前端跨页面通信，你知道哪些方法?

引言

在浏览器中，我们可以同时打开多个Tab页，每个Tab页可以粗略理解为一个“独立”的运行环境，即使是全局对象也不会在多个Tab间共享。然而有些时候，我们希望能在这些“独立”的Tab页面之间同步页面的数据、信息或状态。

正如下面这个例子：我在列表页点击“收藏”后，对应的详情页按钮会自动更新为“已收藏”状态；类似的，在详情页点击“收藏”后，列表页中按钮也会更新。

这就是我们所说的前端跨页面通信。

你知道哪些跨页面通信的方式呢？如果不清楚，下面我就带大家来看看七种跨页面通信的方式。

一、同源页面间的跨页面通信

浏览器的同源策略在下述的一些跨页面通信方法中依然存在限制。因此，我们先来看看，在满足同源策略的情况下，都有哪些技术可以用来实现跨页面通信。

1. BroadCast Channel

BroadCast Channel 可以帮我们创建一个用于广播的通信频道。当所有页面都监听同一频道的消息时，其中某一个页面通过它发送的消息就会被其他所有页面收到。它的API和用法都非常简单。

下面的方式就可以创建一个标识为 `AlienZHOU` 的频道：

```
复制代码
const bc = new BroadcastChannel('AlienZHOU');
```

各个页面可以通过 `onmessage` 来监听被广播的消息：

- 复制代码
  bc.onmessage = function (e) { const data = e.data; const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[BroadcastChannel] receive message:', text);};

要发送消息时只需要调用实例上的 `postMessage` 方法即可：

```
复制代码
bc.postMessage(mydata);
```

> Broadcast Channel 的具体的使用方式可以看这篇《【3分钟速览】前端广播式通信：Broadcast Channel》。

2. Service Worker

Service Worker 是一个可以长期运行在后台的 Worker，能够实现与页面的双向通信。多页面共享间的 Service Worker 可以共享，将 Service Worker 作为消息的处理中心（中央站）即可实现广播效果。

> Service Worker 也是 PWA 中的核心技术之一，由于本文重点不在 PWA ，因此如果想进一步了解 Service Worker，可以阅读我之前的文章【PWA学习与实践】(3) 让你的WebApp离线可用。

首先，需要在页面注册 Service Worker：

```
复制代码
/* 页面逻辑 */navigator.serviceWorker.register('../util.sw.js').then(function () {    console.log('Service Worker 注册成功');});
```

其中 `../util.sw.js` 是对应的 Service Worker 脚本。Service Worker 本身并不自动具备“广播通信”的功能，需要我们添加些代码，将其改造成消息中转站：

```
复制代码
/* ../util.sw.js Service Worker 逻辑 */self.addEventListener('message', function (e) {    console.log('service worker receive message', e.data);  e.waitUntil(        self.clients.matchAll().then(function (clients) {    if (!clients || clients.length === 0) {                return;            }            clients.forEach(function (client) {  client.postMessage(e.data);            });        })    );});
```

我们在 Service Worker 中监听了 `message` 事件，获取页面（从 Service Worker 的角度叫 client）发送的信息。然后通过 `self.clients.matchAll()` 获取当前注册了该 Service Worker 的所有页面，通过调用每个client（即页面）的 `postMessage` 方法，向页面发送消息。这样就把从一处（某个Tab页面）收到的消息通知给了其他页面。

处理完 Service Worker，我们需要在页面监听 Service Worker 发送来的消息：

- 复制代码
  /* 页面逻辑 */navigator.serviceWorker.addEventListener('message', function (e) { const data = e.data; const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[Service Worker] receive message:', text);});

最后，当需要同步消息时，可以调用 Service Worker 的 `postMessage` 方法：

```
复制代码
/* 页面逻辑 */navigator.serviceWorker.controller.postMessage(mydata);
```

3. LocalStorage

LocalStorage 作为前端最常用的本地存储，大家应该已经非常熟悉了；但 `StorageEvent` 这个与它相关的事件有些同学可能会比较陌生。

当 LocalStorage 变化时，会触发 `storage` 事件。利用这个特性，我们可以在发送消息时，把消息写入到某个 LocalStorage 中；然后在各个页面内，通过监听 `storage` 事件即可收到通知。

- 复制代码
  window.addEventListener('storage', function (e) { if (e.key === 'ctc-msg') { const data = JSON.parse(e.newValue); const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[Storage I] receive message:', text); }});

