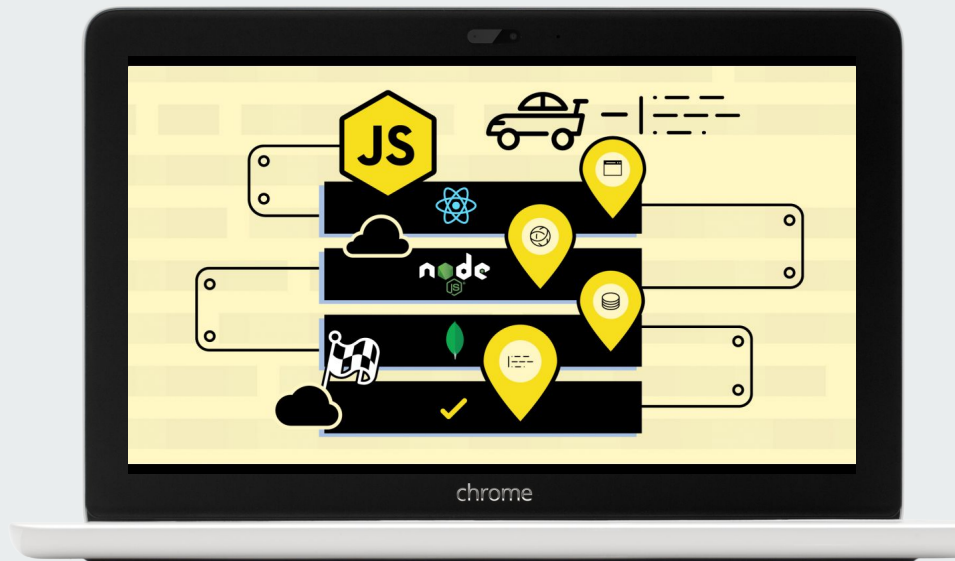


JavaScript 沙箱 机制演进历史

[hijiangtao](#)



目录

1. 微前端与 JavaScript 沙箱介绍
2. Proxy + Reflect API
3. 基于 Proxy 的沙箱实现及降级方案
4. 基于 iframe 的沙箱实现
5. 各类沙箱机制效果对比
6. 基于 ES 提案 ShadowRealm API 介绍
7. 总结

微前端与 JavaScript 沙箱



什么是微前端

微前端是将Web应用由单一的单体应用转变为多个小型前端应用聚合为一的一种手段。

1. 主应用: 在微前端方案中, 区分主子应用, 主应用通常负责全局资源的加载、隔离、控制运行, 用户登陆信息等全局状态的管理等等, 也被称为基座、微前端全局环境等;
2. 子应用: 微前端方案中可以独立加载运行的一个Web应用, 通常需要一个完备的隔离环境供其加载, 文中提到的沙箱激活/卸载也是为其服务, 也称微应用;

single-spa 没有开箱即用的 JavaScript 隔离

1. 基于 Proxy 快照存储 + window 修改的实现
 2. 基于 Proxy 代理拦截 + window 激活/卸载的实现
 3. 基于普通对象快照存储的 window 属性 diff 实现
 4. 基于 iframe + 消息通信的实现
 5. 基于 ShadowRealm 提案的实现
 6. 基于 with + eval 的简单实现
 7.
-



什么是 JavaScript 沙箱

1. 沙箱, 即 sandbox, 意指一个允许你独立运行程序的虚拟环境, 沙箱可以隔离当前执行的环境作用域和外部的其他作用域, 外界无法修改该环境内任何信息, 沙箱内的东西单独运行, 环境间相互不受影响。
2. 当一个沙箱支持解析或着执行不可信的 JavaScript 时, 这样的环境可被称为 JavaScript 沙箱。



如何实现一个 JavaScript 沙箱

1. 构建闭包环境
2. 模拟原生浏览器对象

Proxy API 介绍



Proxy

Proxy 是一个标准 Web API, 在 ES6 版本中被推出, 这个对象可以用于创建一个对象的代理, 从而实现基本操作的拦截和自定义(如属性查找、赋值、枚举、函数调用等)。

```
1 let proxy = new Proxy(target, handler)
```

