Electron Security

ASAR完整性检查

NOP Team

ASAR 完整性检查 | Electron 安全

0x01 简介

大家好，今天和大家讨论的是 `ASAR` 完整性检查， `ASAR` 不是一种策略，而是一种文件格式

Electron 的 `asar` （Archive）是一种将多个文件和目录打包成一个单一文件的归档格式，专为 Electron 应用设计。它类似于一个压缩包，但具有特殊的设计以便于 Electron 能够直接从这样的归档文件中加载资源，而无需先解压。使用 `asar` 的主要好处包括：

1. **性能优化**：通过减少文件系统的 I/O 操作，提高应用的加载速度。因为 Electron 可以直接从单个 `asar` 文件中读取资源，而不需要遍历多个小文件。
2. **保护源代码**：将应用的源代码和资源打包进一个不可直接浏览的归档文件中，增加了一层保护，使得最终用户更难以直接查看或修改应用内部的文件结构和源代码。
3. **简化部署**：一个 `asar` 包代替了原来的多个文件和目录，使得应用的分发和更新过程更加简便。

使用 MacOS 的用户可能非常好理解，MacOS 中应用程序的后缀为 `.app` ，可以双击执行，但也可以通过右键 -> 显式包内容进入到该路径中

也有点像 Linux 中的 tar 文件，就是把一堆文件捆在一起了

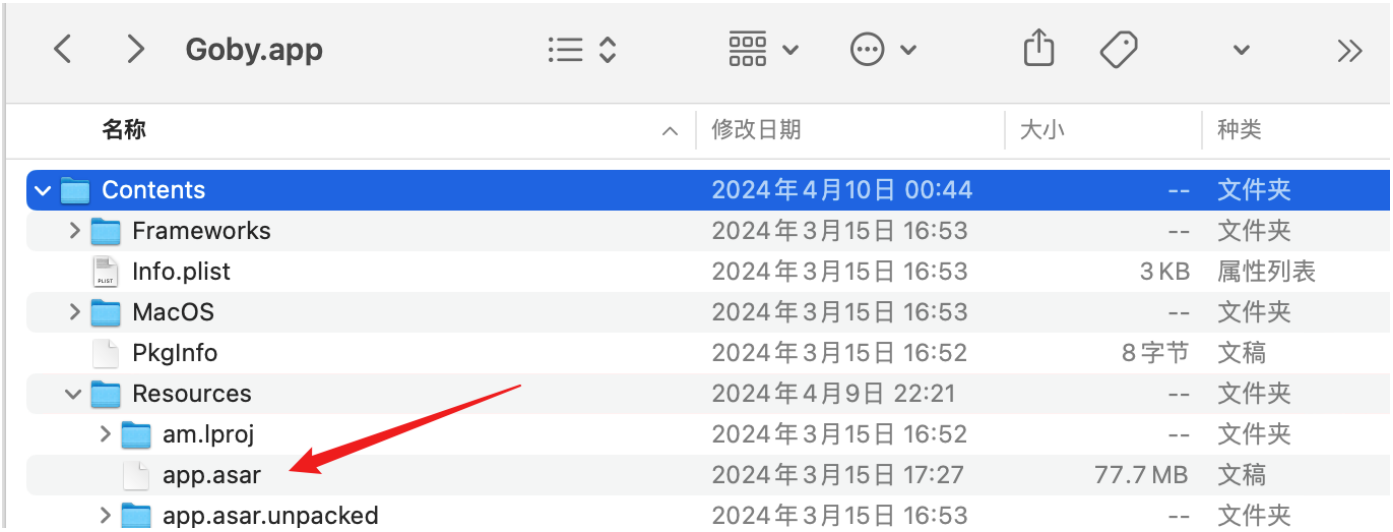
把大家捆在一起除了解决长路径问题，还有就是好做完整形检查，只需要对捆后对文件进行校验就可以了，这样可以确保程序能够及时发现源代码被篡改的情况

0x02 ASAR 文件使用方法

官方将文件捆绑在一起，要么你就搞成一个类似 `ELF` 这种文件，加载资源的时候通过偏移去查找文件，要么就是搞一个文件系统，通过引用的方式能够一一映射，如果说到本质上，肯定还是一回事

在官方的视角里，它们是将捆绑后的 `.asar` 文件视为一个具有虚拟目录的文件系统，之后官方围绕这个特性，完善了 `Node API` 和 `Web API` ，使其支持这种格式的调用，但可想而知，不可能全部都修改为支持，所以官方列出了一些可以使用的方法

在 MacOS 上， `asar` 文件位于 `/Applications/xxx.app/Contents/Resources/app.asar`



1. Node API

由于 Electron 的特殊补丁程序, Node API 比如 `fs.readFile` 和 `require` 使用 ASAR 就像是使用虚拟目录一样, 里面的文件也像是在文件系统内一样

例如，假设我们在 `/path/to` 文件夹下有个 `example.asar` 包：

1) 列出 asar 内部文件列表

```
asar list /path/to/example.asar
```

```
$ asar list /path/to/example.asar
/app.js
/file.txt
/dir/module.js
/static/index.html
/static/main.css
/static/jquery.min.js
```

```
/A/G/C/Resources ❯❯❯ asar list ./app.asar
/dist
/dist/electron
/dist/electron/01e2cda9c9d8cc91685a.css
/dist/electron/02fa85859ba87025cec8.css
/dist/electron/0358361000cdc691e4fb.css
/dist/electron/064c3638f89c85c1a776.css
/dist/electron/0825b948efe10bcd5de5.css
/dist/electron/0a037139bb6e0e8fd4fd.css
/dist/electron/0cb2c639cb172ff3cafb.css
/dist/electron/0e65386dd24af10361ff.css
/dist/electron/0f592d817ca3d923d7b6.css
/dist/electron/0f6615155bb833a58700.css
/dist/electron/10e5b1117e8ffa63f60f.css
/dist/electron/120c4fe4240d6bf22b26.css
```

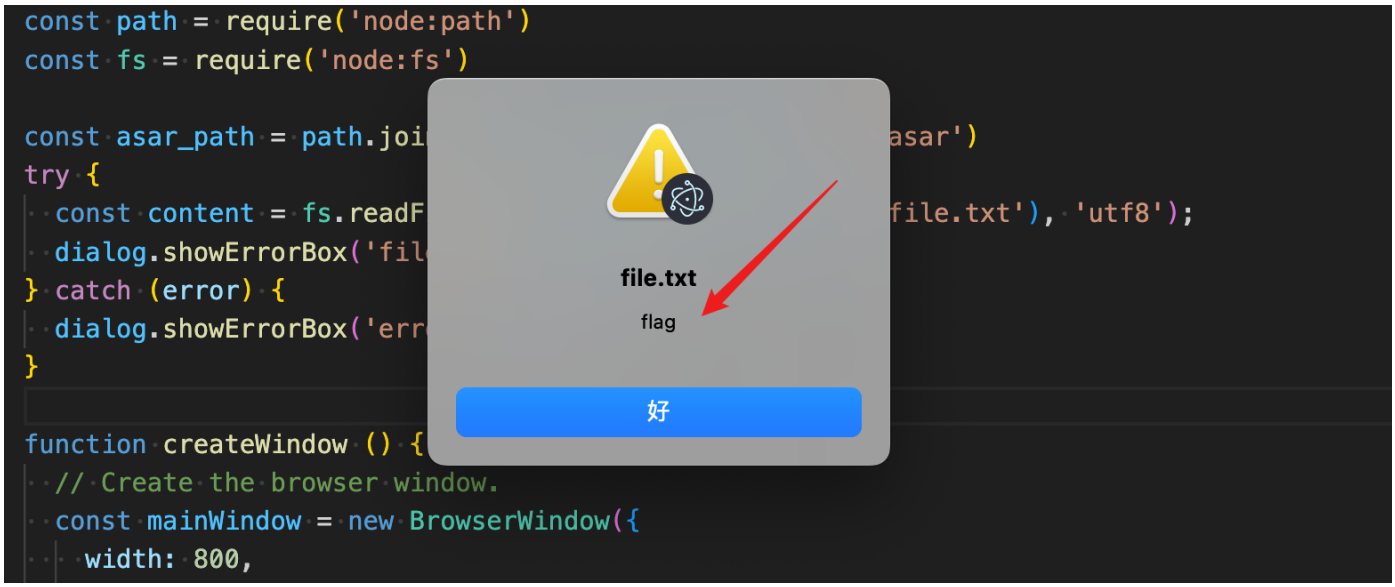
2) 读取 asar 中的文件

```
const fs = require('node:fs')
fs.readFileSync('/path/to/example.asar/file.txt')
```

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

const asar_path = path.join(process.resourcesPath, 'app.asar')
try {
  const content = fs.readFileSync(path.join(asar_path, 'file.txt'), 'utf8');
  dialog.showErrorBox('file.txt', content)
} catch (error) {
  dialog.showErrorBox('error', error)
}

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })
```

`file.txt` 里放入字符 `flag`，通过 `process.resourcesPath` 可以返回资源路径，拼接后就是当前路径

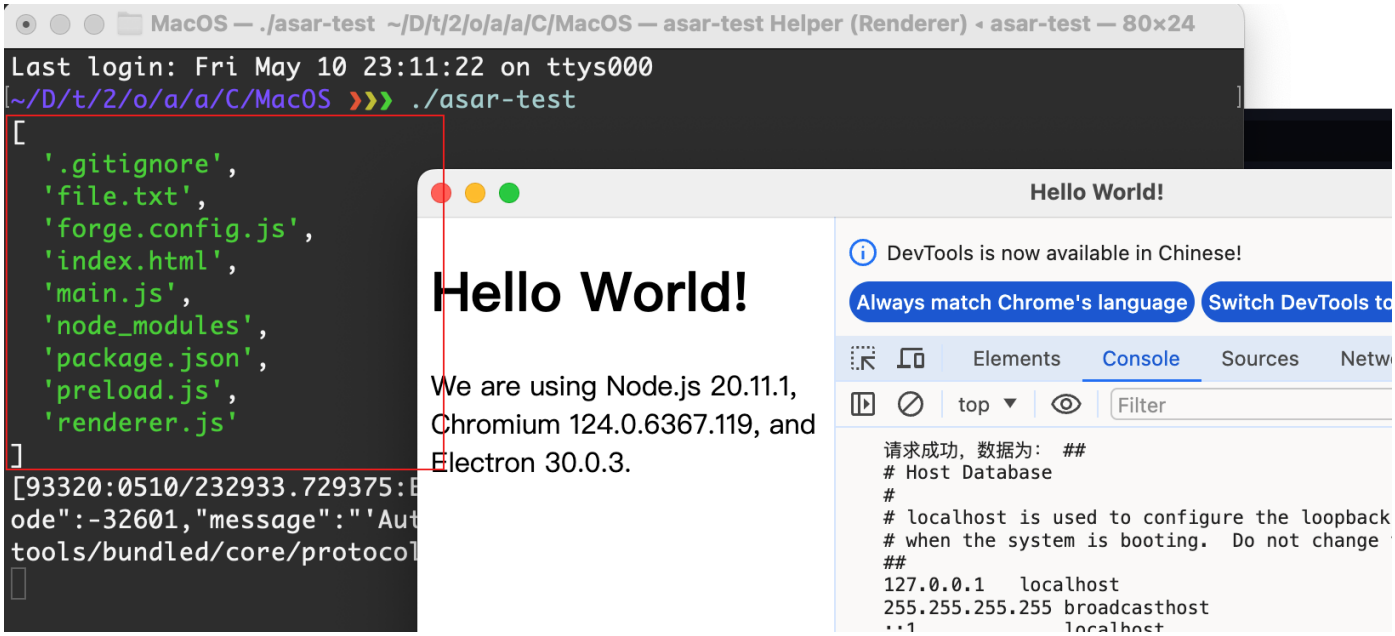
打包后，执行，看看是否能够成功输出



3) 获取 asar 根目录下的所有文件

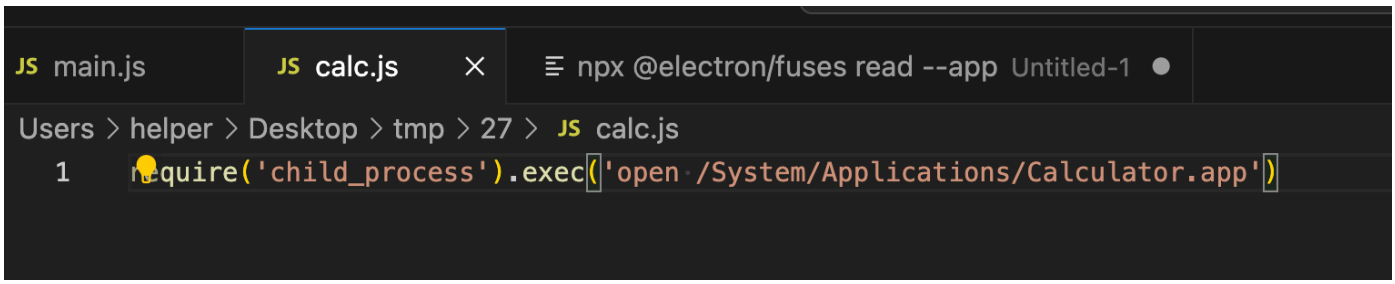
```
const fs = require('node:fs')
fs.readdirSync('/path/to/example.asar')
```

