# qiankun实战

- 简单：任意 js 框架均可使用。微应用接入像使用接入一个 iframe 系统一样简单， 但实际不是 iframe 。
- 完备：几乎包含所有构建微前端系统时所需要的基本能力，如 样式隔离、 js 沙箱、 预加载等。
- 生产可用：已在蚂蚁内外经受过足够大量的线上系统的考验及打磨，健壮性值得信 赖。

## 一.主应用搭建

> 主应用我们采用react作为基座

```
npx create-react-app substrate
npm install react-router-dom qiankun
```

```
import {BrowserRouter,Link} from 'react-router-dom'
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <Link to="/react">React项目</Link>
        <Link to="/vue">Vue项目</Link>
      </BrowserRouter>
      {/* 路由切换时 应用渲染到这里 */}
      <div id="container"></div>
    </div>
  );
}
export default App;
```

> 接入 React 和 Vue 微应用 registerApps.js

```
import { registerMicroApps, start } from
'qiankun';
const loader = (loading) => {
    console.log(loading)
}
registerMicroApps([
    {
        name: 'reactApp',
        entry: '//localhost:4000',
        container: '#container',
        activeRule: '/react',
        loader
    },
    {
        name: 'vueApp',
        entry: '//localhost:5000',
        container: '#container',
        activeRule: '/vue',
        loader
    }
], {
    beforeLoad: () => {
        console.log('beforeLoad')
    },
    beforeMount: () => {
        console.log('beforeMount')
    },
    afterMount: () => {
        console.log('adterMount')
    },
    beforeUnmount: () => {
        console.log('beforeUnmount')
    },
    afterUnmount: () => {
        console.log('afterUnmount')
    }
})
start();
```

# 二.React微应用

```
npx create-react-app m-react
```

## 1.接入协议配置

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
let root
function render(props) {
  const container = props.container
  root = ReactDOM.createRoot(container ?
container.querySelector('#root') :
document.querySelector('#root'));
  root.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>
  );
}
// 如果不是在qiankun中引入
if (!window.__POWERED_BY_QIANKUN__) {
  render({});
}
export async function bootstrap() {
  console.log('react bootstraped')
}
export async function mount(props) {
  render(props)
}
export async function unmount(props) {
  root.unmount()
}
```

## 2. `.env`环境变量配置

```
PORT=4000
WDS_SOCKET_PORT=4000
```

## 3.打包配置

### 改造 react 项目配置文件

```
npm i -D @rescripts/cli --force
```

### .rescriptsrc.js

```
module.exports = {
    webpack: (config) => {
        config.output.library = `m-react`;
        config.output.libraryTarget = 'umd';
        return config;
    },
    devServer: (config) => {
        config.headers = {
            'Access-Control-Allow-Origin': '*',
        };
        return config;
    },
};
```

### package.json

修改启动方式

```
"scripts": {
    "start": "rescripts start",
    "build": "rescripts build",
    "test": "rescripts test",
    "eject": "rescripts eject"
},
```

### 4.配置publicPath

更改加载子应用时，子应用的静态资源路径 `path-path.js`

```
if (window.__POWERED_BY_QIANKUN__) {
  // eslint-disable-next-line no-undef
  __webpack_public_path__ =
window.__INJECTED_PUBLIC_PATH_BY_QIANKUN__;
}
```

> 在index.js中引入 `path-path.js`

## 三.Vue微应用

```
vue create m-vue
```

### 1.接入协议配置

```
import { createApp } from 'vue'
import { createRouter, createWebHistory } from
'vue-router';
import App from './App.vue'
import routes from './router'

let router = null;
let app = null;
let history = null;
```

```
function render(props) {
    const { container } = props;
    history =
createWebHistory(window.__POWERED_BY_QIANKUN__ ?
'/vue' : '/');
    router = createRouter({
        history,
        routes,
    });
    app = createApp(App);
    app.use(router);
    app.mount(container ?
container.querySelector('#app') : '#app');
}
if (!window.__POWERED_BY_QIANKUN__) {
    render({});
}
export async function bootstrap() {
    console.log('vue bootstraped')
}
export async function mount(props) {
    render(props)
}
export async function unmount() {
    app.unmount();
    app = null;
    router = null;
    history.destroy();
}
```

## 2.打包配置

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,
  devServer: {
    port: 20000,
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  },
  configureWebpack: {
    output: {
      library: 'm-vue',
      libraryTarget: 'umd'
    }
  }
})
```

