同源限制

浏览器安全的基石是“同源政策”（same-origin policy）。

1. 概述

1995 年，同源政策由 Netscape 公司引入浏览器。目前，所有浏览器都实行这个政策。

最初，它的含义是指，`A` 网页设置的 `Cookie`，`B` 网页不能打开，除非这两个网页“同源”。所谓“同源”指的是“三个相同”。

> - 协议相同
> - 域名相同
> - 端口相同

举例来说，`http://www.example.com/dir/page.html` 这个网址，协议是 `http://`，域名是 `www.example.com`，端口是 `80`（默认端口可以省略），它的同源情况如下。

- `http://www.example.com/dir2/other.html`：同源
- `http://example.com/dir/other.html`：不同源（域名不同）
- `http://v2.www.example.com/dir/other.html`：不同源（域名不同）
- `http://www.example.com:81/dir/other.html`：不同源（端口不同）
- `https://www.example.com/dir/page.html`：不同源（协议不同）

标准规定端口不同的网址不是同源（比如 `8000` 端口和 `8001` 端口不是同源），但是浏览器没有遵守这条规定。实际上，同一个网域的不同端口，是可以互相读取 Cookie 的。

2. 目的

同源政策的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。

设想这样一种场景：`A` 网站是一家银行，用户登录以后，`A` 网站在用户的机器上设置了一个 `Cookie`，包含了一些隐私信息。用户离开 `A` 网站以后，又去访问 `B` 网站，如果没有同源限制，`B` 网站可以读取 `A` 网站的 `Cookie`，那么隐私就泄漏了。更可怕的是，`Cookie` 往往用来保存用户的登录状态，如果用户没有退出登录，其他网站就可以冒充用户，为所欲为。 **浏览器规定，提交表单不受同源政策的限制。**

由此可见，同源政策是必需的，否则 `Cookie` 可以共享，互联网就毫无安全可言了。

2.1. 限制范围

随着互联网的发展，同源政策越来越严格。目前，如果非同源，共有三种行为受到限制。

- 无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB。
- 无法接触非同源网页的 DOM。
- 无法向非同源地址发送 AJAX 请求（可以发送，但浏览器会拒绝接受响应）。

另外，通过 JavaScript 脚本可以拿到其他窗口的 `window` 对象。如果是非同源的网页，目前允许一个窗口可以接触其他网页的 `window` 对象的九个属性和四个方法。

- window.closed
- window.frames
- window.length
- window.location
- window.opener
- window.parent
- window.self
- window.top
- window.window
- window.blur()
- window.close()
- window.focus()
- window.postMessage()

上面的九个属性之中，只有 window.location 是可读写的，其他八个全部都是只读。而且，即使是 location 对象，非同源的情况下，也只允许调用 location.replace() 方法和写入 location.href 属性。

虽然这些限制是必要的，但是有时很不方便，合理的用途也受到影响。

2.2. Cookie

Cookie 是服务器写入浏览器的一小段信息，只有同源的网页才能共享。如果两个网页一级域名相同，只是次级域名不同，浏览器允许通过设置 document.domain 共享 Cookie。

举例来说，A 网页的网址是 http://w1.example.com/a.html，B 网页的网址是 http://w2.example.com/b.html，那么只要设置相同的 document.domain，两个网页就可以共享 Cookie。因为浏览器通过 document.domain 属性来检查是否同源。

```
// 两个网页都需要设置
document.domain = 'example.com';
```

A 和 B 两个网页都需要设置 document.domain 属性，才能达到同源的目的。 因为设置 document.domain 的同时，会把端口重置为 null，因此如果只设置一个网页的 document.domain，会导致两个网址的端口不同，还是达不到同源的目的。

现在，A 网页通过脚本设置一个 Cookie。

```
document.cookie = "test1=hello";
```

B 网页就可以读到这个 Cookie。

```
let allCookie = document.cookie;
```

这种方法只适用于 Cookie 和 iframe 窗口，LocalStorage 和 IndexedDB 无法通过这种方法，规避同源政策，而要使用 PostMessage API。

另外，服务器也可以在设置 Cookie 的时候，指定 Cookie 的所属域名为一级域名，比如 .example.com。

```
Set-Cookie: key=value; domain=.example.com; path=/
```

指定 Cookie 的 domain 为一级域名，二级域名和三级域名不用做任何设置，都可以读取这个 Cookie。

2.3. iframe 和多窗口通信

iframe 元素可以在当前网页之中，嵌入其他网页。每个 iframe 元素形成自己的窗口，即有自己的 window 对象。iframe 窗口之中的脚本，可以获得父窗口和子窗口。但是，只有在同源的情况下，父窗口和子窗口才能通信；如果跨域，就无法拿到对方的 DOM。

