简介

如何使用 Node.js 和 Electron APIs。

所有的[Node.js's built-in modules](https://nodejs.org/api/)在 Electron 中都可用，并且所有的 node 的第三方组件也可以放心使用（包括[自身的模块](https://electron.org.cn/doc/tutorial/using-native-node-modules.html)）。

Electron 也提供了一些额外的内置组件来开发传统桌面应用。一些组件只可以在主进程中使用，一些只可以在渲染进程中使用（ web 页面），但是也有部分可以在这2种进程中都可使用。

基本规则：GUI模块或者系统底层的模块只可以在主进程中使用。要使用这些模块，你应当很熟悉[主进程vs渲染进程](https://electron.org.cn/doc/tutorial/quick-start.html#main-process)脚本的概念。

主进程脚本看起来像个普通的 Node.js 脚本：

const {app, BrowserWindow} = require('electron')

let win = null

app.on('ready', () => {

win = new BrowserWindow({width: 800, height: 600})

win.loadURL('https://github.com')

})

渲染进程和传统的 web 界面一样，除了它具有使用 node 模块的能力：

<!DOCTYPE html>

<html>

<body>

<script>

const {app} = require('electron').remote

console.log(app.getVersion())

</script>

</body>

</html>

如果想运行应用，参考 [Run your app](https://electron.org.cn/doc/tutorial/quick-start.html#run-your-app)。

解构任务

在 v0.37 版本之后，你可以使用[destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) 来更加简单地使用内置模块。

const {app, BrowserWindow} = require('electron')

let win

app.on('ready', () => {

win = new BrowserWindow()

win.loadURL('https://github.com')

})

如果你需要整个 electron 模块，你可以加载并使用 从 electron 解构访问各个模块。

const electron = require('electron')

const {app, BrowserWindow} = electron

let win

app.on('ready', () => {

win = new BrowserWindow()

win.loadURL('https://github.com')

})

这相当于以下代码：

const electron = require('electron')

const app = electron.app

const BrowserWindow = electron.BrowserWindow

let win

app.on('ready', () => {

win = new BrowserWindow()

win.loadURL('https://github.com')

})

# 进程

process 对象扩展。

Process: [Main](https://electron.org.cn/doc/glossary.html#main-process), [Renderer](https://electron.org.cn/doc/glossary.html#renderer-process)

Electron 的 process 对象是 [Node.js process 对象](https://nodejs.org/api/process.html) 的扩展。 它添加了以下事件、属性和方法：

## 事件

### 事件: 'loaded'

在Electron已经加载了其内部预置脚本和它准备加载网页或者主进程的时候触发。

当node被完全关闭的时候，它可以被预加载脚本使用来添加(原文: removed)与node无关的全局符号来回退到全局范围：

// preload.js

const \_setImmediate = setImmediate

const \_clearImmediate = clearImmediate

process.once('loaded', () => {

global.setImmediate = \_setImmediate

global.clearImmediate = \_clearImmediate

})

## 属性

### process.noAsar

设置它为 true 可以使 asar 文件在node的内置模块中失效。

### process.type

当前 process 的类型，值为"browser" (即主进程) 或 "renderer"。

### process.versions.electron

Electron的版本号。

### process.versions.chrome

Chrome的版本号。

### process.resourcesPath

资源文件夹的路径。

### process.mas

在 Mac App Store 的构建中，该属性为 true, 其他平台的构建均为 undefined。

### process.windowsStore

如果 app 是运行在 Windows Store app (appx) 中，该属性为 true, 其他情况均为 undefined。

### process.defaultApp

当 app 在启动时，被作为参数传递给默认应用程序，在主进程中该属性为 true, 其他情况均为 undefined。

## 方法

process 对象有如下方法：

### process.crash()

使当前进程的主线程崩溃。

### process.hang()

使当前进程的主线程挂起。

### process.setFdLimit(maxDescriptors)macOSLinux

* maxDescriptors Integer

设置文件描述符软限制于 maxDescriptors 或硬限制于OS, 无论它是否低于当前进程。

### process.getProcessMemoryInfo()

返回 Object：

* workingSetSize Integer - 当前固定到实际物理内存的内存量。
* peakWorkingSetSize Integer - 被固定在实际物理内存上的最大内存量。
* privateBytes Integer - 不被其他进程共享的内存量，如JS堆或HTML内容。
* sharedBytes Integer - 进程之间共享的内存量，通常是 Electron 代码本身所消耗的内存。

返回当前进程的内存使用统计信息的对象。请注意，所有数据的单位都是KB。

### process.getSystemMemoryInfo()

返回 Object：

* total Integer - 系统的物理内存总量。
* free Integer - 未被应用程序或磁盘缓存使用的物理内存总量。
* swapTotal Integer - 系统 swap 分区(虚拟内存)总量。 WindowsLinux
* swapFree Integer - 系统剩余可用的 swap 分区(虚拟内存)量。 WindowsLinux

返回系统的内存使用统计信息的对象。请注意，所有数据的单位都是KB。

支持的 Chrome 命令行开关

Electron 支持的命令行开关。

你可以在 [app](https://electron.org.cn/doc/api/app.html) 模块的 [ready](https://electron.org.cn/doc/api/app.html#event-ready) 事件发出之前使用 [app.commandLine.appendSwitch](https://electron.org.cn/doc/api/app.html#appcommandlineappendswitchswitch-value) 来添加它们到你应用的 main 脚本里面：

const {app} = require('electron')

app.commandLine.appendSwitch('remote-debugging-port', '8315')

app.commandLine.appendSwitch('host-rules', 'MAP \* 127.0.0.1')

app.on('ready', () => {

// Your code here

})

--client-certificate=path

设置客户端的证书文件 path .

