Electron Security

nodeIntegrationInWorker

NOP Team

nodeIntegrationInWorker

0x01 简介

这篇文章很重要，希望大家能看完

大家好，今天和大家讨论 `nodeIntegrationInWorker` ，这个选项看起来和 `nodeIntegration` 很想，不过后面跟了 `InWorker` ，说明是在 `Worker` 中开启 `Node.js` ，默认值为 `false`

问题来了，什么是 `Worker` 呢？ 我看我像是 `Worker` ，但是这里说的是 `Web Worker`

> Web Worker 是一项 HTML5 提出的技术，它允许在Web应用程序中创建后台线程，以实现JavaScript的多线程处理能力。具体来说，Web Worker 是一个独立于主线程（即浏览器的UI线程）运行的JavaScript线程，用于执行耗时的、计算密集型或其他可能阻塞用户界面的任务，确保这些任务不会影响到页面的响应性和用户体验。
>
> Web Worker 常用于以下场景：
>
> - **大数据处理**：如批量数据过滤、排序、计算等复杂算法。
> - **图像处理**：如像素操作、压缩、解码等图形处理任务。
> - **科学计算**：如数学模型的迭代计算、物理模拟等高性能计算需求。
> - **离线存储处理**：如 IndexedDB 数据的批量读写、同步操作。
> - **长时间运行的任务**：如长轮询、定时任务、长时间运行的计数器等，避免影响页面响应性。
> - **网络通信**：处理 XMLHttpRequest 或 Fetch API 请求，尤其是处理大量并发请求或流式数据。
>
> 通过使用 Web Worker，开发者能够有效地解决JavaScript单线程环境下可能出现的性能瓶颈问题，确保即使在执行繁重任务时，Web应用仍能保持流畅的用户界面和良好的响应速度。

`JavaScript` 多线程一直是一个非常别扭的事情，用过的人都迷糊，有了 `Worker` 以后应该会缓解一些

> https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Workers_API/Using_web_workers
>
> https://www.electronjs.org/zh/docs/latest/api/structures/browser-window-options

> https://www.electronjs.org/zh/docs/latest/tutorial/multithreading

0x02 Web Worker

1. Web Worker 简介

一个 `worker` 是使用一个构造函数创建的一个对象（例如 `Worker()`）运行一个命名的 `JavaScript` 文件

这个文件包含将在 `worker` 线程中运行的代码; **`worker` 运行在另一个全局上下文中，不同于当前的 `window`**。因此，在 `Worker` 内通过 `window` 获取全局作用域（而不是 `self`）将返回错误

`Worker` 分为两类

- 专用 `Worker`

  一对一关联，即一个 `Worker` 服务于一个主线程，由创建它的脚本独享

- 共享 `Worker`

  一对多关联，一个共享 `Worker` 可以被多个页面（主线程）访问和通信，适用于跨页面共享资源或协同工作

从 `Electron` 的官方描述来看， `nodeIntegrationInWorker` 目前只支持专用 `Worker` ，而且必须将 `sandbox` 设置为 `false` 才有效

关键的是，从目前 `Electron` 官方描述来看， `Node.js` 似乎还没有做好关于 `Worker` 的准备

原生Node.js模块

在Web Workers里可以直接加载任何原生Node.js模块，但不推荐这样做。 大多数现存的原生模块是在假设单线程环境的情况下编写的，如果把它们用在Web Workers里会导致崩溃和内存损坏。

> https://www.electronjs.org/zh/docs/latest/tutorial/multithreading

更多关于 `Web Worker` 的介绍可以参考下面的文章

> https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Workers_API/Using_web_workers#%E4%B8%93%E7%94%A8_worker
>
> https://www.ruanyifeng.com/blog/2018/07/web-worker.html

2. 创建 Web Worker

如何创建一个专用 `Worker` 呢?

```
const myWorker = new Worker("worker.js");
```

这样就生成了一个 `Web Worker` ， `Web Worker` 会运行 `worker.js` 中的代码，其中 `worker.js` 也可以是一个 `URL` ，但必须是同源的

3. 主线程与 Worker 通信

这有点像 `Electron` 中的主进程和渲染进程通信了。主线程和 `Worker` 线程的通信是通过 `postMessage` 和 `onmessage` 进行通信的

`worker.js`

```
// worker.js
self.addEventListener('message', function(e) {
  const data = e.data;
  // 处理收到的数据并进行计算或处理
  const result = performComputation(data);

  // 将结果发送回主线程
  self.postMessage(result);
}, false);

function performComputation(inputData) {
  // 在这里编写具体的计算逻辑
  // ...
  return computedResult;
}
```

上面的代码是一个简单的计算 `demo` ,主进程发送数据后，它便进行一些运算，并通过 `postMessage` 返回给主线程

主线程这边

```
// 创建 Worker，传入 Worker 脚本文件的路径
const myWorker = new Worker('worker.js');

// 主线程向 Worker 发送消息
myWorker.postMessage(someInputData);

// 监听 Worker 返回的结果
myWorker.addEventListener('message', function(e) {
  const result = e.data;
  console.log('Received result from Worker:', result);
  // 根据结果进行后续操作
}, false);
```

