思源笔记折腾记录 - 实现简单的笔记发布

作者：leolee
原文链接：https://ld246.com/article/1669963134657
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一、前情提要：

我们之前通过对接wechatsync浏览器插件，实现了把笔记发布到知乎等各个平台的功能。

但是这样发布之后，笔记可能跟思源里面显示的就不太一样了，那么有没有办法“所见即所得”地把己的笔记发布为网站呢？

二、前置知识：

nodejs

还记得在很早的时候，我们实现对代码片段文件夹监听并且自动重新加载思源界面的时候，使用了这一个函数吗？

```
const fs = require('fs')
```

当时我顺嘴提过，这个是nodejs模块的引入方法。

那么nodejs是个什么东西呢？

看看这里（重复，敲键盘，跟我复读一遍：菜鸟教程是个好网站）：

[Node.js 教程 | 菜鸟教程 (runoob.com)]

谁适合阅读本教程?

如果你是一个前端程序员，你不懂得像 PHP、Python 或 Ruby 等动态编程语言，然后你想创建自己的服务，那么 Node.js 是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉 Javascript，那么你将会很容易的学会 Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习 Node.js 也是一个非常好的选择。

前端程序员估计还不算，但是不懂其他编程语言，额，说的就是我了。

ok，那很适合我。

electron

额，为什么又说到这个了呢？

还是一样的，看文档吧：

[简介 | Electron (electronjs.org)]

Electron是一个使用 JavaScript、HTML 和 CSS 构建桌面应用程序的框架。 嵌入 **Chromium** 和 **Node.js** 到 二进制的 Electron 允许您保持一个 JavaScript 代码代码库并创建 在Windows上运行的跨平台应用 macOS和Linux——不需要本地开发 经验。

入门指南

我们推荐您从 **教程**开始， 在开发Electron应用程序并将其分发给用户的过程中向您提供指导。 **示例** 与 **API 文档** 也是浏览并发现新事物的好地方。

Electron Fiddle 运行实例

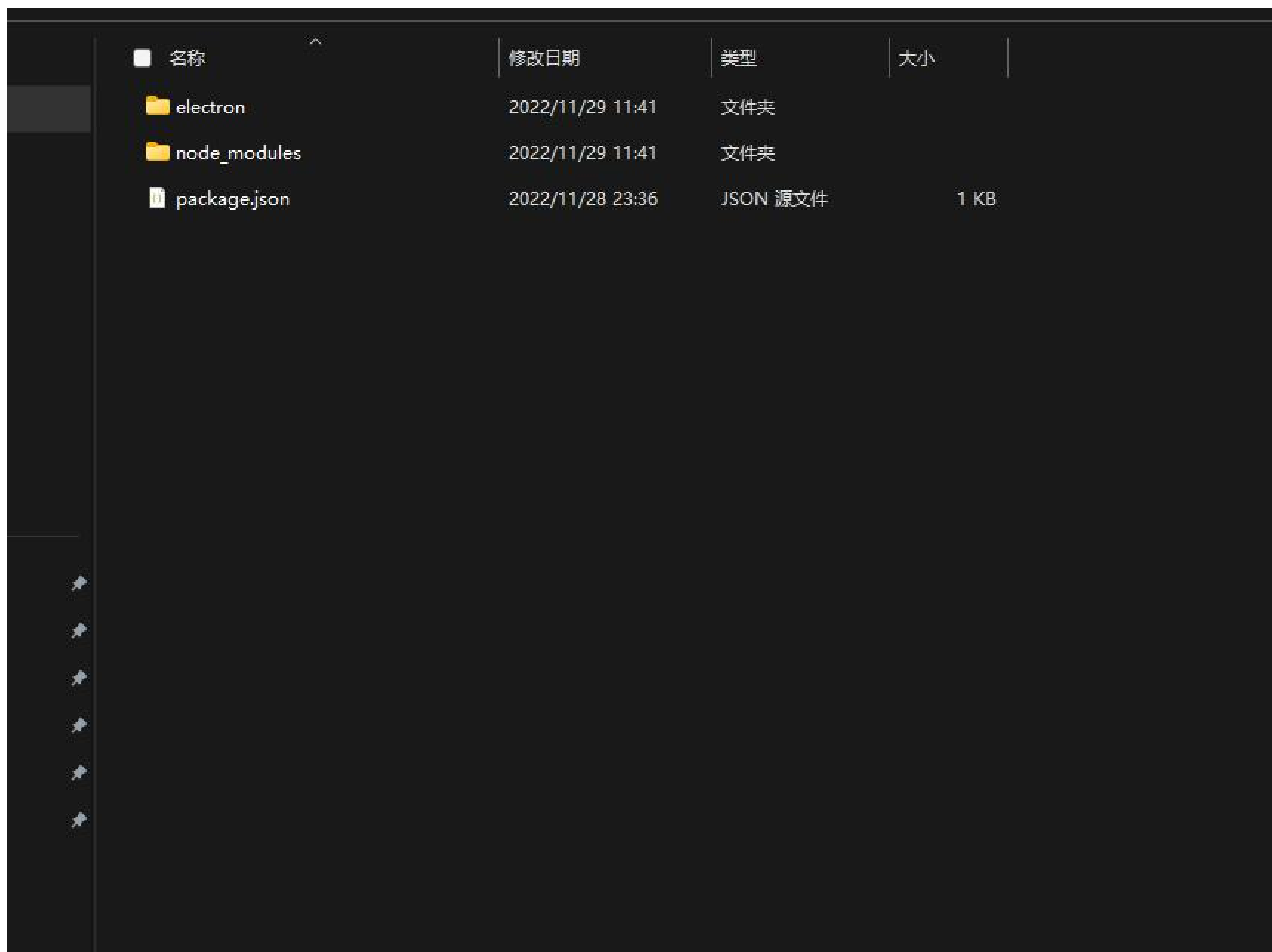
Electron Fiddle 是由 Electron 开发并由其维护者支持的沙盒程序。 我们强烈建议将其作为一个学习工具来安装，以便在开发过程中对Electron的api进行实验或对特性进行原型化。

然后我们去看一下，思源的安装文件夹：

名称	修改日期	类型	大小
locales	2022/11/29 11:41	文件夹	
resources	2022/11/29 11:41	文件夹	
chrome_100_percent.pak	2022/11/28 23:36	PAK 文件	127 KB
chrome_200_percent.pak	2022/11/28 23:36	PAK 文件	176 KB
d3dcompiler_47.dll	2022/11/28 23:36	应用程序扩展	4,777 KB
ffmpeg.dll	2022/11/28 23:36	应用程序扩展	2,676 KB
icudtl.dat	2022/11/28 23:36	DAT 文件	10,206 KB
libEGL.dll	2022/11/28 23:36	应用程序扩展	464 KB
libGLESv2.dll	2022/11/28 23:36	应用程序扩展	7,160 KB
LICENSE	2022/11/28 23:36	文件	35 KB
LICENSE.electron.txt	2022/11/28 23:36	文本文档	2 KB
LICENSES.chromium.html	2022/11/28 23:36	Microsoft Edge ...	6,483 KB
resources.pak	2022/11/28 23:36	PAK 文件	5,278 KB
SiYuan.exe	2022/11/28 23:36	应用程序	150,553 KB
snapshot_blob.bin	2022/11/28 23:36	BIN 文件	410 KB
Uninstall SiYuan.exe	2022/11/28 23:37	应用程序	166 KB
v8_context_snapshot.bin	2022/11/28 23:36	BIN 文件	711 KB
vk_swiftshader.dll	2022/11/28 23:36	应用程序扩展	4,927 KB
vk_swiftshader_icd.json	2022/11/28 23:36	JSON 源文件	1 KB
vulkan-1.dll	2022/11/28 23:36	应用程序扩展	859 KB

