

Electron Security

web嵌入



NOP Team



web嵌入 | Electron 安全

0x01 简介

大家好，今天和大家讨论的是 Web 嵌入，无论是网站还是应用程序，在部分场景下我们需要嵌入一些第三方的 web 内容，例如我写了篇技术文章，其中部分包含视频内容，我上传到 B 站上了，我想把这段内容嵌入到我的技术文章中，就可能要使用 web 嵌入技术

在 Electron 中有三种方式可以让你在 Electron 的 `BrowserWindow` 里集成（第三方）web 内容，`<iframe>` 和 `<webview>` 和 `WebContentsView` 每个功能都略有不同，适用于不同的情况。

其实要是扣字眼的话，web 嵌入范围会很大，一个 `img` 或 `video` 标签也可以算得上是 web 嵌入，今天讨论的 web 嵌入主要是嵌入第三方网站这类的操作

在 Electron 官方介绍中，并没有介绍在 `iframe` 之前出现的 web 嵌入技术 —— `object` 和 `embed`，在 `Java Applet` 和 `Flash` 那个时代，它们的嵌入就是通过 `object` 和 `embed` 实现的

所以今天的文章中，我们都尝试一下，看看它们在 Electron 中是否还可以使用

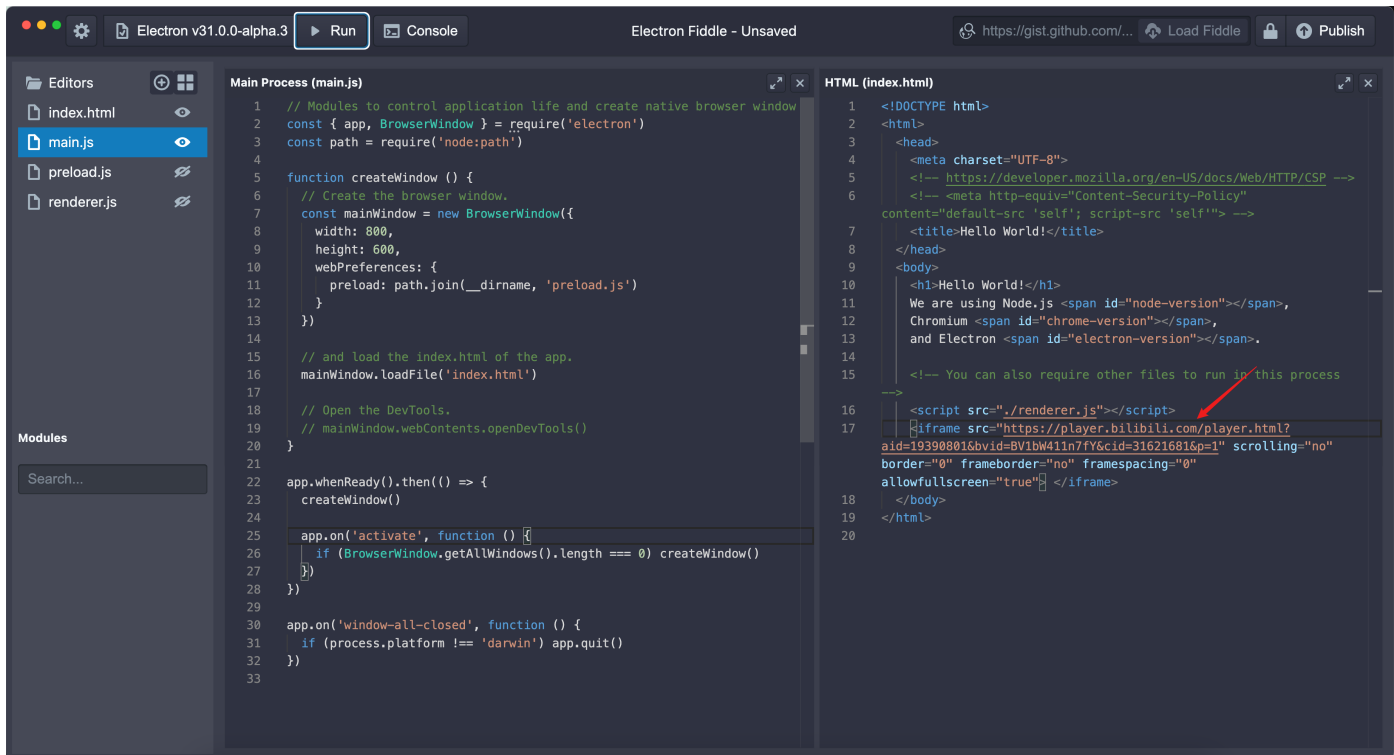
<https://www.electronjs.org/zh/docs/latest/tutorial/web-embeds>

https://developer.mozilla.org/zh-CN/docs/Learn/HTML/Multimedia_and_embedding/Other_embedding_technologies

0x02 iframe

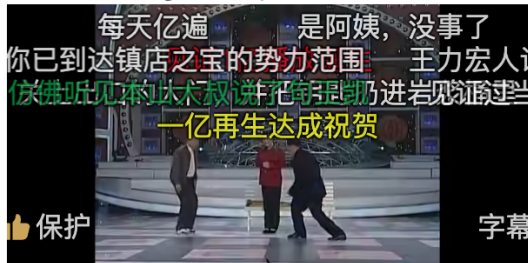
在之前的 `nodeIntegrationInSubFrames` 文章中，已经对 `iframe` 进行了部分介绍，这是一种现在通用的 web 嵌入方案，既然要加载第三方页面，那么肯定是允许跨域的，但跨域请求的地址受 `CSP` 策略的限制

关于 `CSP` 策略可以查看 [CSP | Electron 安全](#) 这篇文章



Hello World!

We are using Node.js 20.12.2, Chromium 125.0.6412.0, and Electron 31.0.0-alpha.3.



1. iframe 属性

`iframe` 元素包含全局属性，也就是包含那些所有标签都可以使用的属性

1) allow

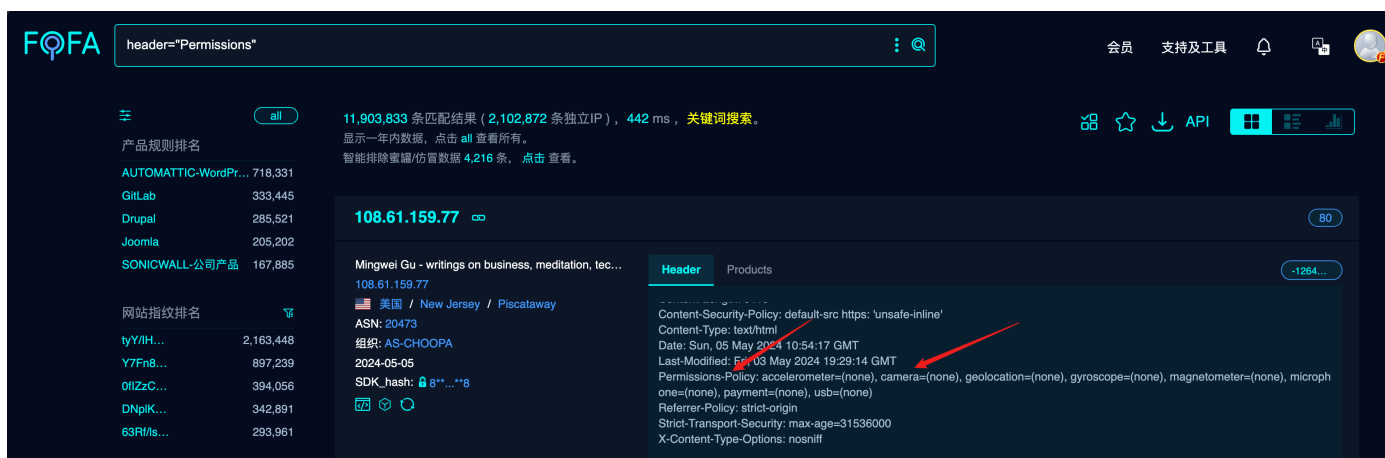
用于为 `iframe` 指定一个权限策略，该策略定义哪些功能可用于（例如，访问麦克风、摄像头、电池、Web共享等） `<iframe>` 根据请求的来源。

权限策略的意义如下：

- 改变手机和第三方视频自动播放的默认行为
- 限制网站使用相机、麦克风、扬声器等敏感设备
- 允许 iframe 使用全屏 API
- 如果项目在视口中不可见，则停止对其进行脚本处理，以提高性能

权限策略 (Permissions-Policy) 提供两种指定策略的方法：

- HTTP 头 - Permissions-Policy
- iframe 中的 allow 属性，当然控制的是 iframe 中使用的特性



权限策略采用继承制度，假如说页面的权限策略禁止访问麦克风，那么页面中嵌入的 `iframe` 会继承该策略，禁止使用麦克风，如果嵌入的 `iframe` 在 `allow` 属性中设置了自己的权限策略，那么就取子集

权限策略详细内容参考

https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Permissions_Policy

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Permissions-Policy#iframes>

2) allowfullscreen

设置为 `true` 时，可以通过调用 `<iframe>` 的 `requestFullscreen()` 方法激活全屏模式

其实在上面的权限策略已经包含全屏模式的问题了，所以这是一个历史遗留属性

3) csp

对嵌入的资源配置内容安全策略

4) height

以 CSS 像素格式，或像素格式，或百分比格式指定 frame 的高度。默认值为 150

5) importance

这是个实验性的属性，表示 `<iframe>` 的 `src` 属性指定的资源的加载优先级。允许的值有：

- auto (default)
不指定优先级。浏览器根据自身情况决定资源的加载顺序
- high
资源的加载优先级较高
- low
资源的加载优先级较低

6) name

用于定位嵌入的浏览上下文的名称

该名称可以用作 `a` 标签与 `form` 标签的 `target` 属性值，也可以用作 `input` 标签和 `button` 标签的 `formtarget` 属性值，还可以用作 `window.open()` 方法的 `windowName` 参数值

7) referrerpolicy

表示在获取 iframe 资源时如何发送 `referrer` 头部

这个其实在之前文章 [一次失败的漏洞串联尝试](#) 中有提过，具体可以取值如下：

取值	含义
<code>no-referrer</code>	不发送 <code>Referer</code> 头
<code>no-referrer-when-downgrade</code> (default)	向不受 TLS (HTTPS) 保护的 <code>origin</code> 发送请求时，不发送 <code>Referer</code> 头
<code>origin</code>	referrer 头中仅包含来源页面的源。换言之，仅包含来源页面的 <code>scheme</code> , <code>host</code> , 以及 <code>port</code>
<code>origin-when-cross-origin</code>	发起跨域请求时，仅在 referrer 中包含来源页面的源。发起同源请求时，仍然会在 referrer 中包含来源页面在服务器上的路径信息
<code>same-origin</code>	对于 <code>same origin</code> (同源) 请求，发送 referrer 首部，否则不发送
<code>strict-origin</code>	仅当被请求页面和来源页面具有相同的协议安全等级时才发送 referrer 首部（比如从采用 HTTPS 协议的页面请求另一个采用 HTTPS 协议的页面）。如果被请求页面的协议安全等级较低，则不会发送 referrer 首部（比如从采用 HTTPS 协议的页面请求采用 HTTP 协议的页面）
<code>strict-origin-when-cross-origin</code>	当发起同源请求时，在 referrer 首部中包含完整的 URL。当被请求页面与来源页面不同源但是有相同协议安全等级时（比如 HTTPS→HTTPS），在 referrer 首部中仅包含来源页面的源。当被请求页面的协议安全等级较低时（比如 HTTPS→HTTP），不发送 referrer 首部
<code>unsafe-url</code>	始终在 referrer 首部中包含源以及路径（但不包括 <code>fragment</code> ， <code>密码</code> ，或 <code>用户名</code> ）。 这个值是不安全的 ，因为这样做会暴露受 TLS 保护的资源的源和路径信息

8) sandbox

控制应用于嵌入在 `<iframe>` 中的内容的限制。该属性的值可以为空以应用所有限制，也可以为空格分隔的标记以解除特定的限制

这里必须注意，并不是说默认就开启 `sandbox`，而是需要显式的设置 `<iframe sandbox>`
`</iframe>` 或 `<iframe sandbox=""></iframe>`

这个属性也是与我们关系比较大的内容，采用了默认即安全的配置方式

取值	含义
<code>allow-downloads-without-user-activation</code>	实验性: 允许在没有征求用户同意的情况下下载文件
<code>allow-forms</code>	允许嵌入的浏览上下文提交表单。如果没有使用该关键字，则无法提交表单
<code>allow-modals</code>	允许嵌入的浏览上下文打开模态窗口
<code>allow-orientation-lock</code>	允许嵌入的浏览上下文锁定屏幕方向（译者注：比如智能手机、平板电脑的水平朝向或垂直朝向）
<code>allow-pointer-lock</code>	允许嵌入的浏览上下文使用 Pointer Lock API .
<code>allow-popups</code>	允许弹窗 (例如 <code>window.open</code> , <code>target="_blank"</code> , <code>showModalDialog</code>) 如果没有使用该关键字，相应的功能将自动被禁用
<code>allow-popups-to-escape-sandbox</code>	允许沙箱化的文档打开新窗口，并且新窗口不会继承沙箱标记。例如，安全地沙箱化一个广告页面，而不会在广告链接到的新页面中启用相同的限制条件
<code>allow-presentation</code>	允许嵌入的浏览上下文开始一个 presentation session
<code>allow-same-origin</code>	如果没有使用该关键字，嵌入的浏览上下文将被视为来自一个独立的源，这将使

	same-origin policy 同源检查失败
allow-scripts	允许嵌入的浏览上下文运行脚本（但不能创建弹窗） 如果没有使用该关键字，就无法运行脚本
allow-storage-access-by-user-activation	实验性: 允许嵌入的浏览上下文通过 Storage Access API 使用父级浏览上下文的存储功能
allow-top-navigation	允许嵌入的浏览上下文导航（加载）内容到顶级的浏览上下文
allow-top-navigation-by-user-activation	允许嵌入的浏览上下文在经过用户允许后导航（加载）内容到顶级的浏览上下文

备注：当被嵌入的文档与主页面同源时，强烈建议不要同时使用 `allow-scripts` 和 `allow-same-origin`。如果同时使用，嵌入的文档就可以通过代码删除 `sandbox` 属性，如此，就安全性而言还不如不用 `sandbox`。

- 如果攻击者可以在沙箱化的 `iframe` 之外展示内容，例如用户在新标签页中打开内联框架，那么沙箱化也就没有意义了。建议把这种内容放置到独立的专用域中，以减小可能的损失。
- 沙箱属性 (sandbox) 在 Internet Explorer 9 及更早的版本上不被支持。

这其中 `allow-scripts`、`allow-popups`、`allow-popups-to-escape-sandbox`、`allow-same-origin`、`allow-top-navigation` 与安全关系比较大

`allow-scripts` 是允许 `iframe` 嵌入的网页内部执行 JavaScript，如果没有设置则不允许执行

我们测试一下，`iframe` 远程加载我们的页面 `http://192.168.31.216/1.html`

1.html

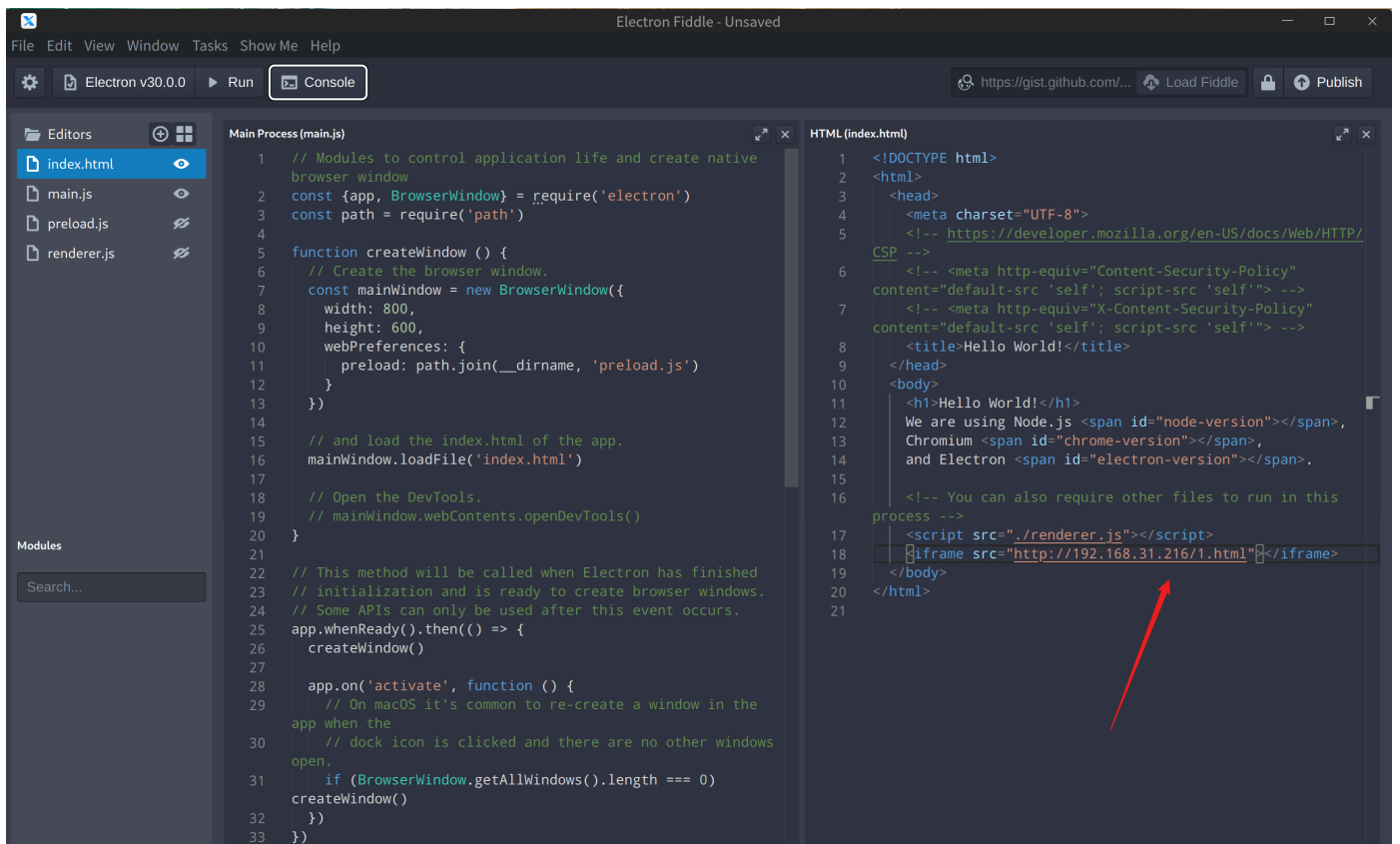
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
```

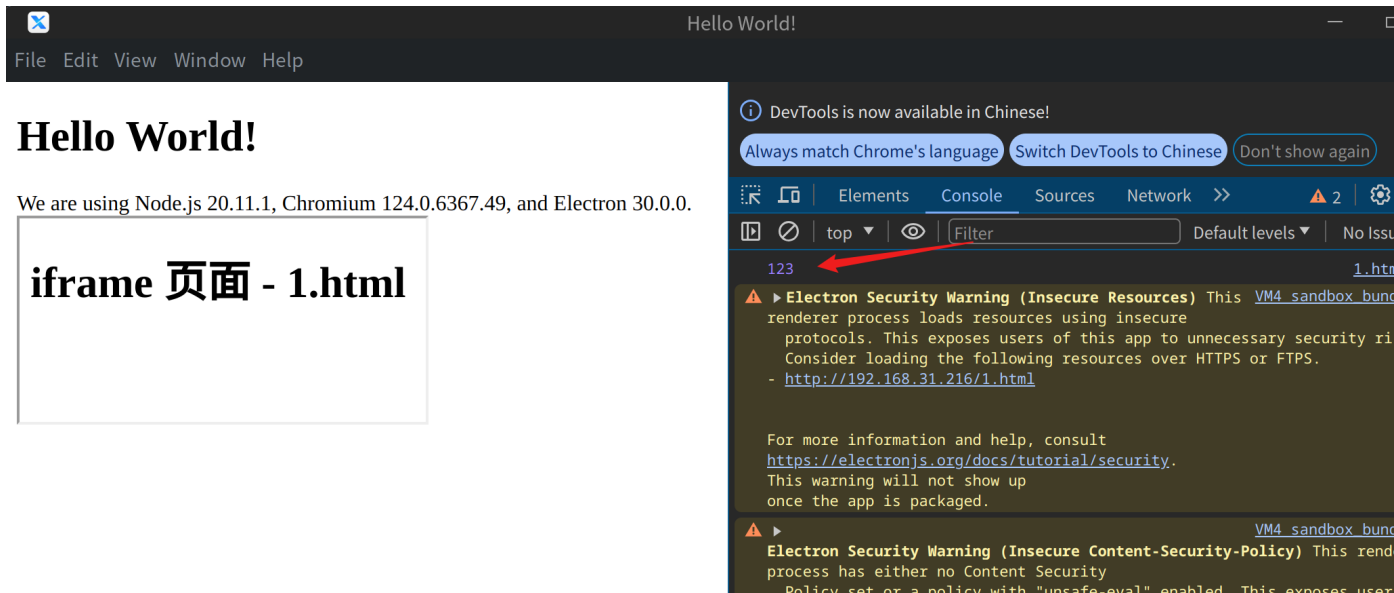
```

</head>
<body>
  <div>
    <h1>iframe 页面 - 1.html</h1>
    <script>
      console.log(123)
    </script>
  </div>
</body>
</html>

```

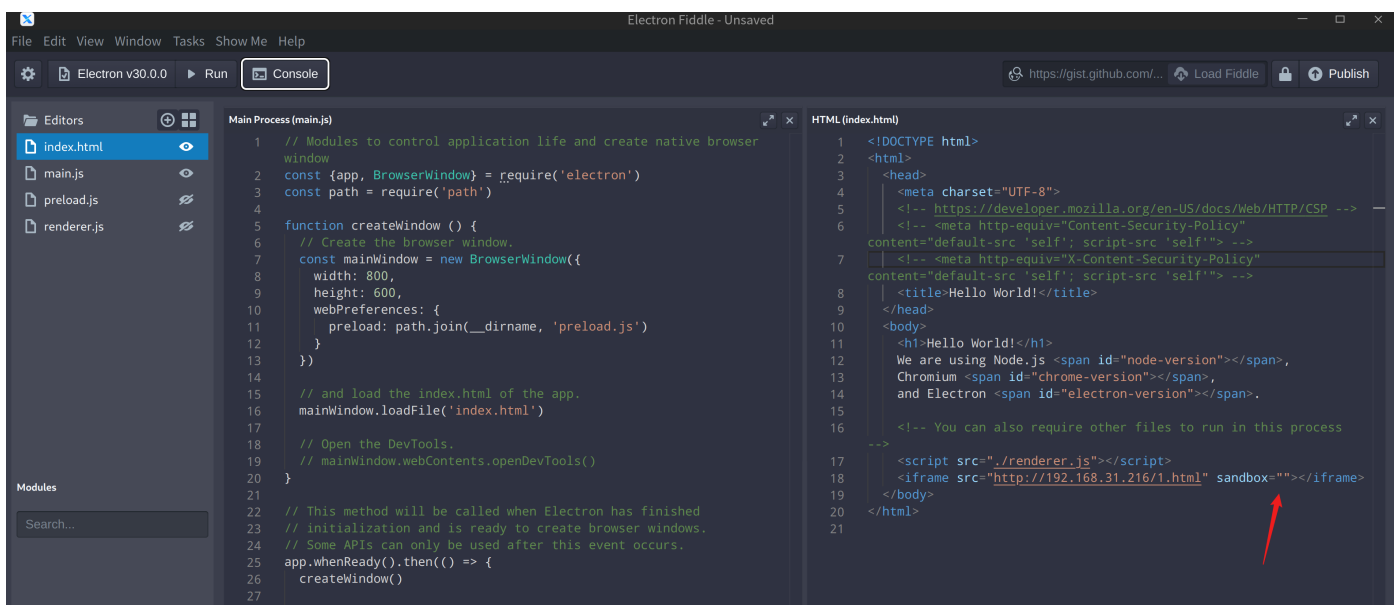
未显式这是 `sandbox`

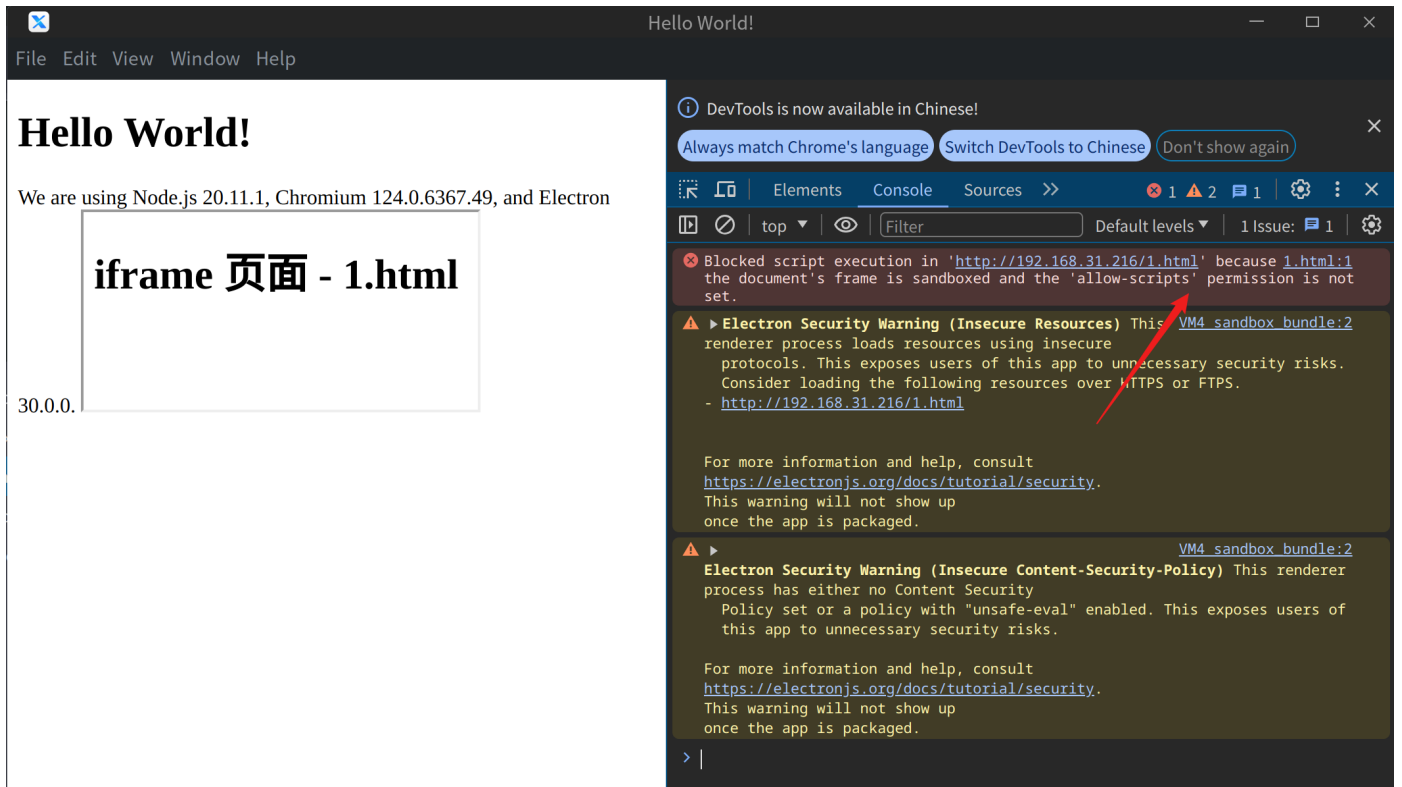




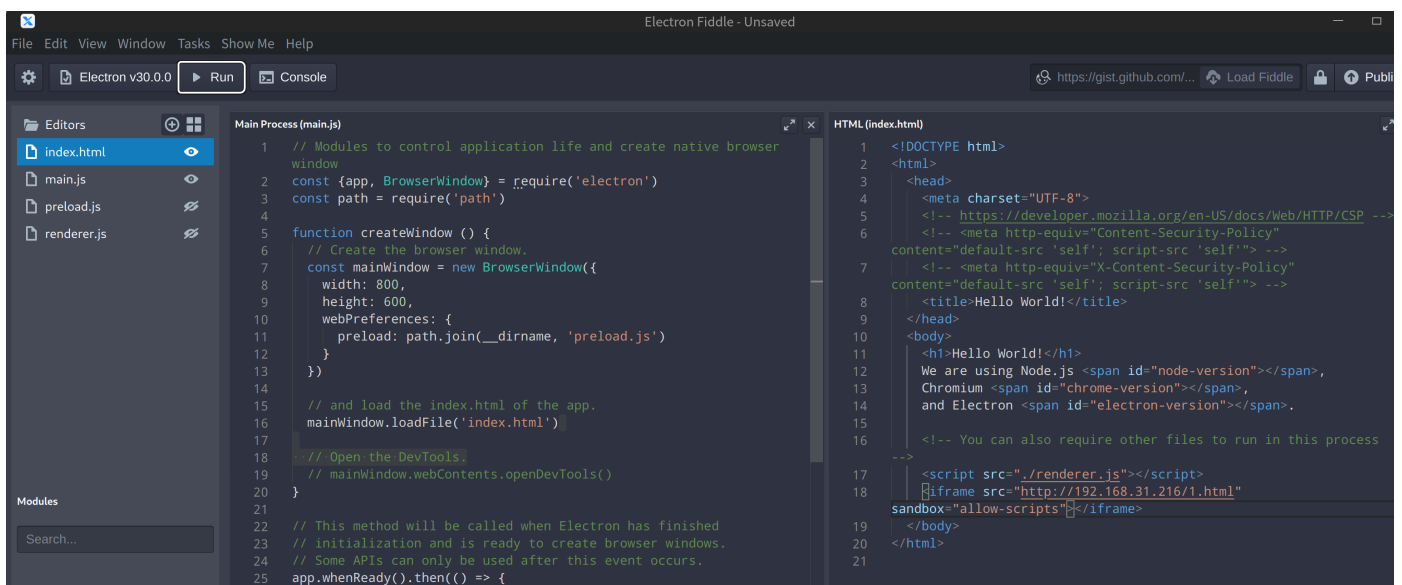
iframe 内部的脚本可以成功执行

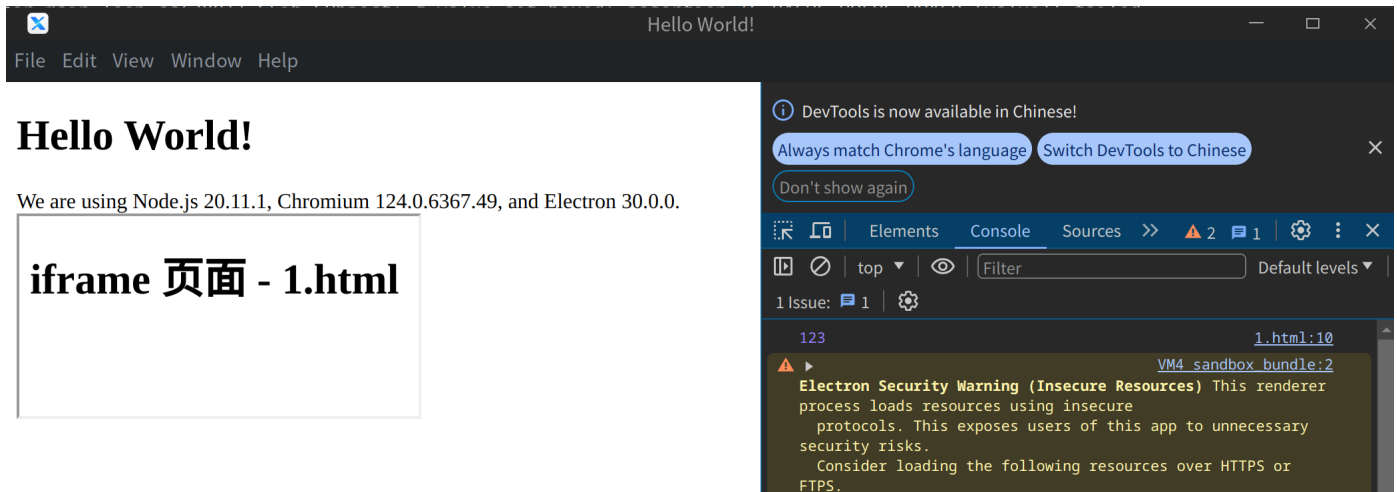
显式地设置 sandbox





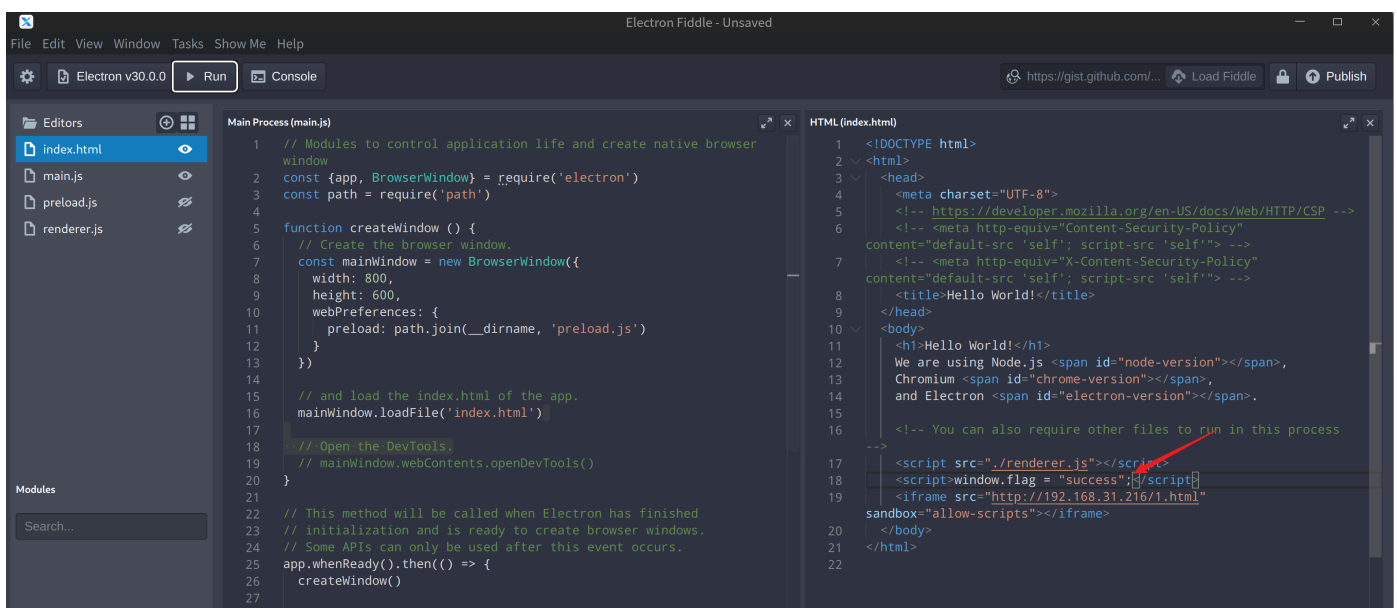
当 sandbox 设置了 allow-scripts 时





这里有一个问题，未设置 `sandbox` 或 `sandbox="allow-scripts"` 时 `iframe` 中的 JavaScript 和 Electron 渲染页面的 JavaScript 是同一个上下文吗？

在渲染页面设置 `window.flag = "success"`



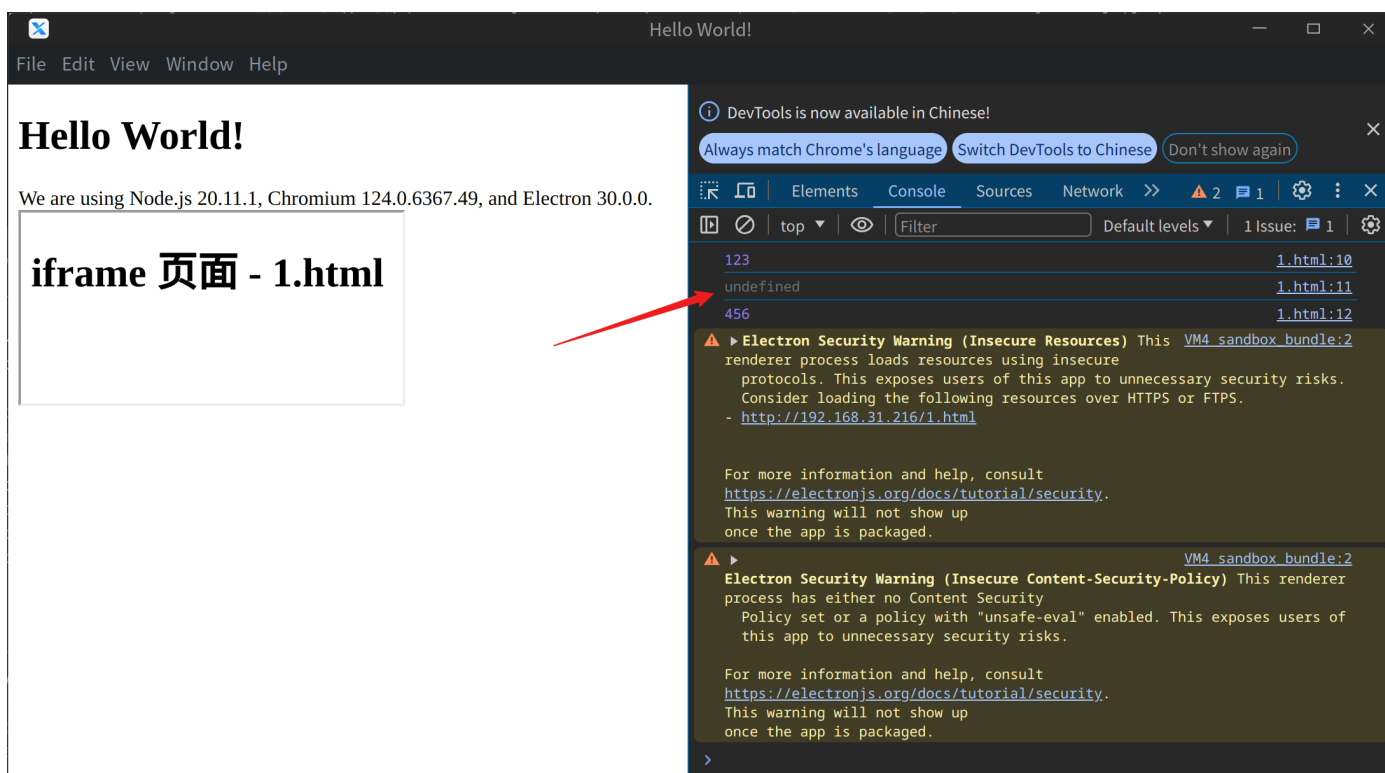
在 `iframe` 嵌入的内容中控制台输出 `window.flag`

