一.single-spa原理剖析

1.基本使用

```
<script
src="https://cdn.bootcdn.net/ajax/libs/single-
spa/5.9.3/umd/single-spa.min.js"></script>
<script>
  let { registerApplication, start } = singleSpa
  const customProps = { data: 'message' };
  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')
    }
  }
</script>
```

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

```
// 注册应用
registerApplication(
  'app1',
  async () => { console.log('app1加载了'); return
app1 },
  location => location.hash.startsWith('#/a'),
  customProps
);
registerApplication(
  'app2',
  async () => { console.log('app2加载了'); return
app2 },
  location => location.hash.startsWith('#/b'),
  customProps
)
start(); // 挂载应用
```

2.实现SingleSpa加载

通过ES6Module引入 single-spa

```html
<script type="module">
  import { registerApplication, start } from
'./single-spa/single-spa.js'
  // ...
</script>
```

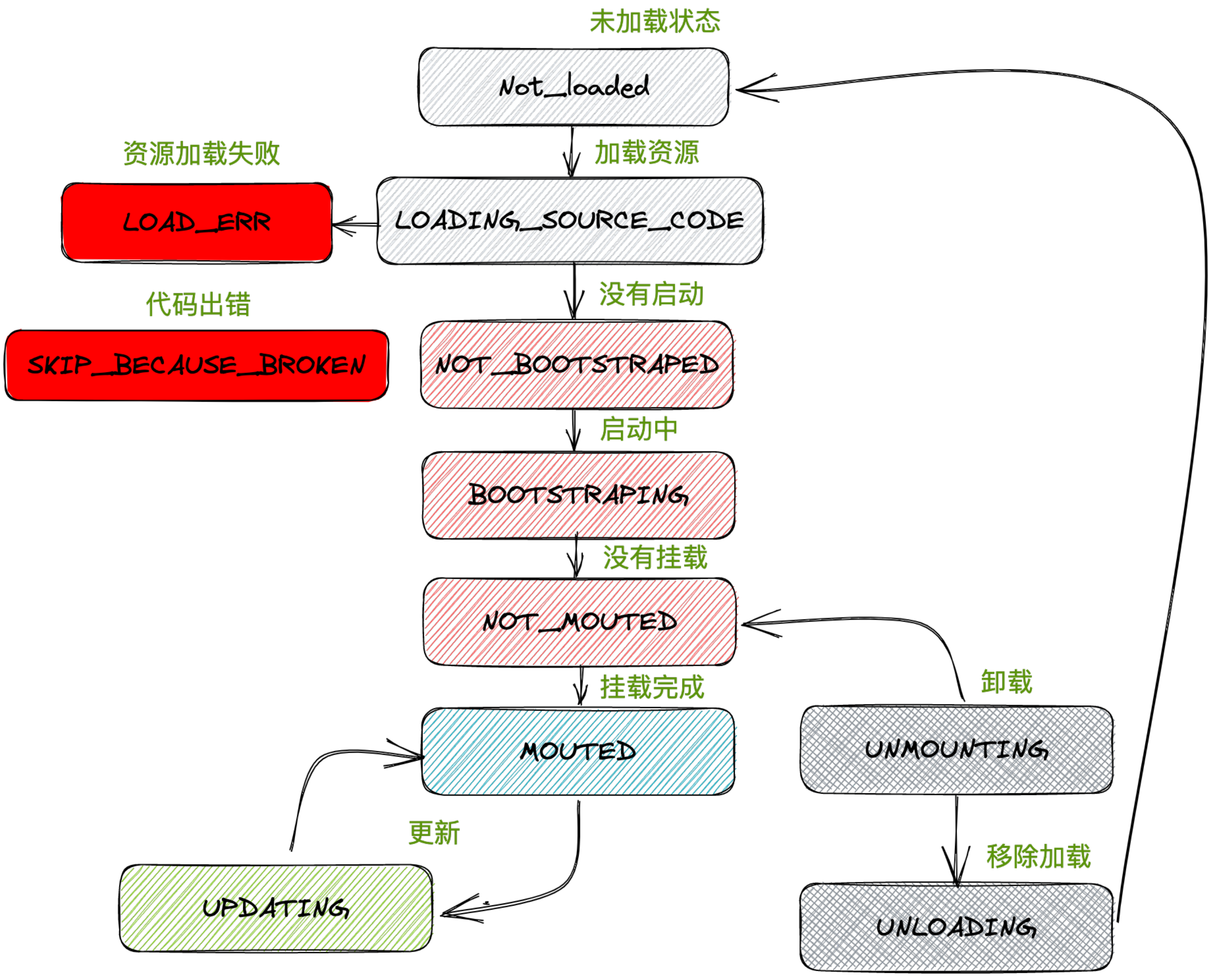
```js
export { registerApplication } from
'./applications/apps.js'
export { start } from './start.js'
```

1).实现registerApplication

single-spa/applications/apps.js

```
export 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);
}
```

2).应用加载状态



single-spa/applications/app.helpers.js

```
// App statuses
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 MOUNTING = "MOUNTING";  // 挂载中
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

// 当前应用是否被激活
export function isActive(app) {
    return app.status = MOUNTED
}

// 当前应用是否应该被激活
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, // app的名字
    loadApp, // 要加载的app
    activeWhen, // 何时加载
    customProps, // 自定义属性
+   status: NOT_LOADED
}
apps.push(registeration);
+ reroute(); // 根据路径进行加载路由
```

3).reroute实现

single-spa/navigation/reroute.js

```
export function reroute(){
    // 获取app当前所有状态
    const { appsToLoad, appsToMount,
appsToUnmount } = getAppChanges();
}
```

single-spa/applications/app.helpers.js

```
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，要挂载
                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(){
    // 获取app当前所有状态
    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
        return
Promise.all(appsToLoad.map(toLoadPromise)); //
加载应用
    }
}
```

