架并存方案 如 Single-spa
* 自组织模式： 通过约定进行互调
* Monorepo + npm 包组合
* Amd, cmd, Umd 直接引用
* Webpack 5 联邦模式

|  |
| --- |
| 评价：  1) 自组织模式，其实有点类似于 npm 模式，包括 amd, cmd, umd，一定会遇到上述 npm 的一些问题，另外 Webpack 5 联邦模式 虽然 在npm 的基础上解决了 独立构建 以及 分发 注册的 很多问题，但是 webpack 5 的配套框架尚未完善，故放弃  2) 基座模式：由于Keep体系 采用 Vue 体系，并统一 构建 流程，所以采用统一框架的约定方案（当然这里后续和 美团的方案 其实 不同很多，后续会有拆解） |

**3> 调研总结**

* Keep 自身 的 前端框架的统一化，构建统一化 决定了 像 Qiankun 框架 大锅烩的框架 （**1. 需要对 web 应用 出口，入口封装；2. 沙箱环境导致的Proxy 代理window带来的问题**）；所以只需要对单一框架类型（vue）阶段做处理，做精细即可
* Keep 的自有前端项目 复杂程度中等，对于像EMP等去中心自组织模式（**这个框架 就是 用了 webpack5 的联邦模式，实现了 远程分享组件，并在构建端做了一些事情，所有 Application 的概念 很弱，更不要考虑 沙盒等 关键性东西了**）不符合Keep 的 流程集中管控的 模式
* **最终决定 采用参考 genesis 的组合式路由应用分发结构，基座模式交由Keepfe 统一管理，提供每个业务的 跨端调用的能力**

|  |
| --- |
| 另外，具体的的方案探讨调研可详见文档 |

(微前端设计调研初)[https://docs.google.com/document/d/1UOeJAbQfgS2kQRuyEr48OvWqyzgv4-FkzUVSPwZLBfI/edit#]

**二、 设计详情**

**1> 概要设计**

**A. 准备：**

|  |
| --- |
| 基础背景组件介绍（准备）： |

\*Keepfe 是 Keep 前端的统一前端 流程化工具，\*

* Keepfe （cli) : Keep前端项目统一模板工具
* Keepfe （build）：Keep 前端统一构建工具
* Keepfe （node）：Keep 前端统一服务承载

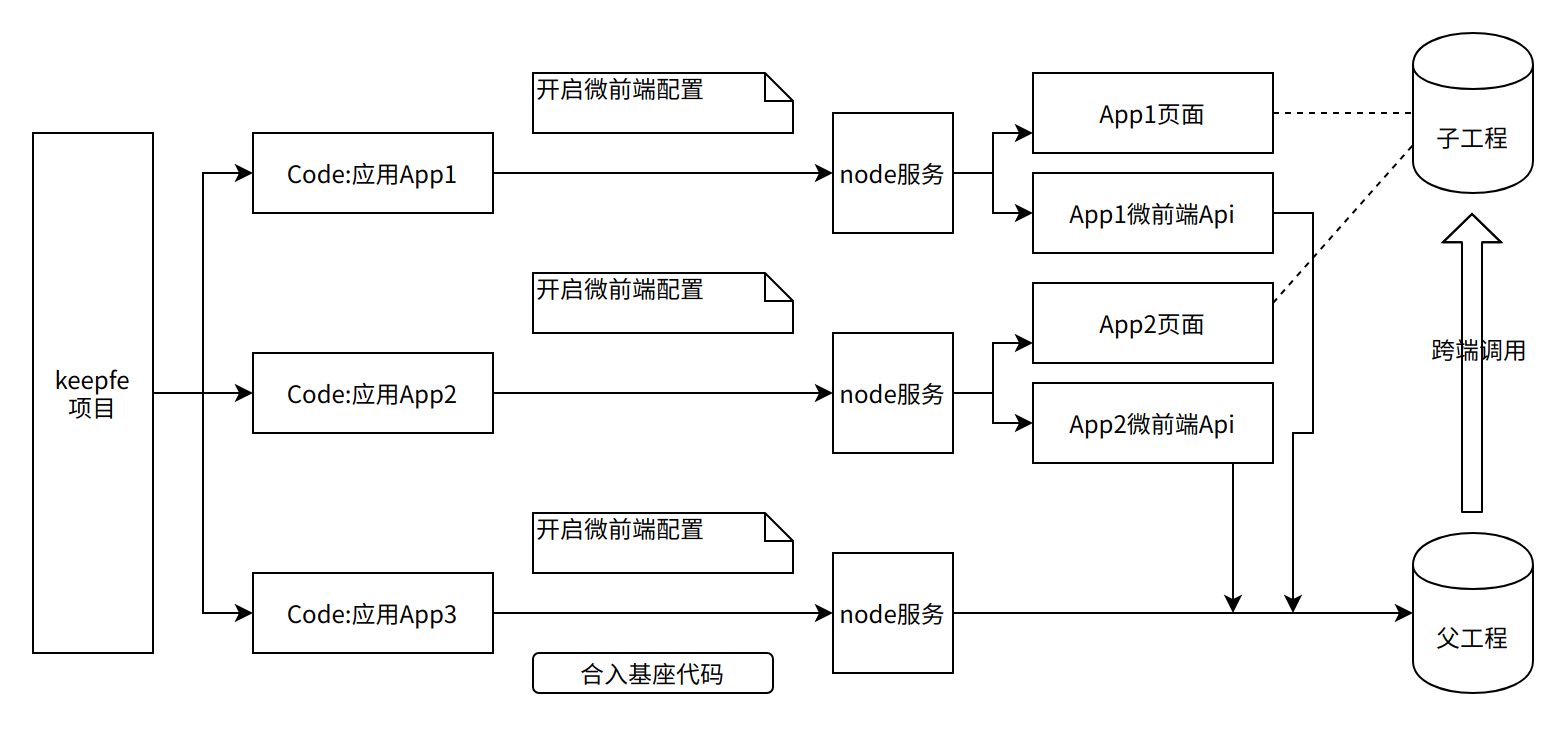
|  |
| --- |
| @Keepfe/micro 微前端整体设计方案： |

