

预习资料：

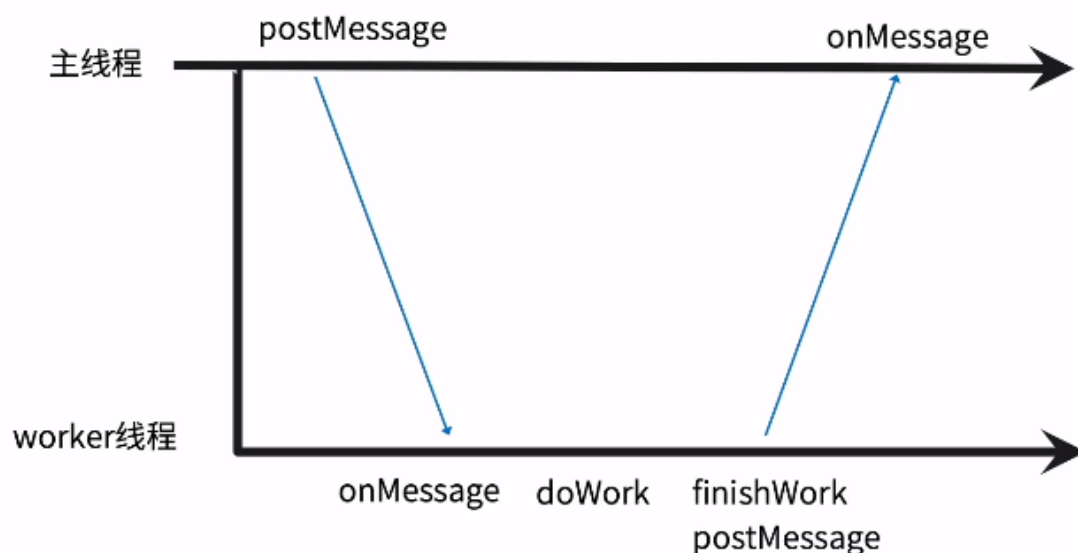
预习资料名称	链接	备注
Blob对象用法	https://developer.mozilla.org/zh-CN/docs/Web/API/Blob	嵌入式webworkers会用到Blob，在看视频前或者看完视频后看本篇资料都可以
JavaScript中的Blob对象	https://juejin.im/entry/5937c98eac502e0068cf31ae	掘金上比较基础的一篇文章，在看视频前或者看完视频后看本篇资料都可以，也可以自行搜索下Blob的扩展用法
webWorker基本用法	http://www.ruanyifeng.com/blog/2018/07/web-worker.html	

1. webworker介绍

webworker是什么？说白了就是一个webApi，就是浏览器提供一个js运行环境的能力，这个环境是分配一个线程来实现的，它是一个独立于主线程的后台线程，我们可以通过webwork运行多个脚本操作。

我们思考一下，浏览器为什么要提供这种能力呢？主要是出于性能考虑。我们打个更加生动的比喻，我们可以把主线程比喻成一个店主，当店里不忙的时候，店主还可以应付，但是店里很忙的时候就需要的再雇一个店员来帮忙干一些其他的脏活累活...浏览器也是如此，主线程主要用来渲染界面和js执行。其他的繁琐，大量的计算可以新开一个线程去处理，这样也能够有更好的体验。

主线程和worker线程如何通信？



主线程通过一个postMessage发送消息给worker线程，分配任务给worker线程，worker线程有通过一个onMessage接收消息，并进行dowork相应的任务，完成finishWork后，也有一个postMessage方法发送消息个主线程，告诉主线程任务完成。然后主线程也有一个onMessage方法接收worker发送过来的消息。worker任务已经完成，主线程只关注结果，如何使用这个结果即可。

接下来我们来看一段代码：

代码示例：

```
worker.js
1 function fibonacci(n) {
2   if (n == 1 || n == 2) {
3     return 1;
4   }
5   return fibonacci(n - 2) + fibonacci(n - 1);
6 }
7 postMessage(fibonacci(40));
8 onmessage = function (e) {
9   console.log(e);
10 };
```

```
webworker.js
1 var worker = new Worker("worker.js");
2
3 worker.onmessage = function (e) {
4   console.log("worker通知的message", e);
5   worker.postMessage("message收到了");
6 };
7
```

webworker.html

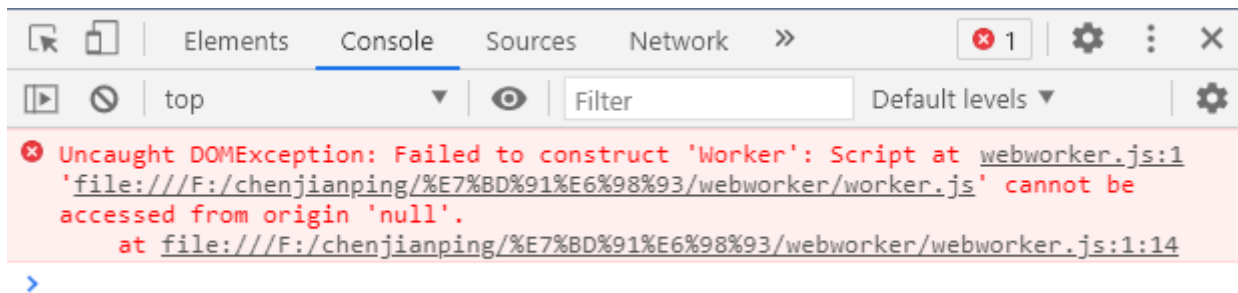
```
1 <!doctype html>
2 <html>
3
4 <head>
5     <title>web workers example</title>
6 </head>
7
8 <body>
9     <script src='./webworker.js'></script>
10 </body>
11
12 </html>
```

- 第一块worker.js是worker线程里面执行的js，它计算了一个斐波那契数列，然后执行了一下postMessage方法，传入了计算40个斐波那契数列的任务，然后再onmessage的时候把信息打印出来
- 第二块webworker.js是浏览器主线程执行的js，结合第三块webworker.html来看，webworker.html就是一个普通的html文件，文件引入了webworker.js也就是第二块代码，这块代码中先创建了一个worker实例，new Worker(xxx)里面传入的就是第一块worker.js，然后监听了worker实例上的onmessage，监听到message时打印worker通知的message是什么，并执行worker的postMessage方法，告诉worker信息收到了

我们在浏览器中看一下上面代码的运行过程

先演示一个错误运行示范：

首先进入文件夹直接双击文件打开webworker.html，可以看到会报错

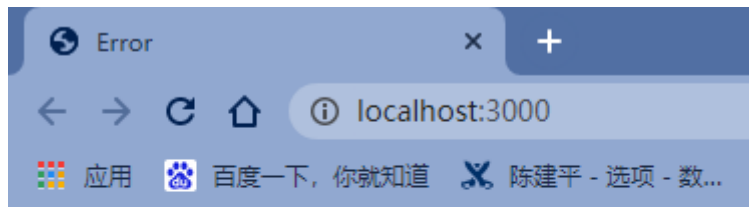


这个报错的原因是由于worker的使用是有限制的，不是在本地直接跑一个静态文件就可以使用它，因为new Worker('worker.js')中的参数路径不能通过本地文件去取，如果直接访问这个静态页面，它的协议还是file协议，这样是不行的。

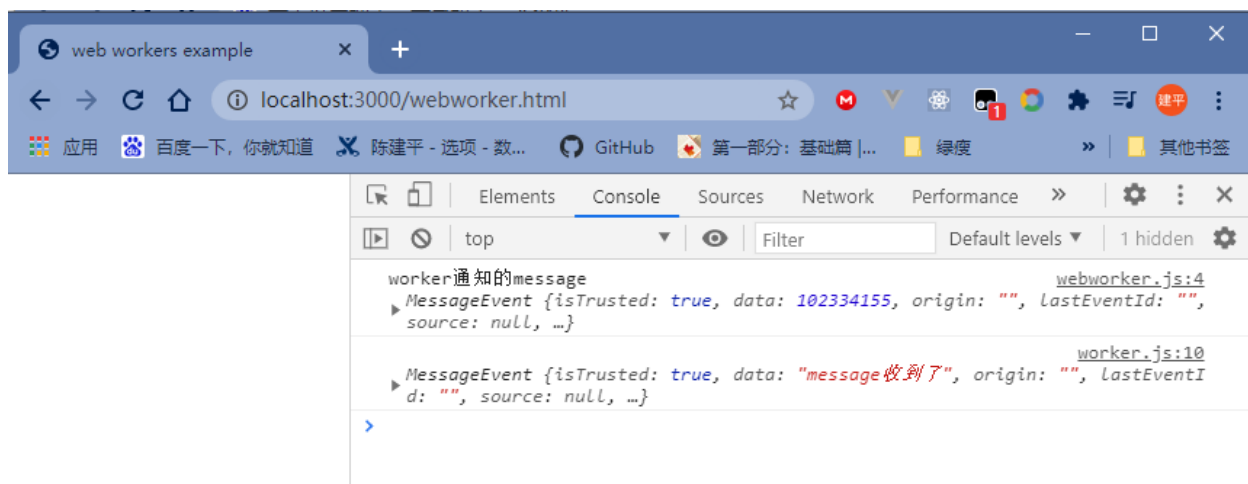
正确运行示范：

如果我们要运行这段代码，需要启动一个server，启动静态资源的server有很多，比如 anywhere、http-server、browser-sync等等，这里使用browser-sync举例。（在IDE编辑器中运行页面到浏览器也可以正常访问）

