请确认你的使用的文档匹配你的 Electron 版本。你可以在页面的 URL 中找到版本号。 如果不是，你可能在使用开发分支版本的文档，其中包括了一些与你的 Electron 版本不兼容的 API。 如果想要访问老版本的文档，你可以通过 GitHub 的[切换分支功能](https://github.com/electron/electron/tree/v1.4.0)，在下拉菜单中选择与你的版本匹配的分支。

常见问题

这里是一些被经常问到的问题，在提 issue 之前请先看一下这里。

* [Electron 常见问题](https://electron.org.cn/doc/faq/electron-faq.html)

向导

* [术语表](https://electron.org.cn/doc/glossary.html)
* [支持平台](https://electron.org.cn/doc/tutorial/supported-platforms.html)
* [安全性](https://electron.org.cn/doc/tutorial/security.html)
* [Electron 版本管理](https://electron.org.cn/doc/tutorial/electron-versioning.html)
* [分发应用](https://electron.org.cn/doc/tutorial/application-distribution.html)
* [提交应用到 Mac App Store](https://electron.org.cn/doc/tutorial/mac-app-store-submission-guide.html)
* [Windows 商店提交指引](https://electron.org.cn/doc/tutorial/windows-store-guide.html)
* [打包应用](https://electron.org.cn/doc/tutorial/application-packaging.html)
* [使用 Node 原生模块](https://electron.org.cn/doc/tutorial/using-native-node-modules.html)
* [主进程调试](https://electron.org.cn/doc/tutorial/debugging-main-process.html)
* [使用 Selenium 和 WebDriver](https://electron.org.cn/doc/tutorial/using-selenium-and-webdriver.html)
* [使用开发人员工具扩展](https://electron.org.cn/doc/tutorial/devtools-extension.html)
* [使用 Pepper Flash 插件](https://electron.org.cn/doc/tutorial/using-pepper-flash-plugin.html)
* [使用 Widevine CDM 插件](https://electron.org.cn/doc/tutorial/using-widevine-cdm-plugin.html)
* [通过自动化持续集成系统（CI）进行测试 (Travis, Jenkins)](https://electron.org.cn/doc/tutorial/testing-on-headless-ci.html)
* [离屏渲染](https://electron.org.cn/doc/tutorial/offscreen-rendering.html)
* [快捷键](https://electron.org.cn/doc/tutorial/keyboard-shortcuts.html)

教程

* [快速入门](https://electron.org.cn/doc/tutorial/quick-start.html)
* [桌面环境集成](https://electron.org.cn/doc/tutorial/desktop-environment-integration.html)
* [在线/离线事件探测](https://electron.org.cn/doc/tutorial/online-offline-events.html)
* [交互式解释器（REPL）](https://electron.org.cn/doc/tutorial/repl.html)

API文档

* [简介](https://electron.org.cn/doc/api/synopsis.html)
* [进程对象](https://electron.org.cn/doc/api/process.html)
* [支持的 Chrome 命令行开关](https://electron.org.cn/doc/api/chrome-command-line-switches.html)
* [环境变量](https://electron.org.cn/doc/api/environment-variables.html)

自定义的 DOM 元素:

* [File 对象](https://electron.org.cn/doc/api/file-object.html)
* [<webview> 标签](https://electron.org.cn/doc/api/webview-tag.html)
* [window.open 函数](https://electron.org.cn/doc/api/window-open.html)

在主进程内可用的模块:

* [app](https://electron.org.cn/doc/api/app.html)
* [autoUpdater](https://electron.org.cn/doc/api/auto-updater.html)
* [BrowserWindow](https://electron.org.cn/doc/api/browser-window.html)
* [contentTracing](https://electron.org.cn/doc/api/content-tracing.html)
* [dialog](https://electron.org.cn/doc/api/dialog.html)
* [globalShortcut](https://electron.org.cn/doc/api/global-shortcut.html)
* [ipcMain](https://electron.org.cn/doc/api/ipc-main.html)
* [Menu](https://electron.org.cn/doc/api/menu.html)
* [MenuItem](https://electron.org.cn/doc/api/menu-item.html)
* [net](https://electron.org.cn/doc/api/net.html)
* [powerMonitor](https://electron.org.cn/doc/api/power-monitor.html)
* [powerSaveBlocker](https://electron.org.cn/doc/api/power-save-blocker.html)
* [protocol](https://electron.org.cn/doc/api/protocol.html)
* [session](https://electron.org.cn/doc/api/session.html)
* [systemPreferences](https://electron.org.cn/doc/api/system-preferences.html)
* [Tray](https://electron.org.cn/doc/api/tray.html)
* [webContents](https://electron.org.cn/doc/api/web-contents.html)

在渲染进程（网页）内可用的模块:

* [desktopCapturer](https://electron.org.cn/doc/api/desktop-capturer.html)
* [ipcRenderer](https://electron.org.cn/doc/api/ipc-renderer.html)
* [remote](https://electron.org.cn/doc/api/remote.html)
* [webFrame](https://electron.org.cn/doc/api/web-frame.html)

在两种进程中都可用的模块:

* [clipboard](https://electron.org.cn/doc/api/clipboard.html)
* [crashReporter](https://electron.org.cn/doc/api/crash-reporter.html)
* [nativeImage](https://electron.org.cn/doc/api/native-image.html)
* [screen](https://electron.org.cn/doc/api/screen.html)
* [shell](https://electron.org.cn/doc/api/shell.html)

开发

* [代码规范](https://electron.org.cn/doc/development/coding-style.html)
* [在 C++ 代码中使用 clang-format 工具](https://electron.org.cn/doc/development/clang-format.html)
* [源码目录结构](https://electron.org.cn/doc/development/source-code-directory-structure.html)
* [与 NW.js（原 node-webkit）在技术上的差异](https://electron.org.cn/doc/development/atom-shell-vs-node-webkit.html)
* [构建系统概览](https://electron.org.cn/doc/development/build-system-overview.html)
* [构建步骤（macOS）](https://electron.org.cn/doc/development/build-instructions-osx.html)
* [构建步骤（Windows）](https://electron.org.cn/doc/development/build-instructions-windows.html)
* [构建步骤（Linux）](https://electron.org.cn/doc/development/build-instructions-linux.html)
* [调试步骤 (macOS)](https://electron.org.cn/doc/development/debugging-instructions-macos.html)
* [调试步骤 (Windows)](https://electron.org.cn/doc/development/debug-instructions-windows.html)
* [在调试中使用 Symbol Server](https://electron.org.cn/doc/development/setting-up-symbol-server.html)
* [文档风格指南](https://electron.org.cn/doc/styleguide.html)
* [升级 Chrome](https://electron.org.cn/doc/development/upgrading-chrome.html)
* [Chromium 开发](https://electron.org.cn/doc/development/chromium-development.html)
* [V8 开发](https://electron.org.cn/doc/development/v8-development.html)

相关npm包集合

* [electron-packager](https://npm.js.cn/package/electron-packager)
* [electron-builder](https://npm.js.cn/package/electron-builder)
* [electron-updater](https://npm.js.cn/package/electron-updater)
* [electron-download](https://npm.js.cn/package/electron-download)
* [electron-apps](https://npm.js.cn/package/electron-apps)
* [electron-to-chromium](https://npm.js.cn/package/electron-to-chromium)
* [awesome-electron](https://npm.js.cn/package/awesome-electron)
* [electron-debug](https://npm.js.cn/package/electron-debug)
* [electron-rebuild](https://npm.js.cn/package/electron-rebuild)