**B. 概要设计简图：**

*下图为Keep微前端实现方案 ：*

1. *首先KeepfeCli 生成 微前端模型的模板 交给业务使用*
2. *业务代码拿到微前端模板 结合业务 逻辑 可以生成基座模式，或者配置api*
3. *通过Keepfe 的Node 服务，所有的应用均可以独立允许作为 独立app 打开*

*通过Keepfe的Node服务，应用也可以作为api模式被基座应用调用，加载，执行*



**2> 详细设计**

**A. 核心概念：**

|  |
| --- |
| 微前端的几个核心模块 |

* 基座：

微服务的应用切换和加载是需要 壳子 来 展示和承载的，而基座 就是 这个壳子

* 应用 Api 化

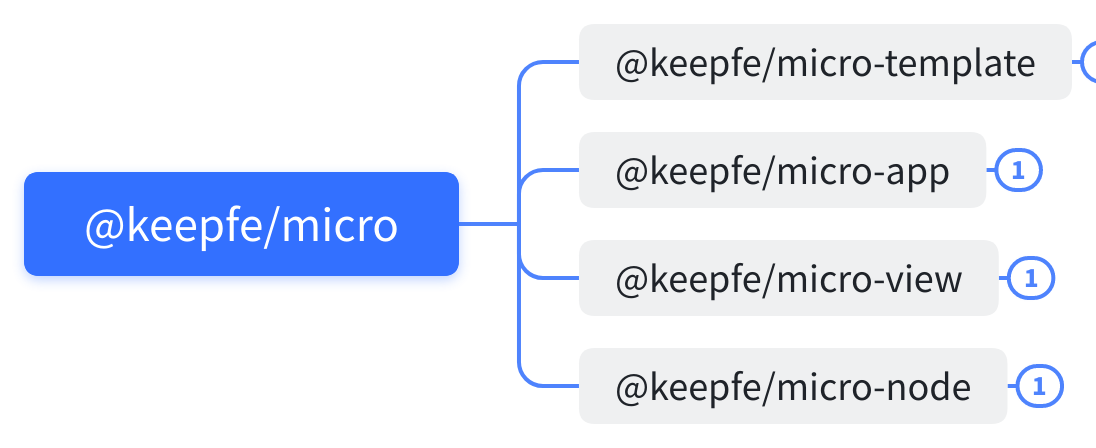
所谓应用 api 化 即 我们会定义一套约定，将应用的 信息 通过 api 来传递，而 这个 api 信息 则可以被调用方执行