### 3.配置publicPath

更改加载子应用时，子应用的静态资源路径 `path-path.js`

```
if (window.__POWERED_BY_QIANKUN__) {
  // eslint-disable-next-line no-undef
  __webpack_public_path__ =
window.__INJECTED_PUBLIC_PATH_BY_QIANKUN__;
}
```

> 在index.js中引入 `path-path.js`

## 四.样式隔离

- 子应用之间的样式隔离： Dynamic Stylesheet 切换应用时将老应用样式移除
- 主应用和子应用之间的样式隔离
  - css-modules 、Scoped CSS 打包时生成不冲突的选择器名
  - BEM(Block Element Modifier) 规范
  - css-in-js 不在推荐使用
  - Shadow DOM 真正意义上的隔离

```
start({
    sandbox: {
        experimentalStyleIsolation: true,//
【data-qiankun=“应用名”】动态样式表
        strictStyleIsolation: true // shadowDOM
的实现
    }
});
```

```
 const appContent = `<div id="qiankun">
        <div id="inner">内层</div>
        <style>div{color:red}</style>
        </div>`;
// 好比qiankun中获取的html，我们拿到后包裹了一层
const containerElement =
document.createElement('div');
containerElement.innerHTML = appContent;
const appElement = containerElement.firstChild
// 拿出第一个儿子，中的内容
const { innerHTML } = appElement;
appElement.innerHTML = '';
let shadow = appElement.attachShadow({ mode:
'open' }); // 将父容器变为
shadow.innerHTML = innerHTML; // 将内容插入到
shadowDOM中
document.body.appendChild(appElement);
```

```
// open 和 closed 的区别在这个变量上
console.dir(appElement.shadowRoot)
```

## 五.应用通信

### 1.主应用

```
const { onGlobalStateChange, setGlobalState } =
initGlobalState()
setGlobalState({
    name:'jw'
})
onGlobalStateChange((newVal,oldVal)=>{
    console.log(newVal,oldVal,'parent')
})
```

### 2.子应用

```
export async function mount(props) {
  props.onGlobalStateChange((newVal,oldVal)=>{
    console.log(newVal,oldVal,'child')
  })
  props.setGlobalState({name:'j sir'})
  render(props)
}
```

## 七.公共组件

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible"
content="IE=edge">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>静态页面</title>
</head>
<body>
    <div id="static"></div>
    <script >
        const app =
document.getElementById('static')
        window['m-html'] = {
            bootstrap:async ()=>{
                console.log('bootstrap')
            },
            mount:async ()=>{
                app.innerHTML = 'static page'
            },
            unmount:async()=>{
                app.innerHTML = ''
            }
        }
    </script>
</body>
</html>
```

> 运行服务：http-server --port 30000  --cors

```
import { BrowserRouter, Link } from 'react-
router-dom'
import { loadMicroApp } from 'qiankun';
import React, { useEffect } from 'react'
function App() {
  let containerRef = React.createRef();
  useEffect(() => {
    loadMicroApp({
      name: 'staticApp',
      entry: '//localhost:30000',
      container: containerRef.current,
    });
  })
  return (
    <div className="App" >
      <div ref={containerRef}></div>
      <BrowserRouter>
        <Link to="/react">React项目</Link>
        <Link to="/vue">Vue项目</Link>
      </BrowserRouter>
      {/* 路由切换时 应用渲染到这里 */}
      <div id="container"></div>
    </div>
  );
}
export default App;
```

# 八.JS隔离

## 1.SnapshotSandbox

```
class SnapshotSandbox {
  constructor() {
    // 1) 记录沙箱开启时的属性修改
    this.modifyPropsMap = {}
  }
  active() {
    // 2) 激活时创造window的快照
    this.windowSnapshot = {};
    Object.keys(window).forEach((prop) => {
      this.windowSnapshot[prop] = window[prop]
    });
    // 5) 并且将上次修改的内容还原到window上
```