相关站点资源

* [React中文网](https://reactjs.org.cn/)
* [ReactNative中文网](https://react-native.org.cn/)
* [Npm中文网](https://npm.js.cn/)
* [Yarn中文网](https://yarn.org.cn/)
* [GraphQL查询语言](https://graphql.org.cn/)
* [Kotlin编程语言](https://kotlin.hk/)

# 术语表

这篇文档说明了一些经常在 Electron 开发中使用的专业术语。

### ASAR

ASAR 代表了 Atom Shell Archive Format。一个 [asar](https://github.com/electron/asar) 压缩包就是一个简单的 tar 文件-就像将那些有联系的文件格式化至一个单独的文件中。Electron 能够任意读取其中的文件并且不需要解压缩整个文件。

ASAR 格式主要是为了提升 Windows 平台上的性能。TODO

### Brightray

[Brightray](https://github.com/electron/brightray) 是能够简单的将 [libchromiumcontent](https://electron.org.cn/doc/glossary.html#libchromiumcontent) 应用到应用中的一个静态库。它是专门开发给 Electron 使用，但是也能够使用在那些没有基于 Electron 的原生应用来启用 Chromium 的渲染引擎。

Brightray 是 Electron 中的一个低级别的依赖，大部分的 Electron 用户不用关心它。

### DMG

是指在 macOS 上使用的苹果系统的磁盘镜像打包格式。DMG 文件通常被用来分发应用的 "installers"（安装包）。[electron-builder](https://github.com/electron-userland/electron-builder) 支持使用 dmg 来作为编译目标。

### IPC

IPC 代表 Inter-Process Communication。Electron 使用 IPC 来在 [主进程] 和 [渲染进程] 之间传递 JSON 信息。

### libchromiumcontent

一个单独的开源库，包含了 Chromium 的模块以及全部依赖（比如 Blink, [V8](https://electron.org.cn/doc/glossary.html#v8) 等）。

### main process

主进程，通常是值 main.js 文件，是每个 Electron 应用的入口文件。它控制着整个 APP 的生命周期，从打开到关闭。它也管理着原生元素比如菜单，菜单栏，Dock 栏，托盘等。主进程负责创建 APP 的每个渲染进程。而且整个 Node API 都集成在里面。

每个 app 的主进程文件都定义在 package.json 中的 main 属性当中，这也是为什么 electron . 能够知道应该使用哪个文件来启动。

参见： [process](https://electron.org.cn/doc/glossary.html#process), [renderer process](https://electron.org.cn/doc/glossary.html#renderer-process)

### MAS

是指苹果系统上的 Mac App Store 的缩略词。有关于如何提交你的 app 至 MAS ，详见 [Mac App Store Submission Guide](https://electron.org.cn/doc/tutorials/mac-app-store-submission-guide.html) 。

### native modules

原生模块 （在 Node.js 里也叫 [addons](https://nodejs.org/api/addons.html)），是一些使用 C or C++ 编写的能够在 Node.js 中加载或者在 Electron 中使用 require() 方法来加载的模块，它使用起来就如同 Node.js 的模块。它主要用于桥接在 JavaScript 上运行 Node.js 和 C/C++ 的库。

Electron 支持了原生的 Node 模块，但是 Electron 非常可能安装一个不一样的 V8 引擎通过 Node 二进制编码，所以在打包原生模块的时候你需要指定具体的 Electron 本地头文件。

参见： [Using Native Node Modules](https://electron.org.cn/doc/tutorial/using-native-node-modules.html).

## NSIS

Nullsoft Scriptable Install System 是一个微软 Windows 平台上的脚本驱动的安装制作工具。它发布在免费软件许可证书下，是一个被广泛使用的替代商业专利产品类似于 InstallShield。[electron-builder](https://github.com/electron-userland/electron-builder) 支持使用 NSIS 作为编译目标。

### process

一个进程是计算机程序执行中的一个实例。Electron 应用同时使用了 [main](https://electron.org.cn/doc/glossary.html#main-process) (主进程) 和一个或者多个 [renderer](https://electron.org.cn/doc/glossary.html#renderer-process) （渲染进程）来运行多个程序。

在 Node.js 和 Electron 里面，每个运行的进程包含一个 process 对象。这个对象作为一个全局的提供当前进程的相关信息，操作方法。作为一个全局变量，它在应用内能够不用 require() 来随时取到。

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

### renderer process

渲染进程是你的应用内的一个浏览器窗口。与主进程不同的是，它能够同时存在多个而且运行在不一样的进程。而且它们也能够被隐藏。

在通常的浏览器内，网页通常运行在一个沙盒的环境挡住并且不能够使用原生的资源。然而 Electron 的用户在 Node.js 的 API 支持下可以在页面中和操作系统进行一些低级别的交互。

参见： [process](https://electron.org.cn/doc/glossary.html#process), [main process](https://electron.org.cn/doc/glossary.html#main-process)

### Squirrel

Squirrel 是一个开源的框架来让 Electron 的应用能够自动的更新到发布的新的版本。详见 [autoUpdater](https://electron.org.cn/doc/api/auto-updater.html) API 了解如何开始使用 Squirrel。

### userland

"userland" 或者 "userspace" 术语起源于 Unix 社区，当程序运行在操作系统内核之外。最近这个术语被推广在 Node 和 npm 社区用于区分 "Node core" 与发布的包的功能，对于在 npm 上注册的广大 "user（用户）" 们。

就像 Node ，Electron 致力于使用一些少量的设置和 API 来提供所有的必须的支持给开发中的跨平台应用。这个设计理念让 Electron 能够保持灵活而不被过多的规定有关于如何应该被使用。Userland 让用户能够创造和分享一些工具来提供额外的功能在这个能够使用的 "core（核心）"之上。

### V8

V8 是谷歌公司的开源的 JavaScript 引擎。它使用 C++ 编写并使用在谷歌公司开源的的浏览器 Google Chrome 上。V8 能够单独运行或者集成在任何一个 C++ 应用内。

### webview

webview 标签用于集成 'guest（访客）' 内容（比如外部的网页）在你的 Electron 应用内。它们类似于 iframe，但是不同的是每个 webview 运行在独立的进程中。 作为页面它拥有不一样的权限并且所有的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。

# 支持的平台

目前 Electron 支持以下平台：

### macOS

对于 macOS 仅提供64位版本，并且只支持 macOS 10.9 或更高版本。

### Windows

仅支持 Windows 7 或更高版本。

对于 Windows 提供 ia32 (x86) 和 amd64 (x64) 版本。需要注意的是 ARM 版本的 Windows 目前尚不支持。

### Linux

Electron 的 ia32 (i686) 和 x64 (amd64) 预编译版本均是在Ubuntu 12.04 下编译的，arm 版的二进制文件是在 ARM v7（硬浮点 ABI 与 Debian Wheezy 版本的 NEON）下完成的。

预编译版本是否能够正常运行，取决于其中是否包含了编译平台的链接库。所以只有 Ubuntu 12.04 是可以保证能正常运行的，并且以下平台也被证实可以正常运行 Electron 的预编译版本：

* Ubuntu 12.04 或更高版本
* Fedora 21
* Debian 8

# 安全，本地功能和你的责任

作为 web 开发人员，我们通常喜欢网络安全性更强大的浏览器 - 与我们编写的代码相关的风险相对较小。我们的网站在沙箱中获得有限的权限，我们相信我们的用户可以享受由大量工程师团队构建的浏览器，能够快速响应新发现的安全威胁。

当使用 Electron 时，要知道 Electron 不是一个 Web 浏览器很重要。它允许您使用熟悉的 Web 技术构建功能丰富的桌面应用程序，但是您的代码具有更强大的功能。 JavaScript 可以访问文件系统，用户 shell 等。这允许您构建更高质量的本机应用程序，但是内在的安全风险会随着授予您的代码的额外权力而增加。