比如，父窗口运行下面的命令，如果 iframe 窗口不是同源，就会报错。

```
document
.getElementById("myIFrame")
.contentWindow
.document
// Uncaught DOMException: Blocked a frame from accessing a cross-origin frame. 阻止一个 frame 访问另一个跨站点 frame
```

上面命令中，父窗口想获取子窗口的 DOM，因为跨域导致报错。

反之亦然，子窗口获取主窗口的 DOM 也会报错。

```
window.parent.document.body; // 报错
```

这种情况不仅适用于 iframe 窗口，还适用于 window.open 方法打开的窗口，只要跨域，父窗口与子窗口之间就无法通信。

如果两个窗口一级域名相同，只是二级域名不同，那么设置上一节介绍的 document.domain 属性，就可以规避同源政策，拿到 DOM。

对于完全不同源的网站，目前有两种方法，可以解决跨域窗口的通信问题。

- 片段识别符（fragment identifier）
- 跨文档通信 API（Cross-document messaging）

2.3.1 片段识别符

片段标识符（fragment identifier）指的是，URL 的 # 号后面的部分，比如 http://example.com/x.html#fragment 的 #fragment。如果只是改变片段标识符，页面不会重新刷新。

父窗口可以把信息，写入子窗口的片段标识符。

```
let src = originURL + '#' + data;
document.getElementById('myIFrame').src = src;
```

上面代码中，父窗口把所要传递的信息，写入 iframe 窗口的片段标识符。

子窗口通过监听 hashchange 事件得到通知。

```
window.onhashchange = checkMessage;

function checkMessage() {
    let message = window.location.hash;
    // ...
}
```

同样的，子窗口也可以改变父窗口的片段标识符。

```
parent.location.href = target + '#' + hash;
window.postMessage()
```

上面的这种方法属于破解，HTML5 为了解决这个问题，引入了一个全新的 API：跨文档通信 API（Cross-document messaging）。

这个 API 为 window 对象新增了一个 window.postMessage 方法，允许跨窗口通信，不论这两个窗口是否同源。举例来说，父窗口 aaa.com 向子窗口 bbb.com 发消息，调用 postMessage 方法就可以了。

```
// 父窗口打开一个子窗口
let popup = window.open('http://bbb.com', 'title');
// 父窗口向子窗口发消息
popup.postMessage('Hello World!', 'http://bbb.com');
```

postMessage 方法的第一个参数是具体的信息内容，第二个参数是接收消息的窗口的源（origin），即"协议 + 域名 + 端口"。也可以设为 *，表示不限制域名，向所有窗口发送。

子窗口向父窗口发送消息的写法类似。

```
// 子窗口向父窗口发消息
window.opener.postMessage('Nice to see you', 'http://aaa.com');
```

父窗口和子窗口都可以通过 message 事件，监听对方的消息。

```
// 父窗口和子窗口都可以用下面的代码，
// 监听 message 消息
window.addEventListener('message', function (e) {
    console.log(e.data);
},false);
```

`message` 事件的参数是事件对象 `event`，提供以下三个属性。

- `event.source`：发送消息的窗口
- `event.origin`: 消息发向的网址
- `event.data`: 消息内容

下面的例子是，子窗口通过 `event.source` 属性引用父窗口，然后发送消息。

```
window.addEventListener('message', receiveMessage);
function receiveMessage(event) {
    event.source.postMessage('Nice to see you!', '\*');
}
```

上面代码有几个地方需要注意。首先，`receiveMessage` 函数里面没有过滤信息的来源，任意网址发来的信息都会被处理。其次，`postMessage` 方法中指定的目标窗口的网址是一个星号，表示该信息可以向任意网址发送。通常来说，这两种做法是不推荐的，因为不够安全，可能会被恶意利用。

`event.origin` 属性可以过滤不是发给本窗口的消息。

```
window.addEventListener('message', receiveMessage);
function receiveMessage(event) {
    if (event.origin !== 'http://aaa.com') return;
    if (event.data === 'Hello World') {
        event.source.postMessage('Hello', event.origin);
    } else {
        console.log(event.data);
    }
}
```

3. LocalStorage

通过 `window.postMessage`，读写其他窗口的 LocalStorage 也成为了可能。

下面是一个例子，主窗口写入 `iframe` 子窗口的 `localStorage`。

```
window.onmessage = function(e) {
    if (e.origin !== 'http://bbb.com') {
        return;
    }
    let payload = JSON.parse(e.data);
```

```
    localStorage.setItem(payload.key, JSON.stringify(payload.data));
};
```

上面代码中，子窗口将父窗口发来的消息，写入自己的 `LocalStorage`。

父窗口发送消息的代码如下。

```
let win = document.getElementsByTagName('iframe')[0].contentWindow;
let obj = { name: 'Jack' };
win.postMessage(JSON.stringify({key: 'storage', data: obj}), 'http://bbb.com');
```

