【前端面试】浏览器多标签页Tab之间如何...

前言

我们都知道浏览器是可以打开很多标签页的，如果每个标签页代表的是单独的一个网站，那么这些标签页之间肯定是不能通信的，如果能通信那估计我们都得凉凉。但是在很多情况下，浏览器中的很多标签页都属于某一个网站，而且这些标签页之间会使用一些相同的数据，这个时候我们就需要让这些标签页的数据都保持同步。

比如很多博客网站，点击文章列表通常是打开一个新的标签页进入文章详情页，那么如果我们在文章详情页点赞、评论等操作，而文章列表页也使用了这些数据，这个时候我们需要保持两边的数据一致，衍生出来就是详情页改了数据，需要让列表页知道。

总结来看：在某些情况下，实现多标签页之间通信是必要的！

1.localStorage 实现通信

借助 localStorage 实现标签页之间通信在实际项目中使用的很多，因为它操作简单，易于理解。如果你还不是早 localStorage 的用法，那你一定得恶补了。

localStorage 的特点：

- 同域共享存储空间
- 持久化将数据存储来浏览器
- 提供事件监听 localStorage 变化

这里我们需要重点关注同域共享，如果多个标签页跨域了，那么数据将无法共享。

代码演示：

我们新建两个页面 pageA 和 pageB，利用 localStorage 实现两个页面之间的通信。

示例代码：

pageA

```
// pageA.html
<body>
  <h1>pageA</h1>
</body>
<script>
  window.addEventListener("storage", (e) => {
    console.info("localStorage 发生变化：", e)
  })
```

```
</script>
```

pageB

```
// pageB.html
<body>
  <h1>pageB</h1>
  <button id="btnB">添加数据到 localStorage</button>
</body>
<script>
  let btnB = document.getElementById("btnB");
  let num = 0;
  btnB.addEventListener("click", () => {
    localStorage.setItem("num", num++)
  })
</script>
```

当我们点击 pageB 中的按钮时，会更改 localStorage 中的值。然后在 pageA 中的 storage 监听函数便会监听到 localStorage 发生变化。

pageA 输出结果：

可以看到在 pageA 中不仅可以拿到改变后的值，还可以拿到改变之前的值。通过这种方式，我们就可以将两个页面的数据进行同步了。

注意点：

- pageA 和 pageB 同源，即域名、端口、协议等都是相同的。
- 使用 storage 事件监听 localStorage 变化

当然，如果你只是需要两个页面之间数据共享，那么可以不使用 storage 监听方法，直接通过 localStorage.getItem()获取即可。

2.使用 websocket

websocket 是一种网络通讯协议。我们都知道在使用 HTTP 协议的时候，我们与服务端都是通过发请求的方式进行通讯的，而且这种通讯只能由客户端发起。websocket 协议就弥补了这一缺点，它是一个全双工通信的协议，意味着客户端和服务端可以互相通信，享受平等关系。

最简单列子就是聊天室，我们在聊天室里面可以收消息，也可以发消息，只要我们与服务端通过 websocket 建立好了连接。

websocket 特点：

- 保持连接状态，HTTP 协议是无状态连接，即请求完毕后就会关闭连接。
- 全双工通信，客户端和服务端平等对待，可以互相通信。
- 建立在 TCP 协议之上
- 没有同源共享策略，即可实现跨域共享

通过以上 websocket 的特点，我们再来思考如何利用 websocket 实现多标签页通信?

其实实现原理页比较简单，假如我们 pageA 和 pageB 都与服务器建立了 websocket 连接，那么连个页面都可以实时接收服务端发来的消息，也可以实时向服务端发送消息。如果 pageA 更改了数据，那么向服务端发送一条消息或数据，服务端在将这条消息或数据发送给 pageB 即可，这样就简单实现了两个标签页之间的通信。

原理有点类似于”中介“，我们可以通过中介来进行沟通。

代码演示：

我们先来搭建一个简单的 websocket 服务器，用于 pageA 和 pageB 的连接，新建 index.js 文件。

初始化命令：

```
npm init -y
npm install --save ws
运行命令：node index.js
```

代码如下：

index.js

```
// index.js
let WebSocketServer = require("ws").Server;
let wss = new WebSocketServer({ port: 3000 });

// 创建保存所有已连接到服务器的客户端对象的数组
let clients = [];

// 为服务器添加 connection 事件监听，当有客户端连接到服务端时，立刻将客户端对象保存进
wss.on("connection", function (client) {
  console.log("一个客户端连接到服务器");
  if (clients.indexOf(client) === -1) {
    clients.push(client);
    // 接收客户端发送的消息
    client.on("message", function (msg) {
      console.log("收到消息:" + msg);
      // 将消息发送给非自己的客户端
      for (let key of clients) {
```

```
        if (key != client) {
          key.send(msg.toString());
        }
      }
    });
  }
});