考虑到这一点，请注意，在 Electron 上处理任何情况下显示来自不受信任的来源的任何内容将带来了严重的安全风险。事实上，最流行的 Electron 应用程序（Atom，Slack，Visual Studio Code 等）主要显示本地内容（或没有 Node 集成的可信安全远程内容） - 如果您的应用程序从在线源执行代码，那么您有责任确保代码不是恶意的。

## 报告安全问题

有关如何正确上报 Electron 漏洞的信息，参阅 [SECURITY.md](https://github.com/electron/electron/tree/master/SECURITY.html)

## Chromium 安全问题和升级

尽管 Electron 努力尽快支持新版本的 Chromium，但开发人员应该意识到，升级是一项严肃的工作 - 涉及手动编辑几十个甚至几百个文件。 考虑到当前的资源和贡献，Electron 通常不会是最新版本的 Chromium，总是落后于一两天或几周。

我们认为，我们当前的更新 Chromium 组件的系统在我们可用的资源和构建在框架之上的大多数应用程序的需求之间取得了适当的平衡。 我们绝对有兴趣听听更多关于在 Electron 上构建事物的人的具体用例。 非常欢迎提出请求并且捐助支持我们的努力。

## 除了以上建议

每当您从远程目标收到代码并在本地执行它时，就会存在安全问题。 举个例子，比如在浏览器窗口内显示的远程网站。 如果攻击者以某种方式设法改变所述内容（通过直接攻击源或者通过在应用和实际目的地之间进行攻击），他们将能够在用户的机器上执行本地代码。

:警告: 在任何情况下都不应该在启用了 Node 集成时加载并执行远程代码. 反而应该只使用本地文件（与应用程序一起打包）来执行 Node 代码。要显示远程内容, 应使用 webview 标签并确保禁用了 nodeIntegration.

#### 检查列表

这并不是万无一失的，但至少，你应该尝试以下内容：

* 只显示安全的内容(https)
* 在显示远程内容的所有渲染器中禁用 Node 集成 (在 webPreferences 中设置 nodeIntegration 为 false)
* 在显示远程内容的所有渲染器中启用上下文隔离 (在 webPreferences 中设置 contextIsolation 为 true)
* 在所有加载远程内容的会话中使用 ses.setPermissionRequestHandler() .
* 不要禁用 webSecurity. 禁用它将禁用同源策略.
* 定义一个 [Content-Security-Policy](http://www.html5rocks.com/en/tutorials/security/content-security-policy/) , 并使用限制规则 (即： script-src 'self')
* 覆盖并禁用 [eval](https://github.com/nylas/N1/blob/0abc5d5defcdb057120d726b271933425b75b415/static/index.js#L6-L8) , 它允许字符串作为代码执行.
* 不要设置 allowRunningInsecureContent 为 true.
* 不要启用 experimentalFeatures 或 experimentalCanvasFeatures 除非你知道你在做什么.
* 不要使用 blinkFeatures 除非你知道你在做什么.
* WebViews: 不要填加 nodeintegration 属性.
* WebViews: 不要使用 disablewebsecurity
* WebViews: 不要使用 allowpopups
* WebViews: 不要使用 insertCSS 或 executeJavaScript 操作远程 CSS/JS.

强调一下，这份列表只是将风险降到最低，并不会完全屏蔽风险。 如果您的目的是展示一个网站，浏览器将是一个更安全的选择。

# Electron 版本管理

如果你是一个经验丰富的Node开发人员，你肯定知道semver - 然而这里给你的依赖管理系统可能只有粗略的指导建议而不是固定的版本号。由于对 Node 和 Chromium 的硬性依赖，Electron 处于一个稍微复杂的境地，而且不遵循semver。因此，您应该始终引用特定版本的Electron。

版本号使用参照以下规则:

* 主要版本: 适用于 Electron API 的突破性变更 - 如果您从 0.37.0 升级到 1.0.0, 您将需要升级您的应用程序。
* 次要版本: 适用于 Chrome 主要版本 和 Node 次要版本升级; 或重大的 Electron 变动 - 如果您从 1.0.0 升级到 1.1.0, 您的应用程序仍然可以正常运行, 但你可能需要解决一些小幅的变动。
* 补丁版本: 适用于新功能的添加和 bug 修复 - 如果您从 1.0.0 升级到 1.0.1, 你的应用程序仍然像之前一样正常运行。

如果你使用 electron 或 electron-prebuilt，我们建议您设置固定的版本号（如 1.1.0 而不是 ^1.1.0），以确保Electron的所有升级都是由您（开发人员）进行的手动操作。

# 应用部署

为了使用 Electron 部署你的应用程序，你需要下载 Electron 的[prebuilt binaries](https://electron.org.cn/release.html)。 接下来，你存放应用程序的文件夹需要叫做 app 并且需要放在 Electron 的 资源文件夹下（在 macOS 中是指 Electron.app/Contents/Resources/，在 Linux 和 Windows 中是指 resources/） 就像这样：

在 macOS 中:

electron/Electron.app/Contents/Resources/app/

├── package.json

├── main.js

└── index.html

在 Windows 和 Linux 中:

electron/resources/app

├── package.json

├── main.js

└── index.html

然后运行 Electron.app （或者 Linux 中的 electron，Windows 中的 electron.exe）, 接着 Electron 就会以你的应用程序的方式启动。electron 文件夹将被部署并可以分发给最终的使用者。

## 将你的应用程序打包成一个文件

除了通过拷贝所有的资源文件来分发你的应用程序之外，你可以通过打包你的应用程序为一个 [asar](https://github.com/electron/asar) 库文件以避免暴露你的源代码。

为了使用一个 asar 库文件代替 app 文件夹，你需要修改这个库文件的名字为 app.asar ， 然后将其放到 Electron 的资源文件夹下，然后 Electron 就会试图读取这个库文件并从中启动。 如下所示：

在 macOS 中:

electron/Electron.app/Contents/Resources/

└── app.asar

在 Windows 和 Linux 中:

electron/resources/

└── app.asar

更多的细节请见 [Application packaging](https://electron.org.cn/doc/tutorial/application-packaging.html).

## 更换名称与下载二进制文件

在使用 Electron 打包你的应用程序之后，你可能需要在分发给用户之前修改打包的名字。

### Windows

你可以将 electron.exe 改成任意你喜欢的名字，然后可以使用像 [rcedit](https://github.com/atom/rcedit) 编辑它的 icon 和其他信息。

### macOS

你可以将 Electron.app 改成任意你喜欢的名字，然后你也需要修改这些文件中的 CFBundleDisplayName， CFBundleIdentifier以及 CFBundleName 字段。 这些文件如下：

* Electron.app/Contents/Info.plist
* Electron.app/Contents/Frameworks/Electron Helper.app/Contents/Info.plist

你也可以重命名帮助应用程序以避免在应用程序监视器中显示 Electron Helper， 但是请确保你已经修改了帮助应用的可执行文件的名字。

一个改过名字的应用程序的构造可能是这样的：

MyApp.app/Contents

├── Info.plist

├── MacOS/

│   └── MyApp

└── Frameworks/

├── MyApp Helper EH.app

| ├── Info.plist

| └── MacOS/

|    └── MyApp Helper EH

├── MyApp Helper NP.app

| ├── Info.plist

| └── MacOS/

|    └── MyApp Helper NP

└── MyApp Helper.app

├── Info.plist

└── MacOS/

   └── MyApp Helper

### Linux

你可以将 electron 改成任意你喜欢的名字。

## 打包工具

除了手动打包应用程序，你还可以选择使用第三方打包工具来做这项工作：

* [electron-builder](https://github.com/electron-userland/electron-builder)
* [electron-packager](https://github.com/electron-userland/electron-packager)

## 通过重编译源代码来更换名称

通过修改产品名称并重编译源代码来更换 Electron 的名称也是可行的。 你需要修改 atom.gyp 文件并彻底重编译一次。

### grunt打包脚本

手动检查 Electron 代码并重编译是很复杂晦涩的，因此有一个Grunt任务可以自动的处理 这些内容 [grunt-build-atom-shell](https://github.com/paulcbetts/grunt-build-atom-shell).