```
 Object.keys(this.modifyPropsMap).forEach((prop)
=>{
      window[prop] = this.modifyPropsMap[prop]
    })
  }
  inactive(){
    this.modifyPropsMap = {};
    Object.keys(window).forEach(prop=>{
      // 3) 失活前window中改变的属性先保存起来
      // 4) 用之前的快照还原window
      if(window[prop] !==
this.windowSnapshot[prop]){
        this.modifyPropsMap[prop] =
window[prop];
        window[prop] =
this.windowSnapshot[prop];
      }
    })
  }
}
const sandbox = new SnapshotSandbox();
sandbox.active(); // 1).激活时创建快照
window.a = 100;   // 2).新增了个a属性
console.log(window.a);
sandbox.inactive(); // 失活时，将window上修改的属性
暂存起来，用快照还原
```

```
console.log(window.a)
sandbox.active();    // 再次激活时，用之前的缓存的修改
属性进行还原
console.log(window.a);
```

## 2.LegacySandbox

```
class LegacySandbox {
  constructor() {
    // 1) 沙箱期间新增的全局变量
    this.addedPropsMapInSandbox = new Map();
    // 2) 沙箱期间更新的全局变量
    this.modifiedPropsOriginalValueMapInSandbox
= new Map()
    // 3) 有修改就记录
    this.currentUpdatedPropsValueMap = new
Map();

    const fakeWindow = Object.create(null);
    const proxy = new Proxy(fakeWindow, {
      get:(target, key)=> {
        return window[prop]
      },
      set:(target, key, value)=> {
        // 1).如果window中没有此属性,加入到新增列表
        if (!window.hasOwnProperty(key)) {
          this.addedPropsMapInSandbox.set(key,
value);
          // 2).如果是修改则保存修改属性
        } else if
(!this.modifiedPropsOriginalValueMapInSandbox.ha
s(key)) {

 this.modifiedPropsOriginalValueMapInSandbox.set
(key, window[key]);
        }

 this.currentUpdatedPropsValueMap.set(key,
value)
        window[key] = value;
        return true
      },

    });

    this.proxy = proxy;
  }
  setWindowProp(prop, value, isToDelete) {
    if (value === undefined && isToDelete) {
      delete window[prop]
    } else {
      window[prop] = value
    }
  }
  active() {
    // 恢复上一次该微应用处于运行状态时，对window 上做
的所有应用的修改

 this.currentUpdatedPropsValueMap.forEach((value
, prop) => {
      this.setWindowProp(prop, value)
    })
  }
  inactive() {
    // 还原window上的属性

 this.modifiedPropsOriginalValueMapInSandbox.for
Each((value, prop) => {
      this.setWindowProp(prop, value)
    })
    // 删除 window 新增的属性
    this.addedPropsMapInSandbox.forEach((value,
prop) => {
      this.setWindowProp(prop, undefined, true)
    })
  }
}
window.a = 100
let legacySandbox = new LegacySandbox()
console.log(window.a)
```

```
legacySandbox.active()
legacySandbox.proxy.a = 200
console.log(window.a)
legacySandbox.inactive()
console.log(window.a)
```

## 3.ProxySandbox

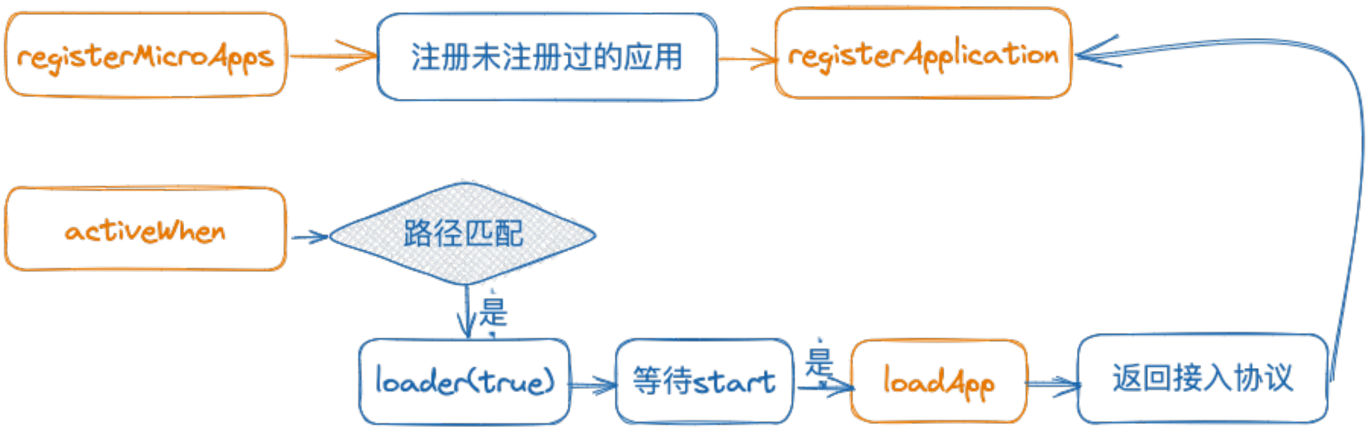
```
class ProxySandbox {
  constructor(){
    this.sandboxRunning = false;
    const fakeWindow = Object.create(null)
    const proxy = new Proxy(fakeWindow,{
      get:(target,key)=>{
        return key in target ? target[key] :
window[key]
      },
      set:(target,key,value)=>{
        if(this.sandboxRunning){
          target[key] = value
        }
        return true
      }
    });
    this.proxy = proxy;
  }
  active(){
    if(!this.sandboxRunning) this.sandboxRunning
= true
  }
  inactive(){
    this.sandboxRunning = false;
  }
}
const sandbox1 = new ProxySandbox();
const sandbox2 = new ProxySandbox();

