代码的天敌就是代码量

H2 微前端

H3 前言

什么是微前端？？

微前端是一种类似于微服务的架构，它将微服务的理念应用于浏览器端，即将单页面前端应用由单一的单体应用转变为把多个小型前端应用聚合起来的应用。各个前端应用可以使用不同的技术栈独立开发、独立运行、独立部署

微前端架构具备以下几个 核心价值：

- 技术栈无关，接入友好
  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（业务域）
  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
- 增量升级

  在面对各种复杂场景时，我们通常很难对一个已经存在的系统做全量的技术栈升级或重构，而微前端是一种非常好的实施渐进式重构的手段和策略
- 独立运行时
  每个子应用之间状态隔离，运行时状态不共享

解决的问题！！

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。

H3 Why Not Iframe?

https://www.yuque.com/kuitos/gky7yw/gesexv

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。但他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，随之带来的开发体验、产品体验的问题。

1. url 不同步。浏览器刷新 iframe url 状态丢失、后退前进按钮无法使用。
2. UI 不同步，DOM 结构不共享。想象一下屏幕右下角 1/4 的 iframe 里来一个带遮罩层的弹框，同时我们要求这个弹框要浏览器居中显示，还要浏览器 resize 时自动居中..
3. 全局上下文完全隔离，内存变量不共享。iframe 内外系统的通信、数据同步等需求，主应用的 cookie 要透传到根域名都不同的子应用中实现免登效果。
4. 慢。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。

其中有的问题比较好解决(问题1)，有的问题我们可以睁一只眼闭一只眼(问题4)，但有的问题我们则很难解决(问题3)甚至无法解决(问题2)，而这些无法解决的问题恰恰又会给产品带来非常严重的体验问题， 最终导致我们舍弃了 iframe 方案。

H3 跟 iFrame、Web Components、NPM包、路由分发、插件有什么区别?

微前端	Widget / 业务组件
架构体系。用来实现大型Web应用	以库(外联/npm)的形式实现复用
生产方式	生产工具
通过隔离机制实现技术栈无关	**需要人工解决依赖和冲突问题**
单独构建 \ 单独发布 \ 热升级	整体构建 \ 整体发布
体系化治理，可控性强	可控性差
主从关系(路由映射、消息机制)	相互无关
微应用是产品的子集(粒度大)	通用功能(粒度小)
变化快	变化小
若干微应用的组合	“外挂”

H3 微前端架构实践中的问题

- SPA VS MPA

  MPA 方案的优点在于 部署简单、各应用之间硬隔离，天生具备技术栈无关、独立开发、独立部署的特性。缺点则也很明显，应用之间**切换会造成浏览器重刷**，由于产品域名之间相互跳转，流程体验上会存在断点。

  SPA 则天生具备体验上的优势，应用直接无刷新切换，能极大的保证多产品之间流程操作串联时的流程性。缺点则在于各应用技术栈之间是强耦合的。

  那我们有没有可能将 MPA 和 SPA 两者的优势结合起来，构建出一个相对完善的微前端架构方案呢?

- 构建时组合 VS 运行时组合
- JS Entry vs HTML Entry

  优势就是html-entry 巧妙的避开了js-entry加载子应用js的hash问题

  html entry 的好处是子应用依赖的资源不用过于关心

- 样式隔离

  微前端只能做到子应用之间是不会相互干扰的，父应用一般做的很少，就只有左侧菜单个顶部导航栏，很少会有跟子应用之间有样式之间的冲突，如果有的话就把父应用的权重提高就行了。

  - Shadow DOM

    基于 Web Components 的 Shadow DOM 能力（内外完全没联系），我们可以将每个子应用包裹到一个 Shadow DOM 中，保证其运行时的样式的绝对隔离。

但 Shadow DOM 方案在工程实践中会碰到一个常见问题，比如我们这样去构建了一个在 Shadow DOM 里渲染的子应用：

```
const shadow =
document.querySelector('#hostElement').attachShadow({mode:
'open'});
shadow.innerHTML = `
    <style>
    h2{
    color:red
    }
    </style>
    <h2>Shadow</h2>
`;
```

由于子应用的样式作用域仅在 shadow 元素下，那么一旦子应用中出现运行时越界跑到外面构建 DOM 的场景，必定会导致构建出来的 DOM 无法应用子应用的样式的情况。

比如 sub-app 里调用了 antd modal 组件，由于 modal 是动态挂载到 document.body 的，而由于 Shadow DOM 的特性 antd 的样式只会在 shadow 这个作用域下生效，结果就是弹出框无法应用到 antd 的样式。解决的办法是把 antd 样式上浮一层，丢到主文档里，但这么做意味着子应用的样式直接泄露到主文档了。gg...