这个任务会自动的处理编辑 .gyp 文件，从源代码进行编译，然后重编译你的应用程序的本地 Node 模块以匹配这个新的可执行文件的名称。

### 创建一个自定义 Electron 分支

如果只是为了构建你的 app，你不需要创建一个自定义的 Electron 分支， 即使是“生产级”的应用程序。 可以使用工具，如 electron-packager 或 electron-builder 来 “重塑” 你的 Electron app。

当你想直接进入 Electron 来修改 C++ 代码，你需要 fork Electron ，不能使用 upstreamed 或者 rejected 官方版本。 作为 Electron 的维护者，我们非常希望你的方案能够工作，并尽最大的努力让你的修改进入 Electron 的官方版本，创建自定义分支对你来说更简单，最后， 我们感谢你的帮助。

#### 通过 surf-build 创建一个自定义版本

1. 通过 npm 下载 [Surf](https://github.com/surf-build/surf): npm install -g surf-build@latest
2. 创建一个新的 S3 bucket 并按照以下结构创建文件夹：

`

* + atom-shell/
    - symbols/
    - dist/ `

1. 设置以下环境变量：

* ELECTRON\_GITHUB\_TOKEN - 一个在 GitHub 创建版本的 token
* ELECTRON\_S3\_ACCESS\_KEY, ELECTRON\_S3\_BUCKET, ELECTRON\_S3\_SECRET\_KEY - 你将上传 Node.js 的 headers 以及符号的位置
* ELECTRON\_RELEASE - 设置为 true，上传部分将运行，不设置      和 surf-build 只是做 CI-type 的检查，适合每一个运行      拉取请求。
* CI - 设置为 true ，否则无效
* GITHUB\_TOKEN - 设置为与 ELECTRON\_GITHUB\_TOKEN 相同
* SURF\_TEMP - 在 Windows 下设置为 C:\Temp 来防止路径太长的问题
* TARGET\_ARCH - 设置为 ia32 或 x64

1. 在 script/upload.py，你\_必须\_为你的分支(MYORG/electron)设置 ELECTRON\_REPO， 尤其如果你本身是一个 Electron 贡献者。
2. surf-build -r https://github.com/MYORG/electron -s YOUR\_COMMIT -n 'surf-PLATFORM-ARCH'
3. 需要很长的时间来等待构建完成

# Windows商店指南

在 Windows 10 中, 一些不错的旧 win32 程序迎来了一个新朋友: 通用Windows平台(UWP)。 新的 .appx 格式不仅启用了许多新的强大的 API，如 Cortana 或推送通知，而且通过Windows 应用商店，也同时简化了安装和更新。

Microsoft 开发了一个工具，将 Electron 应用程序[编译为 .appx 软件包](https://github.com/catalystcode/electron-windows-store)，使开发人员能够使用新应用程序模型中的一些好东西。 本指南解释了如何使用它 - 以及 Electron AppX 包的功能和限制。

## 背景和要求

Windows 10 的 "周年更新" 能够运行 win32 .exe 程序并且它们的虚拟化文件系统和注册表跟随一起启动。 两者都是通过在 Windows 容器中运行应用程序和安装器编译后创建的，允许 Windows 在安装过程中正确识别操作系统进行了哪些修改。 将可执行文件和虚拟文件系统与虚拟注册表配对, 允许 Windows 启用一键安装和卸载。

此外，exe 在 appx 模型内启动 - 这意味着它可以使用通用 Windows 平台可用的许多 API。 为了获得更多的功能，Electron 应用程序可以与一个看不见的 UWP 后台任务配合使用，它与 exe 一起启动，作为后台运行任务的接收器，接收推送通知或与其他 UWP 应用程序通信 。

要编译任何现有的 Electron 应用程序，请确保满足以下要求:

* Windows 10及周年更新 (2016年8月2日发布的)
* Windows 10 SDK, [这里下载](https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk)
* 最新的 Node 4 (运行 node -v 来确认)

然后, 安装 electron-windows-store CLI:

npm install -g electron-windows-store

## 步骤 1: 打包你的 Electron 应用程序

打包应用程序使用 [electron-packager](https://github.com/electron-userland/electron-packager) (或类似工具). 确保在最终的应用程序中删除不需要的 node\_modules, 因为这些你不需要模块只会额外增加你的应用程序的大小.

结构输出应该看起来大致像这样:

├── Ghost.exe

├── LICENSE

├── content\_resources\_200\_percent.pak

├── content\_shell.pak

├── d3dcompiler\_47.dll

├── ffmpeg.dll

├── icudtl.dat

├── libEGL.dll

├── libGLESv2.dll

├── locales

│   ├── am.pak

│   ├── ar.pak

│   ├── [...]

├── natives\_blob.bin

├── node.dll

├── resources

│   ├── app

│   └── atom.asar

├── snapshot\_blob.bin

├── squirrel.exe

└── ui\_resources\_200\_percent.pak

## 步骤 2: 运行 electron-windows-store

从提权的 PowerShell（用管理员身份运行它）中,以所需的参数运行 electron-windows-store，传递输入和输出目录，应用程序的名称和版本，以及确认node\_modules应该是扁平的。

electron-windows-store `

--input-directory C:\myelectronapp `

--output-directory C:\output\myelectronapp `

--flatten true `

--package-version 1.0.0.0 `

--package-name myelectronapp

一旦执行，工具就开始工作：它接受您的 Electron 应用程序作为输入，展平 node\_modules。 然后，它将应用程序归档为 app.zip。 使用安装程序和 Windows 容器，该工具创建一个“扩展的” AppX 包 - 包括 Windows 应用程序清单 （AppXManifest.xml）以及虚拟文件系统和输出文件夹中的虚拟注册表。

当创建扩展的 AppX 文件后，该工具使用 Windows App Packager（MakeAppx.exe）将磁盘上的这些文件创建为单文件 AppX 包。 最后，该工具可用于在计算机上创建可信证书，以签署新的 AppX 包。 使用签名的 AppX 软件包，CLI也可以自动在您的计算机上安装软件包。

## 步骤 3: 使用 AppX 包

为了运行您的软件包，您的用户将需要将 Windows 10 安装“周年纪念更新” - 有关如何更新Windows的详细信息可以在[这里](https://blogs.windows.com/windowsexperience/2016/08/02/how-to-get-the-windows-10-anniversary-update)找到

与传统的UWP应用程序不同，打包应用程序目前需要进行手动验证过程，您可以在[这里](https://developer.microsoft.com/en-us/windows/projects/campaigns/desktop-bridge)申请. 在此期间，所有用户都能够通过双击安装包来安装您的程序，所以如果您只是寻找一个更简单的安装方法，可能不需要提交到商店。

在受管理的环境中（通常是企业）, Add-AppxPackage PowerShell Cmdlet 可用于以[自动方式安装](https://technet.microsoft.com/en-us/library/hh856048.aspx)它。

另一个重要的限制是编译的 AppX 包仍然包含一个 win32 可执行文件，因此不会在 Xbox，HoloLens 或 Phones 中运行。

## 可选: 使用 BackgroundTask 添加 UWP 功能

您可以将 Electron 应用程序与不可见的 UWP 后台任务配对，以充分利用 Windows 10 功能，如推送通知，Cortana 集成或活动磁贴。

如何使用 Electron 应用程序通过后台任务发送 Toast 通知和活动磁贴,请查看[微软提供的案例](https://github.com/felixrieseberg/electron-uwp-background).