```

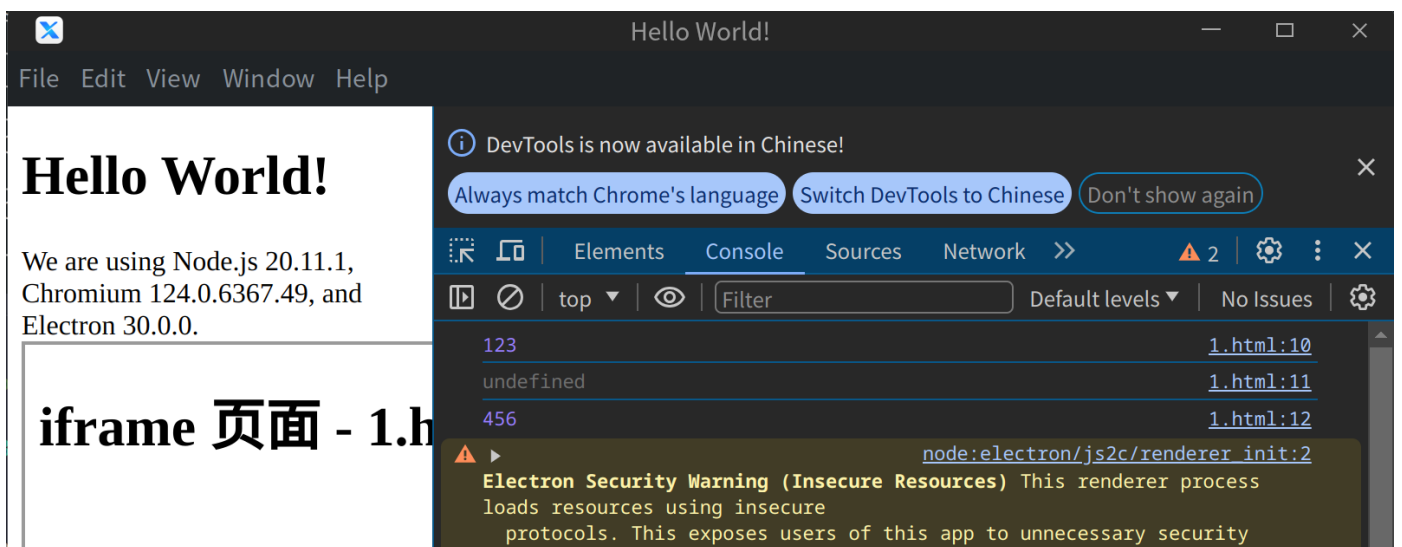
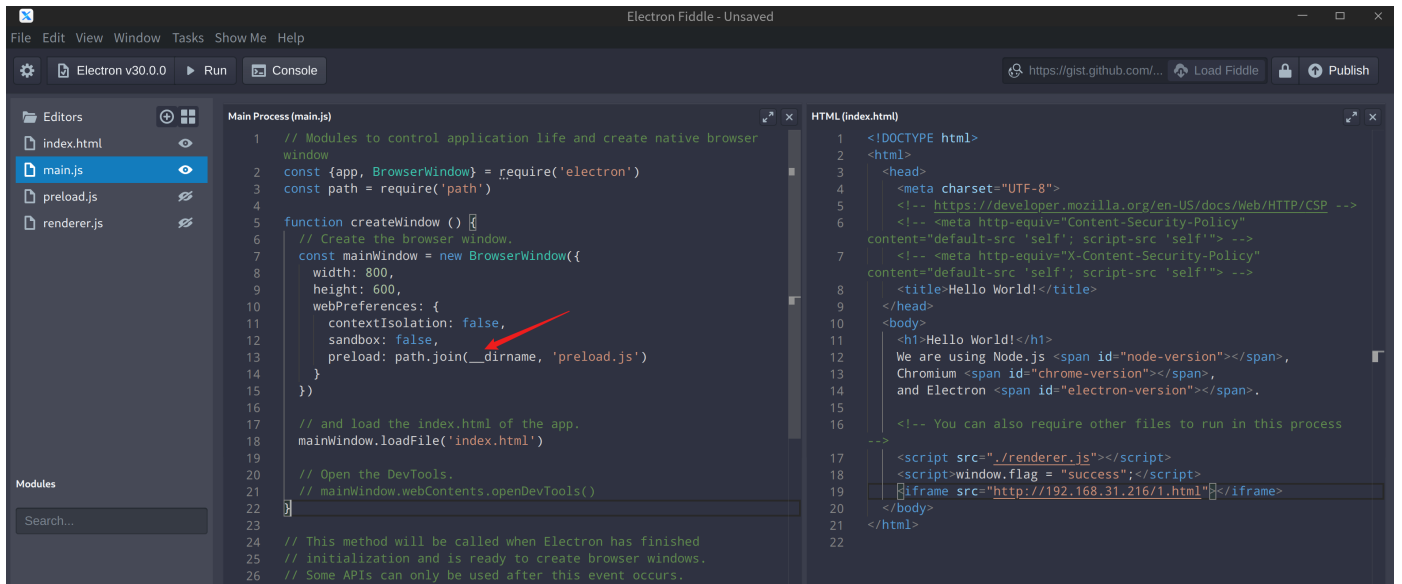
1  <!DOCTYPE html>
2  <html>
3  <head>
4  |  <meta charset="UTF-8">
5  </head>
6  <body>
7  |  <div>
8  |  |  <h1>iframe 页面 - 1.html</h1>
9  |  |  <script>
10 |  |  |  console.log(123);
11 |  |  |  console.log(window.flag);
12 |  |  |  console.log(456);|
13 |  |  </script>
14 |  </div>
15 </body>
16 </html>

```

进行测试

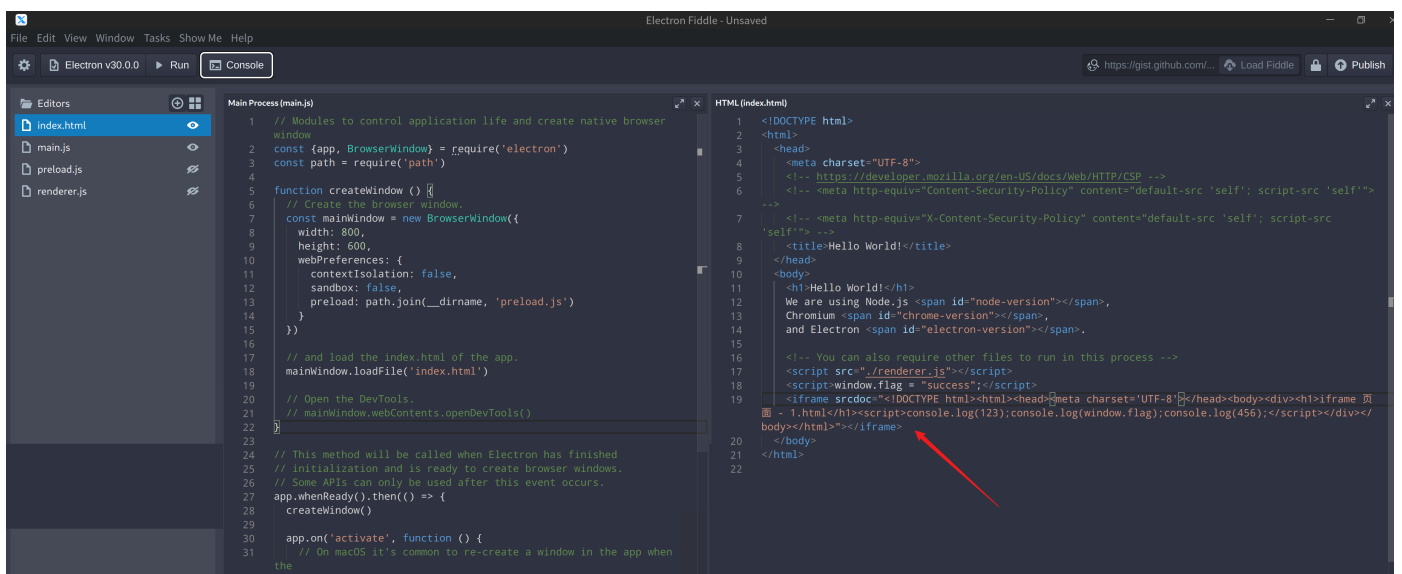


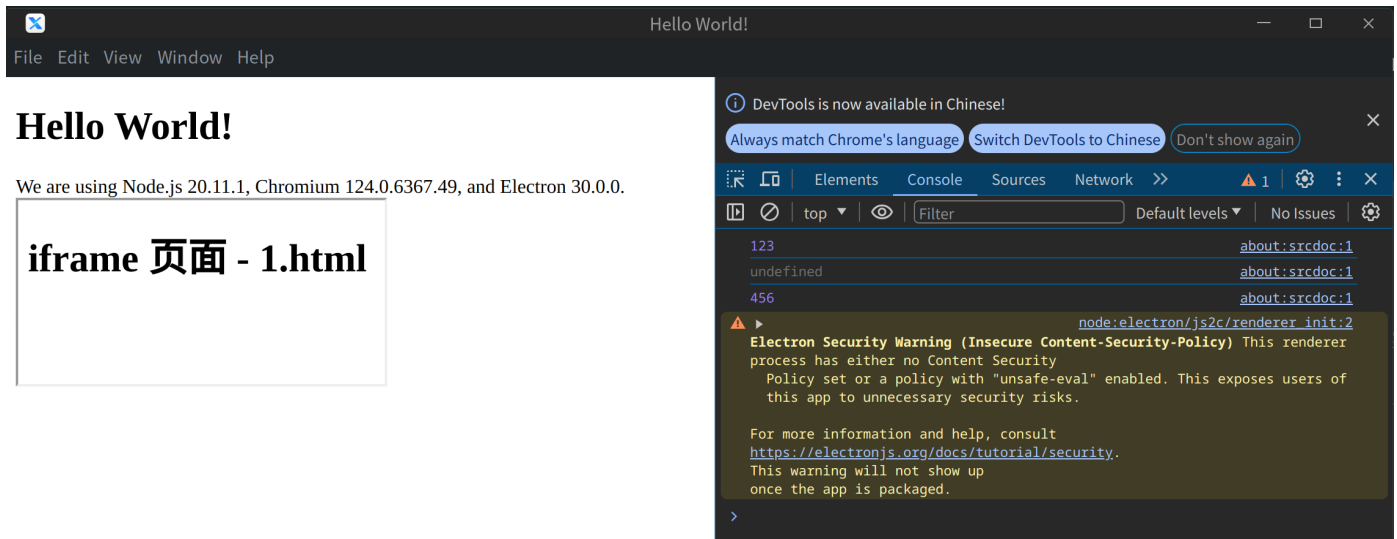
如果设置 `contextIsolation: false` 呢?



这也和之前文章介绍的一致，`iframe` 内部是一个独立的上下文

使用 `srcdoc` 执行也是一样的

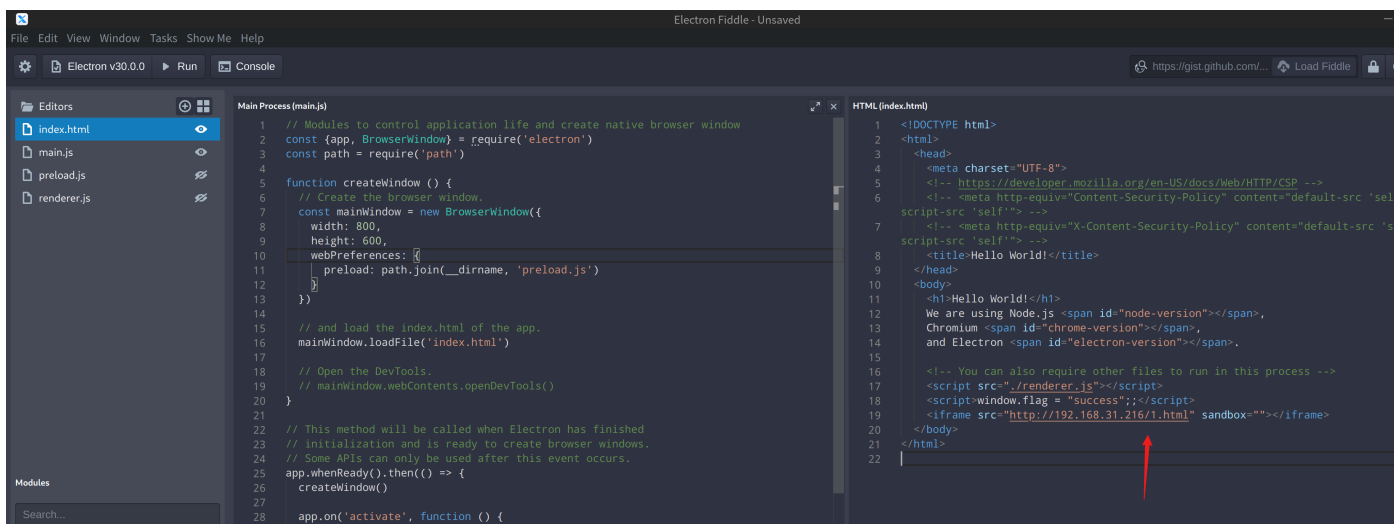




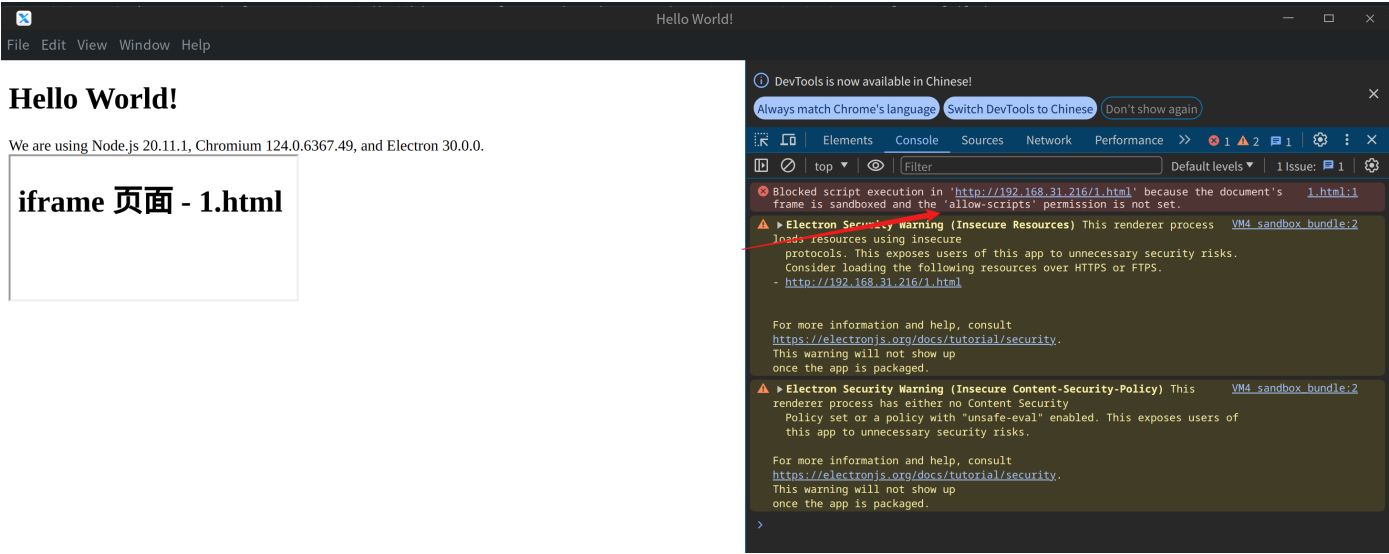
`allow-popups` 是允许弹窗，这里的弹窗并不是 `alert` 函数这种，而是 `window.open` 打开的这种真的窗口

在 `iframe` 加载的内容中，使用 `window.open` 打开 `https://www.baidu.com/`

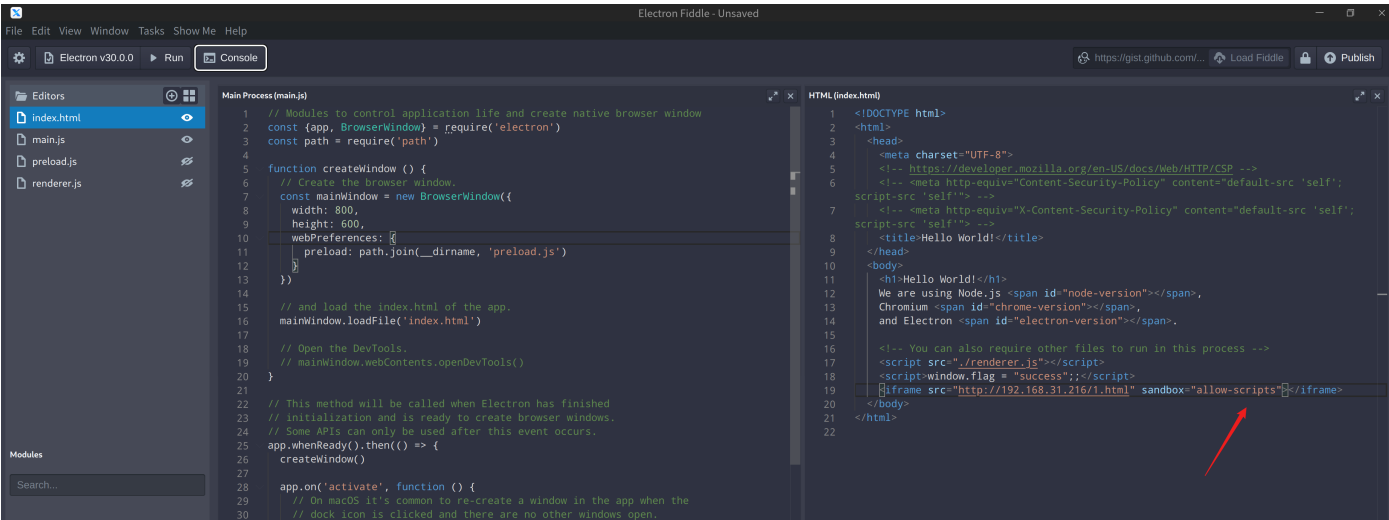
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <div>
    <h1>iframe 页面 - 1.html</h1>
    <script>
      window.open("https://www.baidu.com/")
    </script>
  </div>
</body>
</html>
```

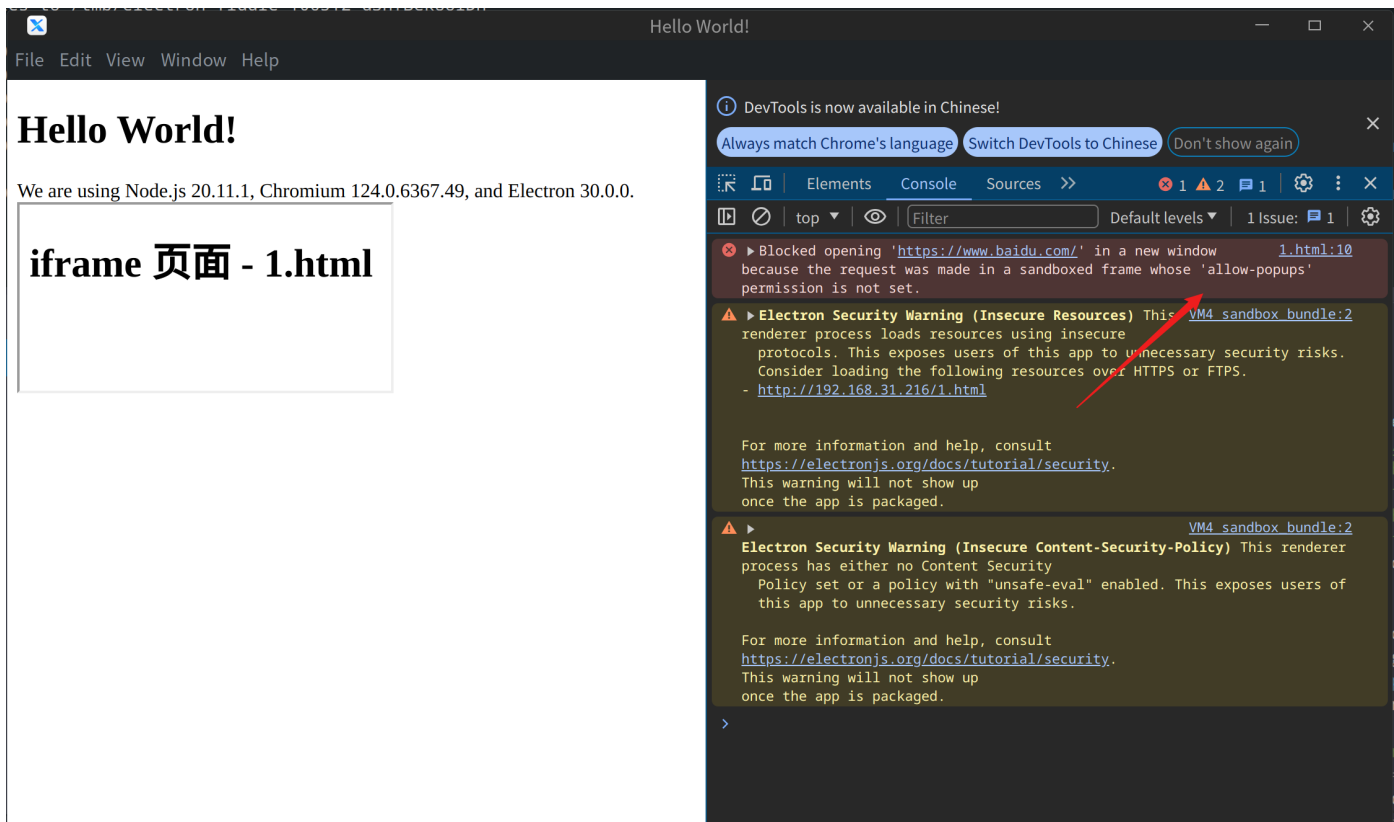


执行测试

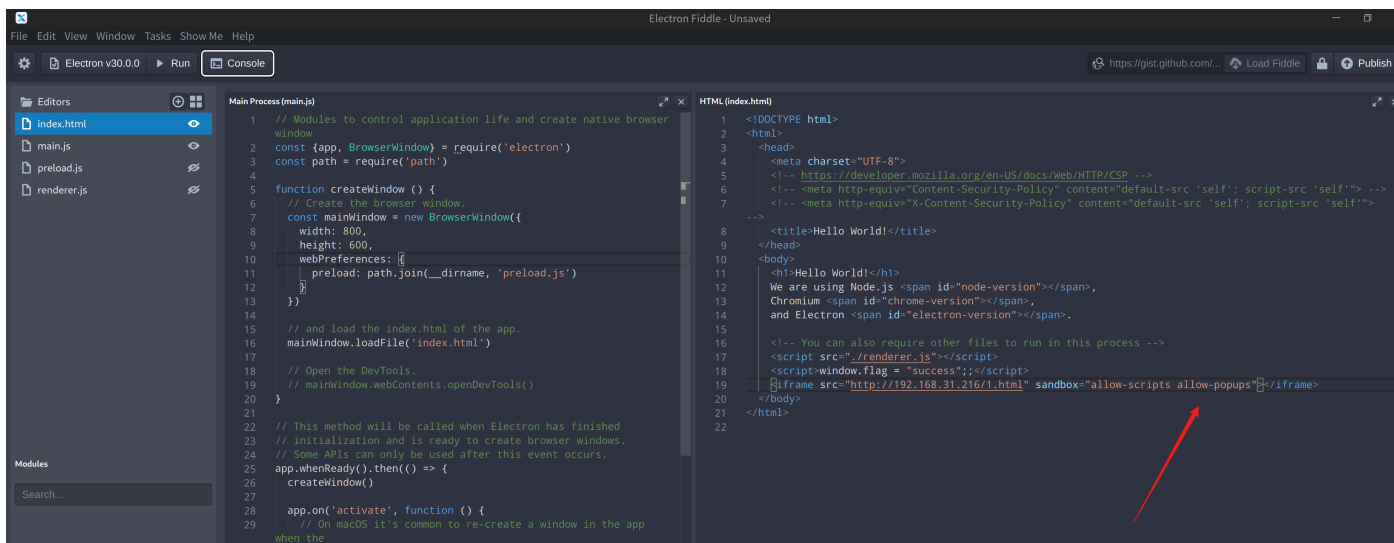


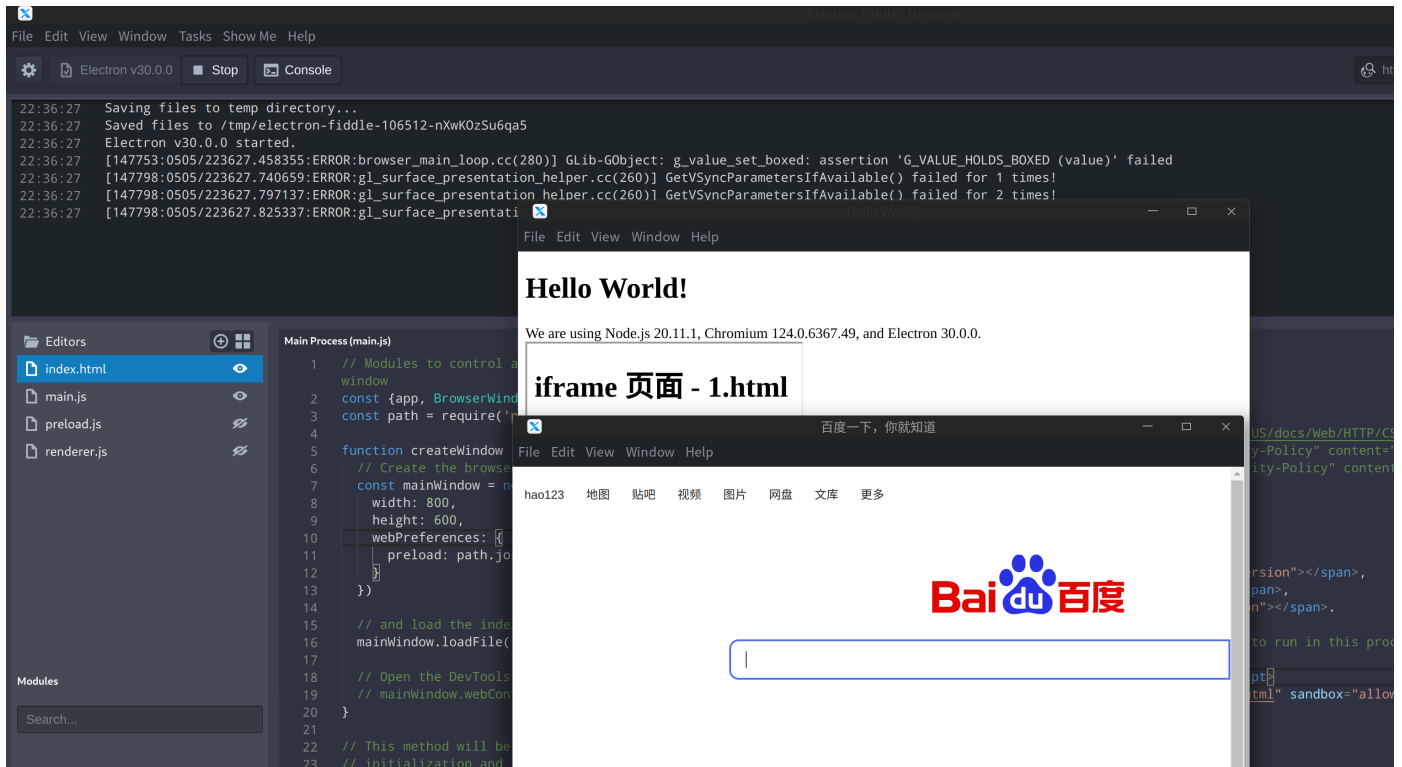
window.open 的执行被拦截，因为默认不允许执行 JavaScript ，我们加上 allow-scripts





window.open 的执行还是被拦截了，我们添加 allow-popups





成功打开百度的页面

`allow-popups-to-escape-sandbox` 是让新窗口创建时，不会自动继承 `iframe` 的 `sandbox` ，这可能会放宽安全措施

`allow-same-origin` 允许同源策略，可能部分朋友就蒙了，这些 `sandbox` 的选项不是在默认的限制中启用特权吗？怎么还来了一个允许同源策略呢？本来不就应该允许同源策略吗？

默认情况下，当一个 `<iframe>` 使用了 `sandbox` 属性而没有特别指定 `allow-same-origin` 时，该 `<iframe>` 中的文档会被视为来自一个独特的、无权限的源，即使实际上它与包含页面同源。这意味着即便内外页面同源，它们也不能直接互相访问DOM、Cookie或者使用localStorage等存储。

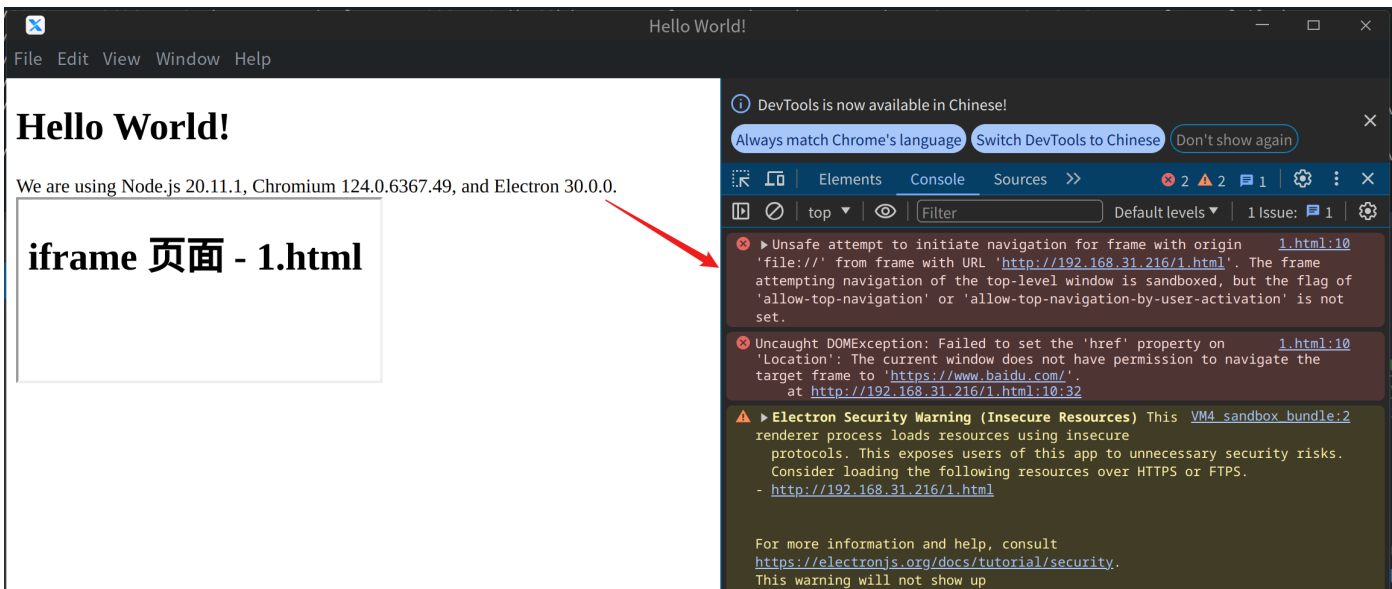
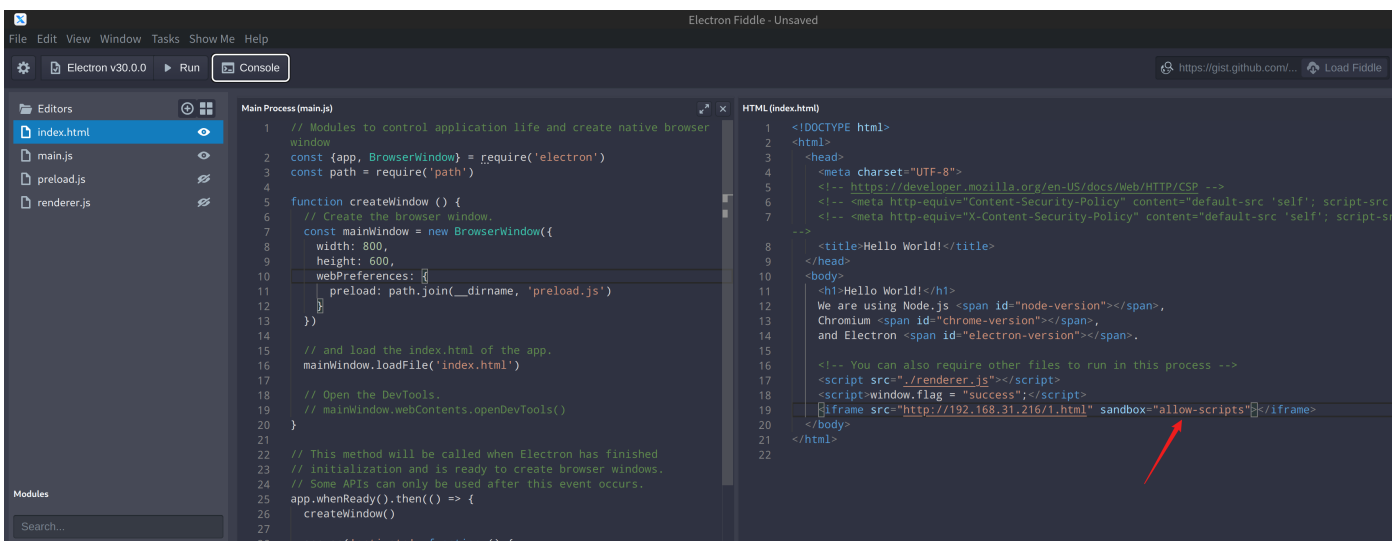
`allow-top-navigation` 比较重要，它用于控制嵌入在 `<iframe>` 中的页面是否有权限导航其顶层浏览上下文（即改变父窗口或顶级窗口的location）。默认情况下，当 `sandbox` 属性被应用时，这样的导航行为是被严格禁止的，以防止嵌入的内容对用户界面进行未经许可的修改，比如重定向主页面到恶意站点。

默认情况下，设置顶层导航 `window.top.location.href = "https://www.baidu.com/"`

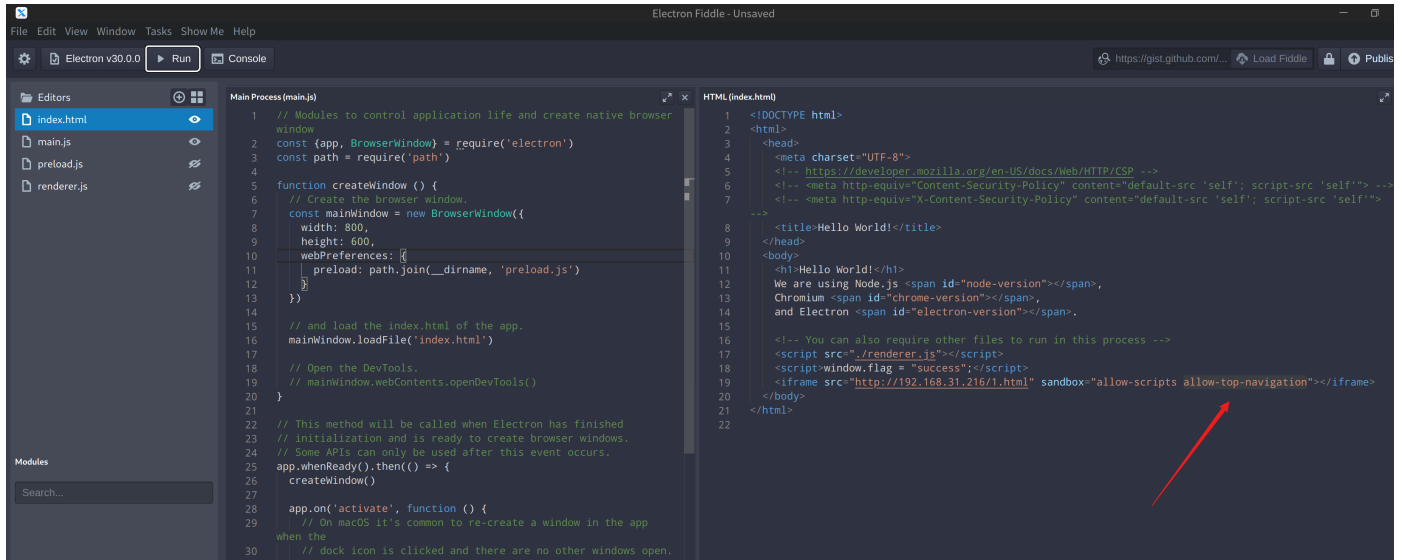
```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  |  <meta charset="UTF-8">
5  </head>
6  <body>
7  |  <div>
8  |  |  <h1>iframe 页面 - 1.html</h1>
9  |  |  <script>
10 |  |  |  window.top.location.href = "https://www.baidu.com/";
11 |  |  |  location.reload();
12 |  |  </script>
13 |  </div>
14 </body>
15 </html>
16
17

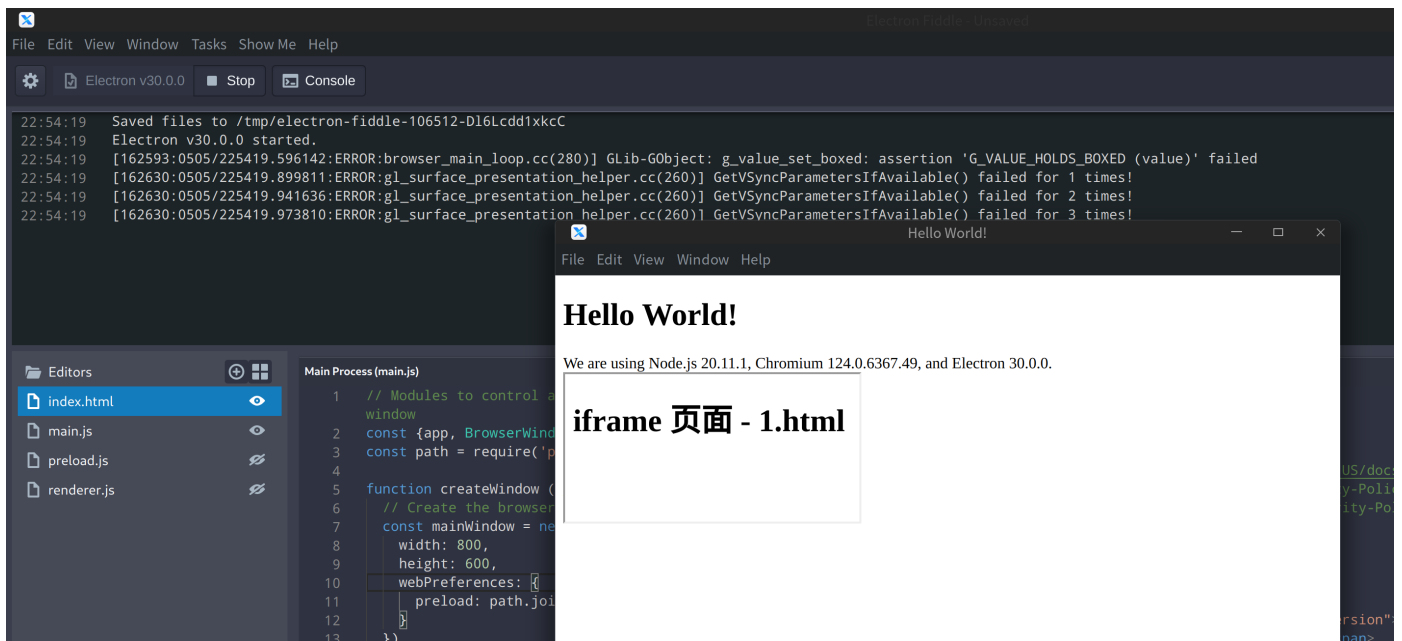
```



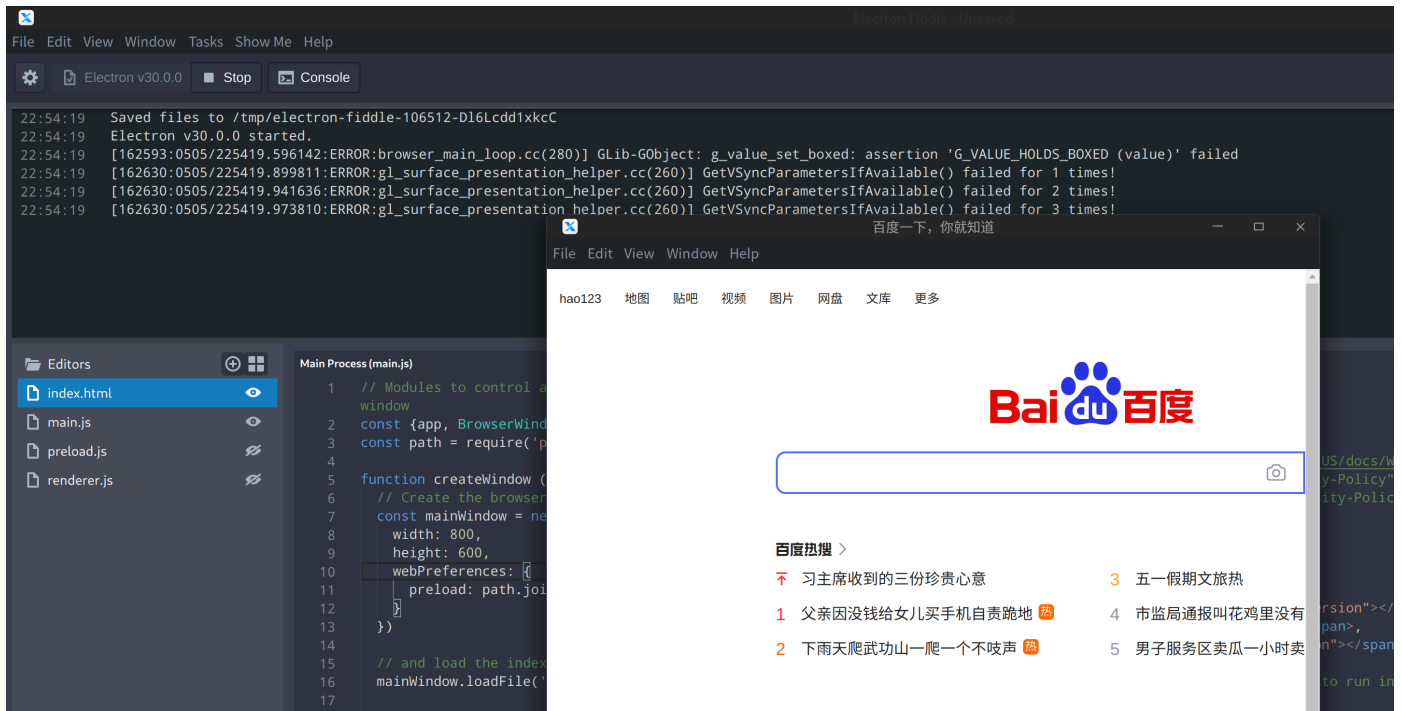
设置导航失败， sandbox 属性中添加 allow-top-navigation