- CSS Module?　BEM?

  社区通常的实践是通过约定 css 前缀的方式来避免样式冲突(人肉不推荐)，即各个子应用使用特定的前缀来命名 class，或者直接基于 css module 方案写样式。对于一个全新的项目，这样当然是可行，但是通常微前端架构更多的目标是解决**存量/遗产** 应用的接入问题。**很显然遗产应用通常是很难有动力做大幅改造的**。

  最主要的是，约定的方式有一个无法解决的问题，假如子应用中使用了三方的组件库，三方库在写入了大量的**全局样式**的同时又不支持定制化前缀？比如 a 应用引入了 antd 2.x，而 b 应用引入了 antd 3.x，两个版本的 antd 都写入了全局的 `.menu class`，但又彼此不兼容怎么办？ antd

- Dynamic Stylesheet

  动态 加载/卸载 样式表

  解决方案其实很简单，我们只需要在应用切出/卸载后，同时卸载掉其样式表即可，原理是浏览器会对所有的样式表的插入、移除做整个 CSSOM 的重构，从而达到 插入、卸载 样式的目的。这样即能保证，在一个时间点里，只有一个应用的样式表是生效的。

  上文提到的 HTML Entry 方案则天生具备样式隔离的特性，因为应用卸载后会直接移除去 HTML 结构，从而自动移除了其样式表。

  比如 HTML Entry 模式下，子应用加载完成的后的 DOM 结构可能长这样：

```
<html>

  <body>
    <main id="subApp">
      // 子应用完整的 html 结构
      <link rel="stylesheet" href="//alipay.com/subapp.css">
      <div id="root">....</div>
    </main>
  </body>

</html>
```

当子应用被替换或卸载时，`subApp` 节点的 innerHTML 也会被复写，`//alipay.com/subapp.css` 也就自然被移除样式也随之卸载了。

