从零实现微前端框架

一.初始化开发环境

初始化配置安装 `rollup`

```sh
npm init -y
npm install rollup rollup-plugin-serve
```

```js
import serve from 'rollup-plugin-serve'
export default {
    input:'./src/single-spa.js',
    output:{
        file:'./lib/umd/single-spa.js',
        format:"umd",
        name:'singleSpa',
        sourcemap:true
    },
    plugins:[
        serve({
            openPage:'/index.html',
            contentBase:'',
            port:3000
        })
    ]
}
```

> 这里我们一切从简，只借助 `rollup` 模块化和打包的能力~，不进行过多的 `rollup` 配置，把精力放到编写微前端的核心逻辑上~~~

二. SignleSpa 的使用方式

```js
singleSpa.registerApplication('app1',
    async () => {
        return {
```

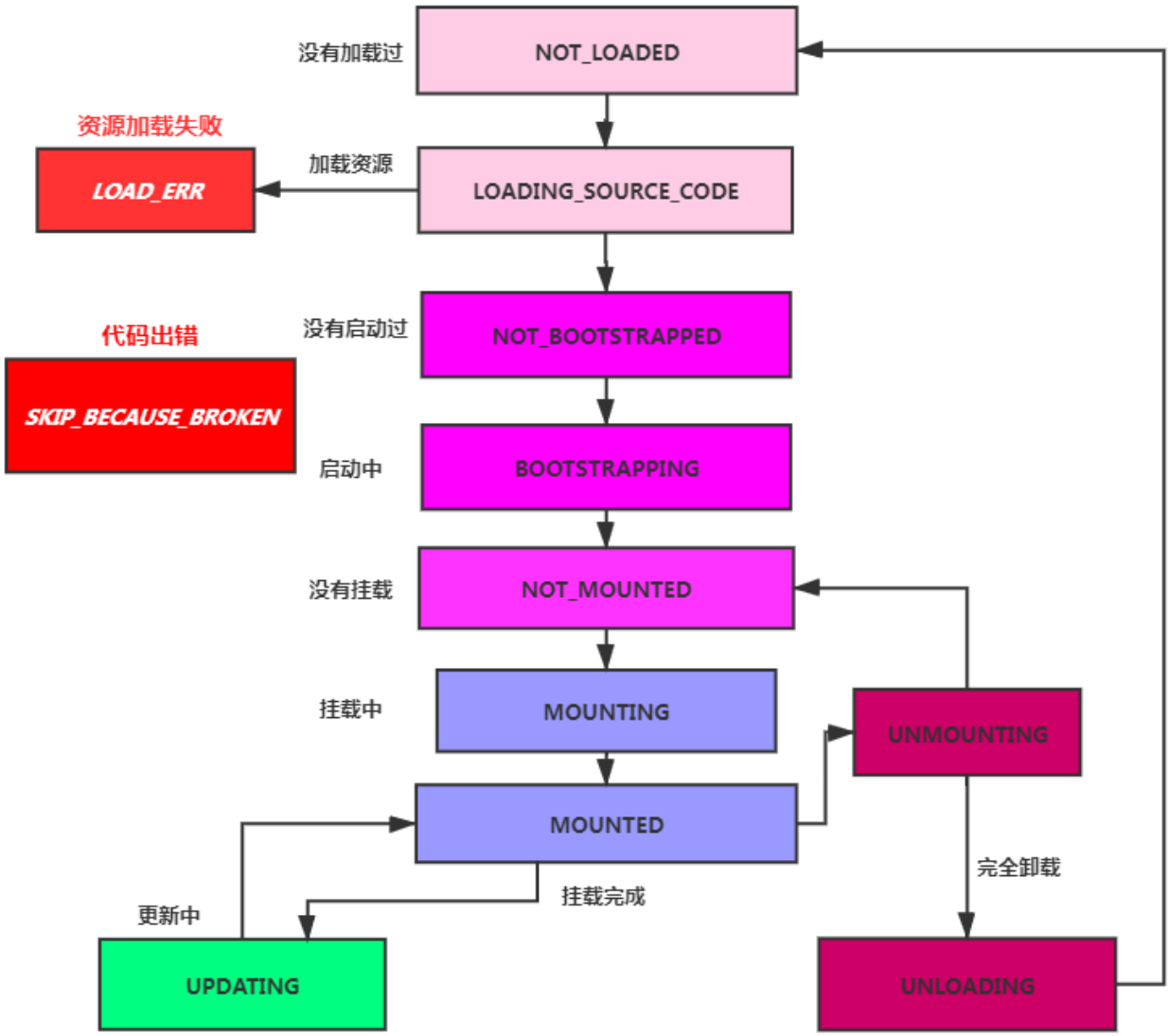
```
            bootstrap:async()=>{
                console.log('应用启动');
            },
            mount:async()=>{
                console.log('应用挂载');
            },
            unmount:async()=>{
                console.log('应用卸载')
            }
        }
    },
    location => location.hash.startsWith('#/app1'),
    { store: { name: 'zf' } }
);
singleSpa.start();
```