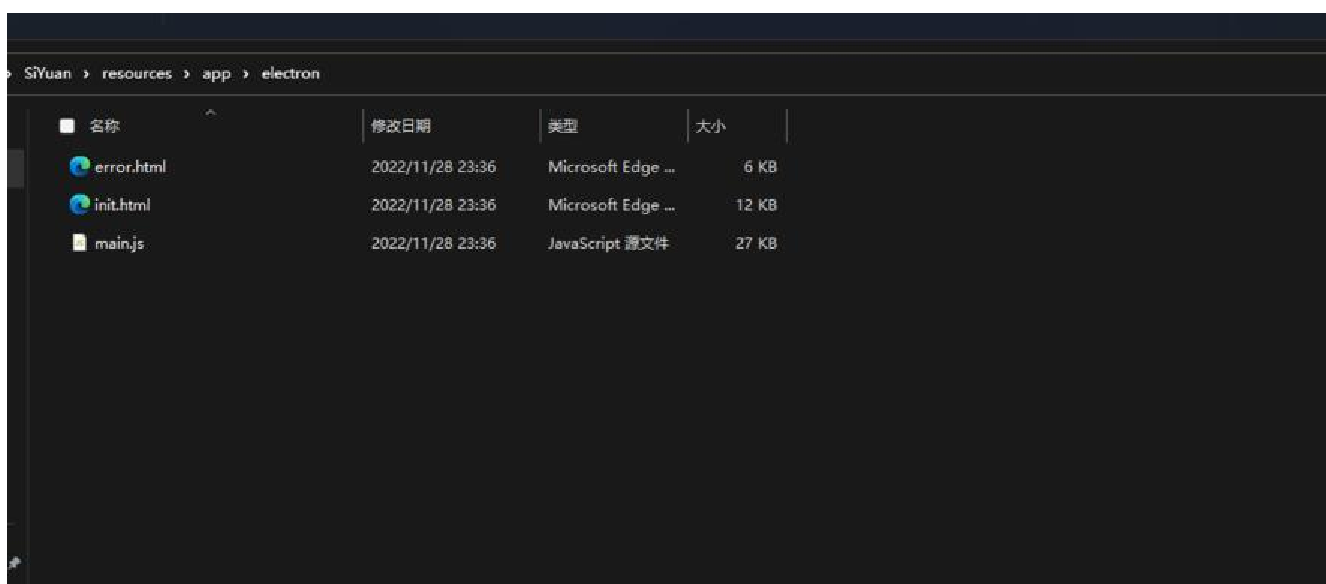
这还看不出来啥对吧

再看看里面的\resources\app

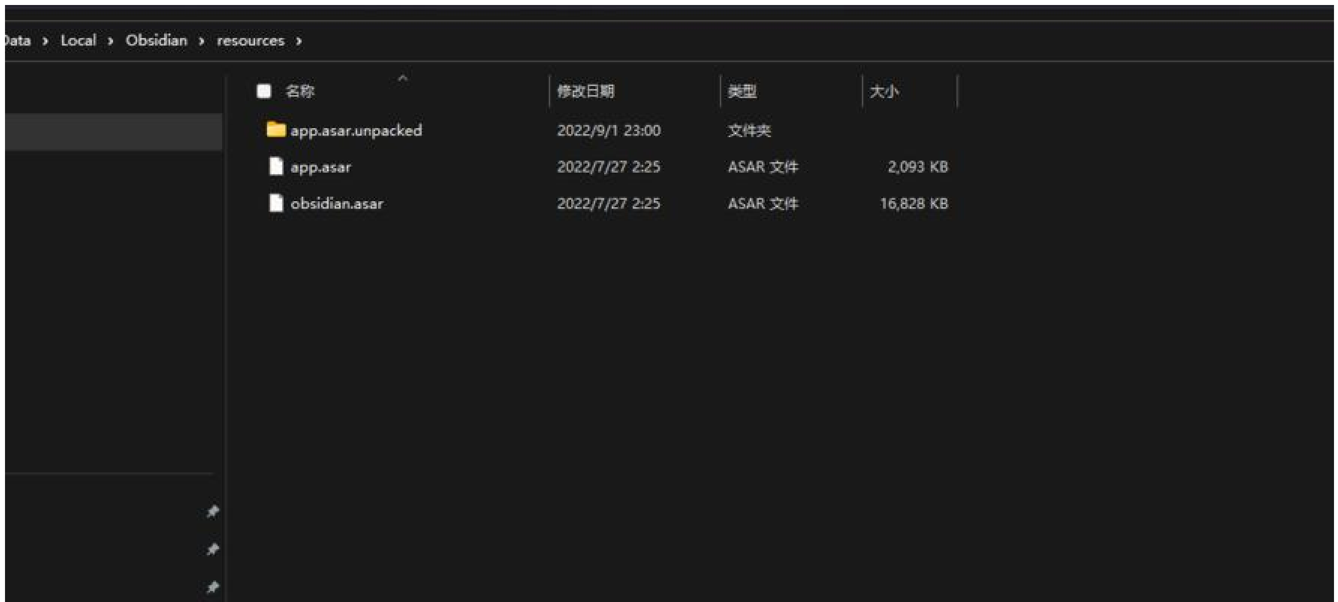


嘿嘿嘿，看到了没有：electron，破案了（屁咧，隔几次升级公告里都有升级electron版本的好嘛）。

额，这里扩展一点说，为什么我经常说思源的扩展比obsidian要容易：



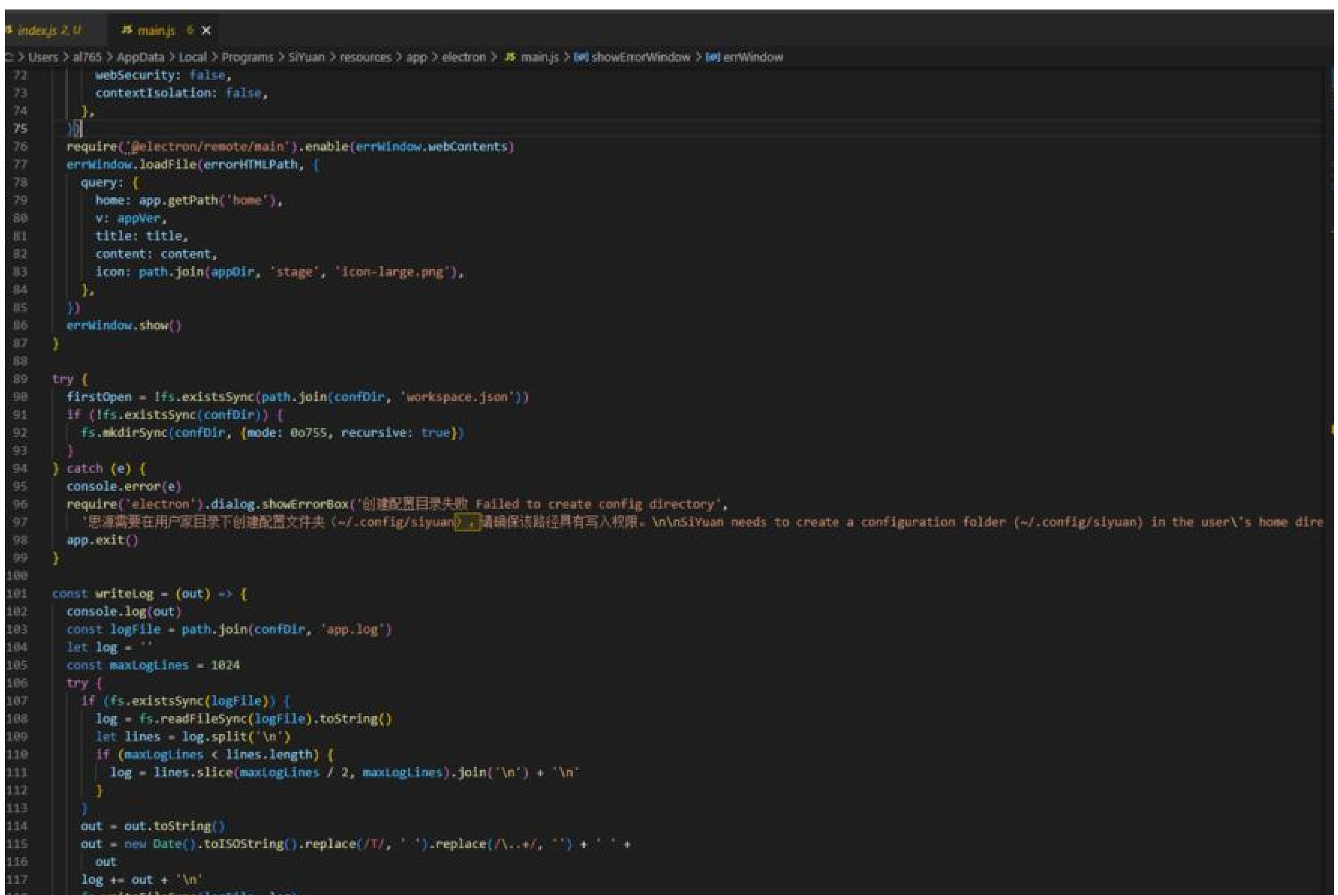
上面这是思源的resources文件夹，下面这个是obsidian的：



能看出区别吗?

思源的源代码（编译后的）是直接暴露在这个文件夹下面的，obsidian里面只有一个app.sar。

然后更进一步的，我们打开思源的main.js:



这个就是思源启动的时候，前端所运行的后台代码了（有点绕，其实是主进程的代码，但是这个不是在的重点），你可以通过改动它来修改思源的启动过程。

所以说不要再闹出这种笑话了好吗：

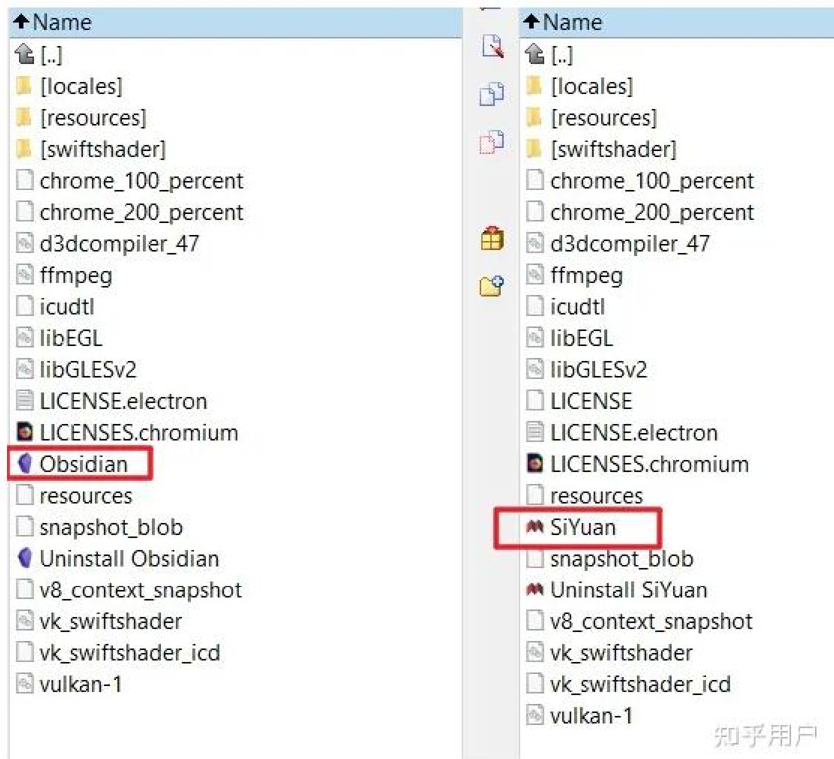
原文链接：思源笔记折腾记录 - 实现简单的笔记发布

看到上面的没有，这个的意思是说，如果窗口在创建的时候，

匿名用户

1 人赞同了该回答

是兄弟吧?



编辑于 2022-02-05 21:33

electron的渲染进程和BrowserWindow

`for-tab` 事件。

- `webPreferences` Object (可选) - 网页功能设置。
  - `devTools` boolean (可选) - 是否开启 DevTools. 如果设置为 `false`, 则无法使用 `BrowserWindow.webContents.openDevTools ()` 打开 DevTools。 默认值为 `true`。
  - `nodeIntegration` boolean (可选) - 是否启用Node integration. 默认值为 `false`.
  - `nodeIntegrationInWorker` boolean (可选) - 是否在Web工作器中启用了Node集成. 默认值为 `false`. 更多内容参见 **多线程**.
  - `nodeIntegrationInSubFrames` boolean (可选项)(实验性)，是否允许在子页面(iframe)或子窗口(child window)中集成Node.js； 预先加载的脚本会被注入到每一个iframe，你可以用 `process.isMainFrame` 来判断当前是否处于主框架（main frame）中。

看到上面的没有，这个的意思是说，如果窗口在创建的时候，nodeIntegration为true的话，就可以窗口的中获取nodejs环境。

而思源创建的主窗口是这样的：

```