* Api 应用化

能够将 约定 的 api 调用 在 基座中 应用展开

**B. 核心模块：**

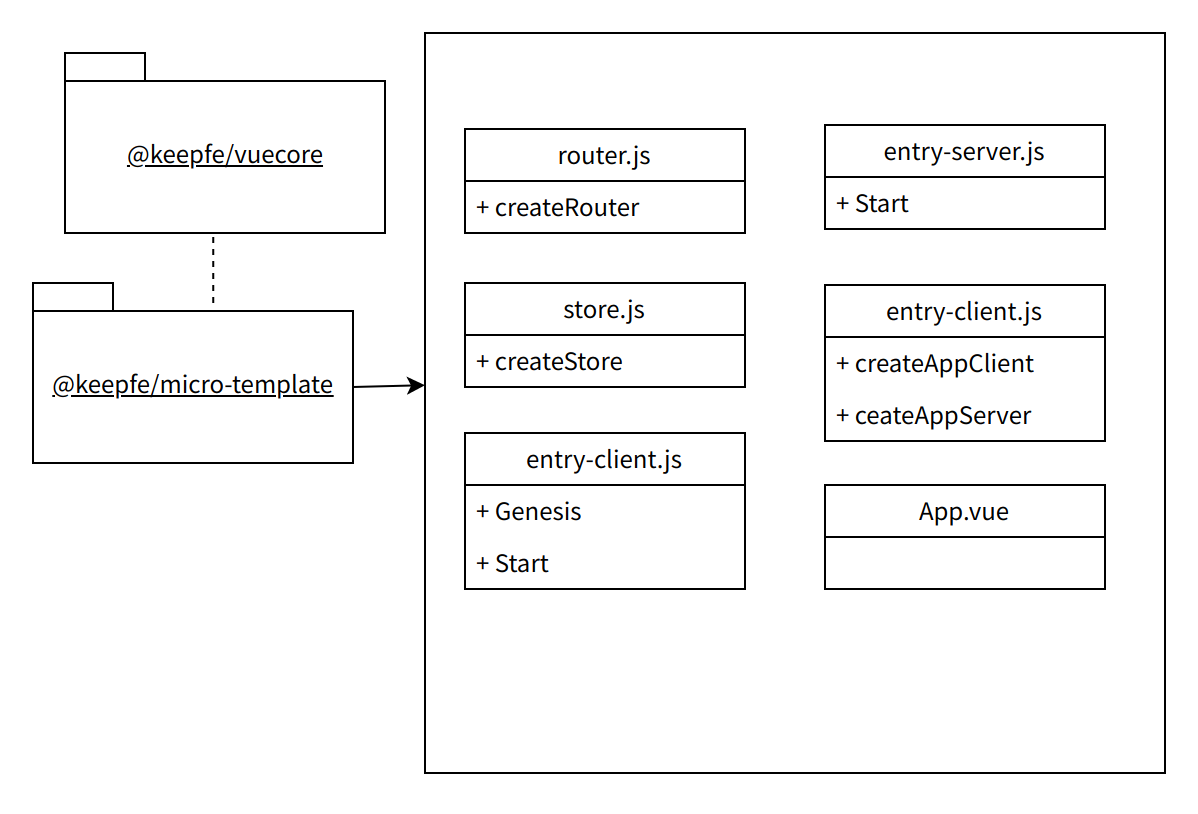
|  |
| --- |
| @Keep-micro 核心模块 |



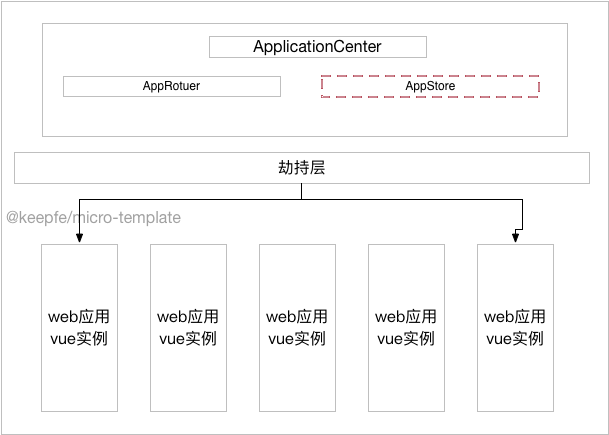
**@keepfe/micro-template**

|  |
| --- |
| @keepfe/micro-template |

* @keepfe/micro-template 是 提供 微服务 应用的 模板 文件：

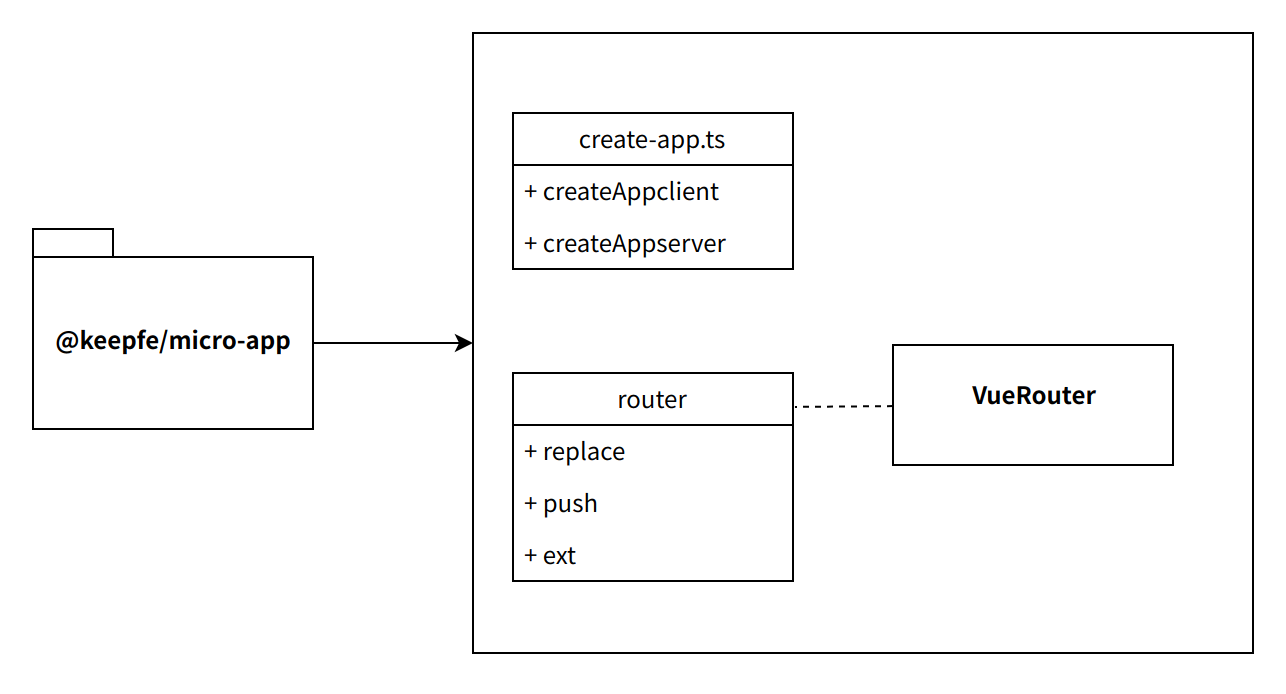


*@keepfe/tempate 对 原有的 Keepfe 项目的 启动函数core 做了核心更改，实现了多Vue实例，多Router实例，多Store实例，以及对 Router 的穿透做了相应的处理* ， 而Genesis 则作为下图的 Application Center 负责微服务的 的 多实例控制



**@keepfe/micro-app**

|  |
| --- |
| @keepfe/micro-app |



*@keepfe/micro-app 是服务于 micro-template 中的 工具，create-app.ts 提供 Vue 实例的创建，Router则继承VueRouter 对 微服务模式下 的路由规则 进行处理*