要包装的对象

捕捉器/拦截器

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy



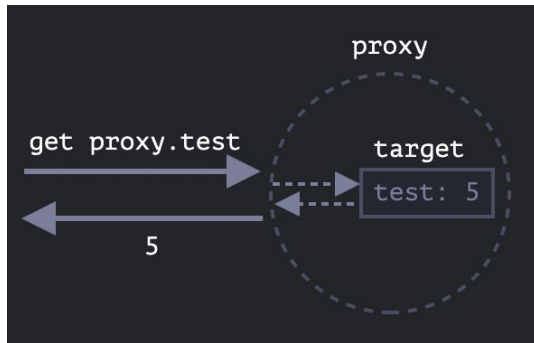
Proxy 捕捉器方法列表

内部方法	Handler 方法	何时触发
[[Get]]	get	读取属性
[[Set]]	set	写入属性
[[HasProperty]]	has	in 操作符
[[Delete]]	deleteProperty	delete 操作符
[[Call]]	apply	函数调用
[[Construct]]	construct	new 操作符
[[GetPrototypeOf]]	getPrototypeOf	Object.getPrototypeOf
[[SetPrototypeOf]]	setPrototypeOf	Object.setPrototypeOf
[[IsExtensible]]	isExtensible	Object.isExtensible
[[PreventExtensions]]	preventExtensions	Object.preventExtensions
[[DefineOwnProperty]]	defineProperty	Object.defineProperty, Object.defineProperties
[[GetOwnProperty]]	getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor, for..in, Object.keys/values/entries
[[OwnPropertyKeys]]	ownKeys	Object.getOwnPropertyNames, Object.getOwnPropertySymbols, for..in, Object.keys/values/entries

<https://xtc39.es/ecma262/#sec-proxy-object-internal-methods-and-internal-slots>

Proxy - 空拦截器

```
1 let target = {};  
2 let proxy = new Proxy(target, {}); // 空的 handler 对象  
3  
4 proxy.test = 5; // 写入 proxy 对象 (1)  
5 alert(target.test); // 5, test 属性出现在了 target 中!  
6  
7 alert(proxy.test); // 5, 我们也可以从 proxy 对象读取它 (2)  
8  
9 for(let key in proxy) alert(key); // test, 迭代也正常工作 (3)
```





Proxy - get

```
const handler = {
  get: function(obj, prop) {
    return prop in obj ? obj[prop] : 37;
  }
};

const p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;

console.log(p.a, p.b);      // 1, undefined
console.log('c' in p, p.c); // false, 37
```



Proxy - has

```
1 let range = {
2   start: 1,
3   end: 10
4 };
5
6 range = new Proxy(range, {
7   has(target, prop) {
8     return prop >= target.start && prop <= target.end;
9   }
10 });
11
12 alert(5 in range); // true
13 alert(50 in range); // false
```



Reflect

Reflect 是一个内置的对象, 它提供拦截 JavaScript 操作的方法。

```
1 let user = {};  
2  
3 Reflect.set(user, 'name', 'John');  
4  
5 alert(user.name); // John
```

调用 [[Set]]

分别为对象、属性名、取值



Reflect 与 Proxy 的关系

对于每个可被 Proxy 捕获的内部方法，在 Reflect 中都有一个对应的方法，其名称和参数与 Proxy 捕捉器相同。

```
1 let user = {
2   name: "John",
3 };
4
5 user = new Proxy(user, {
6   get(target, prop, receiver) {
7     alert(`GET ${prop}`);
8     return Reflect.get(target, prop, receiver); // (1)
9   },
10  set(target, prop, val, receiver) {
11    alert(`SET ${prop}=${val}`);
12    return Reflect.set(target, prop, val, receiver); // (2)
13  }
14 });
15
16 let name = user.name; // 显示 "GET name"
17 user.name = "Pete"; // 显示 "SET name=Pete"
```

```
1 let user = {
2   _name: "Guest",
3   get name() {
4     return this._name;
5   }
6 };
7
8 let userProxy = new Proxy(user, {
9   get(target, prop, receiver) {
10    return target[prop]; // (*) target = user
11  }
12 });
13
14 let admin = {
15   __proto__: userProxy,
16   _name: "Admin"
17 };
18
19 // 期望输出: Admin
20 alert(admin.name); // 输出: Guest (?!?)
```

```
1 let user = {
2   _name: "Guest",
3   get name() {
4     return this._name;
5   }
6 };
7
8 let userProxy = new Proxy(user, {
9   get(target, prop, receiver) { // receiver = admin
10    return Reflect.get(target, prop, receiver); // (*)
11  }
12 });
13
14
15 let admin = {
16   __proto__: userProxy,
17   _name: "Admin"
18 };
19
20 alert(admin.name); // Admin
```