- 参数分别是:
- `appName` : 当前注册应用的名字
- `loadApp` : 加载函数(必须返回的是promise)，返回的结果必须包含 `bootstrap` 、 `mount` 和 `unmount` 做为接入协议
- `activityWhen` : 满足条件时调用 `loadApp` 方法
- `customProps` :自定义属性可用于父子应用通信

根据使用方式编写源码

```js
const apps = [];
export function registerApplication(appName,loadApp,activeWhen,customProps){
    apps.push({
        name:appName,
        loadApp,
        activeWhen,
        customProps,
    });
}
export function start(){
    // todo...
}
export {registerApplication} from './applications/app.js';
export {start} from './start.js';
```

三.应用加载状态 - 生命周期




```js
export const NOT_LOADED = "NOT_LOADED"; // 没有加载过
export const LOADING_SOURCE_CODE = "LOADING_SOURCE_CODE"; // 加载原代码
export const NOT_BOOTSTRAPPED = "NOT_BOOTSTRAPPED"; // 没有启动
export const BOOTSTRAPPING = "BOOTSTRAPPING"; // 启动中
export const NOT_MOUNTED = "NOT_MOUNTED"; // 没有挂载
export const MOUNTING = "MOUNTING"; // 挂载中
export const MOUNTED = "MOUNTED"; // 挂载完毕
export const UPDATING = "UPDATING"; // 更新中
export const UNMOUNTING = "UNMOUNTING"; // 卸载中
export const UNLOADING = "UNLOADING"; // 没有加载中
export const LOAD_ERROR = "LOAD_ERROR"; // 加载失败
export const SKIP_BECAUSE_BROKEN = "SKIP_BECAUSE_BROKEN"; // 运行出错

export function isActive(app) { // 当前app是否已经挂载
    return app.status === MOUNTED;
}
export function shouldBeActive(app) { // 当前app是否应该激活
    return app.activeWhen(window.location);
}
```

标注应用状态

```js
import { NOT_LOADED } from './app.helpers';
apps.push({
    name: appName,
    loadApp,
    activeWhen,
    customProps,
    status: NOT_LOADED // 默认应用为未加载
});
```

四.加载应用并启动

```js
import {reroute} from '../navigation/reroute.js';
export function registerApplication(appName, loadApp, activeWhen, customProps)
    // ...
    reroute(); // 这个是加载应用
}
```

```js
import {reroute} from './navigation/reroute'
export let started = false;
export function start(){
    started = true;
    reroute(); // 这个是启动应用
}
```