```

pageA

```
// pageA
<script>
  // 创建一个 websocket 连接
  var ws = new WebSocket('ws://localhost:3000/');
  // WebSocket 连接成功回调
  ws.onopen = function () {
    console.log("websocket 连接成功")
  }
  // 这里接受服务器端发过来的消息
  ws.onmessage = function (e) {
    console.log("服务端发送的消息", e.data)
  }
</script>
```

pageB

```
<script>
  let btnB = document.getElementById("btnB");
  let num = 0;
  btnB.addEventListener("click", () => {
    ws.send(`客户端 B 发送的消息:${num++}`);
  })
  // 创建一个 websocket 连接
  var ws = new WebSocket('ws://localhost:3000/');
  // WebSocket 连接成功回调
  ws.onopen = function () {
    console.log("websocket 连接成功")
  }
</script>
```

当我们点击 pageB 中的按钮时，会通过 websocket 向服务端发送一条消息，服务端接收到这条消息之后，会将消息转发给 pageA，这样 pageA 就得到了 pageB 传来的数据。

pageA 输出结果：

```
websocket连接成功
服务端发送的消息 客户端B发送的消息:0
服务端发送的消息 客户端B发送的消息:1
服务端发送的消息 客户端B发送的消息:2
服务端发送的消息 客户端B发送的消息:3
服务端发送的消息 客户端B发送的消息:4
服务端发送的消息 客户端B发送的消息:5
>
```

总体来说，原理很简单，只是需要了解 websocket。通常情况下，我们不建议使用 websocket 来进行多标签页通信，因为这回增加服务器的负担。

3.SharedWorker

我们都知道 JavaScript 是单线程的，单线程有好处也有坏处。为了弥补 JS 单线程的坏处，webWorker 随之被提出，它可以为 JS 创造多线程环境。如果还不了解 webWorker 的可以去官网初步了解一下。

sharedWorker 就是 webWorker 中的一种，它可以由所有同源页面共享，利用这个特性，我们就可以使用它来进行多标签页之前的通信。

sharedWorker 特点：

- 跨域不共享，即多个标签页不能跨域
- 使用 port 发送和接收消息
- 如果 url 相同，且是同一个 js，那么只会创建一个 sharedWorker，多个页面共享这个 sharedWorker

其实它和我们的 webSocket 实现多页面通讯的原理很类似，都是发送数据和接收数据这样的步骤，shardWorker 就好比我们的 webSocket 服务器。

代码演示：

新建一个 worker.js，编写代码。

代码如下：

```
// worker.js
const set = new Set()
onconnect = event => {
  const port = event.ports[0]
  set.add(port)

  // 接收信息
  port.onmessage = e => {
    // 广播信息
    set.forEach(p => {
      p.postMessage(e.data)
    })
  }
```

```
  // 发送信息
  port.postMessage("worker 广播信息")
}
```

pageA

```
<script>
  const worker = new SharedWorker('./worker.js')
  worker.port.onmessage = e => {
    console.info("pageA 收到消息", e.data)
  }
</script>
```

pageB

```
<script>
  const worker = new SharedWorker('./worker.js')
  let btnB = document.getElementById("btnB");
  let num = 0;
  btnB.addEventListener("click", () => {
    worker.port.postMessage(`客户端 B 发送的消息:${num++}`)
  })
</script>
```

上面的代码就是一个最简单的 sharedWorker 的应用，我们在 pageA 页面中初始化了 sharedWorker，并且设置了接收消息的监听函数，当 sharedWorker 初始化完成之后，pageA 便会接收到一条消息，如下图：

```
pageA收到消息 worker广播信息
>
```

然后我们在 pageB 中同样初始化了 sharedWorker 的示例，点击按钮广播消息，此时 pageA 便可以收到消息，是不是和 websocket 的原理很像啊。

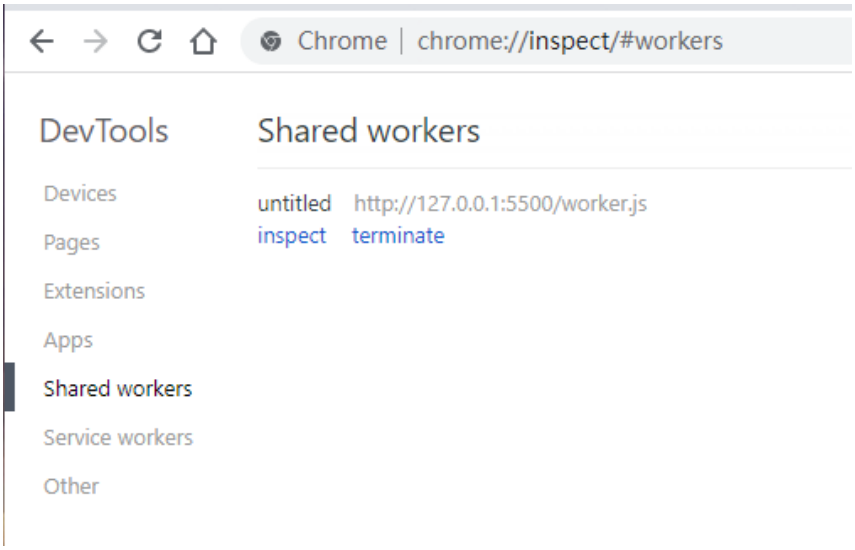
pageA 输出结果：

```
pageA收到消息 worker广播信息
pageA收到消息 客户端B发送的消息:1
pageA收到消息 客户端B发送的消息:2
pageA收到消息 客户端B发送的消息:3
pageA收到消息 客户端B发送的消息:4
>
```

