

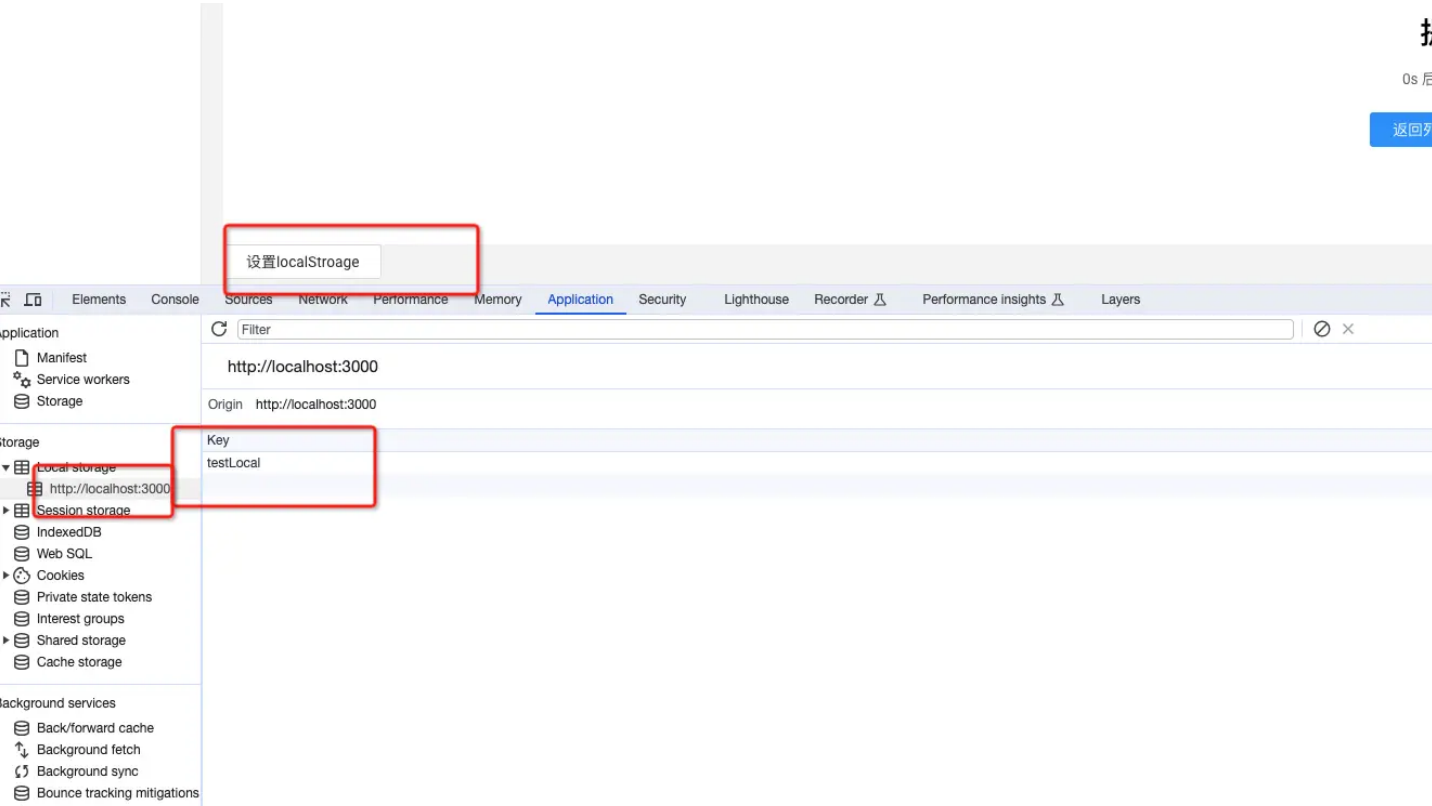
面试官：跨页面通信的方式有哪些？

同源页面间通信

什么是同源这个不过多解释了，大家可以自己了解一下 [浏览器同源策略]