4. 错误处理

为确保程序健壮性，应在主线程中监听 `Worker` 的 `error` 事件以处理 `Worker` 执行过程中的错误

```
myWorker.addEventListener('error', function(errorEvent) {
  console.error('Error occurred in Worker:', errorEvent.message);
}, false);
```

5. 关闭 Worker

当不再需要 `Worker` 时，调用 `worker.terminate()` 方法来停止 `Worker` 并释放其资源。

```
myWorker.terminate();
```

0x03 测试 nodeIntegrationInWorker

`main.js`

```
// Modules to control application life and create native browser window
const { app, BrowserWindow } = require('electron')
const path = require('path')

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 1600,
    height: 1200,
    webPreferences: {
      nodeIntegrationInWorker: true,
      sandbox: false,
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')
  // mainWindow.loadURL('https://iqinban.com/')

  // Open the DevTools.
  mainWindow.webContents.openDevTools()
}

app.whenReady().then(() => {
  createWindow()

  app.on('activate', function () {
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') app.quit()
})
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self';
script-src 'self'">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using Node.js <span id="node-version"></span>,
    Chromium <span id="chrome-version"></span>,
    and Electron <span id="electron-version"></span>.

    <!-- You can also require other files to run in this process -->
    <script src="./renderer.js"></script>
  </body>
</html>
```

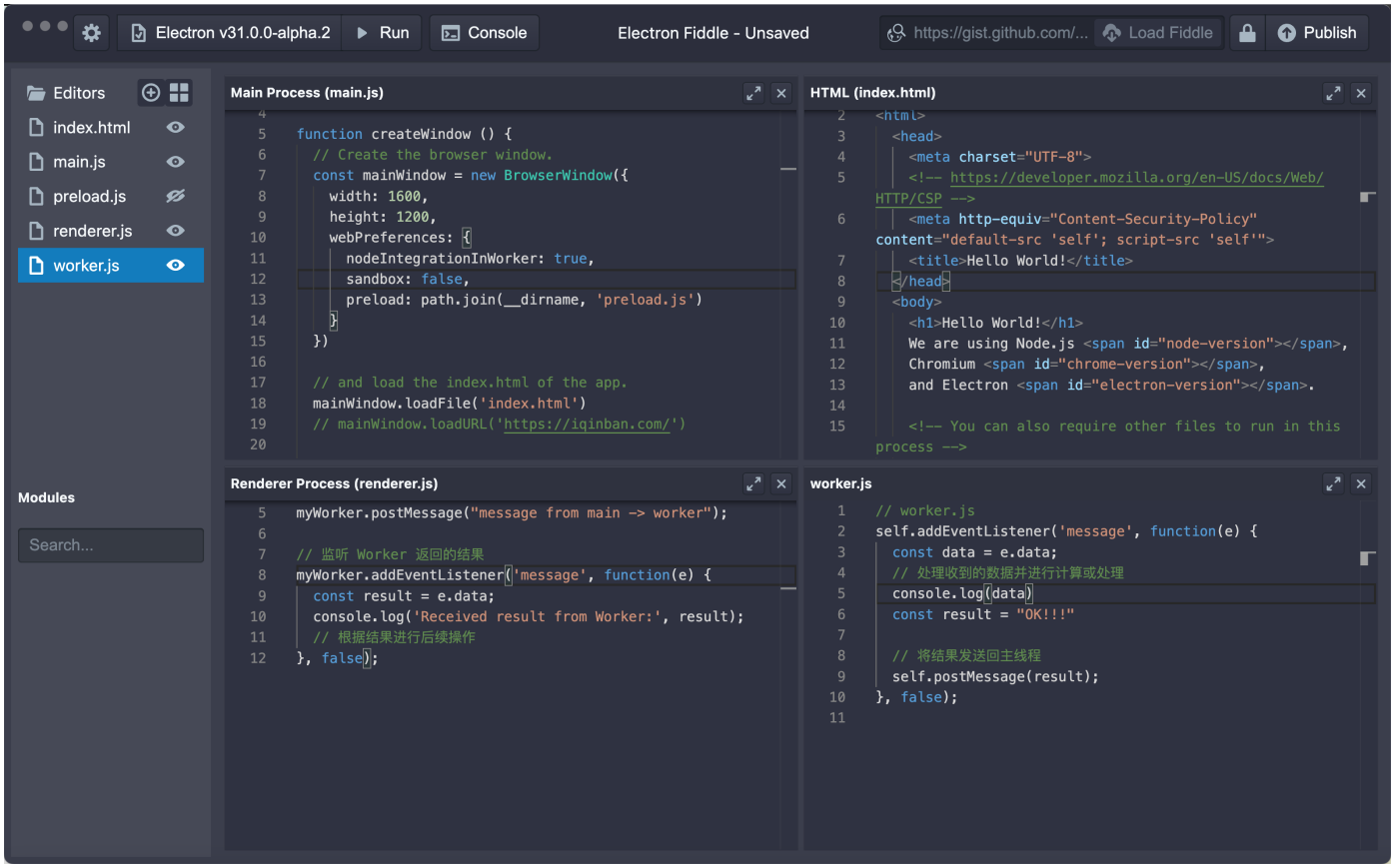
renderer.js

```
// 创建 Worker，传入 Worker 脚本文件的路径
const myWorker = new Worker('worker.js');

// 主线程向 Worker 发送消息
myWorker.postMessage("message from main -> worker");

// 监听 Worker 返回的结果
myWorker.addEventListener('message', function(e) {
  const result = e.data;
  console.log('Received result from Worker:', result);
  // 根据结果进行后续操作
}, false);
```