基于 Proxy 实现的沙箱机制



思考

拦截内容应包含

1. window 属性读写操作
2. 具有内部插槽的内建对象, 如 Map
3. 其他对象, 如 Array
4. 取值逃逸

```
const object1 = {  
  property1: 42  
};  
  
object1[Symbol.unscopables] = {  
  property1: true  
};  
  
with (object1) {  
  console.log(property1);  
  // expected output: Error: property1 is not defined  
}
```

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol/unscopables



基于 Proxy 的两类沙箱

微前端框架 qiankun 中共存在三类沙箱，基于 Proxy 实现方式不同及是否支持多实例，可以分为两类：

1. 支持子应用单实例沙箱
(LegacySandbox)
2. 支持子应用多实例沙箱
(ProxySandbox)



LegacySandbox

本处以 qiankun 为例

```
/** 沙箱期间新增的全局变量 */  
private addedPropsMapInSandbox = new Map<PropertyKey, any>();  
  
/** 沙箱期间更新的全局变量 */  
private modifiedPropsOriginalValueMapInSandbox = new Map<PropertyKey, any>()  
  
/** 持续记录更新的(新增和修改的)全局变量的 map, 用于在任意时刻做 snapshot */  
private currentUpdatedPropsValueMap = new Map<PropertyKey, any>();
```



LegacySandbox

```
active() {  
  if (!this.sandboxRunning) {  
    this.currentUpdatedPropsValueMap.forEach(  
      (v, p) => this.setWindowProp(p, v)  
    );  
  }  
  
  this.sandboxRunning = true;  
}
```

```
inactive() {  
  this.modifiedPropsOriginalValueMapInSandbox.forEach(  
    (v, p) => this.setWindowProp(p, v)  
  );  
  
  this.addedPropsMapInSandbox.forEach(  
    (_, p) => this.setWindowProp(p, undefined, true)  
  );  
  
  this.sandboxRunning = false;  
}
```



ProxySandbox

LegacySandbox 的思路在于虽然建立了沙箱代理, 但在子应用运行过程中, 所有的赋值仍旧会直接操作 window 对象, 代理所做的事情就是记录变化(形成快照); 而针对激活和卸载, 沙箱会在激活时还原子应用的状态, 而卸载时还原主应用的状态, 以此达到沙箱隔离的目的。

ProxySandbox 的方案是同时用 Proxy 给子应用运行环境做了 get 与 set 拦截。



ProxySandbox

由于状态池与子应用绑定，那么运行多个子应用，便可以产生多个相互独立的沙箱环境。

由于取值赋值均在建立的状态池上操作，因此，在第一种沙箱环境下激活和卸载需要做的工作，这里也就不需要了。

基于属性 diff 实现的沙箱机制


```
// iter 为一个遍历对象属性的方法

active() {
  // 记录当前快照
  this.windowSnapshot = {} as Window;
  iter(window, (prop) => {
    this.windowSnapshot[prop] = window[prop];
  });

  // 恢复之前的变更
  Object.keys(this.modifyPropsMap).forEach((p: any) => {
    window[p] = this.modifyPropsMap[p];
  });

  this.sandboxRunning = true;
}
```

```
inactive() {
  this.modifyPropsMap = {};

  iter(window, (prop) => {
    if (window[prop] !== this.windowSnapshot[prop]) {
      // 记录变更, 恢复环境
      this.modifyPropsMap[prop] = window[prop];
      window[prop] = this.windowSnapshot[prop];
    }
  });

  this.sandboxRunning = false;
}
```

