# Electron 教程简介

由 425389019 创建，最后一次修改 2016-10-04

所有的[Node.js's built-in modules](http://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）。

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，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渲染进程脚本的概念。

主进程脚本看起来像个普通的nodejs脚本

**const** electron = require('electron');

**const** app = electron.app;

**const** BrowserWindow = electron.BrowserWindow;

**var** window = null;

app.on('ready', **function**() {

window = **new** BrowserWindow({width: 800, height: 600});

window.loadURL('https://www.w3cschool.cn');

});

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** remote = require('electron').remote;

console.log(remote);

console.log(remote.app.getVersion());

</**script**>

</**body**>

</**html**>

如果想运行应用，参考 Run your app 。

## 解构任务

如果你使用的是CoffeeScript或Babel，你可以使用[destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)来让使用内置模块更简单:

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

然而如果你使用的是普通的JavaScript，你就需要等到Chrome支持ES6了。

## 使用内置模块时禁用旧样式

在版本v0.35.0之前，所有的内置模块都需要按造 require('module-name') 形式来使用，虽然它有很多[弊端](https://github.com/electron/electron/issues/387)，我们仍然在老的应用中友好的支持它。

为了完整的禁用旧样式，你可以设置环境变量 ELECTRON\_HIDE\_INTERNAL\_MODULES :

process.env.ELECTRON\_HIDE\_INTERNAL\_MODULES = 'true'

或者调用 hideInternalModules API:

require('electron').hideInternalModules()

**Electron 快速入门**

由 425389019 创建，最后一次修改 2016-10-04

**快速入门**

**简介**

Electron 可以让你使用纯 JavaScript 调用丰富的原生 APIs（me：借助于node.js的api） 来创造桌面应用。你可以把它看作是专注于桌面应用而不是 web 服务器的，io.js 的一个变体。

这不意味着 Electron 是绑定了 GUI 库的 JavaScript。相反，Electron 使用 web 页面作为它的 GUI(me:那它由什么来渲染这些web页面呢？Chromium)，所以你能把它看作成一个被 JavaScript 控制的，精简版的 Chromium 浏览器。

**主进程 main process**

在 Electron 里，运行 package.json 里 main 脚本的进程被称为**主进程(在package.json中定义main的路径)**。在主进程运行的脚本可以以创建 web 页面的形式展示 GUI。

**渲染进程**

由于 Electron 使用 Chromium 来展示页面，所以 Chromium 的多进程结构也被充分利用。每个 Electron 的页面都在运行着自己的进程（me：仅仅是渲染进程），这样的进程我们称之为**渲染进程**。

在一般浏览器中，网页通常会在沙盒环境下运行，并且不允许访问原生资源。然而，Electron 用户（me:也不允许在页面中调用操作系统的api，而是通过与main进程通信来间接调用）拥有在网页中调用 io.js 的 APIs 的能力，可以与底层操作系统直接交互。

**主进程与渲染进程的区别**

主进程使用 BrowserWindow 实例创建网页。每个 BrowserWindow 实例都在自己的渲染进程里运行着一个网页。当一个 BrowserWindow 实例被销毁后（me：可以这样销毁myBrowserWindow = null），相应的渲染进程也会被终止。

主进程管理所有页面和与之对应的渲染进程。每个渲染进程都是相互独立的，并且只关心他们自己的网页。

由于在网页里管理原生 GUI 资源是非常危险而且容易造成资源泄露，所以在网页面调用 GUI 相关的 APIs 是不被允许的。如果你想在网页里使用 GUI 操作，其对应的渲染进程必须与主进程进行通讯(me：通过ipc通信)，请求主进程进行相关的 GUI 操作。

在 Electron，我们提供用于在主进程与渲染进程之间通讯的 [ipc](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/api/ipc-main-process.md) 模块。并且也有一个远程进程调用风格的通讯模块 [remote](https://www.w3cschool.cn/electronmanual/electronmanual-remote.html)。