1. 首先打开命令行工具，cd进入文件夹，然后输入browser-sync start -s，确认后看到启动了一个网页

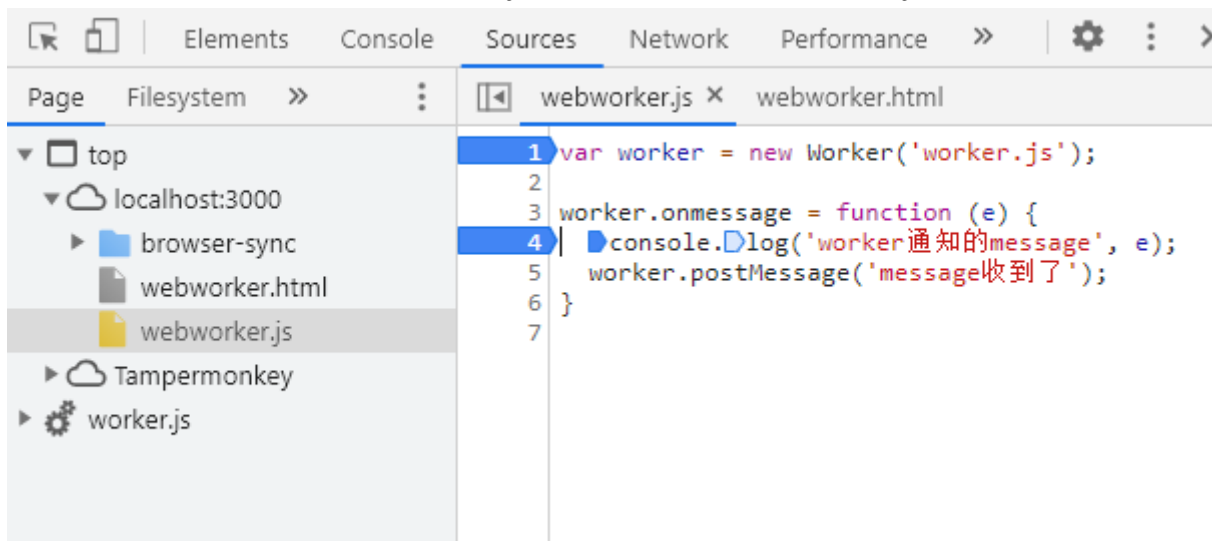


2. 启动之后修改url地址直接访问这个路径，可以看到已经正确执行了worker

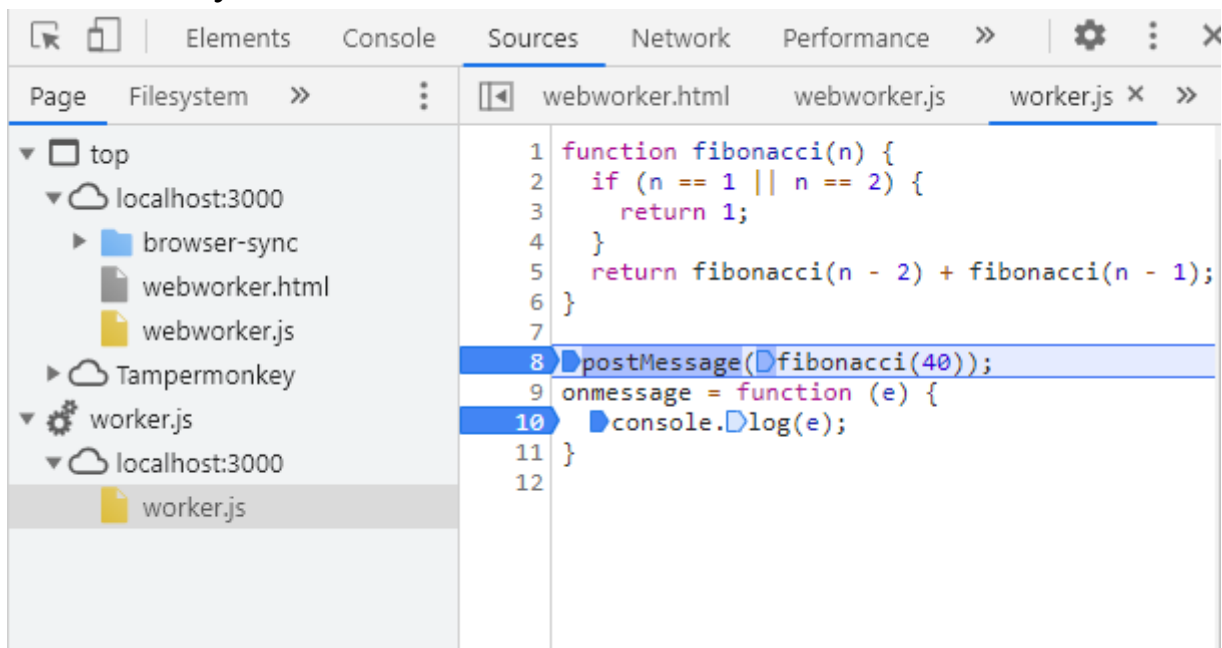


3. 我们在代码中加上断点来看这段代码的执行过程

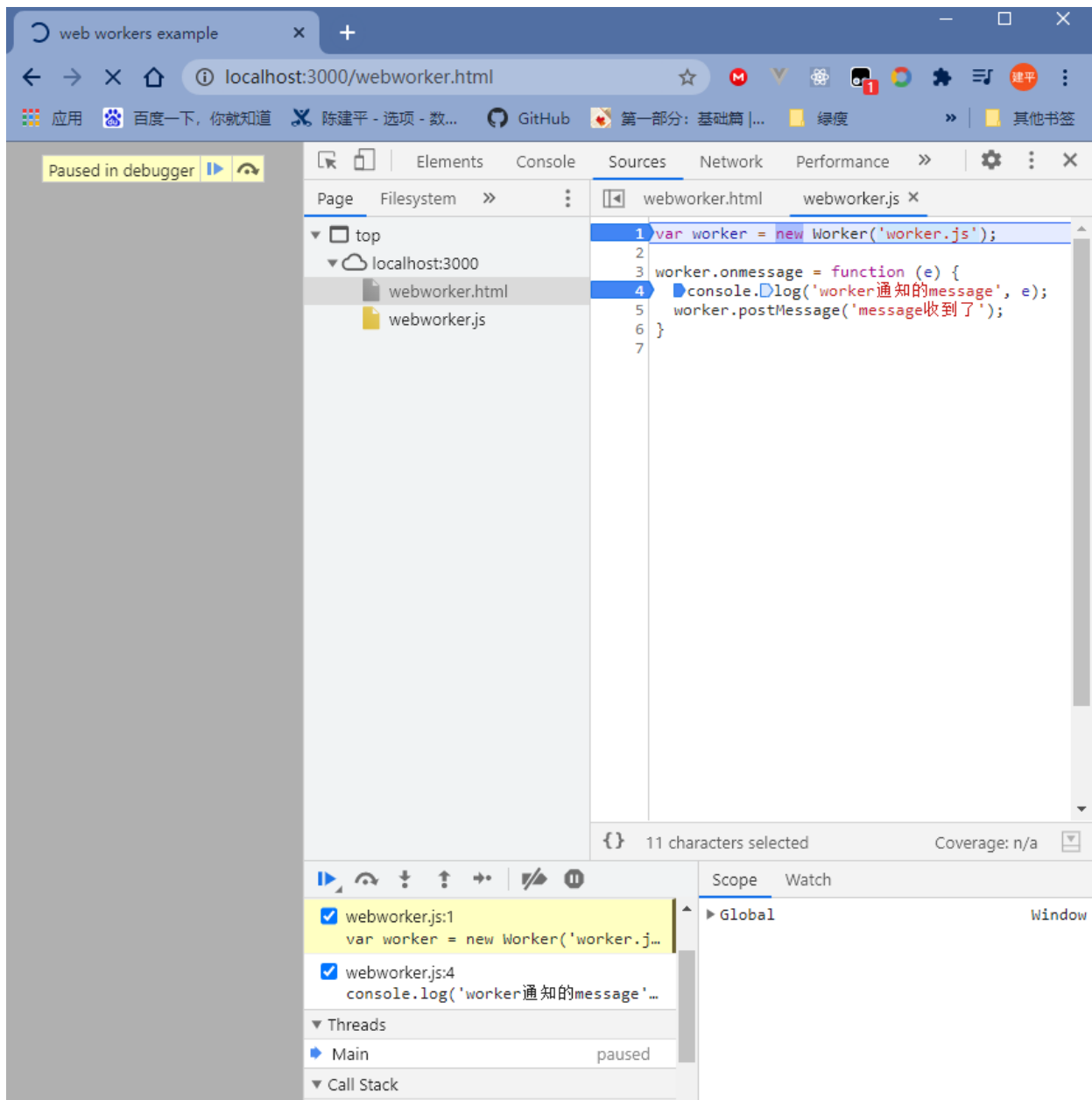
- 首先会进入到webworker.js里面，因为webworker.js就是我们直接在webworker.html直接引入的js，然后我们给webworker.js加上断点如图所示



