# 从零实现single-spa框架

# 一.single-spa的基本使用

> 在html中引入single-spa

```
<script src="https://cdn.bootcdn.net/ajax/libs/single-spa/5.9.3/umd/single-spa.min.js"></script>
```

## 1. 创建AB应用

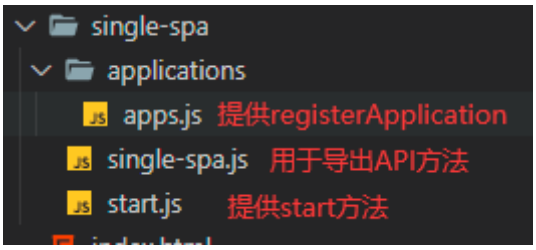
```
let { registerApplication, start } = singleSpa;
const customProps = { name: 'zf' };
let app1 = {
    bootstrap: [
        async () => { console.log('A应用启动1') },
        async () => { console.log('A应用启动2') }
    ],
    mount: async (props) => {
        console.log('A应用挂载', props)
    },
    unmount: async () => {
        console.log('A应用卸载')
    }
}
let app2 = {
    bootstrap: [
        async () => { console.log('B应用启动1') },
    ],
    mount: async (props) => {
        console.log('B应用挂载', props)
    },
    unmount: async () => {
        console.log('B应用卸载')
    }
}
```

> 接入协议,子应用必须要提供`bootstrap`、`mount`、`unmount`方法

## 2.注册应用

```
registerApplication(
    'app1',
    async () => app1,
    location => location.hash.startsWith('#/a'),
    customProps
);
registerApplication(
    'app2',
    async () => app2,
    location => location.hash.startsWith('#/b'),
    customProps
)
start();
```