<https://github.com/umijs/qiankun/blob/dbbc9acdb0733b3ab28e0470c969d65b57653ff0/src/sandbox/snapshotSandbox.ts>



qiankun 类框架接入的限制

构建流水与注意事项

- 产物部署在 CDN 上且须保证无 CORS 限制
- 由于微前端方案中不支持 JavaScript modules 加载, 若使用 Vite 作为构建工具, 需要注意代码转译
- Vue3 项目的路由依赖 single-spa 打开 [urlRerouteOnly](#) 开关, 不同版本的微前端框架对该 API 的处理不一致, 使用需谨慎
- 本地开发调试, 需借助代理工具, 以及状态模拟(全局变量等)

基于 iframe 实现的沙箱机制



Proxy 类沙箱存在的问题

1. 子应用新建了一个 iframe 来做些 JavaScript 逻辑, 但在里面通过 `window.parent.xxx` 无法获取子应用 `window` 上的全局变量?
2. 我本地运行是有值的, 为什么到微前端里就 `undefined` 了呢?



基于 iframe 实现的沙箱机制

常规思路下，大家想到的iframe 都是在页面内起一个iframe 元素，然后将需要加载的 url 填入进行加载

1. 使用简单，一个 url 即可；
2. 利用浏览器的设计，可以实现样式、DOM、JavaScript 代码执行的完美隔离；
3. 页面原则上可以起无数多个iframe 标签来加载应用；
4. 通过 iframe 实现的沙箱可以绕过 eval 执行的限制；

iframe 沙箱的几点考虑

1. 应用间运行时隔离;
2. 应用间通信;
3. 路由劫持;



<iframe>



iframe 沙箱 - 运行环境隔离

```
const scriptInstance = document.createElement('script');
const script = `(function(window, self, document, location, history) {
  ${scriptString}\n
  }).bind(window.proxyWindow)(
  window.proxyWindow,
  window.proxyWindow,
  window.proxyShadowDom,
  window.proxyLocation,
  window.proxyHistory,
  );`;

scriptInstance.text = script;
document.head.appendChild(scriptInstance);
```



iframe 沙箱 - 通信

1. 自定义 event
2. props
3. BroadcastChannel
4. postMessage

```
const iframe = document.createElement('iframe',{url:'about:blank'});
document.body.appendChild(iframe);
const sandboxGlobal = iframe.contentWindow;
```




iframe 沙箱 - 路由同步

1. 让 JavaScript 沙箱内路由变更操作在主应用环境生效;
2. 同步沙箱内路由变化至主应用环境;

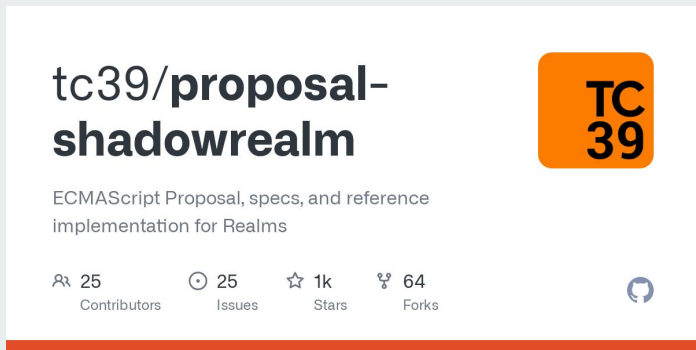
各类沙箱机制对比

	多实例运行	语法兼容	不污染全局环境（主应用）
LegacySanbox	✘	✘	✘
ProxySandbox	✔	✘	✔
SnapshotSandbox	✘	✔	✘
iframe	✔	✔	✔

基于 ES 提案 ShadowRealm API 介绍

ES 提案

ShadowRealm



tc39/proposal-shadowrealm

TC 39

ECMAScript Proposal, specs, and reference implementation for Realms

25 Contributors 25 Issues 1k Stars 64 Forks

ShadowRealm 是一个 ECMAScript 标准提案，旨在创建一个独立的全局环境，它的全局对象包含自己的内建函数与对象（未绑定到全局变量的标准对象，如 `Object.prototype` 的初始值），有自己独立的作用域。