再次执行



一瞬间以后完成页面重载

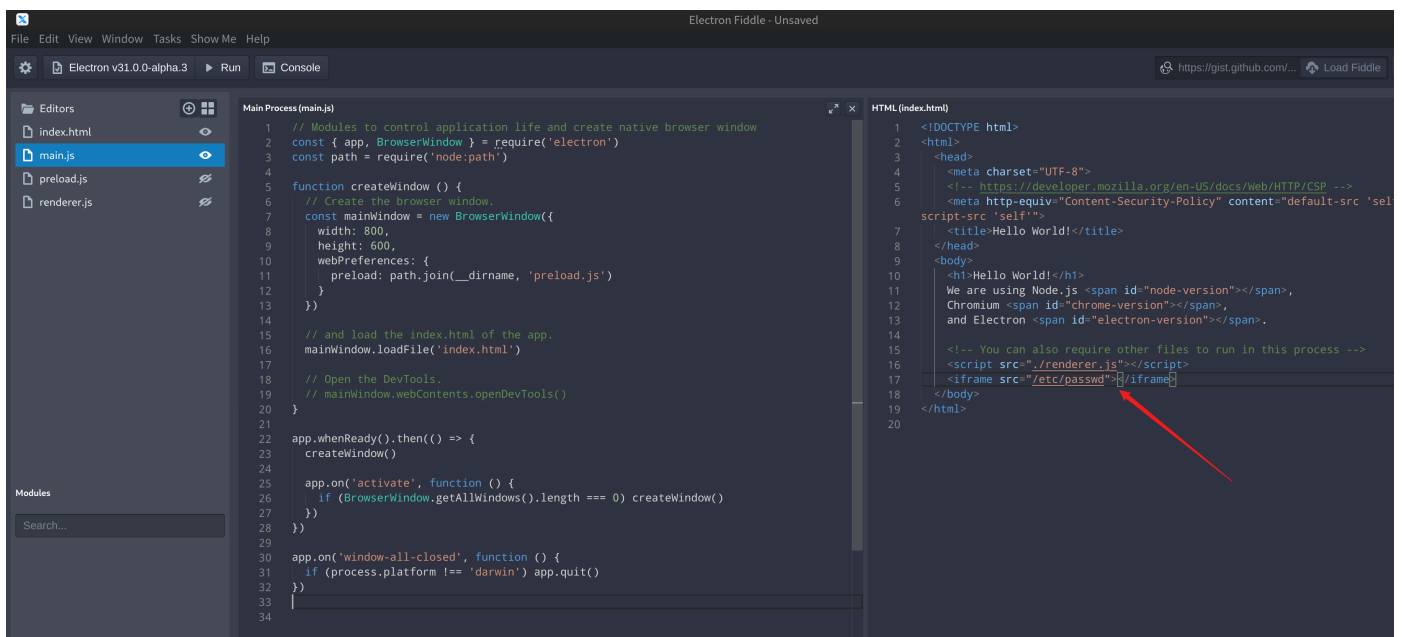


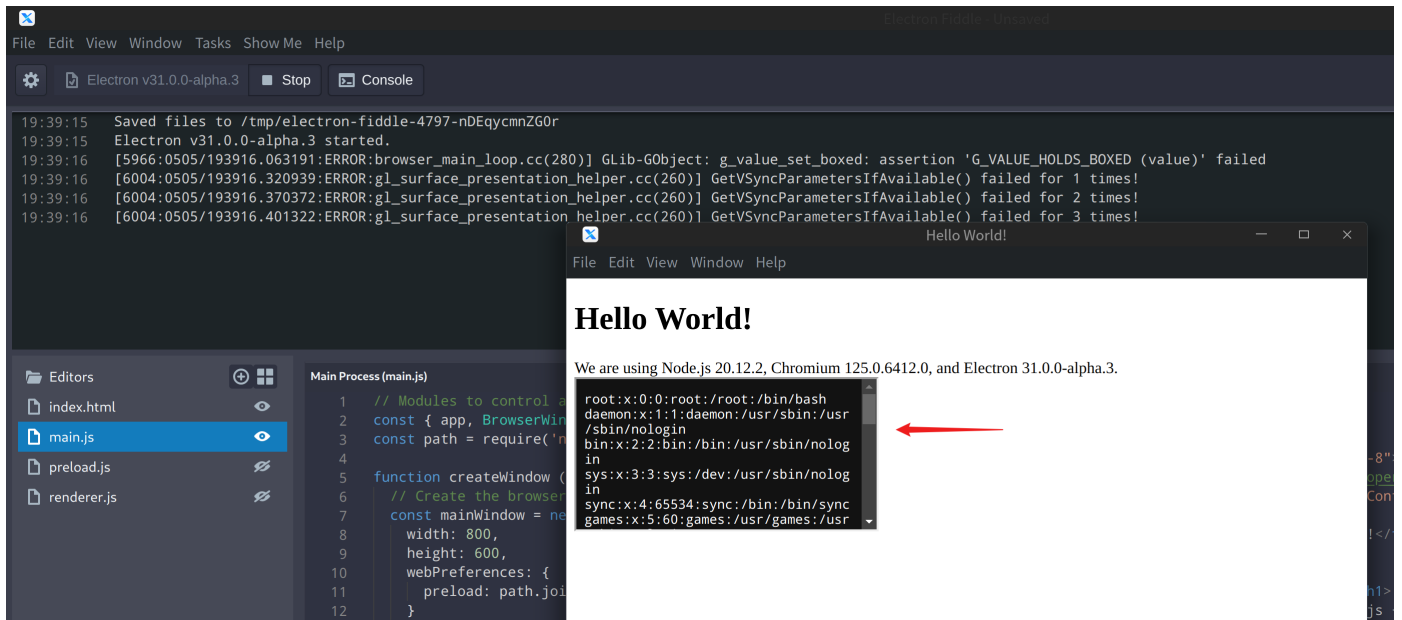
9) src

被嵌入的页面的 URL 地址

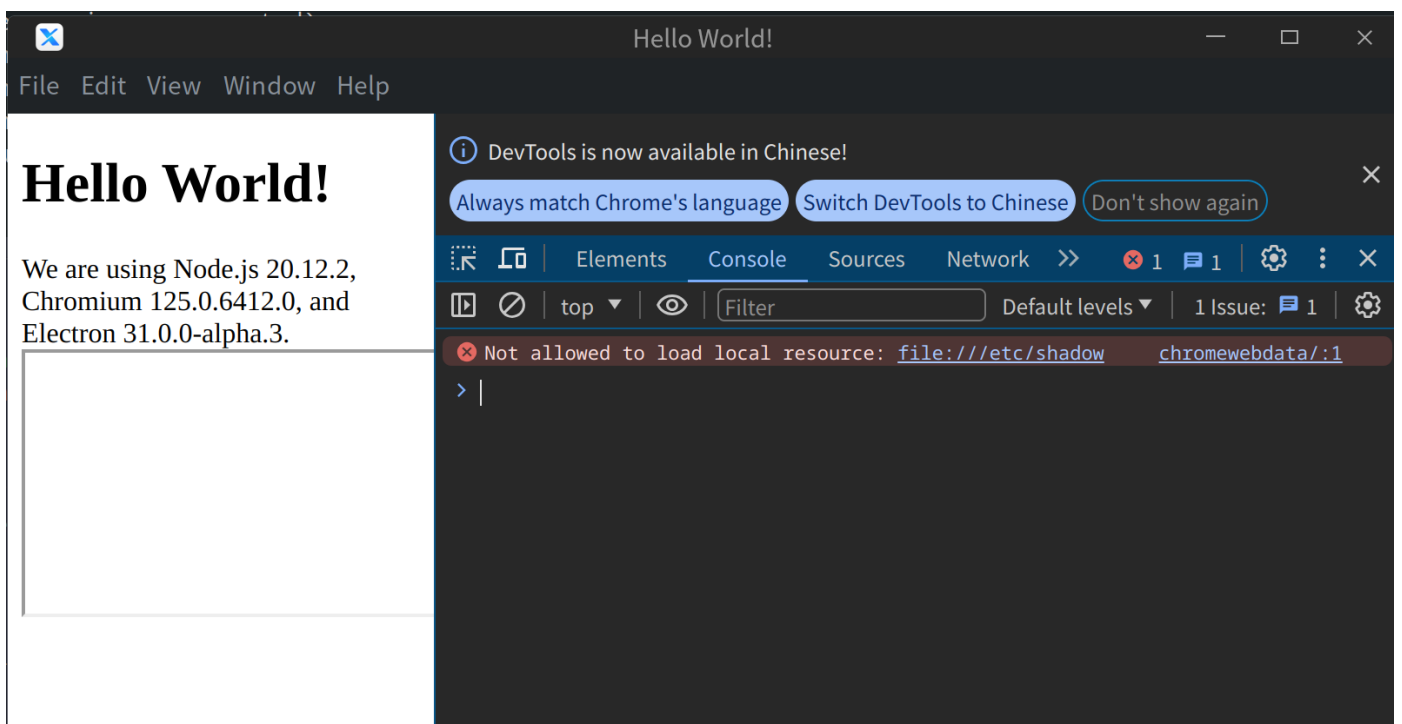
使用 `about:blank` 值可以嵌入一个遵从同源策略的空白页。在 Firefox (version 65 及更高版本)、基于 Chromium 的浏览器、Safari/iOS 中使用代码移除 `iframe` 的 `src` 属性 (例如通过 `Element.removeAttribute()`) 会导致 `about:blank` 被载入 frame。

对我们来说, 比较重要的是 `src` 属性是否可以打开本地文件, 是否会造成二进制文件等执行



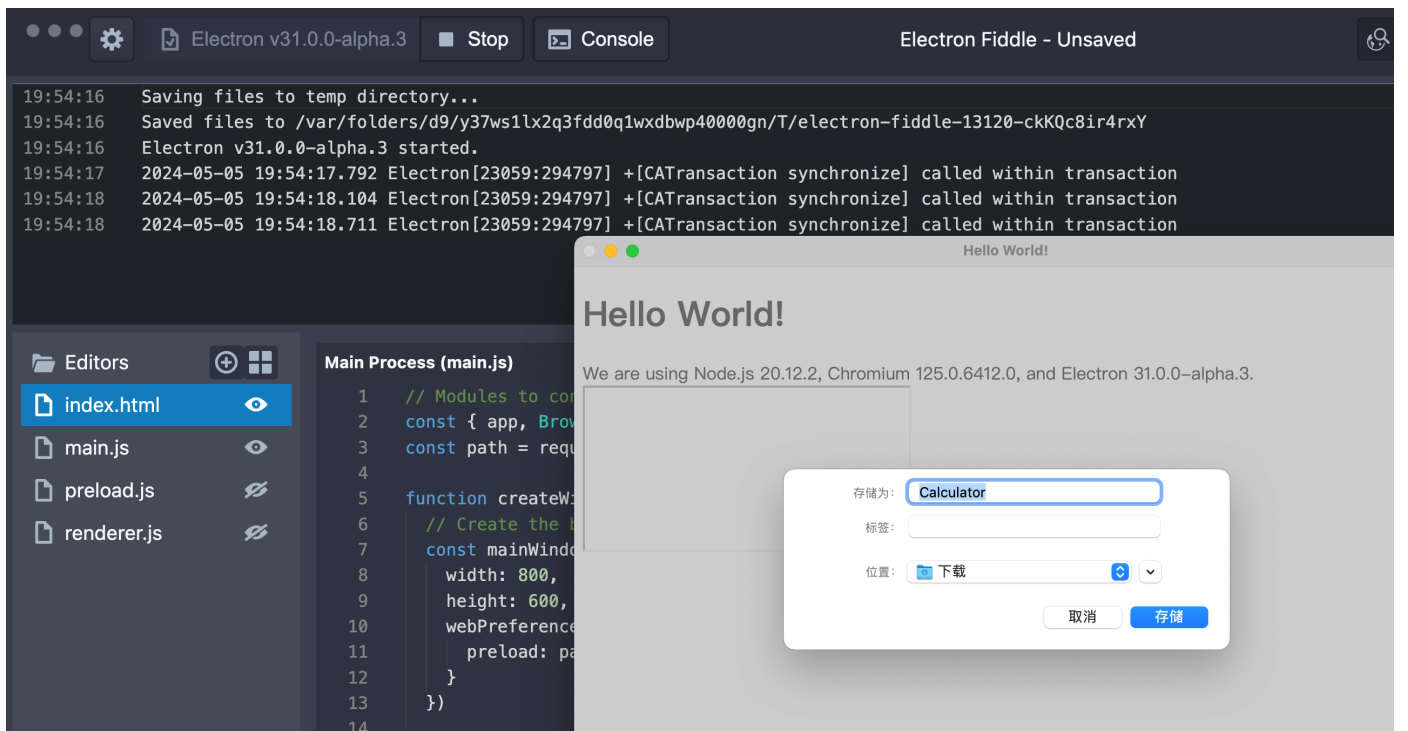
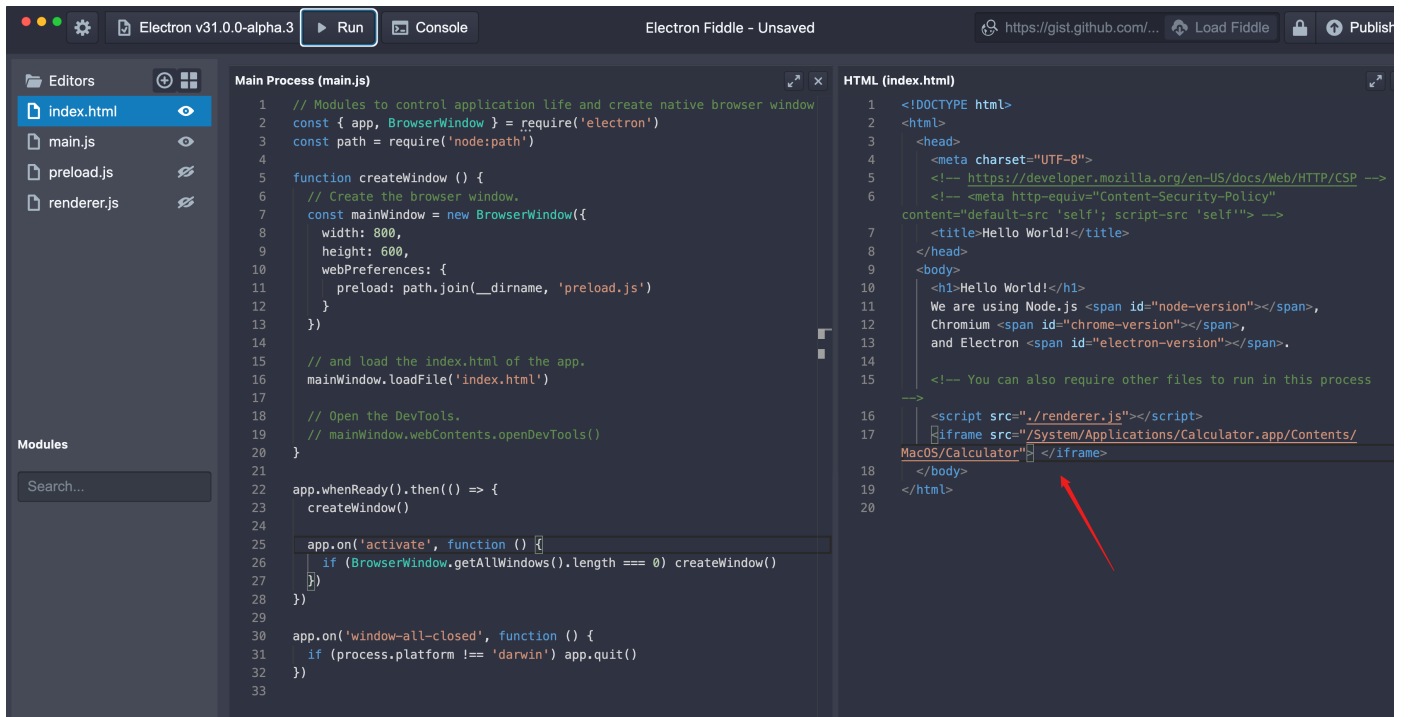


Electron 中 `iframe` 的 `src` 属性可以使用本地文件 (可以加上 `file://`), 当然文件要在权限之内, 例如读取 `/etc/shadow` 就会失败



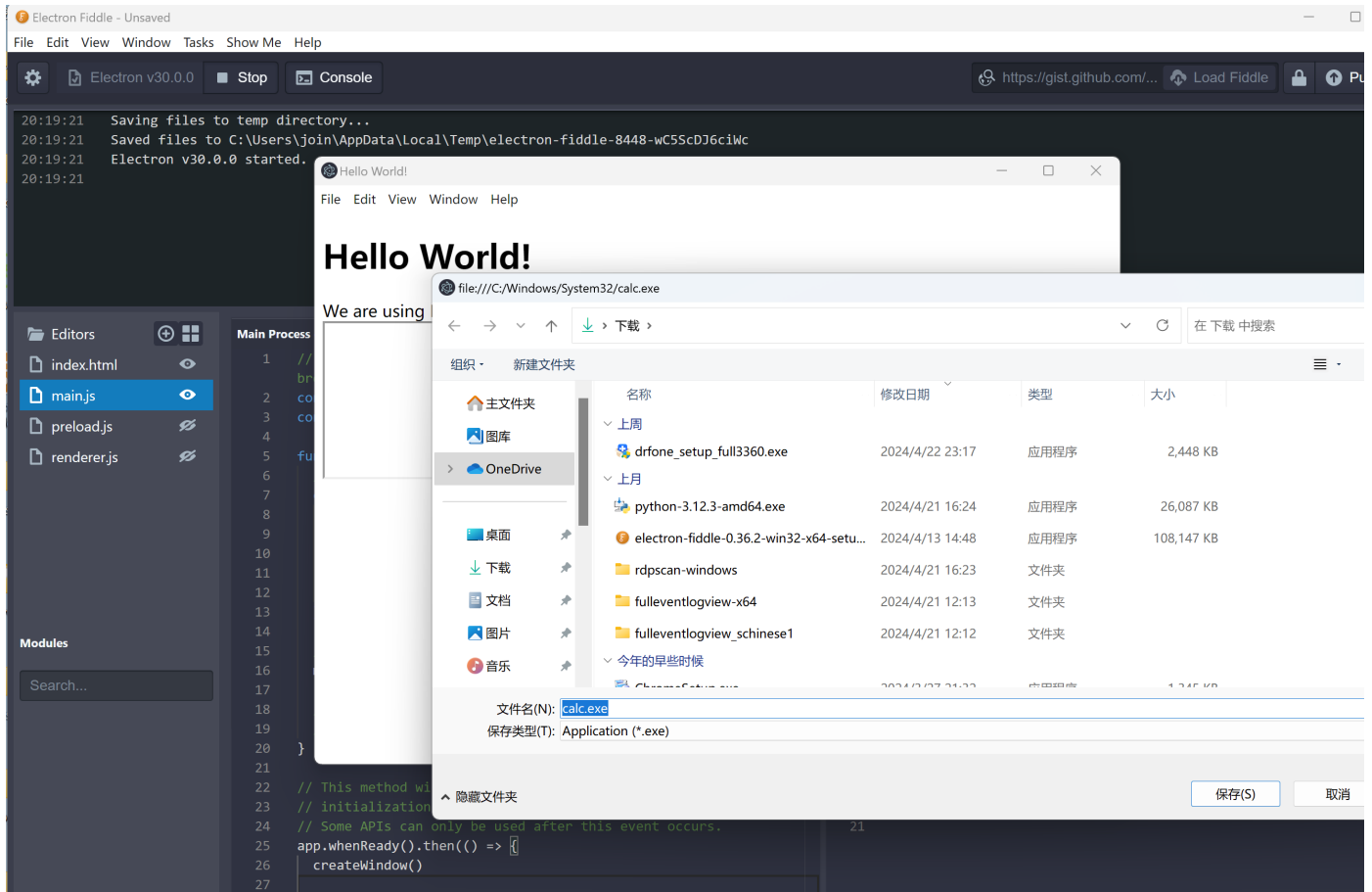
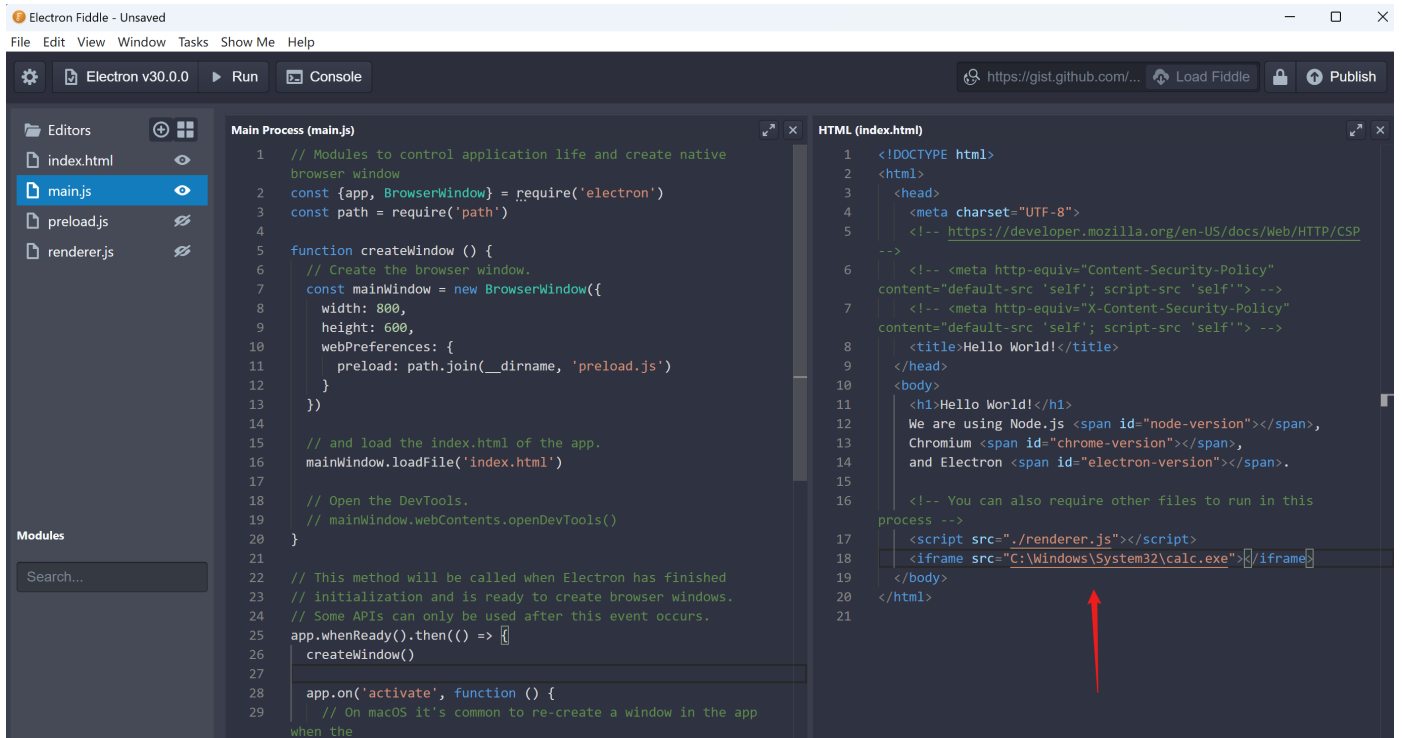
测试一下是否可以触发二进制可执行程序的执行

Deepin Linux



结果与 Deepin Linux 一致

Windows 11



Deepin Linux、MacOS、Windows 11 上不同版本 Electron 表现一致，均为下载，而不是执行

10) srcdoc

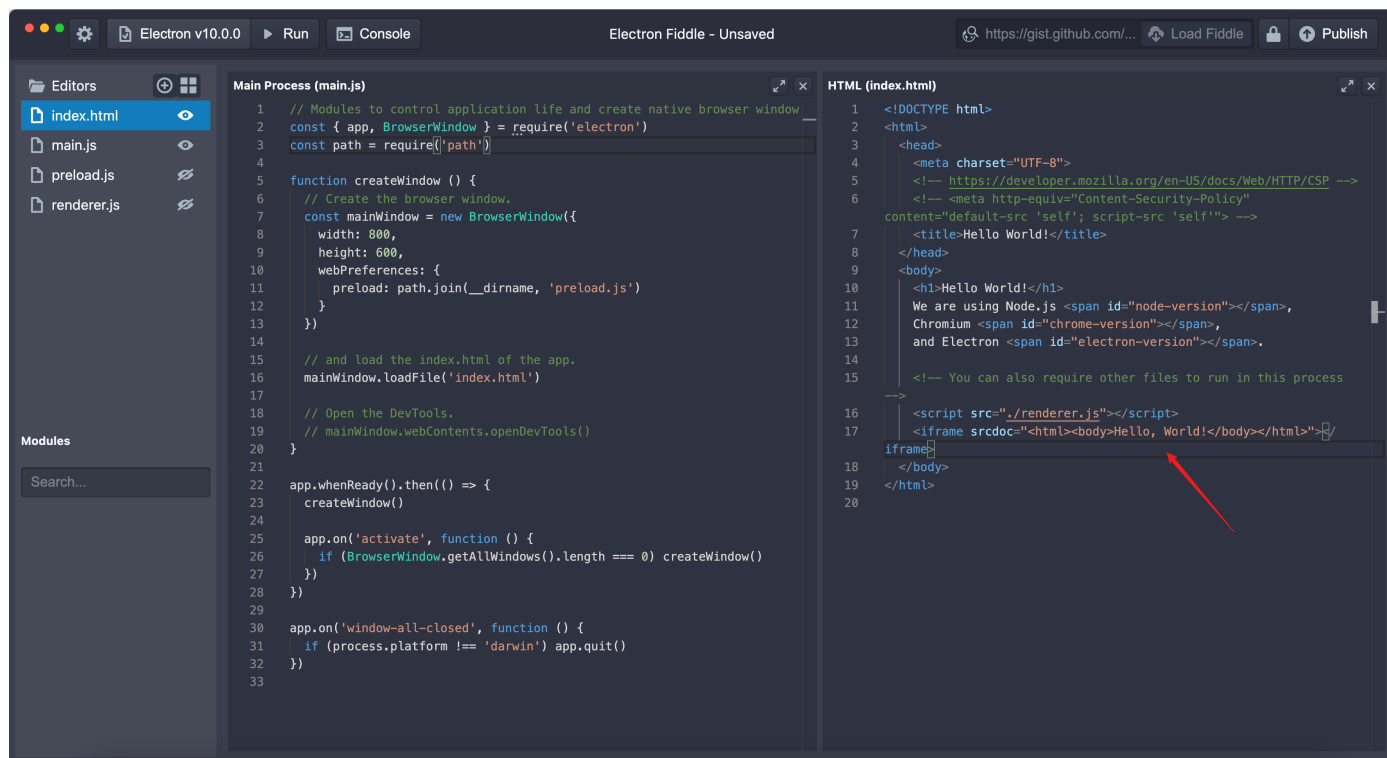
该属性是一段 HTML 代码，这些代码会被渲染到 iframe 中。如果浏览器不支持 `srcdoc` 属性，则会渲染 `src` 属性表示的内容。

有了 `src`，为何还要有一个 `srcdoc`，甚至 `srcdoc` 中的内容可以直接被放到 `iframe` 中渲染，这多少有些奇怪，而且 `srcdoc` 属性还是一个相对新的属性，不是说历史遗留问题

`srcdoc` 相比于 `src` 的一个优势是不需要跨域，实际上就是一段 HTML 代码直接嵌入到 `iframe` 中，而不是让浏览器去加载一个外部的 URL

我们使用 `Electron` 测试一下

```
<iframe srcdoc="<html><body>Hello, World!</body></html>"></iframe>
```





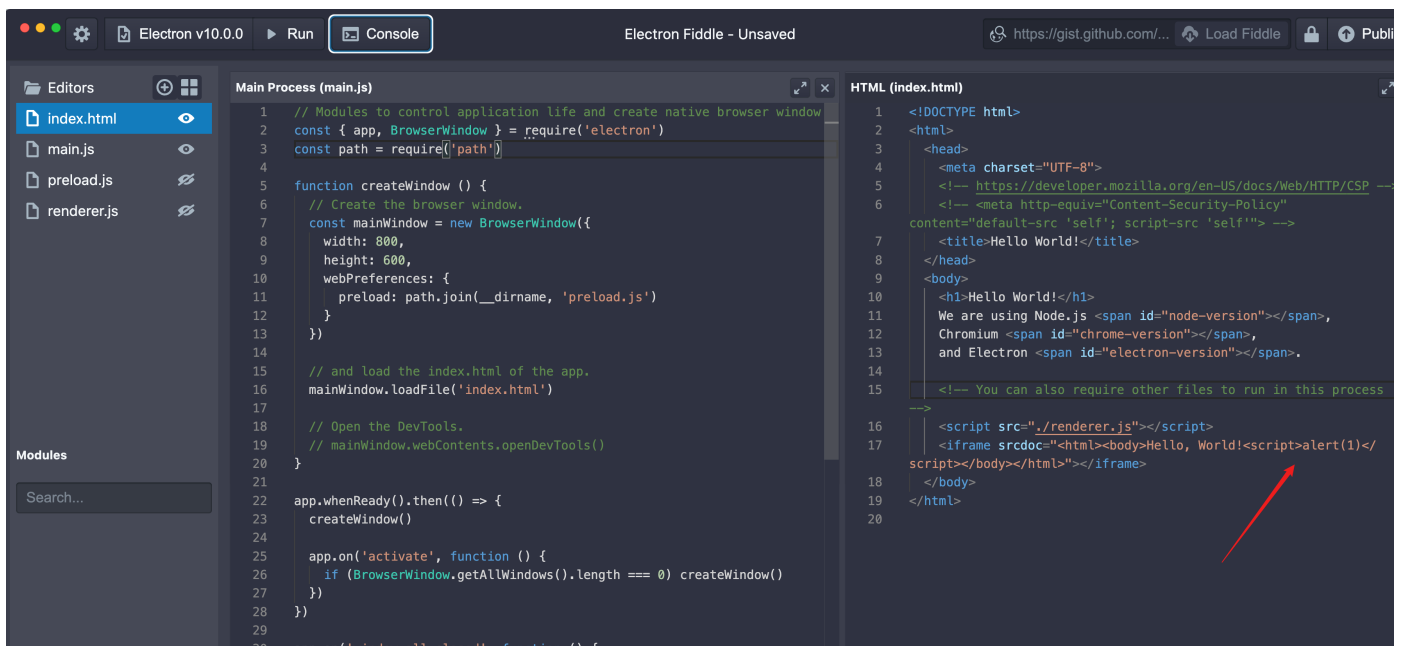
Hello World!

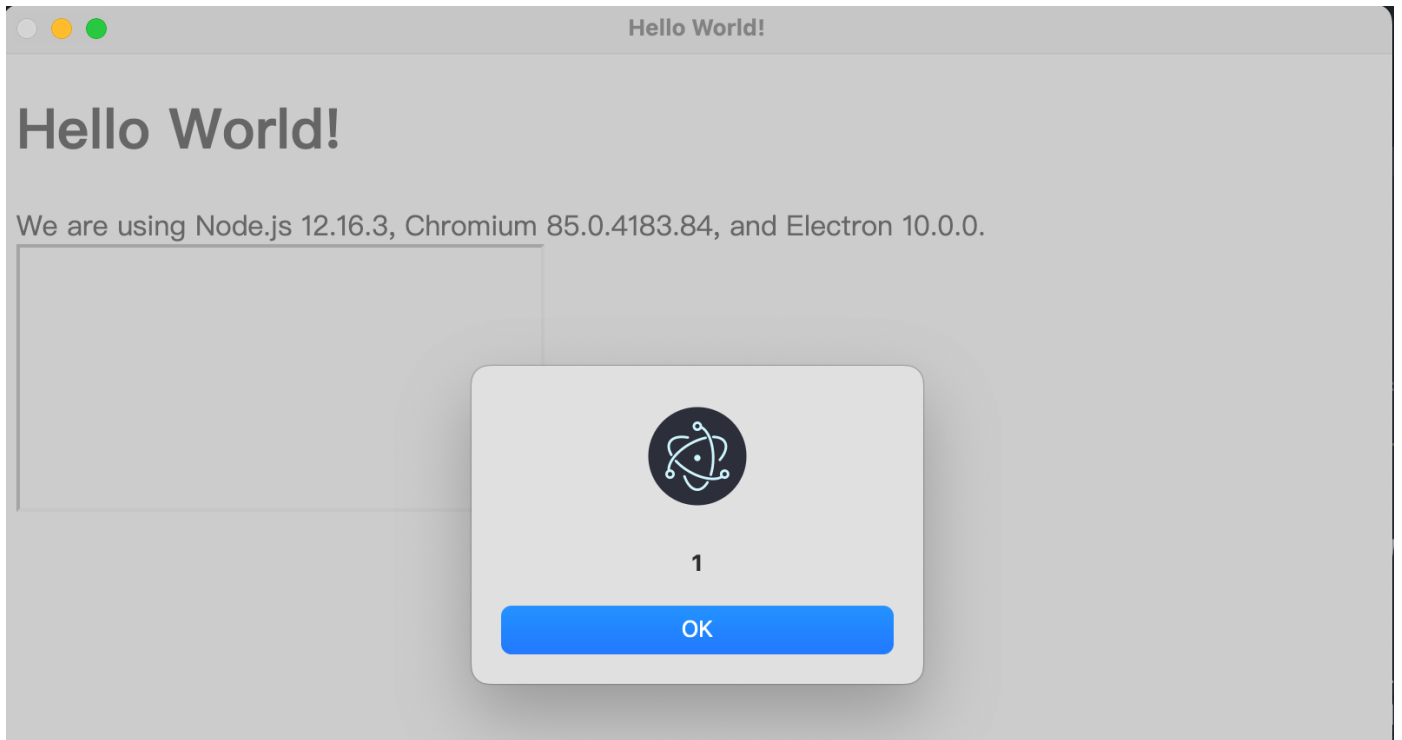
We are using Node.js 12.16.3, Chromium 85.0.4183.84, and Electron 10.0.0.

Hello, World!

Electron 是支持该语法的，在之前的 Electron 与你我息息相关的文章中其实就已经介绍了这个熟悉实现 RCE 等利用的内容

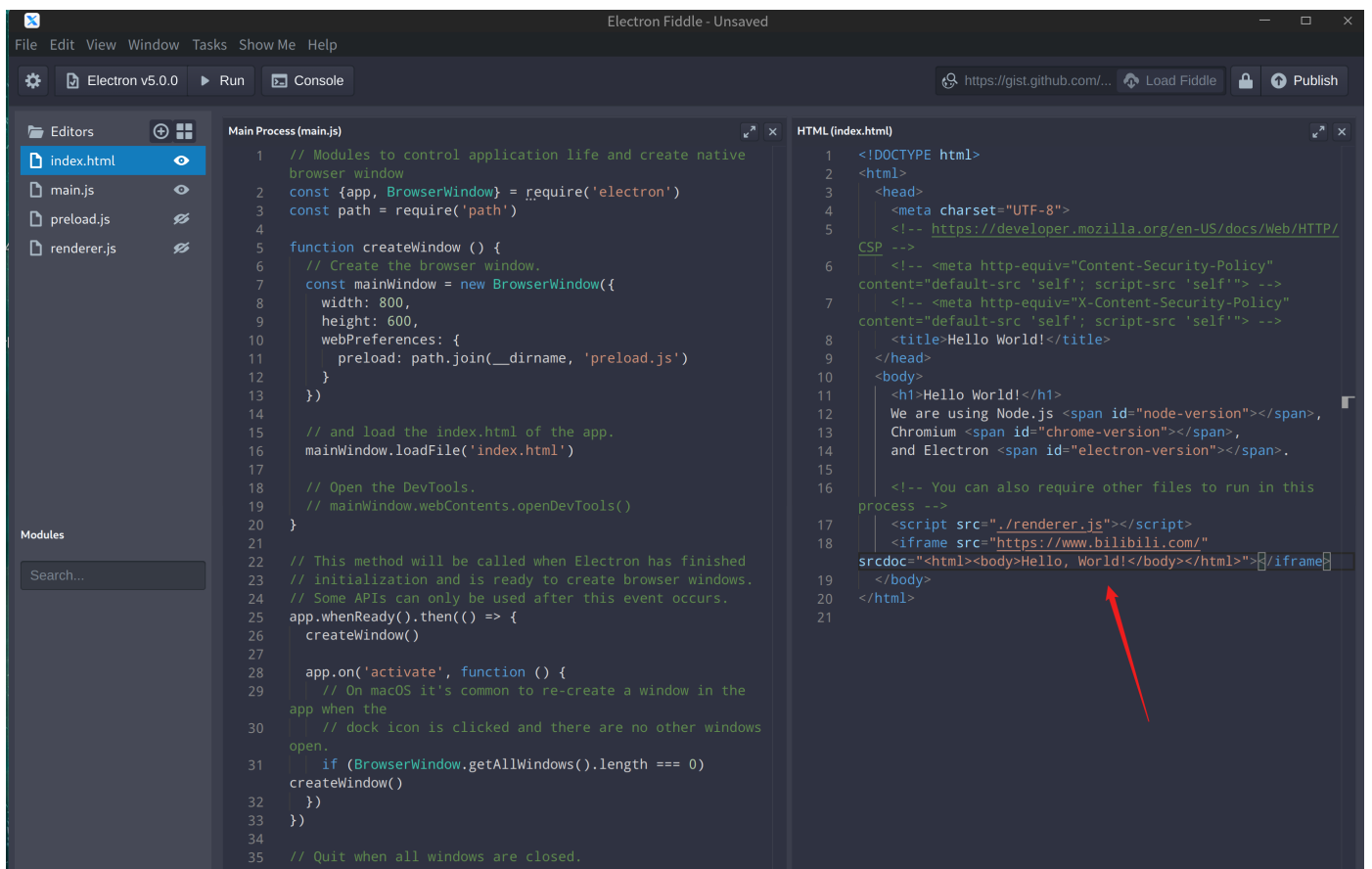
这里的 JavaScript 也是可以执行的

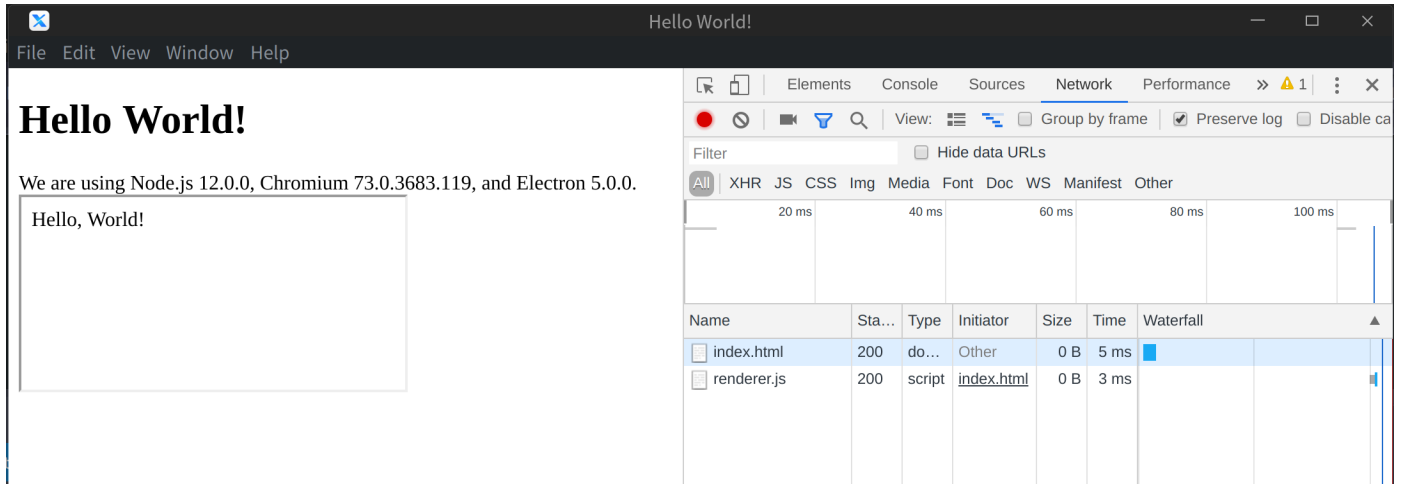




如果同时设置了 `src` 和 `srcdoc` 会怎么样

```
<iframe src="https://www.bilibili.com/" srcdoc="<html><body>Hello, World!</body></html>"></iframe>
```





看来两者同时存在时以 `srcdoc` 优先，浏览器不支持 `srcdoc` 时才使用 `src`

11) width

以 CSS 像素格式，或以像素格式，或以百分比格式指定的 frame 的宽度。默认值是 `300`

接下来的内容是不赞成使用的属性，可能不被所有的浏览器支持

12) align

此元素相对于周围元素的对齐方式

13) frameborder

值为 `1`（默认值）时，显示此框架的边框。值为 `0` 时移除边框。此属性已不赞成使用，请使用 CSS 属性 `border` 代替

14) longdesc

表示框架内容的长描述的 URL。由于广泛的误用，该属性对于无图形界面的浏览器不起作用

从网络层面看，似乎 `Electron` 是不支持该属性的，几乎所有主流浏览器都不支持这个属性

15) marginheight

这个属性定义了框架的内容距其上边框与下边框的距离，单位是像素

16) marginwidth

这个属性定义了框架的内容距其左边框和右边框的距离，单位是像素

17) scrolling

这个属性控制是否要在框架内显示滚动条，允许的值包括：

- auto: 仅当框架的内容超出框架的范围时显示滚动条
- yes: 始终显示滚动条
- no: 从不显示滚动条

2. 渲染页面与 iframe 通信

这分为两种情况，渲染页面与 `iframe` 的地址同源和不同源

不同源的情况之前的文章就介绍过了，使用 `postMessage` 和 `onMessage` 进行通信

对于同源的情况，渲染进程访问 `iframe` 内变量的方式如下

`iframe` 页面设置变量

```
window.flag = "strings for iframe"
```

渲染进程可以使用 `iframe` 的 `name` 属性或者序号来获取 `iframe` 内部内容的 `window`，因为我们只有一个 `iframe`，所以序号为 0

```
setTimeout(function() {  
    console.log(window.frames[0]['flag']);  
}, 1000);
```