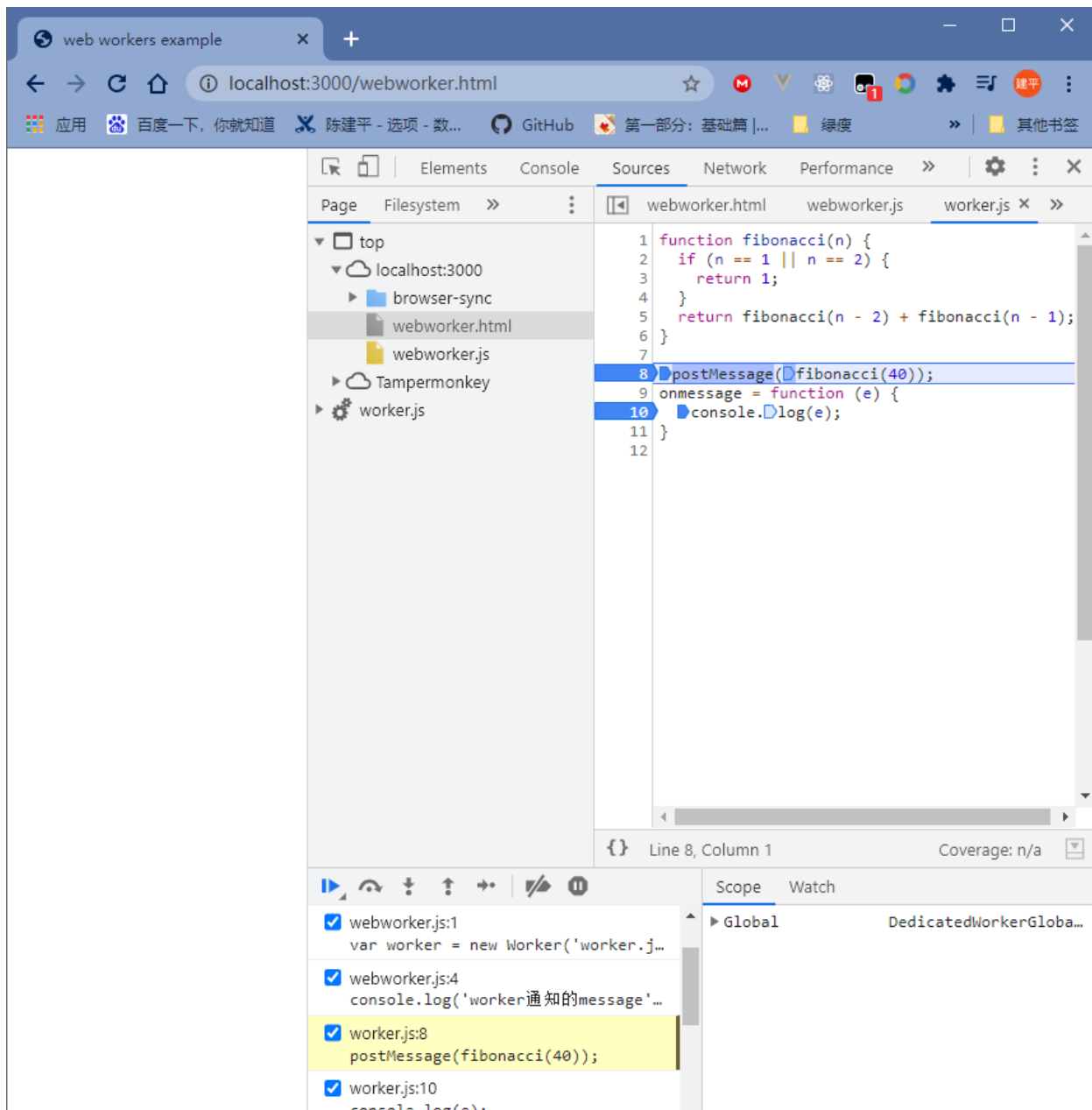
给worker.js加上断点如图所示



- 然后刷新页面，可以看到它首先进入到webworker.js，执行new Worker('worker.js')

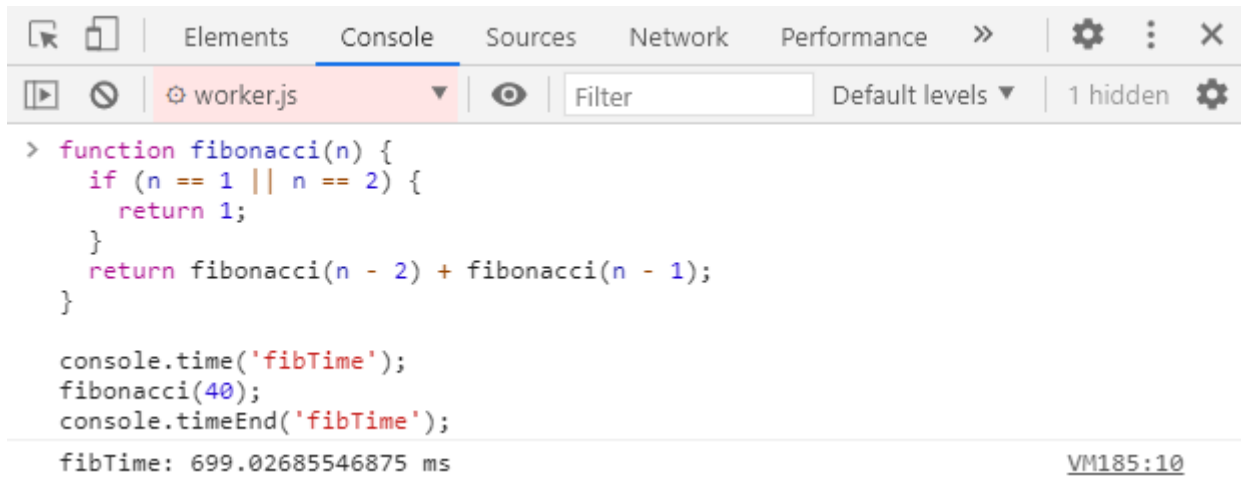


- 点击下一步，可以看到它执行完new Worker之后直接进入到了worker.js里面去执行postMessage



`postMessage`执行的是一个计算40项斐波那契数列，斐波那契数列是个比较耗时的计算，怎么证明它耗时呢？

我们通过`console.time`和`console.timeEnd`（这是一个成对出现的标记）来统计一块代码的执行时间



The screenshot shows a web browser's developer console with the 'Console' tab selected. The file 'worker.js' is loaded. The code being executed is a Fibonacci function that calculates the 40th Fibonacci number. The function is defined as follows:

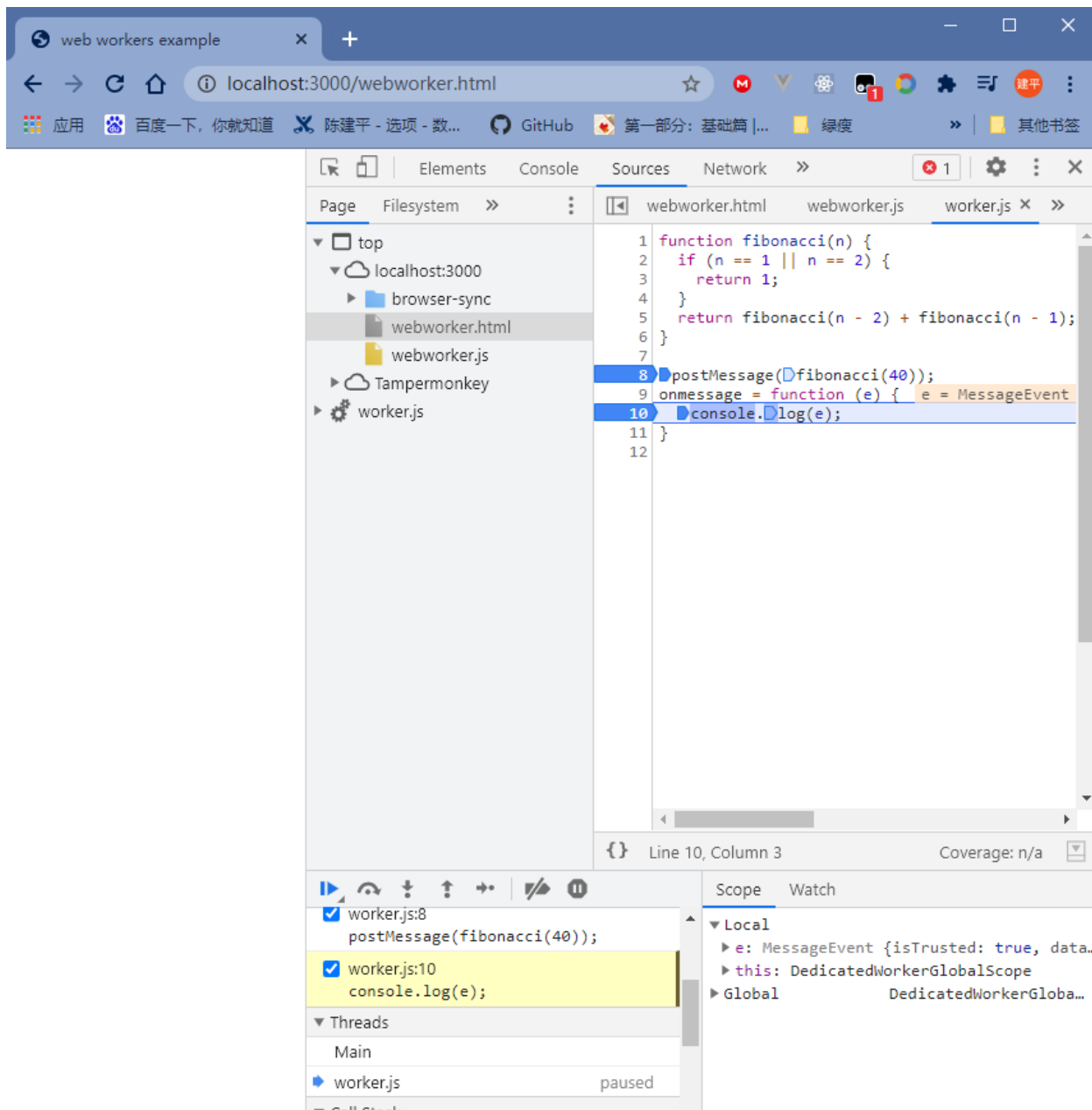
```
> function fibonacci(n) {  
  if (n == 1 || n == 2) {  
    return 1;  
  }  
  return fibonacci(n - 2) + fibonacci(n - 1);  
}  
  
console.time('fibTime');  
fibonacci(40);  
console.timeEnd('fibTime');
```

The console output shows the execution time for 'fibTime' as 699.02685546875 ms. The console also indicates that there is 1 hidden log entry.

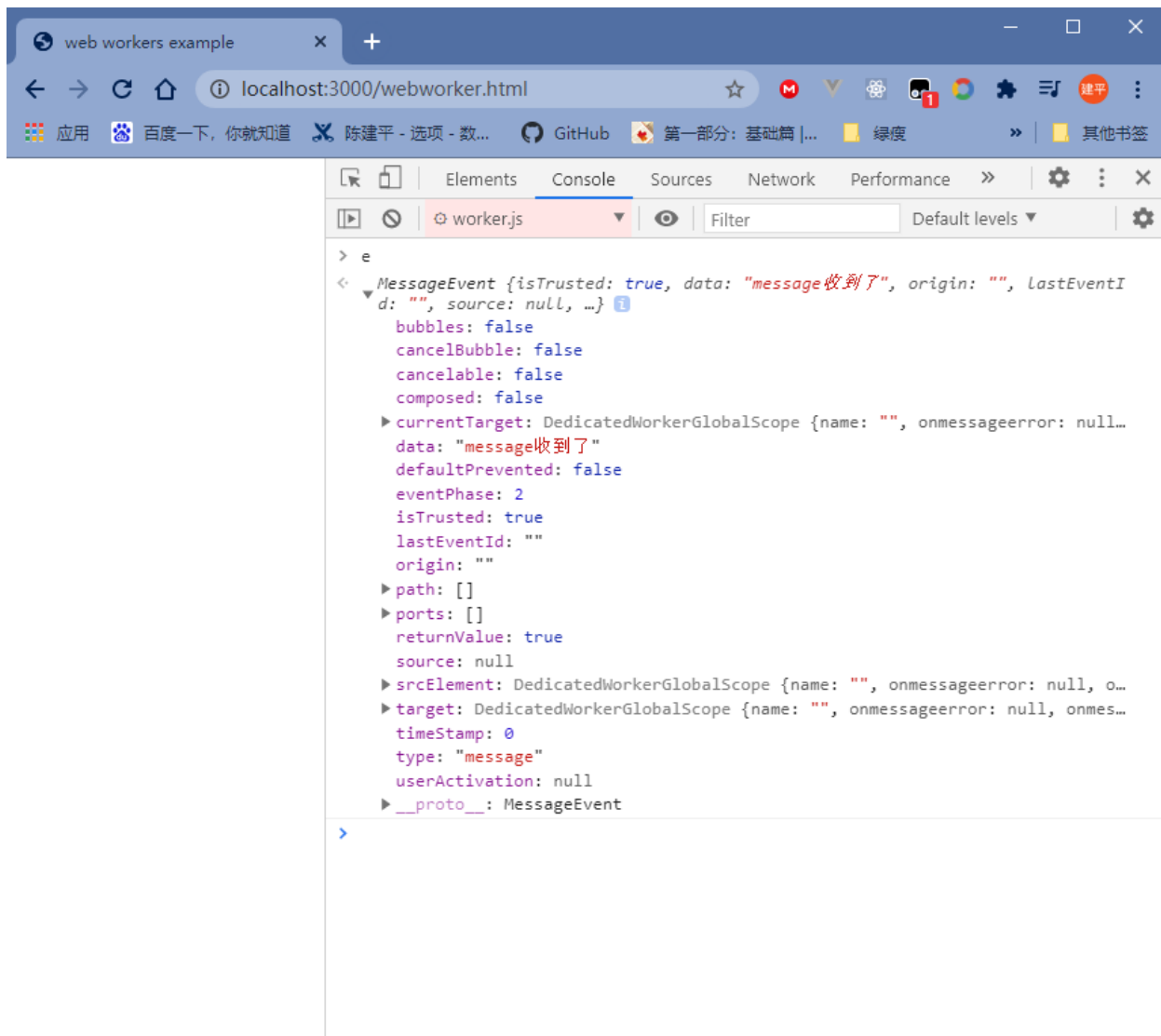
大概用了699毫秒（这个时间跟自己电脑的性能有关系），所以我们得出一个结论：像这种耗时的运算是可以放在worker线程里面的。看完了耗时运算，继续看worker.js中的代码

postMessage的作用是向主线程发送消息，然后把斐波那契数列计算结果发送出去。

- 点击下一步，进入了worker.js，worker线程的onmessage就收到了消息



我们切换到console输出一下e验证一下是不是收到了主线程的消息



可以看到接收到了主线程的信息，e是消息体，e.data就是设置的消息内容

以上就是主线程和worker线程的通信过程，我们可以看出来它就是通过来回的 `postMessage`和`onMessage`来实现通信的。

在学习异步的时候我们说过JavaScript为什么要设计成单线程，原因是由于多个线程操作dom会引起混乱，既然js可以在webworker里面运行，会不会造成这种问题呢？

答案是不会，因为worker的能力是受限制的。

webworker的限制：

- 与主线程脚本同源（就是说在主线程下运行的worker脚本必须与主线程的脚本同源，不可以跨源运行,必须是同源）

- 与主线程的上下文不同（可以将worker线程下的全局对象跟主线程下的全局对象打印出来验证一下）

与主线程的上下文不同会造成什么问题？

1. 无法操作DOM。因为操作DOM是通过一些Web API来操作的，在worker上下文中没有这些Web API，所以它无法使用像document、window这些对象，但它可以使用像navigator和location这些不会引起页面混乱的API

2. 不能执行alert。因为它能力受限，所以它的上下文中没有这个API

- 不能读取本地文件（服务器上的文件一般是http或https开头，本地的文件一般是file://开头，也就是说它的加载文件要来源于网络，这是一个使用约定）

我们把上面的代码再运行验证一下

1、首先刷新页面，此时断点到webworker.js中执行new Worker，然后我们切换console打印this

```
> this
Window {window: Window, self: Window, document: document, name: "", Location: Location, ...}
  ▶ alert: f alert()
  ▶ atob: f atob()
  ▶ blur: f blur()
  ▶ btoa: f btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
  ▶ cancelIdleCallback: f cancelIdleCallback()
  ▶ captureEvents: f captureEvents()
  ▶ chrome: {loadTimes: f, csi: f}
  ▶ clearInterval: f clearInterval()
  ▶ clearTimeout: f clearTimeout()
  ▶ clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor: ...}
  ▶ close: f close()
  ▶ closed: false
  ▶ confirm: f confirm()
  ▶ cookieStore: CookieStore {onchange: null}
  ▶ createImageBitmap: f createImageBitmap()
  ▶ crossOriginIsolated: false
  ▶ crypto: Crypto {subtle: SubtleCrypto}
  ▶ customElements: CustomElementRegistry {}
```