mainWindow = new BrowserWindow({
  show: false,
  backgroundColor: '#FFF', // 桌面端主窗体背景色设置为 `#FFF` Fix https://github.com/siyuan-note/siyuan/issues/4544
  width: windowState.width,
  height: windowState.height,
  x,
  y,
  fullscreenable: true,
  fullscreen: windowState.fullscreen,
  trafficLightPosition: {x: 8, y: 8},
  webPreferences: {
    nodeIntegration: true,
    nativeWindowOpen: true,
    webviewTag: true,
    webSecurity: false,
    contextIsolation: false,
  },
  frame: 'darwin' === process.platform,
  titleBarStyle: 'hidden',
  icon: path.join(appDir, 'stage', 'icon-large.png'),
})
```

goooood，我们有node环境了，所以之前才可以这样做：

```
    if(window.require){
        let {监听文件修改}= await import('./util/file.js')
        let 监听选项 = {
            监听路径:工作空间.代码片段路径,
            监听配置:{
                persistent :true,
                recursive :true
            },
            文件类型:['js'],
            事件类型:['change']
        }
        监听文件修改(监听选项)
    }

export function 监听文件修改(监听选项){
    const fs = require('fs')
    fs.watch(监听选项.监听路径,监听选项.监听配置,(type,fileName)=>{
        if(监听选项.事件类型.indexOf(type)>=0){
            let 扩展名 =  fileName.split('.').pop()
            if(监听选项.文件类型.indexOf(扩展名)>=0){
                window.location.reload()
            }
        }
    })
}
```