sandbox1.active()
sandbox2.active()
// 沙箱激活后值的修改 都会保存在fakeWidow中
sandbox1.proxy.a = 100;
sandbox2.proxy.a = 100;

sandbox1.inactive()
sandbox2.inactive()
// 失效后修改属性不会被记录到fakeWindow中
sandbox1.proxy.a = 200;
sandbox2.proxy.a = 200;
console.log(sandbox1.proxy.a)
console.log(sandbox1.proxy.a)
```

## 九.源码分析

## 1.解析registerMicroApps



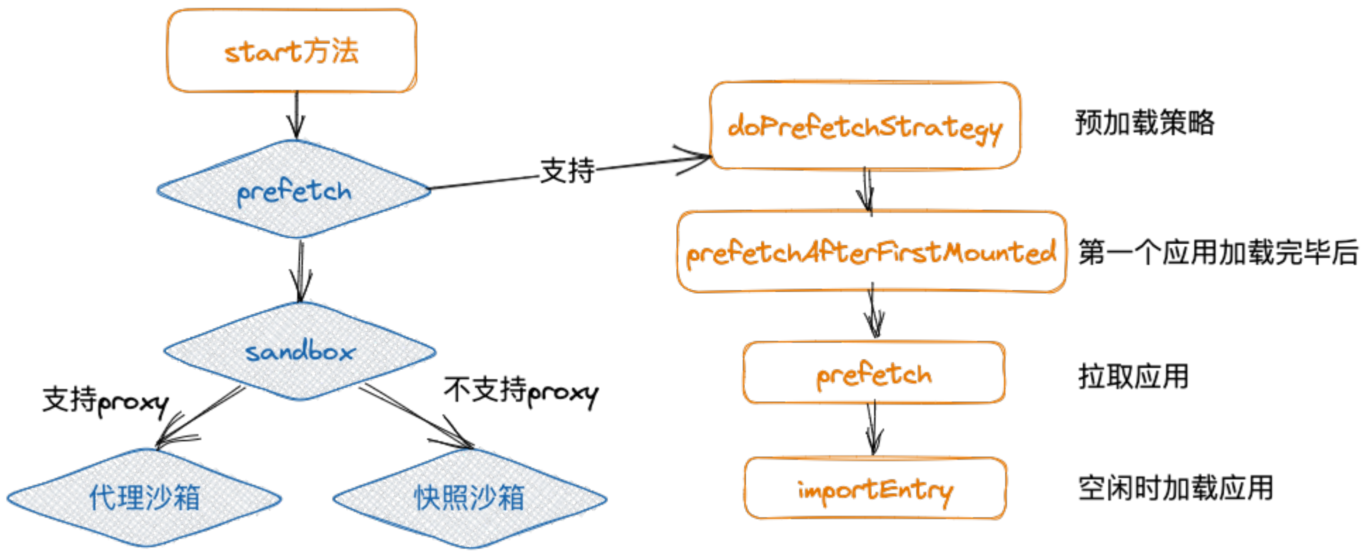
```
export function registerMicroApps<T extends ObjectType>(
  apps: Array<RegistrableApp<T>>,      // 需要注册的应用
  lifeCycles?: FrameworkLifeCycles<T>, // 注册的生命周期
) {
  // 每个应用只注册一次，将本次注册的和已经注册的去重过滤
  const unregisteredApps = apps.filter((app) => !microApps.some((registeredApp) => registeredApp.name === app.name));