1. localStorage

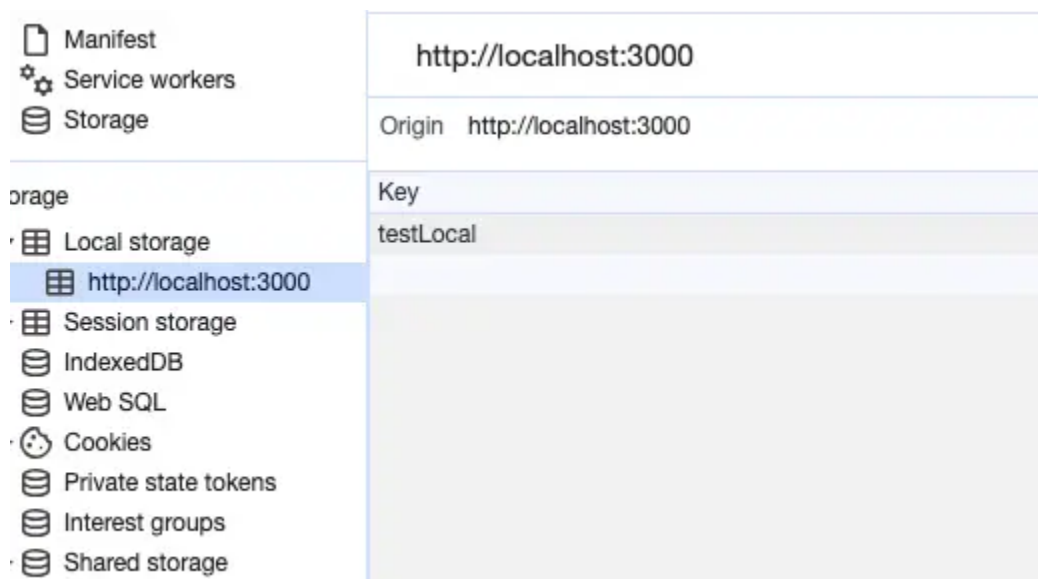
可以通过 `localStorage.setItem()` 在一个页面去写入一个值，然后在另一个页面中去获取。我们在A页面中添加一个按钮，用来设置 `localStorage`，然后在B页面添加一个定时器获取A页面设置的 `localStorage`，分别用直接链接访问和A页面跳转的方式打开B页面看是否能获取到A页面设置的 `localStorage` 的值。