`worker.js`

```
// worker.js
self.addEventListener('message', function(e) {
  const data = e.data;
  // 处理收到的数据并进行计算或处理
  console.log(data)
  const result = "OK!!!"

  // 将结果发送回主线程
  self.postMessage(result);
}, false);
```

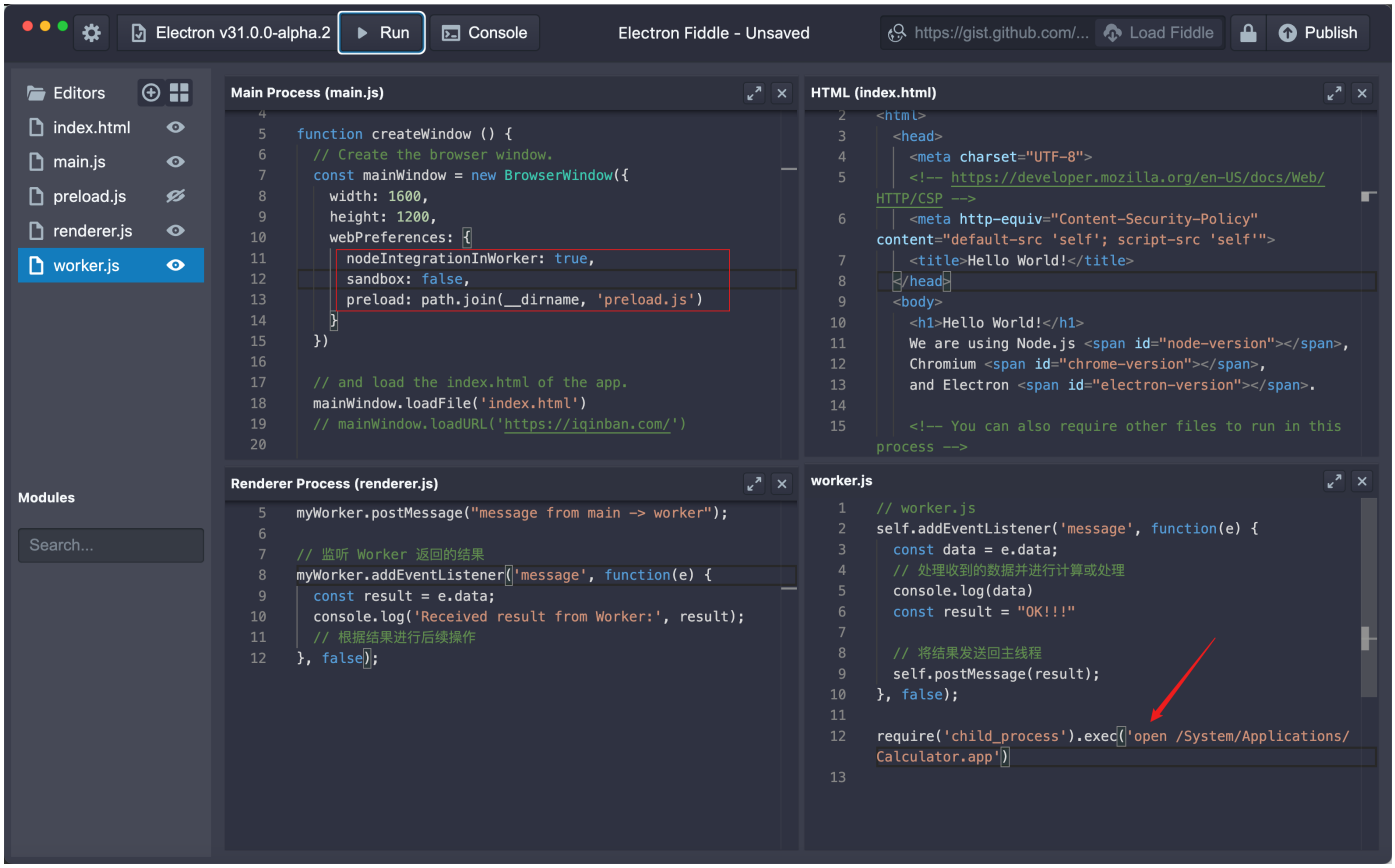
1. 功能测试