在各个页面添加如上的代码，即可监听到 LocalStorage 的变化。当某个页面需要发送消息时，只需要使用我们熟悉的 `setItem` 方法即可：

```
复制代码
mydata.st = +(new Date);window.localStorage.setItem('ctc-msg',
JSON.stringify(mydata));
```

注意，这里有一个细节：我们在mydata上添加了一个取当前毫秒时间戳的 `.st` 属性。这是因为，`storage` 事件只有在值真正改变时才会触发。举个例子：

```
复制代码
window.localStorage.setItem('test', '123');window.localStorage.setItem('test',
'123');
```

由于第二次的值 `'123'` 与第一次的值相同，所以以上的代码只会在第一次 `setItem` 时触发 `storage` 事件。因此我们通过设置 `st` 来保证每次调用时一定会触发 `storage` 事件。

小憩一下

上面我们看到了三种实现跨页面通信的方式，不论是建立广播频道的 Broadcast Channel，还是使用 Service Worker 的消息中转站，抑或是些 tricky 的 `storage` 事件，其都是“广播模式”：一个页面将消息通知给一个“中央站”，再由“中央站”通知给各个页面。

在上面的例子中，这个“中央站”可以是一个 BroadCast Channel 实例、一个 Service Worker 或是 LocalStorage。

下面我们会看到另外两种跨页面通信方式，我把它称为“共享存储+轮询模式”。

4. Shared Worker

Shared Worker 是 Worker 家族的另一个成员。普通的 Worker 之间是独立运行、数据互不相通；而多个 Tab 注册的 Shared Worker 则可以实现数据共享。

Shared Worker 在实现跨页面通信时的问题在于，它无法主动通知所有页面，因此，我们会使用轮询的方式，来拉取最新的数据。思路如下：

让 Shared Worker 支持两种消息。一种是 post，Shared Worker 收到后会将该数据保存下来；另一种是 get，Shared Worker 收到该消息后会将保存的数据通过 `postMessage` 传给注册它的页面。也就是让页面通过 get 来主动获取（同步）最新消息。具体实现如下：

首先，我们会在页面中启动一个 Shared Worker，启动方式非常简单：

```
复制代码
// 构造函数的第二个参数是 Shared Worker 名称，也可以留空const sharedWorker = new SharedWorker('../util.shared.js', 'ctc');
```

然后，在该 Shared Worker 中支持 get 与 post 形式的消息：

```
复制代码
/* ../util.shared.js: Shared Worker 代码 */let data = null;self.addEventListener('connect', function (e) {    const port = e.ports[0];    port.addEventListener('message', function (event) {        // get 指令则返回存储的消息数据        if (event.data.get) {            data && port.postMessage(data);        }        // 非 get 指令则存储该消息数据        else {            data = event.data;        }    });    port.start();});
```

之后，页面定时发送 get 指令的消息给 Shared Worker，轮询最新的消息数据，并在页面监听返回信息：

- 复制代码
  // 定时轮询，发送 get 指令的消息setInterval(function () {
  sharedWorker.port.postMessage({get: true});}, 1000);// 监听 get 消息的返回数据
  sharedWorker.port.addEventListener('message', (e) => { const data = e.data; const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[Shared Worker] receive message:', text);}, false);sharedWorker.port.start();

最后，当要跨页面通信时，只需给 Shared Worker `postMessage` 即可：

```
复制代码
sharedWorker.port.postMessage(mydata);
```

> 注意，如果使用 `addEventListener` 来添加 Shared Worker 的消息监听，需要显式调用 `MessagePort.start` 方法，即上文中的 `sharedWorker.port.start()`；如果使用 `onmessage` 绑定监听则不需要。

5. IndexedDB

除了可以利用 Shared Worker 来共享存储数据，还可以使用其他一些“全局性”（支持跨页面）的存储方案。例如 IndexedDB 或 cookie。

鉴于大家对 cookie 已经很熟悉，加之作为“互联网最早期的存储方案之一”，cookie 已经在实际应用中承受了远多于其设计之初的责任，我们下面会使用 IndexedDB 来实现。

其思路很简单：与 Shared Worker 方案类似，消息发送方将消息存至 IndexedDB 中；接收方（例如所有页面）则通过轮询去获取最新的信息。在这之前，我们先简单封装几个 IndexedDB 的工具方法。