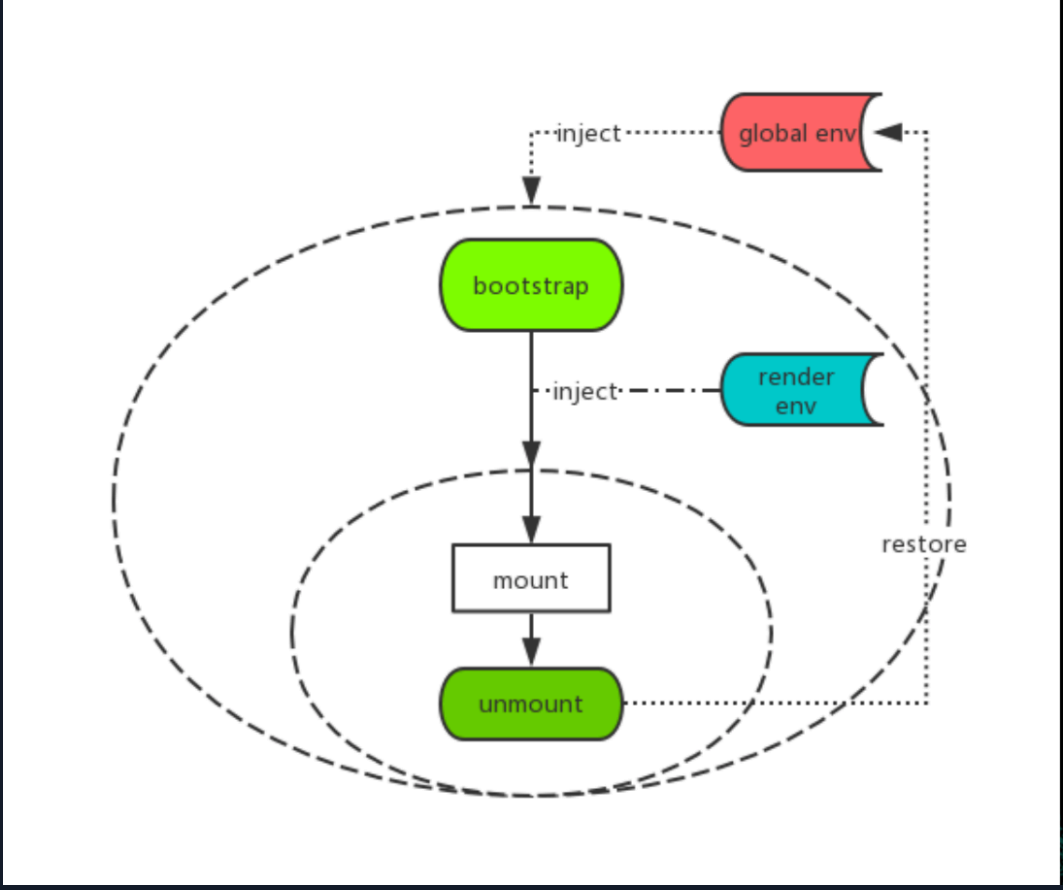
- JS 隔离

  基于proxy

  解决了样式隔离的问题后，有一个更关键的问题我们还没有解决：如何确保各个子应用之间的全局变量不会互相干扰，从而保证每个子应用之间的软隔离？

  这个问题比样式隔离的问题更棘手，社区的普遍玩法是给一些全局副作用加各种前缀从而避免冲突。但其实我们都明白，这种通过团队间的”口头“约定的方式往往低效且易碎，所有依赖人为约束的方案都很难避免由于人的疏忽导致的线上 bug。那么我们是否有可能打造出一个好用的且完全无约束的 JS 隔离方案呢？

  针对 JS 隔离的问题，我们独创了一个运行时的 JS 沙箱。简单画了个架构图：



即在应用的 bootstrap 及 mount 两个生命周期开始之前分别给全局状态打下快照，然后当应用切出/卸载时，将状态回滚至 bootstrap 开始之前的阶段，确保应用对全局状态的污染全部清零。而当应用二次进入时则再恢复至 mount 前的状态的，从而确保应用在 remount 时拥有跟第一次 mount 时一致的全局上下文。

当然沙箱里做的事情还远不止这些，其他的还包括一些对全局事件监听的劫持等，以确保应用在切出之后，对全局事件的监听能得到完整的卸载，同时也会在 remount 时重新监听这些全局事件，从而模拟出与应用独立运行时一致的沙箱环境。

- 资源预加载基座

  在浏览器空闲时间预加载（fetch 跨域）未打开的子应用资源，加速子应用打开速度。

H3 子应用的划分

在微前端架构中，我们应该按业务划分出对应的子应用，而不是通过功能模块划分子应用。这么做的原因有两个：

1. 在微前端架构中，子应用并不是一个模块，而是一个独立的应用，我们将子应用按业务划分可以拥有更好的可维护性和解耦性。
2. ** 子应用应该具备独立运行的能力，防止应用间频繁的通信(减少耦合)

H3 接入qiankun

H4 构建主应用基座

这里用vue作为主应用，接入其他的子应用

乾坤提供的API,一共没几个，接入方式特别简单。

子应用注册信息

```
// mic/micro-app-vue-main/src/micro/app.js
// https://github.com/hzfvictory/mic/blob/master/micro-app-vue-main/src/micro/apps.js#L20-L26
const isProduction = process.env.NODE_ENV === 'production';
const isEnter = isProduction ? '120.79.229.197' : 'localhost';
function genActiveRule(routerPrefix) {
  // 返回 true 就激活了子应用
  return location => location.pathname.startsWith(routerPrefix);
}

const apps = [
  /**
   * name: 微应用名称 - 具有唯一性
```

```
   * entry: 微应用入口 - 通过该地址加载微应用
   * container: 微应用挂载节点 - 微应用加载完成后将挂载在该节点上
   * activeRule: 浏览器url发生变化会调用这个函数，activeRule 返回 true 时表明该子应用需要被激活。
   * props 向子组件传递信息
   */
  {
    name: "ReactMicroApp",
    entry: `//${isEnter}:10100`,
    container: "#wrapper",
    activeRule: genActiveRule("/menu/react"),
    props: {data:[]},
  }
];
export default apps;
```