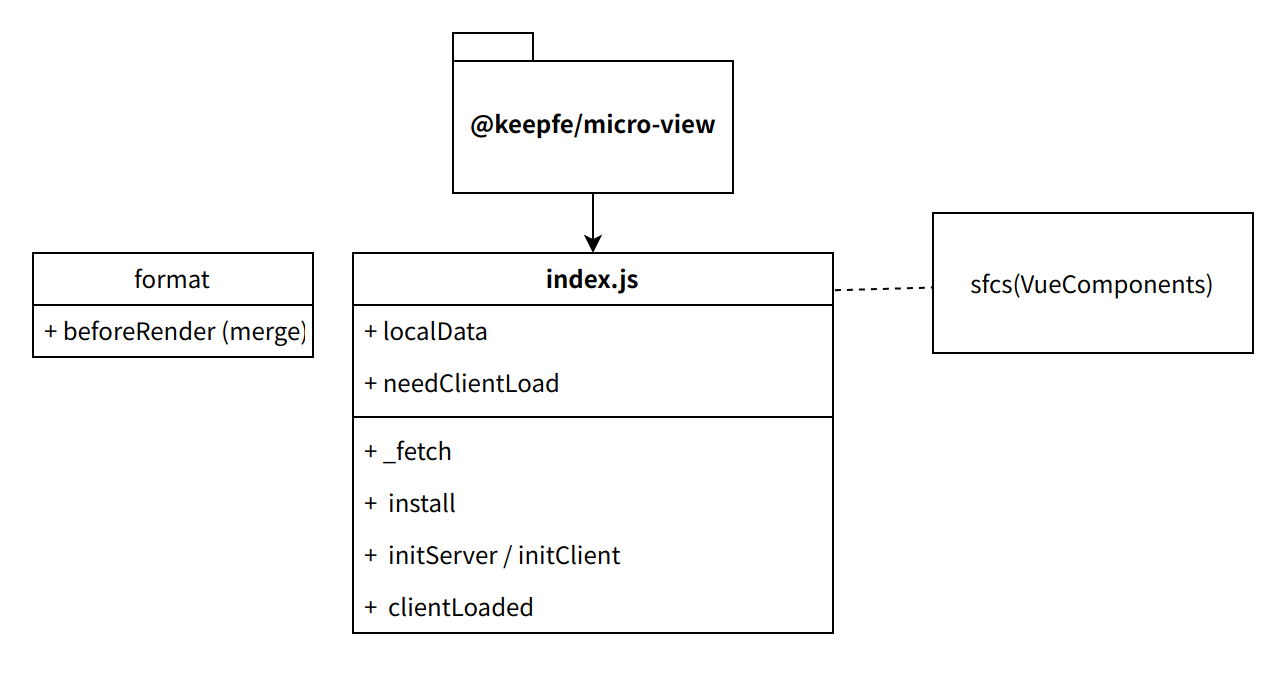
**注意事项** ：

1. 采用微服务模式 禁止对 vue, vuex, vue-router 采用 external 处理，否则会造成污染。

2. 为了能正确使用 跨应用服务的 vueRouter，请确保调用方 匹配 子应用的 router

**@keepfe/micro-view**

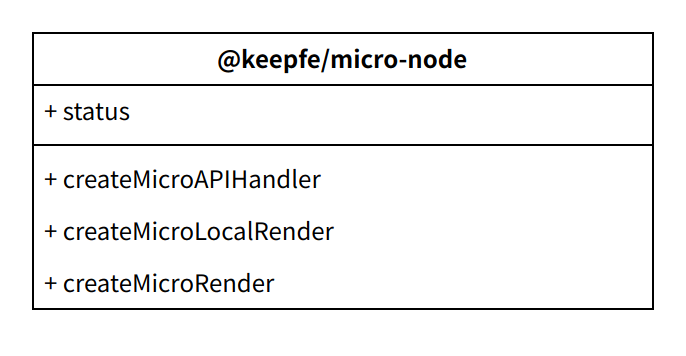
|  |
| --- |
| @keepfe/micro-view |



*@keepfe/micro-view 作为微前端基座，采用了 Vue Components的调用方式，通过Components 作为 View 来管理 微前端容器的边界值，包括 Server 端渲染容器 和 Client 端 渲染容器两种模式，主要负责请求 Api 模式的 服务，并渲染绘制*