- 打开数据库连接：

```
复制代码
function openStore() {    const storeName = 'ctc_aleinzhou';    return new Promise(function (resolve, reject) {        if (!('indexedDB' in window)) {            return reject('don\'t support indexedDB');        }        const request = indexedDB.open('CTC_DB', 1);        request.onerror = reject;        request.onsuccess =  e => resolve(e.target.result);        request.onupgradeneeded = function (e) {            const db = e.srcElement.result;            if (e.oldVersion === 0 && !db.objectStoreNames.contains(storeName)) {                const store = db.createObjectStore(storeName, {keyPath: 'tag'});                store.createIndex(storeName + 'Index', 'tag', {unique: false});            }        };    });}
```

- 存储数据

```
复制代码
function saveData(db, data) {    return new Promise(function (resolve, reject) {        const STORE_NAME = 'ctc_aleinzhou';        const tx = db.transaction(STORE_NAME, 'readwrite');        const store = tx.objectStore(STORE_NAME);        const request = store.put({tag: 'ctc_data', data});        request.onsuccess = () => resolve(db);        request.onerror = reject;    });}
```

- 查询/读取数据

```
复制代码
function query(db) {    const STORE_NAME = 'ctc_aleinzhou';    return new Promise(function (resolve, reject) {        try {            const tx = db.transaction(STORE_NAME, 'readonly');            const store = tx.objectStore(STORE_NAME);            const dbRequest = store.get('ctc_data');            dbRequest.onsuccess = e => resolve(e.target.result);            dbRequest.onerror = reject;        }        catch (err) {            reject(err);        }    });}
```

剩下的工作就非常简单了。首先打开数据连接，并初始化数据：

```
复制代码
openStore().then(db => saveData(db, null))
```

对于消息读取，可以在连接与初始化后轮询：

- 复制代码
  openStore().then(db => saveData(db, null)).then(function (db) { setInterval(function () { query(db).then(function (res) { if (!res || !res.data) { return; } const data = res.data; const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[Storage I] receive message:', text); }); }, 1000);});

最后，要发送消息时，只需向 IndexedDB 存储数据即可：

```
复制代码
openStore().then(db => saveData(db, null)).then(function (db) {    // …… 省略上面的轮询代码    // 触发 saveData 的方法可以放在用户操作的事件监听内    saveData(db, mydata);});
```

小憩一下

在“广播模式”外，我们又了解了“共享存储+长轮询”这种模式。也许你会认为长轮询没有监听模式优雅，但实际上，有些时候使用“共享存储”的形式时，不一定要搭配长轮询。

例如，在多 Tab 场景下，我们可能会离开 Tab A 到另一个 Tab B 中操作；过了一会我们从 Tab B 切换回 Tab A 时，希望将之前在 Tab B 中的操作的信息同步回来。这时候，其实只用在 Tab A 中监听 `visibilitychange` 这样的事件，来做一次信息同步即可。

下面，我会再介绍一种通信方式，我把它称为“口口相传”模式。

---

6. window.open + window.opener

当我们使用 `window.open` 打开页面时，方法会返回一个被打开页面 `window` 的引用。而在未显示指定 `noopener` 时，被打开的页面可以通过 `window.opener` 获取到打开它的页面的引用 —— 通过这种方式我们就将这些页面建立起了联系（一种树形结构）。

首先，我们把 `window.open` 打开的页面的 `window` 对象收集起来：

```
复制代码
let childWins = [];document.getElementById('btn').addEventListener('click',
function () {    const win = window.open('./some/sample');
childWins.push(win);});
```

然后，当我们需要发送消息的时候，作为消息的发起方，一个页面需要同时通知它打开的页面与打开它的页面：

```
复制代码
// 过滤掉已经关闭的窗口childWins = childWins.filter(w => !w.closed);if
(childWins.length > 0) {    mydata.fromOpenner = false;    childWins.forEach(w
=> w.postMessage(mydata));}if (window.opener && !window.opener.closed) {
mydata.fromOpenner = true;    window.opener.postMessage(mydata);}
```

注意，我这里先用 `.closed` 属性过滤掉已经被关闭的 Tab 窗口。这样，作为消息发送方的任务就完成了。下面看看，作为消息接收方，它需要做什么。

> 此时，一个收到消息的页面就不能那么自私了，除了展示收到的消息，它还需要将消息再传递给它所“知道的人”（打开与被它打开的页面）：
>
> 需要注意的是，我这里通过判断消息来源，避免将消息回传给发送方，防止消息在两者间死循环的传递。（该方案会有些其他小问题，实际中可以进一步优化）