```
const asar_path = path.join(process.resourcesPath, 'app.asar')
try {
  const content = fs.readdirSync(asar_path);
  console.log(content)
  // dialog.showErrorBox('asar content', content)
} catch (error) {
  dialog.showErrorBox('error', error)
}
```

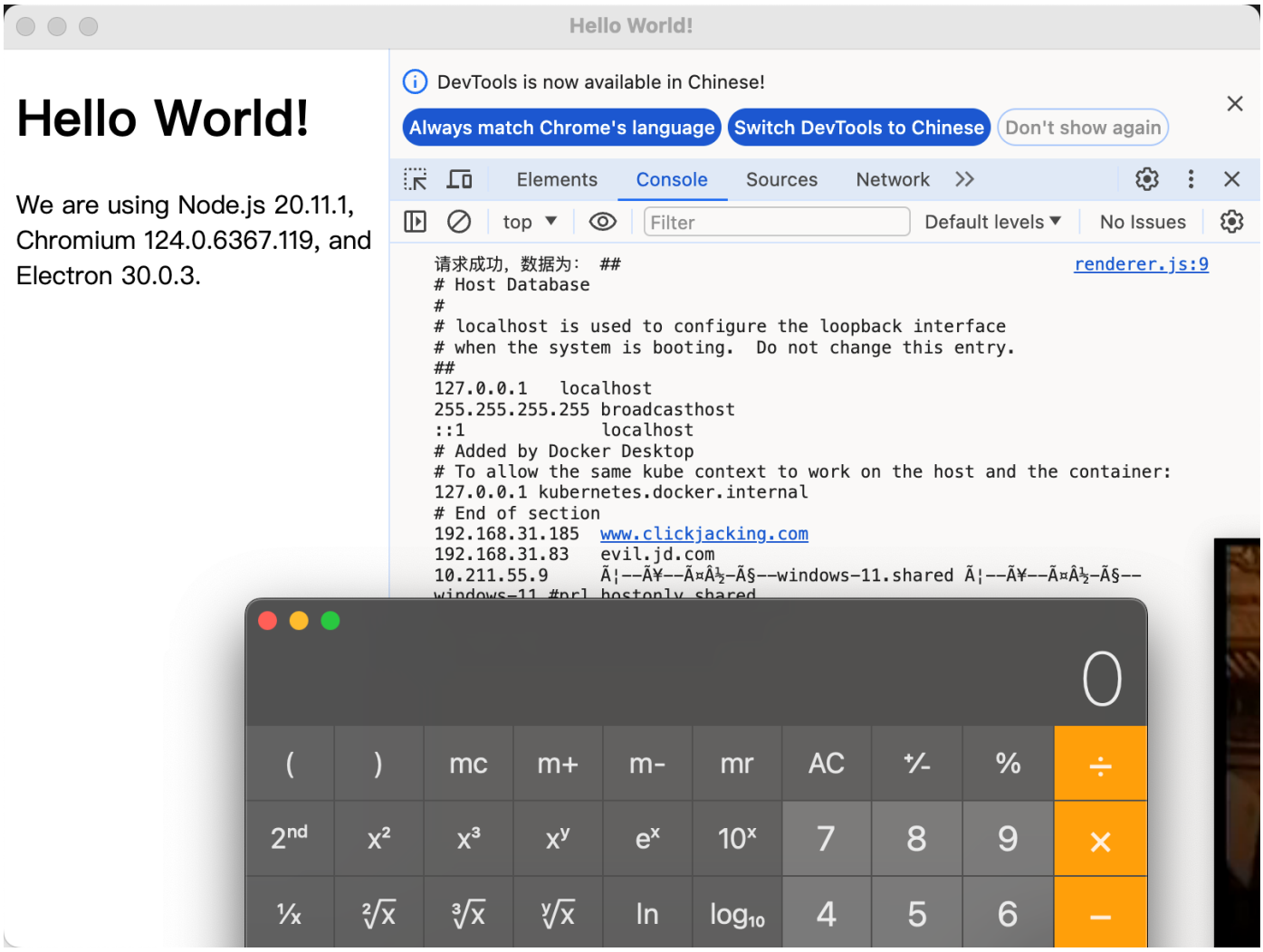


4) 使用 asar 中的模块

```
require('./path/to/example.asar/dir/module.js')
```



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

const asar_path = path.join(process.resourcesPath, 'app.asar')
try {
  require(path.join(asar_path, 'calc.js'))
  // const content = fs.readdirSync(asar_path);
// console.log(content)
  // dialog.showErrorBox('asar content', content)
} catch (error) {
  dialog.showErrorBox('error', error)
}

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
```

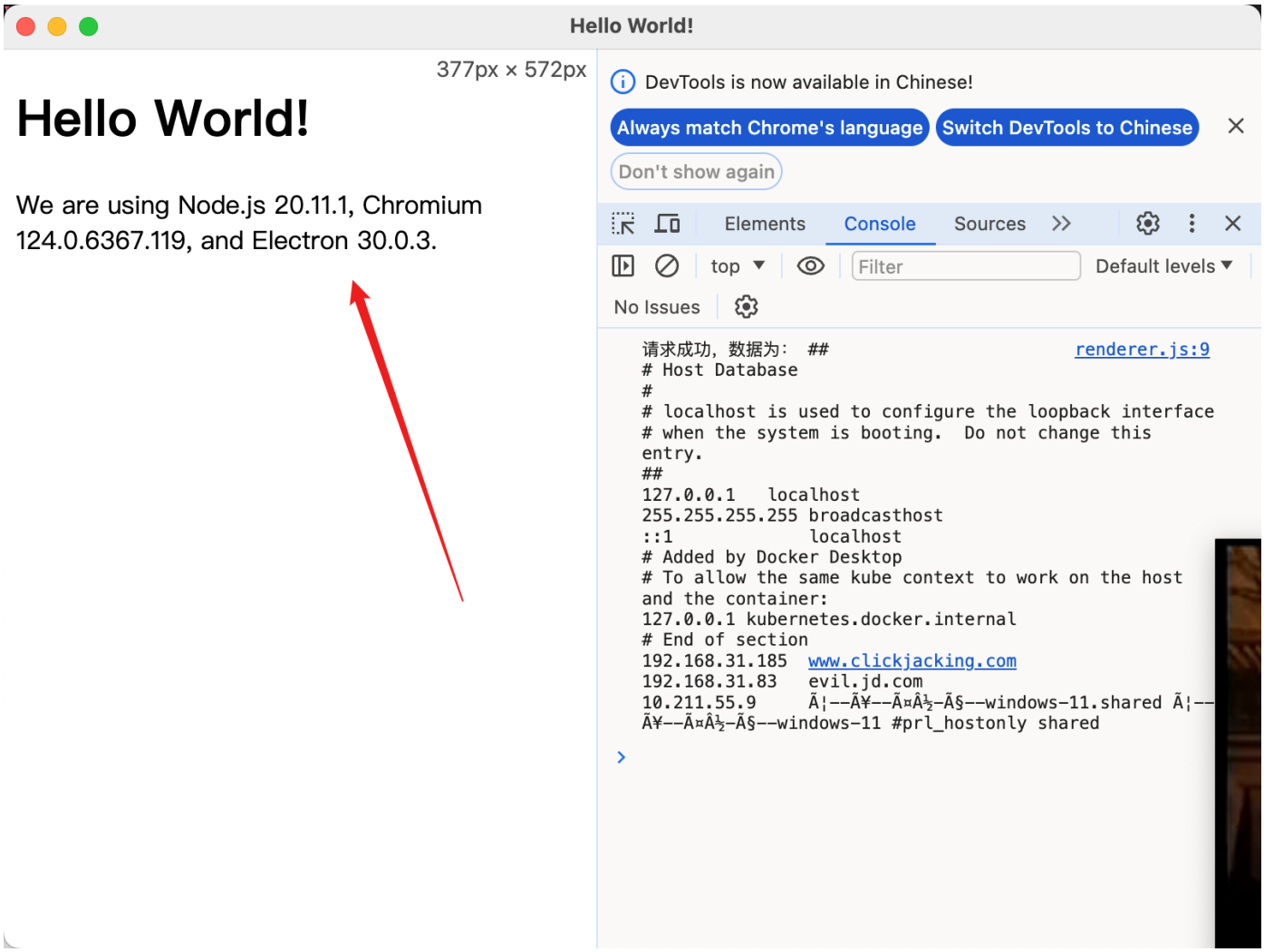


成功加载模块

5) 加载 asar 中的网页

```
const { BrowserWindow } = require('electron')
const win = new BrowserWindow()

win.loadURL('file:///path/to/example.asar/static/index.html')
```

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

const asar_path = path.join(process.resourcesPath, 'app.asar')

function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile(path.join(asar_path, 'index.html'))

```



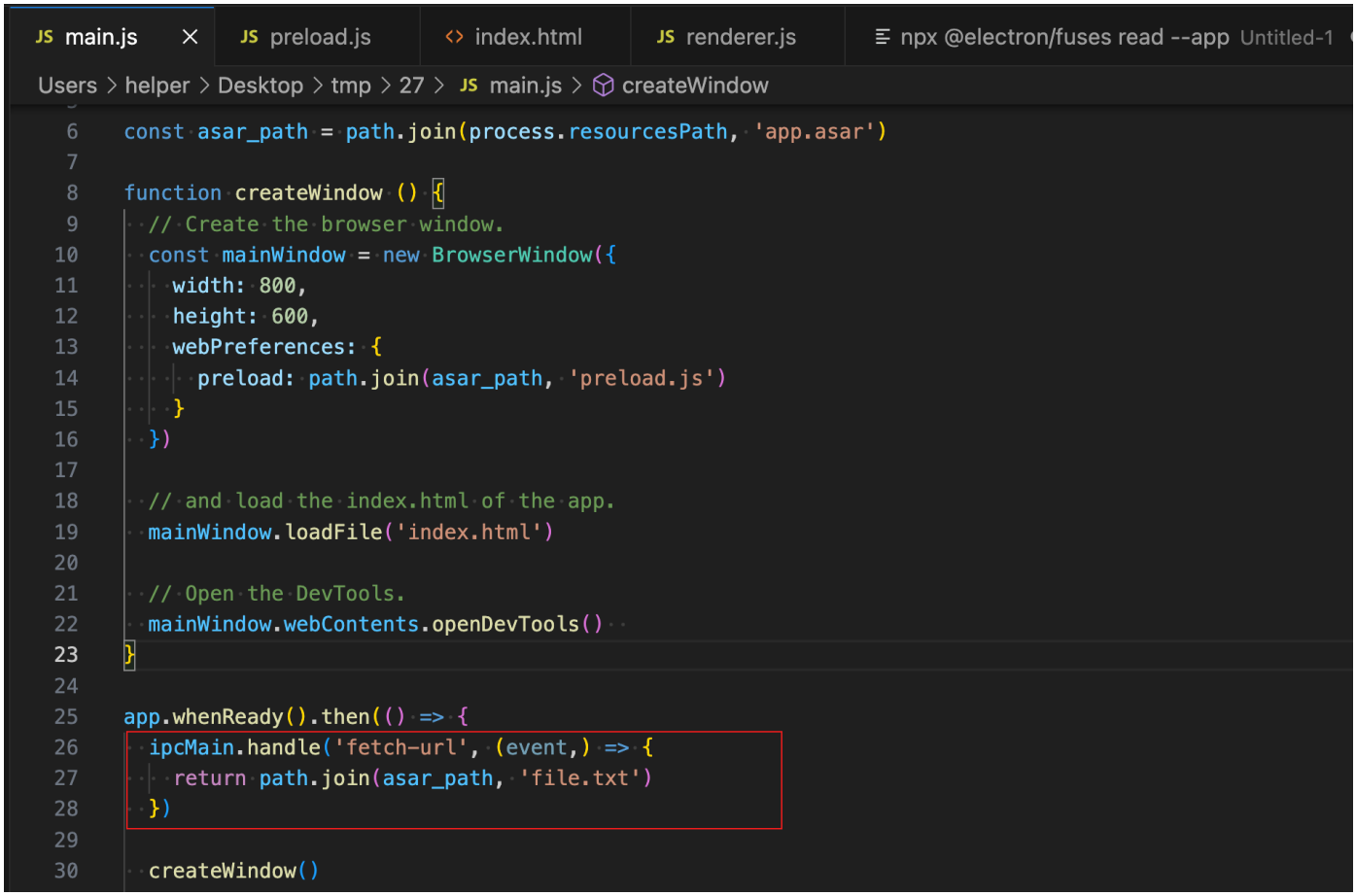
成功加载

2. Web API

1) 将 asar 文件视为文件夹

在网页中，可以使用 `file:` 协议请求归档中的文件。 就像是 Node API, ASAR 存档可以被作为目录处理

```
<script>
let $ = require('./jquery.min.js')
$.get('file:///path/to/example.asar/file.txt', (data) => {
  console.log(data)
})
</script>
```



main.js | preload.js | index.html | renderer.js | npx @electron/fuses read --app Untitled-1

Users > helper > Desktop > tmp > 27 > 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 { ipcRenderer, contextBridge } = require('electron')

contextBridge.exposeInMainWorld('myApi', {
  fetch_url: () => {
    return ipcRenderer.invoke('fetch-url')
  }
})

```

main.js | preload.js | index.html | renderer.js | npx @electron/fuses read --app Untitled-1

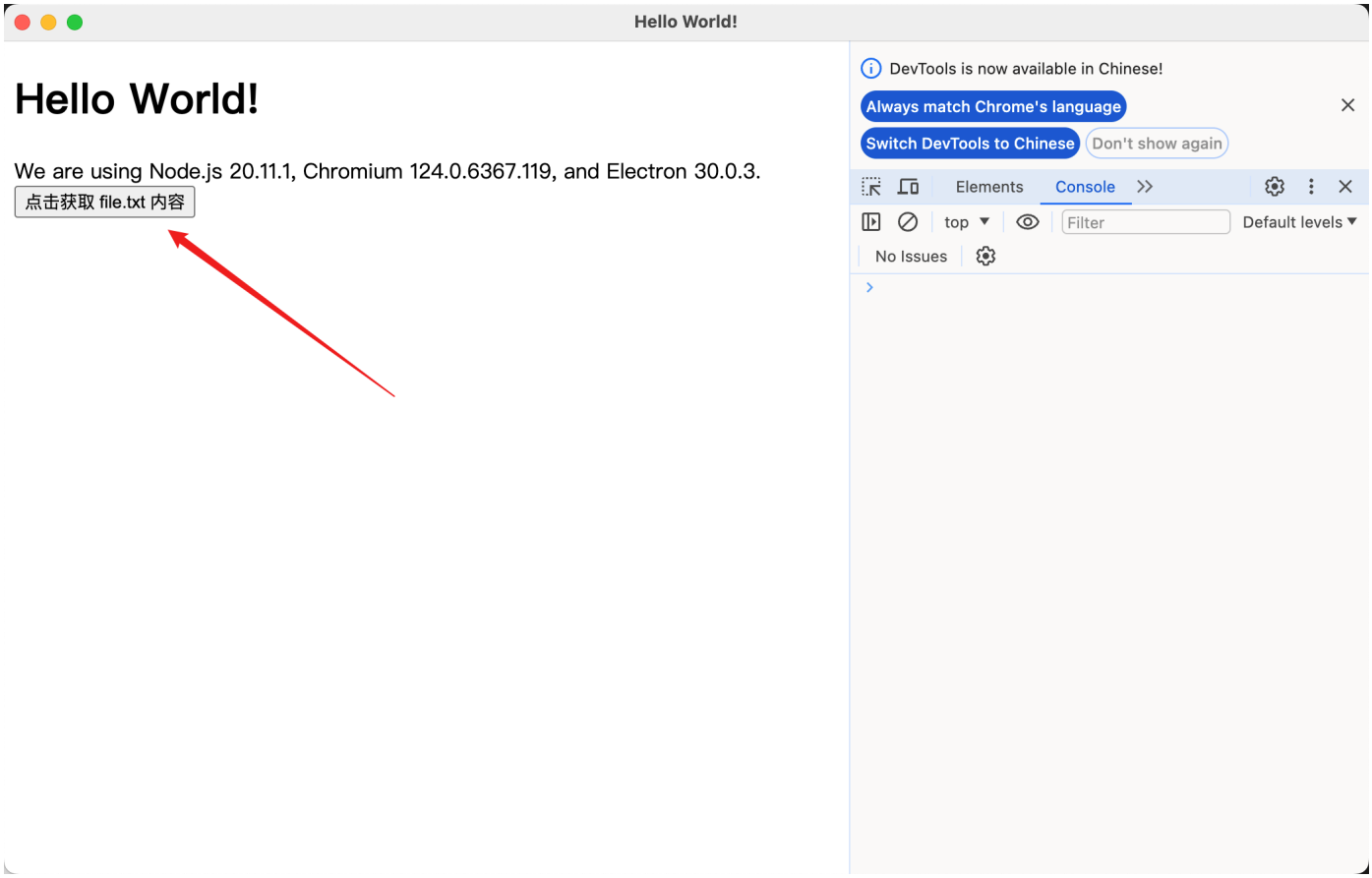
Users > helper > Desktop > tmp > 27 > index.html > html > body > button#fetch-url