A页面设置 `localStorage` 值

直接通过A页面调整B页面的结果





可以看到是可以访问到的。

通过链接直接访问B页面结果

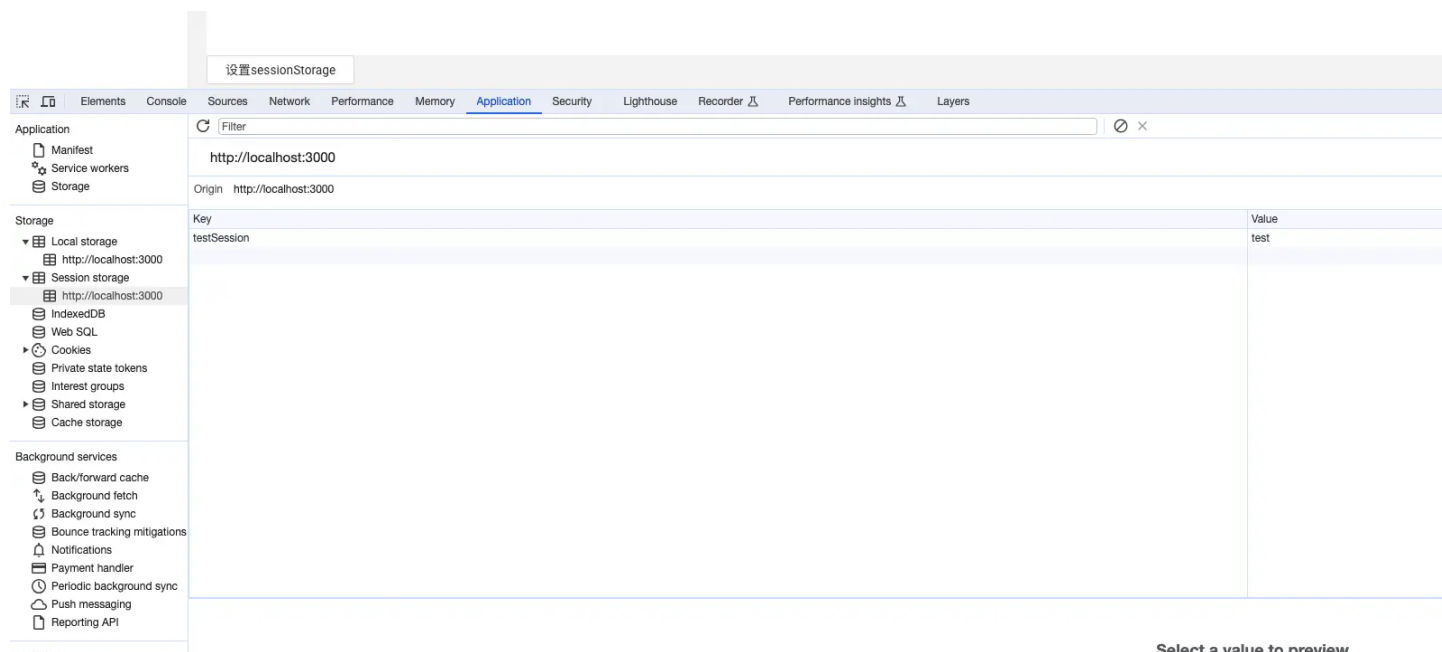


结果同上是可以访问到的。

2. sessionStroage