- 复制代码
  window.addEventListener('message', function (e) { const data = e.data; const text = '[receive] 'data.msg + ' —— tab 'data.from; console.log('[Cross-document Messaging] receive message:', text); // 避免消息回传 if (window.opener && !window.opener.closed && data.fromOpenner) {   window.opener.postMessage(data); } // 过滤掉已经关闭的窗口 childWins = childWins.filter(w => !w.closed); // 避免消息回传 if (childWins && !data.fromOpenner) {   childWins.forEach(w => w.postMessage(data)); }});

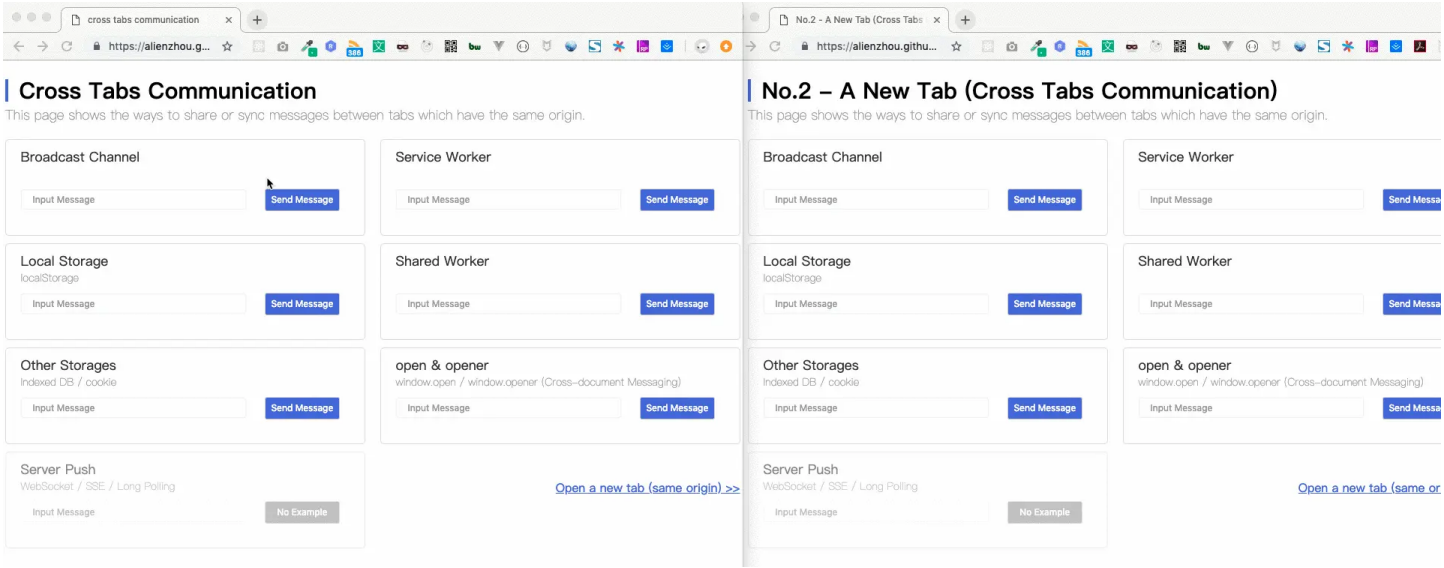
这样，每个节点（页面）都肩负起了传递消息的责任，也就是我说的“口口相传”，而消息就在这个树状结构中流转了起来。

小憩一下

显然，“口口相传”的模式存在一个问题：如果页面不是通过在另一个页面内的 `window.open` 打开的（例如直接在地址栏输入，或从其他网站链接过来），这个联系就被打破了。

除了上面这六个常见方法，其实还有一种（第七种）做法是通过 WebSocket 这类的“服务器推”技术来进行同步。这好比将我们的“中央站”从前端移到了后端。

此外，我还针对以上各种方式写了一个 在线演示的 Demo >>



二、非同源页面之间的通信

上面我们介绍了七种前端跨页面通信的方法，但它们大都受到同源策略的限制。然而有时候，我们有两个不同域名的产品线，也希望它们下面的所有页面之间能无障碍地通信。那该怎么办呢？