# 二.实现single-spa



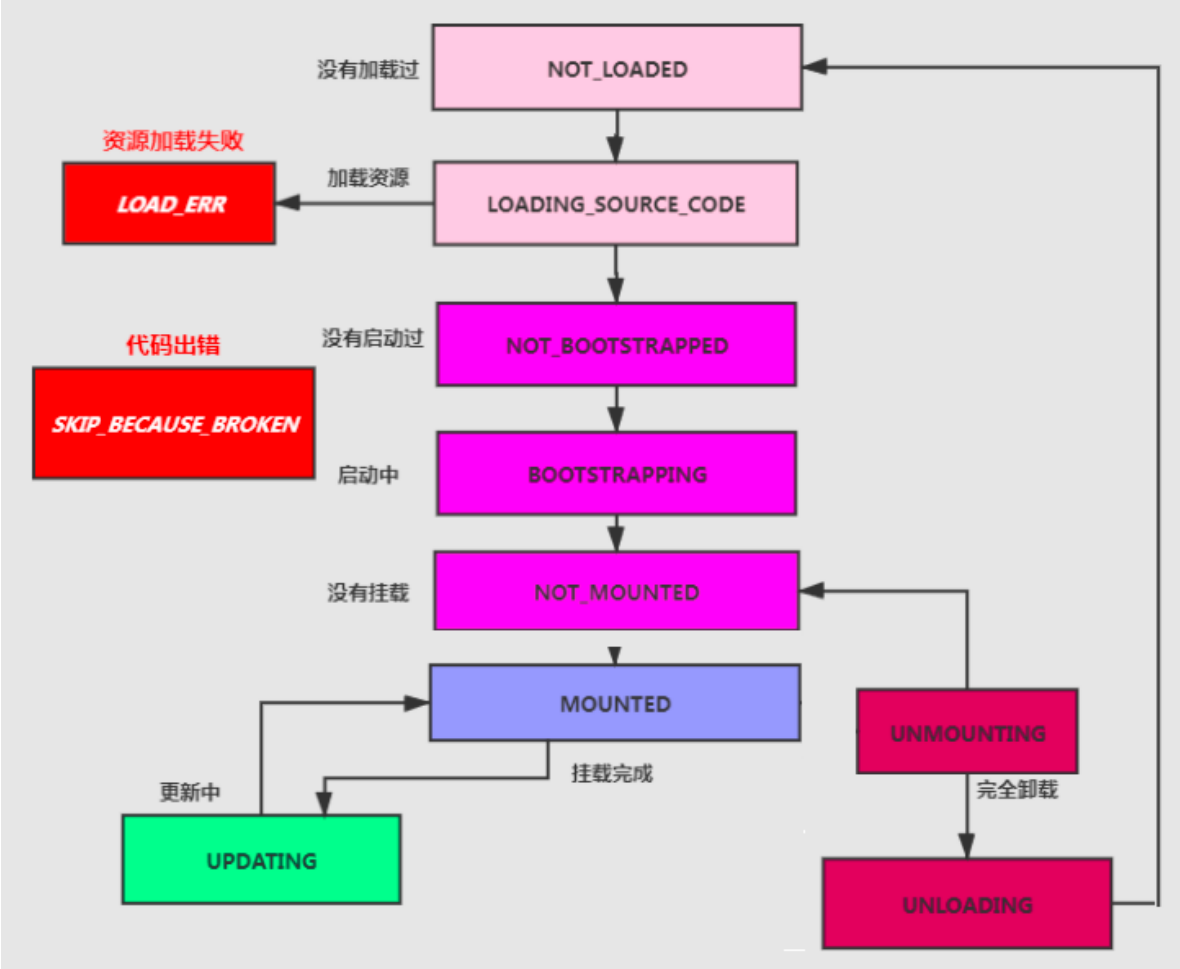

> 通过ES6Module引入`single-spa`

```
<script type="module">
    import { registerApplication, start } from './single-spa/single-spa.js'
    const customProps = { name: 'zf' };
    let app1 = {/*,,,*/}
    let app2 = {/*,,,*/}
    registerApplication(
        'app1',//...
    );
    registerApplication(
        'app2',//...
    )
    start();
</script>
```

## 1.实现registerApplication

```
const apps = [];
export function registerApplication(appName, loadApp, activeWhen,
customProps) {
    const registeration = {
        name:appName, // app的名字
        loadApp, // 要加载的app
        activeWhen, // 何时加载
        customProps // 自定义属性
    }
    apps.push(registeration);
    reroute(); // 重写路由，single-spa的核心逻辑。稍后实现~~~
}
```

## 2.应用加载状态




**applications/app.helper.js**

```
export const NOT_LOADED = 'NOT_LOADED'; // 应用没有加载
export const LOADING_SOURCE_CODE = 'LOADING_SOURCE_CODE'; // 加载资源代
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 MOUNTED = "MOUNTED"; // 挂载完毕
export const UPDATING = "UPDATING"; // 更新中
```

```
export const UNMOUNTING = "UNMOUNTING"; // 回到未挂载状态
export const UNLOADING = "UNLOADING"; // 完全卸载
export const LOAD_ERROR = "LOAD_ERROR";// 资源加载失败
export const SKIP_BECAUSE_BROKEN = "SKIP_BECAUSE_BROKEN"; // 出错

// 是不是挂载完毕
export function isActive(app){
    return app.status = MOUNTED
}
// 路由是否匹配，匹配到才激活
export function shouldBeActive(app){
    return app.activeWhen(window.location);
}
```

> 标记应用默认是未加载状态

```
const registeration = {
    name:appName,
    loadApp,
    activeWhen,
    customProps,
    status:NOT_LOADED
}
```

## 3.reroute实现

**navigation/reroute.js** 此方法是single-spa的核心方法，加载、启动、路由更新都会执行此方法

```
export function getAppChanges(){
    const appsToLoad = []; // 需要加载的应用
    const appsToMount = []; // 需要挂载的应用
    const appsToUnmount = []; // 需要去卸载的应用

    apps.forEach(app => {
        const appShouldBeActive = shouldBeActive(app);
        switch (app.status) {
            case NOT_LOADED:
            case LOADING_SOURCE_CODE: // 还没加载需要加载的
                if(appShouldBeActive){
                    appsToLoad.push(app);
                }
                break;
            case NOT_BOOTSTRAPPED:
            case NOT_MOUNTED: // 还没挂载
                if(appShouldBeActive){
                    appsToMount.push(app)
                }
```

```
                break;
            case MOUNTED: // 已经挂载了，但是路径不匹配
                if(!appShouldBeActive){
                    appsToUnmount.push(app);
                }
            default:
                break;
        }
    });
    return {appsToLoad,appsToMount,appsToUnmount}
}
```