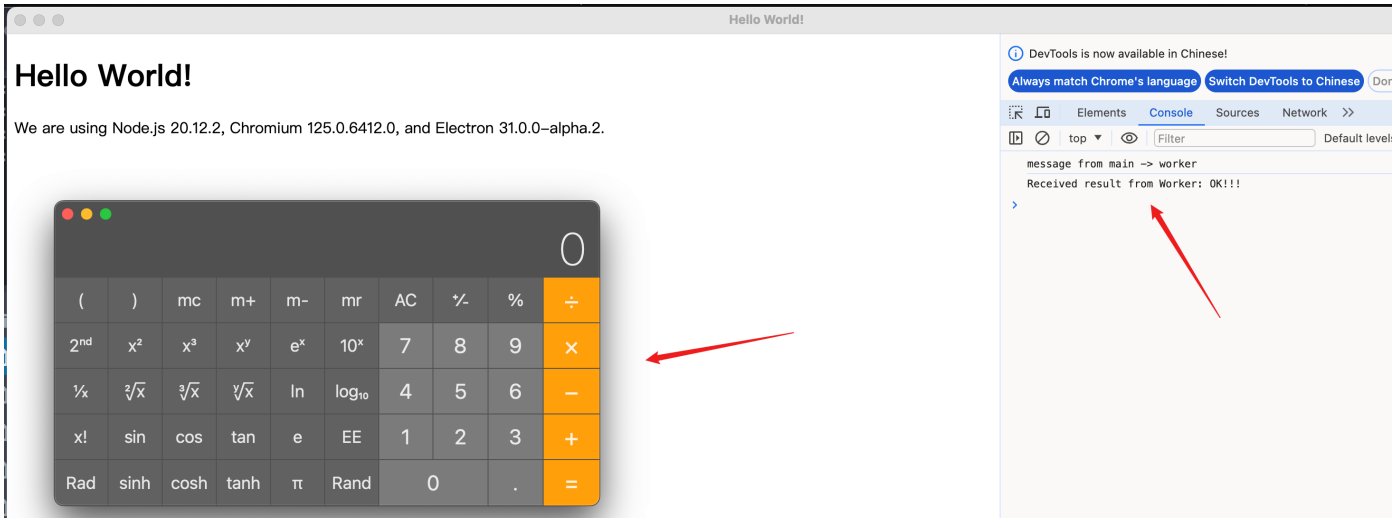


可以看到， `Worker` 功能是没有问题的，现在测试一下 `Node.js` 的能力

添加如下 `Payload`

```
require('child_process').exec('open /System/Applications/Calculator.app')
```





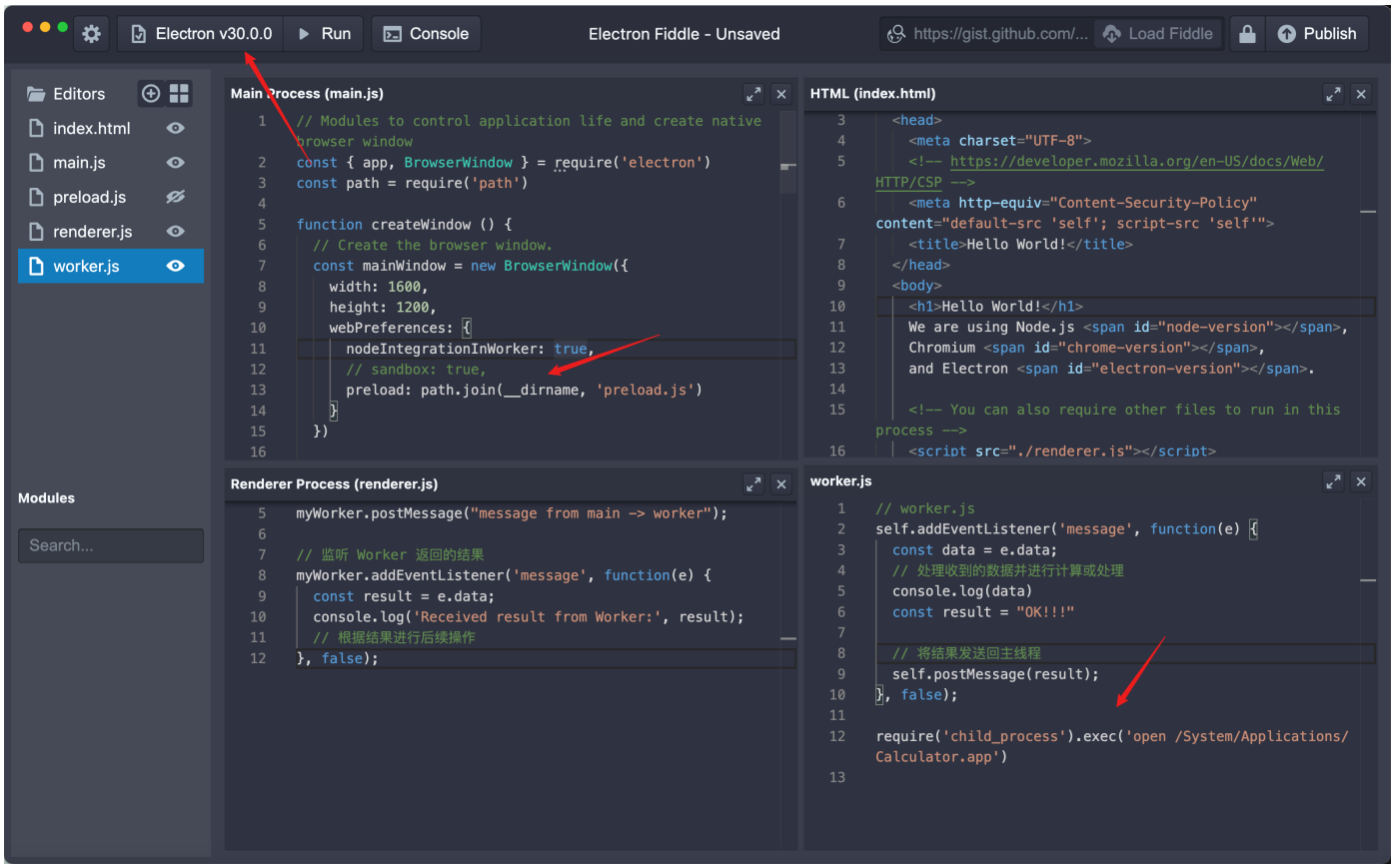
成功打开计算器，`Worker` 确实获得了 `Node.js` 的能力，此时 `nodeIntegration` 处于默认的 `false`，这两个选项之间没有关系