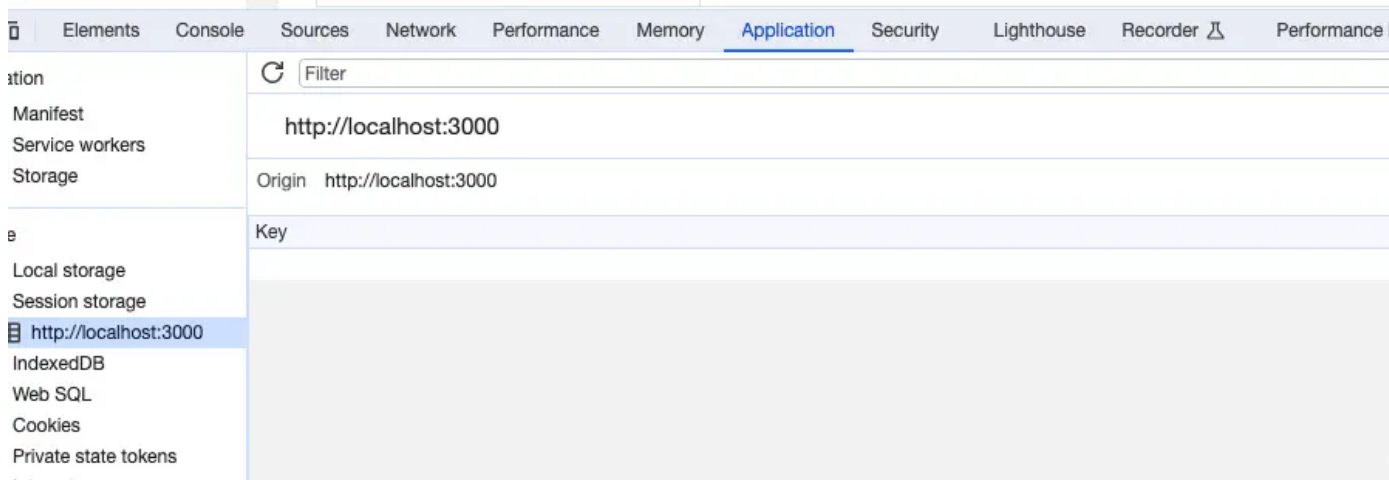
操作方式同上面 localStroage ，下面看一下结果

首先看A页面设置结果：



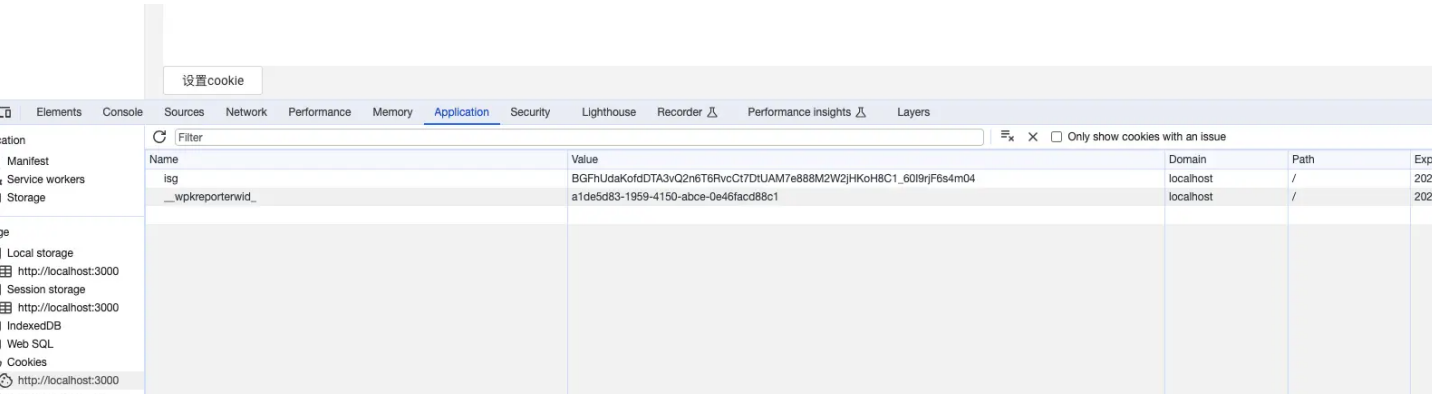
然后通过A页面跳转和直接访问B页面都得到结果是获取不到A页面设置的 sessionStroage：