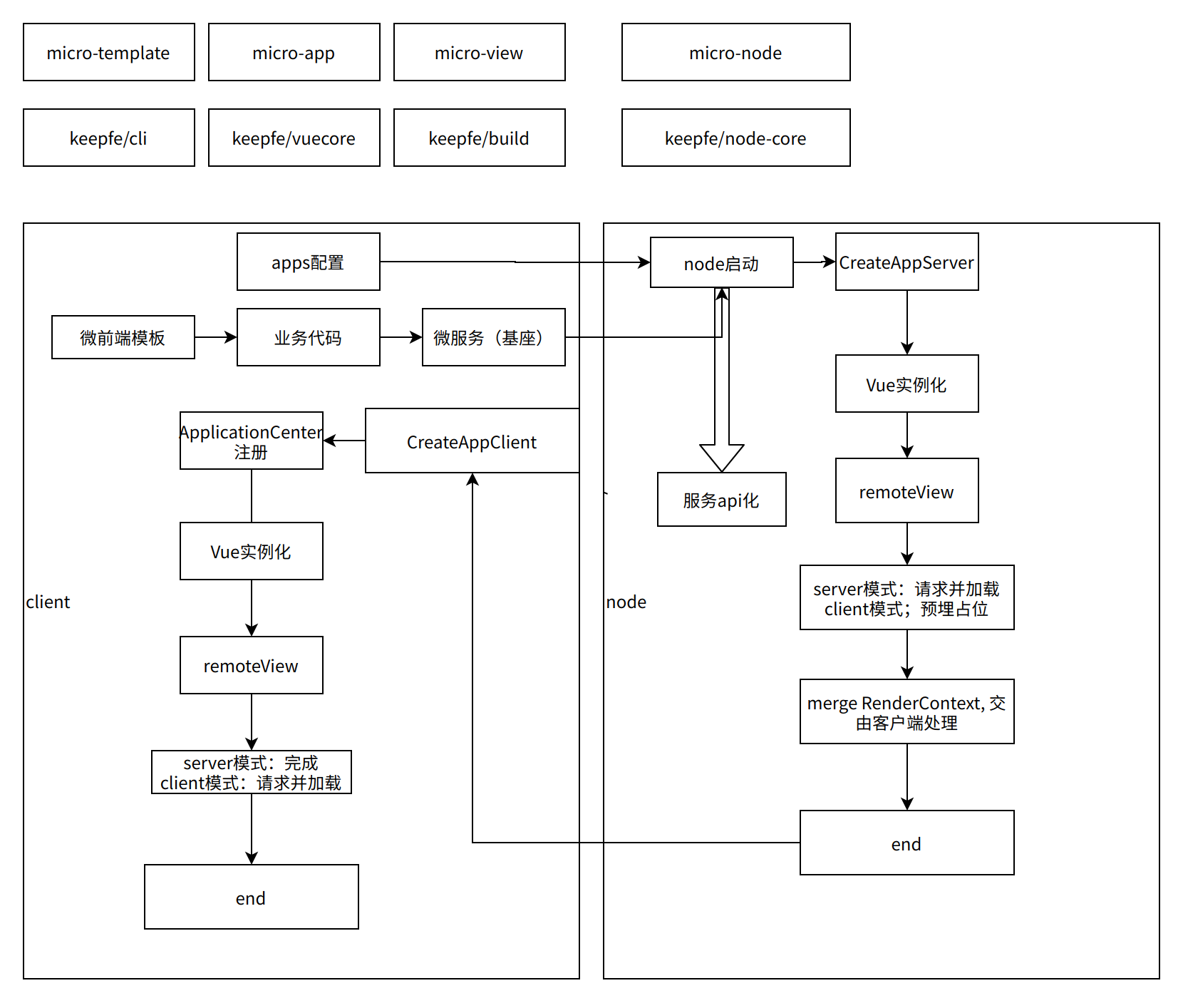
**@keepfe/micro-node**

|  |
| --- |
| @keepfe/micro-node |



\*@keepfe/micro-node 作为 前端 @keepfe/node-core 的通用服务的补充包，实现 对于 微服务模式 类型的 app 一套自有渲染模型，并实现对 服务 api 化调用的 的封装\*

**c. 整体流程图：**



**3> 接入指南**

|  |
| --- |
| 普通项目如何改造微服务应用： |

* 只是想可以被作为微服务调用
* 修改apps.js 的配置，appType 修改为 micro
* 修改apps.js 的配置，nodeRouter 添加 micronew
* 增加 layout模板，加载micro资源

|  |
| --- |
| JavaScript // layout/index.html {% block content %}  {% if is\_ssr\_render -%}  {{ "{{{" }} data && data.scriptState }}}  {{ "{{{" }} data && data.micro\_script && data.micro\_script.replace(/script /g, 'script crossorigin=\"anonymous\" ') }}}  {{ "{{{" }} data && data.micro\_style }}}  {%- endif %} {% endblock %} // 注意，如果开启了 openCSP 需要对资源做 nonce处理 // 演示： {{ "{{{" }} data.micro\_script.replace(/script /g, 'script crossorigin=\"anonymous\" nonce="{{CSP\_nonce}}" ') }}} |

* 修改entry.js 的配置，这里是对之前的的架构做了优化（钩子函数initFn 即可满足 router的所有声明周期）

|  |
| --- |
| JavaScript import iView from 'view-design' import config from '\_app/config' import iviewEntry from '@vuecore/uiLayouts/iview-admin/entry' import 'view-design/dist/styles/iview.css'  let initFn = async (Vue, router, store) => {  console.log('initFn')    router.beforeEach((to, from, next) => {  iviewEntry.beforeEach && iviewEntry.beforeEach(to, from, next, store)  })   let importFilter = await (() => import('\_app/filter'))  let importDirective = await (() => import('\_app/directive'))  Vue.use(iView)    let filterModule = await importFilter()  let directiveModule = await importDirective()   filterModule.default(Vue)  directiveModule.default(Vue)    Vue.config.productionTip = false   Vue.prototype.$config = config  }  export default {  initFn } |

* 支持项目 可以 在 node 端执行（因为要将 项目 api 化，所以必须要能在node端执行，所以ssr项目直接ok, 非 ssr 项目 可以添加一层wrapper 包裹，示例如下）

|  |
| --- |
| JavaScript <template>  <div>  <component :is="appComponent"></component>  </div> </template> <script> import Main from '@vuecore/uiLayouts/iview-admin/components/main'  const Blank = {  render: function (createElement) {  return createElement('div', [])  } }  export default {  computed: {  appComponent() {  // eslint-disable-next-line  if (typeof window !== 'undefined') {  return Main  } else {  return Blank  }  }  } } </script> |