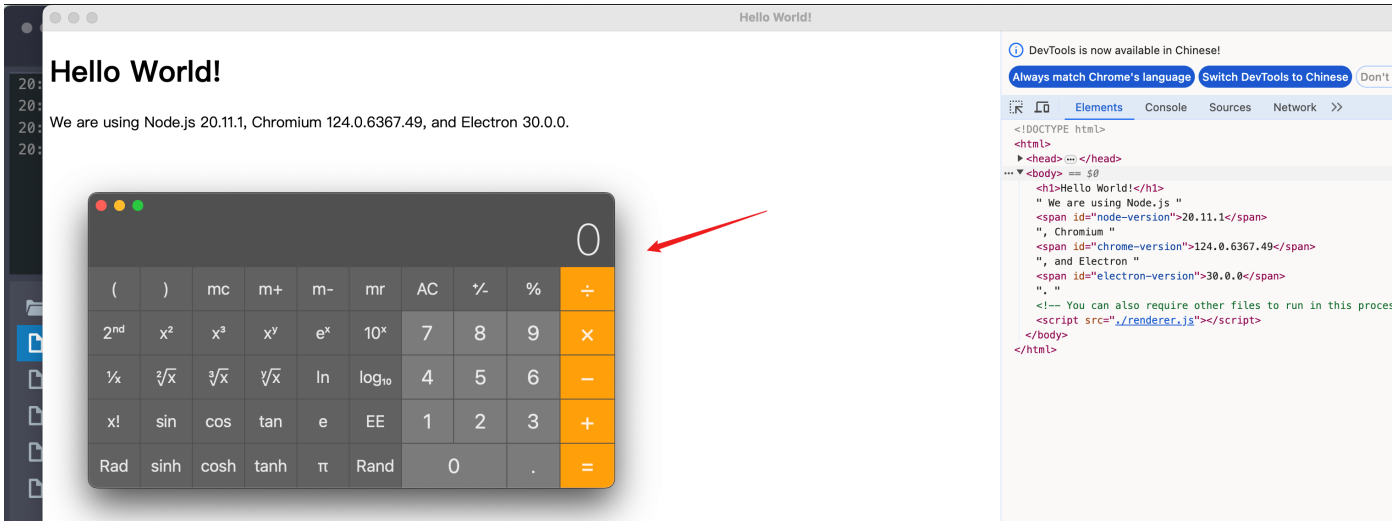
如果开启 `sandbox`

`Worker` 不再具备 `Node.js` 能力

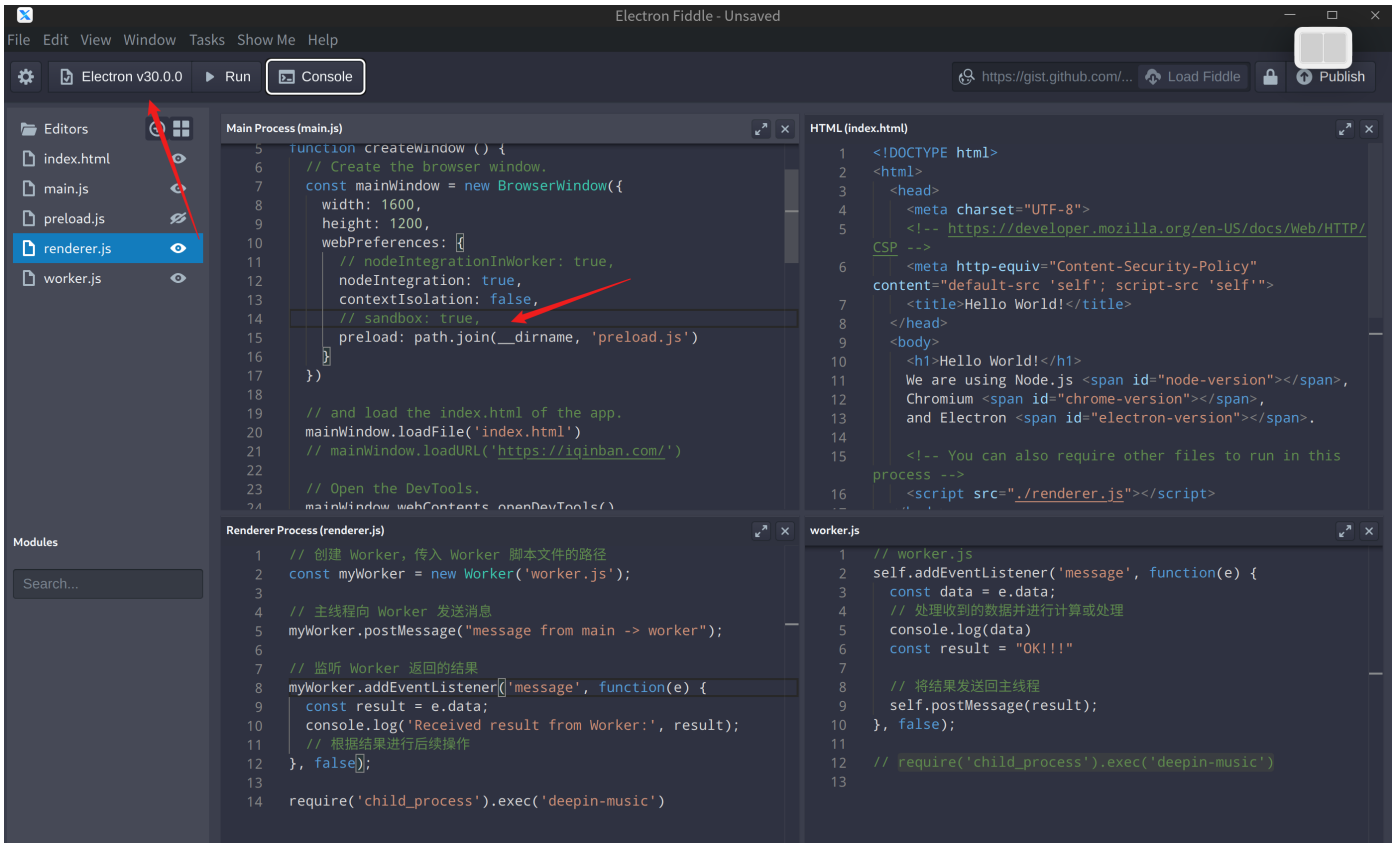
2. 特别注意

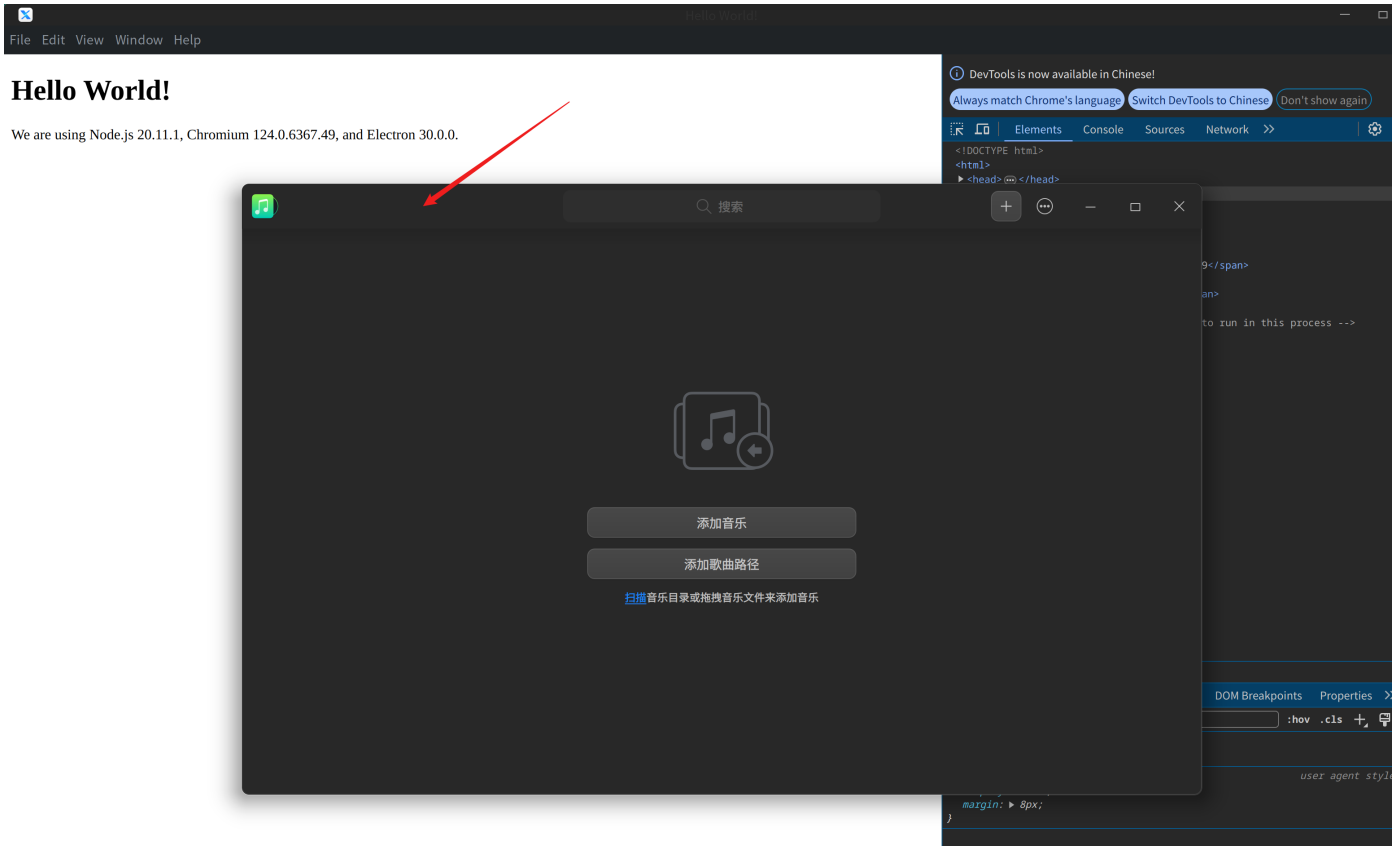
有趣的是，我们知道，`sandbox` 选项默认在 `Electron 20.0` 中开始默认为 `true`，但是经过我的测试，**只有当 `sandbox` 被显式地设置为 `true` 时，才会阻止 `Worker` 获得 `Node.js` 的能力，当然前提是 `nodeIntegrationInWorker` 被设置为 `true`**