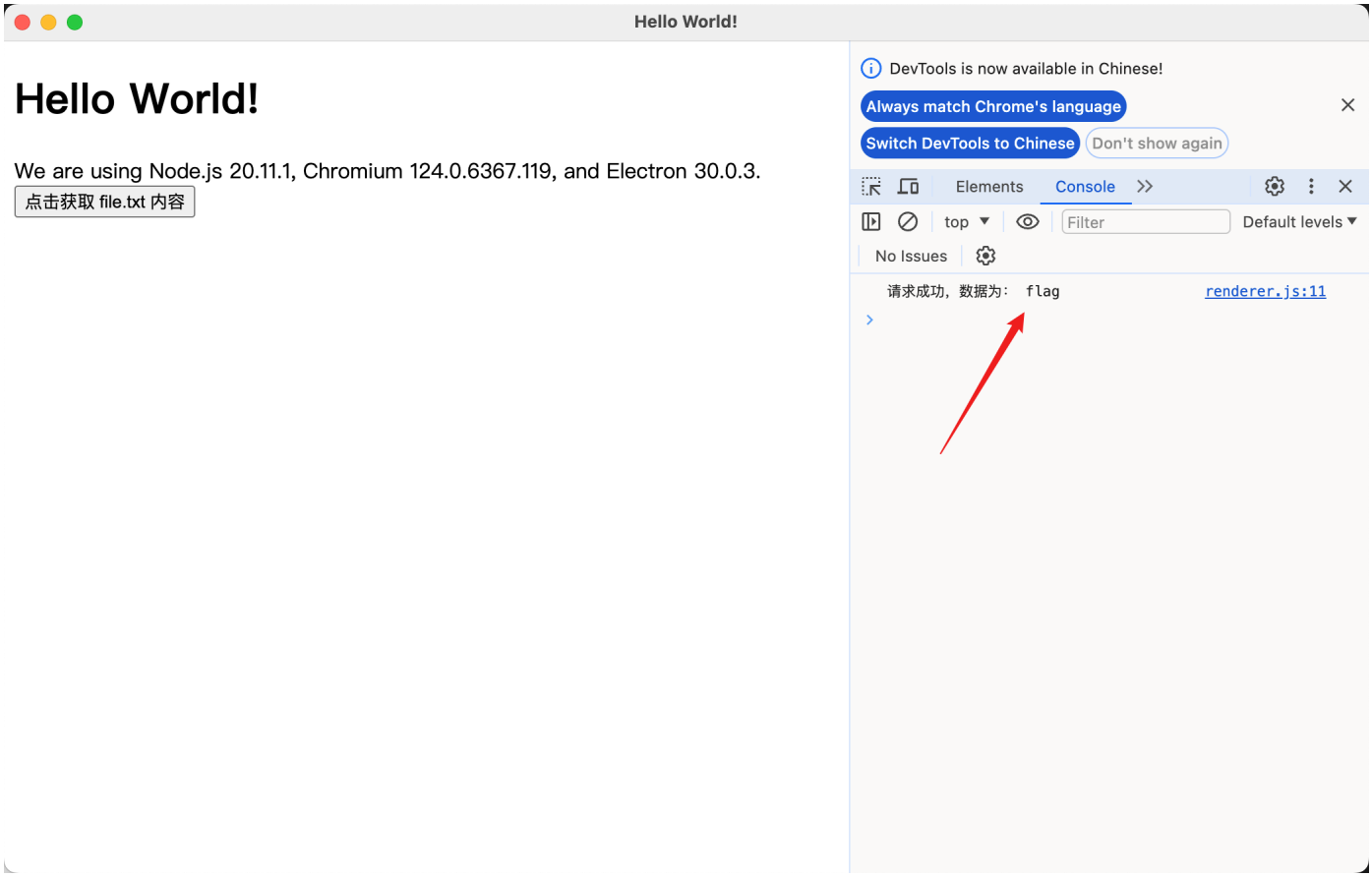
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <!-- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using Node.js <span id="node-version"></span>,
    Chromium <span id="chrome-version"></span>,
    and Electron <span id="electron-version"></span>.
    <br>
    <button id="fetch-url">点击获取 file.txt 内容</button>
    <!-- You can also require other files to run in this process -->
    <script src="./renderer.js"></script>
  </body>
</html>

```

main.js | preload.js | index.html | renderer.js | npx @electron/fuses read --app Untitled-1

Users > helper > Desktop > tmp > 27 > renderer.js > ...

```
const bt = document.getElementById('fetch-url')
bt.addEventListener('click', async () => {
    const url = await window.myApi.fetch_url()
    fetch(url).then(response => {
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        return response.text();
    })
    .then(data => {
        console.log('请求成功，数据为：', data);
    })
    .catch(error => {
        console.error('请求失败：', error);
    });
})

```





2) 将 asar 文件视为独立文件

某些情况下比如对 ASAR 归档文件进行校验，我们需要像读取 “文件” 那样读取 ASAR 文件。为此你可以使用内置的没有 `asar` 功能的和原始 `fs` 模块一模一样的 `original-fs` 模块。

```
const originalFs = require('original-fs')
originalFs.readFileSync('/path/to/example.asar')
```

您也可以将 `process.noAsar` 设置为 `true` 以禁用 `fs` 模块中对 `asar` 的支持：

```
const fs = require('node:fs')
process.noAsar = true
fs.readFileSync('/path/to/example.asar')
```

3. 执行 ASAR 档案中的二进制文件

有一些Node API可以执行二进制文件，例如 `child_process.exec` 、 `child_process.spawn` 和 `child_process.execFile` ，但只有 `execFile` 支持在ASAR档案内执行二进制文件。

因为 `exec` 和 `spawn` 允许 `command` 替代 `file` 作为输入，而 `command` 是需要在 shell 下执行的

目前没有 可靠的方法来判断 command 中是否在操作一个 asar 包中的文件，而且即便可以判断，官方依旧无法保证可以在无任何副作用的情况下替换 command 中的文件路径。

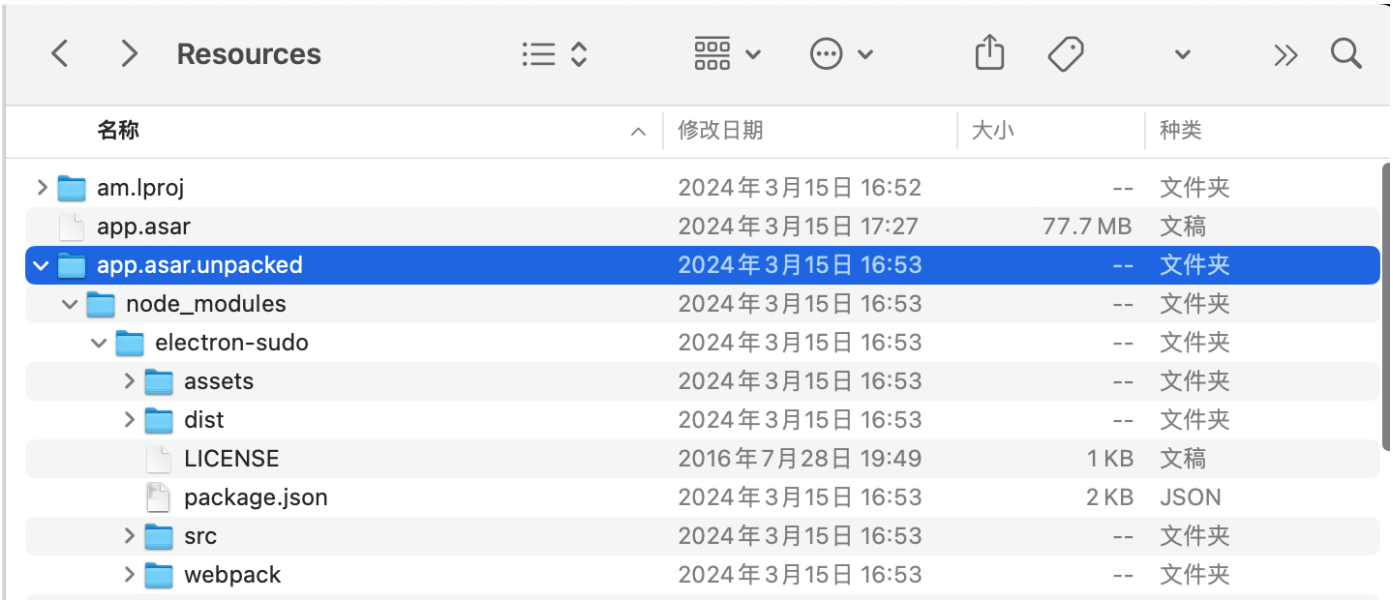
4. 向 ASAR 档案添加未打包的文件

某些 Node API 被调用时会解压文件到文件系统。 除了性能问题外，可能会触犯各种防病毒扫描程序。

你可以把使用 `--unpack` 选项作为将各种文件保持为非压缩状态的一种解决方法。 在下面的示例中，原生Node.js模块的共享库将不会被打包：

```
$ asar pack app app.asar --unpack *.node
```

运行命令后，您将会看到 `app.asar.unpacked` 文件夹与 `app.asar` 文件一起被创建了。 没有被打包的文件和 `app.asar` 会一起存档发布



> 参考文章
>
> https://www.electronjs.org/zh/docs/latest/tutorial/asar-archives

0x03 ASAR 完整性

> ASAR完整性是一项实验性功能，可在运行时验证应用的ASAR归档的内容

目前支持 ASAR 完整性的版本如下

- MacOS 上 `Electron >= 16.0.0`
- Windows 上 `Electron >= 30.0.0`

目前仅支持由 `@electron/asar` 生成的 ASAR 文件的完整性检查

在 `asar@3.1.0` 中引入了支持。请注意，这个包已经迁移到 `@electron/asar` 所有版本的 `@electron/asar` 都支持ASAR完整性

1. 工作原理

每个 ASAR 文件都包含一个 JSON 字符串头部，头部的格式包含一个 `integrity` 对象，这个对象包含一个 16 进制编码的 hash 值，这个 hash 值是整个 ASAR 文件的 hash 值；还包含了一个数组，数组中存储的是每个块的 16 进制编码的 hash 值

```
{
  "algorithm": "SHA256",
  "hash": "...",
  "blockSize": 1024,
  "blocks": ["...", "..."]
}
```

另外，在打包Electron应用程序时，您需要定义整个ASAR头的十六进制编码哈希

启用ASAR完整性后，您的Electron应用程序将在运行时验证ASAR存档的头部哈希。如果没有哈希值，或者哈希值不匹配，应用程序将强制终止

2. 在二进制程序中开启 ASAR 完整性

ASAR完整性检查目前在Electron中默认禁用，可以在构建时通过切换 `EnableEmbeddedAsarIntegrityValidation` 这个 fuse 启用，通常还需要启用 `onlyLoadAppFromAsar` ，否则，可以通过Electron应用程序代码搜索路径绕过有效性检查。

```
const { flipFuses, FuseVersion, FuseV1Options } =
require('@electron/fuses')

flipFuses(
  // E.g. /a/b/Foo.app
  pathToPackagedApp,
  {
    version: FuseVersion.V1,
    [FuseV1Options.EnableEmbeddedAsarIntegrityValidation]: true,
    [FuseV1Options.OnlyLoadAppFromAsar]: true
  }
)
```

当然可以通过 `Electron Forge` 打包时进行配置

3. 采集ASAR头部的hash

ASAR 完整性根据您在打包时提供的头部 hash 来验证 ASAR 存档的内容。提供此打包哈希的过程对于 macOS 和Windows 是不同的

1) 使用 Electron Forge 和 Electron Packager

Electron Forge和Electron Packager自动为您进行此设置，无需额外配置。ASAR完整性所需的最低版本为：

- `@electron/packager@18.3.1`
- `@electron/forge@7.4.0`

2) 使用其他构建系统

MacOS

为macOS打包时，您必须在打包的应用的 `Info.plist` 中填充有效的 `ElectronAsarIntegrity` 字典块。下面是一个例子

```
<key>ElectronAsarIntegrity</key>
<dict>
  <key>Resources/app.asar</key>
  <dict>
    <key>algorithm</key>
    <string>SHA256</string>
    <key>hash</key>

 <string>9d1f61ea03c4bb62b4416387a521101b81151da0cfbe18c9f8c8b818c5cebfac</
string>
  </dict>
</dict>
```

有效的 `algorithm` 值当前为 `SHA256` 。 `hash` 是使用刚刚指定的 `algorithm` 计算ASAR头部得到的哈希

`@electron/asar` 包公开了一个 `getRawHeader` 方法，然后可以对该方法的结果进行散列以生成此值（例如使用 `node:crypto` 模块）。

Windows

打包Windows平台程序时，必须填充类型为 `Integrity` 、名称为 `ElectronAsar` 的有效资源条目。此资源的值应该是 JSON 编码的字典，格式如下：