2、点击下一步断点进入worker.js，然后切换console打印this

```

> this
< ▼ DedicatedWorkerGlobalScope {name: "", onmessage: null, onmessageerror: null, cancelAnimationFrame: f, close: f, ...}
  ► caches: CacheStorage {}
  ► cancelAnimationFrame: f cancelAnimationFrame()
  ▼ close: f close()
    arguments: [Exception: TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or.
    caller: [Exception: TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or th.
    length: 0
    name: "close"
    ► __proto__: f ()
    ► [[Scopes]]: Scopes[0]
    crossOriginIsolated: false
    ► crypto: Crypto {subtle: SubtleCrypto}
    ► fibonacci: f fibonacci(n)
    ► fonts: FontFaceSet {onloading: null, onloadingdone: null, onloadingerror: null, ready: Promise, status: "loaded", ...}
    ► indexedDB: IDBFactory {}
    isSecureContext: true
    ► location: WorkerLocation {origin: "http://localhost:63342", protocol: "http:", host: "localhost:63342", hostname: "localhost", por.
    name: ""
    ► navigator: WorkerNavigator {hardwareConcurrency: 6, appCodeName: "Mozilla", appName: "Netscape", appVersion: "5.0 (Windows NT 10.0
    onerror: null
    onlanguagechange: null
    onmessage: null
    onmessageerror: null
    onrejectionhandled: null
    onunhandledrejection: null
    origin: "http://localhost:63342"
    ► performance: Performance {timeOrigin: 1616470703552.4082, onresourcetimingbufferfull: null}
    ► postMessage: f postMessage()
    ► requestAnimationFrame: f requestAnimationFrame()
    ► self: DedicatedWorkerGlobalScope {name: "", onmessage: null, onmessageerror: null, cancelAnimationFrame: f, close: f, ...}
    ► trustedTypes: TrustedTypePolicyFactory {emptyHTML: , emptyScript: , defaultPolicy: null}
    ► webkitRequestFileSystem: f webkitRequestFileSystem()
    ► webkitRequestFileSystemSync: f webkitRequestFileSystemSync()
    ► webkitResolveLocalFileSystemSyncURL: f webkitResolveLocalFileSystemSyncURL()
    ► webkitResolveLocalFileSystemURL: f webkitResolveLocalFileSystemURL()
    Infinity: Infinity
    ► AbortController: f AbortController()
    ► AbortSignal: f AbortSignal()
    ► AggregateError: f AggregateError()
    ► Array: f Array()
    ► ArrayBuffer: f ArrayBuffer()
    ► Atomics: Atomics {load: f, store: f, add: f, sub: f, and: f, ...}
    ► BackgroundFetchManager: f BackgroundFetchManager()

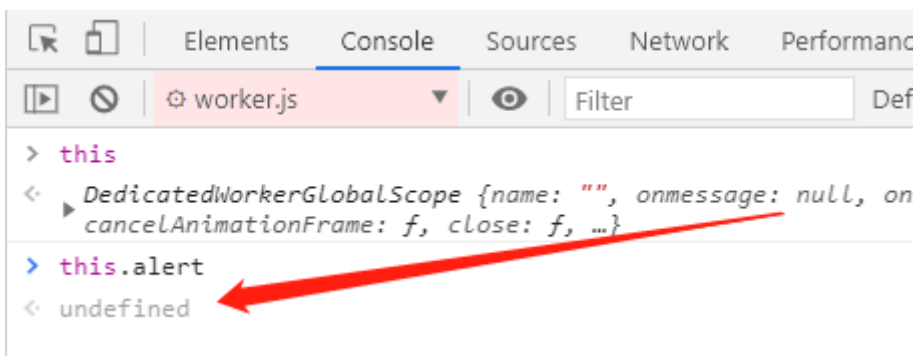
```

可以看到worker线程的全局对象是DedicatedWorkerGlobalScope，它也有很多可以使用的属性和方法，它属于window对象的子集，window对象上有一些属性和方法这个对象上是没有的，因为它的能力是受限的。比如操作dom的document就是没有的，alert也是没有的。

```

> this
< ► DedicatedWorkerGlobalScope {name: "", onmessage: null, onmessageerror: null, cancelAnimationFrame: f, close: f, ...}
> this.document
< undefined
> |

```



The screenshot shows a web browser's developer console with the 'Console' tab selected. The file 'worker.js' is loaded. The console output shows the result of the command 'this.alert', which is 'undefined'. A red arrow points from the text 'alert也是没有的' (alert is also not available) to the 'undefined' result.

```

> this
< ► DedicatedWorkerGlobalScope {name: "", onmessage: null, onmessageerror: null, cancelAnimationFrame: f, close: f, ...}
> this.alert
< undefined

```

通过上面的学习我们了解到webworker的功能是受限制的，new Worker里面的文件路径是不能从本地文件里读取的，但我们现在的写法很多都是模块化的写法，我们很有可能需要用到require.js，那么我们有没有更简单的方式在项目中使用webworker呢？

嵌入式worker就能帮助我们更简单的使用webworker。

什么是嵌入式worker？嵌入式worker的原理是什么？

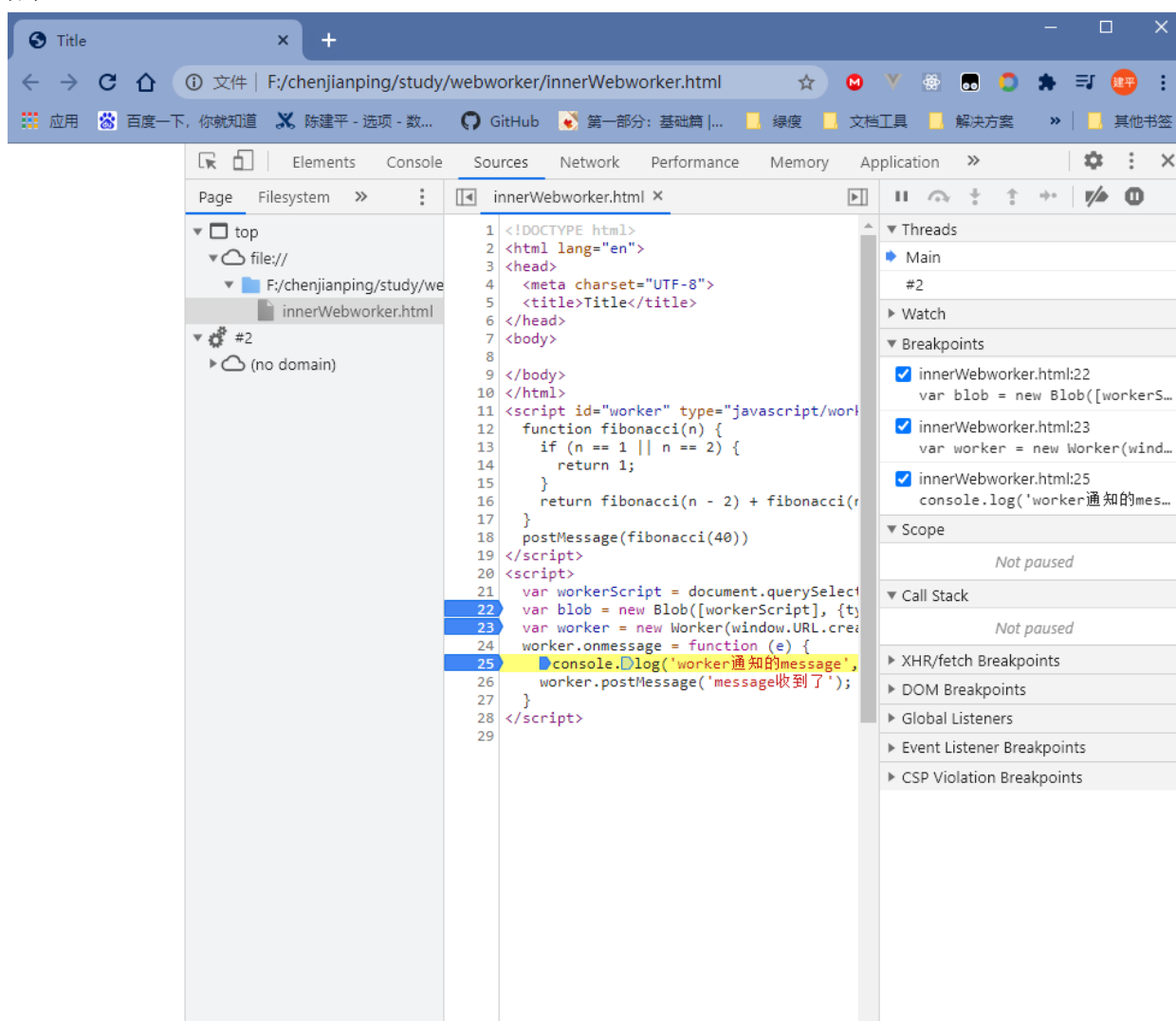
代码示例：

```
1 <script id="worker" type="javascript/worker">
2     function fibonacci(n) {
3         if (n == 1 || n == 2) {
4             return 1;
5         }
6         return fibonacci(n - 2) + fibonacci(n - 1);
7     }
8     postMessage(fibonacci(40));
9 </script>
10 <script>
11     var workerScript = document.querySelector('#worker').textContent;
12     var blob = new Blob([workerScript], { type: 'text/javascript' });
13     var worker = new Worker(window.URL.createObjectURL(blob));
14     worker.onmessage = function (e) {
15         console.log("worker通知的message", e);
16         worker.postMessage("message收到了");
17     };
18 </script>
```