令人震惊的是，我顺带测试了一下 `nodeIntegration`

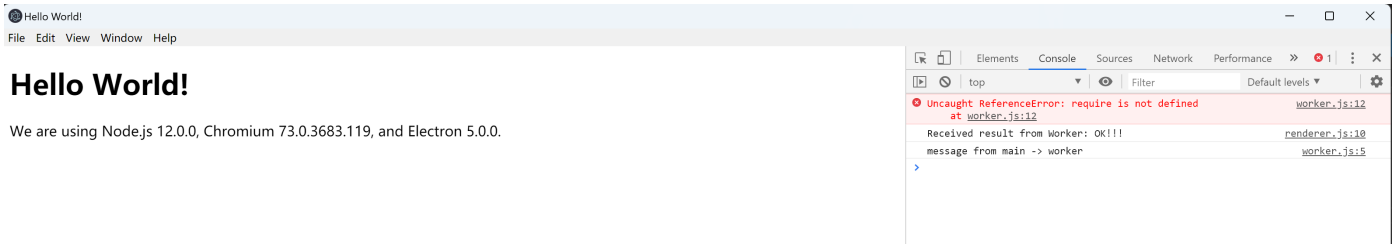


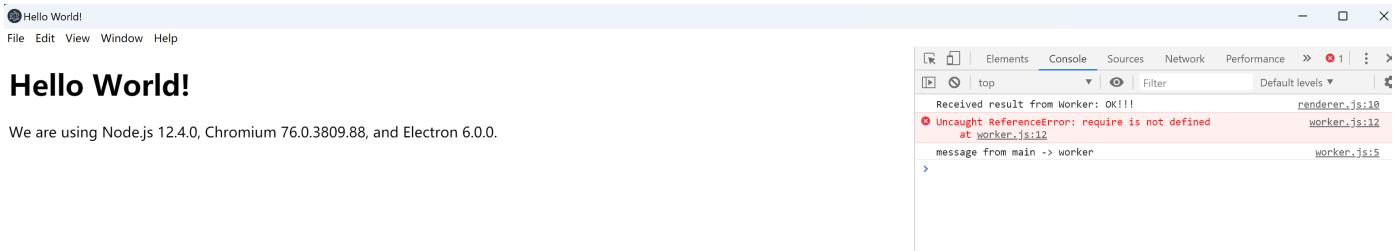


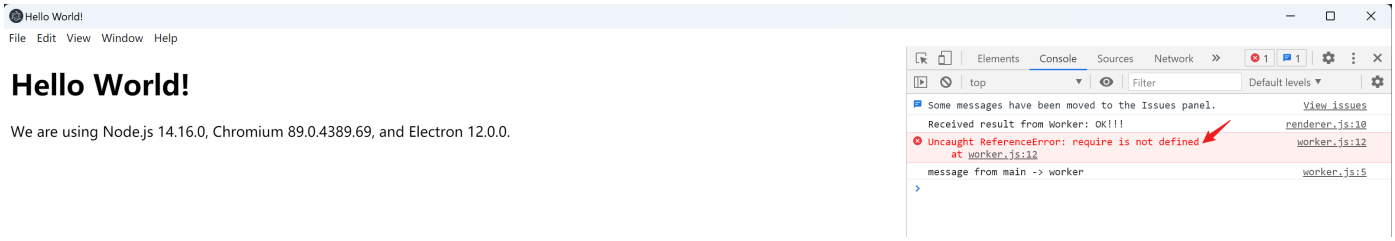
对于 `nodeIntegration` 来说也是一样的

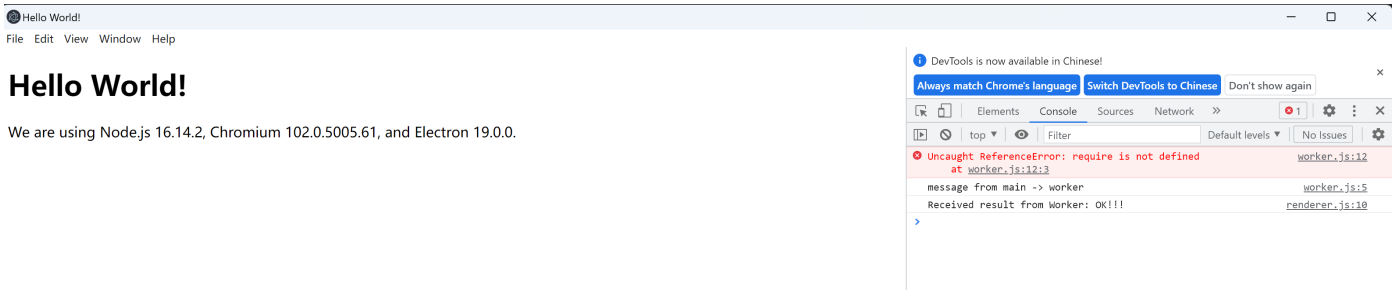
3. sandbox显式地设置为 true

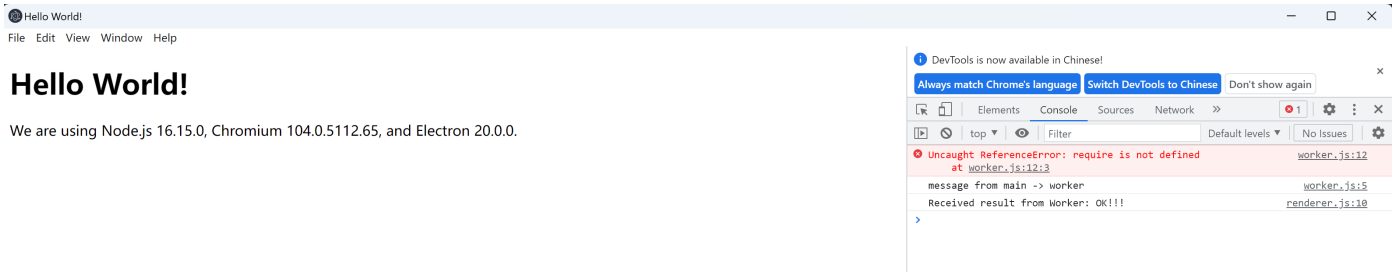
既然有了上面的发现，那么我们接下来的测试将 `sandbox` 显式地设置为 `true` ，看看在各个版本会不会有什么不一样

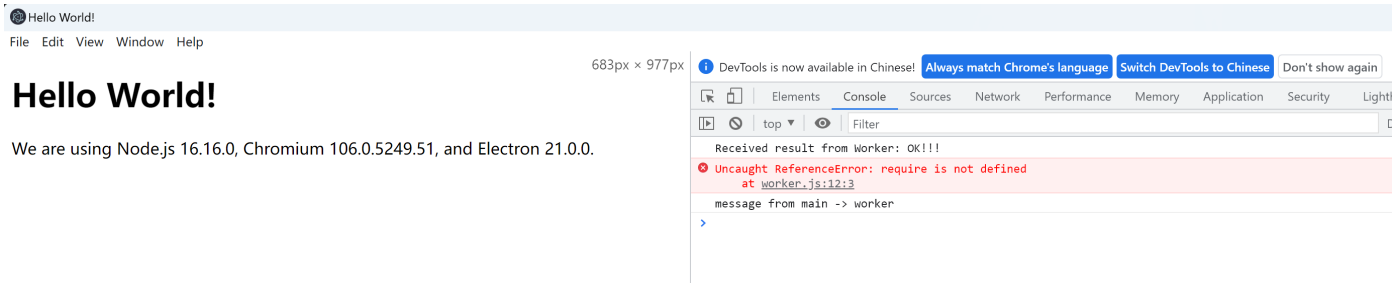


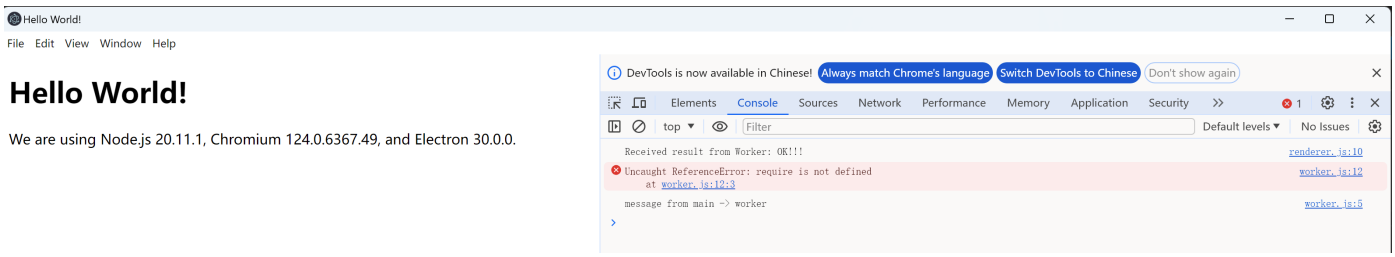












可以看到，测试了 `Electron 5.0、6.0、12.0、19.0、20.0、21.0、30.0` 表现是一致的，如果显式地设置了 `sandbox: true` ，则即使设置 `nodeIntegrationInWorker` 为 `true` ，`Worker` 也不具备 `Node.js` 的能力

0x04 总结

`nodeIntegrationInWorker` 配置项的作用是赋予 `Web Worker` `Node.js` 的能力，这个能力只有在 `sandbox` 没有显式地设置为 `true` 时起作用

一定要注意，虽然官方曾说在 `Electron 20.0` 版本开始，默认对渲染进程沙盒化，但是实际测试发现，如果没有显式的设置 `sandbox:true` ，即使是 `Electron 20.0` 版本以后，也不会对 `nodeIntegration` 、 `nodeIntegrationInWorker` 、 `preload` 的 `Node.js` 执行造成阻碍