* 想调用其他微服务应用
* 调用注册基座，示例如下

|  |
| --- |
| JavaScript <template>  <div>  <div class="micro-page">  <remote-view  v-for="name in names"  v-show="ssrname === name"  :key="name"  :clientFetch="() => clientFetch(name)"  :serverFetch="() => serverFetch(name)"  ></remote-view>  </div>  </div> </template> <script> /\* eslint-disable \*/ import {RemoteView} from '\_app/node\_modules/@keepfe/micro-view/dist/esm/index.js';  export default {  name: 'container',  components: {  RemoteView  },  data() {  return {  names: []  };  },  computed: {  ssrname() {  return this.$route.meta.ssrname;  }  },  watch: {  ssrname() {  if (this.names.indexOf(this.ssrname) > -1) return;  this.names.push(this.ssrname);  }  },  created() {  this.names.push(this.ssrname);  },  methods: {  async clientFetch(ssrname) {  try {  const renderUrl = encodeURIComponent(this.$route.fullPath);  const res = await axios.get(  `http://local.dev.gotokeep.com/${ssrname}/micro-api/`,  {  params: {  routerMode: 'abstract',  renderMode: 'csr-json',  renderUrl  }  }  );  if (res.data.ok) {  return res.data.data;  }  } catch(e) {  console.log(e)  }  return null;  },  async serverFetch(ssrname) {  try {  console.log(ssrname)  const renderUrl = encodeURIComponent(this.$route.fullPath);  const res = await axios.get(  `http://local.dev.gotokeep.com/${ssrname}/micro-api/`,  {  params: {  routerMode: 'abstract',  renderMode: 'csr-json',  renderUrl  }  }  );  if (res.data.ok) {  return res.data.data;  }  } catch(e) {  console.log(e)  }  return null;  }  } }; </script> |

* 补充指南（使用 keepfe iview layout 的必看）

这是个优化选项，因为我们keepfe iview项目 作为子项目加载 会有默认有 导航栏，header显得不是很好看，所以需要：

1. Router 使用时传递参数可以打开 窗口scale模式：

|  |
| --- |
| JavaScript  //routers/\_list.js let list = [  {  path: '/',  redirect: '/home',  component: Main,  name: '\_home',  meta: {  hideInMenu: false,  notCache: true  },  props: {  layoutConfig: {  needScale: true,  needFull: true,  needTag: true,  needCollapsed: true,  selfLogo: false  }  }  } ] |

1. 因为我们中间使用了 warpper的 一层 包裹组件，所以需要参数透传到iview-admin中

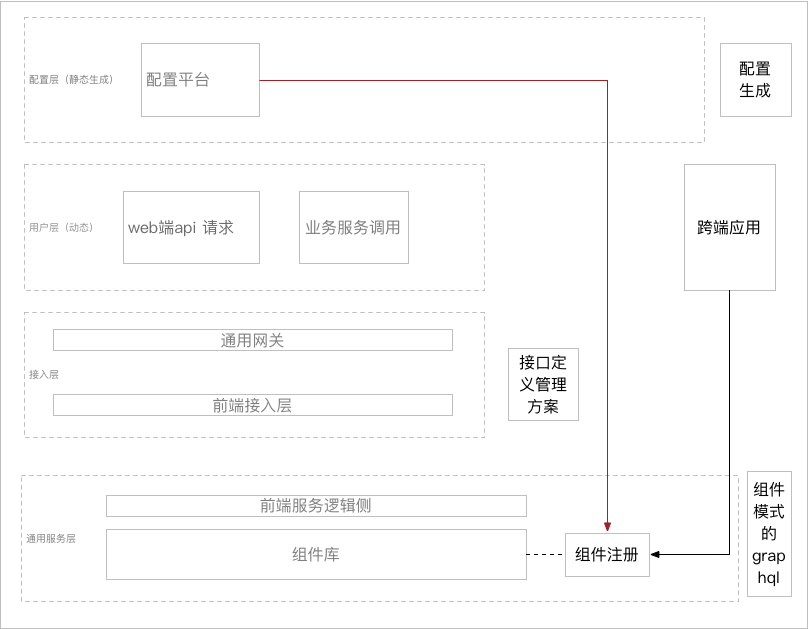
|  |
| --- |
| HTML // wrapper.vue <template>  <div>  <component :is="appComponent" v-bind="currentProperties"></component>  </div> </template> <script> import Main from '@vuecore/uiLayouts/iview-admin/components/main'  const Blank = {  render: function (createElement) {  return createElement('div', [])  } } export default {  props: ['accordion', 'logoInfo', 'layoutConfig', 'selfColor'],  computed: {  currentProperties() {  if (typeof window !== 'undefined') {  return {   accordion: this.accordion,  logoInfo: this.logoInfo,  layoutConfig: this.layoutConfig,  selfColor: this.selfColor  }  } else {  return {}  }  },  appComponent() {  // eslint-disable-next-line  if (typeof window !== 'undefined') {  return Main  } else {  return Blank  }  }  } } </script> |