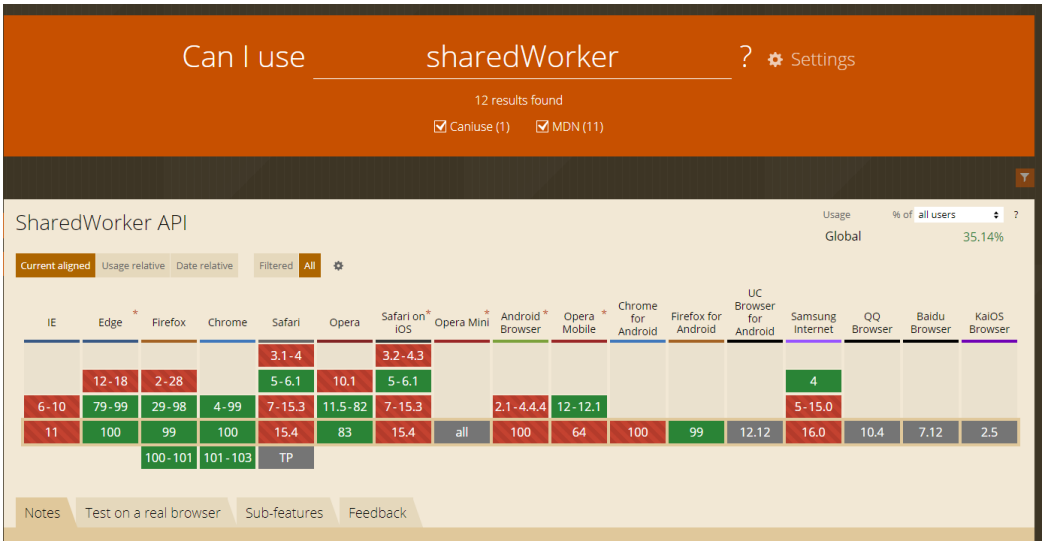
调试 sharedWorker：

我们如何查看当前是运行的哪个 sharedWorker 呢？可以在浏览时输入：chrome://inspect。

找到 sharedWorker 选项，就可以看到运行的 sharedWorker,如下图：



兼容性查看：



总结：

sharedWorker 的原理和 websocket 有点类似，都是广播和接收的原理，但是它也有一些缺点，比如调试不太方便、兼容性不太好。所以使用的时候一定要结合实际情况使用。

4.使用 cookie + setInterval

我们都知道 cookie 可以用来存储数据，而且它是同源共享的，借助它的这些特点，我们就可以利用 cookie 实现多页面的通讯。

cookie 特点：

- 跨域不共享
- 具有存储空间限制
- 请求会自动携带 cookie

示例代码：

pageA

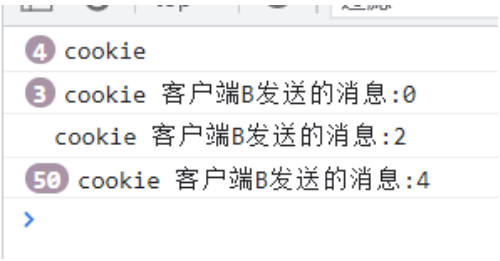
```
<script>
  setInterval(() => {
```

```
    //加入定时器，让函数每一秒就调用一次，实现页面刷新
    console.log("cookie",document.cookie)
  }, 1000);
</script>
```

pageB

```
<script>
  let btnB = document.getElementById("btnB");
  let num = 0;
  btnB.addEventListener("click", () => {
    document.cookie = `客户端 B 发送的消息:${num++}`
  })
</script>
```

输出结果：



这种方式实现的原理非常简单，就是在需要接收消息的页面不断轮询去查询 cookie，然后发送消息的页面将数据存储在 cookie 中，这样就实现了简单的数据共享。

总结

这里介绍了 4 中实现浏览器多标签页之前通讯的方法，它们优缺点也有优点，有的操作简单，有的已于理解等等，需要根据实际场景选择不一样的方法。

实现方式	优缺点
localStorage	优点： • 操作简单，易于理解。 缺点： • 存储大小限制 • 只能监听非己页面 • 跨域不共享 **总体来说较为推荐**
websocket	优点： • 理论上可是实现任何数据共享 • 跨域共享 缺点： • 需要服务端配合 • 增加服务器压力 • 上手不易 **总体不推荐**
sharedWorker	优点： • 理论上可以实现任何数据共享 • 性能较好 缺点： • 跨域不共享 • 调试不方便 • 兼容性不好 **总体推荐一般**
cookie	优点： • 兼容性好 • 易于上手和理解 缺点： • 有存储大小限制 • 轮询消耗性能 • 发请求会携带 cookie **总体不推荐**