> reroute方法就是比较核心的一个方法啦~，当注册应用时reroute的功能是加载子应用，当调用start方法时是挂载应用。

五.reroute方法

这个方法是整个 `Single-SPA` 中最核心的方法,当路由切换时也会执行该逻辑

1).获取对应状态的 app

```js
import {getAppChanges} from '../applications/apps';
export function reroute() {
    const {
        appsToLoad, // 获取要去加载的app
        appsToMount, // 获取要被挂载的
        appsToUnmount // 获取要被卸载的
    } = getAppChanges();
}
```

```js
export function getAppChanges(){
    const appsToUnmount = [];
    const appsToLoad = [];
    const appsToMount = [];
    apps.forEach(app => {
        const appShouldBeActive = app.status !== SKIP_BECAUSE_BROKEN && should
        switch (app.status) { // toLoad
            case STATUS.NOT_LOADED:
            case STATUS.LOADING_SOURCE_CODE:
                if(appShouldBeActive){
                    appsToLoad.push(app);
                }
                break;
            case STATUS.NOT_BOOTSTRAPPED: // toMount
            case STATUS.NOT_MOUNTED:
                if(appShouldBeActive){
                    appsToMount.push(app);
                }
                break
            case STATUS.MOUNTED: // toUnmount
                if(!appShouldBeActive){
                    appsToUnmount.push(app);
                }
        }
    });
    return {appsToUnmount,appsToLoad,appsToMount}
}
```

> 根据状态筛选对应的应用

2). 预加载应用

当用户没有调用 `start` 方法时，我们默认会先进行应用的加载

```js
if(started){
     return performAppChanges();
}else{
     return loadApps();
}
async function performAppChanges(){
    // 启动逻辑
}
async function loadApps(){
    // 预加载应用
}
```

```js
import {toLoadPromise} from '../lifecycles/load';
async function loadApps(){
    // 预加载应用
    await Promise.all(appsToLoad.map(toLoadPromise));
}
```

```js
import { LOADING_SOURCE_CODE, NOT_BOOTSTRAPPED } from "../applications/app.hel
function flattenFnArray(fns) { // 将函数通过then链连接起来
    fns = Array.isArray(fns) ? fns : [fns];
    return function(props) {
        return fns.reduce((p, fn) => p.then(() => fn(props)), Promise.resolve(
    }
}
export async function toLoadPromise(app) {
    app.status = LOADING_SOURCE_CODE;
    let { bootstrap, mount, unmount } = await app.loadApp(app.customProps); //
    app.status = NOT_BOOTSTRAPPED;
    app.bootstrap = flattenFnArray(bootstrap);
    app.mount = flattenFnArray(mount);
    app.unmount = flattenFnArray(unmount);
    return app;
}
```

> 用户load函数返回的 `bootstrap` 、 `mount` 、 `unmount` 可能是数组形式，我们将这些函数进行组合

3). app 运转逻辑

路由切换时卸载不需要的应用

```js
import {toUnmountPromise} from '../lifecycles/unmount';
import {toUnloadPromise} from '../lifecycles/unload';
async function performAppChanges(){
        // 卸载不需要的应用，挂载需要的应用
    let unmountPromises = appsToUnmount.map(toUnmountPromise).map(unmountPromi
}
```

> 这里为了更加直观，我就采用最简单的方法来实现，调用钩子，并修改应用状态

```js
import { UNMOUNTING, NOT_MOUNTED ,MOUNTED} from "../applications/app.helpers";
export async function toUnmountPromise(app){
    if(app.status != MOUNTED){
        return app;
    }
    app.status = UNMOUNTING;
    await app.unmount(app);
    app.status = NOT_MOUNTED;
    return app;
}
```

```
import { NOT_LOADED, UNLOADING } from "../applications/app.helpers";
const appsToUnload = {};
export async function toUnloadPromise(app){
    if(!appsToUnload[app.name]){
        return app;
    }
    app.status = UNLOADING;
    delete app.bootstrap;
    delete app.mount;
    delete app.unmount;
    app.status = NOT_LOADED;
}
```

匹配到没有加载过的应用 (加载=> 启动 => 挂载)

```js
const loadThenMountPromises = appsToLoad.map(async (app) => {
    app = await toLoadPromise(app);
    app = await toBootstrapPromise(app);
```

```
    return toMountPromise(app);
});
```