好了，条件基本上就是这样，我们可以开始鼓捣了。

二、发布服务的实现过程

1、安装nodejs用于包管理

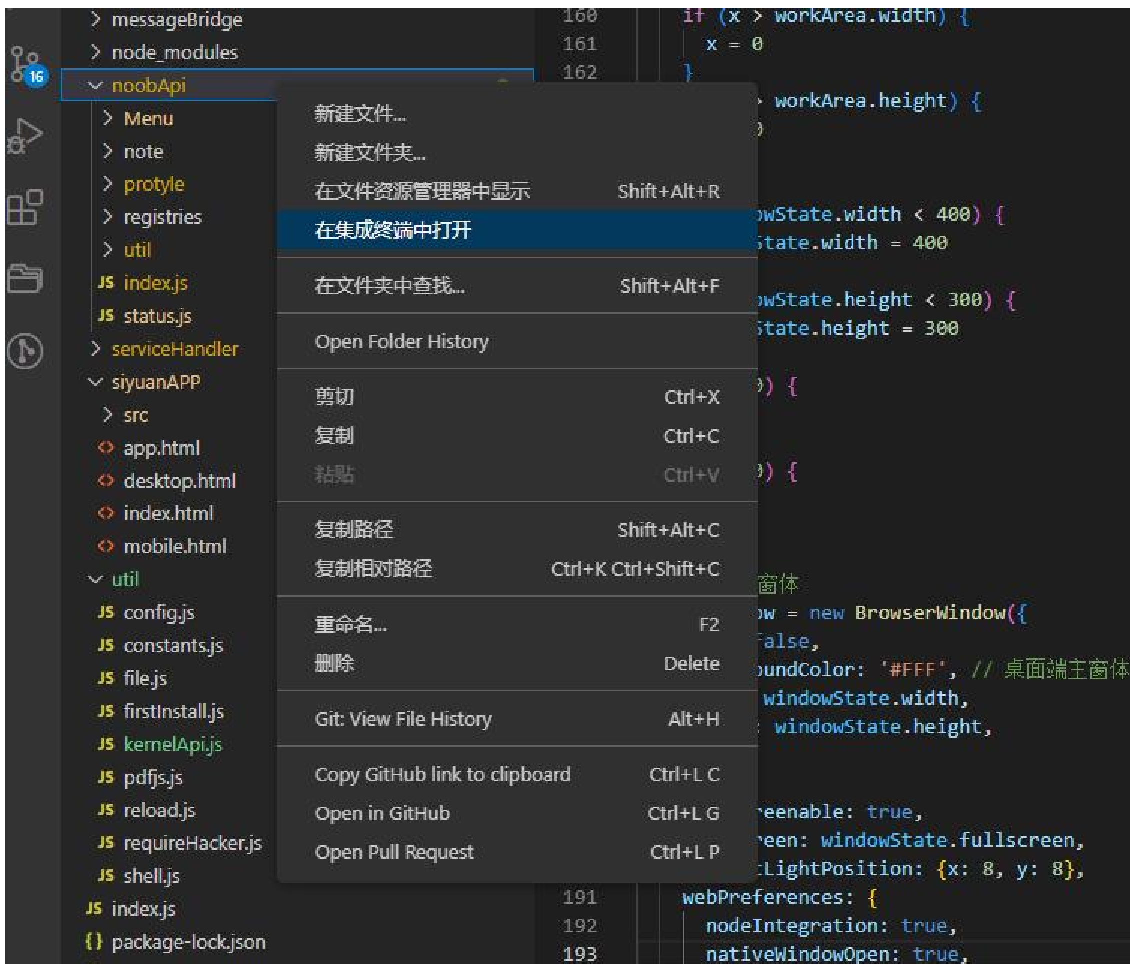
按说思源的渲染进程里面已经有了node环境，我们应该不需要一个nodejs了，但是这里我们还是安一个，不为了别的，单纯就是为了用它管理和下载包方便实现一些功能。

Node.js 安装配置 | 菜鸟教程 (runoob.com)

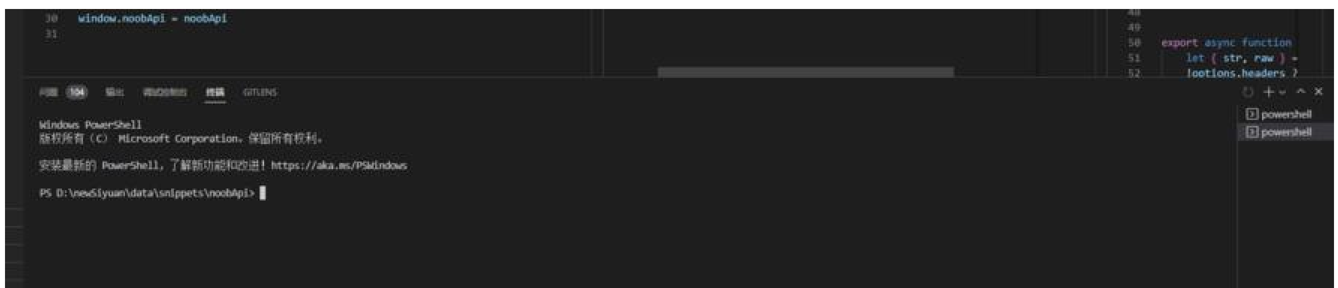
nodejs里有好几个能够实现web服务器的包，这次我们使用express，因为它比较简单嘛。

首先打开vscode（我使用的是它啦，你用别的也不是不行）

然后用打开文件夹的功能，打开我们的snippets文件夹



然后像上面一样，在集成终端中打开noobApi文件夹。



这个时候你应该能够看到上面这样的界面了。

然后在这里输入：

npm i express --registry=https://registry.npmmirror.com

闪过一堆不明觉历的代码之后我们的文件夹里面就多了这么几个文件：

noobApi\package.json

这个是模块配置文件，暂时可以不用管它。

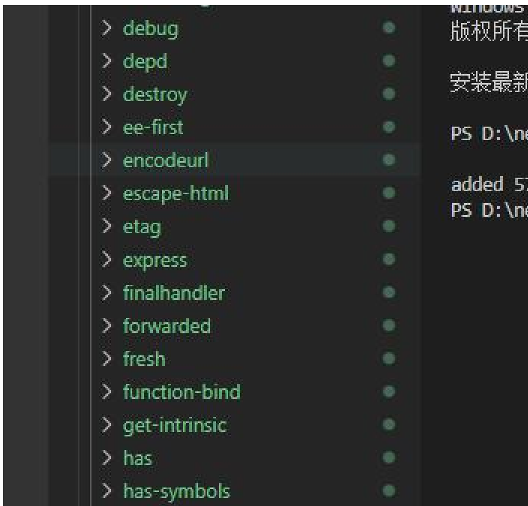
noobApi\package-lock.json

这个是依赖版本信息，暂时也可以不用管它。

noobApi\node_modules

这个文件夹是emm， 黑洞，暂时要管它。

node_modules文件夹是node的依赖包所在的位置,我们之前安装的express就在这个里面。



好了，现在已经有了express我们尝试一下引入它吧：

```
import 工作空间  from './workspace/index.js'
if(!window.noobApi){
    if(window.require){
        let {监听文件修改}= await import('./util/file.js')
        let 监听选项 = {
            监听路径:工作空间.代码片段路径,
            监听配置:{
                persistent :true,
                recursive :true
            },
            文件类型:['js'],
            事件类型:['change']
        }
        监听文件修改(监听选项)
        await import('./server/index.js')
    }
    await import('./api.js')
}

export default window.noobApi
```

noobApi\index.js

```
const express = require('express')
console.log(express)
```

不出意外，一切正常的话，emm，你会看到这样一个友善而且醒目的报错：

```
▸GET https://www.googletagmanager.com/gtag/js?id=G-L7WEXVQCR9 net::ERR_QUIC_PROTOCOL_ERROR 200
▸Uncaught (in promise) Error: Cannot find module 'express'
Require stack:
- electron/js2c/renderer_init
    at Module._resolveFilename (node:internal/modules/cjs/loader:940:15)
    at i._resolveFilename (node:electron/js2c/renderer_init:33:1095)
    at Module._load (node:internal/modules/cjs/loader:785:27)
    at c._load (node:electron/js2c/asar_bundle:5:13339)
    at i._load (node:electron/js2c/renderer_init:33:356)
    at Module.require (node:internal/modules/cjs/loader:1012:19)
    at require (node:internal/modules/cjs/helpers:102:18)
    at index.js:1:17
```

这是为什么呢?