  microApps = [...microApps, ...unregisteredApps]; // 缓存注册的应用

  // 循环每一个未注册的应用，进行注册
  unregisteredApps.forEach((app) => {
    const { name, activeRule, loader = noop,
props, ...appConfig } = app;
    registerApplication({ // single-spa中的
registerApplication
      name,
      app: async () => {
        loader(true);
        // 等待start方法
        await frameworkStartedDefer.promise;
        // 加载app拿到返回结果，也就是获取接入协议
        const { mount, ...otherMicroAppConfigs }
= (
          await loadApp({ name, props,
...appConfig }, frameworkConfiguration,
lifeCycles)
        )();
        return {
          // 增加loading
          mount: [async () => loader(true),
...toArray(mount), async () => loader(false)],
          ...otherMicroAppConfigs,
        };
      },
      activeWhen: activeRule,
      customProps: props,
    });
  });
```

```
}
```

## 2.解析start方法



```
export function start(opts:
FrameworkConfiguration = {}) {
  // prefetch：第一个应用加载完毕后，默认会加载其他应用
  // singular：是否是单例模式
  // sandbox： 默认开启沙箱
  frameworkConfiguration = { prefetch: true,
singular: true, sandbox: true, ...opts };
  const {
    prefetch,
    sandbox,
    singular,
    urlRerouteOnly = defaultUrlRerouteOnly,
    ...importEntryOpts
  } = frameworkConfiguration;
  if (prefetch) { // 需要预加载，会做一个预加载策略
    doPrefetchStrategy(microApps, prefetch,
importEntryOpts);
  }
  // 沙箱自动降级为低版本浏览器
  frameworkConfiguration =
autoDowngradeForLowVersionBrowser(frameworkConfi
guration);

  startSingleSpa({ urlRerouteOnly }); //  调用
single-spa的start方法
  started = true;

  frameworkStartedDefer.resolve(); // 调用start完
毕
}
```

## 1）预加载

```
export function doPrefetchStrategy(
  apps: AppMetadata[],
  prefetchStrategy: PrefetchStrategy,
  importEntryOpts?: ImportEntryOpts,
) {
  const appsName2Apps = (names: string[]):
AppMetadata[] => apps.filter((app) =>
names.includes(app.name));
  // 是数组时，当第一个应用加载完成后加载数组中对应的应用
  if (Array.isArray(prefetchStrategy)) {

 prefetchAfterFirstMounted(appsName2Apps(prefetc
hStrategy as string[]), importEntryOpts);
  } else if (isFunction(prefetchStrategy)) {
    (async () => {
      // critical rendering apps would be
prefetch as earlier as possible
      const { criticalAppNames = [],
minorAppsName = [] } = await
prefetchStrategy(apps);
      // 关键应用尽可能早渲染

 prefetchImmediately(appsName2Apps(criticalAppNa
mes), importEntryOpts);
      // 其它应用等待第一个应用加载完毕后加载

 prefetchAfterFirstMounted(appsName2Apps(minorAp
psName), importEntryOpts);
    })();
  } else {
    switch (prefetchStrategy) { // 预加载参数为
true
```

```
      case true:
        prefetchAfterFirstMounted(apps,
importEntryOpts);
        break;

      case 'all':
        prefetchImmediately(apps,
importEntryOpts);
        break;

      default:
        break;
    }
  }
}
```

```
function prefetchAfterFirstMounted(apps:
AppMetadata[], opts?: ImportEntryOpts): void {
  // 等待第一个应用挂载完成  single-spa触发此事件
  window.addEventListener('single-spa:first-
mount', function listener() {
    // 获取所有没有加载的app
    const notLoadedApps = apps.filter((app) =>
getAppStatus(app.name) === NOT_LOADED);