```
[
  {
    "file": "resources\\app.asar",
    "alg": "sha256",
    "value":
"9d1f61ea03c4bb62b4416387a521101b81151da0cfbe18c9f8c8b818c5cebfac"
  }
]
```

> 有关实现示例，请参见 `Electron Packager` 代码中的 `src/resedit.ts`
>
> https://github.com/electron/packager/blob/main/src/resedit.ts

4. 关于原理的一些疑问?

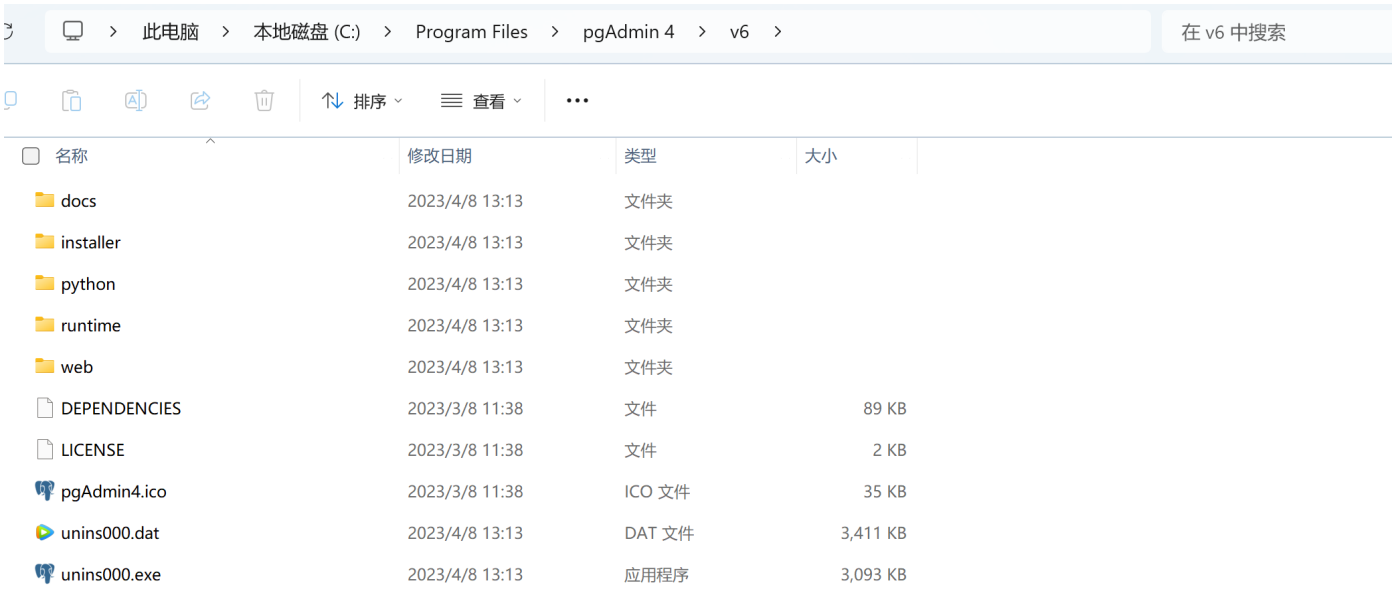
在这里算是把基本原理说清楚了，其实是 `.asar` 文件的整体和分块 hash 被存储在 `.asar` 的头部，当然计算整体 hash 时肯定没有包括头部，不然就出现了逻辑悖论了

之后通过对 `.asar` 的头部字符串进行计算hash 获取一个值，之后将这个值在打包过程中嵌入到二进制可执行文件中

这样如果开启代码完整性检查，则会在运行的时候把这个值拿出来，同 `.asar` 的头部的hash进行比对，如果通过则运行，不通过则退出

这里就有一些问题说得比较模糊了，程序运行的时候是只校验 `.asar` 文件的头部的 hash 吗？ 还是头部的hash校验通过后，会进一步校验 `.asar` 整体和分块的 `hash` 与 `.asar` 头部的记录内容是否一致?

程序打包后可能会生成一个完整的 `.exe` 这类文件，也有一些 `.exe` 在安装后会释放一些文件



现在问题是，那么 ASAR 完整性校验代码是在最初的安装文件里才有，还是在安装文件里和释放后的启动文件(二进制可执行文件)里都有呢?

在查找资料的过程中，发现了开发者和用户曾经在 2019 年进行的一场讨论，就是说如果 `asar` 代码被修改了，添加了恶意代码，如何在 `Electron` 中发现，此时还没有代码完整性检查的 fuse 以及官方技术，官方人员的态度是： **如果攻击者已经可以修改 `.asar` 文件了，说明攻击者已经获取了系统权限，此时应该担心的不是 asar 有没有被修改，而是攻击者已经获取了系统权限**

其中部分开发者及安全人员表达自己的观点，如果官方不提供代码完整性检查，那么攻击者就可以利用有签名的程序加载恶意代码，也就是白加黑，具体讨论内容参考下方链接

> https://github.com/electron/electron/issues/19671

现在有了代码完整性检查，将 ASAR 头部计算得到的 hash 值写入了二进制文件，但是如果攻击者能够同时修改 `.asar` 文件和二进制文件，在 `.asar` 文件中添加恶意代码，生成新的 hash，修改二进制文件中的hash为新 hash ，那么完整性检查还是会被绕过

但此时，二进制文件的签名就会失效，系统的完整性检查会辅助 asar 的完整性检查，所以程序签名几乎是最后一道防线

0x04 测试猜想

目前使用了 ASAR 完整性的程序非常少，毕竟刚出来，就连 `Discord` 、 `VSCode` 这种更新比较快的应用都没有启用

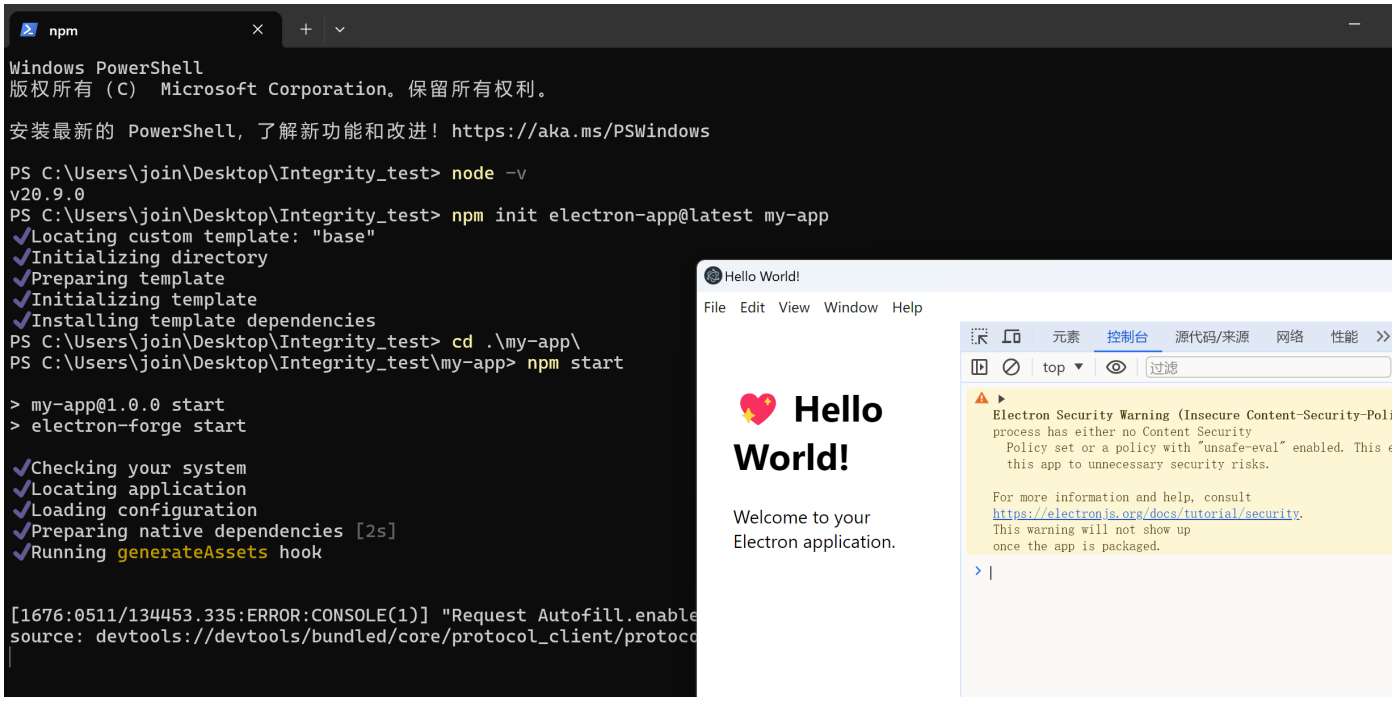
```
PS C:\Program Files\pgAdmin 4\v6> npx @electron/fuses read --app "C:\Users\join\AppData\Local\Programs\Microsoft VS Code
\Code.exe"
Analyzing app: Code.exe
Fuse Version: v1
  RunAsNode is Enabled
  EnableCookieEncryption is Disabled
  EnableNodeOptionsEnvironmentVariable is Enabled
  EnableNodeCliInspectArguments is Enabled
  EnableEmbeddedAsarIntegrityValidation is Disabled
  OnlyLoadAppFromAsar is Disabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
PS C:\Program Files\pgAdmin 4\v6>
```

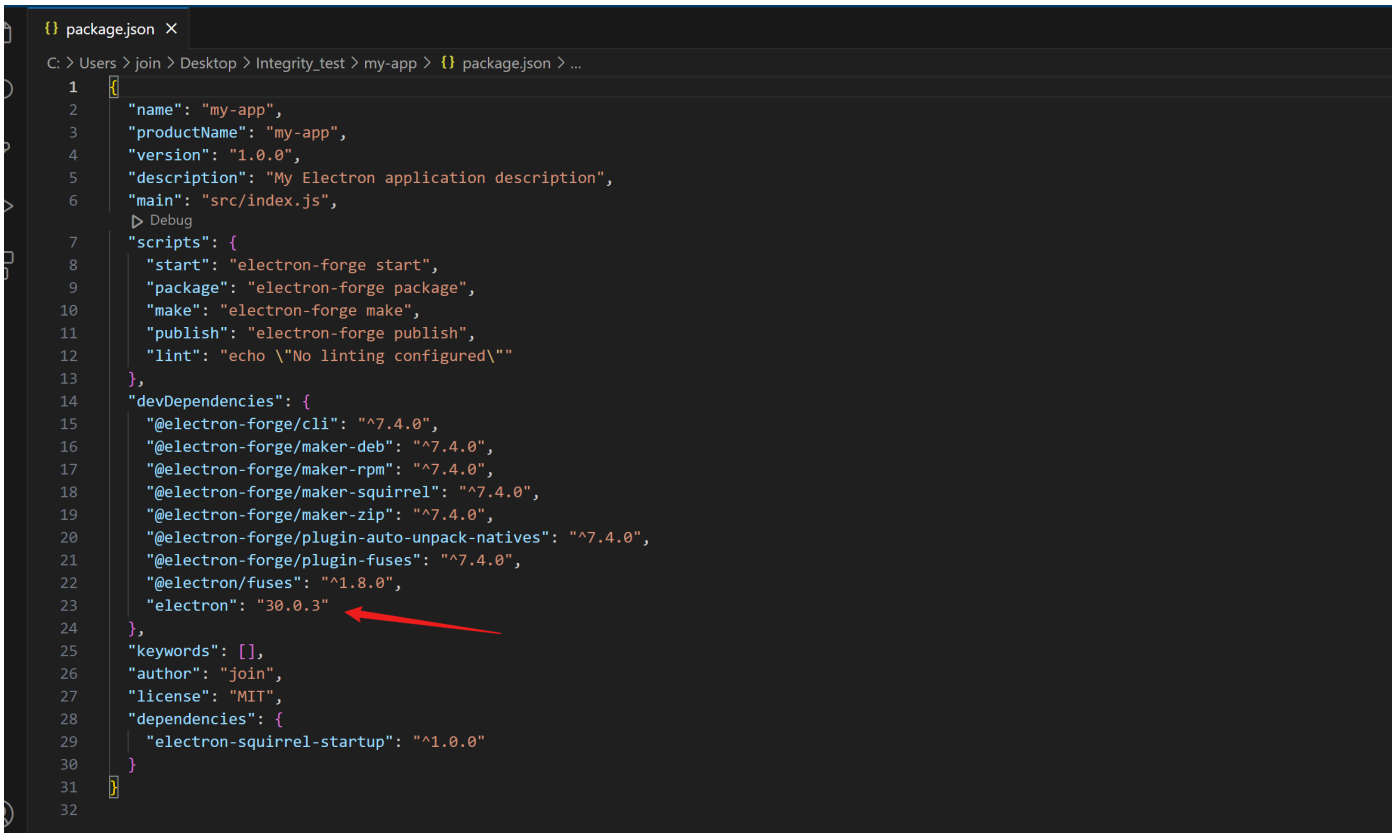
所有我们需要自行开发并打包一款程序进行测试，平台就选择 Windows 11 好了

1. 创建应用程序

直接选用官方程序

```
npm init electron-app@latest my-app
```





`Electron` 版本为 `30.0.3` ，具备代码完整性检查的能力

可以看到，默认情况下会自动打开开发者工具，我们一会要实现的效果就是通过注入恶意代码，取消自动打开开发者工具，好像也没有那么恶意哈

2. 安装 Forge

进入到程序目录，即 `my-app` ，执行安装命令

```
npm install --save-dev @electron-forge/cli
```

```
PS C:\Users\join\Desktop\Integrity_test\my-app> npm install --save-dev @electron-forge/cli
(node:3800) MaxListenersExceededWarning: Possible EventEmitter memory leak detected. 11 close listeners added to [TLSSocket]. Use emitter.setMaxListeners() to increase limit
(Use `node --trace-warnings ...` to show where the warning was created)
npm WARN deprecated gar@1.0.4: Package no longer supported. Contact Support at https://www.npmjs.com/support for more info.

up to date, audited 457 packages in 4s

74 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\join\Desktop\Integrity_test\my-app>
```

3. 设置开启代码完整性检查

只需要修改 `my-app` 目录下的 `forge.config.js` 文件即可

package.json　forge.config.js

C: > Users > join > Desktop > Integrity_test > my-app > forge.config.js > ...