介入乾坤声明周期，错误捕获，导出启动函数

```
// mic/micro-app-vue-main/src/micro/index.js
// https://github.com/hzfvictory/mic/blob/master/micro-app-vue-main/src/micro/index.js#L27
import {Notification} from 'element-ui';

/*进度条插件*/
import NProgress from "nprogress";
import "nprogress/nprogress.css";

import {
  registerMicroApps,
  addGlobalUncaughtErrorHandler,
  start,
  removeGlobalUncaughtErrorHandler
} from "qiankun";

/*子应用注册信息*/
import apps from "./apps";

/**
 * registerMicroApps
 * @param  {array} apps - 必选，子应用的一些注册信息
 * @param  {function} lifeCycles - 可选，全局的子应用生命周期钩子
 * @param  {object} opts - 可选
 *           fetch - Function - 可选
 *           getPublicPath - (url: string) => string - 可选
 *           getTemplate - (tpl: string) => string - 可选
 */
```

```
registerMicroApps(apps, {
  /*qiankun 生命周期钩子 - 加载前*/
  beforeLoad: (app) => {
    // 加载子应用前，加载进度条
    NProgress.start();
    NProgress.set(0.4);

    console.info(`%c挂载前 ${app.name}`, `color:rgb(255, 208, 75);font-size:18px;`);
    return Promise.resolve();
  },
  /*qiankun 生命周期钩子 - 挂载后*/
  afterMount: (app) => {
    // 加载子应用前，进度条加载完成
    NProgress.done();
    console.info(`%c挂载后 ${app.name}`, `color:rgb(255, 208, 75);font-size:18px`);
    return Promise.resolve();
  },
});

/**
 * 添加全局的未捕获异常处理器
 */
addGlobalUncaughtErrorHandler((event) => {
  console.error(event);
  const {message: msg} = event;
  /*加载失败时提示*/
  if (msg && msg.includes("died in status LOADING_SOURCE_CODE")) {
    Notification({
      title: '加载失败',
      message: '子应用加载失败，请检查应用是否可运行',
      type: 'error'
    });
  }
});

removeGlobalUncaughtErrorHandler((err) => {
  console.error('移除未捕获的错误', err);
  return false
})

/*导出 qiankun 的启动函数*/
export default start;
```

然后在mainJs里面启动该函数主应用的任务就完成了。

```
// mic/micro-app-vue-main/src/main.js
// https://github.com/hzfvictory/mic/blob/master/micro-app-vue-main/src/main.js#L47
import startQiankun from "./micro";
/*
* prefetch  预渲染
* singular  是否为 单实例 场景
* jsSandbox 是否开启沙箱  关闭后兼容IE（但要承担关掉沙箱后子应用之间可能造成冲突的风险）
* fetch     自定义的fetch方法
* */

startQiankun({singular: true, prefetch: true});
```

到这一步，我们的主应用基座就创建好啦！

H4 接入子应用

首先，我们在 **React** 的入口文件 **index.js** 中，导出 **qiankun** 主应用所需要的三个生命周期钩子函数，代码实现如下：

```
// mic/micro-app-react/src/index.jsx
// https://github.com/hzfvictory/mic/blob/master/micro-app-react/src/index.js#L46
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import {ConfigProvider} from 'antd';
import zhCN from 'antd/es/locale/zh_CN';
import 'moment/locale/zh-cn';

/**
 * 渲染函数
 * 两种情况：主应用生命周期钩子中运行 / 微应用单独启动时运行
 */
function render() {
  ReactDOM.render(
    <ConfigProvider
      autoInsertSpaceInButton={true}
      locale={zhCN}>
      <App/>
    </ConfigProvider>,
    document.getElementById('root')
  );
}
```

```
// 独立运行时，直接挂载应用
if (!window.__POWERED_BY_QIANKUN__) {
  render();
}

/**
 * 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("ReactMicroApp 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
 * props 是注册的时候传进来的
 */
export async function mount(props) {
  console.log("ReactMicroApp mount", props);
  render(props);
}

/**
 * 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() {
  console.log("销毁");
  ReactDOM.unmountComponentAtNode(document.getElementById("root"));
}
```