方案当前处于 stage 3 阶段。

提案地址 <https://github.com/tc39/proposal-shadowrealm>



Realm - JavaScript 运行环境实例

```
<body>
  <iframe>
</iframe>
  <script>
    const win = frames[0].window;
    console.assert(win.globalThis !== globalThis); // (A)
    console.assert(win.Array !== Array); // (B)
  </script>
</body>
```

本文中提及的 JavaScript 运行环境实例为意译, 原文表述为「an instance of the JavaScript platform」
<https://2ality.com/2022/04/shadow-realms.html>

每个 ShadowRealm 实例都有自己独立的运行环境实例，在 realm 中，提案提供了两种方法让我们来执行运行环境实例中的 JavaScript 代码：

- `.evaluate()`: 同步执行代码字符串，类似 `eval()`。
- `.importValue()`: 返回一个 `Promise` 对象，异步执行代码字符串。

```
declare class ShadowRealm {
  constructor();
  evaluate(sourceText: string): PrimitiveValueOrCallable;
  importValue(specifier: string, bindingName: string): Promise<PrimitiveValueOrCallable>;
}
```



ShadowRealm API - evaluate

```
const sr = new ShadowRealm();  
console.assert(  
  sr.evaluate(`'ab' + 'cd'`) === 'abcd'  
);
```



ShadowRealm API - evaluate

```
globalThis.realm = 'incubator realm';

const sr = new ShadowRealm();
sr.evaluate(`globalThis.realm = 'child realm'`);

const wrappedFunc = sr.evaluate(() => globalThis.realm);
console.assert(wrappedFunc() === 'child realm');
```




ShadowRealm API - importValue

```
// main.js
const sr = new ShadowRealm();
const wrappedSum = await sr.importValue('./my-module.js', 'sum');
console.assert(wrappedSum('hi', ' ', 'folks', '!') === 'hi folks!');

// my-module.js
export function sum(...values) {
  return values.reduce((prev, value) => prev + value);
}
```



ShadowRealm API - 应用场景

- Web 应用诸如 IDE 或绘图等程序可以运行第三方代码, 允许其以插件或者配置的方式引入;
- 利用 ShadowRealms 建立一个可编程环境, 来运行用户的代码;
- 服务器可以在 ShadowRealms 中运行第三方代码;
- 在 ShadowRealms 中可以运行测试运行器 (Test Runner), 这样外部的 JS 执行环境不会受到影响, 并且每个套件都可以在新环境中启动 (这有助于提高可复用性), 这种场景类似于微前端的 JavaScript 沙箱;
- 网页抓取和网页应用测试等;

总结

如果按照沙箱机制在实现时所用到的主要 Web 技术不同, 当下已经论证、开源或者存在实现可能性的 JavaScript 沙箱机制可以分为以下几类:

1. 基于 ES6 API Proxy 实现
2. 基于属性 diff 实现
3. 基于 iframe 实现
4. 基于 ES 提案 ShadowRealm 实现

本文基于个人项目实践、阅读代码梳理等方式对每类沙箱机制均进行了介绍, 部分引用了 qiankun 的代码实现, 部分写了伪代码解释, 部分引用了最新 ECMAScript 提案示例, 但仍未能详尽每一处细节, 比如没有针对微前端框架深入介绍, 也不会就某一个沙箱机制的具体细节实现(比如如何构建闭包环境、属性读取的边界处理等)进行剖析, 但这些对于从更大的层面了解微前端机制都不可或缺。

如果你想了解关于 CSS 样式隔离的内容可以搜索 Shadow DOM 相关内容进一步查阅;

如果你想了解微前端的主子应用加载、运行机制, 可以参考 single-spa 文档、qiankun 文档、ShadowRealm 提案等内容;

如果你想了解文中涉及的一些概念与 API 用法可以在 MDN 进行搜索查阅, 大部分均有对应介绍。

Q&A





References

<https://hijiangtao.github.io/2022/06/11/JavaScript-Sandbox-Mechanism-and-Its-History/>