这里设置了 1 秒的延时，保证 `iframe` 那边设置变量完成

这样就可以直接获取到 `iframe` 中的 `window` 对象了

同源的情况下 `iframe` 访问渲染进程就更简单了

渲染进程设置变量

```
window.abc = "abc"
```

`iframe` 内通过以下代码获取到变量

```
window.parent.abc
```

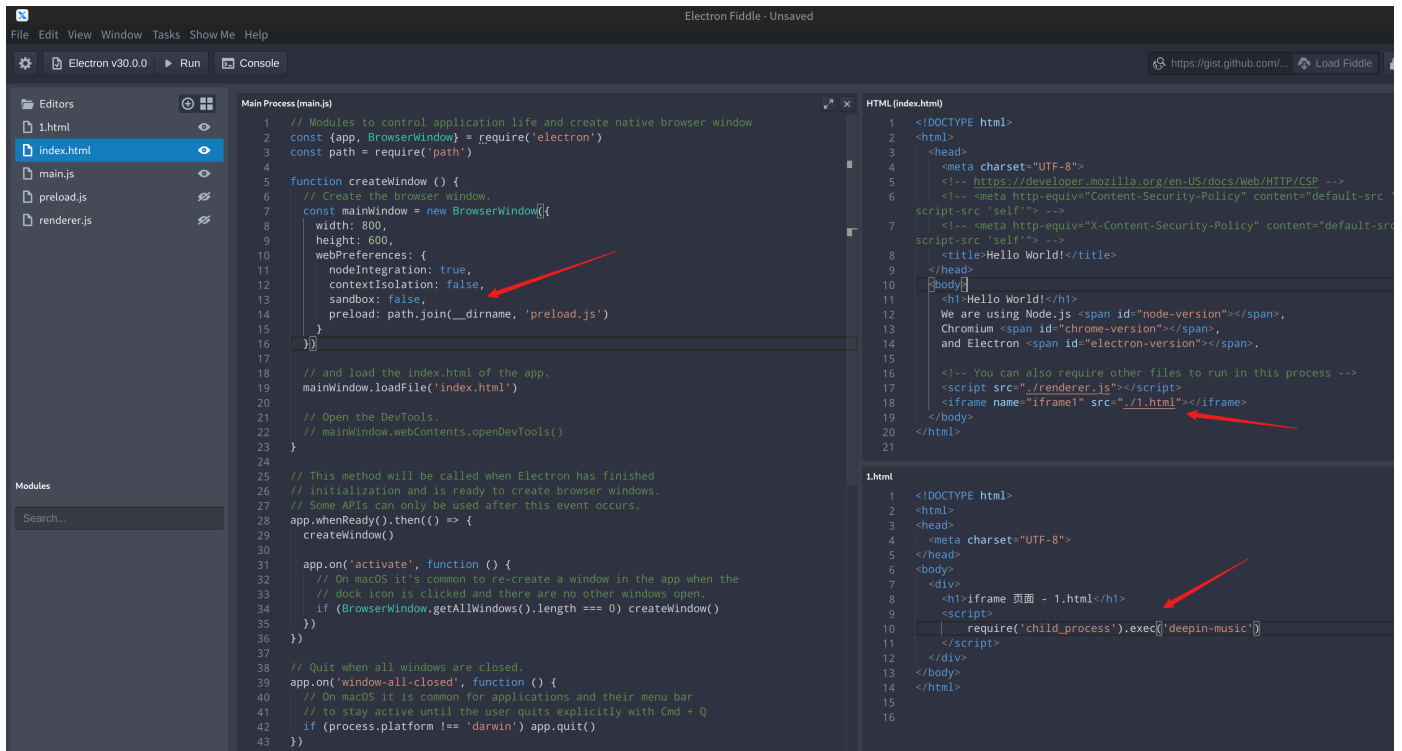
这里需要注意一点，大多数渲染进程的窗口是通过加载本地文件创建的，

- 本地文件创建的主窗口与 加载 `http(s)` 页面的 `iframe` 是不同源的
- 本地文件创建的主窗口与加载本地文件的 `iframe` 是同源

3. `iframe` 执行 `Node.js` 的情况

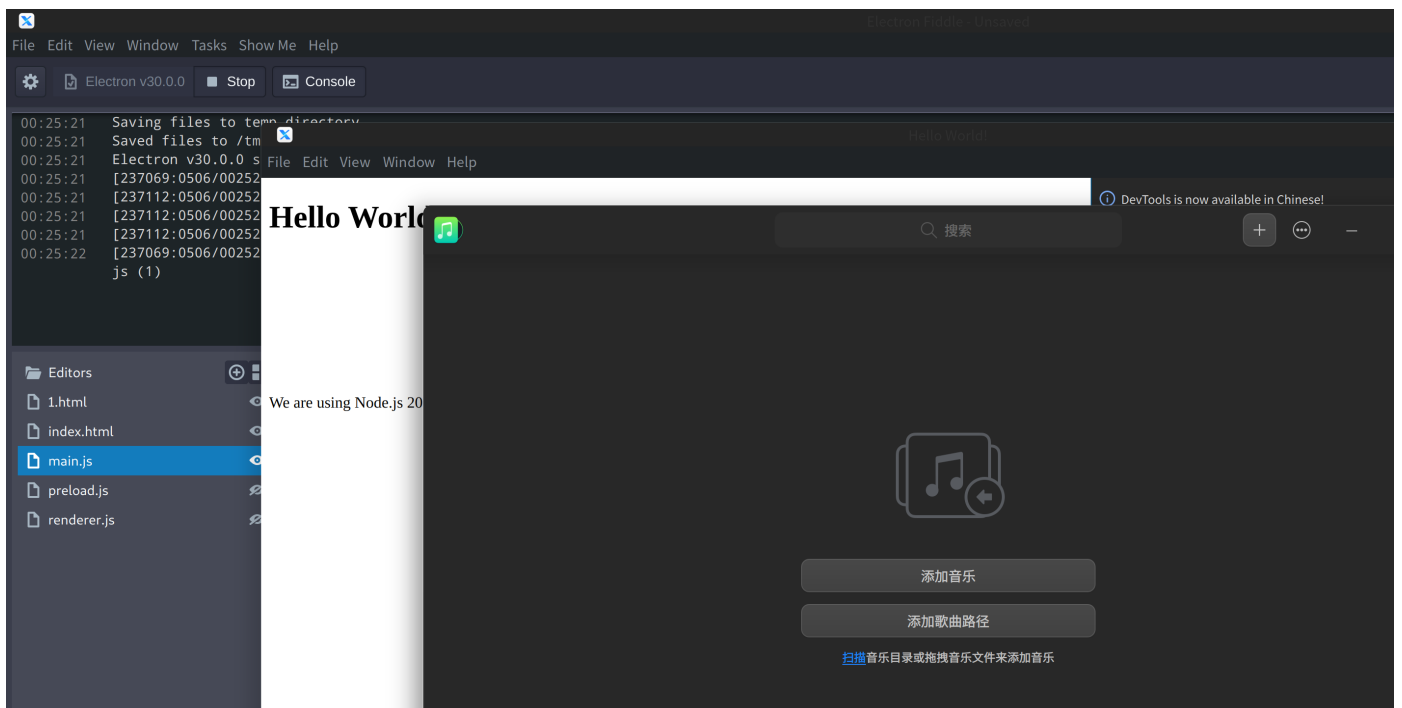
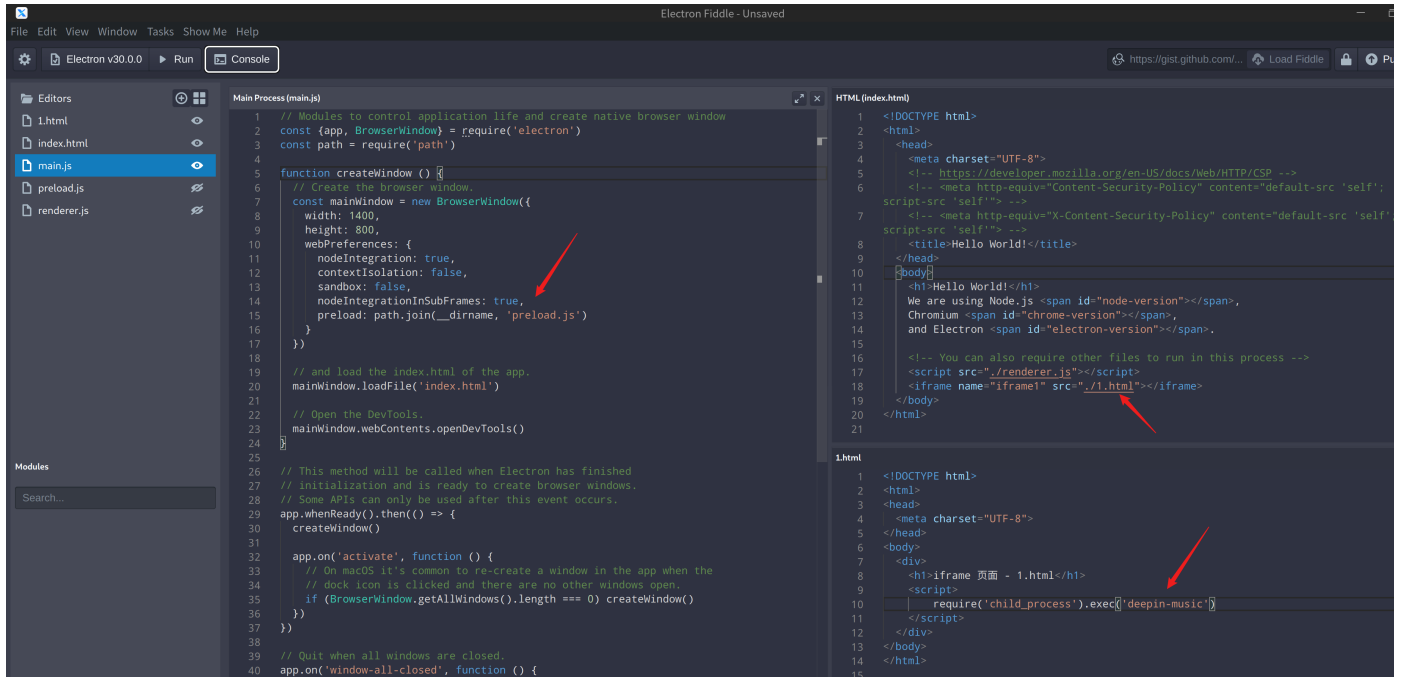
从上面的测试来看，`iframe` 肯定是不能直接执行 `Node.js`，我们先来设置一下安全三大件

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



很遗憾，三大件还不够，再加一个 `nodeIntegrationInSubFrames: true`

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



成功执行，那么是否可以缺少四个安全配置中的一个或几个呢？

经过测试，可以缺少 `sandbox: false` ,但是不能设置 `sandbox: true` ，默认配置是可以执行的，具体为什么参照 [sandbox | Electron 安全](#) 这篇文章

当大家看到这篇文章的时候，上面提到的 `Electron` 三大安全配置应该都在公众号上发表过了，大家可以想一下，我就为了让渲染进程或者渲染进程中的 `iframe` 执行个 `Node.js` ，为什么一定要关闭上下文隔离呢？

其实不难理解，当设置了 `nodeIntegration` 为 `true` 的时候，其实就是单单给 `Preload` 脚本开放了不受限制的 `Node.js` API 访问能力，对于渲染进程的页面的上下文来说，是没有这个能力的

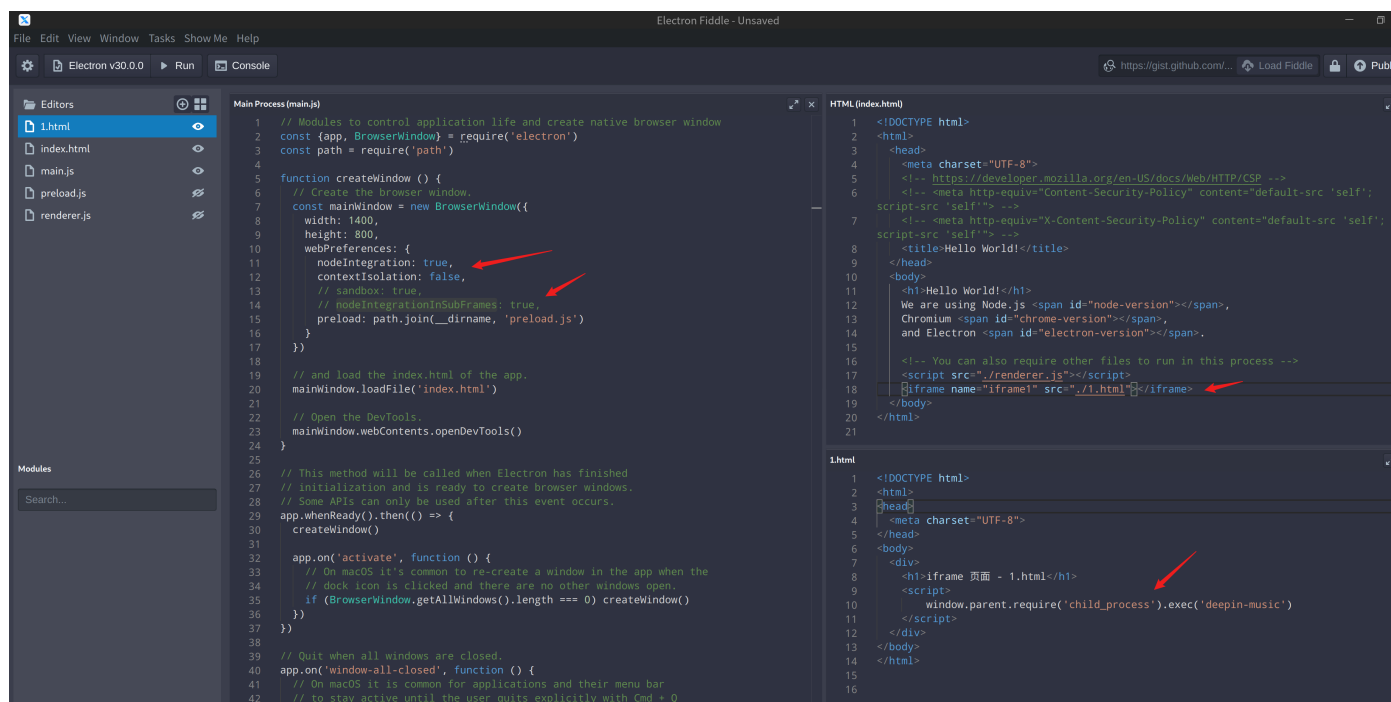
但是如果此时 `contextIsolation` 被设置为了 `false`，也就是关闭了上下文隔离，那么渲染页面就可以访问到 `Preload` 的上下文，获取到 `Preload` 脚本中的 `window.require` 了，进而可以直接调用 `require('child_process').exec('deepin-music')`

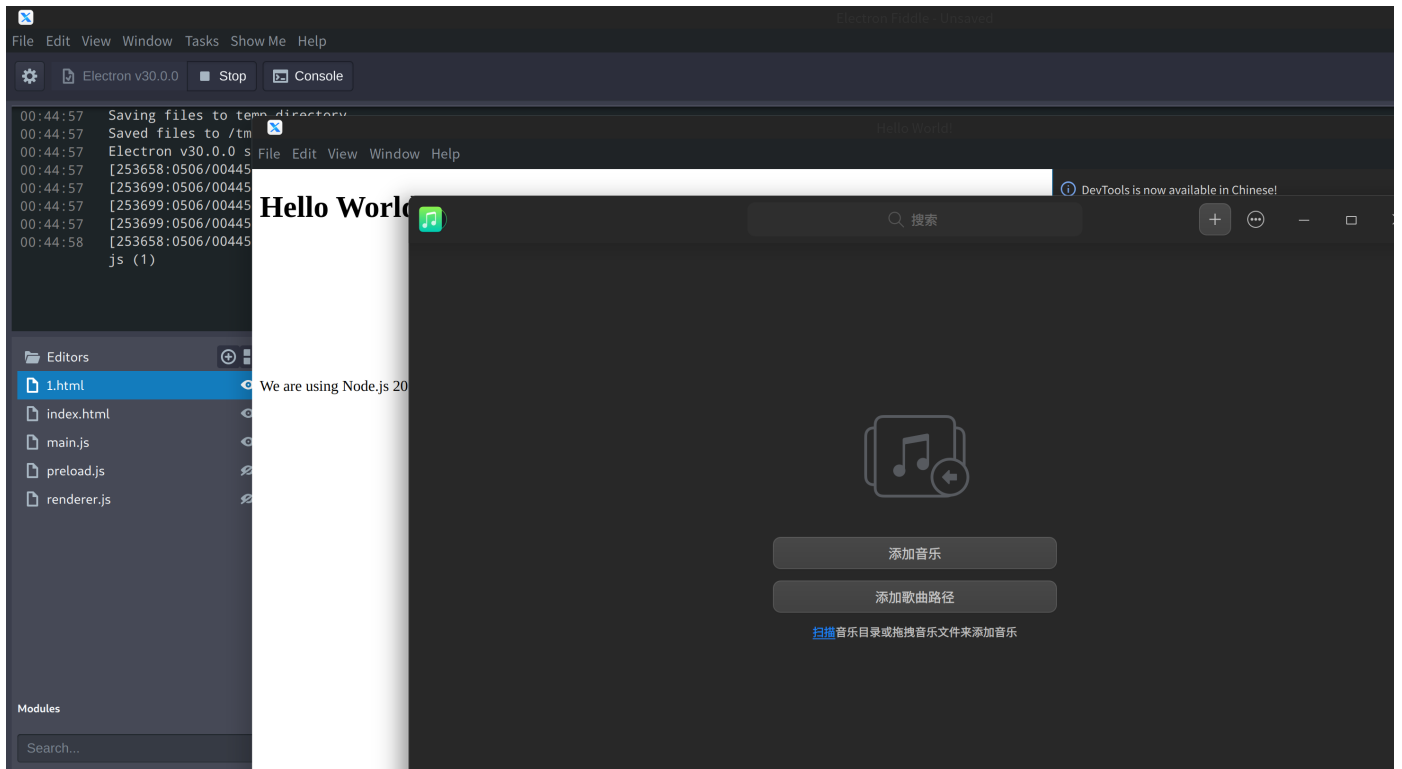
如果开启上下文隔离，之后通过 `contextBridge` 将 `Preload` 的 `require` 暴露给渲染页面，对于我们攻击来说效果是一样的，毕竟我们只用到 `require` 就够了

所以这里有些朋友可能会思考了，如果渲染进程页面和 `iframe` 的地址是同源的，那岂不是 `iframe` 内部直接可以通过下面的方式执行 `Node.js`

```
window.parent.require('child_process').exec('deepin-music')
```

这样的话，即使不开启 `nodeIntegrationInSubFrames: true` 是不是也可以执行 `Node.js` ？





成功执行 `Node.js` 代码

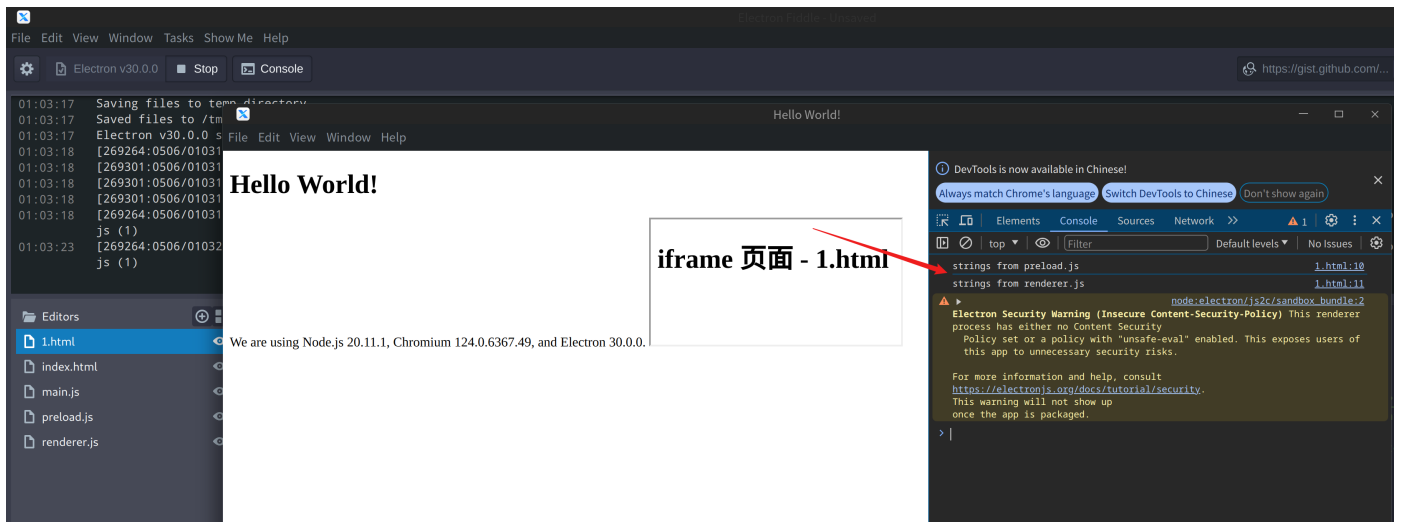
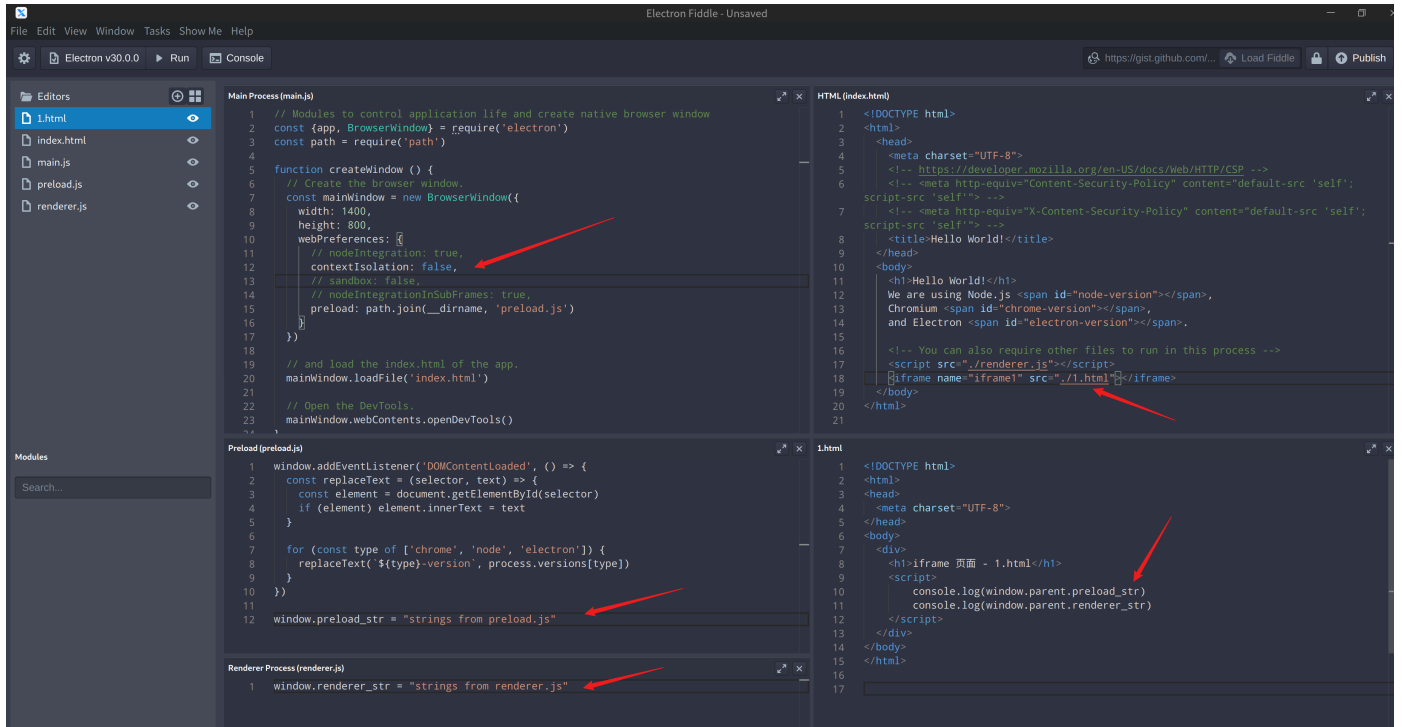
所以需要注意，不开启 `nodeIntegrationInSubFrames` 的情况下 `iframe` 内的代码也是可能可以执行 `Node.js` 的

4. `iframe` 上下文情况

通过上面的测试，这件事也比较清晰了，如果 `iframe` 的地址与渲染页面的地址不同源的话，那么 `iframe` 的上下文就是一个独立的上下文

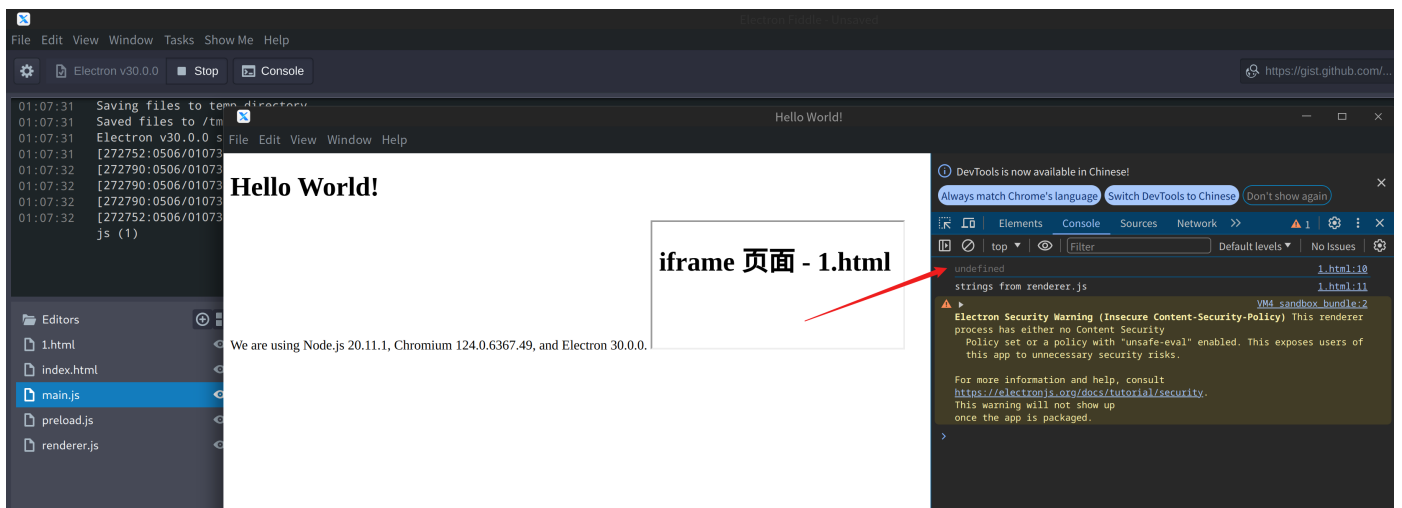
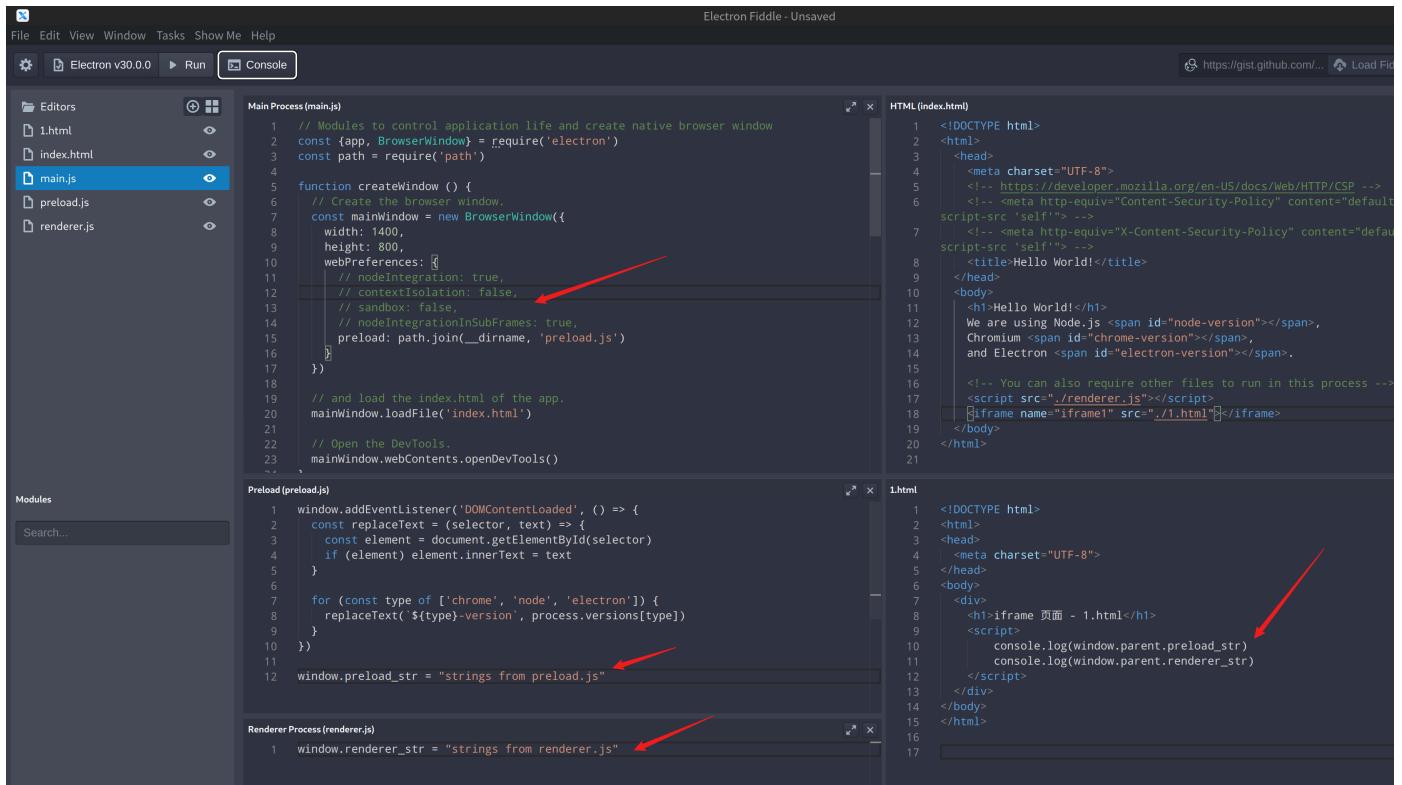
如果 `iframe` 的地址与渲染页面的地址同源，并且关闭了上下文隔离，`iframe` 就可以通过“找爹”的方式获取到渲染页面的上下文，这里有一个问题，既然关闭了上下文隔离，是不是说 `iframe` 就可以一路找上去，获取到 `Preload` 脚本中的上下文呢？

测试一下



果然可以，一路畅通

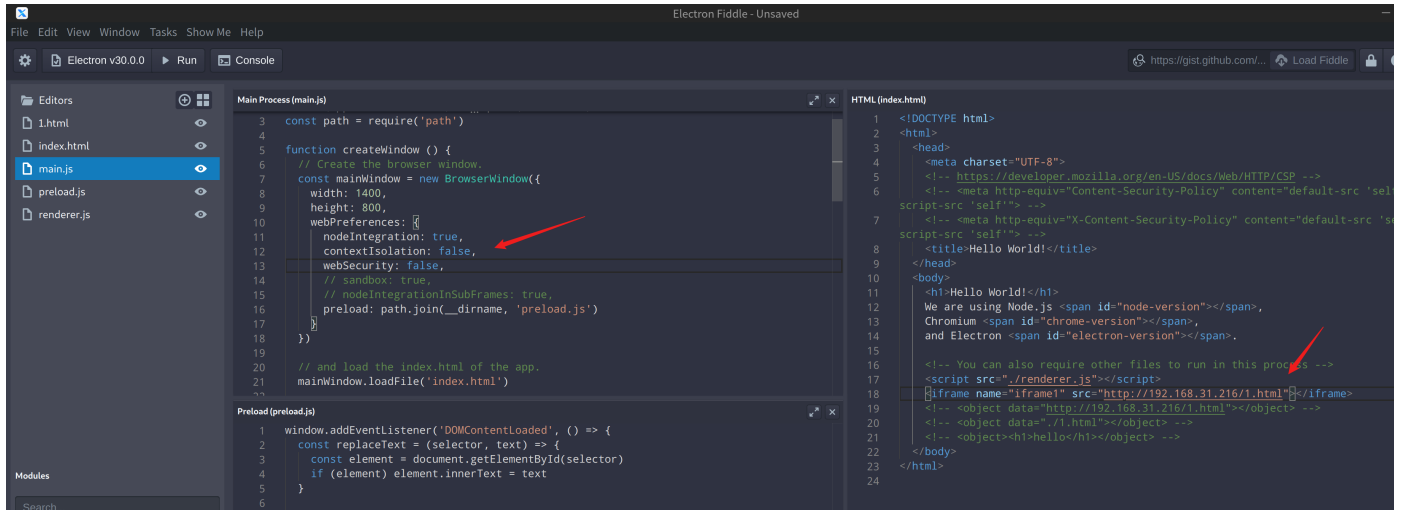
如果 `iframe` 的地址与渲染页面的地址同源，但是开启了上下文隔离呢？

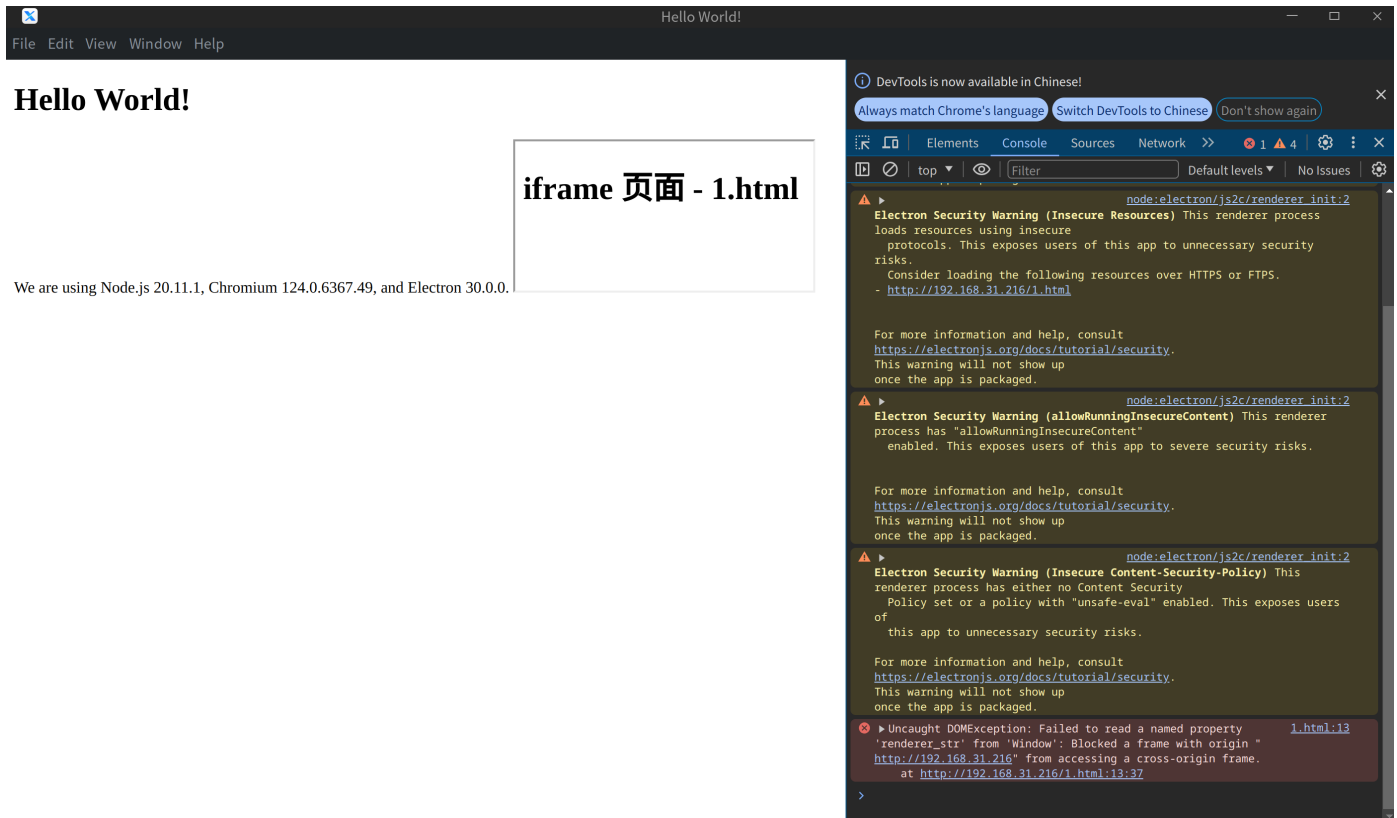


可以看到，此时 `iframe` 还是可以获取到渲染页面的上下文，但是无法获取到 `Preload` 脚本的上下文了

5. 关闭同源策略

通过设置 `webSecurity: false` 关闭同源策略，情况会有不同吗？





并不会有什变化

6. 小结

`iframe` 作为一种常用的嵌入方法，在 `Electron` 中也得到了很好的支持，`iframe` 支持 `sandbox` 属性，但是默认没有设置，需要显式地设置，`sandbox` 或 `sandbox=""` 表示开启所有限制，如果有特例允许的需求，可以在 `sandbox` 属性的值中设置，例如 `sandbox="allow-scripts"`

如果 `iframe` 的地址与渲染页面的地址同源，则可以相互直接通讯，并获取相互的上下文；如果非同源则不行，需要通过 `postMessage` 和 `onMessage` 进行通信

如果 `iframe` 的地址与渲染页面的地址同源，则在以下安全配置时，`iframe` 内可以执行 `Node.js`

- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox` 没有显式地设置为 `true`

如果 `iframe` 的地址与渲染页面的地址不同源，则在以下安全配置时，`iframe` 内才可以执行 `Node.js`

- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox` 没有显式地设置为 `true`
- `nodeIntegrationInSubFrames: true`

如果 `iframe` 的地址与渲染页面的地址同源，并且关闭了上下文隔离，则 `iframe` 可以获取到渲染页面和 `Preload` 的上下文

如果 `iframe` 的地址与渲染页面的地址同源，但是关闭了上下文隔离，则 `iframe` 可以获取到渲染页面的上下文

如果 `iframe` 的地址与渲染页面的地址不同源，则 `iframe` 是一个独立的上下文

关闭同源策略 (`webSecurity: false`) 并不会对上面的结果产生影响

参考文章

<https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/iframe>

0x03 object

HTML `<object>` 元素（或者称作 HTML 嵌入对象元素）表示引入一个外部资源，它用于嵌入各种外部对象到网页中，如图像、多媒体（如音频、视频）、SVG图形、PDF文档、Flash 动画（虽然现代Web已逐步淘汰Flash）等。这个标签提供了一种灵活的方式来整合多种媒体类型和应用程序到网页中，而不局限于单一类型的资源。