> 根据app状态对所有注册的app进行分类

```
export function reroute() {
    // 所有的核心逻辑都在这里
    const { appsToLoad, appsToMount, appsToUnmount } =
getAppChanges();
    return loadApps();

    function loadApps() {
        // 获取所有需要加载的app,调用加载逻辑
        const loadPromises = appsToLoad.map(toLoadPromise); // 调用加
载逻辑
        return Promise.all(loadPromises)
    }
}
```

## 1).load.js

```
function flattenFnArray(fns) {
    fns = Array.isArray(fns) ? fns : [fns];
    return function(props) {
        return fns.reduce((resultPromise,fn)=>
resultPromise.then(()=>fn(props)),Promise.resolve())
    }
}
export function toLoadPromise(app) {
    if (app.status !== NOT_LOADED) { // 状态必须是NOT_LOADED才加载
        return app;
    }
    app.status = LOADING_SOURCE_CODE;
    return app.loadApp(app.customProps).then(val => {
        let { bootstrap, mount, unmount } = val; // 获取接口协议
        app.status = NOT_BOOTSTRAPPED;
        app.bootstrap = flattenFnArray(bootstrap);
        app.mount = flattenFnArray(mount);
        app.unmount = flattenFnArray(unmount);
        return app; // 返回应用
```

```
    })
}
```

## 4.实现start方法

```
import { reroute } from "./navigation/reroute";
export let started = false
export function start(){
    started = true;
    reroute();
}
```

```
export function reroute() {
    // 所有的核心逻辑都在这里
    const { appsToLoad, appsToMount, appsToUnmount } =
getAppChanges();
    if (started) { // 启动应用
        return performAppChanges();
    }
    function performAppChanges() {
        appsToUnmount.map(toUnmoutPromise); // 将不需要的组件全部卸载
        // 将需要加载的组件去加载-> 启动 -> 挂载
        appsToLoad.map(app => toLoadPromise(app).then((app) =>
tryToBootstrapAndMount(app)))
        // 如果已经加载完毕那么，直接启动和挂载
        appsToMount.map(appToMount =>
tryToBootstrapAndMount(appToMount))
    }
}
```

> 核心就是卸载需要卸载的应用-> 加载应用 -> 启动应用 -> 挂载应用

### 1).unmount.js

```
function toUnmoutPromise(app) {
    return Promise.resolve().then(() => {
        if (app.status !== MOUNTED) { // 如果不是挂载直接跳出
            return app;
        }
        app.status = UNMOUNTING;
        return app.unmount(app.customProps). // 调用卸载钩子
        then(() => {
            app.status = NOT_MOUNTED;
        });
    })
}
```

### 2).load.js

```
export function toLoadPromise(app) {
    return Promise.resolve().then(()=>{
        if (app.loadPromise) return app.loadPromise; // 如果正在加载直接返回
        if (app.status !== NOT_LOADED) { // 状态必须是NOT_LOADED才加载
            return app;
        }
        app.status = LOADING_SOURCE_CODE;
        return (app.loadPromise = Promise.resolve().then(() => {
            return app.loadApp(app.customProps).then(val => {
                let { bootstrap, mount, unmount } = val;
                app.status = NOT_BOOTSTRAPPED;
                app.bootstrap = flattenFnArray(bootstrap);
                app.mount = flattenFnArray(mount);
                app.unmount = flattenFnArray(unmount);
                delete app.loadPromise;
                return app;
            })
        }))
    })
}
```

```
function tryToBootstrapAndMount(app, unmountAllPromise) { // 尝试启动和挂载
    if (shouldBeActive(app)) { // 路径匹配去启动加载，保证卸载完毕在挂载最新的
        return toBootstrapPromise(app).then(app =>
            unmountAllPromise.then(() => toMountPromise(app))
        )
    }
}
```

### 3).bootstrap.js

```
function toBootstrapPromise(app){
    return Promise.resolve().then(() => {
        if(app.status !== NOT_BOOTSTRAPPED){ // 不是未启动直接返回
            return app;
        }
        app.status = BOOTSTRAPPING; // 启动中
        return app.bootstrap(app.customProps).then(()=>{
            app.status = NOT_MOUNTED; // 启动完毕后标记没有挂载
            return app;
        })
    })
}
```

### 4).mount.js

```
function toMountPromise(app){
    return Promise.resolve().then(() => {
        if(app.status !== NOT_MOUNTED){ // 不是未挂载状态 直接返回
            return app;
        }
        return app.mount(app.customProps).then(()=>{
            app.status = MOUNTED;
            return app
        })
    })
}
```