    if (process.env.NODE_ENV === 'development')
{
      const mountedApps = getMountedApps();
      console.log(`[qiankun] prefetch starting
after ${mountedApps} mounted...`,
notLoadedApps);
    }
    // 没有加载过的app 做预加载
    notLoadedApps.forEach(({ entry }) =>
prefetch(entry, opts));
    // 移除事件
    window.removeEventListener('single-
spa:first-mount', listener);
  });
}function prefetchAfterFirstMounted(apps:
AppMetadata[], opts?: ImportEntryOpts): void {
  // 等待第一个应用挂载完成  single-spa触发此事件
  window.addEventListener('single-spa:first-
mount', function listener() {
    // 获取所有没有加载的app
    const notLoadedApps = apps.filter((app) =>
getAppStatus(app.name) === NOT_LOADED);

    if (process.env.NODE_ENV === 'development')
{
      const mountedApps = getMountedApps();
      console.log(`[qiankun] prefetch starting
after ${mountedApps} mounted...`,
notLoadedApps);
```

```
    }
    // 没有加载过的app 做预加载
    notLoadedApps.forEach(({ entry }) =>
prefetch(entry, opts));
    // 移除事件
    window.removeEventListener('single-
spa:first-mount', listener);
  });
}
```

```
function prefetch(entry: Entry, opts?:
ImportEntryOpts): void {
  if (!navigator.onLine || isSlowNetwork) { //
慢网，或者不在线直接结束
    // Don't prefetch if in a slow network or
offline
    return;
  }
  // 空闲的时候开始加载
  requestIdleCallback(async () => { // 使用
importEntry 替换掉Systemjs
    const { getExternalScripts,
getExternalStyleSheets } = await
importEntry(entry, opts);
    requestIdleCallback(getExternalStyleSheets);
    requestIdleCallback(getExternalScripts);
  });
}
```

> 这里通过 **importEntry** 加载子应用

## 2）沙箱选择

```
const autoDowngradeForLowVersionBrowser =
(configuration: FrameworkConfiguration):
FrameworkConfiguration => {
  const { sandbox, singular } = configuration;
  if (sandbox) { // 查看是否支持沙箱
    if (!window.Proxy) { // 不支持proxy，则采用快照沙箱
      console.warn('[qiankun] Miss window.Proxy, proxySandbox will degenerate into snapshotSandbox');
      if (singular === false) { // 快照沙箱不支持多例模式
        console.warn(
          '[qiankun] Setting singular as false may cause unexpected behavior while your browser not support window.Proxy',
        );
      }
      // loose:true 就是快照模式
      return { ...configuration, sandbox: typeof sandbox === 'object' ? { ...sandbox, loose: true } : { loose: true } };
    }
  }
  return configuration;
};
```

## 3.loadApp实现

```
export async function loadApp<T extends ObjectType>(
  app: LoadableApp<T>,
  configuration: FrameworkConfiguration = {},
  lifeCycles?: FrameworkLifeCycles<T>,
): Promise<ParcelConfigObjectGetter> {
  const { entry, name: appName } = app;
  const appInstanceId =
genAppInstanceIdByName(appName); // 根据应用名生成
实例的id

  const {
    singular = false,
    sandbox = true,
    excludeAssetFilter,
    globalContext = window,
    ...importEntryOpts
  } = configuration;

  // 通过子应用的entry入口，拿到用户写的模板 （html js
css）
  // template 注释掉外链的html
  // execScripts 执行脚本运行传递proxy
  // assetPublicPath 静态资源路径
  // getExternalScripts 获取额外的资源
  const { template, execScripts,
assetPublicPath, getExternalScripts } = await
importEntry(entry, importEntryOpts);
  // 获取额外的脚本，在调用execScripts保证资源加载完成
  await getExternalScripts();

  // 如果是单例模式 需要先卸载完成
  if (await validateSingularMode(singular, app))
{
    await (prevAppUnmountedDeferred &&
prevAppUnmountedDeferred.promise);
  }
  // 增加了一个div标签，包裹模板
  const appContent =
getDefaultTplWrapper(appInstanceId, sandbox)
(template);
  // 是否采用严格样式隔离 - shadowDOM
  const strictStyleIsolation = typeof sandbox
=== 'object' && !!sandbox.strictStyleIsolation;

  if (process.env.NODE_ENV === 'development' &&
strictStyleIsolation) {
    console.warn(
      "[qiankun] strictStyleIsolation
configuration will be removed in 3.0, pls don't
depend on it or use experimentalStyleIsolation
instead!",
    );
  }
  // 是否启用实验型css隔离
  const scopedCSS = isEnableScopedCSS(sandbox);
  // 创建容器处理严格样式隔离及作用域样式隔离
  let initialAppWrapperElement: HTMLElement |
null = createElement(
    appContent,
    strictStyleIsolation,
    scopedCSS,
    appInstanceId,
  );
  // 初始化应用的容器
  const initialContainer = 'container' in app ?
app.container : undefined;
  const legacyRender = 'render' in app ?
app.render : undefined;

  const render = getRender(appInstanceId,
appContent, legacyRender);