通常使用语法如下

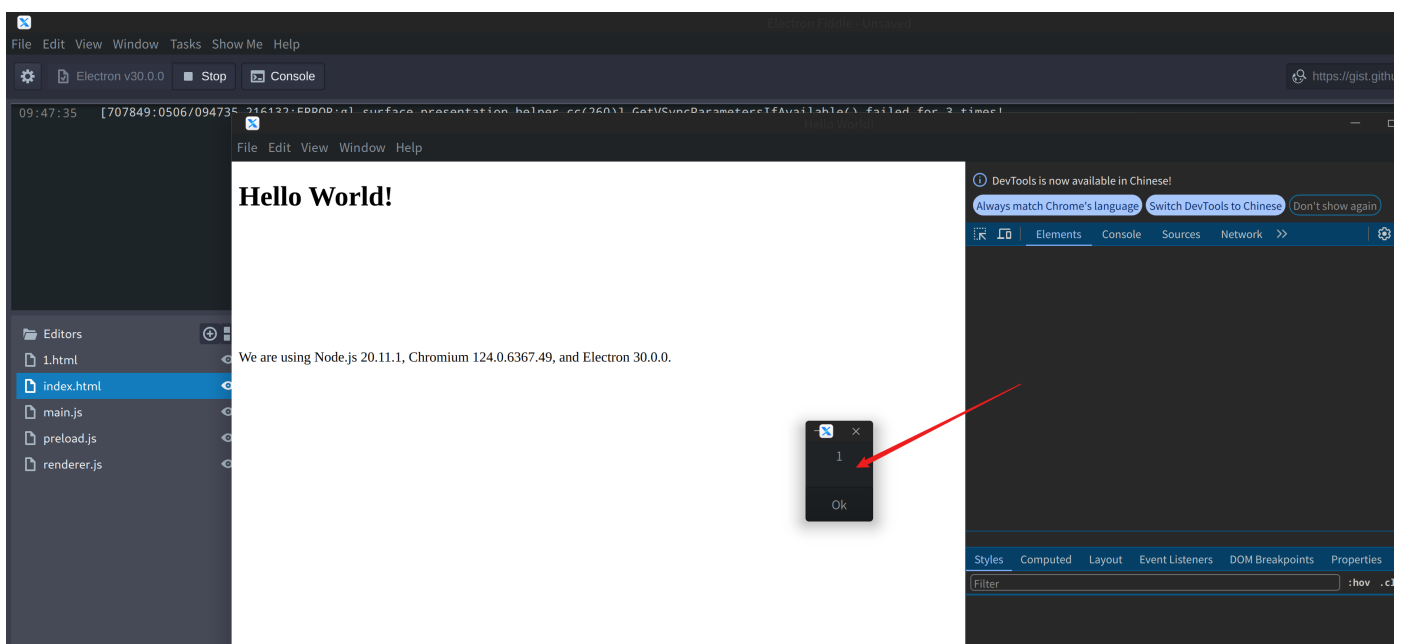
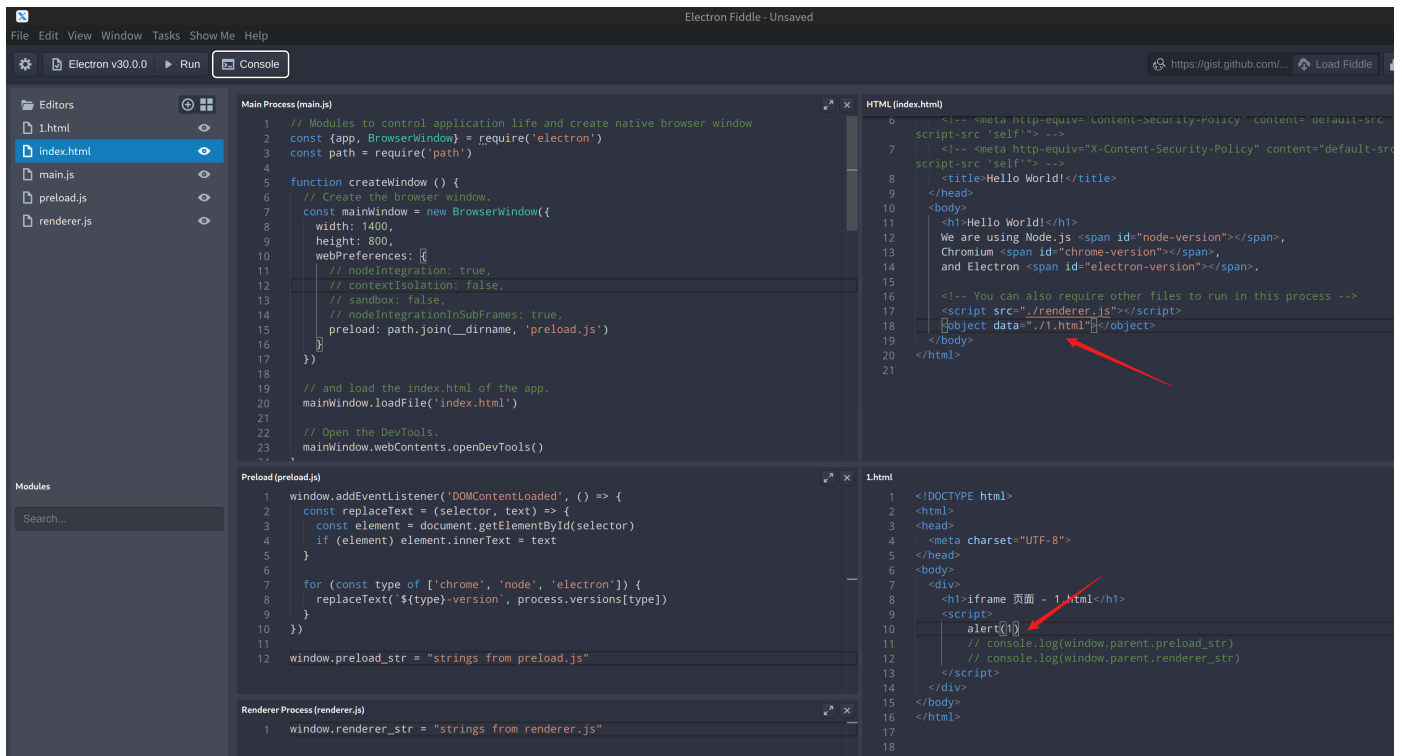
```
<object data="path/to/resource" type="mime-type" width="width-value"
height="height-value">
```

<!-- 可以在这里放置备用内容，供不支持object的浏览器或资源无法加载时显示 -->

```
<param name="parameterName" value="parameterValue"> <!-- 设置对象参数 -->
```

```
<embed src="path/to/resource" type="mime-type" width="width-value"
height="height-value"> <!-- 有时用于兼容性增强 -->
```

```
</object>
```



可以看到，它也是支持加载 HTML 页面的

1. object 属性

`object` 元素包含全局属性，也就是包含那些所有标签都可以使用的属性

1) archive 【已被弃用】

用来指名对象资源列表的以空格分隔的 URI 列表

实测已被弃用

2) border 【已被弃用】

元素周围的边框的宽度，单位为像素

3) classid 【已被弃用】

对象实现的 URI，可以同时与 **data** 属性使用，或者使用 **data** 属性替代

4) codebase 【已被弃用】

解析 **classid**，**data** 或者 **archive** 中定义的相对路径的根路径，如果没有定义，默认为当前文档的 base URI

5) codetype 【已被弃用】

classid 定义的 data 的内容类型

6) data

一个合法的 URL 作为资源的地址，需要为 **data** 和 **type** 中至少一个设置值

7) declare 【已被弃用】

取值为布尔的属性可以设置这个元素为仅声明的格式。对象必须被随后的 `<object>` 元素实例化。在 HTML5 中，完整的重复 `<object>` 元素，可以重用元素

8) form

对象元素关联的 form 元素（属于的 form）。取值必须是同一文档下的一个 form 元素的 ID

9) height

资源显示的高度，单位是 CSS 像素

10) name

浏览上下文名称（HTML5），或者控件名称（HTML 4）

11) standby 【已被弃用】

对象的实现和数据加载过程中，浏览器可以显示的信息

12) tabindex 【已被弃用】

当前元素在文档 Tab 导航中的顺序

13) type

data 指定的资源的 MIME 类型，需要为 **data** 和 **type** 中至少一个设置值

14) usemap 【已被弃用】

指向一个 `map` 元素的 hash-name；格式为‘#’加 `map` 元素 `name` 元素的值

15) width

资源显示的宽度，单位是 CSS 像素

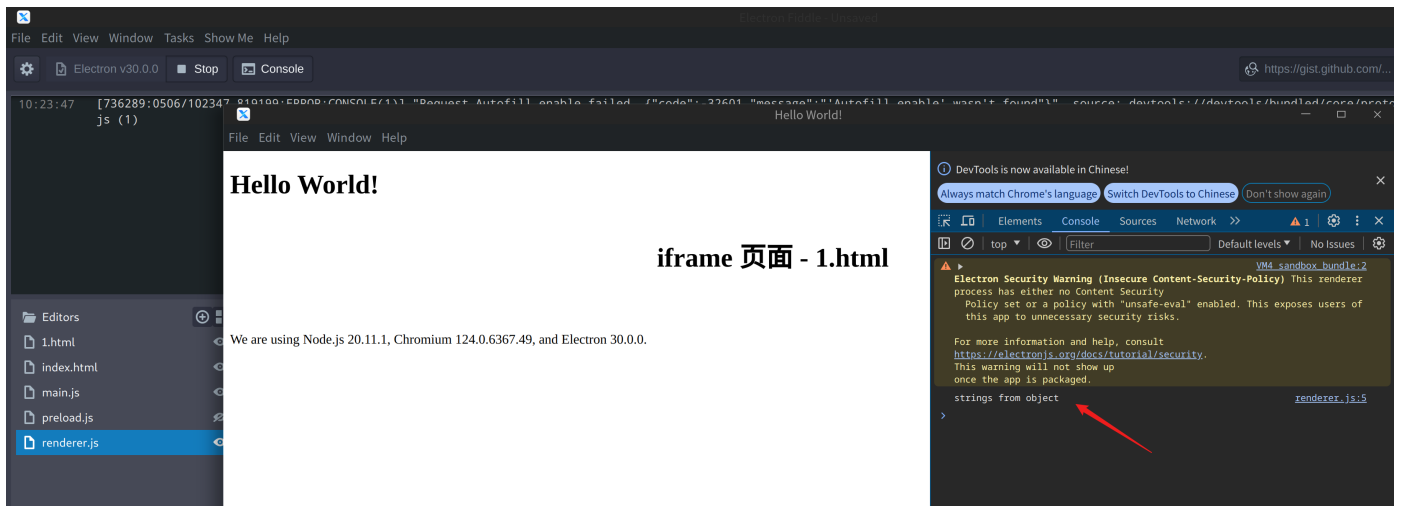
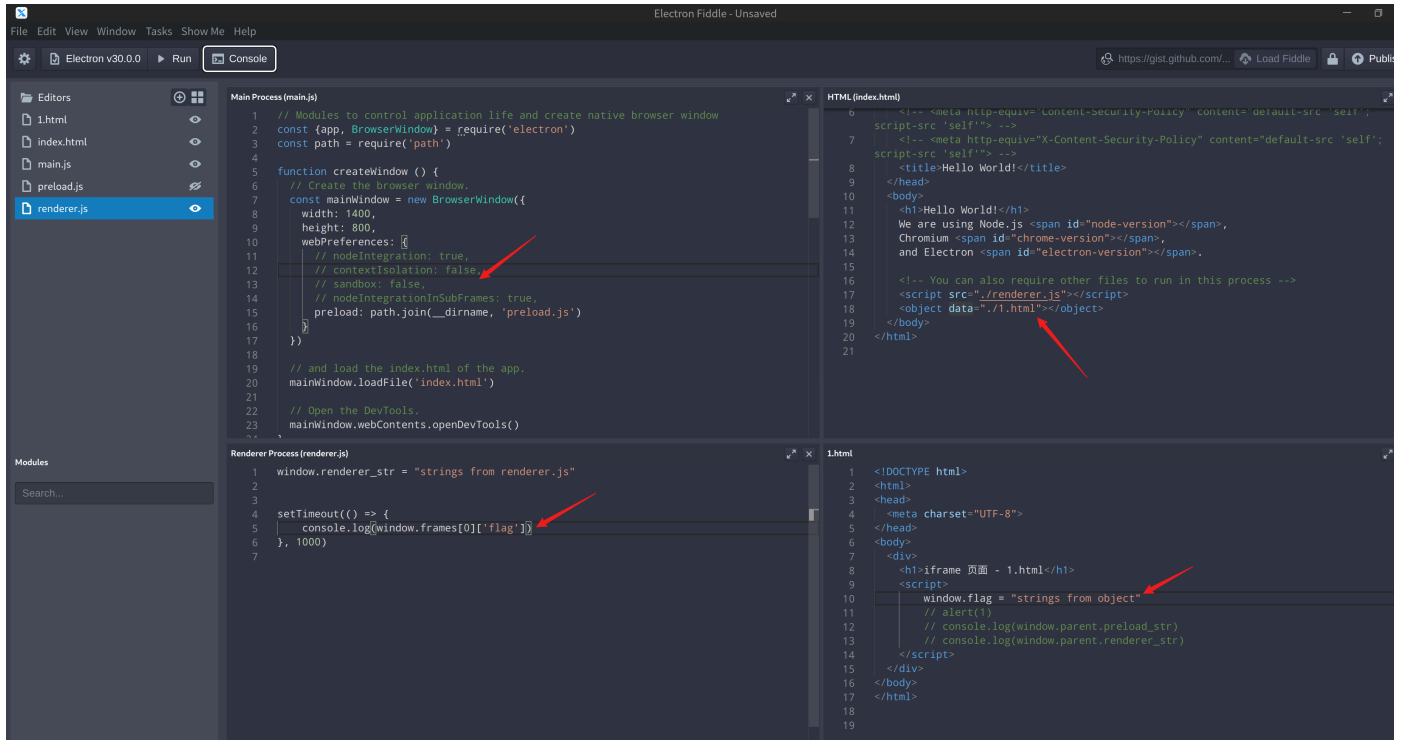
所以能用的也就剩下

- data
- form
- height
- name
- type
- width

2. 渲染页面与 **object** 通信及上下文

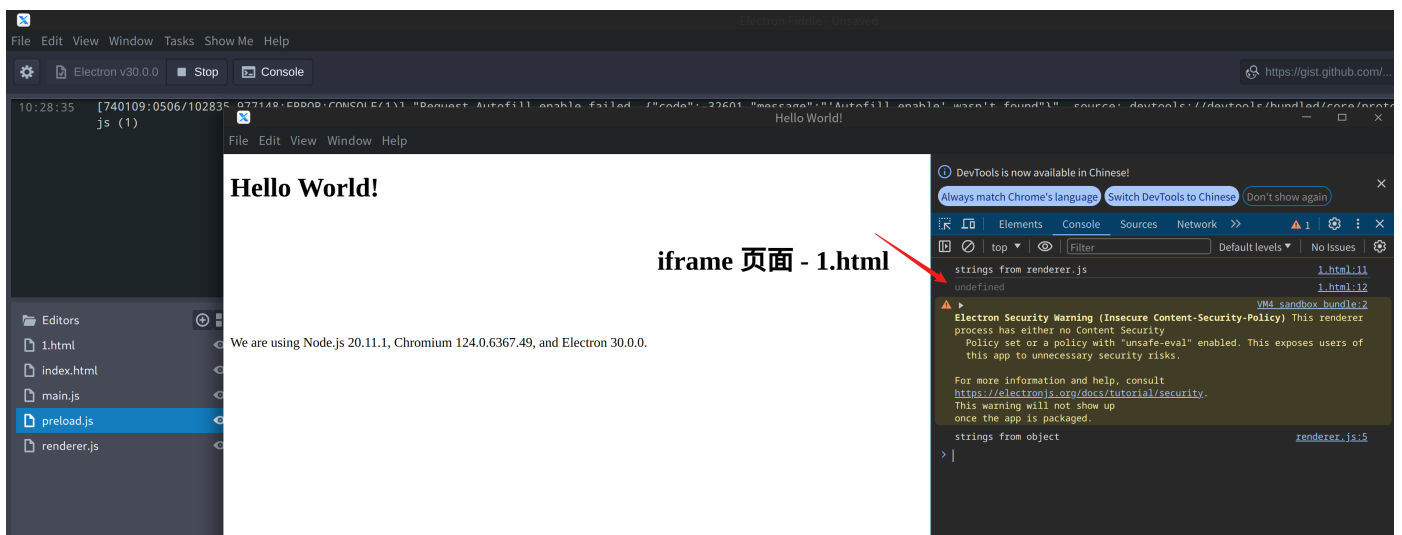
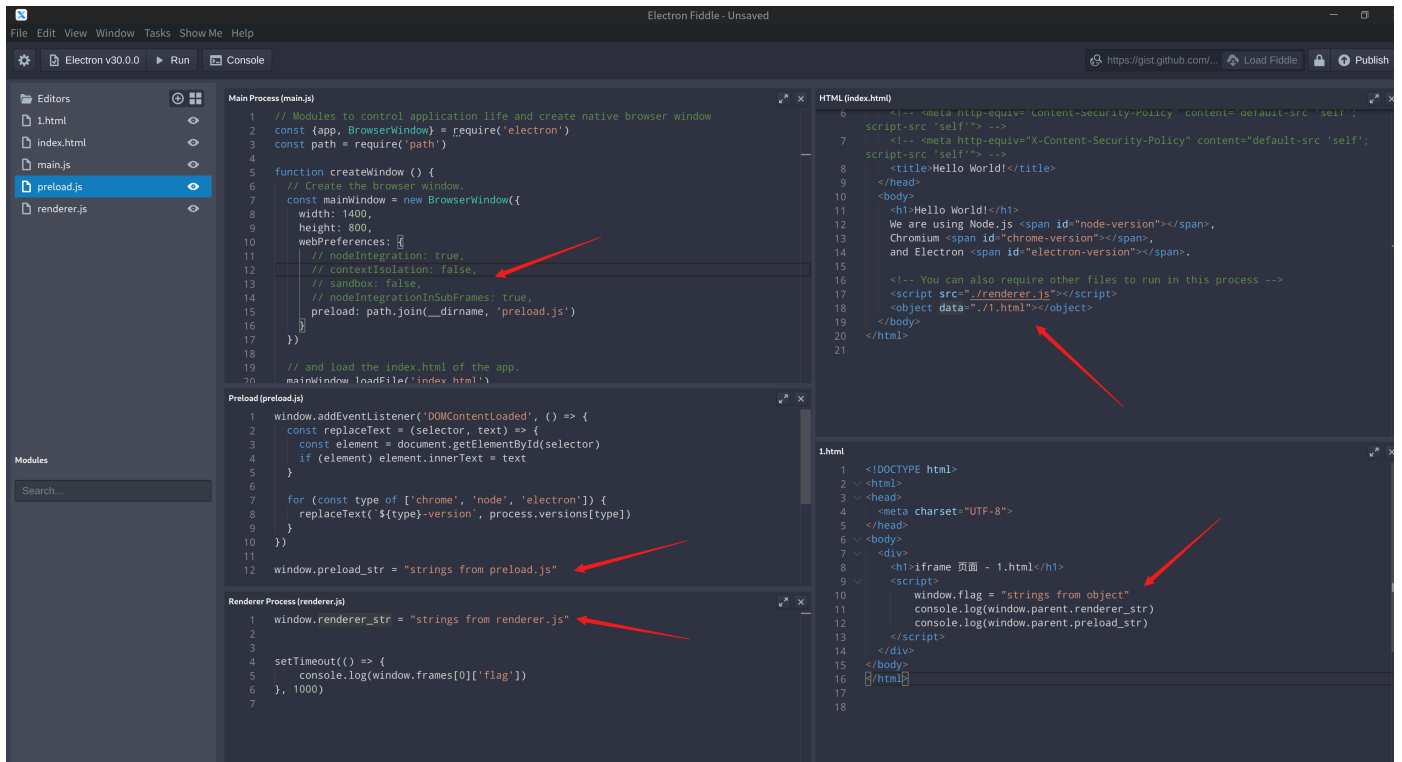
`<object>` 元素自身并不直接提供一种标准化的跨上下文通信机制，类似 `postMessage`

但是我发现，渲染页面与 `object` 的 URL 同源的情况下还是渲染页面还是可以使用 `window.frames` 获取 `object` 的上下文



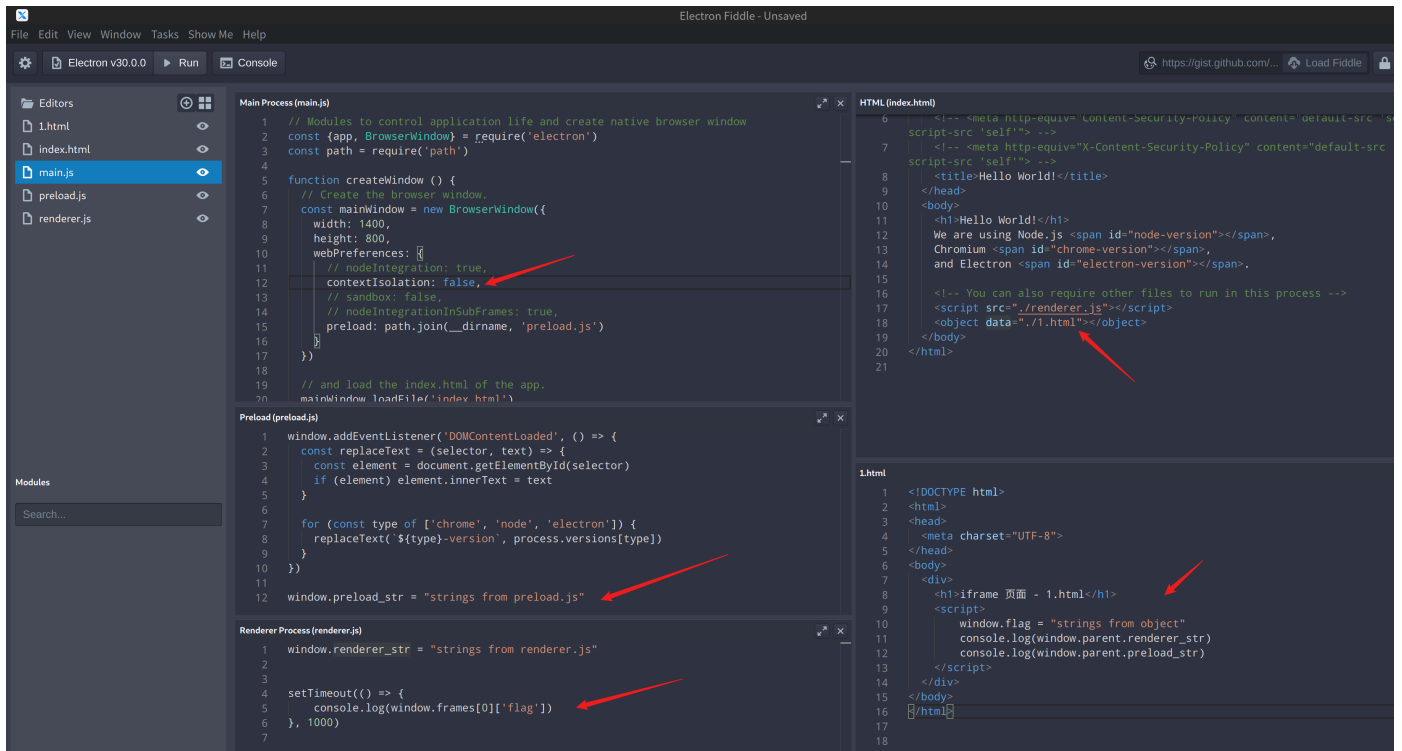
渲染页面成功获取到 `object` 的上下文

测试一下 `object` 是否可以通过 `window.parent` 的方式获取到渲染页面和 `Preload` 的上下文



在开启上下文隔离的情况下， `object` 可以获取到渲染页面的上下文，但是无法获取 `Preload` 的上下文

显式地关闭上下文隔离，再次测试



object 成功获取到渲染页面以及 Preload 脚本的上下文

如果不同源，测试一下

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  </head>
6  <body>
7  <div>
8  <h1>iframe 页面 - 1.html</h1>
9  <script>
10     window.flag = "strings from object"
11     // window.parent.require('child_process').exec('deepin-music')
12     setTimeout(() => {
13         console.log(window.parent.renderer_str)
14         console.log(window.parent.preload_str)
15     }, 2000)
16
17 </script>
18 </div>
19 </body>
20 </html>

```

The screenshot shows the Electron IDE with the following code:

Main Process (main.js)

```

1  // Modules to control application life and create native browser window
2  const {app, BrowserWindow} = require('electron')
3  const path = require('path')
4
5  function createWindow () {
6  // Create the browser window.
7  const mainWindow = new BrowserWindow({
8  width: 1400,
9  height: 800,
10 webPreferences: {
11 // nodeIntegration: true,
12 contextIsolation: true,
13 // sandbox: false,
14 // nodeIntegrationInSubFrames: true,
15 preload: path.join(__dirname, 'preload.js')
16 }
17 })
18
19 // and load the index.html of the app.
20 mainWindow.loadFile('index.html')

```

Preload (preload.js)

```

1  window.addEventListener('DOMContentLoaded', () => {
2  const replaceText = (selector, text) => {
3  const element = document.getElementById(selector)
4  if (element) element.innerText = text
5  }
6
7  for (const type of ['chrome', 'node', 'electron']) {
8  replaceText(`${type}-version`, process.versions[type])
9  }
10 })
11
12 window.preload_str = "strings from preload.js"

```

Renderer Process (renderer.js)

```

1  window.renderer_str = "strings from renderer.js"
2
3
4  // setTimeout(() => {
5  //   console.log(window.frames[0]['flag'])
6  // }, 1000)
7

```

HTML (index.html)

```

6  <!-- <meta http-equiv="Content-Security-Policy" content="default-src=
7  script-src 'self'" -->
8  <!-- <meta http-equiv="X-Content-Security-Policy" content="default-src=
9  script-src 'self'" -->
10 <title>Hello World!</title>
11 </head>
12 <body>
13 <h1>Hello World!</h1>
14 We are using Node.js <span id="node-version"></span>,
15 Chromium <span id="chrome-version"></span>,
16 and Electron <span id="electron-version"></span>.
17
18 <!-- You can also require other files to run in this process -->
19 <script src="._renderer.js"></script>
20 <object data="http://192.168.31.216/1.html"></object>
21 </body>
22 </html>

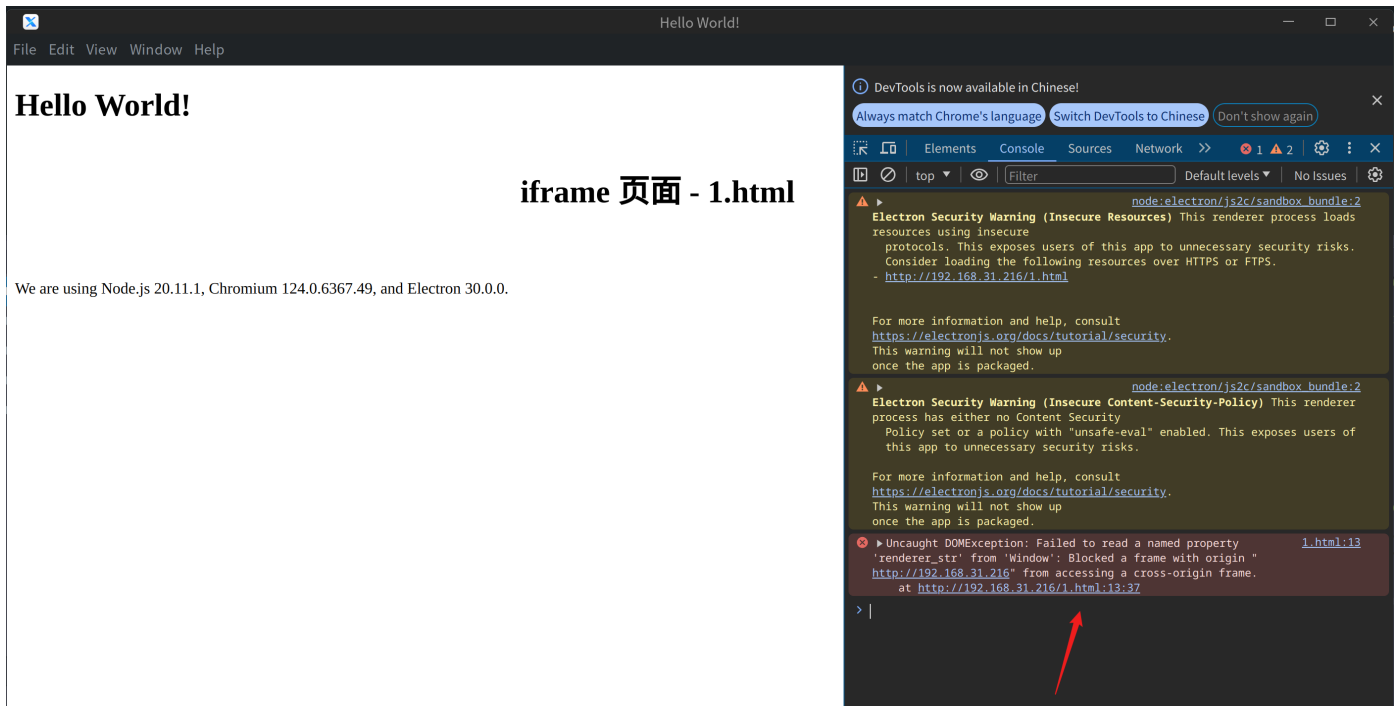
```

1.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  </head>
6  <body>
7  <div>
8  <h1>iframe 页面 - 1.html</h1>
9  <script>
10     window.flag = "strings from object"
11     // window.parent.require('child_process').exec('deepin-music')
12     console.log(window.parent.renderer_str)
13     console.log(window.parent.preload_str)
14 </script>
15 </div>
16 </body>
17 </html>

```

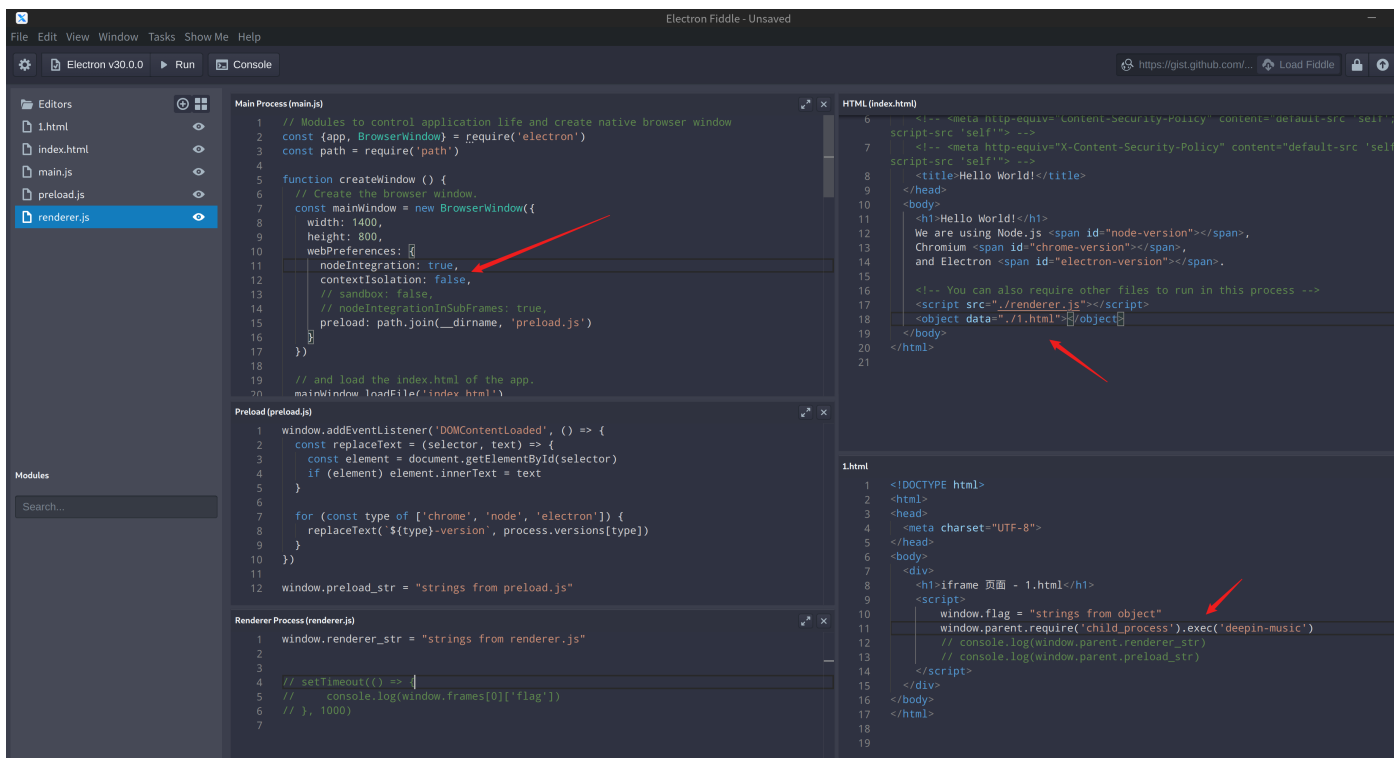


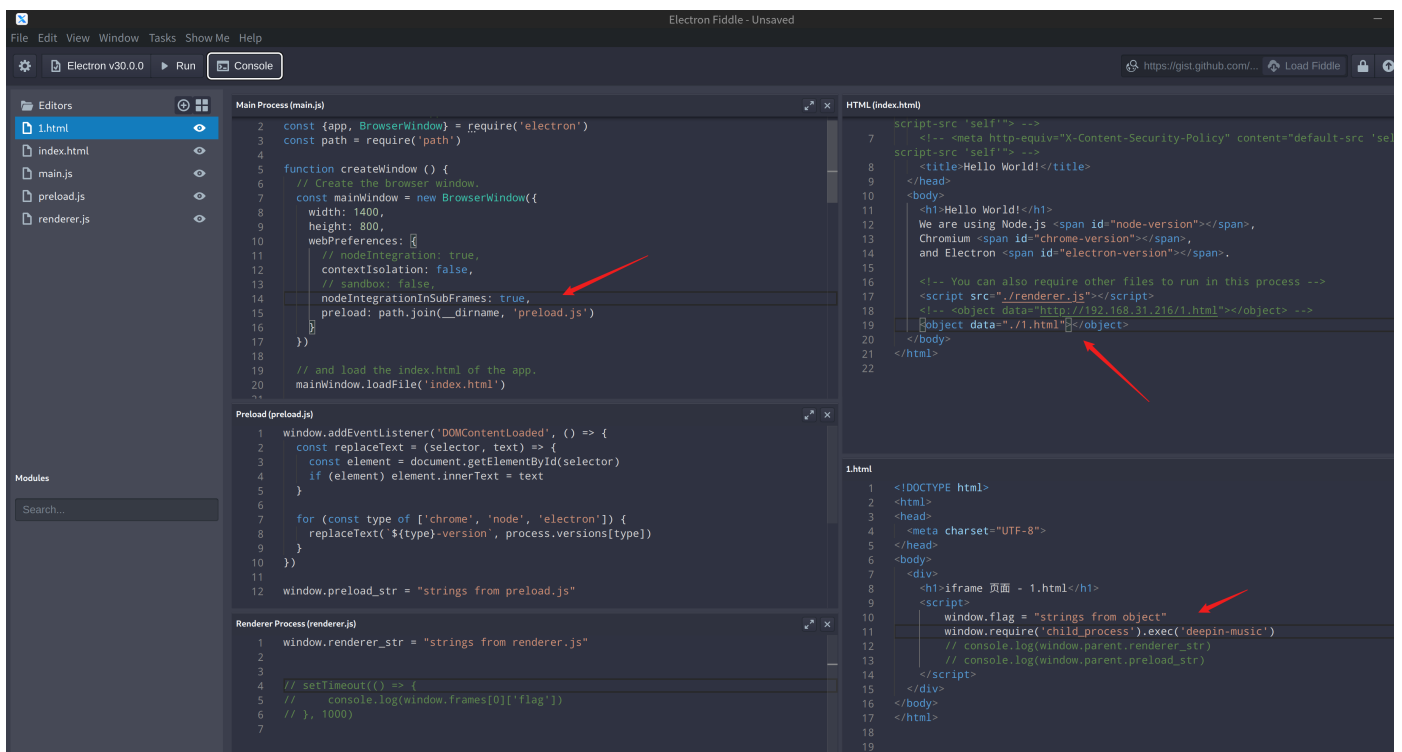
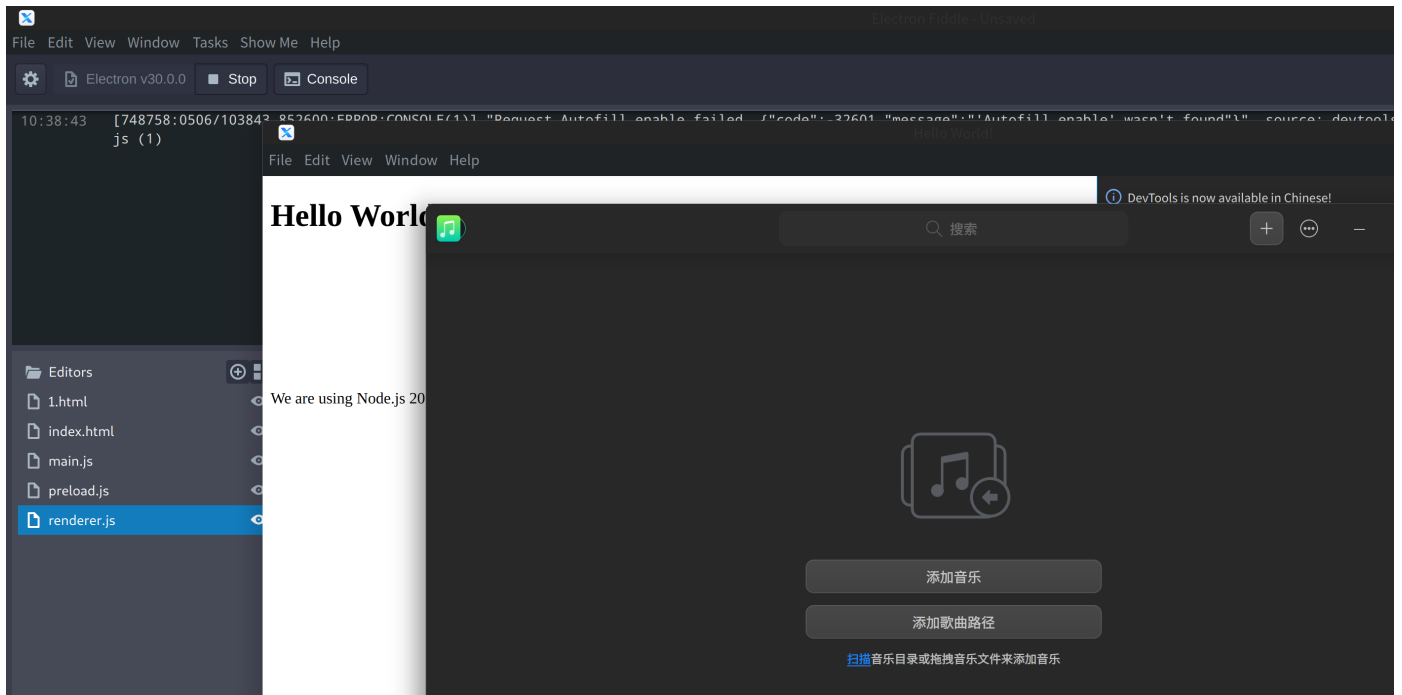
被阻止