**打造你第一个 Electron 应用**

大体上，一个 Electron 应用的目录结构如下：

your-app/

├── **package**.json

├── main.js

└── index.html

package.json的格式和 Node 的完全一致，并且那个被 main 字段声明的脚本文件是你的应用的启动脚本，它运行在主进程上。你应用里的 package.json 看起来应该像：

{

"name" : "your-app",

"version" : "0.1.0",

"main" : "main.js"

}

**注意**：如果 main 字段没有在 package.json 声明，Electron会优先加载 index.js。

main.js 应该用于创建窗口和处理系统时间，一个典型的例子如下：

//app是控制应用生命周期的模块

//创建原生浏览器窗口的模块

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

// 保持一个对于 window 对象的全局引用，不然，当 JavaScript 被 GC，

// window 会被自动地关闭

**var** mainWindow = null;

// 当所有窗口被关闭了，退出。

app.on('window-all-closed', **function**() {

// 在 OS X 上，通常用户在明确地按下 Cmd + Q 之前

// 应用会保持活动状态

**if** (process.platform != 'darwin') {

app.quit();

}

});

// 当 Electron 完成了初始化就会触发ready事件

// 这里注册了一个ready事件，从而使得这个方法就被调用

//可以在这个handler中创建窗口

app.on('ready', **function**() {

// 创建浏览器窗口。每个浏览器是一个BrowserWindow实例，该实例有很多的属性，包括宽度、高度、背景色等

mainWindow = **new** BrowserWindow({width: 800, height: 600});

//窗口可以调用loadURL方法加载本地文件或者远程的网页

//加载本地文件需要使用file协议开头

// 加载应用的 index.html

mainWindow.loadURL('file://' + \_\_dirname + '/index.html');

//调用openDevTools()方法可以打开开发者调试工具

mainWindow.openDevTools();

// 当 mainWindow控制的窗口被关闭，该窗口的closed事件会被发出

mainWindow.on('closed', **function**() {

// 取消引用 window 对象，如果你的应用支持多窗口的话，

// 通常会把多个 window 对象存放在一个数组里面，

// 但这次不是。

mainWindow = null;

});

});

最后，你想展示的 index.html ：

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Hello World!</**title**>

</**head**>

<**body**>

<**h1**>Hello World!</**h1**>

We are using io.js <**script**>document.write(process.version)</**script**>

and Electron <**script**>document.write(process.versions['electron'])</**script**>.

</**body**>

</**html**>

**运行你的应用**

一旦你创建了最初的 main.js， index.html 和 package.json 这几个文件，你可能会想尝试在本地运行并测试，看看是不是和期望的那样正常运行。

**electron-prebuild**

如果你已经用 npm 全局安装了 electron-prebuilt，你只需要按照如下方式直接运行你的应用：

electron .

如果你是局部安装，那运行：

./node\_modules/.bin/electron .

**手工下载 Electron 二进制文件**

如果你手工下载了 Electron 的二进制文件，你也可以直接使用其中的二进制文件直接运行你的应用。

**Windows**

$ .\electron\electron.exe your-app\

**Linux**

$ ./electron/electron your-app/

**OS X**

$ ./Electron.app/Contents/MacOS/Electron your-app/

Electron.app 里面是 Electron 发布包，你可以在[这里](https://github.com/electron/electron/releases)下载到。

**以发行版本运行**

在你完成了你的应用后，你可以按照[应用部署](https://www.w3cschool.cn/electronmanual/i8mc1qkl.html)指导发布一个版本，并且以已经打包好的形式运行应用。

**Electron 桌面环境集成**

由 425389019 创建，最后一次修改 2016-10-04

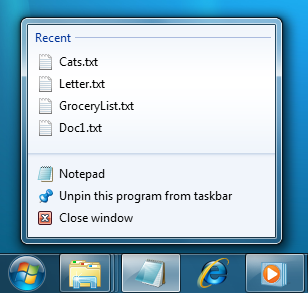
不同的操作系统在各自的桌面应用上提供了不同的特性。例如，在 windows 上应用曾经打开的文件会出现在任务栏的跳转列表，在 Mac 上，应用可以把自定义菜单放在鱼眼菜单上。