```
const { FusesPlugin } = require('@electron-forge/plugin-fuses');
const { FuseV1Options, FuseVersion } = require('@electron/fuses');

module.exports = {
  packagerConfig: {
    asar: true,
  },
  rebuildConfig: {},
  makers: [
    {
      name: '@electron-forge/maker-squirrel',
      config: {},
    },
    {
      name: '@electron-forge/maker-zip',
      platforms: ['darwin'],
    },
    {
      name: '@electron-forge/maker-deb',
      config: {},
    },
    {
      name: '@electron-forge/maker-rpm',
      config: {},
    },
  ],
  plugins: [
    {
      name: '@electron-forge/plugin-auto-unpack-natives',
      config: {},
    },
    // Fuses are used to enable/disable various Electron functionality
    // at package time, before code signing the application
    new FusesPlugin({
      version: FuseVersion.V1,
      [FuseV1Options.RunAsNode]: false,
      [FuseV1Options.EnableCookieEncryption]: true,
      [FuseV1Options.EnableNodeOptionsEnvironmentVariable]: false,
      [FuseV1Options.EnableNodeCliInspectArguments]: false,
      [FuseV1Options.EnableEmbeddedAsarIntegrityValidation]: true,
      [FuseV1Options.OnlyLoadAppFromAsar]: true,
    }),
  ],
};

```

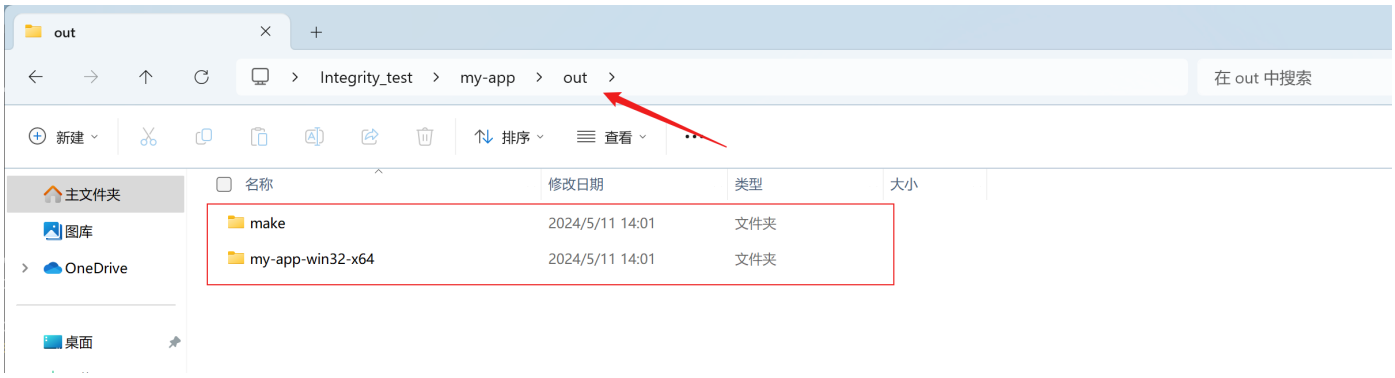
我们发现其实已经默认就设置为 `true` 了

4. 打包程序

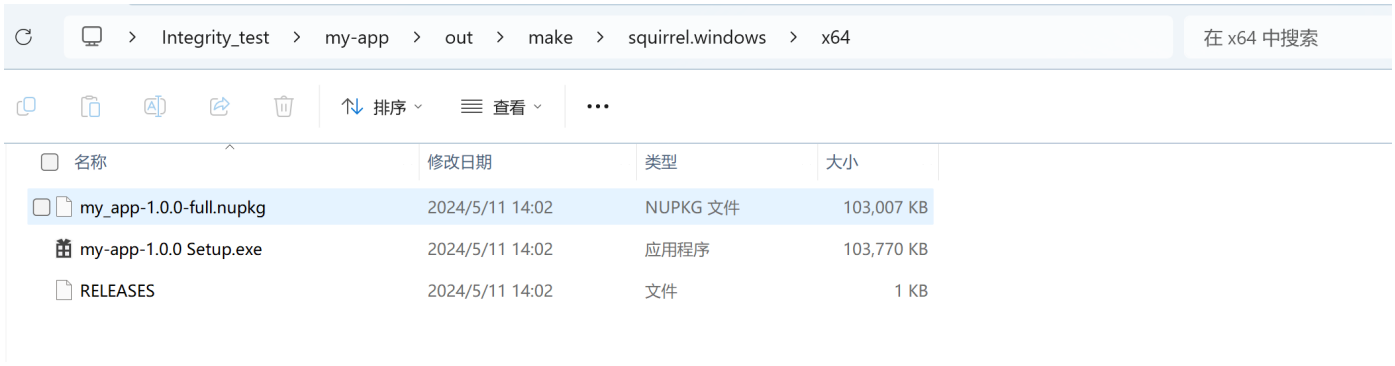
```
npm run make
```

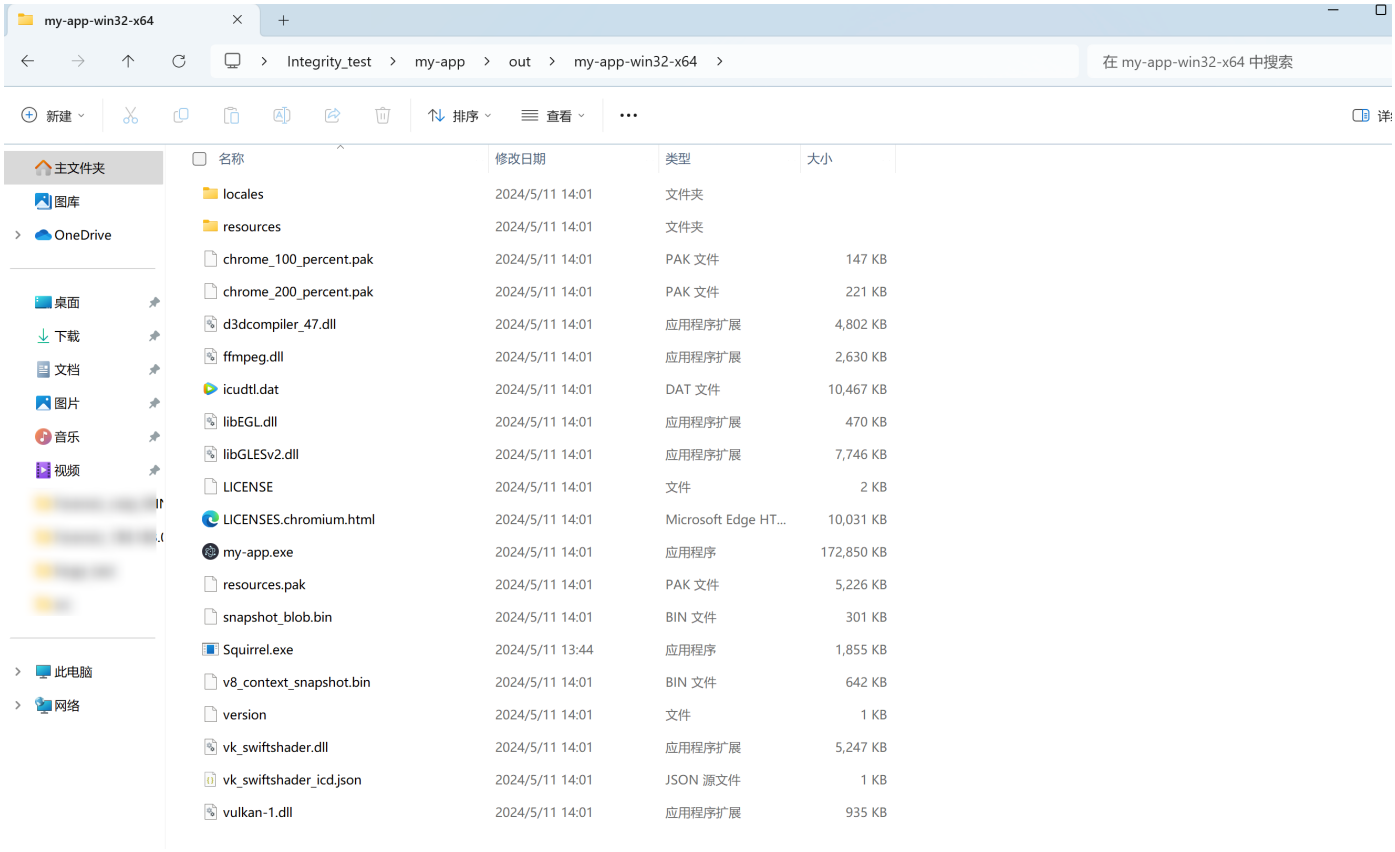
```
PS C:\Users\join\Desktop\Integrity_test\my-app> npm run make

> my-app@1.0.0 make
> electron-forge make

✔Checking your system
✔Loading configuration
✔Resolving make targets
  › Making for the following targets:
✔Running package command
  ✔Preparing to package application
  ✔Running packaging hooks
    ✔Running generateAssets hook
    ✔Running prePackage hook
  ✔Packaging application
    ✔Packaging for x64 on win32 [6s]
  ✔Running postPackage hook
✔Running preMake hook
✔Making distributables
  ✔Making a squirrel distributable for win32/x64 [39s]
✔Running postMake hook
  › Artifacts available at: C:\Users\join\Desktop\Integrity_test\my-app\out\make
PS C:\Users\join\Desktop\Integrity_test\my-app>
```

在 `my-app` 目录下新创建了一个 `out` 目录，官方提示我们 `Artifacts` 在 `out` 目录下的 `make` 目录中

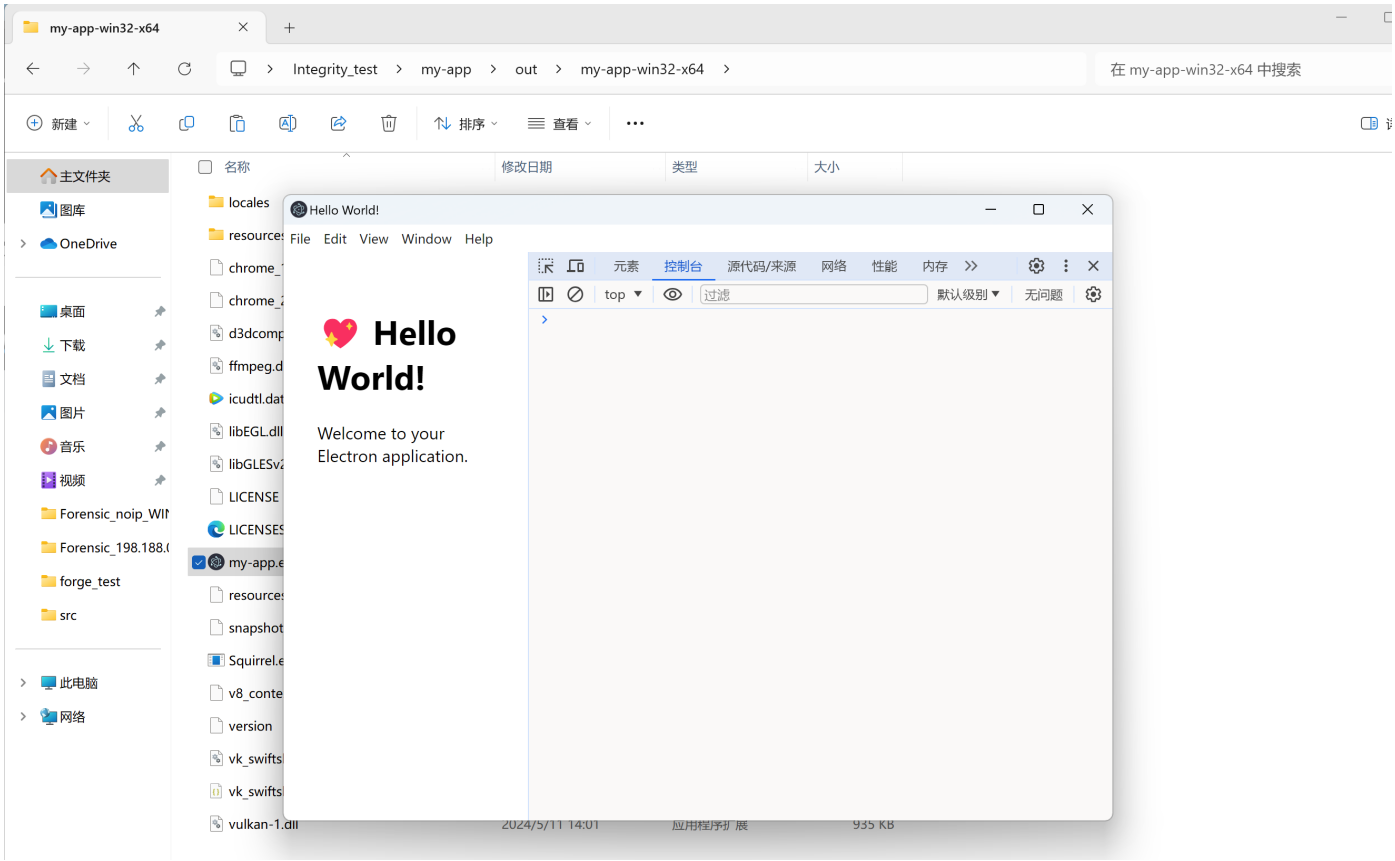


在 `out` 文件夹下有两个文件夹，其中 `make` 文件夹内存放的是编译后的单文件; 另一个以程序名字-操作系统-架构命名的文件夹存放的是包含多文件的目录，其中就包括入口文件





经过测试，默认的单文件并不会涉及到安装过程，也不会解压释放文件目录，所以我们以多文件目录的程序为例



程序正常打开，会自动打开开发者工具

fuse 如下

```
PS C:\Users\join\Desktop\Integrity_test\my-app> npx @electron/fuses read --app ".\out\my-app-win32-x64\my-app.exe"
Analyzing app: my-app.exe
Fuse Version: v1
  RunAsNode is Disabled
  EnableCookieEncryption is Enabled
  EnableNodeOptionsEnvironmentVariable is Disabled
  EnableNodeCliInspectArguments is Disabled
  EnableEmbeddedAsarIntegrityValidation is Enabled
  OnlyLoadAppFromAsar is Enabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
  GrantFileProtocolExtraPrivileges is Enabled
PS C:\Users\join\Desktop\Integrity_test\my-app>
```

5. 查看 ASAR 头部

在 `asar` 的 Github 项目中，有详细介绍 ASAR 头部格式

```
| UInt32: header_size | String: header | Bytes: file1 | ... | Bytes: file42
|
```

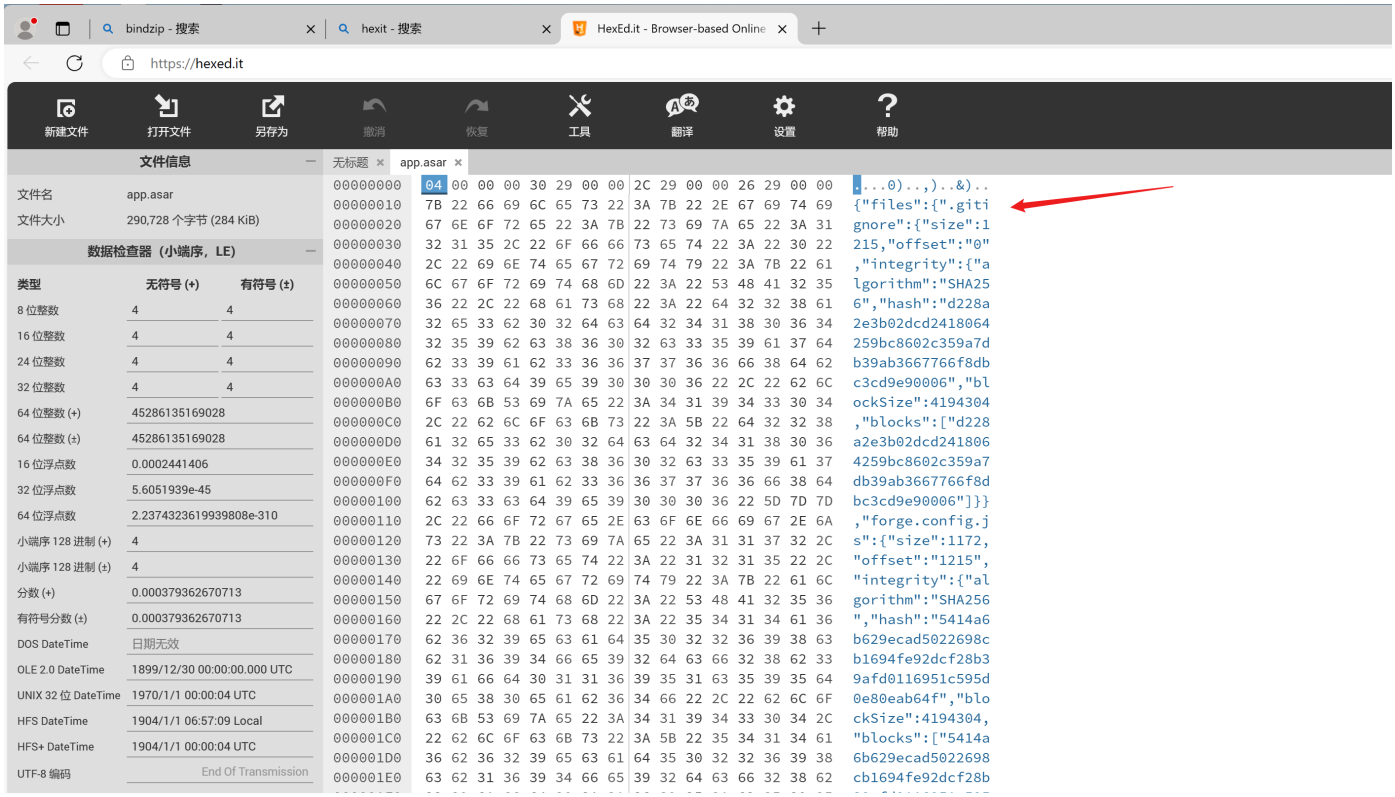
```
{
    "files": {
        "tmp": {
            "files": {}
        },
```