加强版的子窗口接收消息的代码如下。

```
window.onmessage = function(e) {
    if (e.origin !== 'http://bbb.com') return;
    let payload = JSON.parse(e.data);
    switch (payload.method) {
        case 'set':
            localStorage.setItem(payload.key, JSON.stringify(payload.data));
            break;
        case 'get':
            let parent = window.parent;
            let data = localStorage.getItem(payload.key);
            parent.postMessage(data, 'http://aaa.com');
            break;
        case 'remove':
            localStorage.removeItem(payload.key);
            break;
    }
};
```

加强版的父窗口发送消息代码如下。

```
let win = document.getElementsByTagName('iframe')[0].contentWindow;
let obj = { name: 'Jack' };
// 存入对象
win.postMessage(JSON.stringify({key: 'storage', method: 'set', data:
obj}),'http://bbb.com');
// 读取对象
win.postMessage(JSON.stringify({key: 'storage', method: "get"}),"\*");
window.onmessage = function(e) {
    if (e.origin != 'http://aaa.com') return;
    console.log(JSON.parse(e.data).name);
};
```

4. AJAX

同源政策规定，AJAX 请求只能发给同源的网址，否则就报错。

除了架设服务器代理（浏览器请求同源服务器，再由后者请求外部服务），有三种方法规避这个限制。

- JSONP
- WebSocket
- CORS

4.1. JSONP

JSONP 是服务器与客户端跨源通信的常用方法。最大特点就是简单易用，没有兼容性问题，老式浏览器全部支持，服务端改造非常小。但只能发出 Get 请求。

它的做法如下。

第一步，网页添加一个 <script> 元素，向服务器请求一个脚本，这不受同源政策限制，可以跨域请求。

```
<script src="http://api.foo.com?callback=bar"></script>
```

请求的脚本网址有一个 callback 参数（?callback=bar），用来告诉服务器，客户端的回调函数名称（bar）。

第二步，服务器收到请求后，拼接一个字符串，将 JSON 数据放在函数名里面，作为字符串返回（bar({...})）。

第三步，客户端会将服务器返回的字符串，作为代码解析，因为浏览器认为，这是 <script> 标签请求的脚本内容。这时，客户端只要定义了 bar() 函数，就能在该函数体内，拿到服务器返回的 JSON 数据。

一个实例：首先，网页动态插入 <script> 元素，由它向跨域网址发出请求。

```
function addScriptTag(src) {
    let script = document.createElement('script');
    script.setAttribute('type', 'application/javascript');
    script.src = src;
    document.body.appendChild(script);
}

window.onload = function () {
    addScriptTag('http://example.com/ip?callback=foo');
}

function foo(data) {
    console.log('Your public IP address is: ' + data.ip);
};
```

上面代码通过 **动态添加 <script> 元素，向服务器 example.com 发出请求。** 该请求的查询字符串有一个 callback 参数，用来指定回调函数的名字，这对于 JSONP 是必需的。

服务器收到这个请求以后，会将数据放在回调函数的参数位置返回。

```
foo({
    'ip': '8.8.8.8'
});
```

由于 `<script>` 元素请求的脚本，直接作为代码运行。这时，只要浏览器定义了 `foo` 函数，该函数就会立即调用。作为参数的 `JSON` 数据被视为 `JavaScript` 对象，而不是字符串，因此避免了使用 `JSON.parse` 的步骤。

4.2. WebSocket

`WebSocket` 是一种通信协议，使用 `ws://`（非加密）和 `wss://`（加密）作为协议前缀。`WebSocket` **不实行同源政策，只要服务器支持，就可以通过它进行跨源通信。**

下面是一个例子，浏览器发出的 `WebSocket` 请求的头信息（摘自维基百科）。

```
    GET /chat HTTP/1.1
    Host: server.example.com
    Upgrade: websocket
    Connection: Upgrade
    Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
    Sec-WebSocket-Protocol: chat, superchat
    Sec-WebSocket-Version: 13
    Origin: http://example.com
```

上面代码中，有一个字段是 `Origin`，表示该请求的请求源（`origin`），即发自哪个域名。

正是因为有了 `Origin` 这个字段，所以 `WebSocket` 才没有实行同源政策。因为服务器可以根据这个字段，判断是否许可本次通信。如果该域名在白名单内，服务器就会做出如下回应。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

4.4. CORS

CORS 是跨源资源分享（Cross-Origin Resource Sharing）的缩写。它是 W3C 标准，属于跨源 AJAX 请求的根本解决方法。**相比 JSONP 只能发 GET 请求，CORS 允许任何类型的请求。**