要实现该功能，可以使用一个用户不可见的 iframe 作为“桥”。由于 iframe 与父页面间可以通过指定 `origin` 来忽略同源限制，因此可以在每个页面中嵌入一个 iframe （例如：`http://sample.com/bridge.html`），而这些 iframe 由于使用的是一个 url，因此属于同源页面，其通信方式可以复用上面第一部分提到的各种方式。

页面与 iframe 通信非常简单，首先需要在页面中监听 iframe 发来的消息，做相应的业务处理：

```
复制代码
/* 业务页面代码 */window.addEventListener('message', function (e) {     // …… do something});
```

然后，当页面要与其他的同源或非同源页面通信时，会先给 iframe 发送消息：

```
复制代码
/* 业务页面代码 */window.frames[0].window.postMessage(mydata, '*');
```

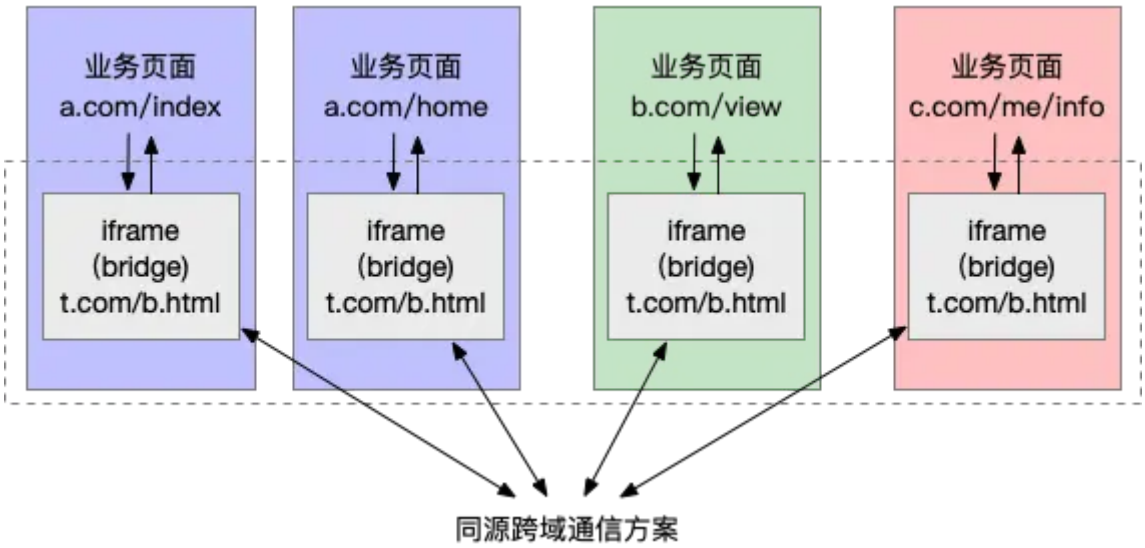
其中为了简便此处将 `postMessage` 的第二个参数设为了 `'*'` ，你也可以设为 iframe 的 URL。iframe 收到消息后，会使用某种跨页面消息通信技术在所有 iframe 间同步消息，例如下面使用的 Broadcast Channel：

```
复制代码
/* iframe 内代码 */const bc = new BroadcastChannel('AlienZHOU');// 收到来自页面的消息后，在 iframe 间进行广播window.addEventListener('message', function (e) {bc.postMessage(e.data);});
```

其他 iframe 收到通知后，则会将该消息同步给所属的页面：

```
复制代码
/* iframe 内代码 */// 对于收到的（iframe）广播消息，通知给所属的业务页面bc.onmessage = function (e) {    window.parent.postMessage(e.data, '*');};
```

下图就是使用 iframe 作为“桥”的非同源页面间通信模式图。



其中“同源跨域通信方案”可以使用文章第一部分提到的某种技术。

总结

今天和大家分享了一下跨页面通信的各种方式。

对于同源页面，常见的方式包括：

- 广播模式：Broadcast Channe / Service Worker / LocalStorage + StorageEvent

- 共享存储模式：Shared Worker / IndexedDB / cookie
- 口口相传模式：window.open + window.opener
- 基于服务端：Websocket / Comet / SSE 等

而对于非同源页面，则可以通过嵌入同源 iframe 作为“桥”，将非同源页面通信转换为同源页面通信。