4).load.js

single-spa/lifecycles/load.js

```
import {  LOADING_SOURCE_CODE, NOT_BOOTSTRAPPED,
NOT_LOADED } from
"../applications/app.helpers.js";

function flattenFnArray(fns) {
    fns = Array.isArray(fns) ? fns : [fns]; //
包装成数组，进行组合
    return function (props) {
        return fns.reduce((resultPromise, fn) =>
resultPromise.then(() => fn(props)),
Promise.resolve())
    }
}
export function toLoadPromise(app) {
    return Promise.resolve().then(() => {
        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
        })
    })
}
```

5).实现start方法

single-spa/start.js

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

single-spa/navigation/reroute.js

```
export function reroute() {
    // 获取app当前所有状态
    const { appsToLoad, appsToMount,
appsToUnmount } = getAppChanges();
    if (started) { // 挂载应用及后续切换路由的逻辑
        return performAppChanges();
    }
    function performAppChanges() {
        // 1.将需要卸载的应用进行卸载
        let unmountAllPromise =
Promise.all(appsToUnmount.map(toUnmoutPromise));
        // 2.加载应用（可能已经加载过了，需要防止重新加
载）-> 卸载之前的 -> 进行挂载
        const loadMountPromises =
appsToLoad.map(app =>
toLoadPromise(app).then((app) =>
            tryToBootstrapAndMount(app,
unmountAllPromise)))
        // 3.如果已经加载完毕那么，直接启动和挂载
        const mountPromise =
appsToMount.map(appToMount =>
            tryToBootstrapAndMount(appToMount,
unmountAllPromise))
    }
    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))
            )
        }
    }
}
```

> 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

6).unmount.js

```
import { MOUNTED, NOT_MOUNTED, UNMOUNTING } from
"../applications/app.helpers.js";

export 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;
            });
    })
}
```

7).bootstrap.js

```
import { BOOTSTRAPPING, NOT_BOOTSTRAPPED,
NOT_MOUNTED } from
"../applications/app.helpers.js";

export 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;
        })
    })
}
```

8).mount.js

```
import { MOUNTED, NOT_MOUNTED } from
"../applications/app.helpers.js";

export 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
        })
    })
}
```

3.路由重写

single-spa/navigation/navigation-event.js

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

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;

window.addEventListener = function (eventName,
fn) {
    if
(routingEventsListeningTo.includes(eventName) &&
 !capturedEventListeners[eventName].some(listene
r => 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"));// 触发popstate事件
        }
        return result;
    }
}
window.history.pushState =
patchedUpdateState(window.history.pushState,
'pushState');
window.history.replaceState =
patchedUpdateState(window.history.replaceState,
'replaceState')
```

3).触发事件

```
function loadApps() {
  // 获取所有需要加载的app,调用加载逻辑
+ return
Promise.all(appsToLoad.map(toLoadPromise)).then(
callAllEventListeners); // 加载应用
}

function performAppChanges() {
  // 1.将需要卸载的应用进行卸载
  let unmountAllPromise =
Promise.all(appsToUnmount.map(toUnmoutPromise));
  // 2.加载应用（可能已经加载过了，需要防止重新加载）->
卸载之前的 -> 进行挂载
  const loadMountPromises = appsToLoad.map(app
=> toLoadPromise(app).then((app) =>



tryToBootstrapAndMount(app, unmountAllPromise)))
  // 3.如果已经加载完毕那么，直接启动和挂载
  const mountPromise =
appsToMount.map(appToMount =>

tryToBootstrapAndMount(appToMount,
unmountAllPromise))
+ 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
+}
```

4).路由频繁触发

```
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 = []) {

    // 获取app当前所有状态
    const { appsToLoad, appsToMount,
appsToUnmount } = getAppChanges();

+   if (appChangeUnderway) { // 正在改就存起来
        return new Promise((resolve, reject) =>
{
            peopleWaitingOnAppChange.push({
                    resolve,
                    reject,
                    eventArguments
                })
            })
        }


        if (started) {
+           appChangeUnderway = true;
            return performAppChanges();
        }


        function performAppChanges() {
            // 1.将需要卸载的应用进行卸载
            let unmountAllPromise =
Promise.all(appsToUnmount.map(toUnmoutPromise));
            // 2.加载应用（可能已经加载过了，需要防止重新加
载） -> 卸载之前的 -> 进行挂载
            const loadMountPromises =
appsToLoad.map(app =>
toLoadPromise(app).then((app) =>
                tryToBootstrapAndMount(app,
unmountAllPromise)))
            // 3.如果已经加载完毕那么，直接启动和挂载
        const mountPromise =
appsToMount.map(appToMount =>
            tryToBootstrapAndMount(appToMount,
unmountAllPromise))

        return unmountAllPromise.then(() => { //
组件卸载完毕调用事件
            callAllEventListeners();

+           return
Promise.all(loadMountPromises.concat(mountPromis
e)).then(() => {
                appChangeUnderway = false;
                if
(peopleWaitingOnAppChange.length > 0) {
                    const nextPendingPromises =
peopleWaitingOnAppChange;
                    peopleWaitingOnAppChange =
[];
                    reroute(null,
nextPendingPromises); // 再次发生跳转
                }
            })
        })
    }

+   function callAllEventListeners() {
        pendingPromises.forEach((pendingPromise)
=>
callCapturedEventListeners(pendingPromise.eventA
rguments));

 callCapturedEventListeners(eventArguments);//
调用捕获到的事件
    }
}
```