上面的代码示例中，先看id为worker的script，注意这个script的type不是text/javascript类型，而是javascript/worker类型，这个类型是不会执行的，相当于我们只在页面上添加了这块代码，但是这个代码对页面是无感知的，加这块代码的原因是我们后面要取这个script里面的代码内容。

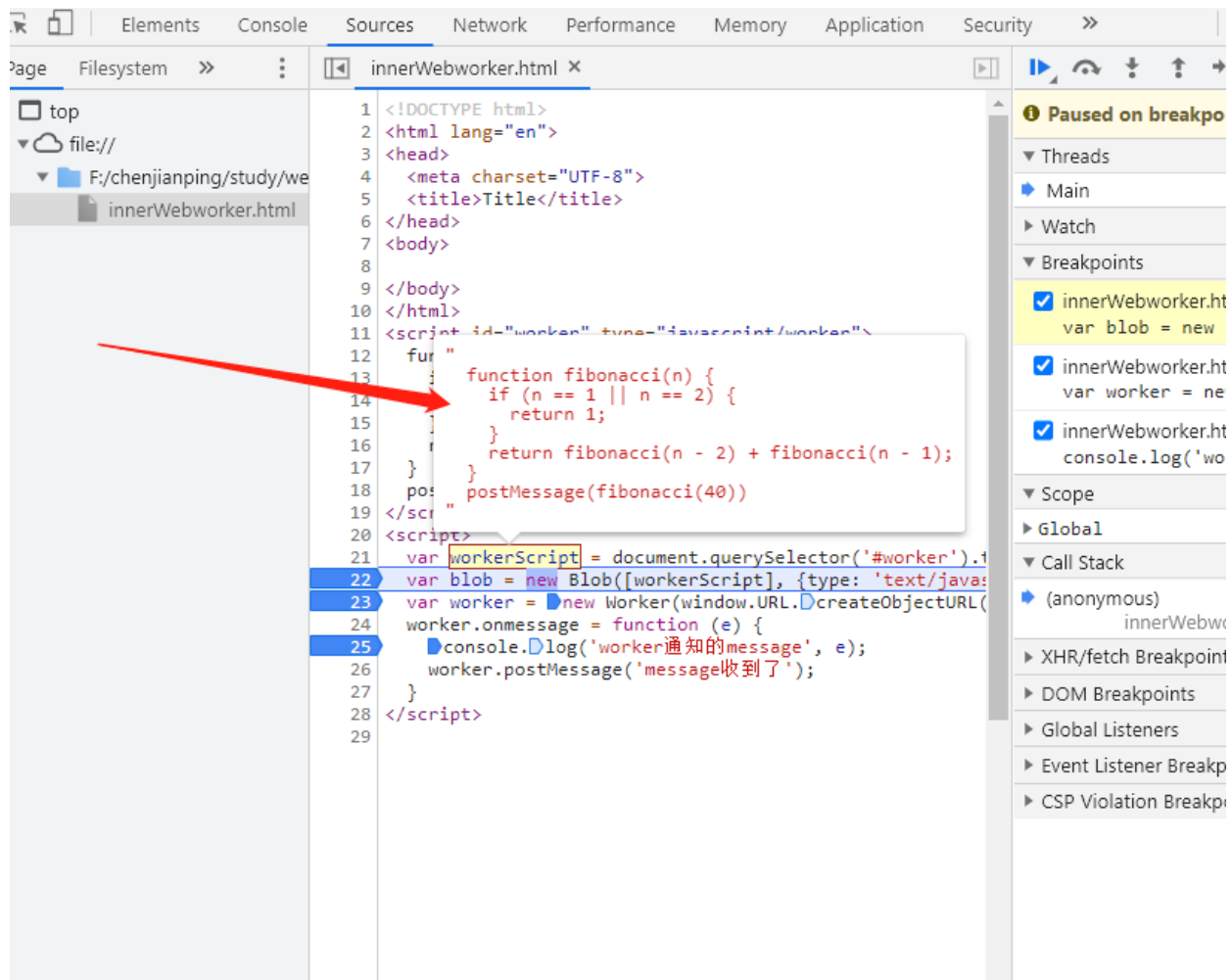
后一个script中，第11行代码就是获取id为worker的script中的代码字符串，然后通过Blob这个构造函数，把workerScript用数组包装一下放进去并加上一个类型，生成了一个blob类型的实例对象，这个对象通过window.URL.createObjectURL方法生成了一个url，然后把url放入worker的构造函数中实例化出一个worker对象，这样就可以使用worker了。