## 可选: 使用容器虚拟化进行转换

要生成 AppX 包，electron-windows-store CLI 使用的模板应该适用于大多数 Electron 应用程序。 但是，如果您使用自定义安装程序，或者您遇到生成的包的任何问题，您可以尝试使用 Windows 容器编译创建包 - 在该模式下，CLI 将在空 Windows 容器中安装和运行应用程序，以确定应用程序正在对操作系统进行哪些修改。

在运行 CLI 之前，您必须设置 “Windows Desktop App Converter” 。 这将需要几分钟，但不要担心 - 你只需要这样做一次。 从这里下载 [Desktop App Converter](https://www.microsoft.com/en-us/download/details.aspx?id=51691)

您将得到两个文件： DesktopAppConverter.zip 和 BaseImage-14316.wim.

1. 解压 DesktopAppConverter.zip. 打开提权的 PowerShell (用"以管理员权限运行"打开, 确保您的系统执行策略允许我们通过调用 Set-ExecutionPolicy bypass 来运行我们想要运行的一切).
2. 然后, 通过调用 .\DesktopAppConverter.ps1 -Setup -BaseImage .\BaseImage-14316.wim, 运行 Desktop App Converter 安装，并传递 Windows 基本映像的位置 (下载的 BaseImage-14316.wim).
3. 如果运行以上命令提示您重新启动，请重新启动计算机，并在成功重新启动后再次运行上述命令。

当安装成功后，您可以继续编译你的 Electron 应用程序。

# 应用打包

为舒缓 Windows 下路径名过长的问题[issues](https://github.com/joyent/node/issues/6960)， 也略对 require 加速以及简单隐匿你的源代码，你可以通过极小的源代码改动将你的应用打包成 [asar](https://github.com/atom/asar)。

## 生成 asar 包

[asar](https://github.com/atom/asar) 是一种将多个文件合并成一个文件的类 tar 风格的归档格式。 Electron 可以无需解压，即从其中读取任意文件内容。

参照如下步骤将你的应用打包成 asar：

### 1. 安装 asar

$ npm install -g asar

### 2. 用 asar pack 打包

$ asar pack your-app app.asar

## 使用 asar 包

在 Electron 中有两类 APIs：Node.js 提供的 Node API 和 Chromium 提供的 Web API。 这两种 API 都支持从 asar 包中读取文件。

### Node API

由于 Electron 中打了特别补丁， Node API 中如 fs.readFile 或者 require 之类 的方法可以将 asar 视之为虚拟文件夹，读取 asar 里面的文件就和从真实的文件系统中读取一样。

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

$ asar list /path/to/example.asar

/app.js

/file.txt

/dir/module.js

/static/index.html

/static/main.css

/static/jquery.min.js

从 asar 包读取一个文件：

const fs = require('fs')

fs.readFileSync('/path/to/example.asar/file.txt')

列出 asar 包中根目录下的所有文件：

const fs = require('fs')

fs.readdirSync('/path/to/example.asar')

使用 asar 包中的一个模块：

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

你也可以使用 BrowserWindow 来显示一个 asar 包里的 web 页面：

const BrowserWindow = require('electron').BrowserWindow

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

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

### Web API

在 Web 页面里，用 file: 协议可以获取 asar 包中文件。和 Node API 一样，视 asar 包如虚拟文件夹。

例如，用 $.get 获取文件:

<script>

var $ = require('./jquery.min.js');

$.get('file:///path/to/example.asar/file.txt', function(data) {

console.log(data);

});

</script>

### 像“文件”那样处理 asar 包

有些场景，如：核查 asar 包的校验和，我们需要像读取“文件”那样读取 asar 包的内容(而不是当成虚拟文件夹)。 你可以使用内置的 original-fs （提供和 fs 一样的 API）模块来读取 asar 包的真实信息。

var originalFs = require('original-fs')

originalFs.readFileSync('/path/to/example.asar')

## Node API 缺陷

尽管我们已经尽了最大努力使得 asar 包在 Node API 下的应用尽可能的趋向于真实的目录结构，但仍有一些底层 Node API 我们无法保证其正常工作。

### asar 包是只读的

asar 包中的内容不可更改，所以 Node APIs 里那些可以用来修改文件的方法在对待 asar 包时都无法正常工作。

### Working Directory 在 asar 包中无效

尽管 asar 包是虚拟文件夹，但其实并没有真实的目录架构对应在文件系统里，所以你不可能将 working Directory 设置成 asar包里的一个文件夹。将 asar 中的文件夹以 cwd 形式作为参数传入一些 API 中也会报错。

### API 中的额外“开箱”

大部分 fs API 可以无需解压即从 asar 包中读取文件或者文件的信息，但是在处理一些依赖真实文件路径的底层 系统方法时，Electron 会将所需文件解压到临时目录下，然后将临时目录下的真实文件路径传给底层系统方法使其正 常工作。 对于这类API，耗费会略多一些。

以下是一些需要额外解压的 API：

* child\_process.execFile
* child\_process.execFileSync
* fs.open
* fs.openSync
* process.dlopen - requirenative模块时用到

### fs.stat 获取的 stat 信息不可靠

对 asar 包中的文件取 fs.stat，返回的 Stats 对象不是精确值，因为这些文件不是真实存在于文件系 统里。所以除了文件大小和文件类型以外，你不应该依赖 Stats 对象的值。

### 执行 asar 包中的程序

Node 中有一些可以执行程序的 API，如 child\_process.exec，child\_process.spawn 和 child\_process.execFile 等， 但只有 execFile 可以执行 asar 包中的程序。

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，我们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。

## 打包时排除文件

如上所述，一些 Node API 会在调用时将文件解压到文件系统中，除了效率问题外，也有可能引起杀毒软件的注意！

为解决这个问题，你可以在生成 asar 包时使用 --unpack 选项来排除一些文件，使其不打包到 asar 包中， 下面是如何排除一些用作共享用途的 native 模块的方法：

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

经过上述命令后，除了生成的 app.asar 包以外，还有一个包含了排除文件的 app.asar.unpacked 文件夹， 你需要将这个文件夹一起拷贝，提供给用户。

# 使用原生模块

Electron 同样也支持原生模块，但由于和官方的 Node 相比使用了不同的 V8 引擎，如果你想编译原生模块，则需要手动设置 Electron 的 headers 的位置。

## 如何安装原生模块

如下三种方法教你安装原生模块：

### 通过 npm 安装

只要设置一些系统环境变量，你就可以通过 npm 直接安装原生模块。

为 Electron 安装所有依赖项的一个例子：

# Electron 的版本。

export npm\_config\_target=1.2.3

# Electron 的系统架构, 值为 ia32 或者 x64。

export npm\_config\_arch=x64

export npm\_config\_target\_arch=x64

# 下载 Electron 的 headers。

export npm\_config\_disturl=https://atom.io/download/electron

# 告诉 node-pre-gyp 是为 Electron 构建。

export npm\_config\_runtime=electron

# 告诉 node-pre-gyp 从源代码构建模块。

export npm\_config\_build\_from\_source=true

# 下载所有依赖，并缓存到 ~/.electron-gyp。

HOME=~/.electron-gyp npm install

### 下载并重新编译模块

你可以选择安装其他 Node 项目的模块，通过 [electron-rebuild](https://github.com/paulcbetts/electron-rebuild) 包 重新编译原生模块，它可以识别当前 Electron 版本，帮你自动完成了下载 headers、编译原生模块等步骤。