sessionStroage不支持跨页面通信。



3.cookie

操作方式同上，在A页面设置了 testCookie=123 但是并不会在cookie中马上显示要刷新才能显示



通过两种方式打开B页面的结果

通过两种方式打开的B页面都可以获取到在A页面的cookie，所以可以通过Cookie进行跨页面通信。



White

chloe.white@example.com

ents

Console

Sources

Network

Performance

Memory

Application

Security

Lighthouse

Recorder

Performance insights

Layers

Filter

<

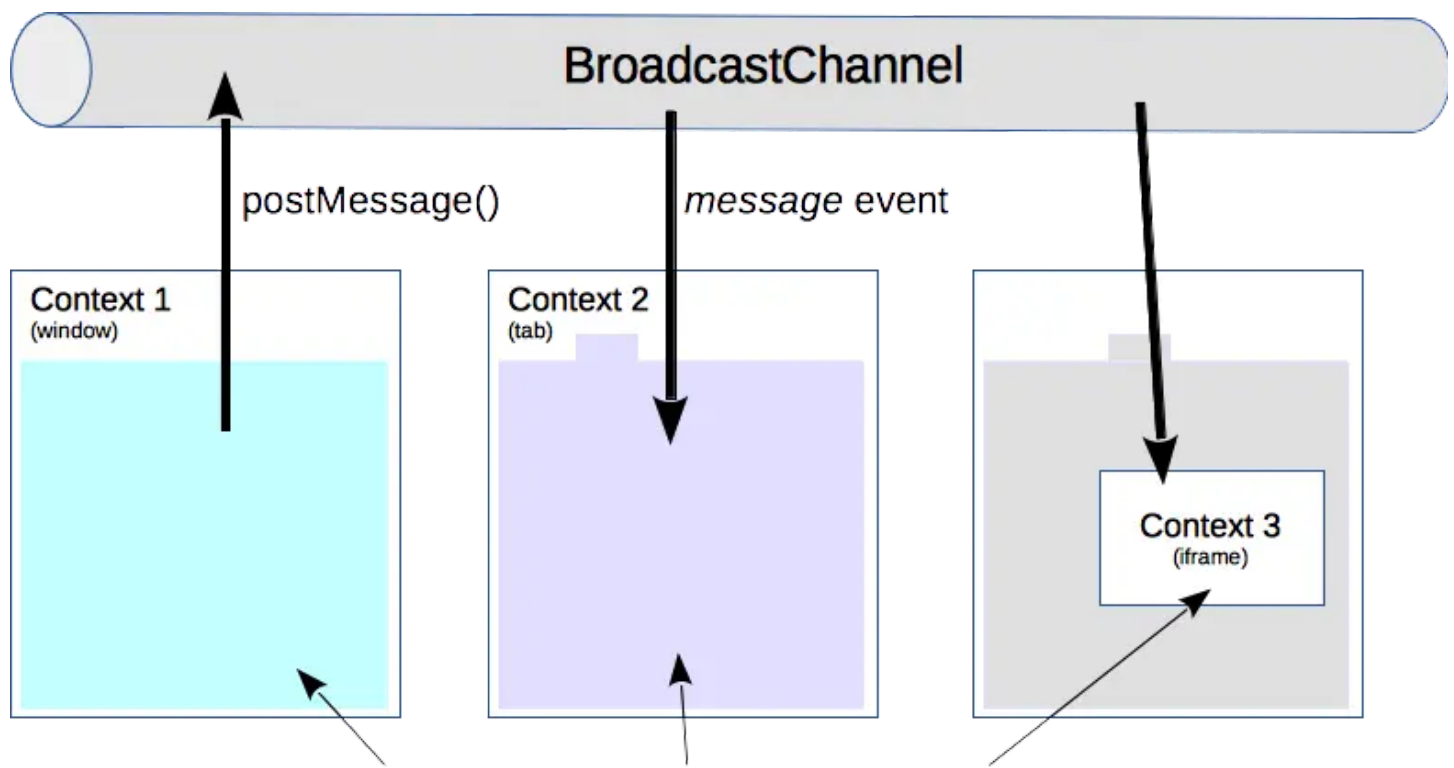
3. indexedDB 和 Web sql

如果页面有大量的数据需要交互，同时也需要做一些持久化的操作，前端的sql操作是一个比较好的选择，可以通过indexedDB创建一个关系型数据库来做数据存储，通过sql操作实现页面间的通信，更多的内容可以了解一下IndexedDB的使用。

5.Broadcast Channel API

Broadcast Channel API是HTML5提供的一种跨页面通信机制。它允许不同页面之间通过共享一个频道来进行通信。一个页面可以向频道发送消息，其他页面可以监听该频道以接收消息。

通过创建一个监听某个频道下的 `BroadcastChannel` 对象，你可以接收发送给该频道的所有消息。一个有意思的点是，你不需要再维护需要通信的 `iframe` 或 `worker` 的索引。它们可以通过构造 `BroadcastChannel` 来简单地“订阅”特定频道，并在它们之间进行全双工（双向）通信。



1. 首先创建一个 `BroadcastChannel` 对象：

```
1 var bc = new BroadcastChannel("test_tab");
```

1. 发送消息：

```
1 bc.postMessage("This is a test message.");
```

1. 接收消息：

```
1 bc.onmessage = function (ev) {  
2   console.log(ev);  
3 };
```

1. 与频道断开连接：

```
1 bc.close();
```

通过上面方法，A页面发送信息，在B页面就可以接收到A页面发送的消息。

A页面代码：

```
1 export default function Success() {  
2   const setLocal = () => {  
3     const bc = new BroadcastChannel('test_tab')  
4     bc.postMessage('this is a test message')  
5   }  
6   return (  
7     <Button onClick={setLocal}>设置消息</Button>  
8   );  
9 }
```