  // 第一次加载设置应用可见区域 dom 结构
  // 确保每次应用加载前容器 dom 结构已经设置完毕
  render({ element: initialAppWrapperElement,
loading: true, container: initialContainer },
'loading');

  // 获取包裹容器，为了兼容ShadowDOM根元素
  const initialAppWrapperGetter =
getAppWrapperGetter(
    appInstanceId,
    !!legacyRender,
    strictStyleIsolation,
    scopedCSS,
    () => initialAppWrapperElement,
  );

  let global = globalContext;
  let mountSandbox = () => Promise.resolve();
  let unmountSandbox = () => Promise.resolve();
  const useLooseSandbox = typeof sandbox ===
'object' && !!sandbox.loose; // 快照沙箱
  // enable speedy mode by default
  const speedySandbox = typeof sandbox ===
'object' ? sandbox.speedy !== false : true;
  let sandboxContainer;
  if (sandbox) {
    // 创建沙箱
    sandboxContainer = createSandboxContainer(
      appInstanceId,
      // FIXME should use a strict sandbox logic
while remount, see
https://github.com/umijs/qiankun/issues/518
      initialAppWrapperGetter,
      scopedCSS,
      useLooseSandbox,
      excludeAssetFilter,
      global,
      speedySandbox,
    );
    // 用沙箱的代理对象作为接下来使用的全局对象
    global = sandboxContainer.instance.proxy as
typeof window;
    mountSandbox = sandboxContainer.mount;
    unmountSandbox = sandboxContainer.unmount;
  }
  // 全局钩子
  const {
    beforeUnmount = [],
    afterUnmount = [],
    afterMount = [],
    beforeMount = [],
    beforeLoad = [],
  } = mergeWith({}, getAddOns(global,
assetPublicPath), lifeCycles, (v1, v2) =>
concat(v1 ?? [], v2 ?? []));
  // 增添qiankun中的标识

  // 执行beforeLoad的链
  await execHooksChain(toArray(beforeLoad), app,
global);

  // 在沙箱中执行用户脚本
```

```
  const scriptExports: any = await
execScripts(global, sandbox && !useLooseSandbox,
{
    scopedGlobalVariables: speedySandbox ?
cachedGlobals : [],
  });
  // 获取接入协议
  const { bootstrap, mount, unmount, update } =
getLifecyclesFromExports(
    scriptExports,
    appName,
    global,
    sandboxContainer?.instance?.latestSetProp,
  );
  // 创建应用之间的状态管理
  const { onGlobalStateChange, setGlobalState,
offGlobalStateChange }: Record<string,
CallableFunction> =
    getMicroAppStateActions(appInstanceId);
  // ......
}
```

```
const parcelConfigGetter:
ParcelConfigObjectGetter = (remountContainer =
initialContainer) => {
  let appWrapperElement: HTMLElement | null;
  let appWrapperGetter: ReturnType<typeof
getAppWrapperGetter>;

  const parcelConfig: ParcelConfigObject = {
    name: appInstanceId,
    bootstrap,
    mount: [
      // 单例模式只能挂载一个，需要等待卸载后才能挂载
      async () => {
        if ((await
validateSingularMode(singular, app)) &&
prevAppUnmountedDeferred) {
          return
prevAppUnmountedDeferred.promise;
        }
        return undefined;
      },
      // 在挂在和重新挂载前获取包裹容器
      async () => {
        appWrapperElement =
initialAppWrapperElement;
        appWrapperGetter = getAppWrapperGetter(
          appInstanceId,
          !!legacyRender,
          strictStyleIsolation,
          scopedCSS,
          () => appWrapperElement,
        );
      },
      // 添加 mount hook, 确保每次应用加载前容器 dom
结构已经设置完毕
      async () => {
        const useNewContainer = remountContainer
!== initialContainer;
        if (useNewContainer ||
!appWrapperElement) {
          appWrapperElement =
createElement(appContent, strictStyleIsolation,
scopedCSS, appInstanceId);