在配置好了入口文件 **index.js** 后，我们还需要配置路由命名空间，以确保主应用可以正确加载微应用，代码实现如下：

```
// mic/micro-app-react/src/app.jsx
// https://github.com/hzfvictory/mic/blob/master/micro-app-react/src/App.js#L11
import React from 'react';
// ....
const BASE_NAME = window.__POWERED_BY_QIANKUN__ ? "/menu/react" : "";

function App() {
  return (
    <Provider store={store}>
      <Router basename={BASE_NAME}>
        <Switch>
```

```
          {renderRoutes(routes.routes)}
        </Switch>
      </Router>
    </Provider>
  );
}
export default App;
```

接下来要配置webpack

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

module.exports = {
  webpack: (config) => {
    // https://webpack.docschina.org/guides/author-libraries/#expose-the-library
    // 微应用的包名，这里与主应用中注册的微应用名称一致,添加这个后乾坤可以获取函数里面的值
    config.output.library = `${packageName}App`;
    // 将你的 library 暴露为`所有的模块`定义下都可运行的方式
    // https://webpack.docschina.org/configuration/output/#outputlibrarytarget
    // https://zhuanlan.zhihu.com/p/71168066
    config.output.libraryTarget = "umd";
    // 按需加载相关
    config.output.jsonpFunction = `webpackJsonp_${packageName}App`

    config.resolve.alias = {
      ...config.resolve.alias,
      "@": path.resolve(__dirname, "src"),
    };
    return config;
  },
  devServer: function (configFunction) {
    return function (proxy, allowedHost) {
      const config = configFunction(proxy, allowedHost);
      // 关闭主机检查，使微应用可以被 fetch
      config.disableHostCheck = true;
      // 配置跨域请求头，解决开发环境的跨域问题
      config.headers = {
        "Access-Control-Allow-Origin": "*",
      };
      // 配置 history 模式
      config.historyApiFallback = true;
```

```
      config.hot = true;
      config.open = false;  // 子应用设置false
      return config;
    };
  },
};
```

我们需要重点关注一下 `output` 选项，当我们把 `libraryTarget` 设置为 `umd` 后，我们的 `library` 就暴露为所有的模块定义下都可运行的方式了，主应用就可以获取到微应用的生命周期钩子函数了。

到这里，`React` 微应用就接入成功了！其他的技术栈接入方式大同小异，不在一一列举，具体看下方github。

H3 通信

示例：子应用跳转到另一个子应用（通过主应用做媒介）

H4 基于浏览器原生事件做通信

CustomEvent API 详情

父应用

首先，我们在主应用中初始化CustomEvent，挂载到window上，然后添加我们要传递的值：

```
// mic/micro-app-vue-main/src/app.vue
export default {
   mounted() {
      function createEvent(params, eventName = 'emit') {
        // 数据必须挂载到detail上
        return new CustomEvent(eventName, {detail: params});
      }
      // 初始化
      window.cEvt = createEvent({handelData: this.handelData, jumpUrl:
this.jumpUrl});
    },
    methods: {
      handelData(...opt) {
        // 为了避免重新渲染 obj 可以放到外面声明
        this.obj = Object.assign(this.obj, ...opt);
        return this.obj
      },
      jumpUrl(url){
```

```
        // 跳转
        this.$router.history.push(url)
      }
    }
}
```

子应用

然后子应用在函数中添加事件监听，执行跳转操作触发事件