3. object 执行 Node.js 的情况

目前来看应该和 `iframe` 是一致的，测试一下

同源情况下



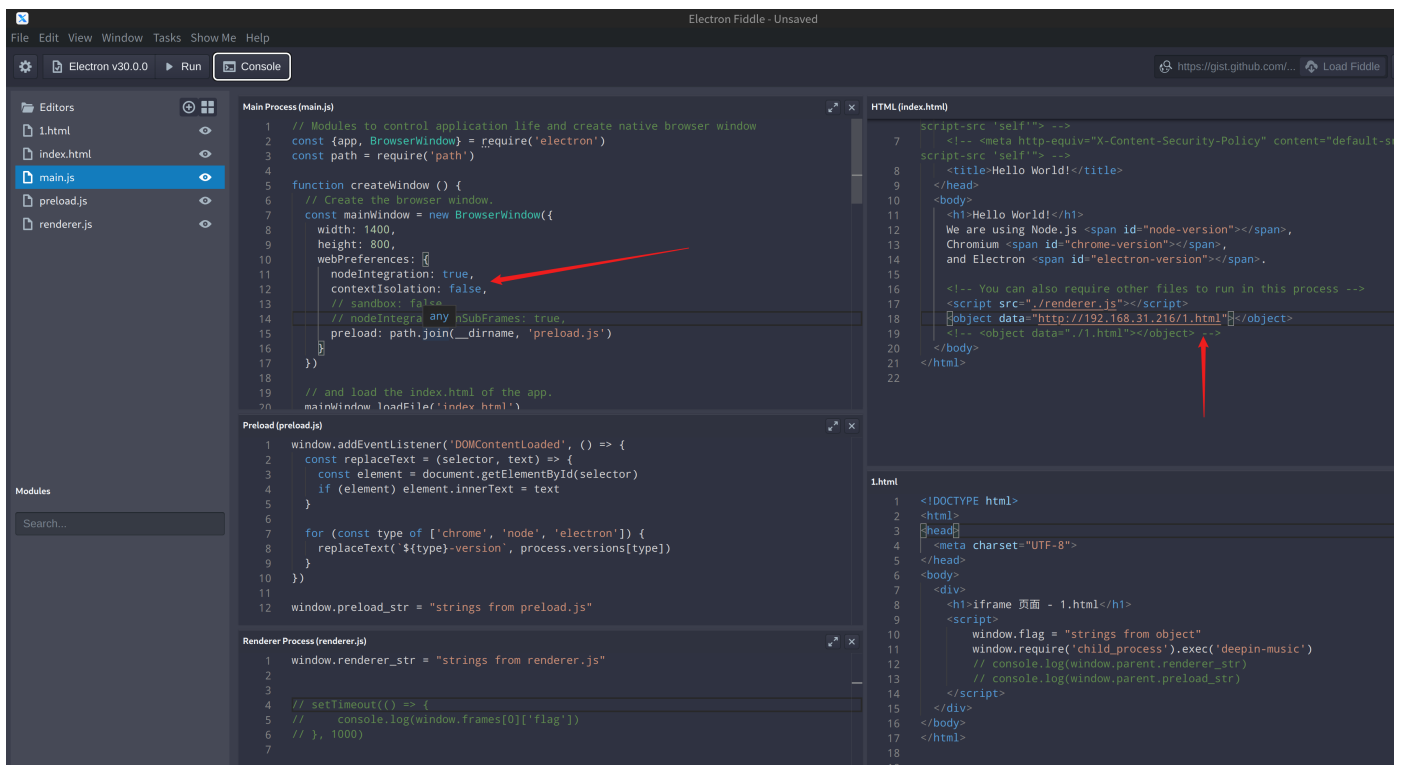




看来在同源情况下，`object` 想要执行 `Node.js` ，需要满足以下条件

- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox` 没有显式地设置为 `true`

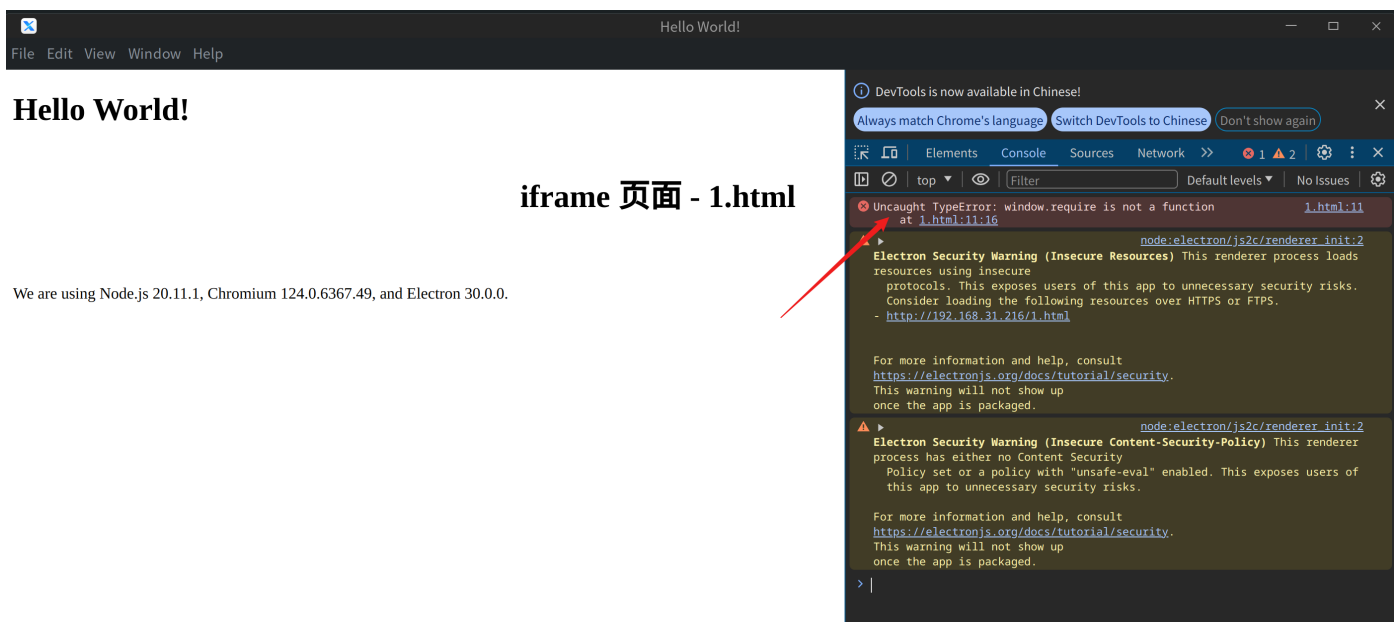
不同源的情况下，进行测试



```

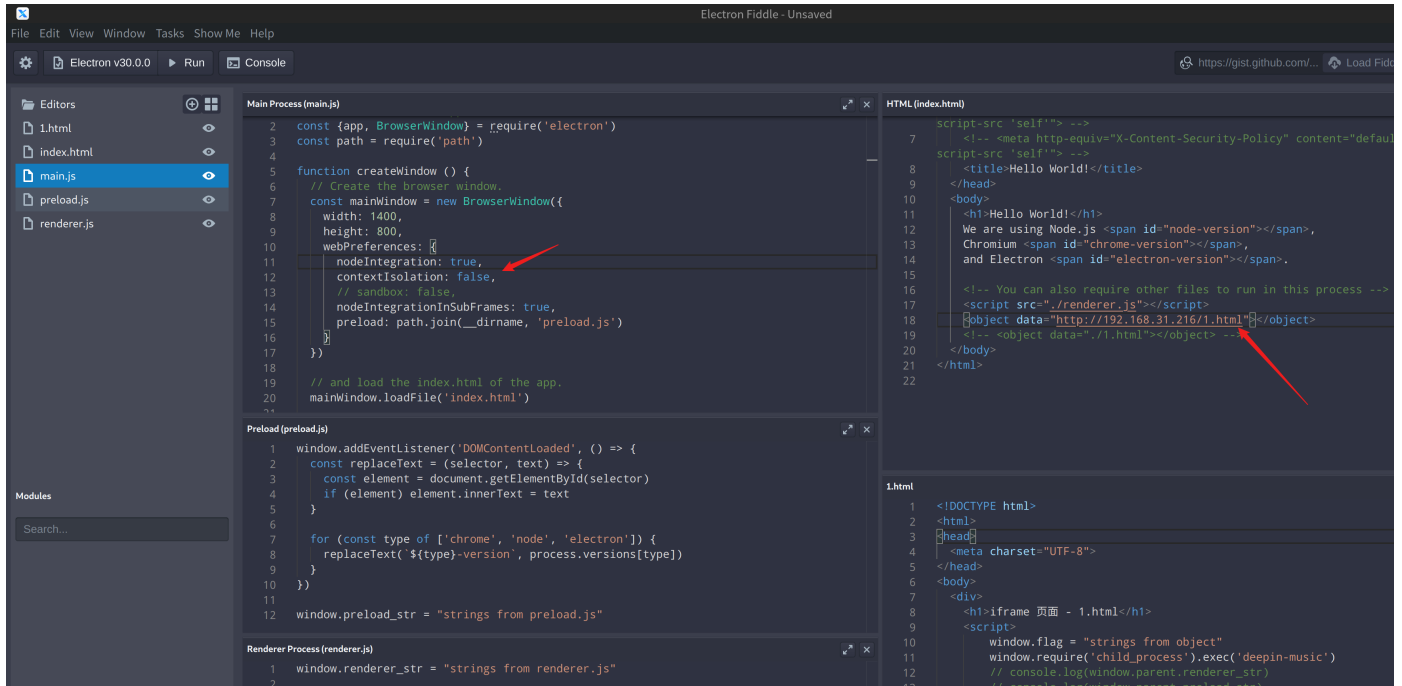
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  </head>
6  <body>
7  <div>
8  <h1>iframe 页面 - 1.html</h1>
9  <script>
10     window.flag = "strings from object"
11     window.require('child_process').exec('deepin-music')
12     // setTimeout(() => {
13     //     console.log(window.parent.renderer_str)
14     //     console.log(window.parent.preload_str)
15     // }, 2000)
16
17 </script>
18 </div>
19 </body>
20 </html>

```

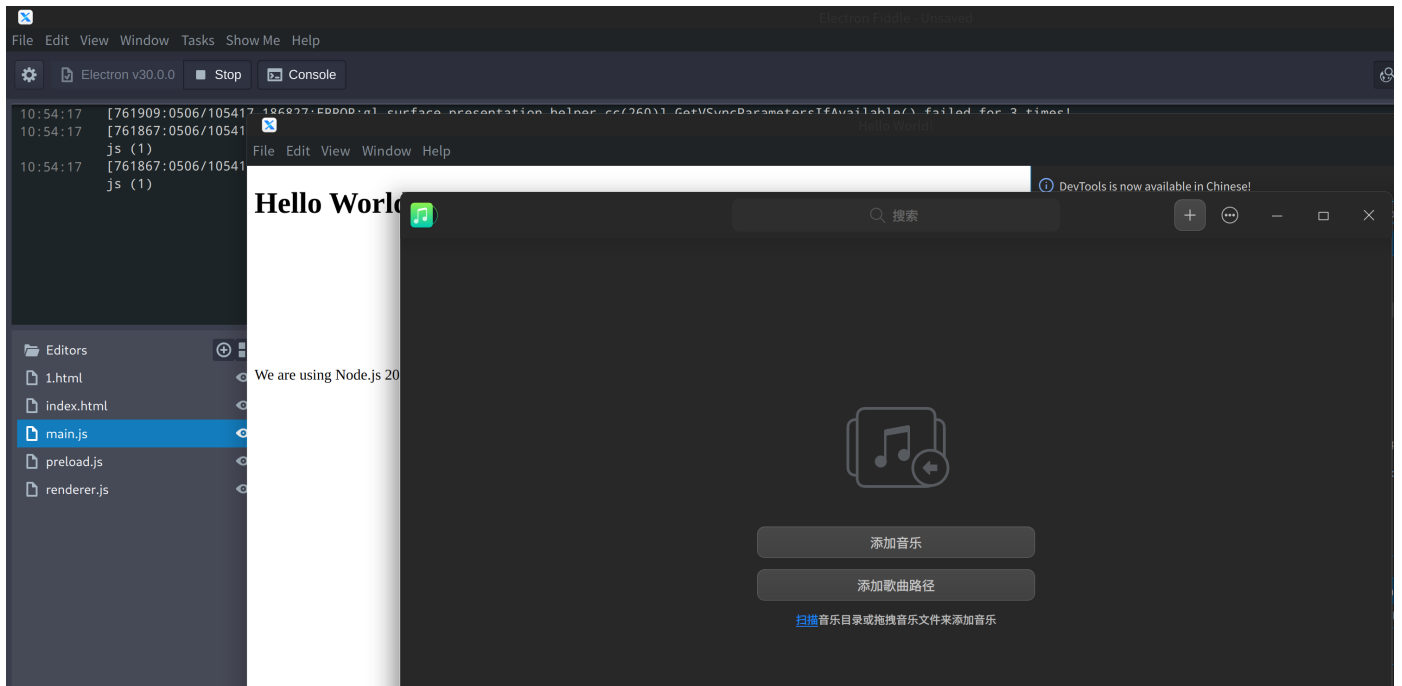


执行失败，换成 `window.parent.require` 也失败

添加 `nodeIntegrationInSubFrames: true`



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="UTF-8">
5 </head>
6 <body>
7   <div>
8     <h1>iframe 页面 - 1.html</h1>
9   </div>
10   <script>
11     window.flag = "strings from object"
12     window.require('child_process').exec('deepin-music')
13     // console.log(window.parent.renderer_str)
14     // console.log(window.parent.preload_str)
15     // }, 2000)
16   </script>
17 </div>
18 </body>
19 </html>
```

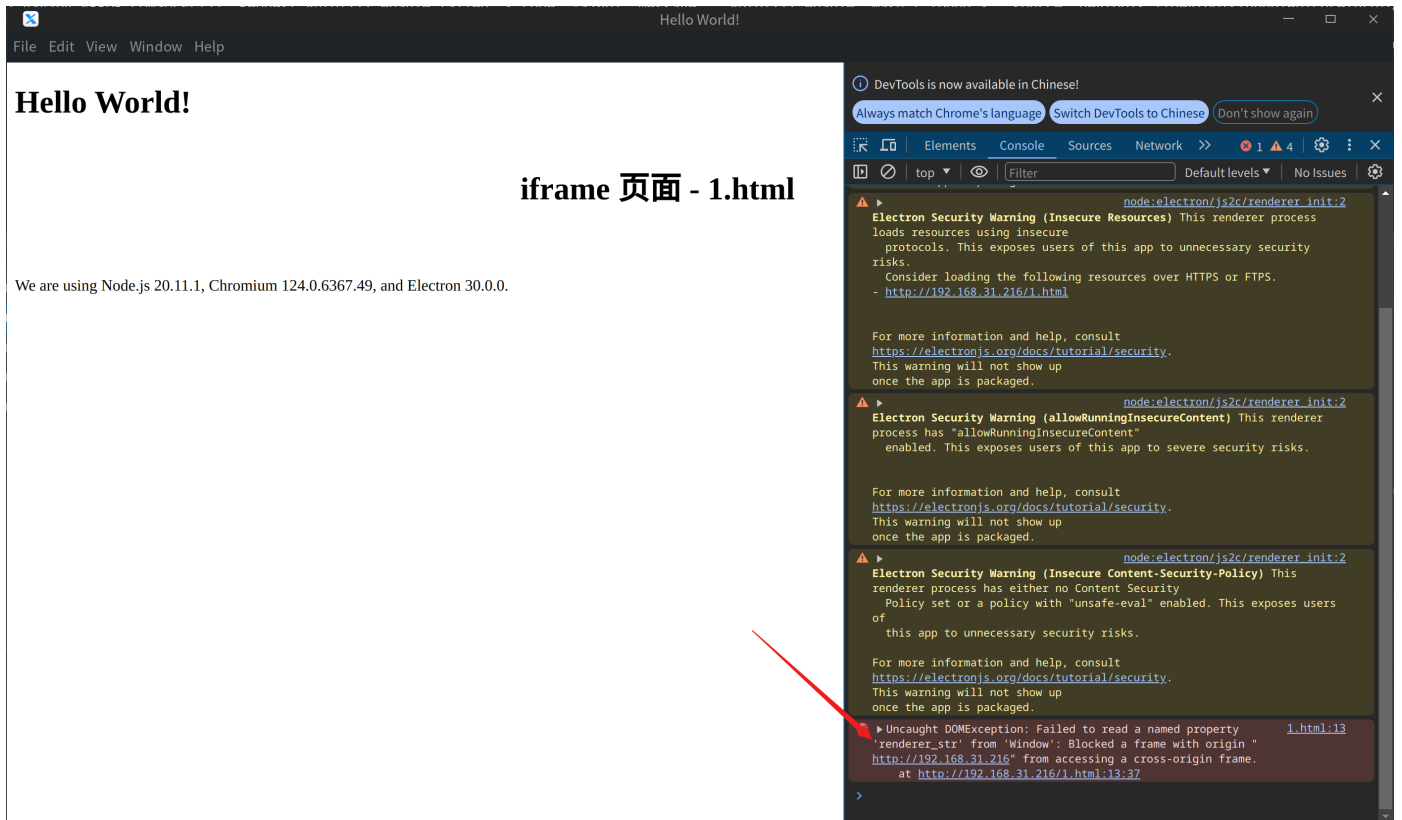


执行成功，开启上下文隔离或关闭 `nodeIntegration` 都会导致执行失败，所以不同源的情况下 `object` 执行 `Node.js` 的条件是：

- `nodeIntegration`: `true`
- `contextIsolation`: `false`
- `sandbox` 没有显式地设置为 `true`
- `nodeIntegrationInSubFrames`: `true`

4. 关闭同源策略

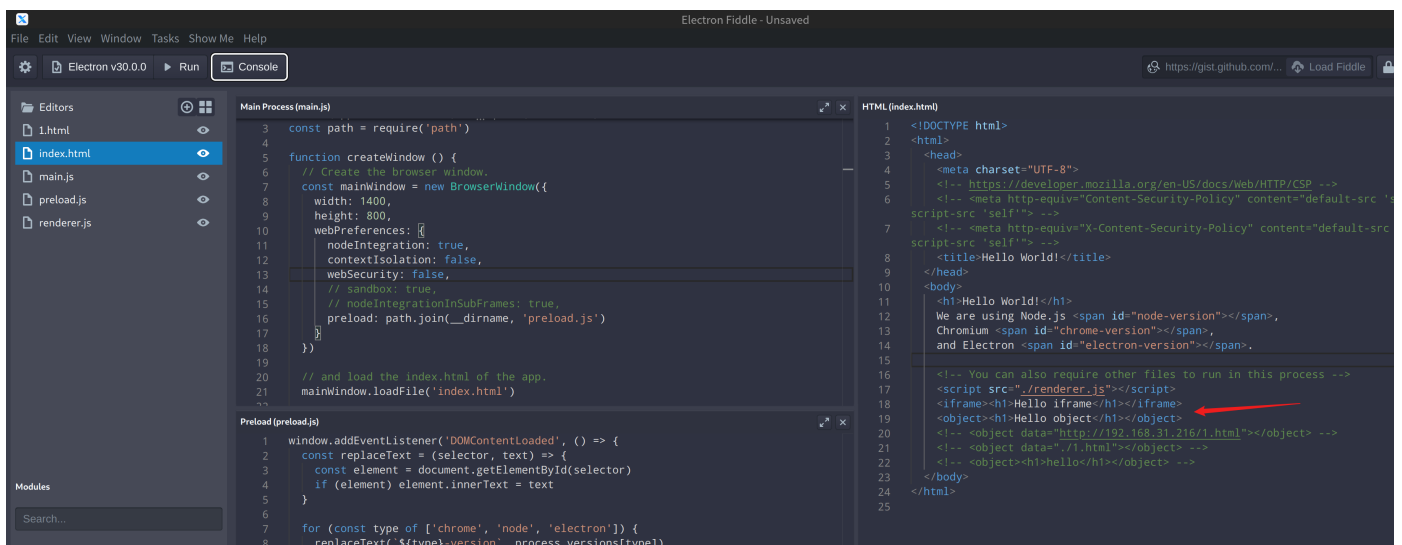
如果关闭同源策略，会让不同源的 `object` 通过 `window.parent` 获取到渲染进程的上下文吗？

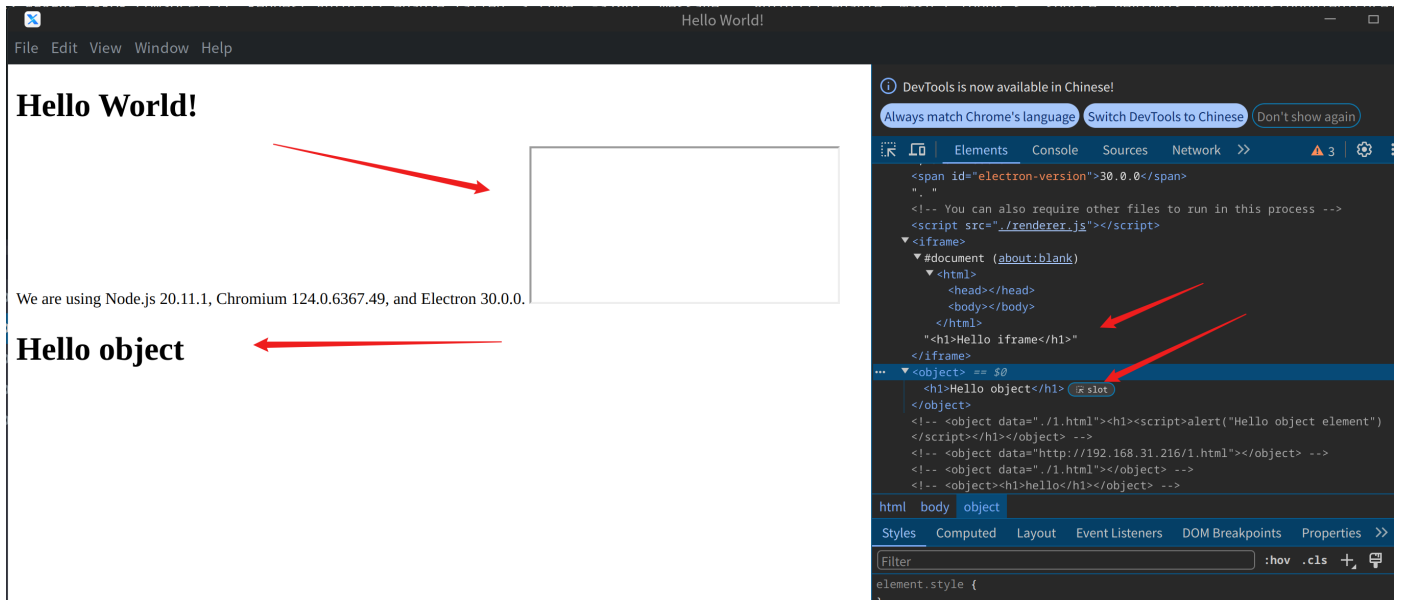


并不能

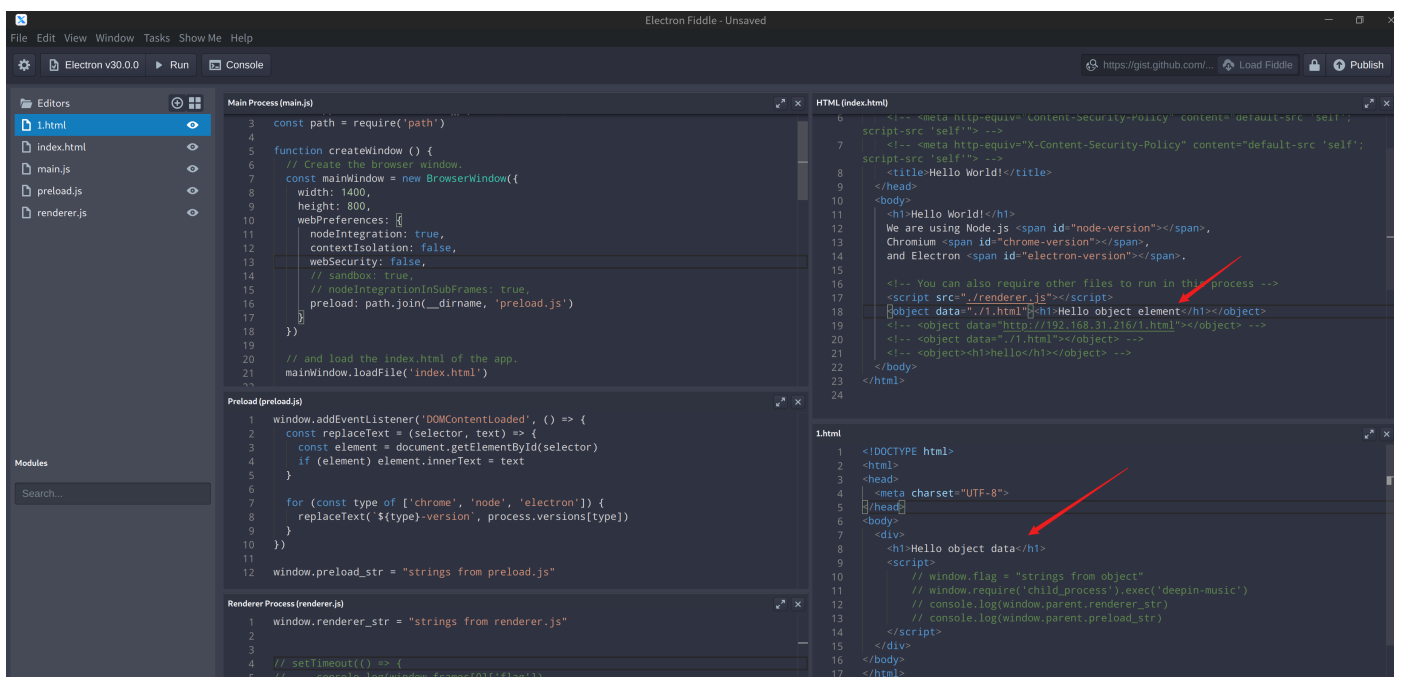
5. object 和 iframe 的不同

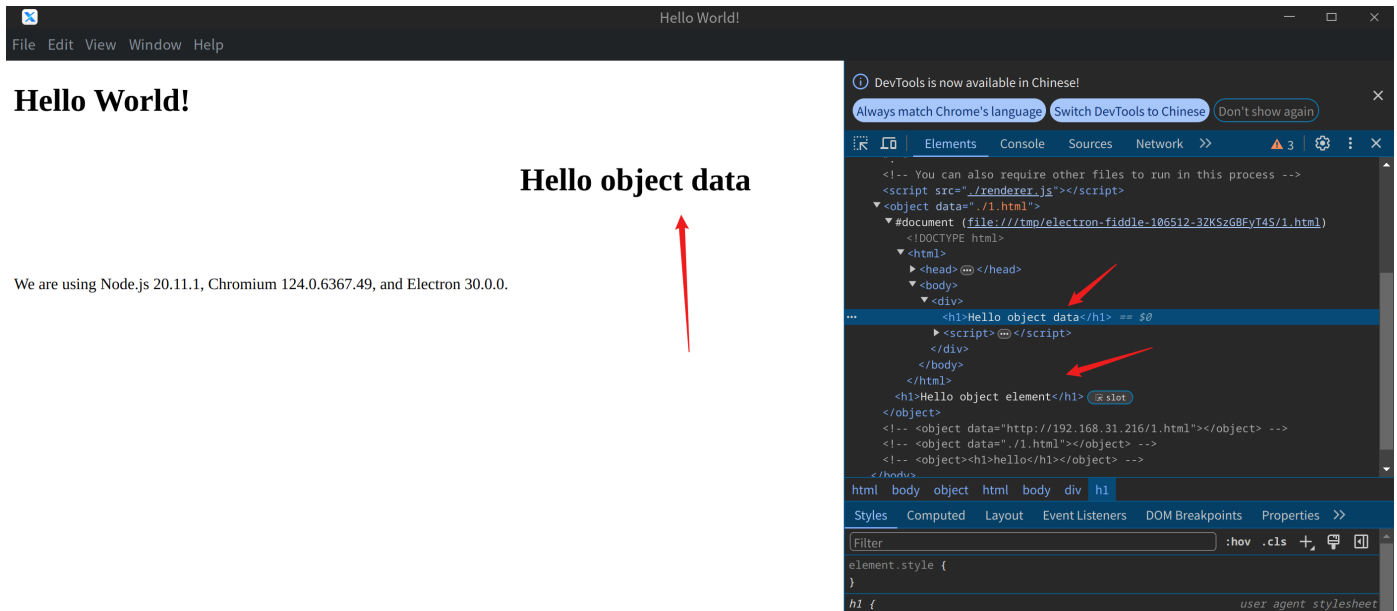
虽然 `object` 和 `iframe` 标签都是通过指定外部 URL 进行加载资源的，但是 `iframe` 标签内的内容不会被解析成HTML，`object` 标签内的内容会被解析成HTML





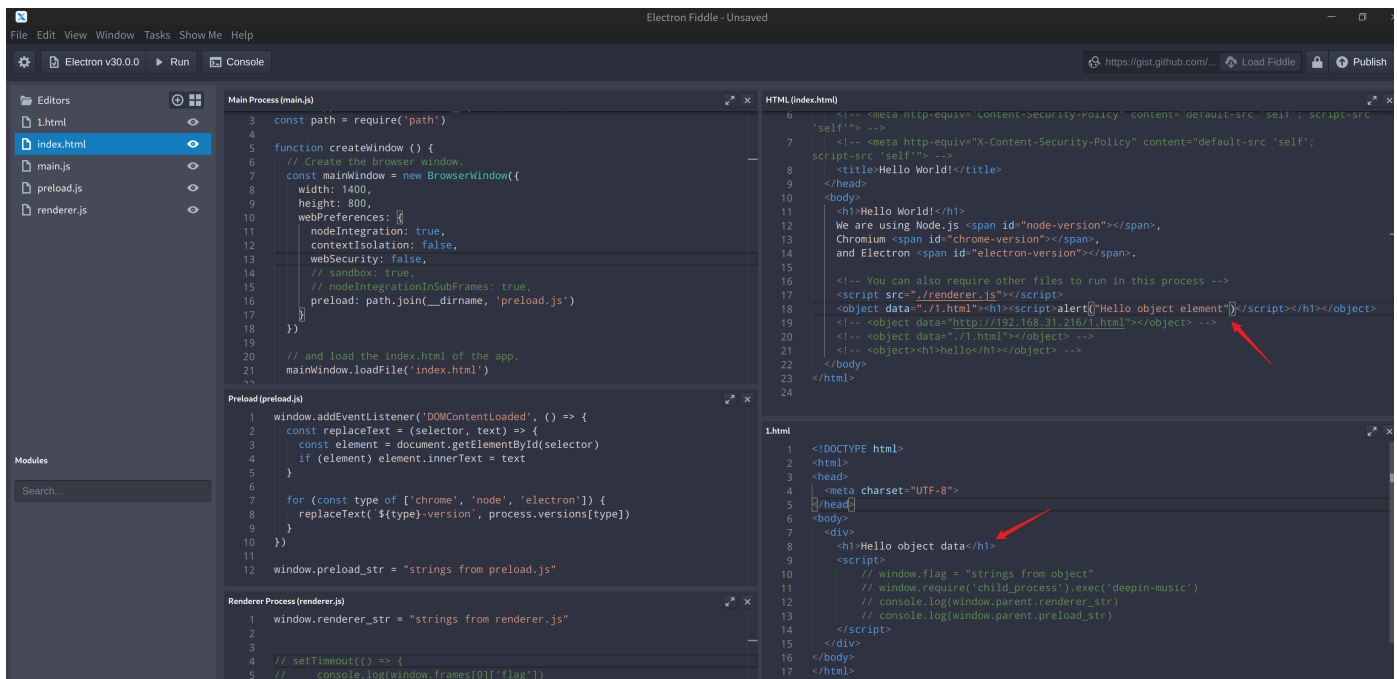
如果 `object` 标签内的内容和 `data` 属性都存在，会解析哪一个呢？

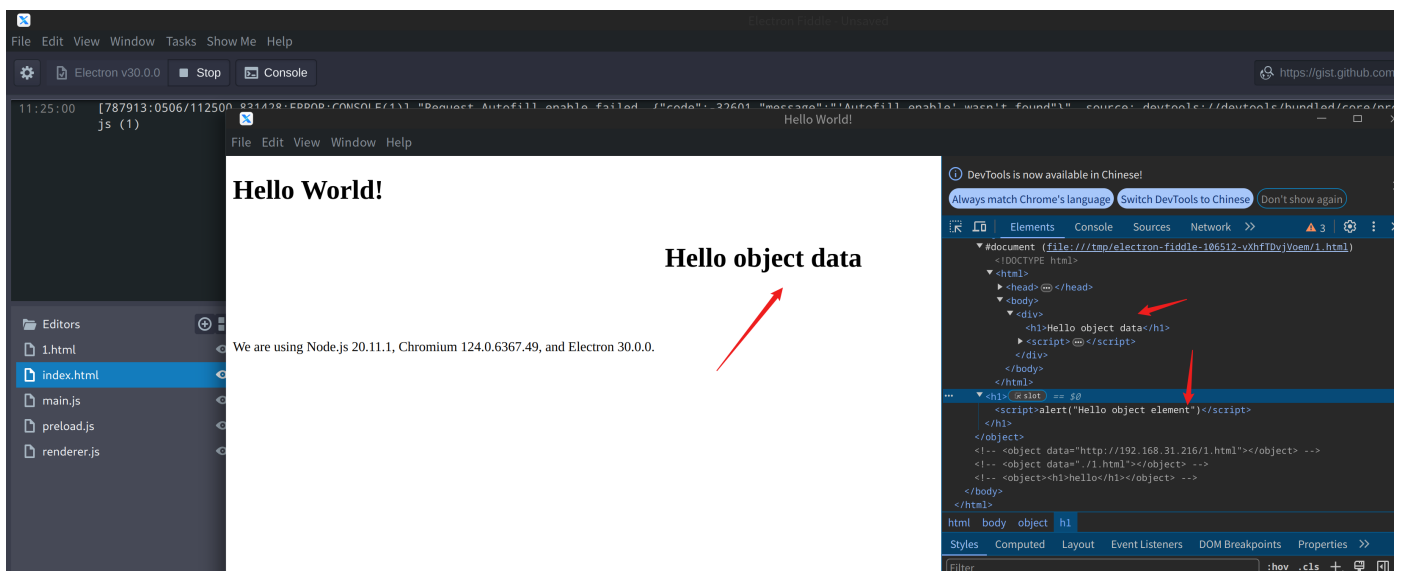
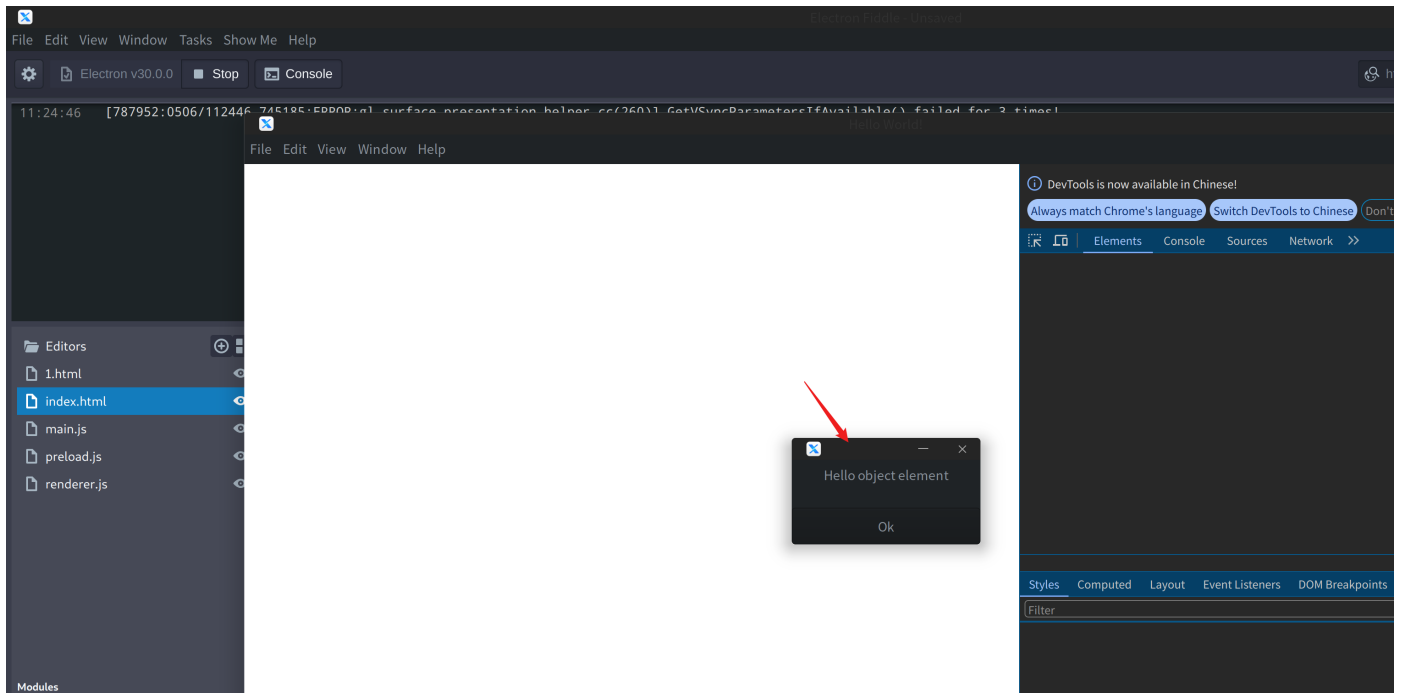




We are using Node.js 20.11.1, Chromium 124.0.6367.49, and Electron 30.0.0.

页面显示了 `data` 指定的内容，但是从页面 HTML 看，标签内的内容也解析了，我们换一个更加明显的 `alert`





6. object 标签内执行情况

object 标签内的情况就和一个 div 标签一样，并不是所谓的子 frame ，所以标签内的内容就是所谓的渲染页面

7. 小结

object 作为旧时代的嵌入，在 Electron 中也得到了很好的支持，测试效果与 iframe 基本一致

如果 `object` 的地址与渲染页面的地址同源，则可以相互直接通讯，并获取相互的上下文；如果非同源则需要依赖嵌入的内容自己的 API 是否支持通讯

如果 `object` 的地址与渲染页面的地址同源，则在以下安全配置时，`object` 内可以执行 `Node.js`

- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox` 没有显式地设置为 `true`

如果 `object` 的地址与渲染页面的地址不同源，则在以下安全配置时，`object` 内才可以执行 `Node.js`

- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox` 没有显式地设置为 `true`
- `nodeIntegrationInSubFrames: true`

如果 `object` 的地址与渲染页面的地址同源，并且关闭了上下文隔离，则 `object` 可以获取到渲染页面和 `Preload` 的上下文

如果 `object` 的地址与渲染页面的地址同源，但是关闭了上下文隔离，则 `object` 可以获取到渲染页面的上下文

如果 `object` 的地址与渲染页面的地址不同源，则 `object` 是一个独立的上下文

关闭同源策略 (`webSecurity: false`) 并不会对上面的结果产生影响

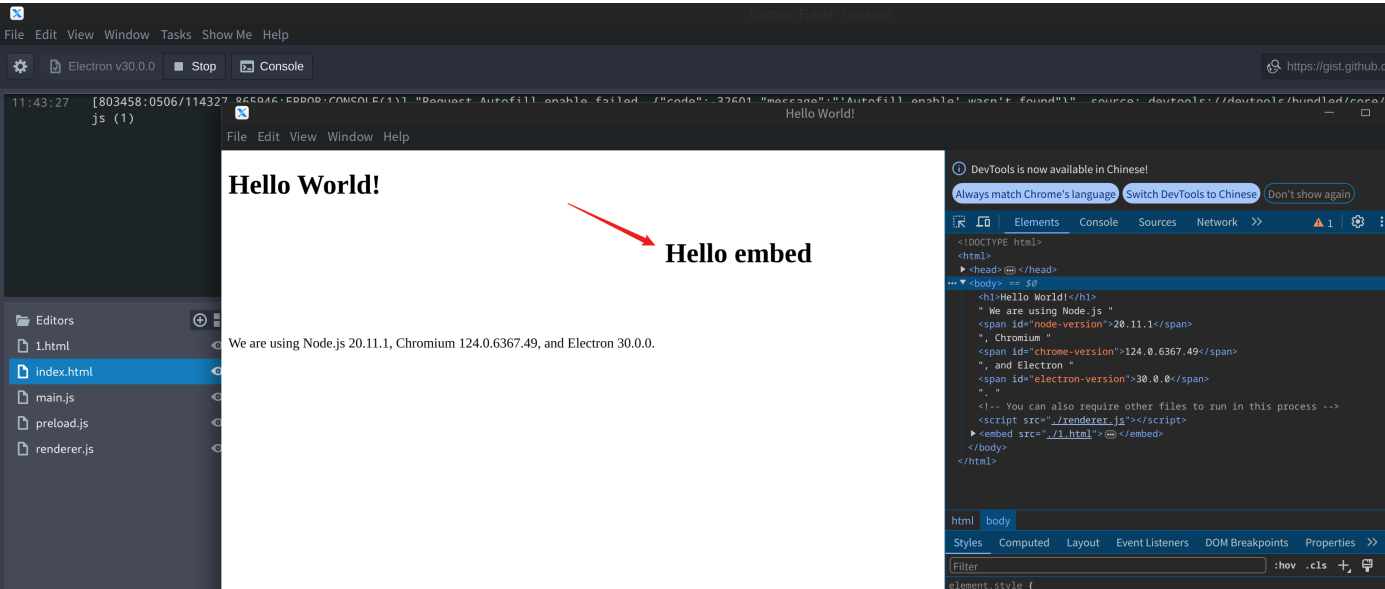
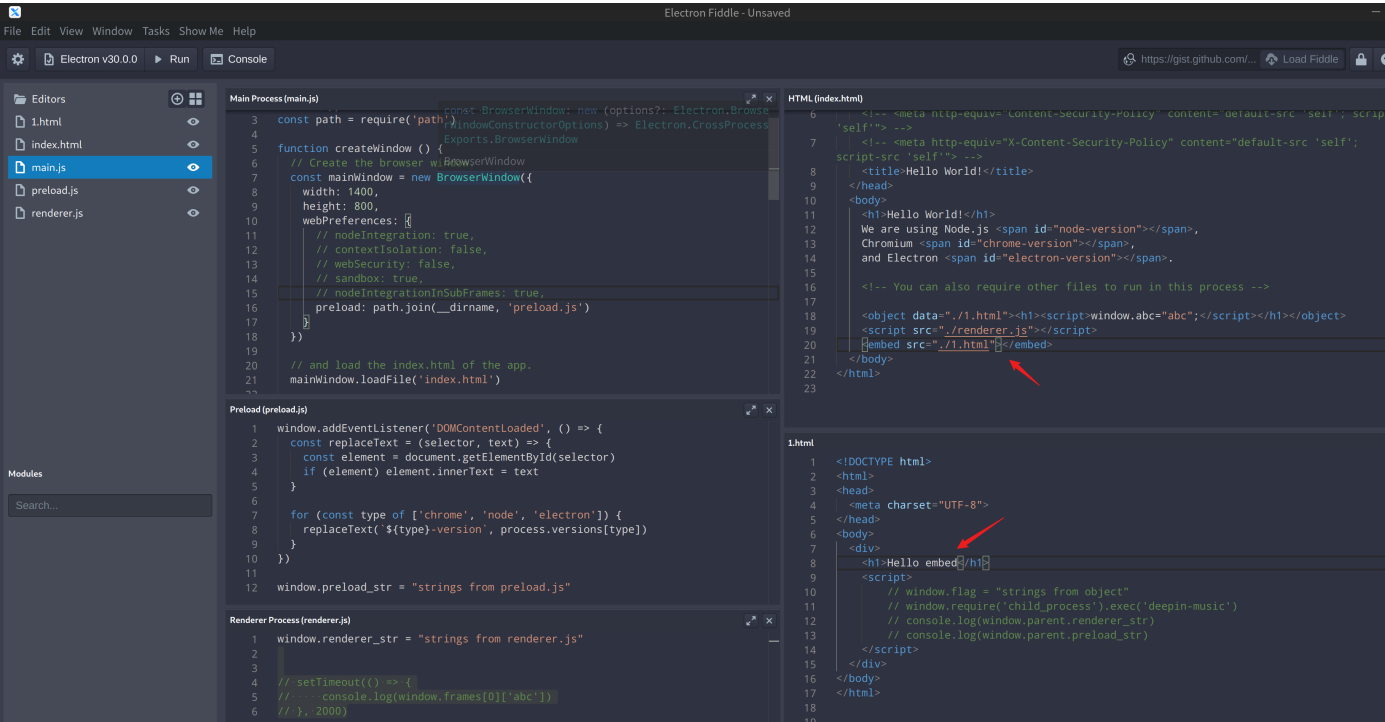
`object` 可以作为一个类似 `div` 的通用标签，内部的内容会当作正常的 HTML 渲染，`data` 和内部的代码同时存在时，`data` 部分正常执行，内部的 HTML 似乎不会渲染在页面上显示，但是内部的 `JavaScript` 会正常执行，执行限制和渲染页面策略一致，而不是和 `data` 指向的页面策略一致

参考文章

<https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/object>

0x04 embed

HTML `<embed>` 元素将外部内容嵌入文档中的指定位置。此内容由外部应用程序或其他交互式内容源（如浏览器插件）提供，与 `object` 基本一致，只是更倾向于多媒体



1. embed 属性

1) height

资源显示的高度

2) src

被嵌套的资源的 URL

3) type

用于选择插件实例化的 MIME 类型

4) width

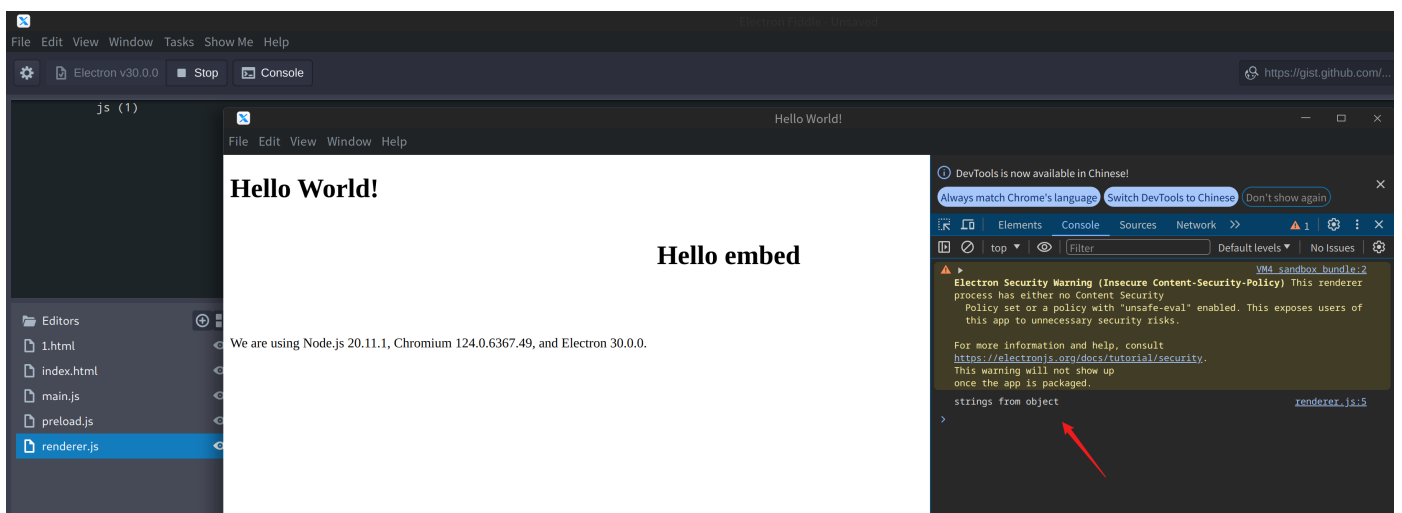
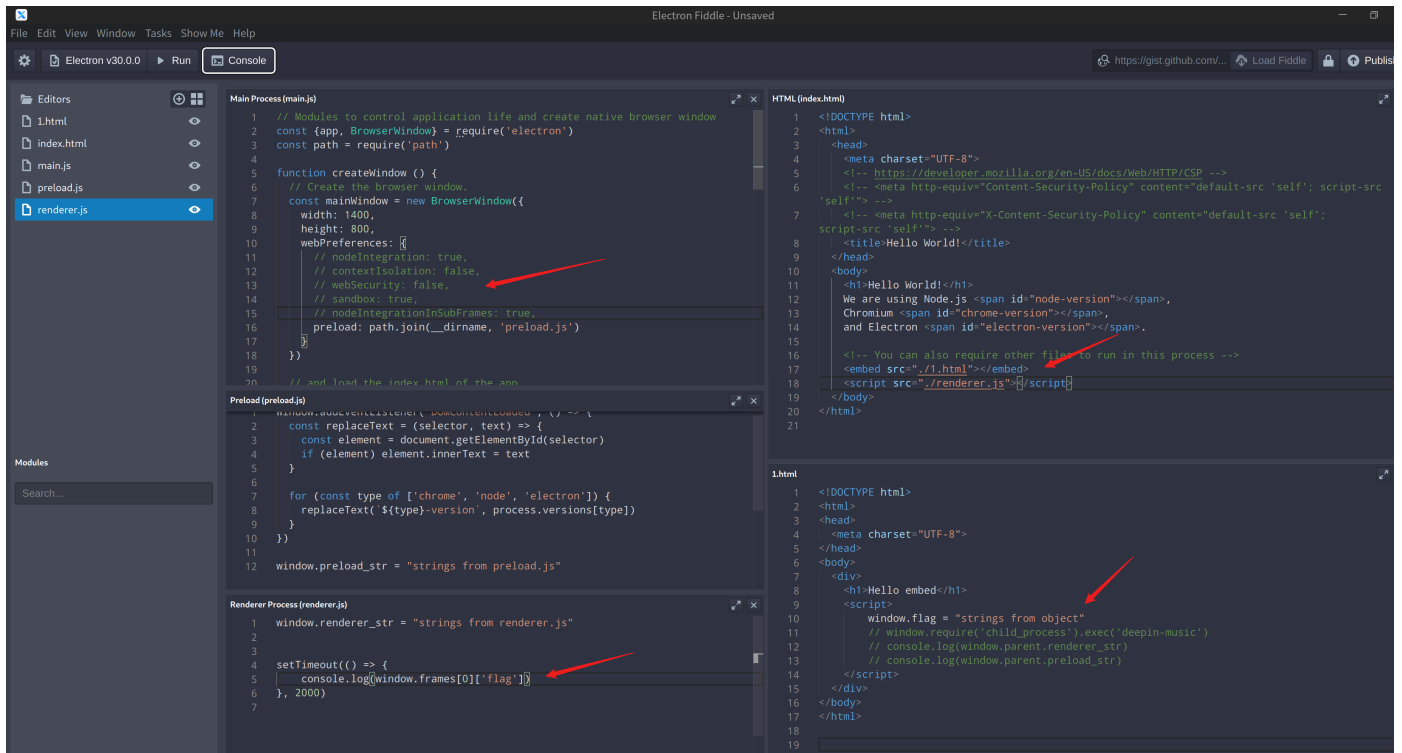
资源显示的宽度

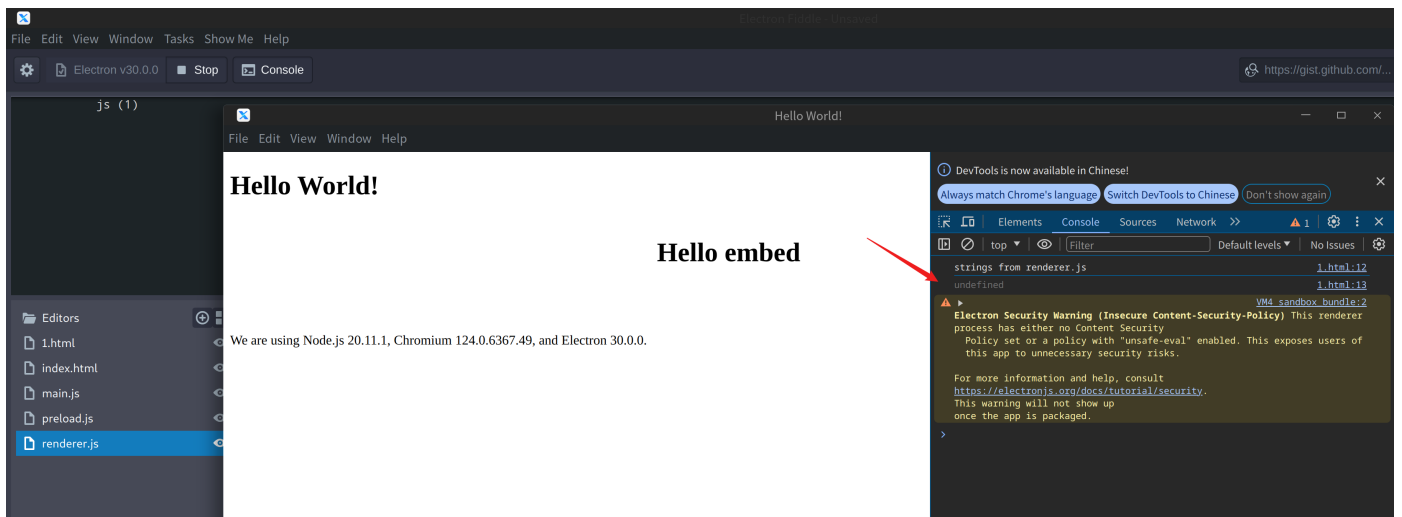
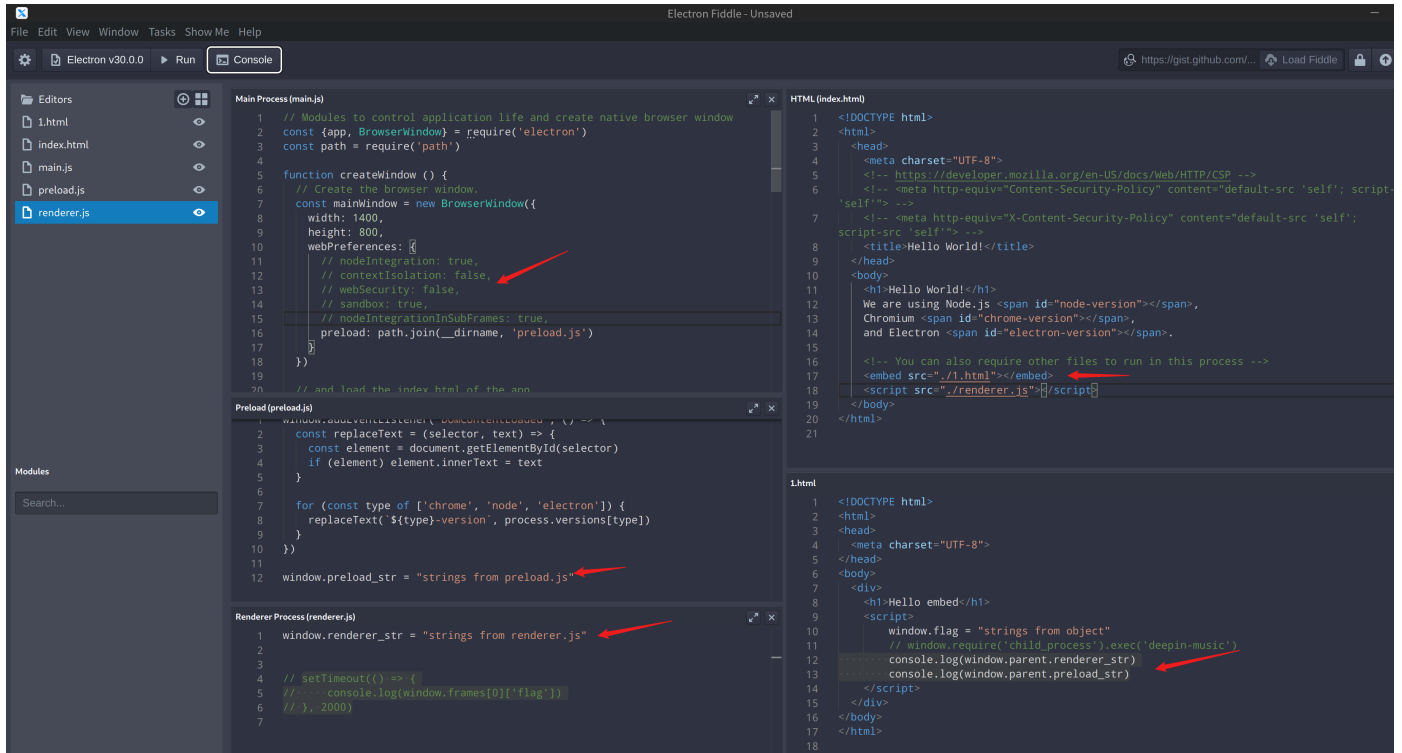
就这么几个属性

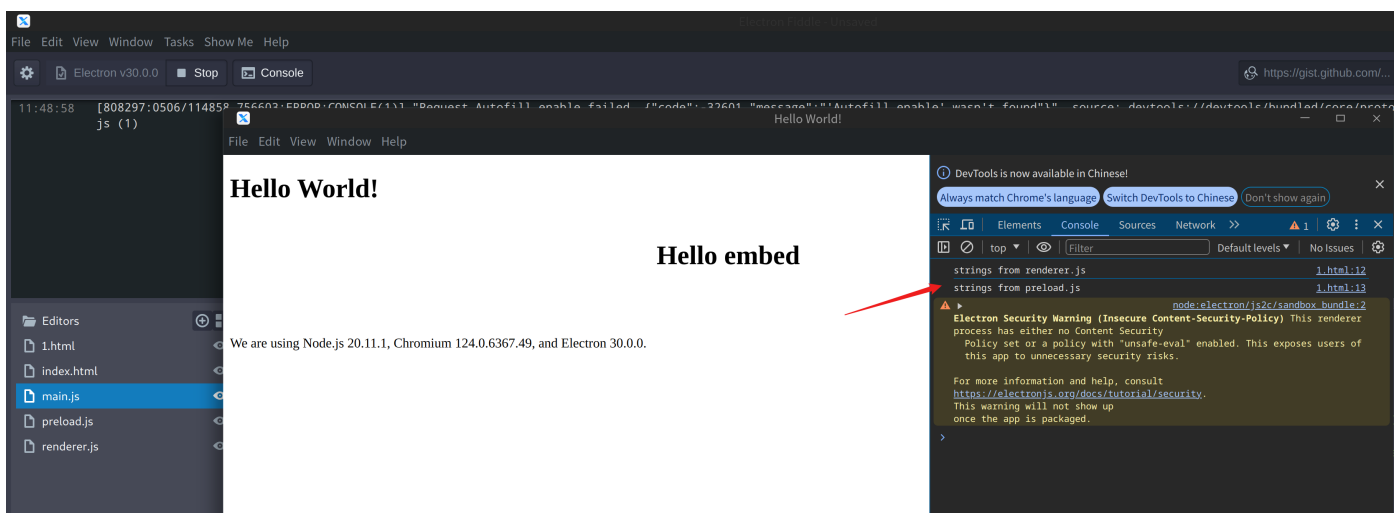
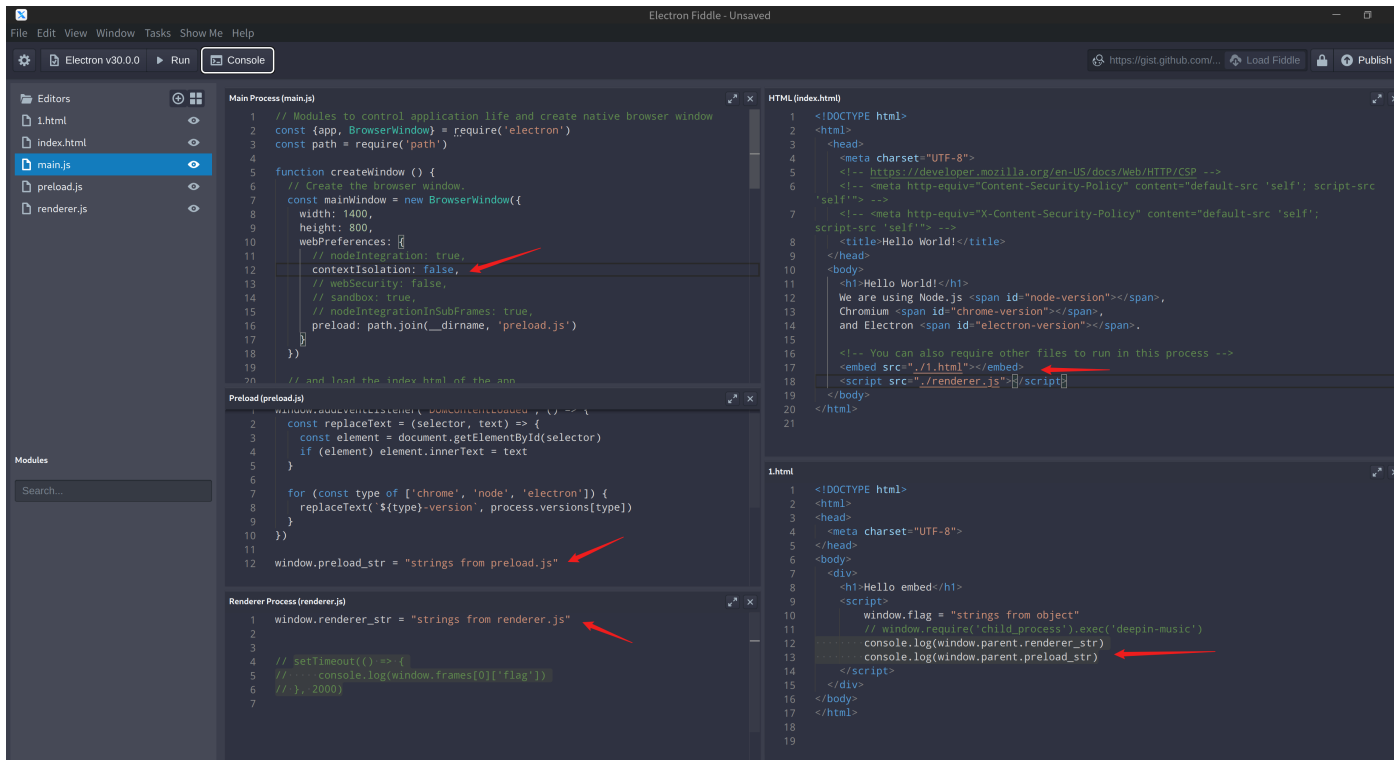
2. 渲染页面与 embed 通信及上下文

直接测试

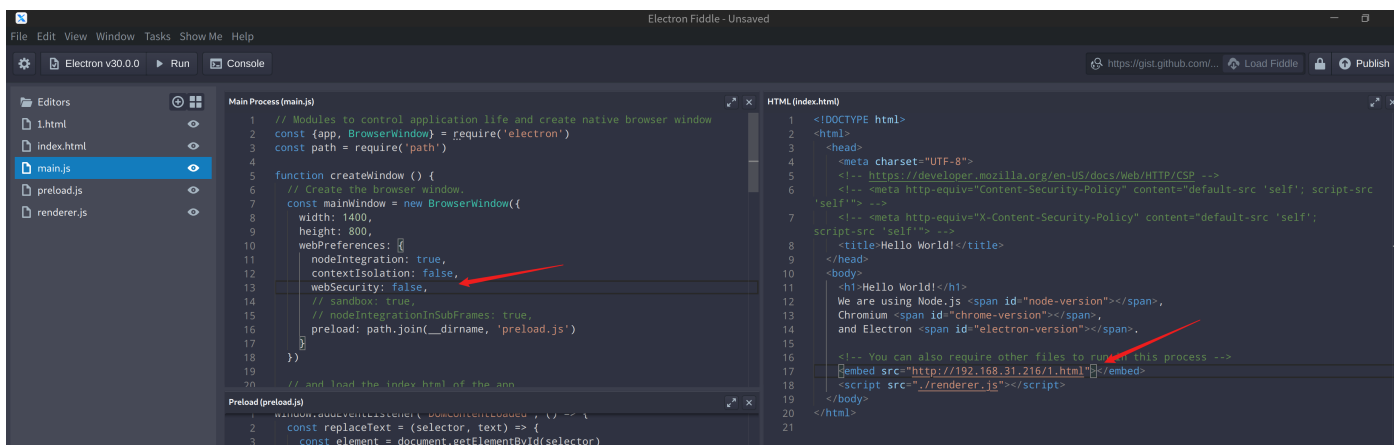
同源情况下







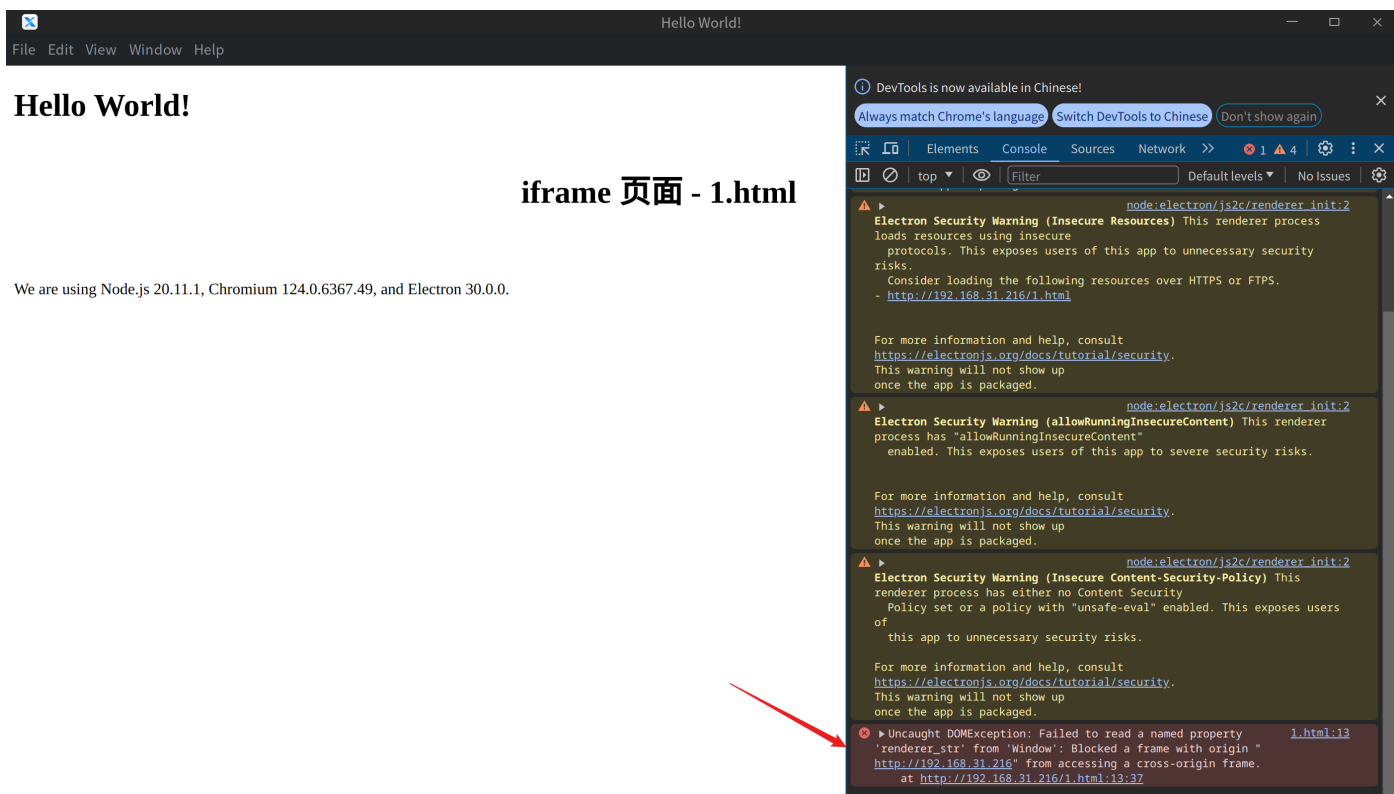
不同源