```
    "usr" : {
       "files": {
         "bin": {
           "files": {
             "ls": {
               "offset": "0",
               "size": 100,
               "executable": true,
               "integrity": {
                 "algorithm": "SHA256",
                 "hash": "...",
                 "blockSize": 1024,
                 "blocks": ["...", "..."]
               }
             },
             "cd": {
               "offset": "100",
               "size": 100,
               "executable": true,
               "integrity": {
                 "algorithm": "SHA256",
                 "hash": "...",
                 "blockSize": 1024,
                 "blocks": ["...", "..."]
               }
             }
           }
         }
       }
    },
    "etc": {
       "files": {
         "hosts": {
           "offset": "200",
           "size": 32,
           "integrity": {
              "algorithm": "SHA256",
              "hash": "...",
              "blockSize": 1024,
```

```
                "blocks": ["...", "..."]
            }
        }
    }
}
```

从头部结构来看，也是通过偏移确定文件位置

我们可以直接将 `resources` 里面的 `app.asar` 拖进 16 进制查看器，我这里选择在线的



我们发现除了文件幻数以外，剩下都是明文的，那就好办了

> 参考文章
>
> https://github.com/electron/asar

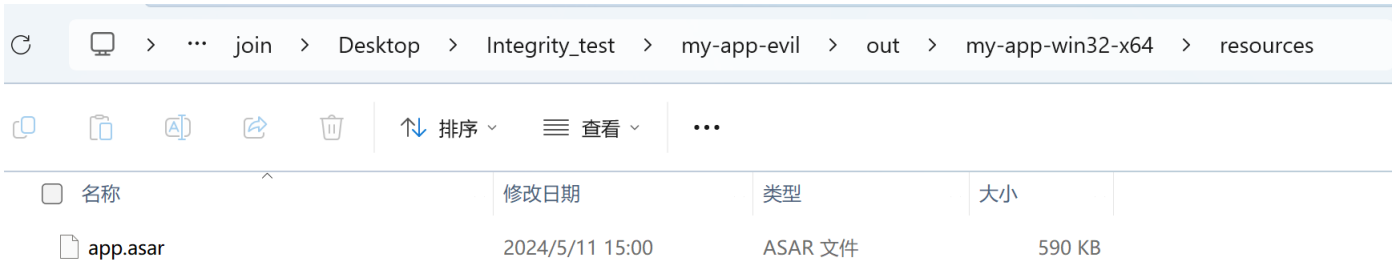
6. 制作恶意 ASAR

默认情况下， `ASAR` 内部是只读的，所以无法直接通过 `Electron` 的 API 去修改，

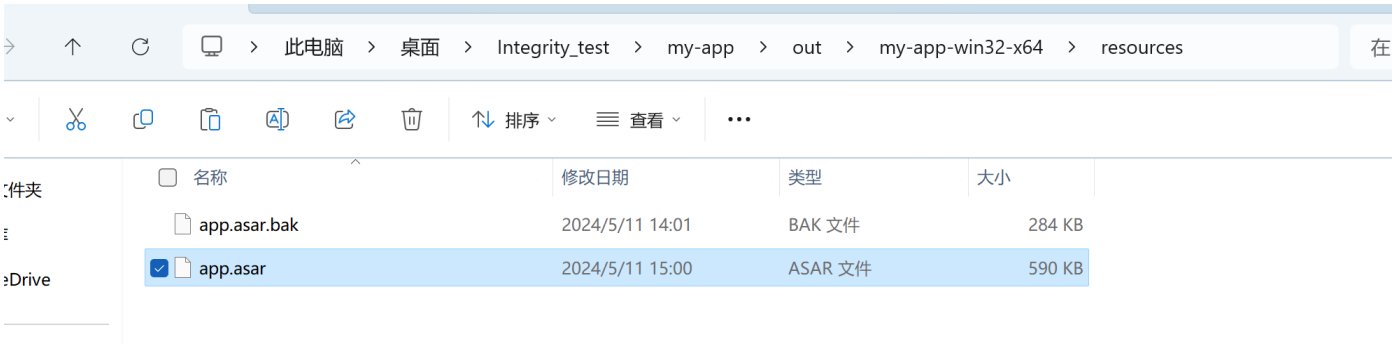
我们可以编译两份 APP，文件名称保持一致，其中一份内容里关闭掉开发者工具，也就是模拟恶意行为，之后同样使用 `Electron Forge` 打包，之后使用原本正常的头部替换掉不正常的头部，再将组合成的恶意asar文件替换到正常的文件，我们看一下，此时程序是否正常，是否能够发现篡改行为

1) 生成恶意 ASAR 文件

```
PS C:\Users\join\Desktop\Integrity_test\my-app-evil> npm run make

> my-app@1.0.0 make
> electron-forge make

✔Checking your system
✔Loading configuration
✔Resolving make targets
  › Making for the following targets:
✔Running package command
  ✔Preparing to package application
  ✔Running packaging hooks
    ✔Running generateAssets hook
    ✔Running prePackage hook
  ✔Packaging application
    ✔Packaging for x64 on win32 [7s]
  ✔Running postPackage hook
✔Running preMake hook
✔Making distributables
  ✔Making a squirrel distributable for win32/x64 [39s]
✔Running postMake hook
  › Artifacts available at: C:\Users\join\Desktop\Integrity_test\my-app-evil\out\make
PS C:\Users\join\Desktop\Integrity_test\my-app-evil>
```



我们先直接覆盖一下正常文件试试，看看能不能直接就成功，不成功显示什么？ 当然要备份好原文件

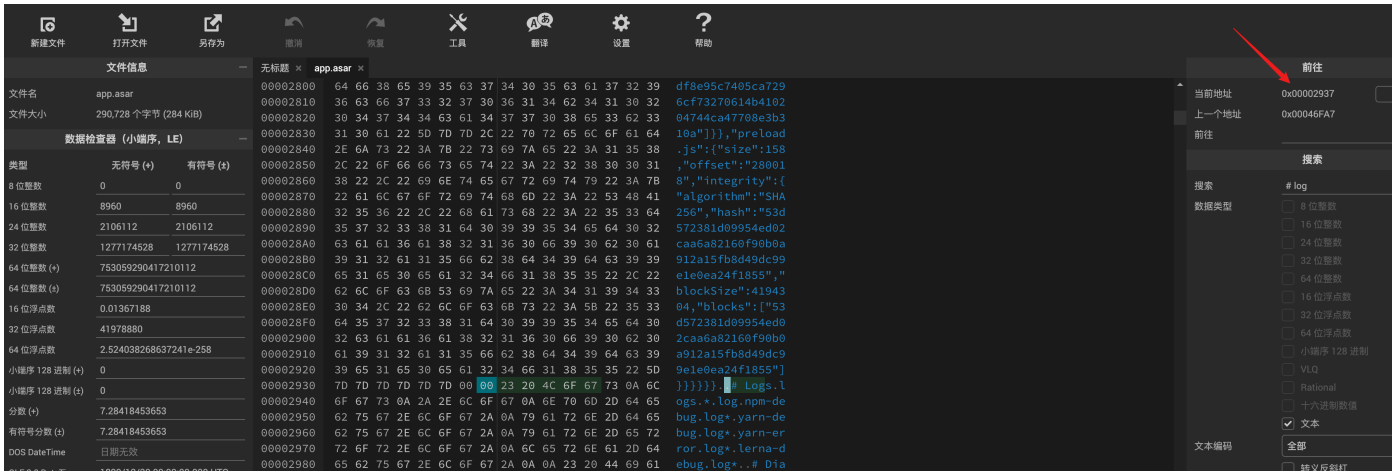



```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> .\my-app.exe
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
[4332:0511/150425.386:FATAL:asar_util.cc(163)] Integrity check failed for asar archive (e88c71ebbaac19ccea7667b5249e84890c89ecb0104b4bf5a0
594f6305020f48b vs e9a168d2ad3d4657d65296abbd8d451016f8fcdb4a6b180026656b496b8aa4b3)
[4332:0511/150425.386:ERROR:crashpad_client_win.cc(868)] not connected
```

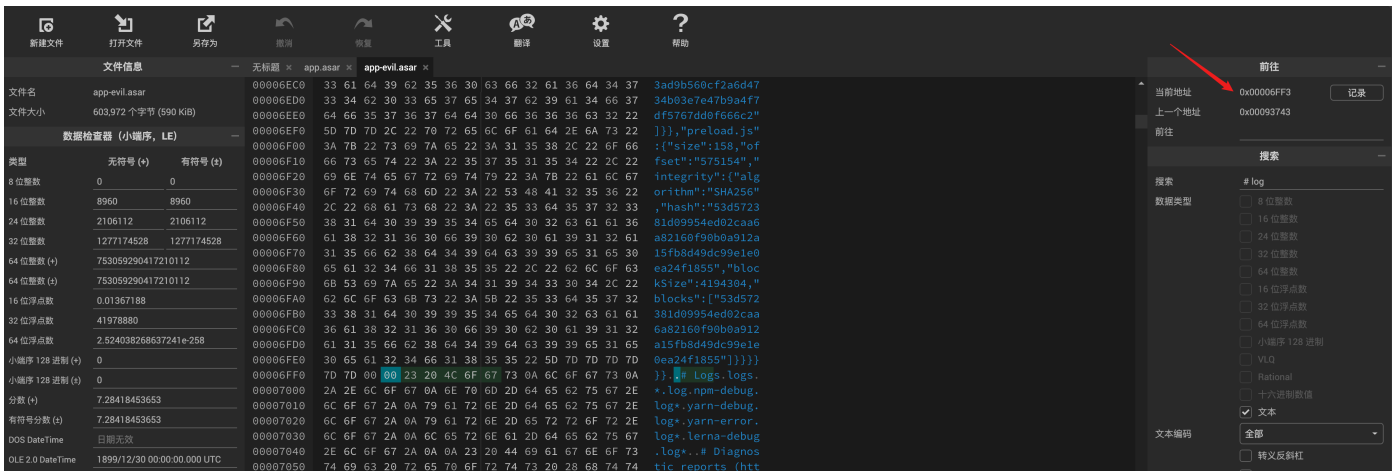
看起来直接替换是不行的

2) 修改恶意 asar 头部后覆盖测试

正常 `.asar` 文件校验头位置为 `0x00002937` ，10 进制就是 `10551`



恶意 `.asar` 文件校验头位置为 `0x00006FF3` ，10 进制就是 `28659`



所以我们粗暴的替换的话，直接去掉恶意文件的头，之后将正常文件的头放到恶意文件上去

做这种事，还是得把战线拉到自己擅长的领域，命令行启动

去掉恶意 `.asar` 文件的头，保留剩下部分