```
// mic/micro-app-react/src/pages/detail/index.jsx
import React, {Fragment, useEffect, useRef} from "react"

const Index = (props) => {
  let msgRef = useRef(null);

  useEffect(() => {
    document.addEventListener('emit', queryData);
    return () => {
      // 移除事件监听器
      document.removeEventListener('emit', queryData);
    }
  }, [])
  const queryData = ({detail: {handelData, jumpUrl}}) => {
    console.log(handelData({msg: msgRef.current}));
    jumpUrl('/menu/vue/list')
  }
  const dispatchData = (msg) => () => {
    msgRef.current = msg
    /* 触发自定义事件 通信*/
    document.dispatchEvent(window.cEvt);
  }
  return (
    <Fragment>
       <h2 onClick={dispatchData('hzf')}>跳转</h2>
    </Fragment>
  )
}

export default Index
```

别忘了移除事件监听器

为了防止子应用独立运行的时候报错需要在子应用加载的时候加上错误提示。

```
// mic/micro-app-react/src/index.jsx
// 独立运行时，直接挂载应用
if (!window.__POWERED_BY_QIANKUN__) {
  function createEvent(params, eventName = 'emit') {
    return new CustomEvent(eventName, {detail: params});
  }

  window.cEvt = createEvent({
    handelData: () => console.error('不能运行'),
    jumpUrl: () => console.error('不能运行')
  })
  render();
}
```

这种优势就是纯原生方便包装，使用简单，适合简单的通信。

H4 基于qiankun提供的API

`qiankun` 内部提供了 `initGlobalState` 方法用于注册 `MicroAppStateActions` 实例用于通信，该实例有三个方法，分别是：

- `setGlobalState` ：设置 `globalState` - 设置新的值时，内部将执行 `浅检查` ，如果检查到 `globalState` 发生改变则触发通知，通知到所有的 `观察者` 函数。
- `onGlobalStateChange` ：注册 `观察者` 函数 - 响应 `globalState` 变化，在 `globalState` 发生改变时触发该 `观察者` 函数。
- `offGlobalStateChange` ：取消 `观察者` 函数 - 该实例不再响应 `globalState` 变化

主应用

首先，我们在主应用中注册一个 `MicroAppStateActions` 实例并导出，代码实现如下：

```
// mic/micro-app-vue-main/src/shared/actions.ts
import {initGlobalState} from "qiankun";
import router from '@/router'

const initialState = {
  jumpUrl: (url) => {
    router.history.push(url)
  }
};
const actions = initGlobalState(initialState);

export default actions;
```

在注册 **MicroAppStateActions** 实例后，我们在需要通信的组件中使用该实例，并注册 **观察者** 函数

```
// mic/micro-app-vue-main/src/app.vue
import actions from "@/shared/actions";

export default {
   mounted() {
     actions.onGlobalStateChange((state, prevState) => {
        // state: 变更后的状态; prevState: 变更前的状态
        console.log("主应用观察者: 改变前的 ", prevState);
        console.log("主应用观察者: 改变后的 ", state);
      },
     // 第二个参数表示立即执行
    }
}
```

子应用

我们首先来改造我们的 **Vue** 子应用，首先我们设置一个 **Actions** 实例，代码实现如下：

```
// mic/micro-app-vue-main/src/shared/actions.js
const emptyAction = () => {
  console.warn("当前执行的actions为空!");
}

class Actions {
  // 默认值为空 Action
  actions = {
    onGlobalStateChange: emptyAction,
```

```
    setGlobalState: emptyAction
  };
  // 设置 actions
  setActions(actions) {
    this.actions = actions;
  }
  // 映射
  onGlobalStateChange(...args) {
    return this.actions.onGlobalStateChange(...args);
  }
  // 映射
  setGlobalState(...args) {
    return this.actions.setGlobalState(...args);
  }
}

const actions = new Actions();
export default actions;
```

我们创建 **actions** 实例后，我们需要为其注入真实 **Actions**。我们在入口文件 **main.js** 的 **render** 函数中注入，代码实现如下：

```
// mic/micro-app-vue-main/src/main.js
function render(props) { // mount方法传递进来的
  if (props) {
    // 注入 actions 实例
    actions.setActions(props);
  }
  router = new VueRouter({
    // 运行在主应用中时，添加路由命名空间 /vue
    base: window.__POWERED_BY_QIANKUN__ ? "/menu/vue" : "/",
    mode: "history",
    routes,
  });