这里需要注意一下，可能还有没加载完的应用这里不要进行重复加载

```js
export async function toLoadPromise(app) {
    if(app.loadPromise){
        return app.loadPromise;
    }
    if (app.status !== NOT_LOADED) {
        return app;
    }
    app.status = LOADING_SOURCE_CODE;
    return (app.loadPromise = Promise.resolve().then(async ()=>{
        let { bootstrap, mount, unmount } = await app.loadApp(app.customProps)

        app.status = NOT_BOOTSTRAPPED;
        app.bootstrap = flattenFnArray(bootstrap);
        app.mount = flattenFnArray(mount);
        app.unmount = flattenFnArray(unmount);
        delete app.loadPromise;
        return app;
    }));
}
```

```js
import { BOOTSTRAPPING, NOT_MOUNTED,NOT_BOOTSTRAPPED } from "../applications/a
export async function toBootstrapPromise(app) {
    if(app.status !== NOT_BOOTSTRAPPED){
        return app;
    }
    app.status = BOOTSTRAPPING;
    await app.bootstrap(app.customProps);
    app.status = NOT_MOUNTED;
    return app;
}
```

```js
import { MOUNTED, MOUNTING,NOT_MOUNTED } from "../applications/app.helpers.js"
export async function toMountPromise(app) {
    if (app.status !== NOT_MOUNTED) {
        return app;
    }
```

```
    app.status = MOUNTING;
    await app.mount();
    app.status = MOUNTED;
    return app;
}
```

已经加载过了的应用 (启动 => 挂载)

```js
const mountPromises = appsToMount.map(async (app) => {
    app = await toBootstrapPromise(app);
    return toMountPromise(app);
});
await Promise.all(unmountPromises); // 等待先卸载完成
await Promise.all([...loadThenMountPromises,...mountPromises]);
```

六.路由劫持

```js
import { reroute } from "./reroute.js";
export const routingEventsListeningTo = ["hashchange", "popstate"];
const capturedEventListeners = { // 存储hashchang和popstate注册的方法
    hashchange: [],
    popstate: []
}
function urlReroute() {
    reroute([], arguments)
}
// 劫持路由变化
window.addEventListener('hashchange', urlReroute);
window.addEventListener('popstate', urlReroute);
// 重写addEventListener方法
const originalAddEventListener = window.addEventListener;
const originalRemoveEventListener = window.removeEventListener;

window.addEventListener = function(eventName, fn) {
    if (routingEventsListeningTo.indexOf(eventName) >= 0 && !capturedEventList
        capturedEventListeners[eventName].push(fn);
        return;
    }
    return originalAddEventListener.apply(this, arguments);
}
window.removeEventListener = function(eventName, listenerFn) {
    if (routingEventsListeningTo.indexOf(eventName) >= 0) {
```

```
            capturedEventListeners[eventName] = capturedEventListeners[
                eventName
            ].filter((fn) => fn !== listenerFn);
            return;
        }
        return originalRemoveEventListener.apply(this, arguments);
    };
    function patchedUpdateState(updateState, methodName) {
        return function() {
            const urlBefore = window.location.href;
            const result = updateState.apply(this, arguments);
            const urlAfter = window.location.href;
            if (urlBefore !== urlAfter) {
                urlReroute(new PopStateEvent('popstate', { state }));
            }
            return result;
        }
    }
    // 重写pushState 和 repalceState方法
    window.history.pushState = patchedUpdateState(window.history.pushState, 'pushS
    window.history.replaceState = patchedUpdateState(window.history.replaceState,

    // 在子应用加载完毕后调用此方法，执行拦截的逻辑（保证子应用加载完后执行）
    export function callCapturedEventListeners(eventArguments) {
        if (eventArguments) {
            const eventType = eventArguments[0].type;
            if (routingEventsListeningTo.indexOf(eventType) >= 0) {
                capturedEventListeners[eventType].forEach((listener) => {
                    listener.apply(this, eventArguments);
                });
            }
        }
    }
```