**4> 未来设计方向**

1. 下沉组件注册到通用服务层

理解：传统的前端组件往往基于前端引用，无外乎npm化，cdn化均是 js资源形式引用或者不引用。我们后续是不是可以下沉通用重要组件（登录，权限，身份)等相关组件 下沉服务，既可以在服务端加工封装，也更保证多样性和安全性`

1. 跨端应用组件调用 可 graphql 化

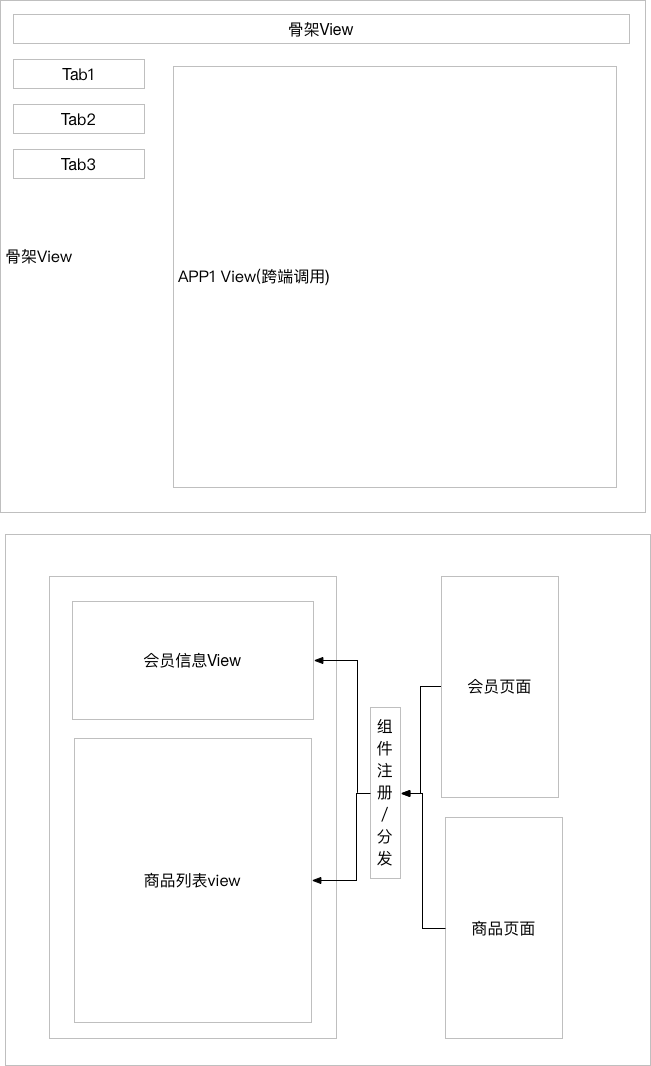


理解：这里其实是对组件服务下沉Api 的一种发散，既然组件 可以 服务下沉到 后端服务侧，那么我们是否可以采用向 graphql 的思路：对下沉组件服务 + 配置文件 定义多样性。

1. 真正意义的无需发布切换ABtest：

**三、 总结与预览**

|  |
| --- |
| 模拟场景演示 |



|  |
| --- |
| 实际场景演示 |

|  |  |  |
| --- | --- | --- |
|  |  |  |

线上demo:

[newshare 空白项目 加载 infra-cms 项目](https://m.pre.gotokeep.com/newshare/infra-cms/home)

|  |
| --- |
| 总结： |

其实 如果把 npm, cdn 等形式的插件 引用 看作是 二维的 插件引用，那么微前端 跨应用的调用可以看成三维的调用，实现的是对某种能力的调用自执行：即 不需要主动控制调用应用，对应用的控制调用不必参数化，而是抽象化

**五、补充信息**

1. **需要注意的地方**

***\* 选填。本次设计不尽完善的地方，之前遇到的坑等***

2. **附录**

1. [架构设计：微前端架构](https://zhuanlan.zhihu.com/p/79388540)
2. [微前端在美团外卖的实践](https://juejin.cn/post/6844904073972432903)

**设计评审记录**

***\* 选填。如果该设计参加了技术评审，在这里补充当时评审过程中的讨论记录。***

|  |  |  |  |
| --- | --- | --- | --- |
| 时间 | 技术评审会议 notes |  | 是否通过评审 |
|  | 要求：  Q&A: |  |  |