一个下载 electron-rebuild 并重新编译的例子：

npm install --save-dev electron-rebuild

# 每次运行"npm install"时，也运行这条命令

./node\_modules/.bin/electron-rebuild

# 在windows下如果上述命令遇到了问题，尝试这个：

.\node\_modules\.bin\electron-rebuild.cmd

### 手动编译

如果你是一个原生模块的开发人员，想在 Electron 中进行测试， 你可能要手动编译 Electron 模块。你可以 使用 node-gyp 直接编译：

cd /path-to-module/

HOME=~/.electron-gyp node-gyp rebuild --target=1.2.3 --arch=x64 --dist-url=https://atom.io/download/electron

HOME=~/.electron-gyp 设置去哪找开发时的 headers。 --target=0.29.1 设置了 Electron 的版本。 --dist-url=... 设置了 Electron 的 headers 的下载地址。 --arch=x64 设置了该模块为适配64位操作系统而编译。

## 故障排除

如果你安装了一个原生模块并发现它不能工作，你需要检查 以下事项：

* 模块的对应的操作系统和 Electron 对应的操作系统是否匹配(ia32 或 x64)。
* 如果升级了 Electron，你通常需要重新编译这些模块。
* 当有疑问时，请先执行 electron-rebuild。

## prebuild 依赖模块

[prebuild](https://github.com/mafintosh/prebuild) 为多个版本的 Node 和 Electron 提供了一种简单发布预编译二进制原生模块的方法。

如果为 Electron 提供二进制原生模块，请确保删除 --build-from-source 和 npm\_config\_build\_from\_source 环境变量 来充分利用预编译的二进制文件。

## node-pre-gyp 依赖模块

[node-pre-gyp 工具](https://github.com/mapbox/node-pre-gyp) 提供一种部署原生 Node 预编译二进制模块的方法， 许多流行的模块都是使用它。

通常这些模块在 Electron 中工作良好，但有时当 Electron 使用 比 Node 新的 V8 版本时，会有 ABI 改变，可能 发生错误。因此，一般来说，建议始终从源代码编译原生模块。

如果你通过 npm 的方式安装模块，默认情况下这就完成了， 如果没有，你需要传入 --build-from-source 给 npm, 或者设置 npm\_config\_build\_from\_source 环境变量。

# 主进程调试

浏览器窗口的开发工具仅能调试渲染器的进程脚本（比如 web 页面）。为了提供一个可以调试主进程 的方法，Electron 提供了 --debug 和 --debug-brk 开关。

## 命令行开关

使用如下的命令行开关来调试 Electron 的主进程：

### --debug=[port]

当这个开关用于 Electron 时，它将会监听 V8 引擎中有关 port 的调试器协议信息。 默认的 port 是 5858。

### --debug-brk=[port]

就像 --debug 一样，但是会在第一行暂停脚本运行。

## 外部调试器

你将需要使用一个支持 V8 调试器的调试协议， 下面的指南将会帮助你开始：

* [使用 VSCode 进行主进程调试](https://electron.org.cn/doc/tutorial/debugging-main-process-vscode.html)
* [使用 node-inspector 进行主进程调试](https://electron.org.cn/doc/tutorial/debugging-main-process-node-inspector.html)

# 使用 Selenium 和 WebDriver

引自[ChromeDriver - WebDriver for Chrome](https://sites.google.com/a/chromium.org/chromedriver/):

WebDriver 是一款开源的支持多浏览器的自动化测试工具。它提供了操作网页、用户输入、JavaScript 执行等能力。ChromeDriver 是一个实现了 WebDriver 与 Chromium 联接协议的独立服务。它也是由开发了 Chromium 和 WebDriver 的团队开发的。

## 通过 Spectron 配置

[Spectron](https://electron.atom.io/spectron) 是 Electron 官方支持的 ChromeDriver 测试框架。 它是建立在 [WebdriverIO](http://webdriver.io/) 的顶层，并且 帮助你在测试中访问 Electron API 和绑定 ChromeDriver。

$ npm install --save-dev spectron

// 一个简单的测试验证一个带标题的可见的窗口

var Application = require('spectron').Application

var assert = require('assert')

var app = new Application({

path: '/Applications/MyApp.app/Contents/MacOS/MyApp'

})

app.start().then(function () {

// 检查浏览器窗口是否可见

return app.browserWindow.isVisible()

}).then(function (isVisible) {

// 验证浏览器窗口是否可见

assert.equal(isVisible, true)

}).then(function () {

// 获得浏览器窗口的标题

return app.client.getTitle()

}).then(function (title) {

// 验证浏览器窗口的标题

assert.equal(title, 'My App')

}).catch(function (error) {

// 记录任何错误

console.error('Test failed', error.message)

}).then(function () {

// 停止应用程序

return app.stop()

})

## 通过 WebDriverJs 配置

[WebDriverJs](https://code.google.com/p/selenium/wiki/WebDriverJs) 是一个可以配合 WebDriver 做测试的 node 模块，我们会用它来做个演示。

### 1. 启动 ChromeDriver

首先，你要下载 chromedriver，然后运行以下命令：

$ ./chromedriver

Starting ChromeDriver (v2.10.291558) on port 9515

Only local connections are allowed.

记住 9515 这个端口号，我们后面会用到

### 2. 安装 WebDriverJS

$ npm install selenium-webdriver

### 3. 联接到 ChromeDriver

在 Electron 下使用 selenium-webdriver 和其平时的用法并没有大的差异，只是你需要手动设置连接 ChromeDriver，以及 Electron 的路径：

const webdriver = require('selenium-webdriver')

var driver = new webdriver.Builder()

// "9515" 是ChromeDriver使用的端口

.usingServer('http://localhost:9515')

.withCapabilities({

chromeOptions: {

// 这里设置Electron的路径

binary: '/Path-to-Your-App.app/Contents/MacOS/Atom'

}

})

.forBrowser('electron')

.build()

driver.get('http://www.google.com')

driver.findElement(webdriver.By.name('q')).sendKeys('webdriver')

driver.findElement(webdriver.By.name('btnG')).click()

driver.wait(function () {

return driver.getTitle().then(function (title) {

return title === 'webdriver - Google Search'

})

}, 1000)

driver.quit()

## 通过 WebdriverIO 配置

[WebdriverIO](http://webdriver.io/) 也是一个配合 WebDriver 用来测试的 node 模块

### 1. 启动 ChromeDriver

首先，下载 chromedriver，然后运行以下命令：

$ chromedriver --url-base=wd/hub --port=9515

Starting ChromeDriver (v2.10.291558) on port 9515

Only local connections are allowed.