我们直接在本地文件双击运行演示代码，可以看到正常打印了通知，然后在控制台中打上断点

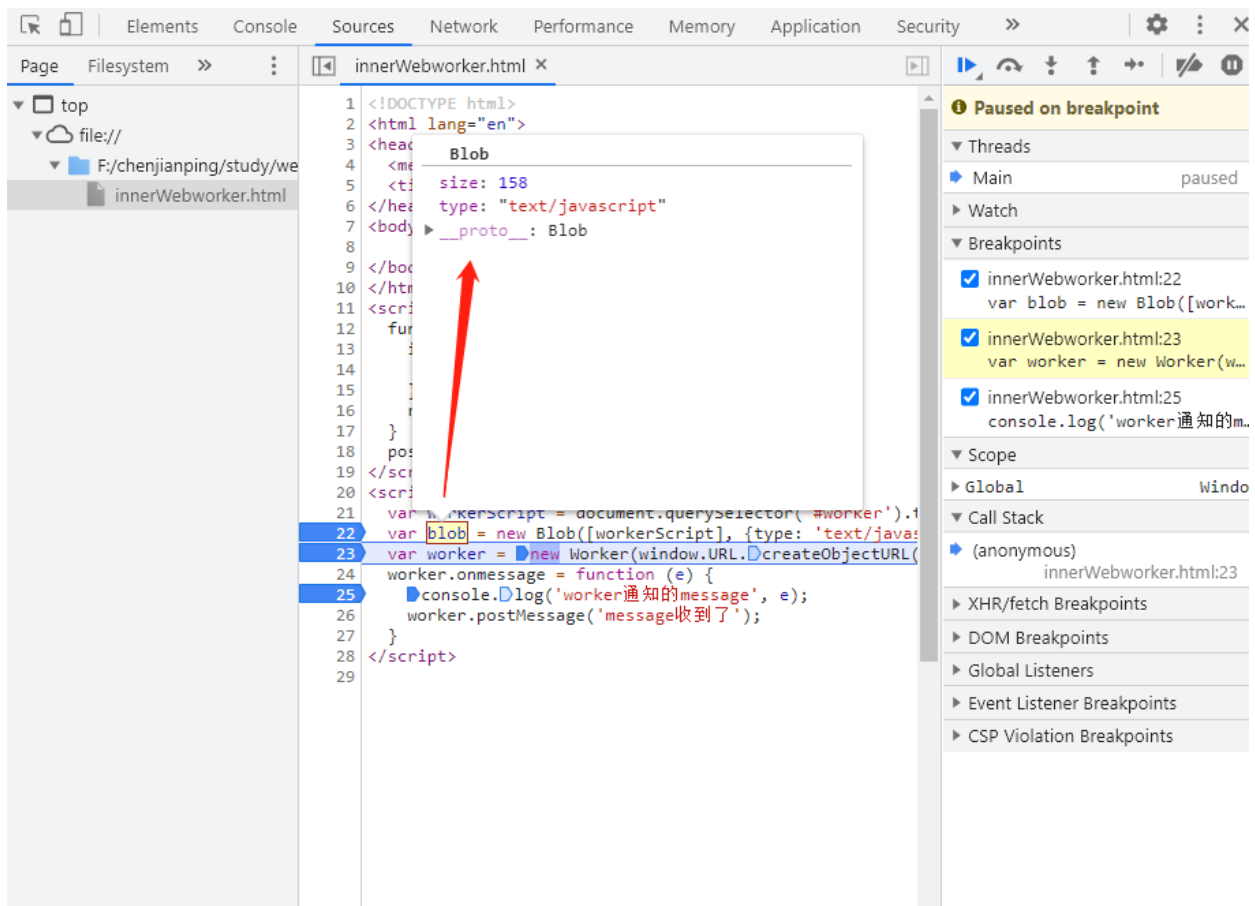


嵌入式worker是直接嵌入到当前页面中，没有通过http请求去读文件，所以它可以不用启动server也可以使用。然后我们看一下每一步的分析

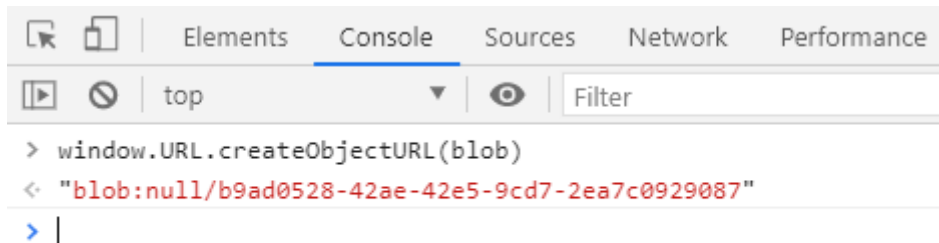
首先workerScript就是取到的worker代码字符串



然后将这个字符串传入Blob构造函数里面，这一步生成的是一个blob对象



我们把blob对象传入createObjectURL中看看这一步生成的是什么

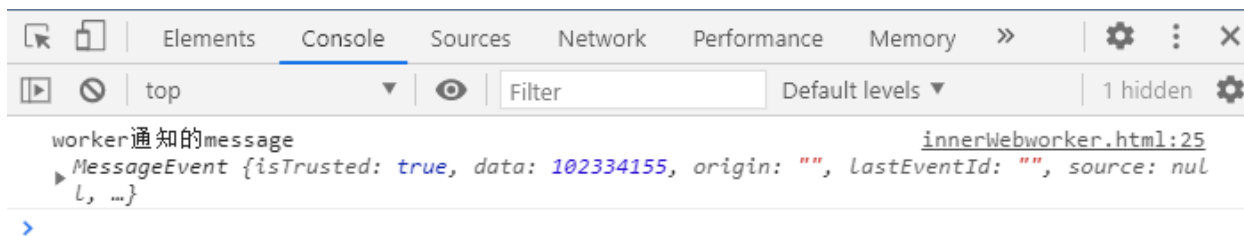


这一步生成的是一个类似于url的东西，这一串字符串可以直接在浏览器打开访问，它对应的就是workerScript里的代码字符串



```
function fibonacci(n) {
  if (n == 1 || n == 2) {
    return 1;
  }
  return fibonacci(n - 2) + fibonacci(n - 1);
}
postMessage(fibonacci(40))
```

继续进入下一步断点，可以看到正常输出了通知，说明嵌入的worker代码已经执行了，也验证了嵌入式worker可以正常工作



在项目中如何使用worker呢？

如果使用上面这种方式嵌入式worker是十分不优雅的，因为嵌入的worker代码可能很长，如果嵌入到页面，会让页面变得复杂，并且它的script类型是worker，也没有高亮显示，写代码的时候不会智能提示。

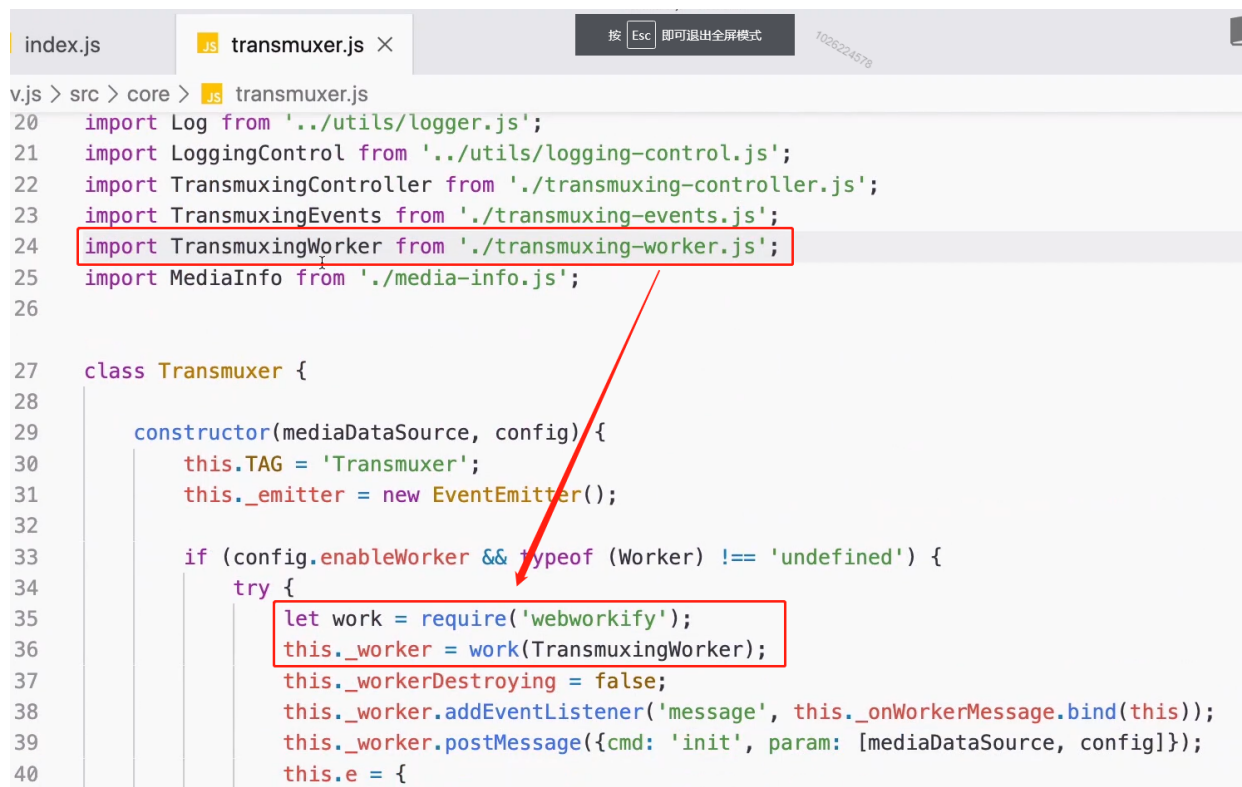
webworkify这个库就是用来解决这个问题的，它的实现原理与上面的嵌入式代码类似，也是通过createObjectURL来实现的

webworkify源码重要部分节选：

```
72   var URL = window.URL || window.webkitURL || window.mozURL || window.msURL;
73
74   var blob = new Blob([src], { type: 'text/javascript' });
75   if (options && options.bare) { return blob; }
76   var workerUrl = URL.createObjectURL(blob);
77   var worker = new Worker(workerUrl);
78   worker.objectURL = workerUrl;
79   return worker;
```