 // 解决ElementUI导航栏中的vue-router在3.0版本以上重复点菜单报错问题
  const originalPush = VueRouter.prototype.push
  VueRouter.prototype.push = function push(location) {
    return originalPush.call(this, location).catch(err => err)
  }
  // 挂载应用
  instance = new Vue({
    router,
    store,
    render: (h) => h(App),
  }).$mount("#app");
}
```

然后在列表页引入当前的actions，执行跳转的方法:

```
// /mic/micro-app-vue/src/shared/actions.js
import actions from "@/shared/actions";

export default {
    data() {
      return {
        jumpUrl: null
      }
    },
    inject: ["reload"],
    created() {
      this.$nextTick(this.queryList);
    },
    mounted() {
      actions.onGlobalStateChange((state) => {
        this.jumpUrl = state.jumpUrl
      }, true);
    },
    methods: {
      jumpReactDetail(options) {
        this.jumpUrl(`/menu/react/detail/${options.id}`)
      }
    }
  }
```

这种的优势就是轻量，官方自带，适合业务划分清晰，比较简单的微前端应用

H4 基于redux

父应用

首先我们需要在主应用中创建 `store` 用于管理全局状态池

```
// mic/micro-app-vue-main/src/shared/store.js
import {createStore} from "redux";
import router from '@/router'

const initialState = {
  jumpUrl: (url) => {
    router.history.push(url)
  },
  detail: {}
};
```

```
// 多个reducer用combineReducers合并
const reducer = (state = initialState, action) => {
  switch (action.type) {
    default:
      return state;
    case "SET_DETAIL":
      return {
        ...state,
        detail: action.payload
      };
  }
};

const store = createStore(reducer);

export default store;
```

然后，我们需要将 **store** 实例通过 **props** 传递给子应用，代码实现如下：

```
// mic/micro-app-vue-main/src/micro/app.js
import store from "@/shared/store";

const apps = [
  {
    name: "ReactMicroApp",
    entry: `//${isEnter}:10100`,
    container: "#wrapper",
    activeRule: genActiveRule("/menu/react"),
    // 通过 props 将 shared 传递给子应用
    props: {store},
  },
  {
    name: "VueMicroApp",
    entry: `//${isEnter}:10200`,
    container: "#wrapper",
    activeRule: genActiveRule("/menu/vue"),
    props: {store},
  }
];

export default apps;
```

子应用

子应用一般会有自己的状态管理，主应用通信的也不多，所以直接简单处理提示下就行了。

```
// mic/micro-app-react/src/shared/store.js
const emptyRedux = () => {
  console.warn("当前执行的redux不存在!");
}

class Store {
  actions = {
    dispatch: emptyRedux,
    getState: emptyRedux,
    replaceReducer: emptyRedux,
    subscribe: emptyRedux
  };
  // 重载
  setStore(actions) {
    this.actions = actions;
  }

  dispatch(...args) {
    return this.actions.dispatch(...args);
  }

  getState() {
    return this.actions.getState() || {
      jumpUrl: () => {} // 这里redux传进来的方法，不然子应用单独打开运行会报错
    };
  }

  replaceReducer(...args) {
    return this.actions.replaceReducer(...args);
  }

  subscribe(...args) {
    return this.actions.subscribe(...args);
  }
}

const store = new Store();
export default store;
```

然后在入口文件处注入 `store`

```
// mic/micro-app-react/src/pages/app.jsx
```

```
import React from 'react';
import ReactDOM from 'react-dom';

function render(props) {
  if (props && props.store) {
    // 注入redux 实例
    store.setStore(props.store)
  }
  ReactDOM.render(
      <App/>,
    document.getElementById('root')
  );
}
```

然后在项目中就可以直接引入使用了