```

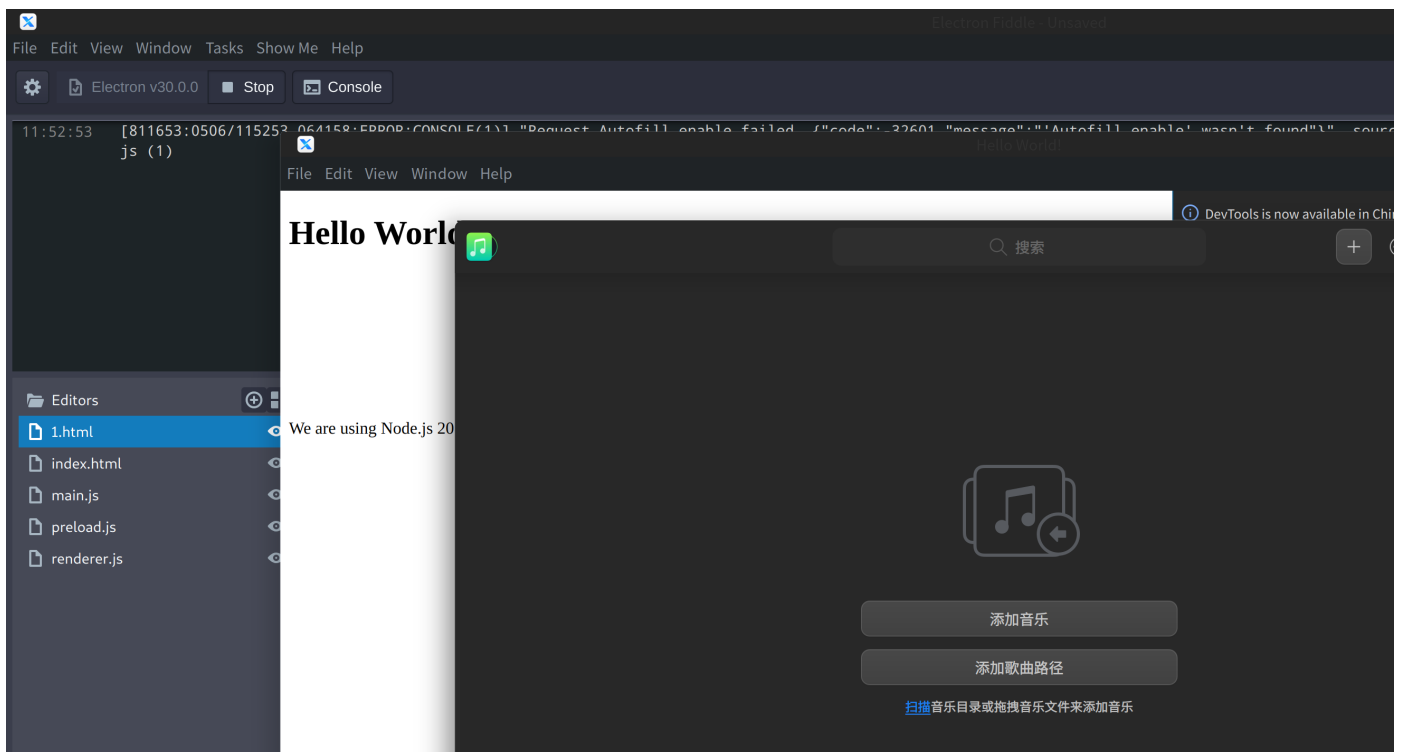
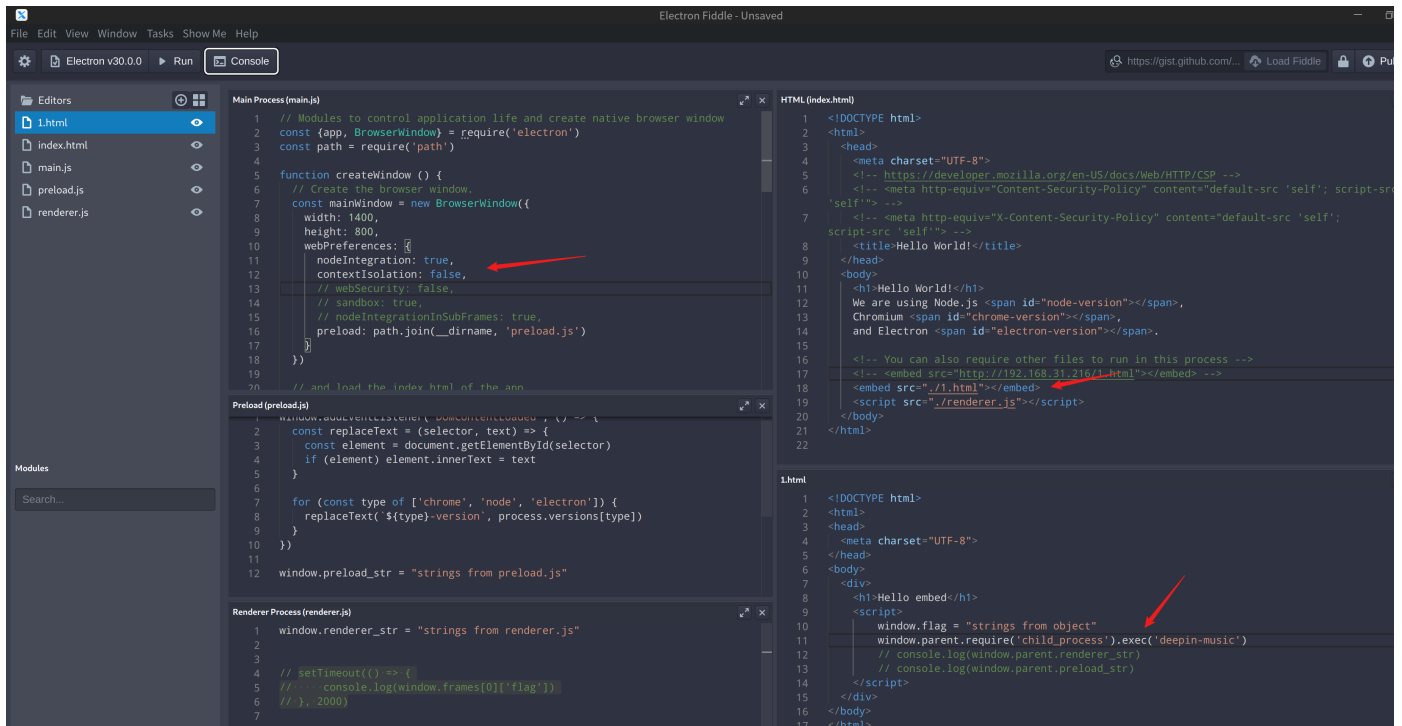
1  <!DOCTYPE html>
2  <html>
3  <head>
4  |  <meta charset="UTF-8">
5  </head>
6  <body>
7  |  <div>
8  |  |  <h1>iframe 页面 - 1.html</h1>
9  |  |  <script>
10 |  |  |  window.flag = "strings from object"
11 |  |  |  // window.require('child_process').exec('deepin-music')
12 |  |  |  setTimeout(() => {
13 |  |  |  |  console.log(window.parent.renderer_str)
14 |  |  |  |  console.log(window.parent.preload_str)
15 |  |  |  |  }, 2000)
16 |  |  |
17 |  |  </script>
18 |  </div>
19 </body>
20 </html>

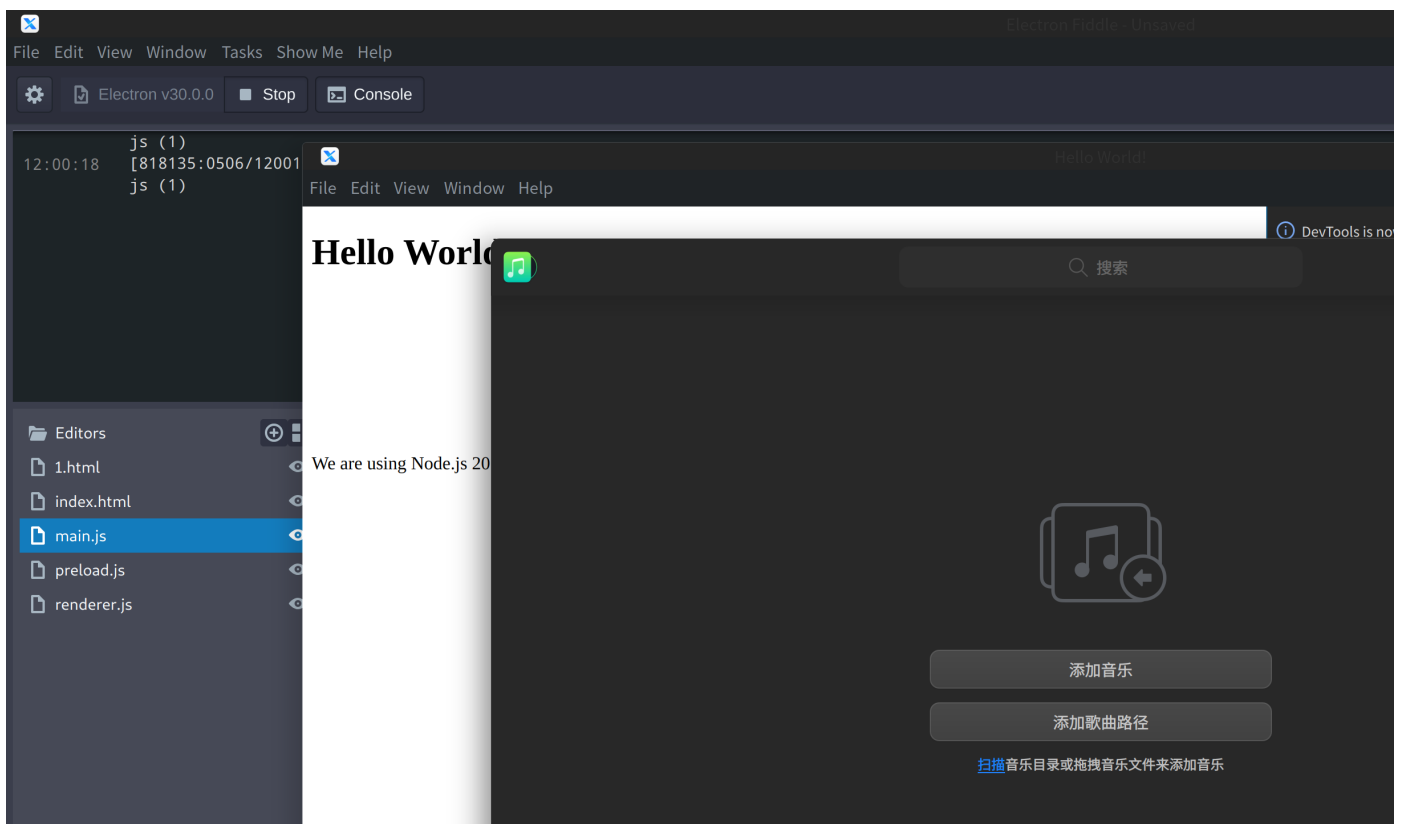
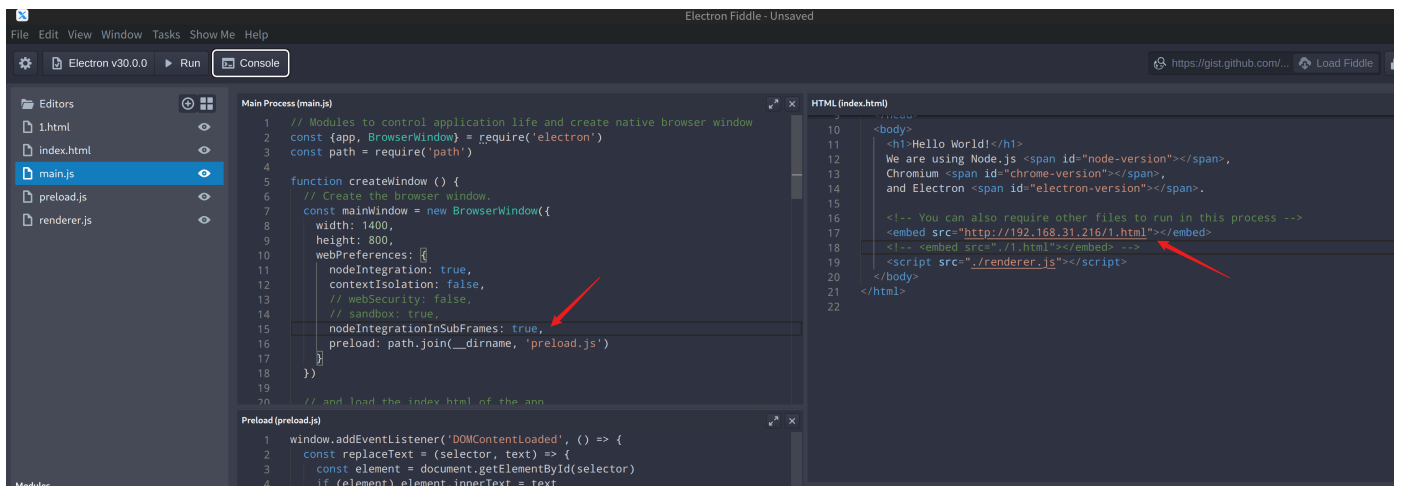
```



结果与 `object` 一致

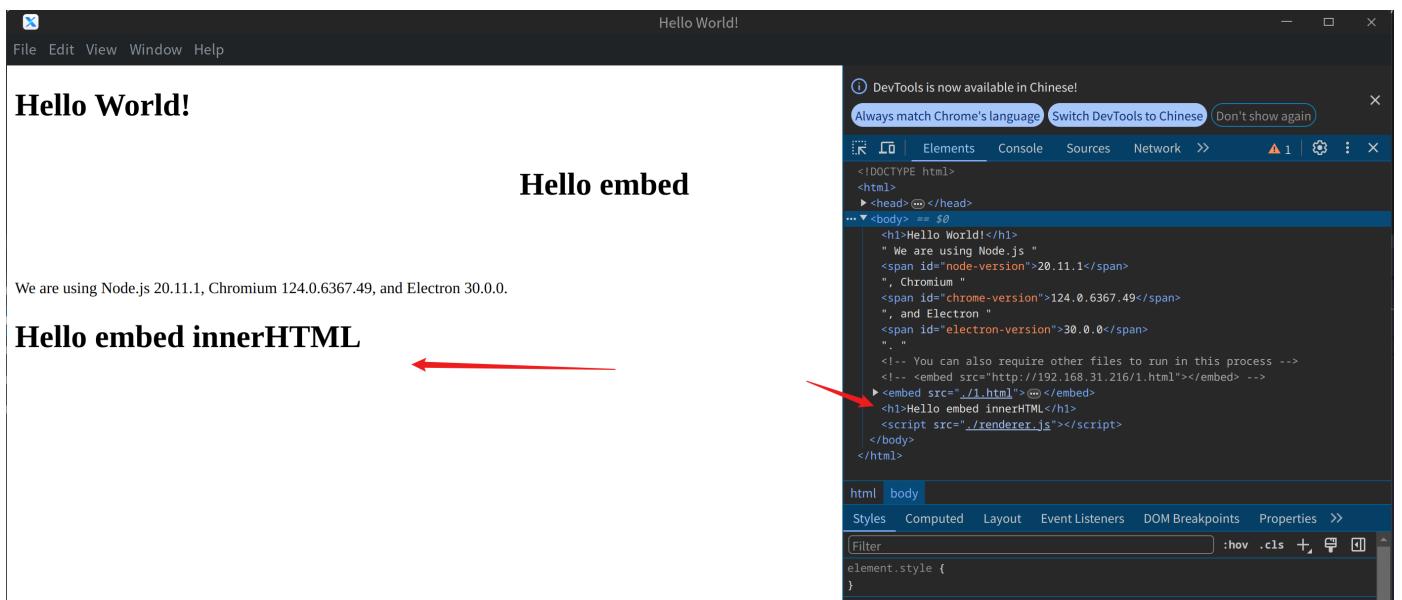
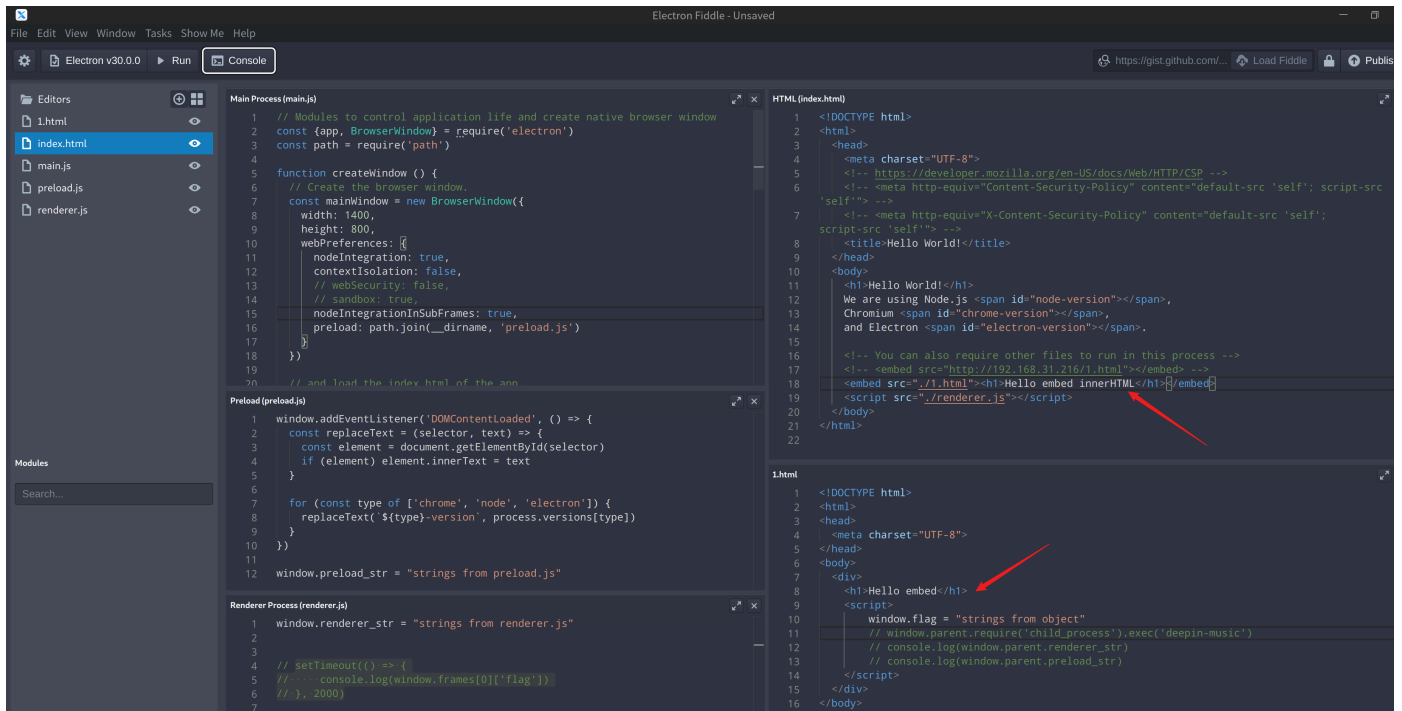
3. embed 执行 Node.js 的情况





结果与 `object` 一致

4. embed 标签内执行情况



`embed` 与 `object` 不同的是，`embed` 标签内的内容渲染时会被放到和 `embed` 标签同级

5. 小结

`embed` 标签与 `object` 标签表现基本一致，有一点不同的是，`embed` 标签内的内容渲染时会被放到和 `embed` 标签同级，而 `object` 标签内的内容渲染时会被放到 `<object>` 内部

参考文章

<https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/embed>

0x05 webview

1. 官方提示

重要提示：我们不建议您使用 `WebView`，因为这个标签会发生剧烈的结构变化，可能会影响您应用程序的稳定性。考虑切换到其他选择，如 `iframe` 和 Electron 的 `BrowserView`，或避免嵌入式内容设计的架构。

Electron 的 `webview` 标签基于 `Chromium webview`，后者正在经历巨大的架构变化。这将影响 `webview` 的稳定性，包括呈现、导航和事件路由。

2. 启用 webview

默认情况下，Electron `>= 5` 禁用 `webview` 标签。在构造 `BrowserWindow` 时，需要通过设置 `webviewTag: true` 来启用 `webview`

3. 应用效果

在一个独立的 frame 和进程里显示外部 web 内容

所以其实可以把它视为一个和主窗口一样的窗口进程看待

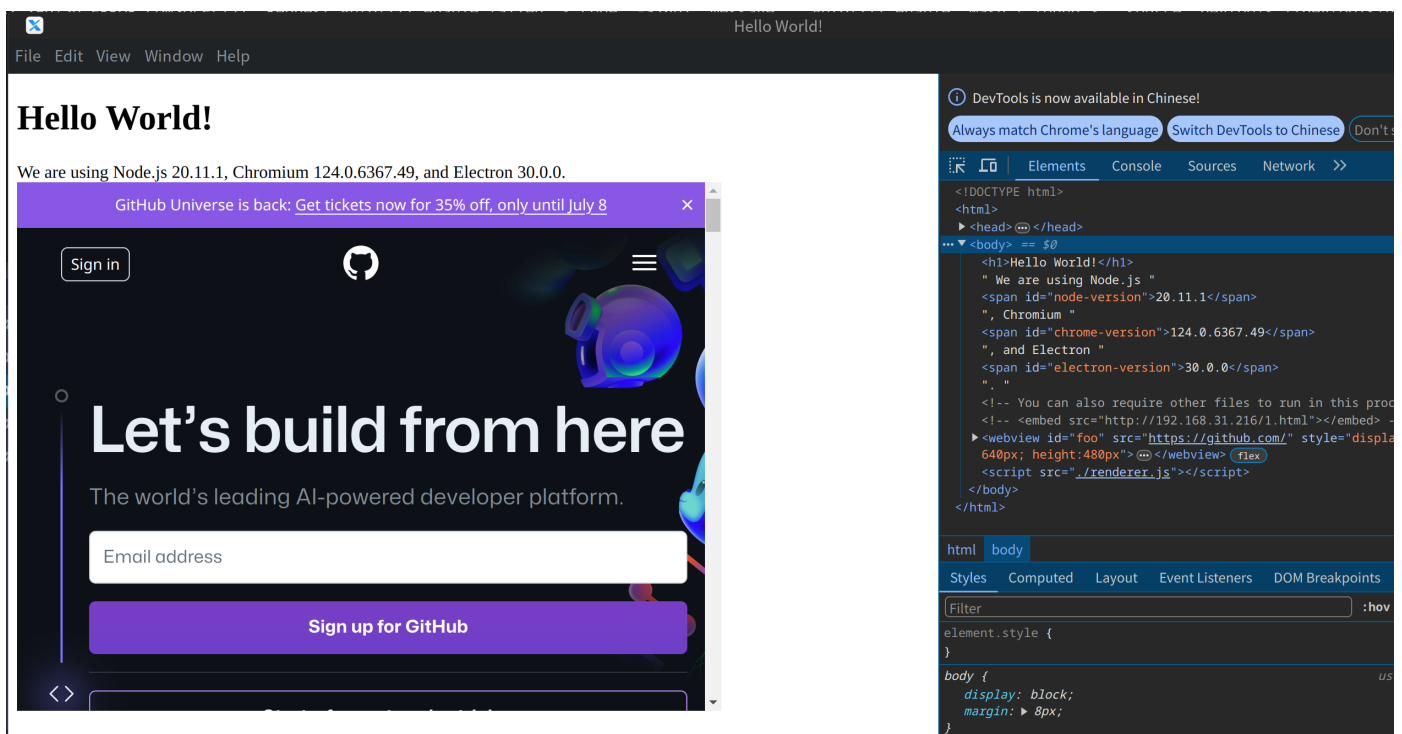
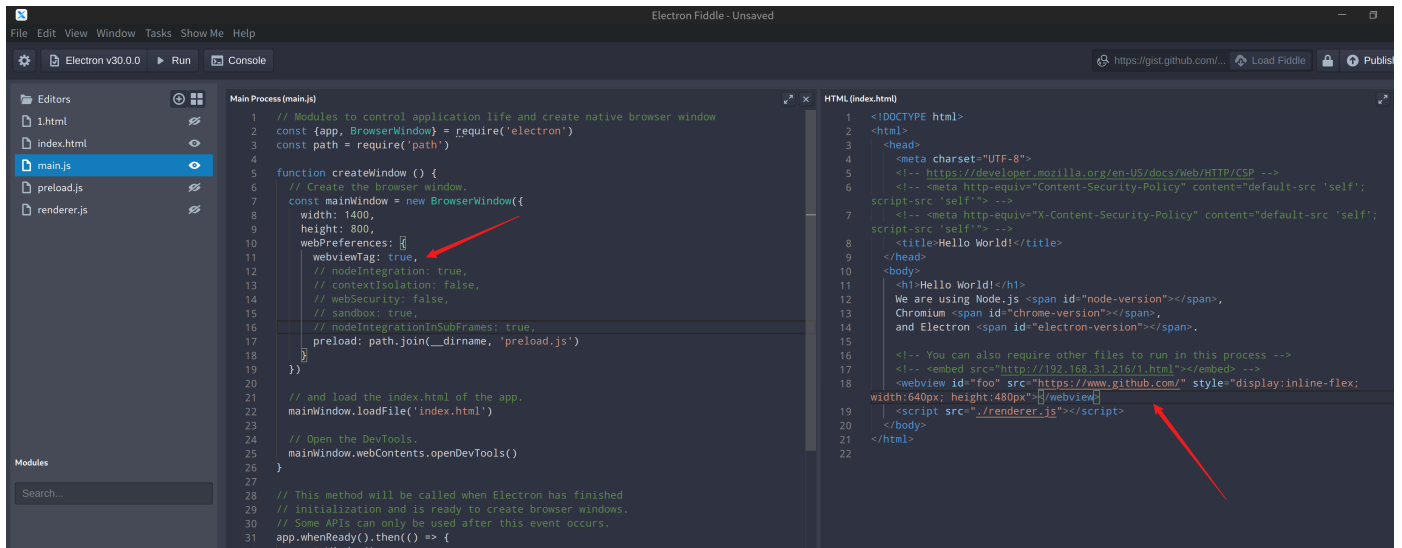
使用 `webview` 标签将'guest'内容 (例如网页) 嵌入到您的 Electron 应用中。Guest 内容包含在 `webview` 容器内。应用中的嵌入页面可以控制外来内容的布局和重绘。

与 `iframe` 不同，`webview` 独立于您的应用程序运行。它拥有和你的页面不一样的权限并且所嵌入的内容和你应用之间的交互都将是异步的。这将保证你的应用对于嵌入的内容的安全性。

注意: 从宿主页上调用 `webview` 的方法大多数都需要对主进程进行同步调用。

官方给的案例如下