 syncAppWrapperElement2Sandbox(appWrapperElement
);
        }
        render({ element: appWrapperElement,
loading: true, container: remountContainer },
'mounting');
      },
      // 挂载沙箱
      mountSandbox,
      // 执行beforeMount
      async () =>
execHooksChain(toArray(beforeMount), app,
global),
      // 调用挂载逻辑
      async (props) => mount({ ...props,
container: appWrapperGetter(), setGlobalState,
onGlobalStateChange }),
      async () => render({ element:
appWrapperElement, loading: false, container:
remountContainer }, 'mounted'),
      // 执行afterMount
      async () =>
execHooksChain(toArray(afterMount), app,
global),
      // 单例模式生成一个稍后用于卸载的Promise
      async () => {
        if (await validateSingularMode(singular,
app)) {
          prevAppUnmountedDeferred = new
Deferred<void>();
        }
      },
      async () => {
        if (process.env.NODE_ENV ===
'development') {
          const measureName = `[qiankun] App
${appInstanceId} Loading Consuming`;
          performanceMeasure(measureName,
markName);
        }
      },
    ],
    unmount: [
      // 执行beforeUnmount
      async () =>
execHooksChain(toArray(beforeUnmount), app,
global),
      // unmount
      async (props) => unmount({ ...props,
container: appWrapperGetter() }),
      // 卸载沙箱
      unmountSandbox,
      // 执行afterUnmount
      async () =>
execHooksChain(toArray(afterUnmount), app,
global),
      async () => {
        // 关闭事件监听操作等
        render({ element: null, loading: false,
container: remountContainer }, 'unmounted');
        offGlobalStateChange(appInstanceId);
        // for gc
        appWrapperElement = null;

 syncAppWrapperElement2Sandbox(appWrapperElement
);
      },
      // 卸载完成更改状态
      async () => {
        if ((await
validateSingularMode(singular, app)) &&
prevAppUnmountedDeferred) {
          prevAppUnmountedDeferred.resolve();
        }
      },
    ],
  };
  if (typeof update === 'function') {
    parcelConfig.update = update;
  }
  return parcelConfig;
};
```

- 通过 importEntry 方法拉取子应用
- 在拉取的模板外面包一层 div ,增加 css 样式隔离 shadowdom 、 scopedCSS将模板进行挂载。
- 创建 js 沙箱 ,获得沙箱开启和沙箱关闭方法
- 合并出 beforeUnmount 、 afterUnmount 、 afterMount 、 beforeMount 、
- beforeLoad 方法。增加 qiankun 标识
- 依次调用 beforeLoad 方法

- 在沙箱中执行脚本， 获取子应用的生命周期 bootstrap 、 mount 、 unmount 、update
- 格式化子应用的 mount 方法和 unmount 方法。
    - 在mount执行前挂载沙箱、依次执行 beforeMount ，之后调用mount方法，将

      全局通信方法传入。mount方法执行完毕后执行 afterMount
    - unmount方法会优先执行 beforeUnmount 钩子，之后开始卸载。
- 增添一个 update 方法

## 十.常见问题

- 依赖复用的问题
    - 创建共享模块，独立打包部署到CDN上，通过加载应用时传入，或者在子应用中引入。
    - 通过联邦模块进行打包处理公共资源。
    - 两个应用之间加载资源的地址相同即可复用（http缓存）
- 应用之间的组件复用问题
    - 应用中将共享的组件进行单独打包，加载应用时进行传入
- Vite支持问题

    - 基于vite构建的项目中`import、export`并没有被转码，会导致直接报错，可以采用生产环境接入vite）
- qiankun嵌套的问题
    - 需要避免多重沙箱嵌套问题，子应用中需要关闭沙箱。
- css沙箱不完美
    - strictStyleIsolation完全隔离问题，样式无法传递到子应用中。
    - experimentalStyleIsolation 子应用dom 结构插入到body中，样式无法生效。

> 后续将移除 globalState、addGlobalUncaughtErrorHandler、shadowDOM样式隔离方案。