在渲染进程中直接使用require引入模块的话，起始地址其实是应用的根目录，也就是之前看到main.j
的地方。

所以我们需要一点点操作：

noobApi\server\util\requireHacker.js

```
import { workspaceDir } from "./file.js"
let re = null
let realRequire = null
if (window.require) {
	const fs = require("fs")
	const path = require("path")
	if (!window) {
		const window = global
	}
	if (window.require.cache) {
		realRequire = window.require
	}
	if (realRequire) {
		const path = require("path")
		re = function (moduleName, base) {
			if (module) {

				let _load = module.__proto__.load
				if (!module.__proto__.load.hacked) {
					module.__proto__.load = function (filename) {
						let realfilename = filename
						try {

							(_load.bind(this))(filename)
						} catch (e) {
							if (e.message.indexOf('Cannot find module') >= 0 && e.me
sage.indexOf(filename) >= 0) {
								if (global.ExternalDepPathes) {
									let flag
									let modulePath
									global.ExternalDepPathes.forEach(depPath => {
										if (fs.existsSync(path.join(depPath, moduleN
me))) {
											if (!flag) {
												console.file_warn ? console.file_wa
n(`模块${moduleName}未找到,重定向到${path.join(depPath, moduleName)}`) : console.warn(`
块${moduleName}未找到,重定向到${path.join(depPath, moduleName)}`)
```

```
                                                                    filename = path.join(depPath, file
ame);
                                                                    try {
                                                                        (_load.bind(this))(filename)

                                                                        flag = true
                                                                    } catch (e) {

                                                                    }
                                                                } else {

                                                                    console.warn(`模块${moduleName
在${modulePath}已经找到,请检查外部路径${path.join(depPath, moduleName)}是否重复安装`)
                                                                }

                                                            }
                                                        });
                                                        if (!flag) {
                                                            console.error(e)
                                                            throw new Error(`无法加载模块${realfilename}
)

                                                        }

                                                    }
                                                    else {
                                                        console.error(e)
                                                        throw new Error(`无法加载模块${realfilename}`)
                                                    }
                                                }
                                                else {

                                                    throw (e)
                                                }

                                            }
                                        }


                                        module.__proto__.load.hacked = true
                                    }

                                }
                                if (!window.realRequire) {
                                    window.realRequire = realRequire
                                }
                                let that = window
                                if (base) {
                                    moduleName = path.resolve(base, moduleName)
                                }

                                if (workspaceDir) {
                                    if (this) {
                                        that = this
```

```
                    }
                    try {
                        if (that.realRequire) {
                            let _module = that.realRequire(moduleName)
                            return _module
                        }
                        else {
                            let _module = window.realRequire(moduleName)
                            return _module
                        }
                    } catch (e) {
                        if (e.message.indexOf('Cannot find module') >= 0) {
                            if (!(moduleName.startsWith("/") || moduleName.startsWith("./")
| moduleName.startsWith("../"))) {
                                if (global.ExternalDepPathes) {
                                    let flag
                                    let modulePath
                                    global.ExternalDepPathes.forEach(depPath => {
                                        if (fs.existsSync(path.join(depPath, moduleName))
 {

                                            if (!flag) {
                                                console.file_warn ? console.file_warn(`
块${moduleName}未找到,重定向到${path.join(depPath, moduleName)}`) : console.warn(`模块$
moduleName}未找到,重定向到${path.join(depPath, moduleName)}`)
                                                moduleName = path.join(depPath, mo
uleName)

                                                modulePath = path.join(depPath, modu
eName)

                                                flag = true
                                            } else {

                                                console.warn(`模块${moduleName}在$
modulePath}已经找到,请检查外部路径${path.join(depPath, moduleName)}是否重复安装`)
                                            }

                                        }
                                    });
                                }
                            } else {
                                moduleName = path.resolve(module.path, moduleName)
                            }
                            try {

                                let _module
                                _module = that.realRequire(moduleName)
                                return _module
                            }
                            catch (e) {
                                throw e
                            }
                        }
                        else {
                            throw e
```

```
                                                        }
                                        }
                                }
                                else return window.require(moduleName)
                        }
                }

        }
if (window.require && re) {
        window.require = re
        window.realRequire = realRequire
        if (window.realRequire && window.realRequire.cache) {
                window.realRequire.cache.electron.__proto__.realRequire = realRequire.cache.electr
n.__proto__.require
                window.realRequire.cache.electron.__proto__.require = re
        }
        window.require.setExternalDeps = (path) => {
                if (!window.ExternalDepPathes) {
                        window.ExternalDepPathes = []
                }
                if (path && !window.ExternalDepPathes.indexOf(path) >= 0) {
                        window.ExternalDepPathes.push(path)
                        window.ExternalDepPathes = Array.from(new Set(window.ExternalDepPathes))
                }
        }
        re.setExternalDeps(`${workspaceDir}`)
        window.require.setExternalBase = (path) => {
                if (!window.ExternalDepPathes) {
                        window.ExternalDepPathes = []
                }
                if (!window.ExternalBase) {
                        window.ExternalBase = path
                }
                else {
                        console.error('不能重复设置外部依赖路径')
                }
        }
}



export default window.require
```

这样做了之后，就可以设置外部依赖来引入包了：

```
import {代码片段路径} from './util/file.js'
import './util/requireHacker.js'
require.setExternalDeps(代码片段路径 + `/noobApi/node_modules`)

const express = require('express')
console.log(express)
```