记住 9515 端口，后面会用到

### 2. 安装 WebdriverIO

$ npm install webdriverio

### 3. 连接到 ChromeDriver

const webdriverio = require('webdriverio')

const options = {

host: 'localhost', // 使用localhost作为ChromeDriver服务器

port: 9515, // "9515"是ChromeDriver使用的端口

desiredCapabilities: {

browserName: 'chrome',

chromeOptions: {

binary: '/Path-to-Your-App/electron', // Electron的路径

args: [] // 可选参数，类似：'app=' + /path/to/your/app/

}

}

}

let client = webdriverio.remote(options)

client

.init()

.url('http://google.com')

.setValue('#q', 'webdriverio')

.click('#btnG')

.getTitle().then(function (title) {

console.log('Title was: ' + title)

})

.end()

## 工作流程

无需重新编译 Electron，只要把 app 的源码放到 [Electron的资源目录](https://github.com/electron/electron/blob/master/docs/tutorial/application-distribution.html) 里就可直接开始测试了。

当然，你也可以在运行 Electron 时传入参数指定你 app 的所在文件夹。这步可以免去你拷贝－粘贴你的 app 到 Electron 的资源目录。

# DevTools扩展

为了使调试更容易，Electron 原生支持 [Chrome DevTools Extension](https://developer.chrome.com/extensions/devtools)。

## 如何加载 DevTools 扩展

对于大多数DevTools的扩展，你可以直接下载源码，然后通过 BrowserWindow.addDevToolsExtension API 加载它们。Electron会记住已经加载了哪些扩展，所以你不需要每次创建一个新window时都调用 BrowserWindow.addDevToolsExtension API。

**注：React DevTools目前不能直接工作，详情留意**[**https://github.com/electron/electron/issues/915**](https://github.com/electron/electron/issues/915)

例如，要用[React DevTools Extension](https://github.com/facebook/react-devtools)，你得先下载他的源码：

$ cd /some-directory

$ git clone --recursive https://github.com/facebook/react-devtools.git

参考 [react-devtools/shells/chrome/Readme.md](https://github.com/facebook/react-devtools/blob/master/shells/chrome/Readme.html) 来编译这个扩展源码。

然后你就可以在任意页面的 DevTools 里加载 React DevTools 了，通过控制台输入如下命令加载扩展：

const BrowserWindow = require('electron').remote.BrowserWindow

BrowserWindow.addDevToolsExtension('/some-directory/react-devtools/shells/chrome')

要卸载扩展，可以调用 BrowserWindow.removeDevToolsExtension API (扩展名作为参数传入)，该扩展在下次打开DevTools时就不会加载了：

BrowserWindow.removeDevToolsExtension('React Developer Tools')

## DevTools 扩展的格式

理论上，Electron 可以加载所有为 chrome 浏览器编写的 DevTools 扩展，但它们必须存放在文件夹里。那些以 crx 形式发布的扩展是不能被加载的，除非你把它们解压到一个文件夹里。

## 后台运行(background pages)

Electron 目前并不支持 chrome 扩展里的后台运行(background pages)功能，所以那些依赖此特性的 DevTools 扩展在 Electron 里可能无法正常工作。

## chrome.\* APIs

有些 chrome 扩展使用了 chrome.\*APIs，而且这些扩展在 Electron 中需要额外实现一些代码才能使用，所以并不是所有的这类扩展都已经在 Electron 中实现完毕了。

考虑到并非所有的 chrome.\*APIs 都实现完毕，如果 DevTools 正在使用除了 chrome.devtools.\* 之外的其它 APIs，这个扩展很可能无法正常工作。你可以通过报告这个扩展的异常信息，这样做方便我们对该扩展的支持。

# 使用 Pepper Flash 插件

Electron 现在支持 Pepper Flash 插件。要在 Electron 里面使用 Pepper Flash 插件，你需 要手动设置 Pepper Flash 的路径和在你的应用里启用 Pepper Flash。

## 保留一份 Flash 插件的副本

在 macOS 和 Linux 上，你可以在 Chrome 浏览器的 chrome://plugins 页面上找到 Pepper Flash 的插件信息。插件的路径和版本会对 Election 对其的支持有帮助。你也可以把插件 复制到另一个路径以保留一份副本。

## 添加插件在 Electron 里的开关

你可以直接在命令行中用 --ppapi-flash-path 和 --ppapi-flash-version 或者 在 app 的准备事件前调用 app.commandLine.appendSwitch 这个 method。同时， 添加 BrowserWindow 的插件开关。 例如：

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

const path = require('path')

// 指定 flash 路径，假定它与 main.js 放在同一目录中。

let pluginName

switch (process.platform) {

case 'win32':

pluginName = 'pepflashplayer.dll'

break

case 'darwin':

pluginName = 'PepperFlashPlayer.plugin'

break

case 'linux':

pluginName = 'libpepflashplayer.so'

break

}

app.commandLine.appendSwitch('ppapi-flash-path', path.join(\_\_dirname, pluginName))

// 可选：指定 flash 的版本，例如 v17.0.0.169

app.commandLine.appendSwitch('ppapi-flash-version', '17.0.0.169')

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

let win = new BrowserWindow({

width: 800,

height: 600,

webPreferences: {

plugins: true

}

})

win.loadURL(`file://${\_\_dirname}/index.html`)

// 还有别的

})

您也可以尝试加载系统安装的 Pepper Flash 插件，而不是装运 插件，其路径可以通过调用 app.getPath('pepperFlashSystemPlugin') 获取。

## 使用 <webview> 标签启用插件

在 <webview> 标签里添加 plugins 属性。

<webview src="http://www.adobe.com/software/flash/about/" plugins></webview>

## 错误解决

您可以通过在控制台打印 navigator.plugins 来检查 Pepper Flash 插件是否加载 （虽然你不知道插件的路径是正确的)。

Pepper Flash 插件的操作系统必须和 Electron 的操作系统匹配。在 Windows 中， 一个常见的错误是对64位版本的 Electron 使用 32bit 版本的 Flash 插件。

在 Windows 中，传递给 --ppapi-flash-path 的路径必须使用 \ 作为路径 分隔符，使用 POSIX-style 的路径将无法工作。

对于一些操作，例如使用 RTMP 的流媒体，有必要向播放器的 .swf 文件授予更多的权限。 实现这一点的一种方式是使用 [nw-flash-trust](https://github.com/szwacz/nw-flash-trust)。

# 使用 Widevine CDM 插件

在 Electron，你可以使用 Widevine CDM 插件装载 Chrome 浏览器。

## 获取插件

Electron 没有为 Widevine CDM 插件配置许可 reasons，为了获得它，首先需要安装官方的 Chrome 浏览器，这匹配了体系架构和 Electron 构建使用的 Chrome 版本。

**注意：** Chrome 浏览器的主要版本必须和 Electron 使用的版本一样，否则插件不会有效，虽然 navigator.plugins 会显示你已经安装了它。

### Windows & macOS

在 Chrome 浏览器中打开 chrome://components/ ，找到 WidevineCdm 并且确定它更新到最新版本，然后你可以从 APP\_DATA/Google/Chrome/WidevineCDM/VERSION/\_platform\_specific/PLATFORM\_ARCH/ 路径找到所有的插件二进制文件。

APP\_DATA 是系统存放数据的地方，在 Windows 上它是 %LOCALAPPDATA%，在 macOS 上它是 ~/Library/Application Support. VERSION 是 Widevine CDM 插件的版本字符串，类似 1.4.8.866. PLATFORM 是 mac 或 win. ARCH 是 x86 或 x64。

在 Windows，必要的二进制文件是 widevinecdm.dll and widevinecdmadapter.dll，在 macOS，它们是 libwidevinecdm.dylib和 widevinecdmadapter.plugin。你可以将它们复制到任何你喜欢的地方，但是它们必须要放在一起。

### Linux

在 Linux，Chrome 浏览器将插件的二进制文件装载在一起，你可以在 /opt/google/chrome 下找到，文件名是 libwidevinecdm.so 和 libwidevinecdmadapter.so。

## 使用插件

在获得了插件文件后，你可以使用 --widevine-cdm-path 命令行开关来将 widevinecdmadapter 的路径传递给 Electron ，插件版本使用 --widevine-cdm-version 开关.