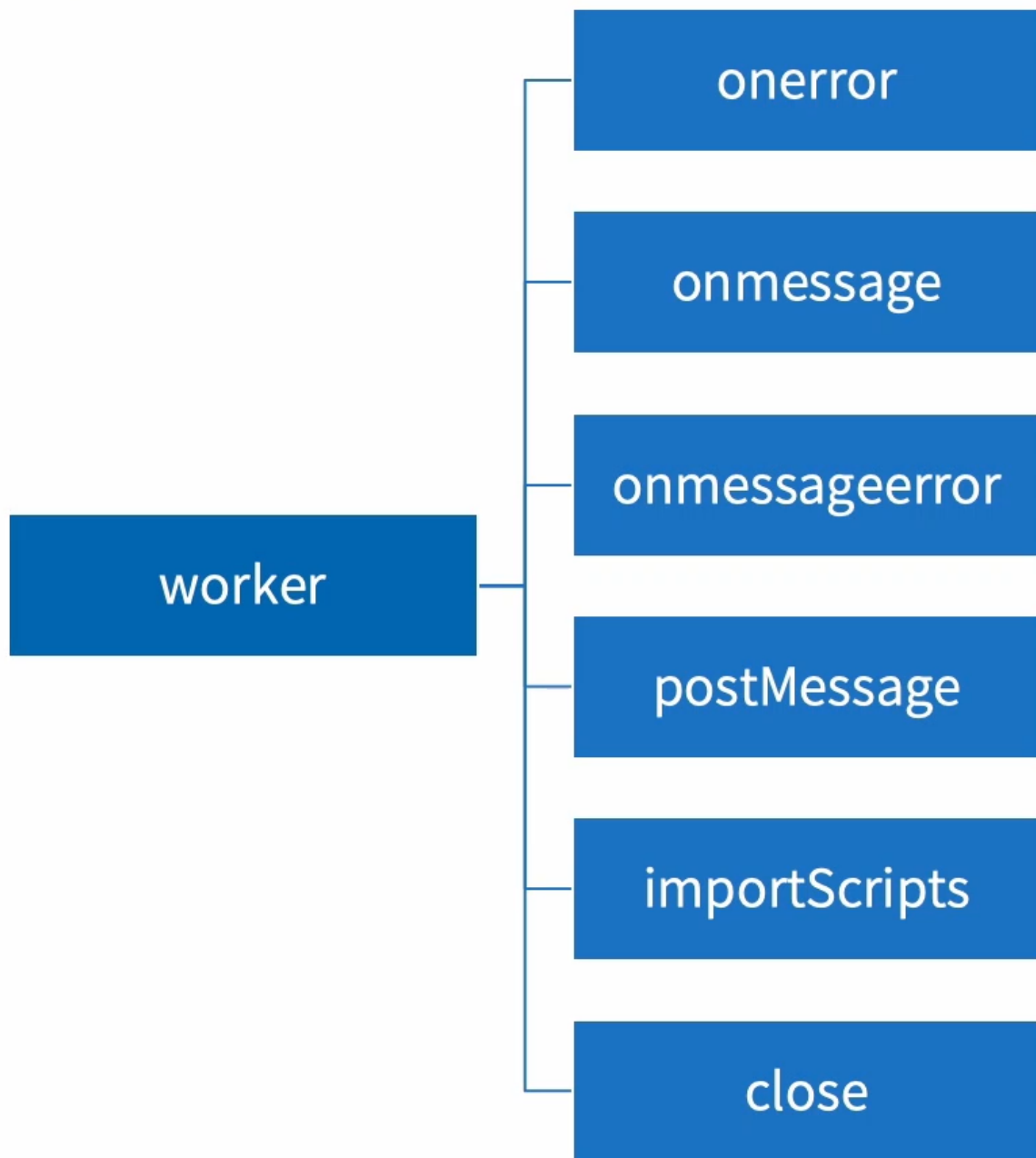
代码是不是很熟悉，跟上面的嵌入式worker示例代码一样，webworkify就相当于给嵌入式worker做了一层封装，以便我们更好的使用，用法如下：

通过require引入webworkify，然后将需要在worker中运行的js放进去即可将worker引用在我们的工程下面了，代码示例：



```
index.js  transmuxer.js × 按 Esc 即可退出全屏模式 1026224578
v.js > src > core > transmuxer.js
20 import Log from '../utils/logger.js';
21 import LoggingControl from '../utils/logging-control.js';
22 import TransmuxingController from './transmuxing-controller.js';
23 import TransmuxingEvents from './transmuxing-events.js';
24 import TransmuxingWorker from './transmuxing-worker.js';
25 import MediaInfo from './media-info.js';
26
27 class Transmuxer {
28
29   constructor(mediaDataSource, config) {
30     this.TAG = 'Transmuxer';
31     this._emitter = new EventEmitter();
32
33     if (config.enableWorker && typeof (Worker) !== 'undefined') {
34       try {
35         let work = require('webworkify');
36         this._worker = work(TransmuxingWorker);
37         this._workerDestroying = false;
38         this._worker.addEventListener('message', this._onWorkerMessage.bind(this));
39         this._worker.postMessage({cmd: 'init', param: [mediaDataSource, config]});
40         this.e = {
```

worker常用的方法如下：



二、Web Workers场景

要了解一个东西的使用场景，就要知道它解决的痛点是什么？

- js执行复杂运算的时候阻塞了页面渲染

webworker其实就是分担主线程的工作，将复杂的运算放到worker线程，所以它的使用场景有如下几点

使用场景：

- 复杂运算

复杂运算在前端的场景比较少，目前为止webworker的使用也不是特别的广，前端基本上是负责显示数据，在前端直接做运算的情况是非常少的。

- 渲染优化

渲染这一块需要优化的场景也是不多的，因为浏览器的渲染性能还是比较好的，如果DOM节点不是非常非常多的话，浏览器基本上不会存在渲染上的瓶颈。什么时候可能会存在渲染问题呢？比如canvas，有过canvas画图经验的应该知道，有时候画图会出现卡顿的情况，尤其是动画的时候，我们有一种优化手段就是利用canvas的离线API叫offline-canvas，通过offline-canvas结合webworker来做canvas渲染优化。

- 流媒体数据处理

它是一个小众的应用场景，下面会介绍。

如何知道哪些项目用到了webworker？

知道了webworker的使用场景后，我们不知道在项目中它具体用在了哪里，有一个办法就是，我们知道webworkify包装了一个嵌入式的webworker，很多工程如果用了webworker会依赖webworkify这个包，所以我们只要查到这个包被哪些项目依赖了就大概知道了webworker都用在了哪些项目里面。

然后打开npm的网站搜索webworkify，点击Dependents就可以看到有哪些项目依赖了webworkify，然后进入and more查看详细列表

Wondering what's next for npm? [Check out our public roadmap!](#)

webworkify

1.5.0 • Public • Published 3 years ago

Readme

Explore BETA

0 Dependencies

139 Dependents

13 Versions

webworkify

launch a web worker that can require() in the browser with browserify

example

First, a `main.js` file will launch the `worker.js` and print its output:

```
var work = require('webworkify');

var w = work(require('./worker.js'));
w.addEventListener('message', function (ev) {
  console.log(ev.data);
});

w.postMessage(4); // send the worker a message
```

then `worker.js` can `require()` modules of its own. The worker function lives inside of the

Install

```
> npm i webworkify
```

Weekly Downloads

52,861

Version	License
1.5.0	MIT

Homepage

github.com/substack/webworkify

Repository

github.com/substack/webworkify

Last publish

3 years ago

webworkify

1.5.0 • Public • Published 3 years ago

Readme

Explore BETA

0 Dependencies

139 Dependents

13 Versions

Dependents (139)

hd-flv @teleport-js/teleport v-flv @ngageoint/geopackage @wolfv/roslib
@dusion/flv.js hky-flv.js @minghan9456/mumble-client-codecs-browser nkn-sdk
flv.hlm.js flv.js.farron @guacalive/flv.js modified-flv.js tuji.flv.js @authereum/sdk
flv.lm.js remix-solidity-sipc newstudio-solidity blocktrail-sdk-proxy
dac-remix-solidity chainsql-remix-solidity libsignal-protocol-nodejs
@efox/ts-service-oversea @thronelless/libsignal-protocol cflv.js
ns-browser-audio-capture flv.ly.js @netology-group/react-hls rpd-mapbox-gl
@jscad/core another-mapmagic-gl vicapow-mapbox-gl @bewithjonam/mapbox-gl
karn-mapbox-gl @3drobotics/mapbox-gl mapbox-gl-mapmagic
aframe-physics-system pec mapbox-gl-testing @jscad/openjscad bergben-pica
@usco/stl-parser @hola.org/hls.js @appearhere/mapbox-gl connect-hls.js
blocktrail-sdk @ivestander/mapbox-gl worker-proxy flv.js
jupyter-glmapi-components

and more...

Install

```
> npm i webworkify
```

Weekly Downloads

52,861

Version	License
1.5.0	MIT

Homepage

github.com/substack/webworkify

Repository

github.com/substack/webworkify

Last publish

3 years ago

Collaborators

[Try on RunKit](#)

packages depending on **webworkify**

flv.js

HTML5 FLV Player

zhengqian published 1.5.0 • 2 years ago

pica

High quality image resize in browser.

vitaly published 6.1.1 • 7 months ago

roslib

The standard ROS Javascript Library

mdevolving published 1.1.0 • a year ago

@jscad/core

Reuseable utilities for the various jscad user interfaces

kaosat-dev published 0.4.0 • 2 years ago

xgplayer-flv.js

Web video player

zhangxin92 published 2.3.0 • 9 days ago

blocktrail-sdk

以flv.js为例，看看这个模块是做什么的，webworker又用在了什么地方

flv.js介绍

它是做什么的？

它让我们能够用HTML5的video标签去播放flv格式的视频。

为什么要用HTML5的video标签去播放flv格式的视频呢？

因为在以前flash是音视频领域的霸主，很多视频格式是根据flash来制定的，http-flv这个视频格式的应用非常广泛，直播、录播等都有用到，但是浏览器如果要使用flash来播放视频的话就会提示用户去安装flash插件，而且chrome放弃了对flash的支持，所以就会存在一种需求：在HTML5的播放器上去播放历史的flash格式的视频文件。它原生就是不支持的。

flv.js就是负责解码http-flv格式的视频，然后通过一个Web API名叫Media Source Extensions API把它喂到HTML5的video播放器里面，它解码的这个步骤就是放在webworker里面的。

作业：

了解SharedWorker和ServiceWorker（PWA的基础，可以用ServiceWorker做一些页面相关的缓存优化之类的，使用场景很多）