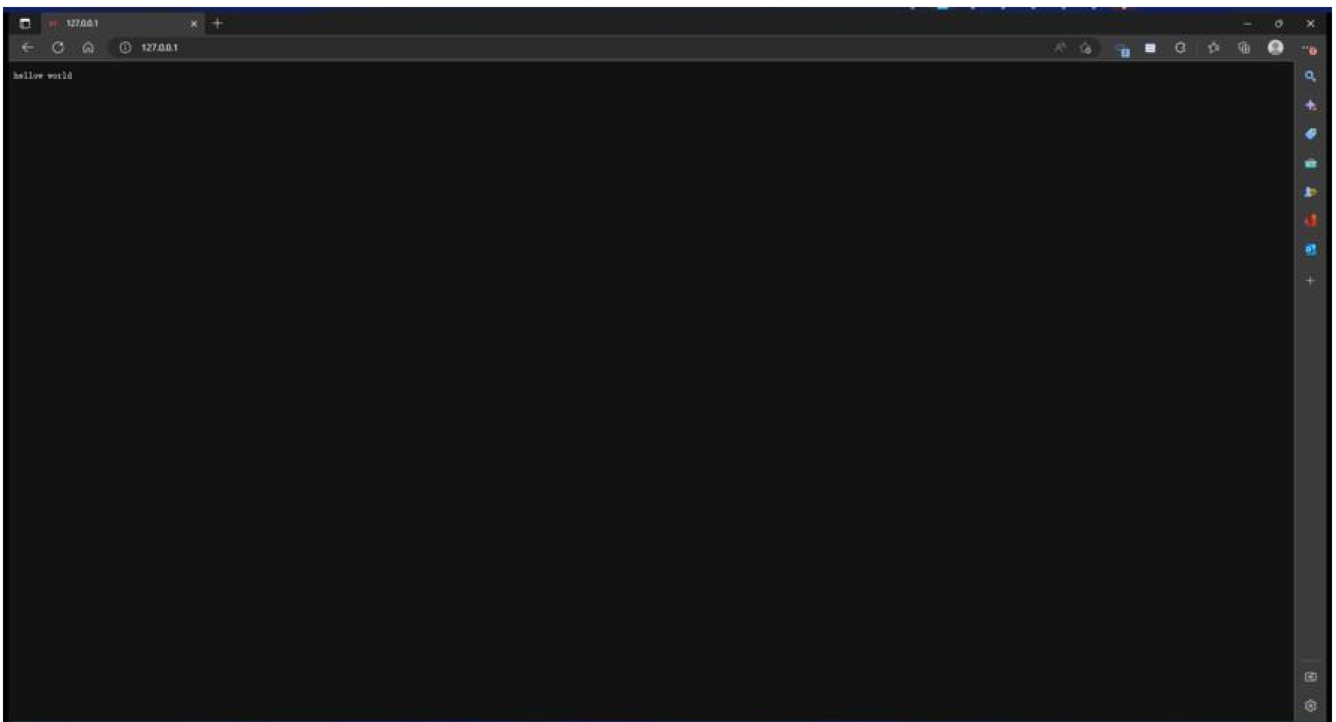
现在没报错了，说明我们的引入是成功的。

现在express也有了，让我们来弄一个hello world

```
import {代码片段路径} from './util/file.js'
import './util/requireHacker.js'
require.setExternalDeps(代码片段路径 + `/noobApi/node_modules`)
const http = require('http')
const express = require('express')
const 发布应用 = express()
const 发布端口 ='80'
发布应用.use('/',(req,res)=>{
    res.end('hellow world')
})
let 发布服务器 = http.createServer(发布应用);
发布服务器.listen(发布端口, () => {
    console.log("发布服务已经启动")

})
```

这时你打开浏览器，输入127.0.0.1就会看到这个：



好了，我们的发布服务就这样写完了。。。。。。吗？

肯定是没有啊，现在它还只能显示一个hello world啊

所以这里要让它能显示出思源的文档。

渲染文档发布结果

我们来创建一个函数，还记得之前做发布到知乎的时候，获取渲染结果的时候，我们使用了这样几个口：

```
let 块id = noobApi.自定义菜单.当前菜单.菜单状态.当前块id
```

```
    let stmt = `select * from blocks where id in (select root_id from blocks  where id = "${块id}"
`
    let 文档数据 = (await noobApi.核心api.sql({ stmt: stmt }))[0]
    let 文档内容 = await noobApi.核心api.exportPreview(
        {
            "id": 文档数据.id
        }
    )
    let 文档属性 = await noobApi.核心api.getDocInfo(
        {
            "id": 文档数据.id
        }
    )
    let 发布数据 = {}
    发布数据.title = 文档属性.ial.title
    发布数据.markdown = 文档数据.markdown
    发布数据.content = 转换图片地址(文档内容)
    发布数据.desc = 文档属性.ial.memo
    发布数据.thumb = 文档属性.ial['title-img']
```

其中exportPreview（api/export/exportPreview）可以获取导出的文档预览。

所以这里我们可以这样做：

```
async function 渲染页面(块id){
    let stmt = `select * from blocks where id in (select root_id from blocks  where id = "${块id}"
`
    let 文档数据 = (await noobApi.核心api.sql({ stmt: stmt }))[0]
    let 文档内容 = await noobApi.核心api.exportPreview(
        {
            "id": 文档数据.id
        }
    )
    let 文档属性 = await noobApi.核心api.getDocInfo(
        {
            "id": 文档数据.id
        }
    )
    let 发布数据 = {}
    发布数据.title = 文档属性.ial.title
    发布数据.markdown = 文档数据.markdown
    发布数据.content = 转换图片地址(文档内容)
    发布数据.desc = 文档属性.ial.memo
    发布数据.thumb = 文档属性.ial['title-img']
    return 发布数据
}
function 转换图片地址(文档内容) {
    let div = document.createElement('div')
    div.innerHTML = 文档内容.content ? 文档内容.content : 文档内容.html
    div.querySelectorAll('[src]').forEach(
        el => {
            if (el.getAttribute('src').startsWith('assets')) {
                el.setAttribute('src', window.location.origin + '/' + el.getAttribute('src'))
```

```
            }
        }
    )
    div.innerHTML += '<p>本文使用<a href="https://b3log.org/siyuan/">思源笔记</a>写作</
>'
    div.innerHTML += '<p>本文使用<a href="http://publish.chuanchengsheji.com/">椽承设计<
a><a href="https://github.com/leolee9086/snippets">小工具</a>配合同步</p>'
    return div.innerHTML
}
```

然后使用这个函数来渲染响应：

```
发布应用.use('/',async(req,res)=>{
//这个id是思源的帮助页面首页啦
    let 块id = '20200812220555-lj3enxa'
    let 页面数据 = await 渲染页面(块id)
    res.end(页面数据.content)
})
```