B页面代码：

```
1 const bc = new BroadcastChannel('test_tab')  
2 bc.onmessage = (ev) => {  
3   console.log('message', ev)  
4 }
```

结果如下：

```
... 2 actionListSpan 3
l... message ▼ MessageEvent {isTrusted: true, data: 'this is a test message', origin: 'http://localhost:3000', lastEventId: '', source: null, ...}
1gs   isTrusted: true
    bubbles: false
    cancelBubble: false
    cancelable: false
    composed: false
    ▶ currentTarget: BroadcastChannel {name: 'test_tab', onmessageerror: null, onmessage: f}
    data: "this is a test message"
    defaultPrevented: false
    eventPhase: 0
    lastEventId: ""
    origin: "http://localhost:3000"
    ▶ ports: []
    returnValue: true
    source: null
    ▶ srcElement: BroadcastChannel {name: 'test_tab', onmessageerror: null, onmessage: f}
    ▶ target: BroadcastChannel {name: 'test_tab', onmessageerror: null, onmessage: f}
    timeStamp: 200282.40000003576
    type: "message"
    userActivation: null
    ▶ [[Prototype]]: MessageEvent
se
关闭了快捷键
```

虽然 Broadcast Channel API 方便好用，但是在兼容性上不是特别好。

Chrome	Edge *	Safari	Firefox	Opera	IE ! *	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *	UC Browser for Android	Android Browser *	Firefox for Android	QQ Browser	Baidu Browser	KaiOS Browser
4-53	12-18	3.1-15.3	2-37	10-40			3.2-15.3	4-5.4								
54-118	79-118	15.4-17.0	38-118	41-102	6-10		15.4-17.0	6.2-22		12-12.1		2.1-4.4.4				2.5
119	119	17.1	119	103	11	119	17.1	23	all	73	15.5	119	119	13.1	13.18	3.1
120-122		17.2-TP	120-122				17.2									

2. PostMessage API

PostMessage API允许在不同浏览上下文（如不同窗口、iframe或跨域页面）之间进行通信。页面可以使用postMessage方法发送消息，接收方页面可以通过监听message事件来接收和处理消息。

```
1 // A页面
2 const targetWindow = window.open("目标页面的URL");
3
4 // 发送消息
5 const message = "Hello, target page!";
6 const targetOrigin = "目标页面的URL";
7 targetWindow.postMessage(message, targetOrigin);
8
```

```
1 // 目标页面
2 window.addEventListener("message", function(event) {
3   // 确认消息来源
4   const allowedOrigin = "源页面的URL";
5   if (event.origin !== allowedOrigin) return;
6
7   // 处理接收到的消息
8   const message = event.data;
9   console.log("接收到的消息:", message);
10 });
```

在A页面中，我们使用postMessage方法向目标页面发送消息。首先，我们通过window.open方法打开目标页面的URL，然后使用postMessage方法发送消息。在发送消息时，我们需要传递两个参数：要发送的消息内容和目标页面的URL。

在目标页面中，我们使用window.addEventListener方法监听message事件。当消息被发送时，页面会触发message事件并传递一个event对象。我们可以通过event.data属性获取接收到的消息内容。在处理消息之前，我们可以验证消息的来源，以确保通信的安全性。

通过使用PostMessage API，源页面和目标页面可以在跨域或不同窗口之间进行通信。这种方式可用于实现多种跨页面交互，如传递数据、同步状态等。请注意，为了确保安全性，应该验证消息的来源和目标，以防止恶意代码的注入。

3. Service Worker

Service Worker 是运行在浏览器后台的脚本，可以拦截和处理网络请求。虽然 Service Worker 主要用于离线缓存和推送通知等功能，但也可以用于实现跨页面通信。