```
dd if=./app-evil.asar of=./app-evil-without-header.asar bs=1 skip=28659
```

```
~/D/t/w/1 ❯❯❯ dd if=./app-evil.asar of=./app-evil-without-header.asar bs=1 skip=28659
575313+0 records in
575313+0 records out
575313 bytes transferred in 1.736637 secs (331280 bytes/sec)
~/D/t/w/1 ❯❯❯
```

获取正常 `.asar` 文件的头

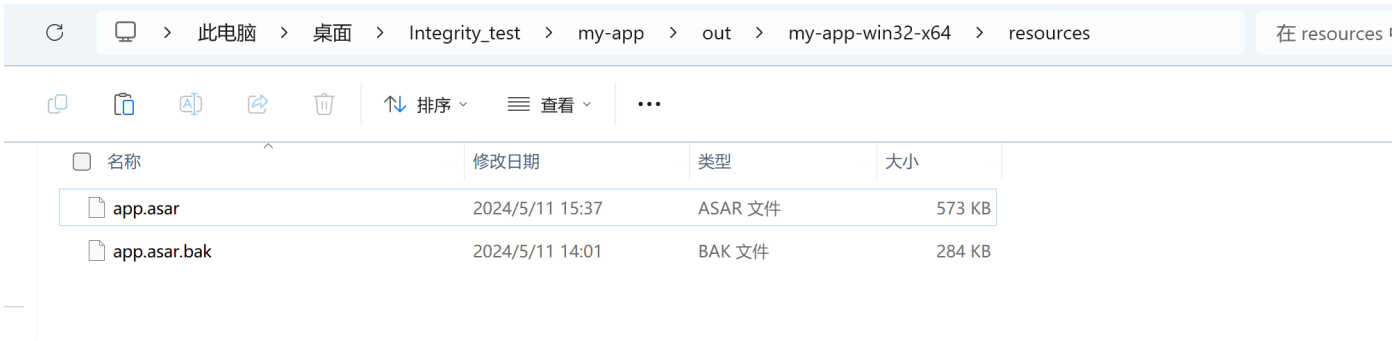
```
dd if=./app.asar of=./app-header.asar bs=1 count=10551
```

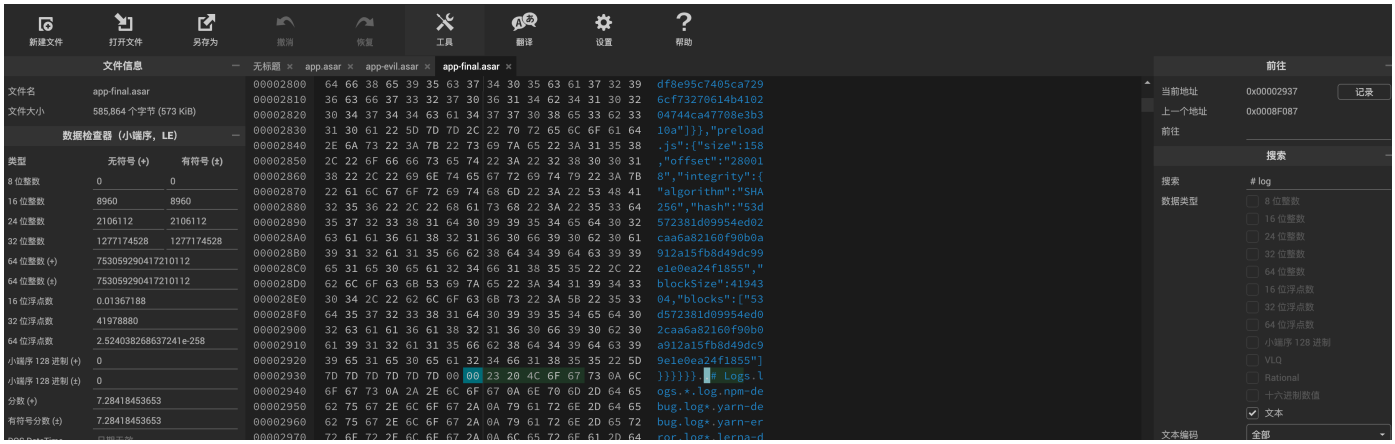
```
~/D/t/w/1 ❯❯❯ dd if=./app.asar of=./app-header.asar bs=1 count=10551
10551+0 records in
10551+0 records out
10551 bytes transferred in 0.037195 secs (283667 bytes/sec)
~/D/t/w/1 ❯❯❯
```

开始拼接两个文件

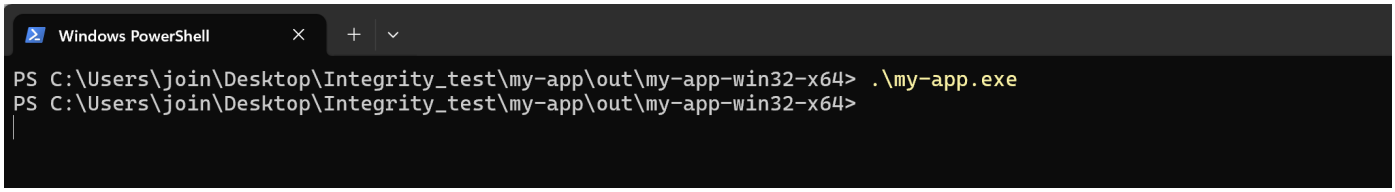
```
~/D/t/w/1 ❯❯❯ cat ./app-header.asar ./app-evil-without-header.asar > app-final.asar
~/D/t/w/1 ❯❯❯ file app-final.asar
app-final.asar: data
~/D/t/w/1 ❯❯❯
```

7. 替换原本的 app.asar



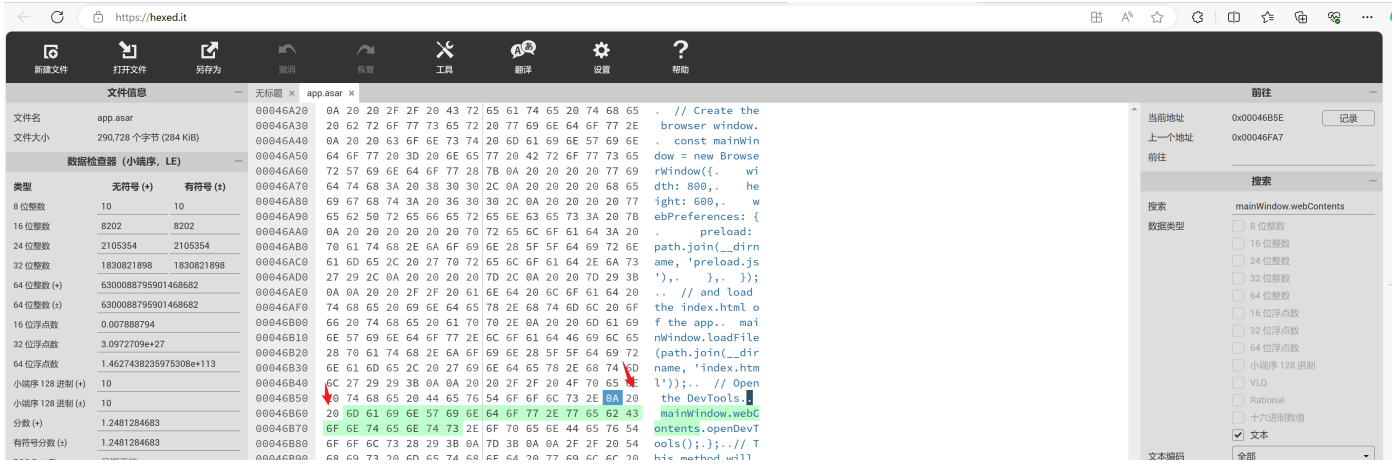


启用 `my-app.exe`

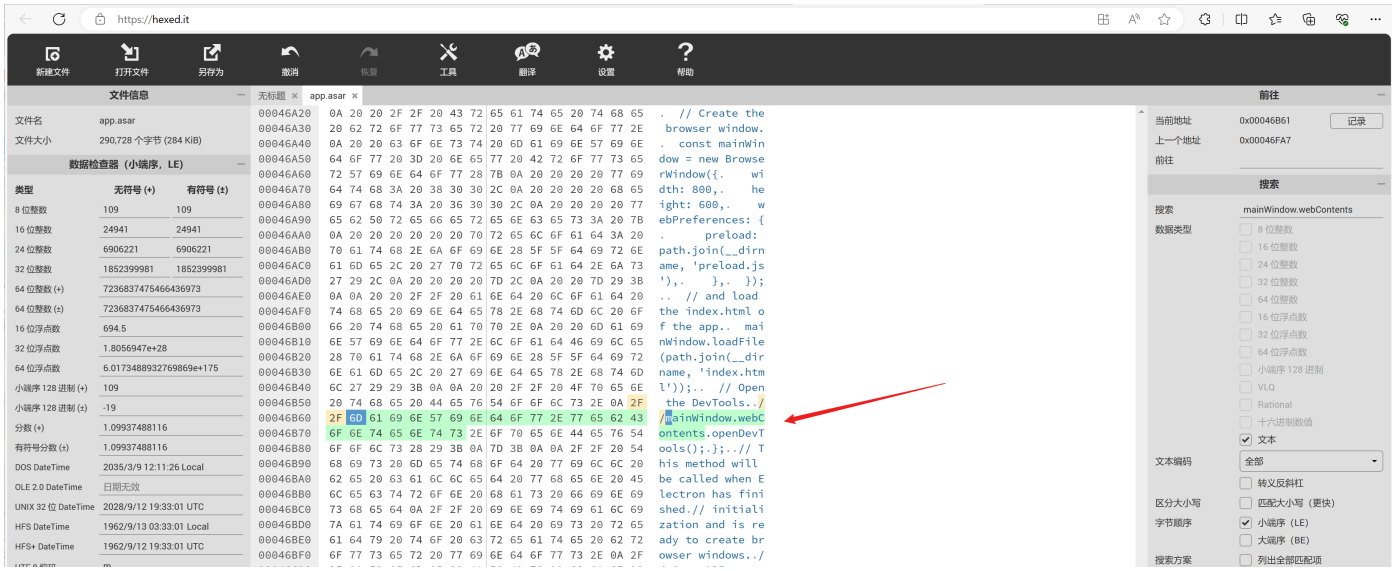


可以看到，完整性校验已经绕过去了，但是程序没有起来

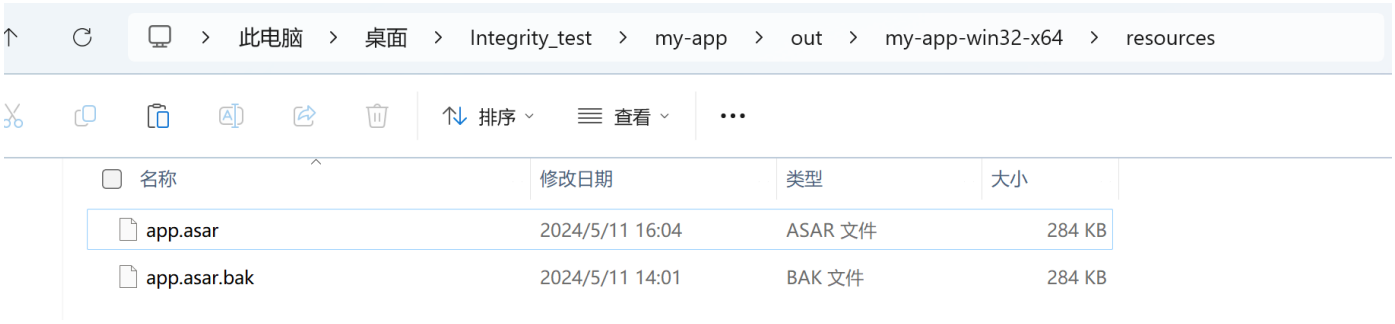
这也不难理解，因为在我们替换的头里，有偏移相关的信息，接下来我们得开始 B 方案了，我直接修改正常的 `app.asar` ，将里面的空格改为注释，这样没有改变文件大小，也没有改变文件位置，如果还启动不起来，那就是 `Electron` 还会校验文件头里的内容



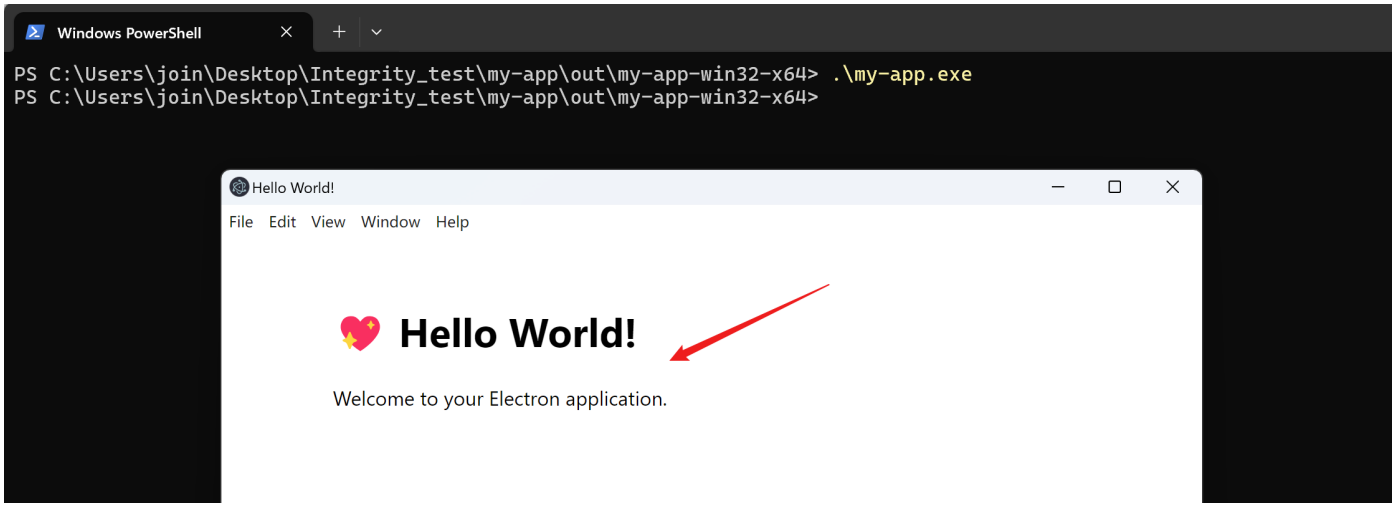
将这两个空格修改为 `//`



将修改后的 `app.asar` 覆盖原本的文件



再次启动 `my-app.exe`



可以看到，修改成功了，没有开发者工具弹出，这说明完整性检查并不可靠，攻击者完全可以通过修改 `app.asar` 进行代码注入

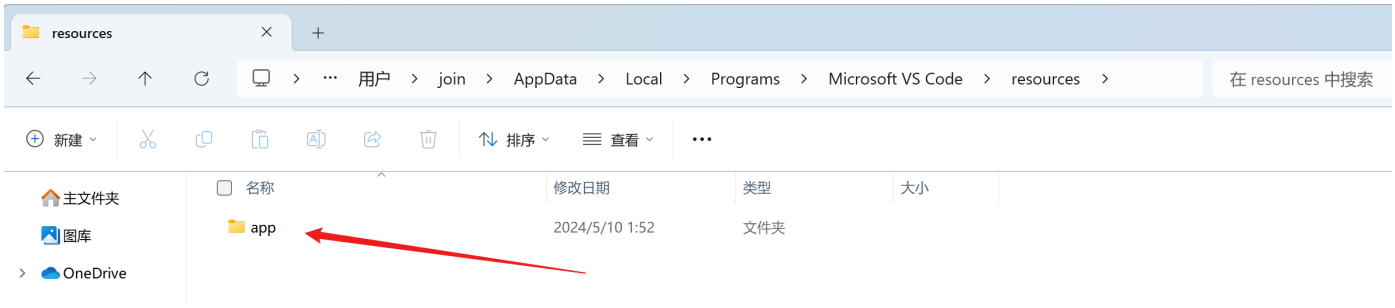
0x05 onlyLoadAppFromAsar

关于 `onlyLoadAppFromAsar` 这个说一下功能，根据官方描述，默认情况下， `Electron` 开发的程序检索 `asar` 文件的顺序是

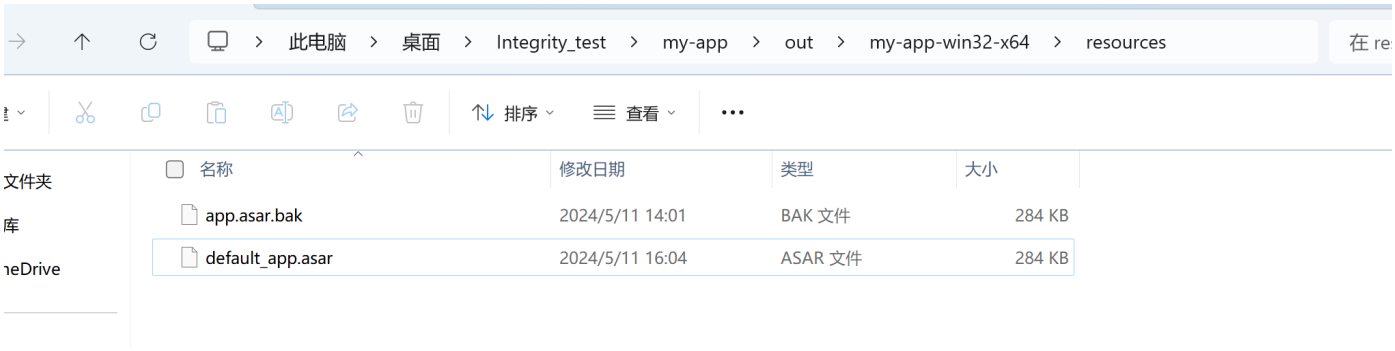
- app.asar
- app
- default_app.asar

当开启 `onlyLoadAppFromAsar` 时，就只使用 `app.asar`

上面提到的 `app` 应该是指目录，微软的 `vscode` 就是使用的 `app` 目录



我们将 `my-app` 程序的 `app.asar` 修改为 `default_app.asar`