> 为了保证应用加载逻辑最先被处理，我们对路由的一系列的方法进行重写，确保加载应用的逻辑最先被调用，其次手动派发事件

七.加载应用

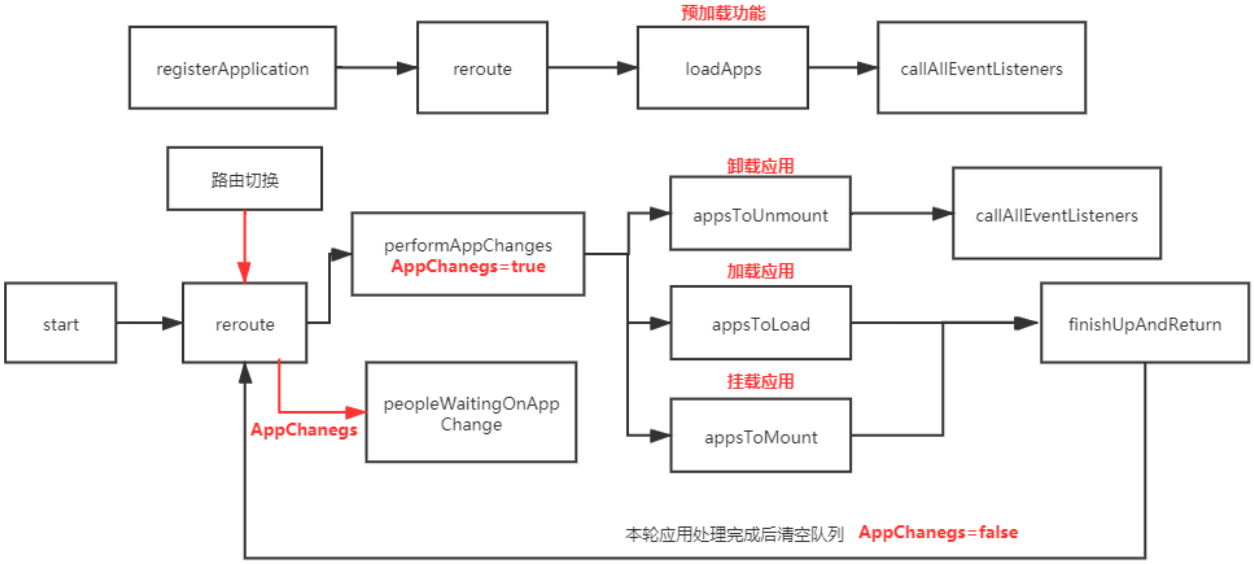
```
await Promise.all(appsToLoad.map(toLoadPromise)); // 加载后触发路由方法
callCapturedEventListeners(eventArguments);
```

```
await Promise.all(unmountPromises); // 等待先卸载完成后触发路由方法
callCapturedEventListeners(eventArguments);
```

校验当前是否需要被激活,在进行启动和挂载

```js
async function tryToBootstrapAndMount(app) {
    if (shouldBeActive(app)) {
        app = await toBootstrapPromise(app);
        return toMountPromise(app);
    }
    return app;
}
```

八.批处理加载等待




```js
export function reroute(pendings = [], eventArguments) {
    if (appChangeUnderway) {
        return new Promise((resolve, reject) => {
            peopleWaitingOnAppChange.push({
                resolve,
                reject,
                eventArguments
            })
        });
    }
    // ...
    if (started) {
```

```
        appChangeUnderway = true;
        return performAppChanges();
    }
    async function performAppChanges() {
        // ...
        finishUpAndReturn(); // 完成后批量处理在队列中的任务
    }
    function finishUpAndReturn(){
        appChangeUnderway = false;
        if(peopleWaitingOnAppChange.length > 0){
            const nextPendingPromises = peopleWaitingOnAppChange;
            peopleWaitingOnAppChange = [];
            reroute(nextPendingPromises)
        }
    }
}
```

> 这里的思路和 `Vue.nextTick` 一样，如果当前应用正在加载时，并且用户频繁切换路由。我们会将此时的reroute方法暂存起来，等待当前应用加载完毕后再次触发reroute渲染应用，从而节约性能!

最终别忘了，完成一轮应用加载时，需要手动触发用户注册的路由事件！

```js
callAllEventListeners();
function callAllEventListeners() {
    pendingPromises.forEach((pendingPromise) => {
        callCapturedEventListeners(pendingPromise.eventArguments);
    });
    callCapturedEventListeners(eventArguments);
}
```

← 珠峰前端架构正式课