在注册 Service Worker 时，我们可以监听 message 事件来接收和处理消息：

```
1 // 注册 Service Worker
2 navigator.serviceWorker.register('service-worker.js');
3
4 // 监听 message 事件
5 navigator.serviceWorker.addEventListener('message', function(event) {
6   // 处理接收到的消息
7   const message = event.data;
8   console.log('接收到的消息: ', message);
9 });
```

在源页面中，我们使用 postMessage 方法向 Service Worker 发送消息：

```
1 // A页面
2 navigator.serviceWorker.controller.postMessage('Hello, Service Worker!');
```

在 Service Worker 脚本中，我们可以监听 message 事件来接收和处理消息，并向所有客户端页面发送消息：

```
1 // B页面
2
3 // 监听 message 事件
```



```

4 self.addEventListener('message', function(event) {
5   // 处理接收到的消息
6   const message = event.data;
7   console.log('接收到的消息: ', message);
8
9   // 向所有客户端页面发送消息
10  self.clients.matchAll().then(function(clients) {
11    clients.forEach(function(client) {
12      client.postMessage('Hello, client page!');
13    });
14  });
15 });

```

在A页面中，使用 `navigator.serviceWorker.controller.postMessage` 方法向 Service Worker 发送消息。在 B页面中，可以通过监听 message 事件来接收和处理消息，并使用 `self.clients.matchAll` 方法获取所有的客户端页面，并向每个页面发送消息。

使用 Service Worker 进行跨页面通信的好处是，Service Worker 可以在后台运行并独立于页面，这意味着即使没有页面打开，也可以进行跨页面通信。这对于实现离线通知、消息同步等功能非常有用。

请注意，Service Worker 只能与同源页面通信，因此源页面和 Service Worker 脚本必须在同一域下。并且由于 Service Worker 生命周期的特性，首次注册成功后，才能在后续页面加载中接收到消息。

4. web Worker

Web Worker 是一种运行在后台的 JavaScript 线程，可以用于执行长时间运行的任务而不会阻塞主线程。Web Worker 本身不能直接进行跨页面通信，但可以通过 MessageChannel API 进行跨页面通信

```

1 // A页面
2 const sharedWorker = new SharedWorker('shared-worker.js');
3 sharedWorker.port.onmessage = function(event) {
4   // 处理接收到的消息
5   const message = event.data;
6   console.log('接收到的消息: ', message);
7 };
8
9 // 发送消息给 Shared Worker
10 sharedWorker.port.postMessage('Hello, Shared Worker!');
11

```

```

1 // B页面
2 const sharedWorker = new SharedWorker('shared-worker.js');
3 sharedWorker.port.onmessage = function(event) {
4   // 处理接收到的消息

```



```
5   const message = event.data;
6   console.log('接收到的消息: ', message);
7 };
8
```

```
1 // shared-worker.js
2
3 // 监听 connect 事件
4 self.onconnect = function(event) {
5   // 获取通信端口
6   const port = event.ports[0];
7
8   // 监听消息
9   port.onmessage = function(event) {
10     // 处理接收到的消息
11     const message = event.data;
12     console.log('接收到的消息: ', message);
13
14     // 向所有连接的源页面发送消息
15     port.postMessage('Hello, source pages!');
16   };
17 };
18
```

5. WebSocket

WebSocket 是一种在 Web 应用程序中实现双向通信的协议。它允许在客户端和服务器之间建立持久的连接，以便实时地发送数据。

可以通过WebSocket建立一个长连接通过后端服务器的中转进行页面间的通信。

非同源

1. 通过链接跳转的方式，将需要用到的参数进行传递。
2. 通过内嵌一个同源的iframe页面，先将参数传给iframe页面，然后让iframe页面进行上面的同源操作进行交互。

小结

通过上面的方法可以进行跨页面通信，我已经揉碎了吃到嘴里了，接下来就是嚼碎了吐给面试官，方便面试官快速消化🐼。