这个时候，我们再去访问127.0.0.1就会看到：

emmmmmmmm

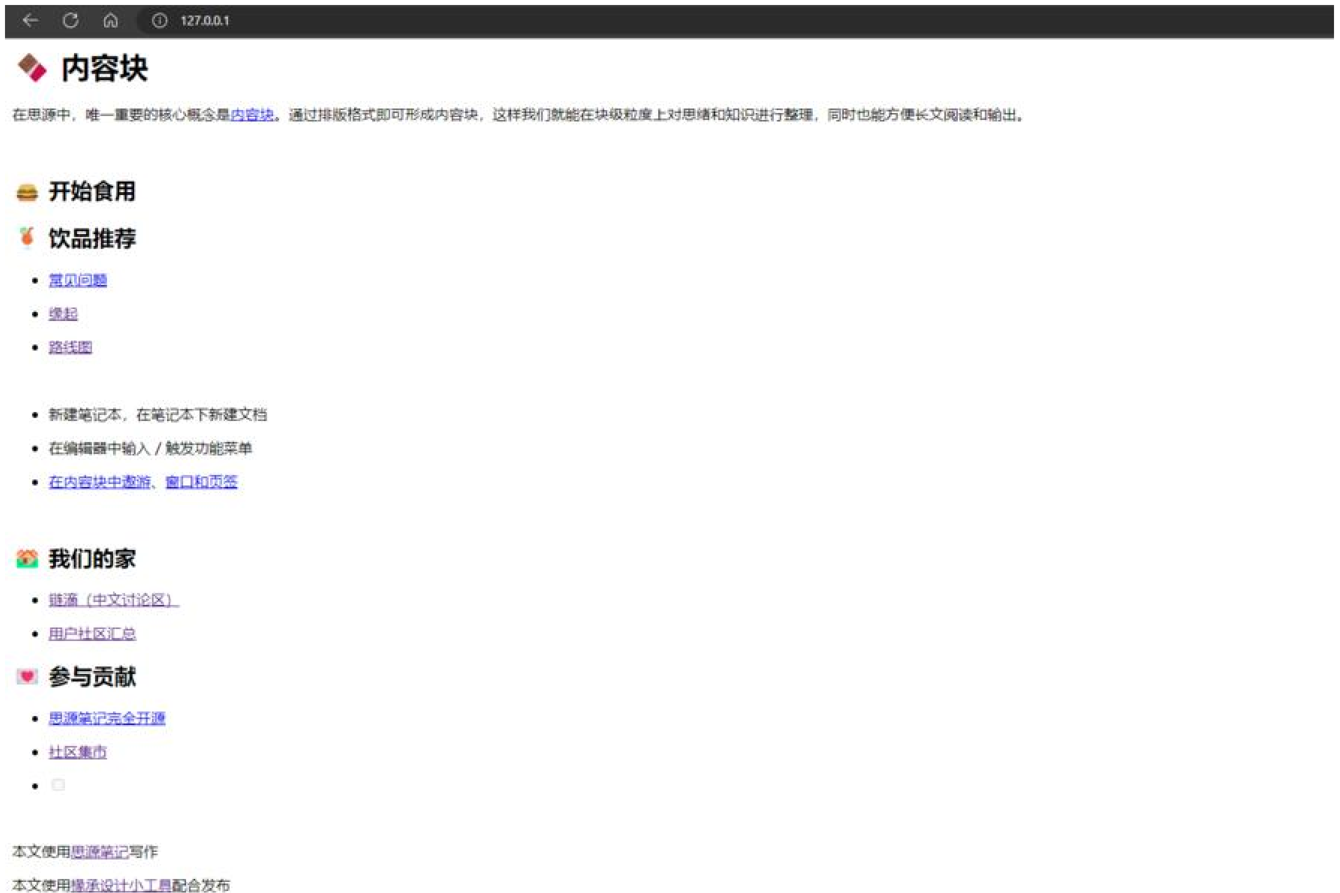
这是因为我们编码问题，如果不使用合适的编码的话，就会出现这样蛋疼的错误。

所以需要改一下正确的编码，其实只需要这样就可以了：

```
发布应用.use('/',async(req,res)=>{
    let 块id = '20200812220555-lj3enxa'
    let 页面数据 = await 渲染页面(块id)
    //指定一下编码
    res.setHeader("Content-Type", "text/html; charset=utf-8")
    res.end(页面数据.content)
```

```
})
```

这个时候再查看它，就会看到：



这样就看到正常的内容了。

不过现在它的id是写死的，所以我们来搞一下根据id渲染：

根据id渲染文档内容：

这里的问题是怎么获取id，我们看一下这个函数：

```
发布应用.use('/',async(req,res)=>{
    let 块id = '20200812220555-lj3enxa'
    let 页面数据 = await 渲染页面(块id)
    res.setHeader("Content-Type", "text/html; charset=utf-8")
    res.end(页面数据.content)
})
```

它的含义是在访问路径匹配'/' 的时候,就使用后面的回调函数来响应请求，而这个回调函数一般有三参数(req,res,next)。

其中req就代表了请求。

显然我们要获取块id的话，只能从请求中获取。

这里有几种办法，可以参考这里：

Express routing - Express 中文文档 | Express 中文网 (expressjs.com.cn)

查了一下文档之后，就试试用这种方式吧：

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

The name of route parameters must be made up of "word characters" ([A-Za-z0-9_]).

Since the hyphen (-) and the dot (.) are interpreted literally, they can be used along with route parameters for useful purposes.

依样画葫芦，把我们的路由改成这样：

```
发布应用.use('/',async(req,res,next)=>{
    console.log(req)
   req.url=='/'? res.redirect('/20200812220555-lj3enxa'):null
   next()
})
发布应用.use('/:blockID',async(req,res)=>{
    console.log(req.params)
    let 块id = req.params.blockID
    let 页面数据 = await 渲染页面(块id)
    res.setHeader("Content-Type", "text/html; charset=utf-8")
    res.end(页面数据.content)
})
```

然后现在我们再试着访问一下（这个id就是这篇文档的）：

127.0.0.1/20221202124825-42vkazf

一、前情提要：

我们之前通过对接wechatsync浏览器插件，实现了把笔记发布到知乎等各个平台的功能。

但是这样发布之后，笔记可能跟思源里面显示的就不太一样了，那么有没有办法“所见即所得”地把自己的笔记发布为网站呢？

二、前置知识：

nodejs

还记得在很早的时候，我们实现对代码片段文件夹监听并且自动重新加载思源界面的时候，使用了这样一个函数吗？

```
const fs = require('fs')
```

当时我顺嘴提过，这个是nodejs模块的引入方法。

那么nodejs是个什么东西呢？

看看这里（重复，敲键盘，跟我复读一遍：菜鸟教程是个好网站）：

Node.js 教程 | 菜鸟教程 (runoob.com)

谁适合阅读本教程?

如果你是一个前端程序员，你不懂得像 PHP、Python 或 Ruby 等动态编程语言，然后你想创建自己的服务，那么 Node.js 是一个非常好的选择。
Node.js 是运行在服务端的 JavaScript，如果你熟悉 Javascript，那么你将会很容易的学会 Node.js。
当然，如果你是后端程序员，想部署一些高性能的服务，那么学习 Node.js 也是一个非常好的选择。