```
// mic/micro-app-react/src/pages/detail/index.jsx
import React, {Fragment, useEffect} from "react"
import store from "@/shared/store"

const Index = (props) => {
  useEffect(()=>{
    // 返回值是取消订阅
    const unSubscribe = store.subscribe(() => {
      // 注册订阅函数
      console.log(store.getState(), '订阅方法');
    })
    return ()=> {
      unSubscribe()
    }
  },[])

  const dispatchRedux = () => {
    store.dispatch({
      type: 'SET_DETAIL',
      payload: {data: [1, 2, 3, 4], kkk: 121}
    });
  }
  const jumpUrl = () => {
    store.getState().jumpUrl('/menu/vue/table-detail')
  }
  return (
    <Fragment>
       <h2 onClick={jumpUrl}>跳转</h2>
       <h2 onClick={dispatchRedux}>修改</h2>
    </Fragment>
  )
}
```

```
export default Index
```

这种的优势就是避免状态随意污染，而且redux提供状态跟踪的插件，适合较为复杂的微前端应用。

H3 项目部署

由于 qiankun 是通过 fetch 去获取子应用的引入的静态资源的，所以必须要求这些静态资源支持跨域

如果是自己的脚本，可以通过开发服务端跨域来支持。如果是三方脚本且无法为其添加跨域头，可以将脚本拖到本地，由自己的服务器 serve 来支持跨域。

子应用nginx处添加 Access-Control-Allow-Origin，如果不想设置 '*'，也可以指定对多个ip开放中间逗号隔开就好。

需要注意的一个有关CORS的点：

> 对于附带身份凭证的请求(即服务器设置**Access-Control-Allow-Credentials: true**)，服务器不得设置 Access-Control-Allow-Origin 的值为"*"。否则请求将会失败。

```
server
        {
        listen 10200; # 监听端口
        server_name 120.79.229.197 ; # 请求到达的服务器名
        index index.html index.htm index.php default.html default.htm
default.php;
        root /home/wwwroot/mic200.jing999.cn/dist; # 指定运行路径

        location / {
            try_files $uri $uri/ /index.html; # 重定向

            if ($request_method = 'OPTIONS') {
              add_header 'Access-Control-Allow-Origin' '*';
              add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS';
              add_header 'Access-Control-Allow-Headers'
              'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-
With,If-Modified-Since,Cache-Control,Content-Type';
              add_header 'Access-Control-Max-Age' 1728000;
              add_header 'Content-Type' 'text/plain charset=UTF-8';
              add_header 'Content-Length' 0;
```

```
            return 204;
        }
        if ($request_method = 'POST') {
            add_header 'Access-Control-Allow-Origin' '*';
            add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS';
            add_header 'Access-Control-Allow-Headers'
            'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-
With,If-Modified-Since,Cache-Control,Content-Type';
        }
        if ($request_method = 'GET') {
            add_header 'Access-Control-Allow-Origin' '*';
            add_header 'Access-Control-Allow-Methods' 'GET, POST,
OPTIONS';
            add_header 'Access-Control-Allow-Headers'
            'DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-
With,If-Modified-Since,Cache-Control,Content-Type';
        }

    }

    include rewrite/none.conf;

    include enable-php.conf;


    location ~ .*\.(gif|jpg|jpeg|png|bmp|swf)$
    {
        # 静态图片 允许跨域请求
        add_header Access-Control-Allow-Origin '*';
        add_header Access-Control-Allow-Headers X-Requested-With;
        add_header Access-Control-Allow-Methods GET,POST,OPTIONS;
        expires 30d;
    }

    location ~ .*\.(js|css)?$
    {
        # 允许跨域请求
        add_header Access-Control-Allow-Origin '*';
        add_header Access-Control-Allow-Headers X-Requested-With;
        add_header Access-Control-Allow-Methods GET,POST,OPTIONS;
        expires 12h;
    }

    location ~ /.well-known {
         allow all;
    }

    location ~ /\.
```

```
        {
            deny all;
        }

        access_log off;
}
```

H3 配置中心

版本管理、监控方案（埋点）、回滚方案

H3 源码解析

H3 参考文档

蚂蚁 有知(乾坤) 沙盒内容

基于 qiankun 的微前端最佳实践（万字长文） - 从 0 到 1 篇

微服务的JavaScript框架 single-spa

乾坤文档

一些关于微前端的文章

微前端在小米 CRM 系统的实践

d2峰会 微前端视频有三篇