**注意:** 虽然只有 widevinecdmadapter 的二进制文件传递给了 Electron，widevinecdm 二进制文件应当放在它的旁边。

必须在 app 模块的 ready 事件触发之前使用命令行开关，并且 page 使用的插件必须激活。

示例代码：

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

// 你必须通过 `widevinecdmadapter` 文件名，它是

// \* `widevinecdmadapter.plugin` on macOS，

// \* `libwidevinecdmadapter.so` on Linux，

// \* `widevinecdmadapter.dll` on Windows。

app.commandLine.appendSwitch('widevine-cdm-path', '/path/to/widevinecdmadapter.plugin')

// 插件版本可以从 Chrome 浏览器的 `chrome://plugins` 页面获得。

app.commandLine.appendSwitch('widevine-cdm-version', '1.4.8.866')

let win = null

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

win = new BrowserWindow({

webPreferences: {

// 这个 `plugins` 必须启用。

plugins: true

}

})

win.show()

})

## 验证插件

为了验证插件是否工作，你可以使用下面的方法：

* 打开开发者工具查看 navigator.plugins 是否包含了 Widevine CDM 插件。
* 打开 https://shaka-player-demo.appspot.com/ 加载一个使用 Widevine 的 manifest。
* 打开 [http://www.dash-player.com/demo/drm-test-area/，检查是否界面输出](http://www.dash-player.com/demo/drm-test-area/%EF%BC%8C%E6%A3%80%E6%9F%A5%E6%98%AF%E5%90%A6%E7%95%8C%E9%9D%A2%E8%BE%93%E5%87%BA)bitdash uses Widevine in your browser，然后播放 video。

# Headless CI Systems 测试 (Travis CI, Jenkins)

Electron 基于 Chromium，所以需要一个显示驱动使其运转。如果 Chromium 无法找到一个显示驱动， ELectron 会启动失败，因此无论你如何去运行它，Electron 不会执行你的任何测试。在 Travis，Circle， Jenkins 或者类似的系统上测试基于Electron的应用时，需要进行一些配置。本质上，我们需要使用一个 虚拟的显示驱动。

## 配置虚拟显示服务器

首先安装 [Xvfb](https://en.wikipedia.org/wiki/Xvfb)。 这是一个虚拟的帧缓冲，实现了X11显示服务协议，所有的图形操作都在内存中表现，而不需要显示在 任何屏幕输出设备上。这正是我们所需要的。

然后创建一个虚拟的xvfb屏幕并且导出一个指向他的名为 DISPLAY 的环境变量。Electron 中的 Chromium 会自动的去寻找 $DISPLAY，所以你的应用不需要再去进行配置。这一步可以通过 Paul Betts 的 [xvfb-maybe](https://github.com/paulcbetts/xvfb-maybe) 实现自动化：如果系统需要，在xvfb-maybe前加上你的测试命令 然后这个小工具会自动的设置 xvfb。在 Windows 或者 macOS 系统下，它不会执行任何东西。

## 在 Windows 或者 macOS，这只是调用 electron-mocha

## 在 Linux， 如果我们在 headless 环境，这将是等同于

## xvfb-run electron-mocha ./test/\*.js

xvfb-maybe electron-mocha ./test/\*.js

### Travis CI

在 Travis 上, 你的 .travis.yml 应该和下面的代码相似:

addons:

apt:

packages:

- xvfb

install:

- export DISPLAY=':99.0'

- Xvfb :99 -screen 0 1024x768x24 > /dev/null 2>&1 &

### Jenkins

Jenkins下, 有一个可用的 [Xvfb插件](https://wiki.jenkins-ci.org/display/JENKINS/Xvfb+Plugin)。

### Circle CI

Circle CI 是非常棒的而且有xvfb，$DISPLAY 也 [已经搭建，所以不需要再进行设置](https://circleci.com/docs/environment#browsers)。

### AppVeyor

AppVeyor 运行于 Windows 上，支持 Selenium，Chromium，Electron 以及一些类似的工具，开箱即用，无需配置。

# 离屏渲染

离线渲染允许您在位图中获取浏览器窗口的内容，因此可以在任何地方渲染，例如在3D场景中的纹理。Electron中的离屏渲染使用与 [Chromium Embedded Framework](https://bitbucket.org/chromiumembedded/cef) 项目类似的方法。

可以使用两种渲染模式，并且只有脏区通过 'paint' 事件才能更高效。渲染可以停止、继续，并且可以设置帧速率。 指定的帧速率是上限值，当网页上没有发生任何事件时，不会生成任何帧。 最大帧速率是60，因为再高没有好处，而且损失性能。

**注意:** 屏幕窗口始终创建为 [Frameless Window](https://electron.org.cn/doc/api/frameless-window.html).

## 两种渲染模式

### GPU加速

GPU加速渲染意味着使用GPU用于合成。因为帧必须从需要更多性能的GPU中复制，因此这种模式比另一个模式慢得多。这种模式的优点是支持WebGL和3D CSS动画。

### 软件输出设备

此模式使用软件输出设备在CPU中渲染，因此帧生成速度更快，因此此模式优先于GPU加速模式。

要启用此模式，必须通过调用 [app.disableHardwareAcceleration()](https://electron.org.cn/doc/api/app.html#appdisablehardwareacceleration) API 来禁用GPU加速。

## 使用

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

app.disableHardwareAcceleration()

let win

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

win = new BrowserWindow({

webPreferences: {

offscreen: true

}

})

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

win.webContents.on('paint', (event, dirty, image) => {

// updateBitmap(dirty, image.getBitmap())

})

win.webContents.setFrameRate(30)

})

# 键盘快捷键

配置本地和全局键盘快捷键

## 本地快捷键

您可以使用 [Menu](https://electron.org.cn/doc/api/menu.html) 模块来配置快捷键，只有在 app 处于焦点状态时才可以触发快捷键。为此，在创建 [MenuItem](https://electron.org.cn/doc/api/menu-item.html) 时必须指定一个 [accelerator](https://electron.org.cn/doc/api/accelerator.html) 属性。

const {Menu, MenuItem} = require('electron')

const menu = new Menu()

menu.append(new MenuItem({

label: 'Print',

accelerator: 'CmdOrCtrl+P',

click: () => { console.log('打印时间') }

}))

还可以很容易的根据用户的操作系统配置不同的组合键。

{

accelerator: process.platform === 'darwin' ? 'Alt+Cmd+I' : 'Ctrl+Shift+I'

}

## 全局快捷键

当应用程式不处于焦点状态时，你可以使用 [globalShortcut](https://electron.org.cn/doc/api/global-shortcut.html) 模块来检测键盘事件，

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

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

globalShortcut.register('CommandOrControl+X', () => {

console.log('CommandOrControl+X is pressed')

})

})

## BrowserWindow 中的快捷键

如果你想处理 [BrowserWindow](https://electron.org.cn/doc/api/browser-window.html) 中的键盘快捷键，你可以监听渲染进程中 window 对象的 keyup 和 keydown 事件。

window.addEventListener('keyup', doSomething, true)

注意第三个参数 true，这意味着监听器总是在其他监听器之前接收按键，所以它们不能对它们调用 stopPropagation()。

如果您不想手动进行快捷键解析，可以使用一些库来进行高级的按键检测。例如 [mousetrap](https://github.com/ccampbell/mousetrap).

Mousetrap.bind('4', () => { console.log('4') })

Mousetrap.bind('?', () => { console.log('show shortcuts!') })

Mousetrap.bind('esc', () => { console.log('escape') }, 'keyup')

// 组合

Mousetrap.bind('command+shift+k', () => { console.log('command shift k') })

// 将多个组合映射到相同的回调

Mousetrap.bind(['command+k', 'ctrl+k'], () => {

console.log('command k or control k')

// 返回 false 以防止默认行为，并阻止事件冒泡

return false

})

// gmail 风格序列

Mousetrap.bind('g i', () => { console.log('go to inbox') })

Mousetrap.bind('\* a', () => { console.log('select all') })

// konami 代码

Mousetrap.bind('up up down down left right left right b a enter', () => {

console.log('konami code')

})