前端程序员估计还不算，但是不懂其他编程语言，额，说的就是我了。

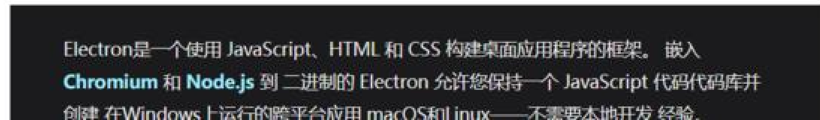
ok，那很适合我。

electron

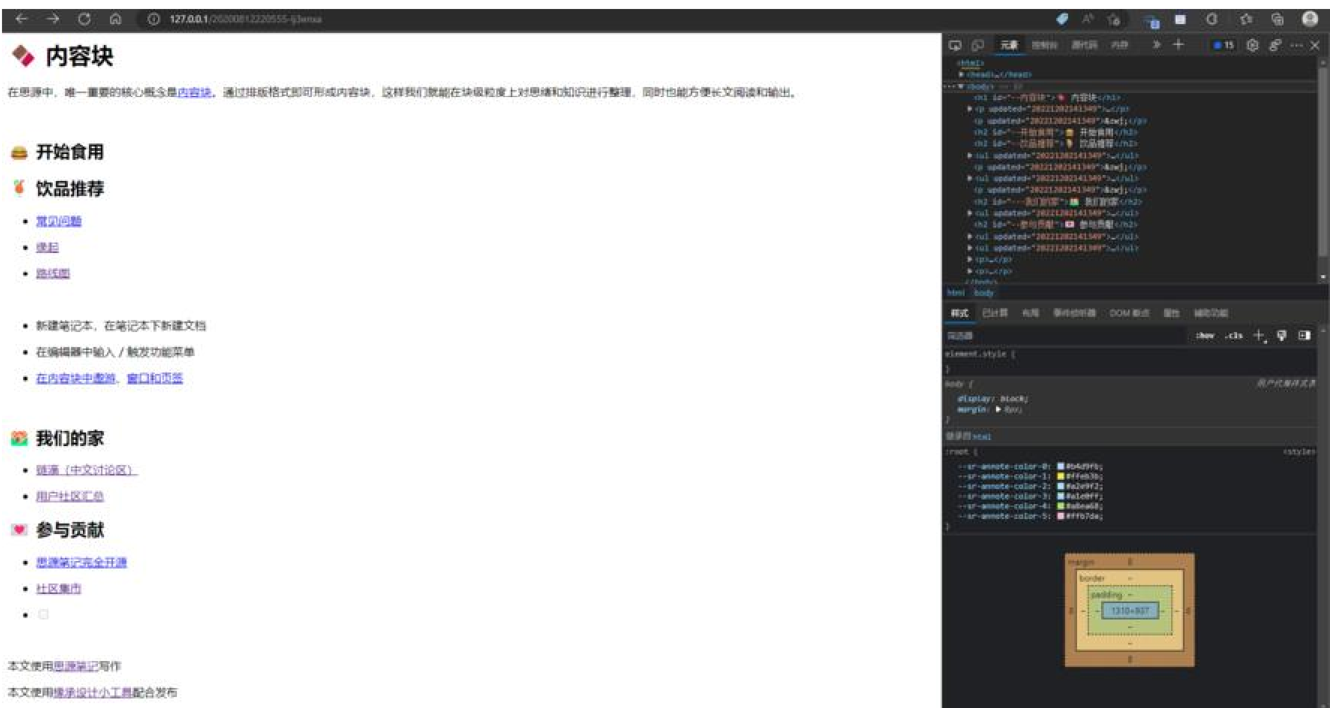
额，为什么又说到这个了呢？

还是一样的，看文档吧：

简介 | Electron (electronjs.org)



访问首页（127.0.0.1）的话，就会重定向到帮助文档首页啦：



现在你在局域网上的小伙伴应该已经能够访问你的文档内容了，但是现在所有的文档都能够被访问，像不大好，我们通过publish-access属性设置一下它，如果这个属性值是public的话，就能够访问，

果不是的话，就返回一个亲切而且友好的提示：

```
if (文档属性.ial && 文档属性.ial['custom-publish-access']) {
        let 发布数据 = {}
        发布数据.title = 文档属性.ial.title
        发布数据.markdown = 文档数据.markdown
        发布数据.content = 转换图片地址(文档内容)
        发布数据.desc = 文档属性.ial.memo
        发布数据.thumb = 文档属性.ial['title-img']
        return 发布数据
    } else {
        return {content:'<h1 style="font-size:200px;color:red">这个文档除了我自己谁都不准看</h
>'}
    }
```

这个文档除了我自己谁都不准看

你看，无论是谁看到这个温暖的大红色，都会理解背后那个温柔而羞涩的作者的对吧。

或者你也可以自己改得更加热情一点：

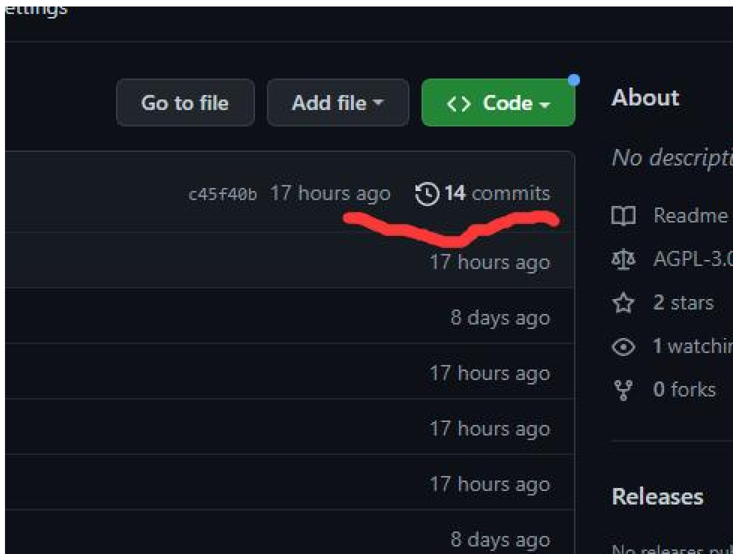
ok，现在你想要把文档给谁看又不想他能够修改的话，只需要把<你的局域网ip>/<块的id>发给他他就能够看到文档的内容啦（我个人建议如果你不希望被打死的话最好不要用上面那两条提示语）。

你说公网发布?这个不是这次要说的,下次再说,下次再说.

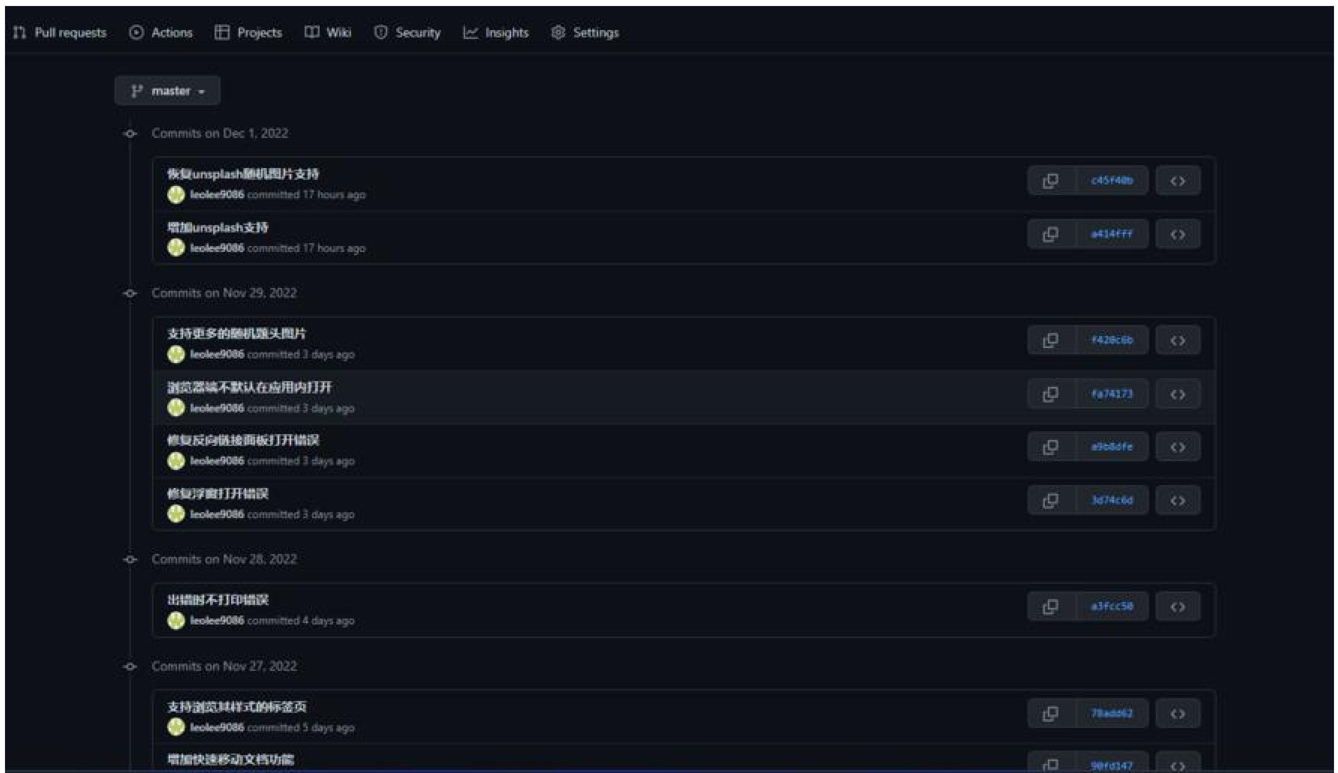
目前的代码片段的地址位于：

[leolee9086/snippets (github.com)](https://github.com/leolee9086/snippets)

如果你希望看到实现过程的话可以看看这个位置



里面能够看到我所有的提交记录。



就像这样，越早的记录对新手应该越友好。。。。。吧。

▯