--ignore-connections-limit=domains

忽略用 , 分隔的 domains 列表的连接限制.

--disable-http-cache

禁止请求 HTTP 时使用磁盘缓存.

--remote-debugging-port=port

在指定的 端口 通过 HTTP 开启远程调试.

--js-flags=flags

指定引擎过渡到 JS 引擎.

在启动Electron时，如果你想在主进程中激活 flags ，它将被转换.

$ electron --js-flags="--harmony\_proxies --harmony\_collections" your-app

--proxy-server=address:port

使用一个特定的代理服务器，它将比系统设置的优先级更高.这个开关只有在使用 HTTP 协议时有效，它包含 HTTPS 和 WebSocket 请求. 值得注意的是，不是所有的代理服务器都支持 HTTPS 和 WebSocket 请求.

--proxy-bypass-list=hosts

让 Electron 使用(原文:bypass) 提供的以 semi-colon 分隔的hosts列表的代理服务器.这个开关只有在使用 --proxy-server 时有效.

例如:

app.commandLine.appendSwitch('proxy-bypass-list', '<local>;\*.google.com;\*foo.com;1.2.3.4:5678')

将会为所有的hosts使用代理服务器，除了本地地址 (localhost, 127.0.0.1 etc.), google.com 子域, 以 foo.com 结尾的hosts，和所有类似 1.2.3.4:5678的.

--proxy-pac-url=url

在指定的 url 上使用 PAC 脚本.

--no-proxy-server

不使用代理服务并且总是使用直接连接.忽略所有的合理代理标志.

--host-rules=rules

一个逗号分隔的 rule 列表来控制主机名如何映射.

例如:

* MAP \* 127.0.0.1 强制所有主机名映射到 127.0.0.1
* MAP \*.google.com proxy 强制所有 google.com 子域 使用 "proxy".
* MAP test.com [::1]:77 强制 "test.com" 使用 IPv6 回环地址. 也强制使用端口 77.
* MAP \* baz, EXCLUDE www.google.com 重新全部映射到 "baz", 除了 "www.google.com".

这些映射适用于终端网络请求 (TCP 连接 和 主机解析 以直接连接的方式, 和 CONNECT 以代理连接, 还有 终端 host 使用 SOCKS 代理连接).

--host-resolver-rules=rules

类似 --host-rules ，但是 rules 只适合主机解析.

--ignore-certificate-errors

忽略与证书相关的错误.

--ppapi-flash-path=path

设置Pepper Flash插件的路径 path .

--ppapi-flash-version=version

设置Pepper Flash插件版本号.

--log-net-log=path

使网络日志事件能够被读写到 path.

--disable-renderer-backgrounding

防止 Chromium 降低隐藏的渲染进程优先级.

这个标志对所有渲染进程全局有效，如果你只想在一个窗口中禁止使用，你可以采用 hack 方法[playing silent audio](https://github.com/atom/atom/pull/9485/files).

--enable-logging

打印 Chromium 信息输出到控制台.

如果在用户应用加载完成之前解析app.commandLine.appendSwitch ，这个开关将实效，但是你可以设置 ELECTRON\_ENABLE\_LOGGING 环境变量来达到相同的效果.

--v=log\_level

设置默认最大活跃 V-logging 标准; 默认为 0.通常 V-logging 标准值为肯定值.

这个开关只有在 --enable-logging 开启时有效.

--vmodule=pattern

赋予每个模块最大的 V-logging levels 来覆盖 --v 给的值.E.g. my\_module=2,foo\*=3 会改变所有源文件 my\_module.\* and foo\*.\* 的代码中的 logging level .

任何包含向前的(forward slash)或者向后的(backward slash)模式将被测试用于阻止整个路径名，并且不仅是E.g模块.\*/foo/bar/\*=2 将会改变所有在 foo/bar 下的源文件代码中的 logging level .

这个开关只有在 --enable-logging 开启时有效.

环境变量

一些 Electron 的行为受到环境变量的控制，因为他们的初始化比命令行和应用代码更早.

POSIX shells 的例子:

$ export ELECTRON\_ENABLE\_LOGGING=true

$ electron

Windows 控制台:

> set ELECTRON\_ENABLE\_LOGGING=true

> electron

ELECTRON\_RUN\_AS\_NODE

类似node.js普通进程启动方式.

ELECTRON\_ENABLE\_LOGGING

打印 Chrome 的内部日志到控制台.

ELECTRON\_LOG\_ASAR\_READS

当 Electron 读取 ASA 文档，把 read offset 和文档路径做日志记录到系统 tmpdir.结果文件将提供给 ASAR 模块来优化文档组织.

ELECTRON\_ENABLE\_STACK\_DUMPING

当 Electron 崩溃的时候，打印堆栈记录到控制台.

如果 crashReporter 已经启动那么这个环境变量实效.

ELECTRON\_DEFAULT\_ERROR\_MODEWindows

当 Electron 崩溃的时候，显示windows的崩溃对话框.

如果 crashReporter 已经启动那么这个环境变量实效.

ELECTRON\_NO\_ATTACH\_CONSOLEWindows

不可使用当前控制台.

ELECTRON\_FORCE\_WINDOW\_MENU\_BARLinux

不可在 Linux 上使用全局菜单栏.

ELECTRON\_HIDE\_INTERNAL\_MODULES

关闭旧的内置模块如 require('ipc') 的通用模块.