```
<webview id="foo" src="https://www.github.com/" style="display:inline-flex; width:640px; height:480px"></webview>
```



4. webview 属性

它的属性大家会非常熟悉，因为和我们之前介绍的 `BrowserWindow` 中的 `webPreferences` 对应

1) src

```
<webview src="https://www.github.com/"></webview>
```

表示可见网址的 `string` 写入此属性将启动顶级跳转
更改 `src` 的值将重新加载当前页面。

`src` 属性还可以接受数据 URL, 如 `data:text/plain, Hello, world!` 。

2) nodeintegration

```
<webview src="https://www.google.com/" nodeintegration></webview>
```

加载的页面是否集成 `Node.js`

3) nodeintegrationinsubframes

```
<webview src="https://www.google.com/" nodeintegrationinsubframes>
</webview>
```

加载的页面内部的 `iframe` 等内容是否获取到 `Preload` 脚本暴露的内容

4) plugins

```
<webview src="https://www.github.com/" plugins></webview>
```

加载的页面是否可以使用浏览器插件

5) preload

```
<!-- 来自文件 -->
<webview src="https://www.github.com/" preload="./test.js"></webview>
<!-- 或从asar归档文件中加载 -->
<webview src="https://www.github.com/" preload="./app.asar/test.js">
</webview>
```

预加载脚本

6) httpreferrer

```
<webview src="https://www.github.com/" httpreferrer="https://example.com/">
</webview>
```

为访客页面设置 referrer URL 的 `string`

7) useragent

```
<webview src="https://www.github.com/" useragent="Mozilla/5.0 (Windows NT
6.1; WOW64; Trident/7.0; AS; rv:11.0) like Gecko"></webview>
```

设置加载页面时使用的 User-Agent

8) disablewebsecurity

```
<webview src="https://www.github.com/" disablewebsecurity></webview>
```

关闭安全策略，默认是开启安全策略的，只有当设置 `disablewebsecurity` 时才关闭安全策略

9) partition

```
<webview src="https://github.com" partition="persist:github"></webview>
<webview src="https://electronjs.org" partition="electron"></webview>
```

设置页面使用的会话的 `string`。

如果 `partition` 以 `persist:` 开头, 该页面将使用持续的 session, 并在所有页面生效, 且使用同一个 `partition`. 如果没有 `persist:` 前缀, 页面将使用 in-memory session. 通过分配相同的 `partition`, 多个页可以共享同一会话。如果没有设置 `partition`, app 将会使用默认的 session。

10) allowpopups

```
<webview src="https://www.github.com/" allowpopups></webview>
```

如果该属性存在，加载的页面将允许打开新窗口。Popup 默认是禁用状态

11) webpreferences

```
<webview src="https://github.com"
webpreferences="allowRunningInsecureContent, javascript=no"></webview>
```

`string` 是一个由逗号分割的字符串列表，其中指定了要设置在 webview 上的 Web 首选项。支持的首选项字符串的完整列表，请查看 [BrowserWindow](#)

此外，`webview` 的 `webpreferences` 还支持以下字符串

- `transparent` boolean (optional) - 加载的页面是否使用透明背景

12) enableblinkfeatures

```
<webview src="https://www.github.com/"
enableblinkfeatures="PreciseMemoryInfo, CSSVariables"></webview>
```

一个字符串列表，表示要启用的 `blink` 特性，这是指启用 `blink` 引擎的特性，属于 `Chromium` 的范畴

13) disableblinkfeatures

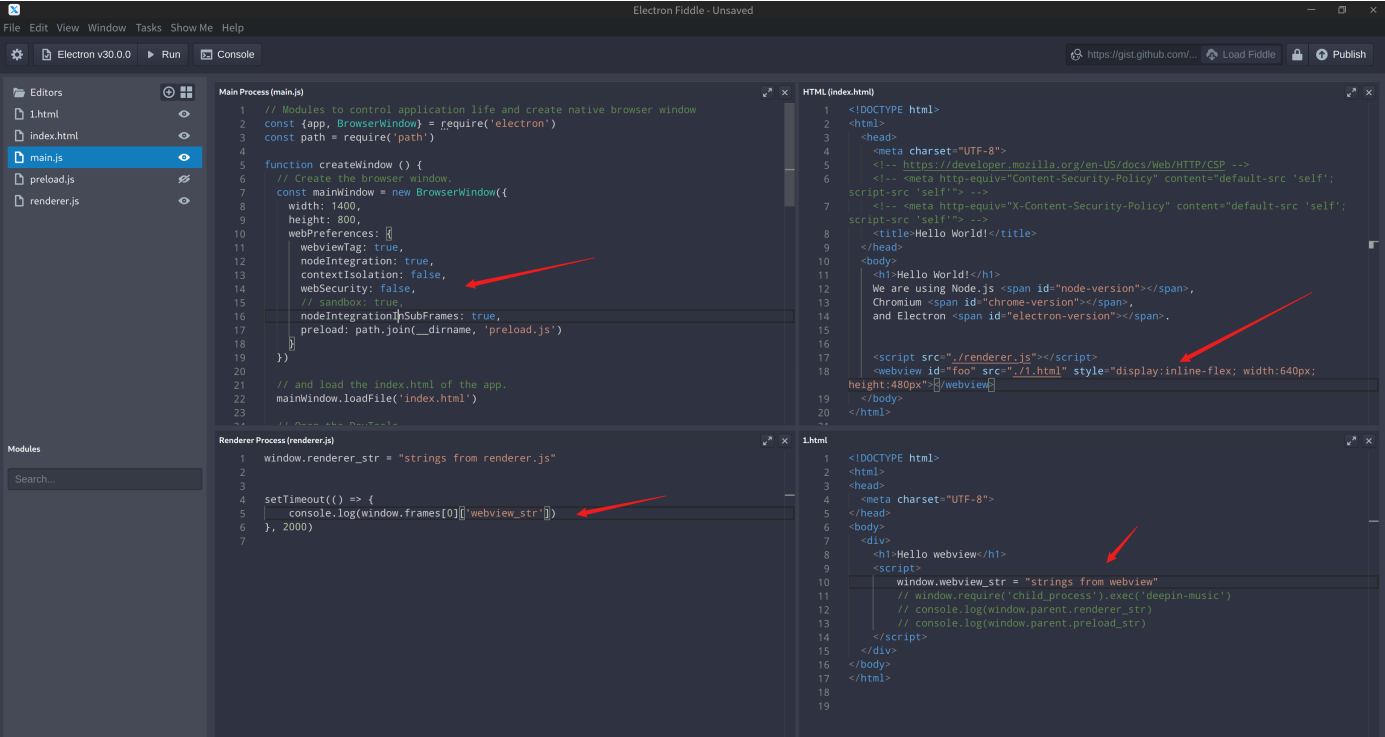
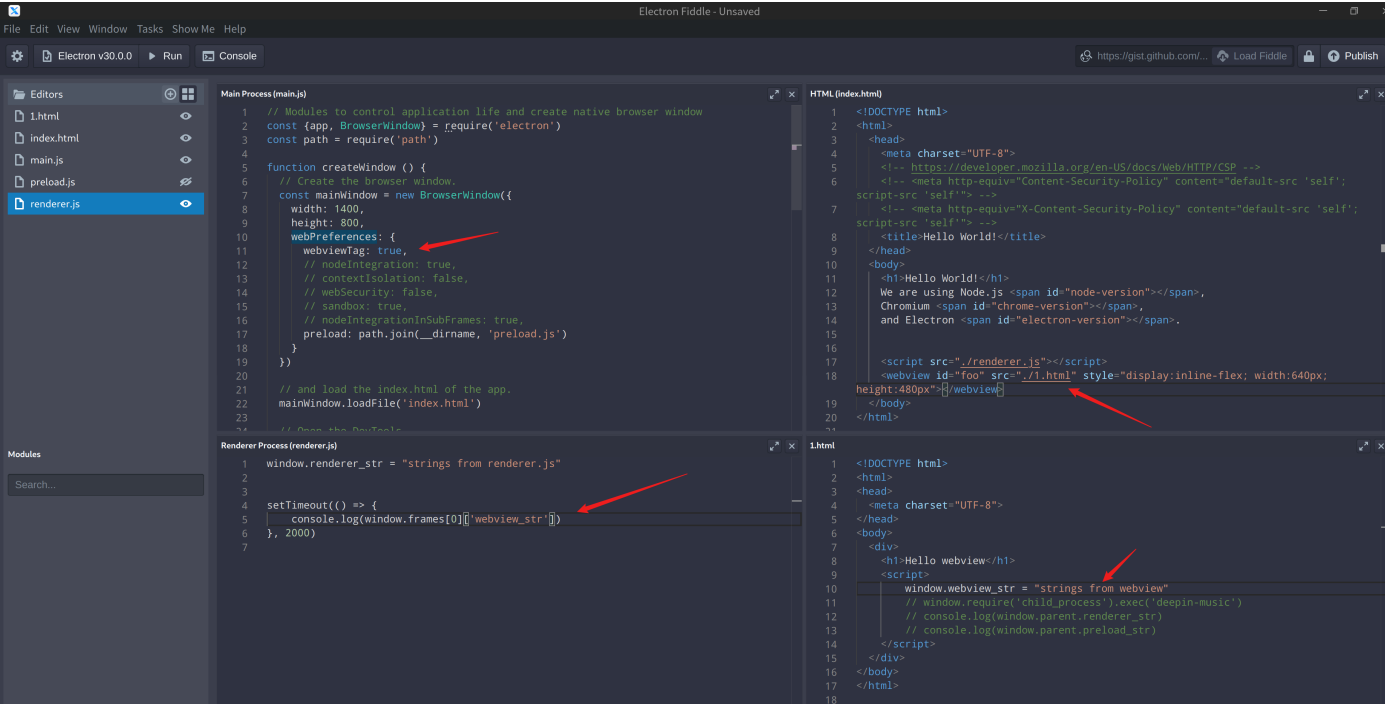
```
<webview src="https://www.github.com/"
disableblinkfeatures="PreciseMemoryInfo, CSSVariables"></webview>
```

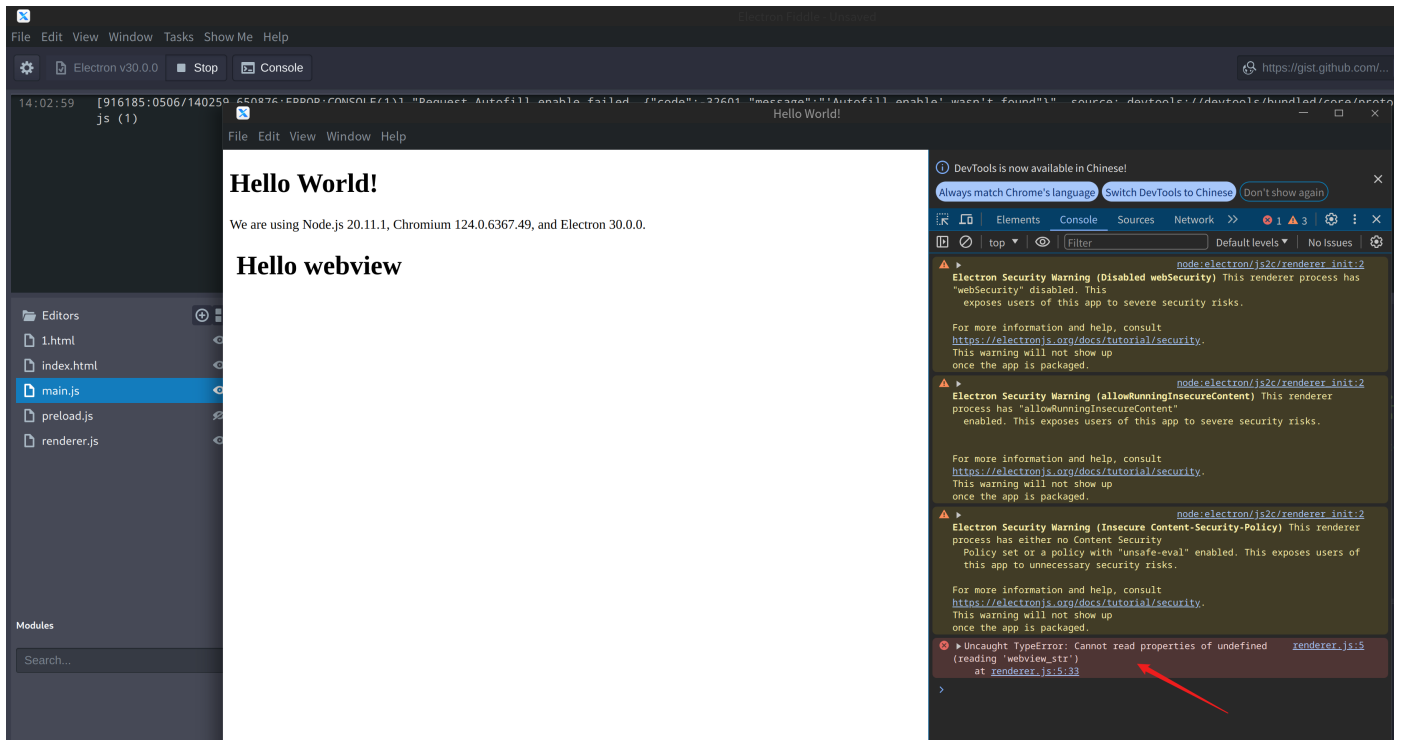
禁用 `blink` 引擎特性的列表

从上面的内容大家可以看出，webview 就相当于一个 BrowerWindow ，所以它还包含大量的方法和 DOM 事件，量太大了，不一一列举了，具体可以看下方参考文章

5. 渲染页面与 webview 上下文情况

从官方描述来看，应该是没有上下文关联的，但是我们还是试试





即使是安全策略全都关掉， `webview` 本身还是独立的上下文

6. webview 执行 Node.js 的情况

经过测试，只有当 `BrowerWindow` 设置为

- `webviewTag: true`
- `nodeIntegration: true`
- `contextIsolation: false`
- `sandbox: false` 或保持默认不设置

的情况下， `webview` 的那些选项才可以生效， `nodeIntegrationInSubFrames` 并不影响 `webview` 本身执行 `Node.js`

在此基础上， `webview` 进行如下配置，加载的页面即可执行 `Node.js`

```
<webview id="foo" src="./1.html" style="display:inline-flex; width:640px; height:480px" nodeintegration webpreferences="contextIsolation=false">
</webview>
```

`webview` 配置中缺少关闭上下文隔离都不行

7. 总结

`webview` 加载页面是一个独立的上下文，想与渲染进程或主进程通信需要使用 IPC，`webview` 中的页面想要执行 `Node.js` 的前提是，外部的渲染进程可以执行 `Node.js` 并且还要加上 `webview` 自己的配置才可以

```
<webview nodeintegration webpreferences="contextIsolation=false"></webview>
```

参考文章

<https://www.electronjs.org/zh/docs/latest/tutorial/web-embeds#webview>

<https://www.electronjs.org/zh/docs/latest/api/webview-tag#warning>

0x06 WebContentsView

`WebContentsView` 和 `BaseWindow` 在 `Electron 30.0` 中添加，用来废弃并替换 `BrowserView`，它们是主进程模块，也就是说比 `webview` 更高级一层，不是渲染进程的一部分，而是由主进程直接进行管理

`WebContentsView` 可以加载一个页面，多个 `WebContentsView` 可以放入到一个 `BaseWindow` 中进行合并、分层等管理

1. 效果展示

该模块需要在 `app` 的 `ready` 事件之后运行才有效果

```

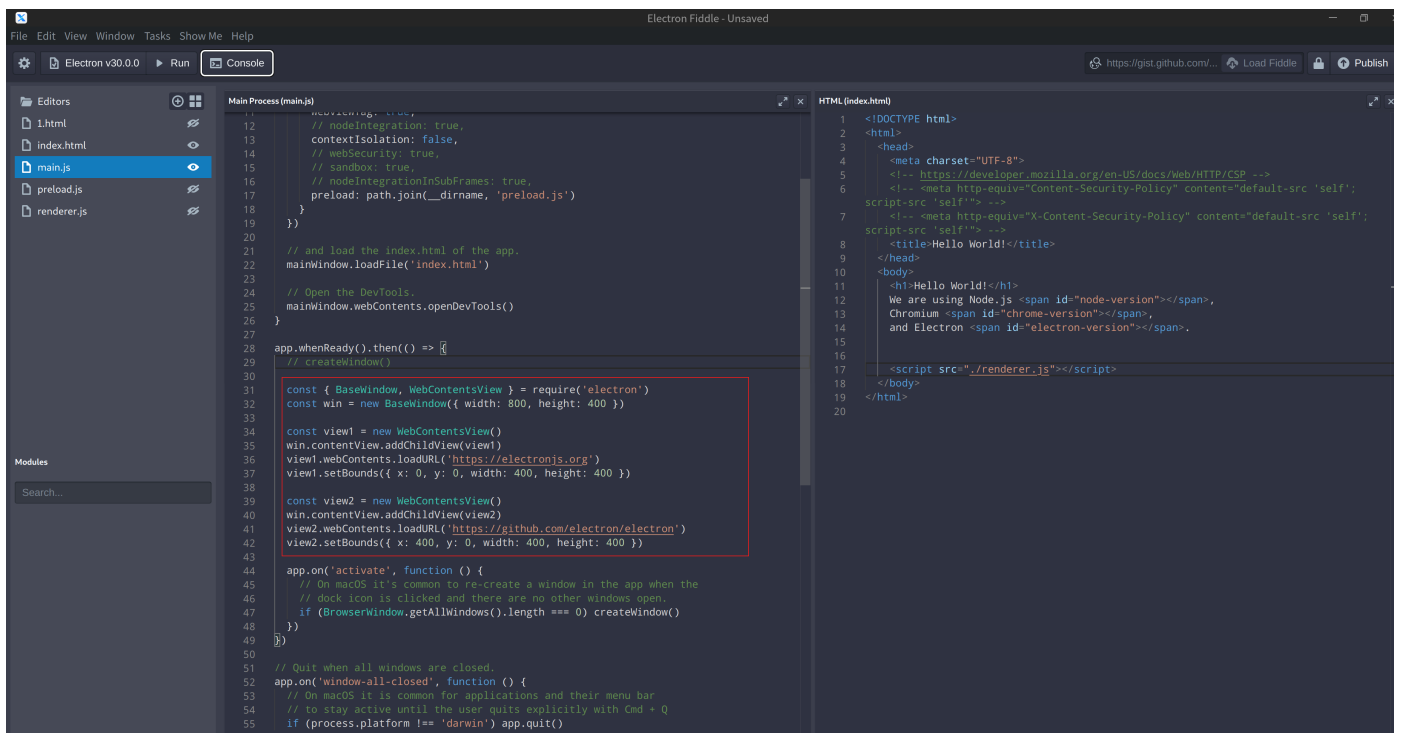
const { BaseWindow, WebContentsView } = require('electron')

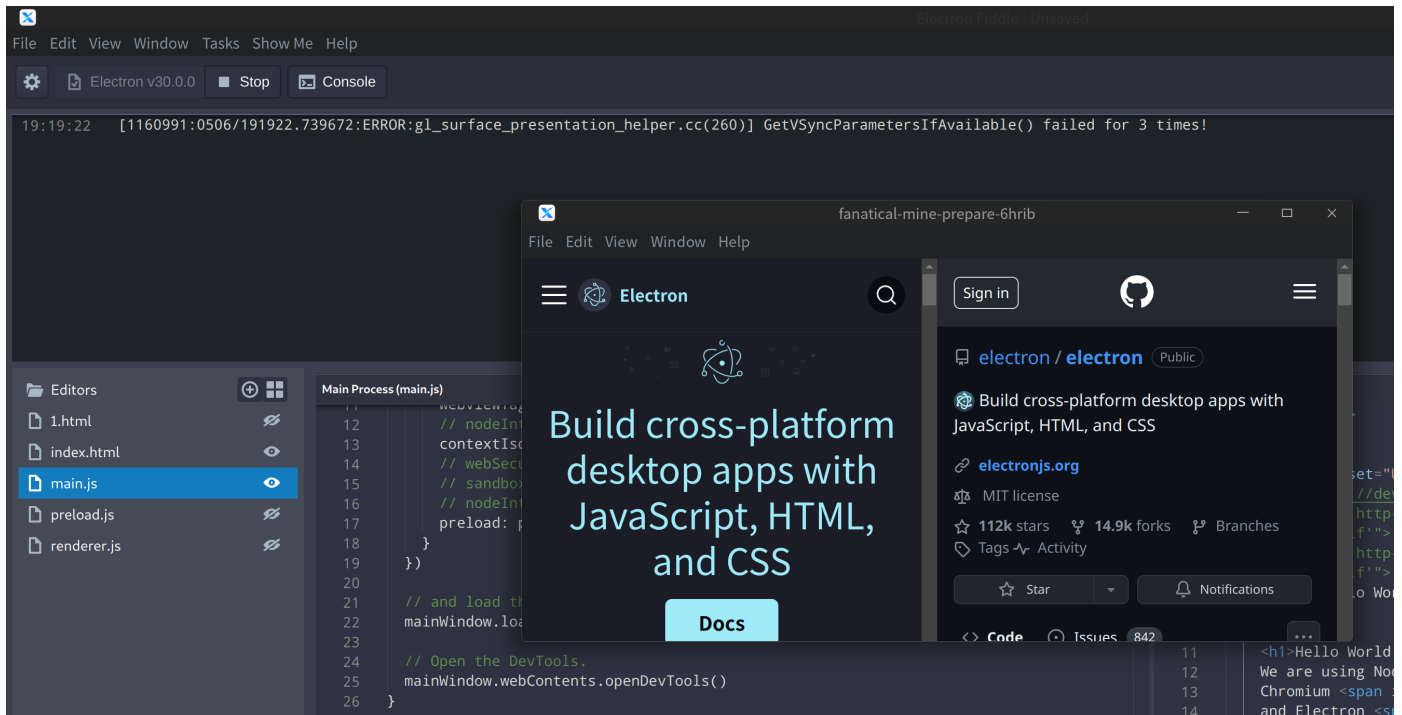
const win = new BaseWindow({ width: 800, height: 400 })

const view1 = new WebContentsView()
win.contentView.addChildView(view1)
view1.webContents.loadURL('https://electronjs.org')
view1.setBounds({ x: 0, y: 0, width: 400, height: 400 })

const view2 = new WebContentsView()
win.contentView.addChildView(view2)
view2.webContents.loadURL('https://github.com/electron/electron')
view2.setBounds({ x: 400, y: 0, width: 400, height: 400 })

```





这里我们将 `createWindow()` 注释掉了，因为从效果看，`BaseWindow` 和 `BrowerWindow` 效果是一个类型的，算是整体的窗口容器吧，而 `WebContentsView` 是在其中嵌入的内容

2. WebContentsView 属性

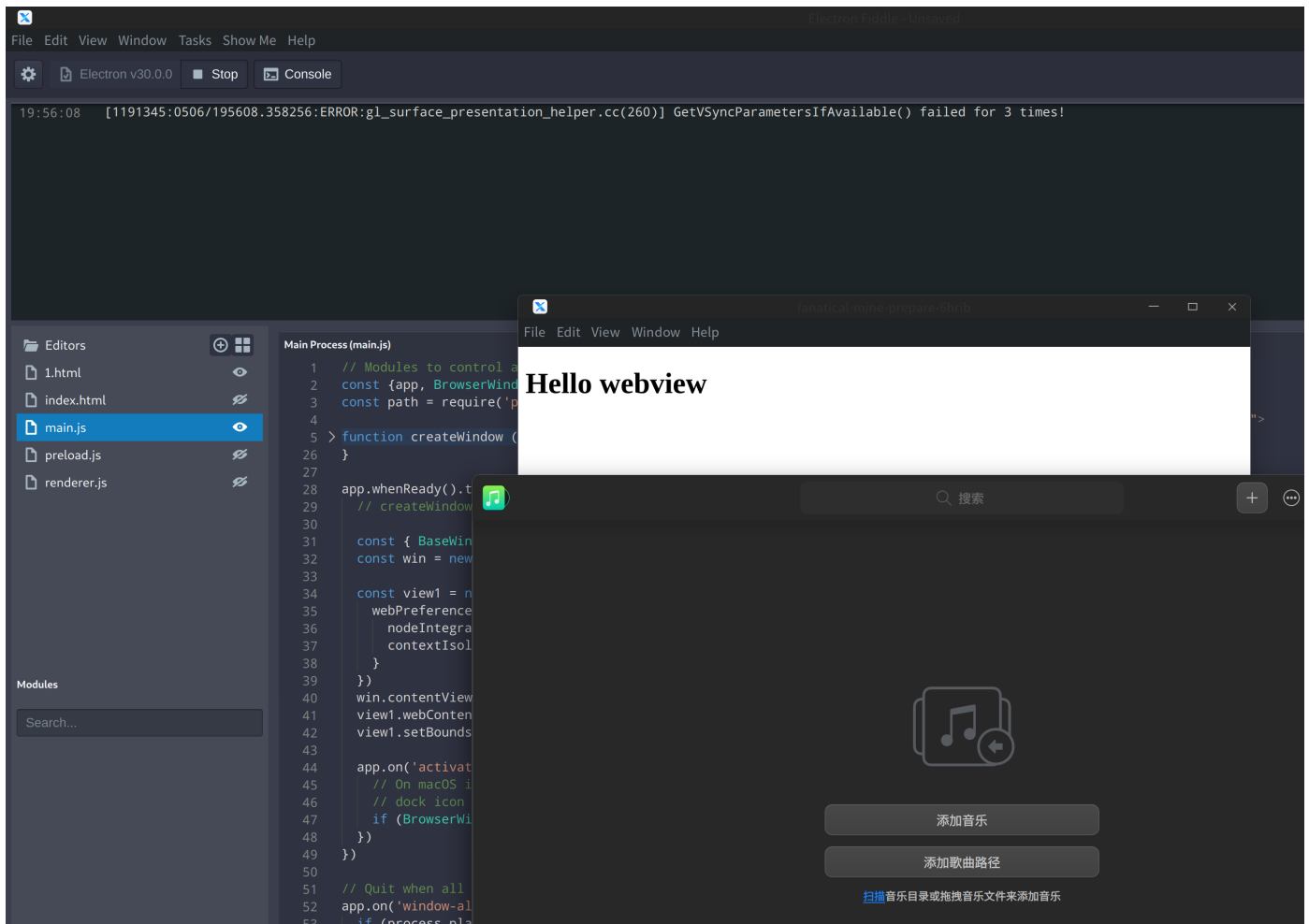
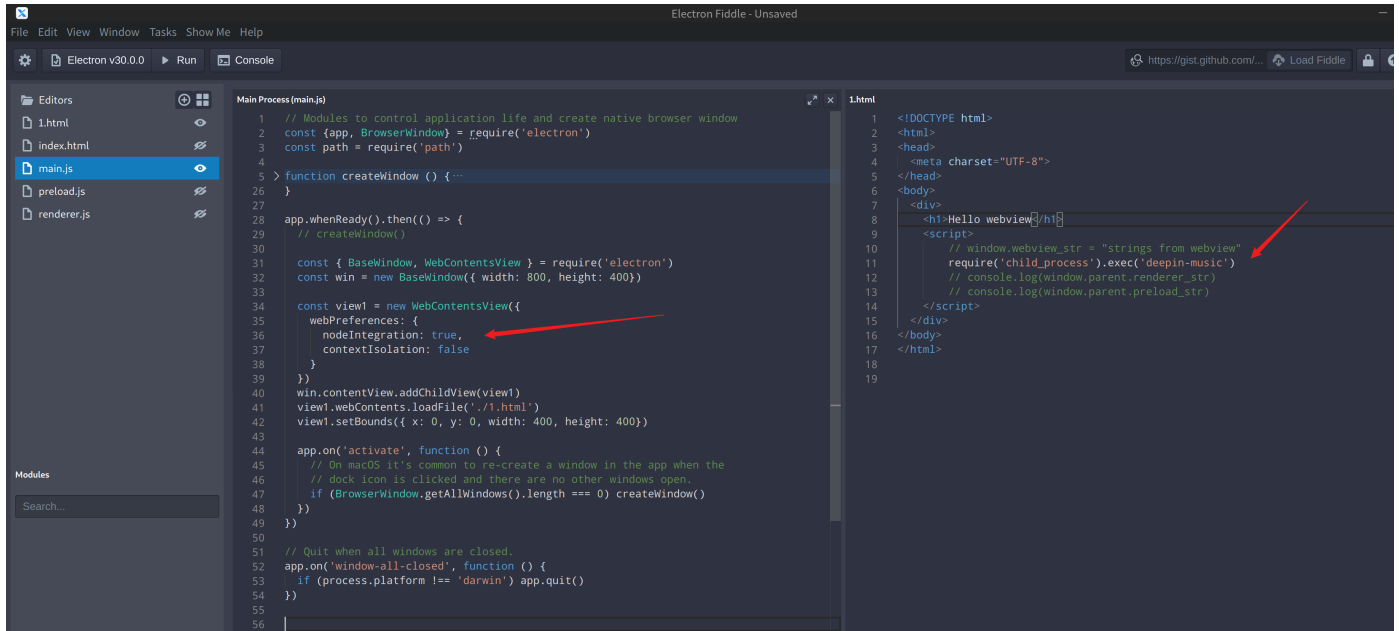
`WebContentsView` 只有一个属性，就是 `webPreferences`，就是我们创建 `BrowserWindow` 时传递安全配置那些

<https://www.electronjs.org/docs/latest/api/structures/web-preferences>

3. 上下文和 Node.js

因为 `WebContentsView` 是主进程模块，按照上面的案例，每个 `WebContentsView` 是有一个实例对象的，可以通过实例对象对其进行管理

是否能够执行 `Node.js` 主要取决于 `WebContentsView` 创建实例时 `webPreferences` 的配置了



4. 小结

从 Electron 30.0 开始 `WebContentsView` 替换了 `BrowserView`，配合 `BaseWindow` 可以有效进行集成管理，`WebContentsView` 创建实例对象时，可以指定 `webPreferences` 进行安全配置，其中配置选项和 `BrowserWindow` 一致，是否可以执行 `Node.js` 等完全取决于 `webPreferences`

参考文章

<https://www.electronjs.org/zh/blog/electron-30-0#%E9%87%8D%E7%82%B9%E5%86%85%E5%AE%B9>

<https://www.electronjs.org/zh/docs/latest/api/web-contents-view#class-webcontentsview-extends-view>

0x07 总结

本篇文章总共介绍了三种web嵌入技术，细数共 5 个

第一种包括 `iframe`、`object`、`embed`，也就是 HTML 原生支持的技术，这个在 `Electron` 中得到了很好的支持，对于它们仨，上下文及 `Node.js` 执行能力的条件基本是一致的，具体与同源还是不同源有很大关系，以 `iframe` 为例来说

如果 `iframe` 的地址与渲染页面的地址同源，则在以下安全配置时，`iframe` 内可以执行 `Node.js`

- `nodeIntegration`: `true`
- `contextIsolation`: `false`
- `sandbox` 没有显式地设置为 `true`

如果 `iframe` 的地址与渲染页面的地址不同源，则在以下安全配置时，`iframe` 内才可以执行 `Node.js`

- `nodeIntegration`: `true`
- `contextIsolation`: `false`
- `sandbox` 没有显式地设置为 `true`
- `nodeIntegrationInSubFrames`: `true`

第二种是 `webview`，属于是来源于 `Electron`（相对 web 三大件来说），存在于渲染页面之中，`webview` 加载页面是一个独立的上下文，想与渲染进程或主进程通信需要使用 IPC

`webview` 中的页面想要执行 `Node.js` 的前提是，外部的渲染进程可以执行 `Node.js` 并且还要加上 `webview` 自己的配置才可以

```
<webview nodeintegration webpreferences="contextIsolation=false"></webview>
```

第三种是 `WebContentsView`，属于是 `Electron` 主进程中管理的模块，从 `Electron 30.0` 开始，它替代了 `BrowserView`，与 `BaseWindow` 组合，达到 web 嵌入的效果

`WebContentsView` 创建实例对象时，可以指定 `webPreferences` 进行安全配置，其中配置选项和 `BrowserWindow` 一致，是否可以执行 `Node.js` 等完全取决于 `webPreferences`



微信搜一搜

🔍 NOP Team