## 5.路由重写实现

```
import { reroute } from "./reroute.js";
export const routingEventsListeningTo = ['hashchange','popstate'];
function urlReroute(){
    reroute(arguments);
}
window.addEventListener('hashchange',urlReroute);
window.addEventListener('popstate',urlReroute);
```

> 监听hashchange和popstate，路径变化时重新初始化应用

### 1).拦截事件

```
const capturedEventListeners = { // 捕获的事件
    hashchange: [],
    popstate: [],
};
const originalAddEventListener = window.addEventListener; // 保留原来的
方法
const originalRemoveEventListener = window.removeEventListener;
// 如果是hashchange、popstate
window.addEventListener = function(eventName, fn) {
    if (routingEventsListeningTo.includes(eventName) &&
!capturedEventListeners[eventName].some(listener => listener == fn))
{
        return capturedEventListeners[eventName].push(fn);
    }
    return originalAddEventListener.apply(this, arguments);
}
window.removeEventListener = function(eventName, listenerFn) {
    if (routingEventsListeningTo.includes(eventName)) {
        capturedEventListeners[eventName] =
capturedEventListeners[eventName].filter((fn) => fn !== listenerFn);
```

```
            return;
        }
        return originalRemoveEventListener.apply(this, arguments);
    };
```

### 2).跳转方法拦截

```
function patchedUpdateState(updateState, methodName) {
    return function() {
        // 例如 vue-router内部会通过pushState() 不改路径改状态，所以还是要处理下
        const urlBefore = window.location.href;
        const result = updateState.apply(this, arguments);
        const urlAfter = window.location.href;
        if (urlBefore !== urlAfter) {
           window.dispatchEvent(new PopStateEvent("popstate"));// 路径不一样，继续重启应用
        }
        return result;
    }
}
window.history.pushState =
patchedUpdateState(window.history.pushState, 'pushState');
window.history.replaceState =
patchedUpdateState(window.history.replaceState, 'replaceState')
```

### 3) 触发事件

```
export function reroute(eventArguments) {
    function loadApps() {// 启动完成调用事件
        return Promise.all(loadPromises).then(callAllEventListeners)
    }
    function performAppChanges() {
        let unmountAllPromise =
Promise.all(appsToUnmount.map(toUnmoutPromise));
        return unmountAllPromise.then(()=>{ // 组件卸载完毕调用事件
            callAllEventListeners();
        })
    }
    function callAllEventListeners() {
        callCapturedEventListeners(eventArguments);// 调用捕获到的事件
    }
}
```

```
export function callCapturedEventListeners(eventArguments) {
    // 触发捕获的事件
    if (eventArguments) {
        const eventType = eventArguments[0].type;
        // 触发缓存中的方法
        if (routingEventsListeningTo.includes(eventType)) {
            capturedEventListeners[eventType].forEach(listener => {
                listener.apply(this, eventArguments);
            })
        }
    }
}
```

```
let appChangeUnderway = false; // 用于标识是否正在调用performAppChanges
let peopleWaitingOnAppChange = []; // 存放用户得逻辑
export function reroute(eventArguments, pendingPromises = []) {
    if (appChangeUnderway) { // 正在改就存起来
        return new Promise((resolve, reject) => {
            peopleWaitingOnAppChange.push({
                resolve,
                reject,
                eventArguments
            })
        })
    }
    if (started) { // 启动应用
        appChangeUnderway = true; // 标记正在调用
        return performAppChanges();
    }
    return loadApps();

    function performAppChanges() {
        return unmountAllPromise.then((arr) => {
            callAllEventListeners();
            // 挂载完毕后触发路由逻辑
            return
Promise.all(loadMountPromises.concat(mountPromise)).then(() => {
                appChangeUnderway = false;
                if (peopleWaitingOnAppChange.length > 0) {
                    const nextPendingPromises =
peopleWaitingOnAppChange;
                    peopleWaitingOnAppChange = [];
                    reroute(null,nextPendingPromises); // 再次发生跳转
                }
            })
        })
    }
    function callAllEventListeners() { // 调用所有事件
```

```
        pendingPromises.forEach((pendingPromise) =>
callCapturedEventListeners(pendingPromise.eventArguments));
        callCapturedEventListeners(eventArguments);
    }
}
```