```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses read --app "my-app.exe"
Analyzing app: my-app.exe
Fuse Version: v1
  RunAsNode is Disabled
  EnableCookieEncryption is Enabled
  EnableNodeOptionsEnvironmentVariable is Disabled
  EnableNodeCliInspectArguments is Disabled
  EnableEmbeddedAsarIntegrityValidation is Enabled
  OnlyLoadAppFromAsar is Enabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
  GrantFileProtocolExtraPrivileges is Enabled
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
```

此时我们尝试打开该 `my-app`

```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses read --app "my-app.exe"
Analyzing app: my-app.exe
Fuse Version: v1
  RunAsNode is Disabled
  EnableCookieEncryption is Enabled
  EnableNodeOptionsEnvironmentVariable is Disabled
  EnableNodeCliInspectArguments is Disabled
  EnableEmbeddedAsarIntegrityValidation is Enabled
  OnlyLoadAppFromAsar is Enabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
  GrantFileProtocolExtraPrivileges is Enabled
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> .\my-app.exe
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
```

没有窗口产生

我们通过 `@electron/fuses` 对 `onlyLoadAppFromAsar` 进行翻转

```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses write --app "my-app.exe" OnlyLoadAppFromAsar=off
Analyzing app: my-app.exe
Fuse Version: v1
  OnlyLoadAppFromAsar is Enabled and will become Disabled
Writing to app: my-app.exe
Fuses written to disk
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses read --app "my-app.exe"
Analyzing app: my-app.exe
Fuse Version: v1
  RunAsNode is Disabled
  EnableCookieEncryption is Enabled
  EnableNodeOptionsEnvironmentVariable is Disabled
  EnableNodeCliInspectArguments is Disabled
  EnableEmbeddedAsarIntegrityValidation is Enabled
  OnlyLoadAppFromAsar is Disabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
  GrantFileProtocolExtraPrivileges is Enabled
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
```

再次执行 `my-app`

```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> .\my-app.exe
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
[6992:0511/162145.807:FATAL:archive_win.cc(152)] Failed to find file integrity info for resources\default_app.asar
[6992:0511/162145.807:ERROR:crashpad_client_win.cc(868)] not connected
```

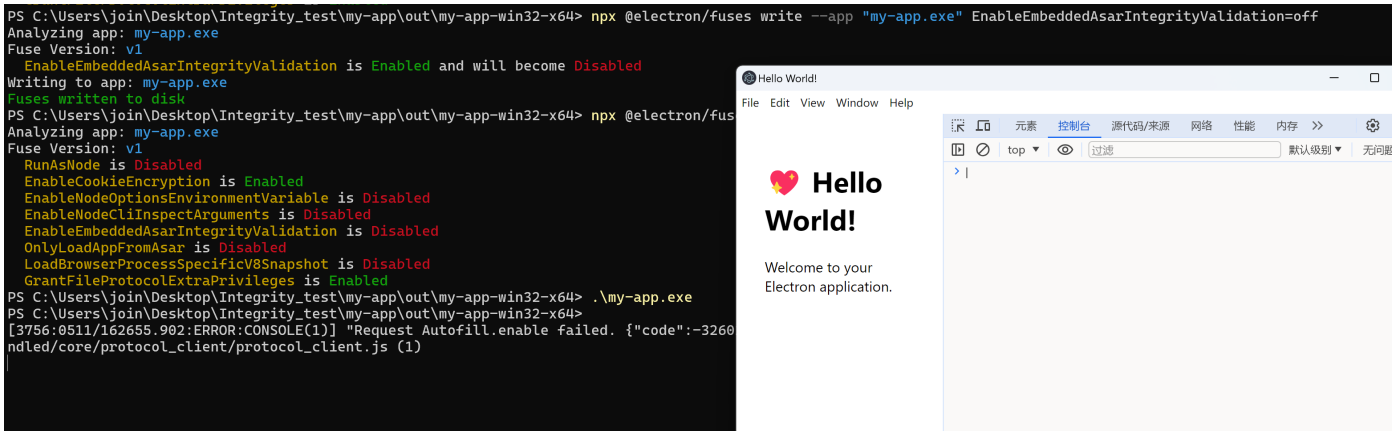
这回完整性校验就失败了，此时我们已经将恶意的 `app.asar` 换回来正常的了，还是完整性校验失败

应该是因为上面过，在 Windows 中做完整性校验是要给定 `.asar` 文件的地址的，当然这个 `Forge` 已经帮我们做了，但我们手动修改 `app.asar -> default_app.asar` ，二进制程序并不知道

此时我们再次通过 `@electron/fuses` 对 `EnableEmbeddedAsarIntegrityValidation` 进行翻转

```
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses write --app "my-app.exe" EnableEmbeddedAsarIntegrityValidation=off
Analyzing app: my-app.exe
Fuse Version: v1
  EnableEmbeddedAsarIntegrityValidation is Enabled and will become Disabled
Writing to app: my-app.exe
Fuses written to disk
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64> npx @electron/fuses read --app "my-app.exe"
Analyzing app: my-app.exe
Fuse Version: v1
  RunAsNode is Disabled
  EnableCookieEncryption is Enabled
  EnableNodeOptionsEnvironmentVariable is Disabled
  EnableNodeCliInspectArguments is Disabled
  EnableEmbeddedAsarIntegrityValidation is Disabled
  OnlyLoadAppFromAsar is Disabled
  LoadBrowserProcessSpecificV8Snapshot is Disabled
  GrantFileProtocolExtraPrivileges is Enabled
PS C:\Users\join\Desktop\Integrity_test\my-app\out\my-app-win32-x64>
```

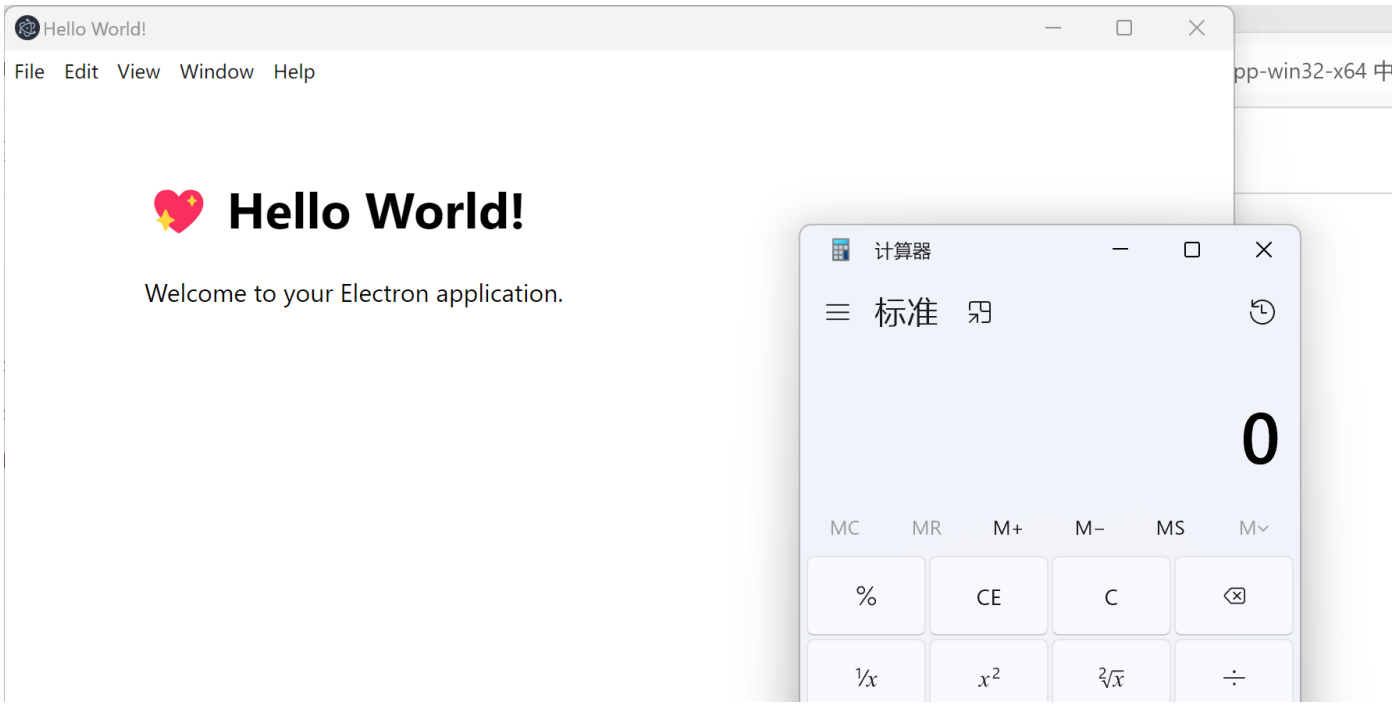
再次执行 `my-app`



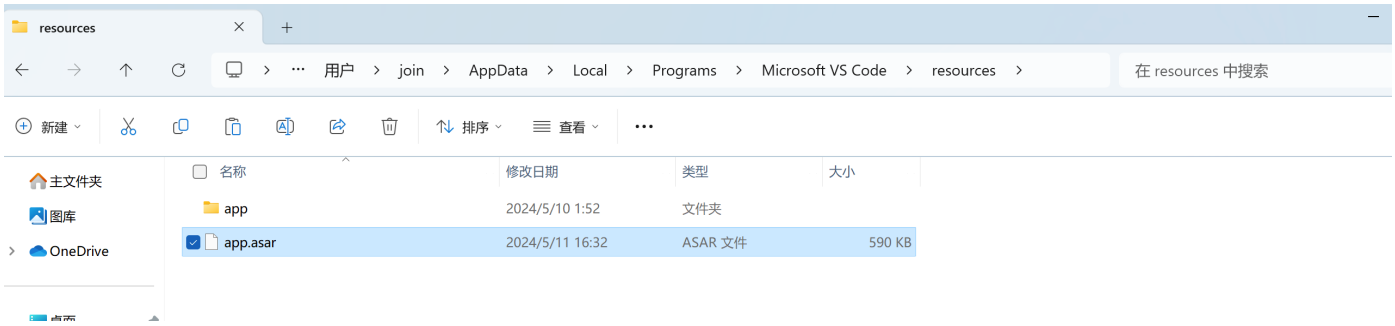
顺利加载了我们的 `default_app.asar`

此时我们就要思考了，其实如果我们想劫持微软的 `vscode` ，我们只需要在它的 `resources` 目录下放置一个 `app.asar` ，可能就可以劫持 `vscode` 了，我们试一下吧

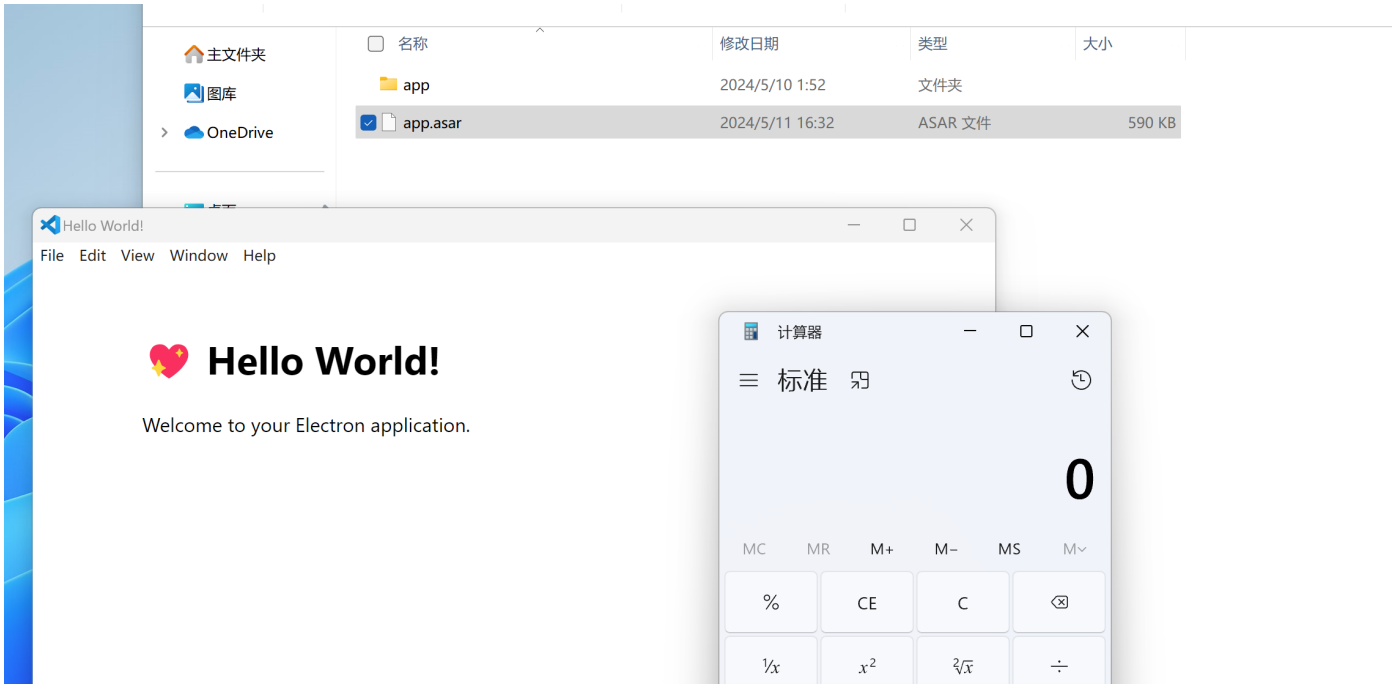
我们修改 `my-app-evil` 项目，让其打开计算器



我们将 `app.asar` 复制到 `vscode` 的 `resources` 目录



打开 `VSCode`



成功劫持 `VSCode`

0x06 总结

`Electron` 官方推出了 ASAR 完整性检查，通过开启 `EnableEmbeddedAsarIntegrityValidation` 这个 fuse 的方式让程序在启动时检查 `.asar` 文件的完整性

工作原理就是在创建 `.asar` 文件时，计算整个文件及分块的 hash ,之后将其按照一定格式存储在 `.asar` 文件的头部，应用程序打包时，会计算该头部的 hash 值，之后固定打包进应用程序

程序执行时，会读取 `.asar` 文件的头部，计算 hash 后和程序内部的值进行对比，如果对比通过了就加载 `.asar` 文件进行执行

这里的问题在于，程序只会校验头部计算后的hash，但不会校验头部中的记录的hash是否有效，因此如果修改了文件内容，文件大小不变，偏移也就不会变（偏移在头部），就能够绕过验证

另一个有趣的 fuse 是 `onlyLoadAppFromAsar` ，这个 `fuse` 如果关闭，程序在加载 `.asar` 文件时会按照以下顺序搜索，加载第一个搜索到的文件

- app.asar
- app
- default_app.asar

如果开启了该fuse，就只加载 `app.asar` ，所以如果关闭该 `fuse` ，就会导致执行流劫持

显然，最新版本的 `VSCode` 就可以实现劫持，因为 `VSCode` 使用的是 `app` 目录

我们将就完整性检查问题再次和 `Electron` 官方交流