本章将会说明怎样使用 Electron APIs 把你的应用和桌面环境集成到一块。

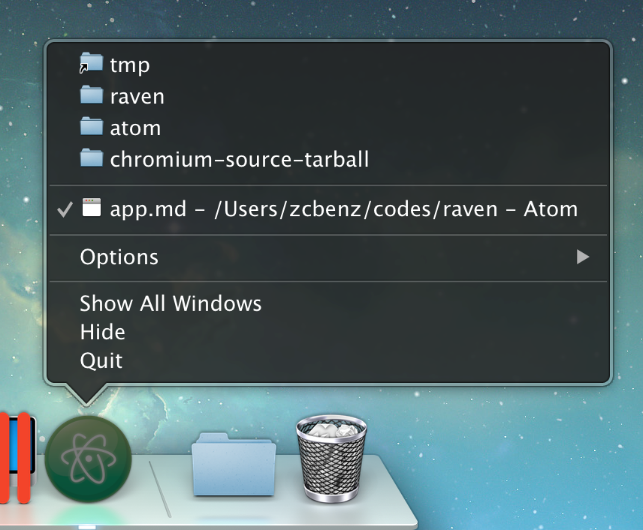
**最近文档 (Windows & OS X)**

Windows 和 OS X 提供获取最近文档列表的便捷方式，那就是打开跳转列表或者鱼眼菜单。

跳转列表：



鱼眼菜单：



为了增加一个文件到最近文件列表，你可以使用 app.addRecentDocument API:

**var** app = require('app');

app.addRecentDocument('/Users/USERNAME/Desktop/work.type');

或者你也可以使用 [app.clearRecentDocuments](https://www.w3cschool.cn/electronmanual/electronmanual-electronapp.html) API 来清空最近文件列表。

app.clearRecentDocuments();

**Windows 需注意**

为了这个特性在 Windows 上表现正常，你的应用需要被注册成为一种文件类型的句柄，否则，在你注册之前，文件不会出现在跳转列表。你可以在 [Application Registration](https://msdn.microsoft.com/en-us/library/windows/desktop/ee872121%28v=vs.85%29.aspx) 里找到任何关于注册事宜的说明。

**OS X 需注意**

当一个文件被最近文件列表请求时，app 模块里的 open-file 事件将会被发出。

**自定义的鱼眼菜单(OS X)**

OS X 可以让开发者定制自己的菜单，通常会包含一些常用特性的快捷方式。

**菜单中的终端**



使用 app.dock.setMenu API 来设置你的菜单，这仅在 OS X 上可行：

**var** app = require('app');

**var** Menu = require('menu');

**var** dockMenu = Menu.buildFromTemplate([

{ label: 'New Window', click: **function**() { console.log('New Window'); } },

{ label: 'New Window with Settings', submenu: [

{ label: 'Basic' },

{ label: 'Pro'}

]},

{ label: 'New Command...'}

]);

app.dock.setMenu(dockMenu);

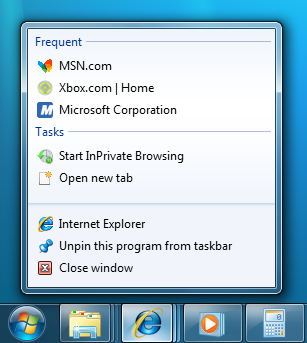
**用户任务(Windows)**

在 Windows，你可以特别定义跳转列表的 Tasks 目录的行为，引用 MSDN 的文档：

Applications define tasks based on both the program's features and the key things a user is expected to do with them. Tasks should be context-free, in that the application does not need to be running for them to work. They should also be the statistically most common actions that a normal user would perform in an application, such as compose an email message or open the calendar in a mail program, create a new document in a word processor, launch an application in a certain mode, or launch one of its subcommands. An application should not clutter the menu with advanced features that standard users won't need or one-time actions such as registration. Do not use tasks for promotional items such as upgrades or special offers.

It is strongly recommended that the task list be static. It should remain the same regardless of the state or status of the application. While it is possible to vary the list dynamically, you should consider that this could confuse the user who does not expect that portion of the destination list to change.

**IE 的任务**



不同于 OS X 的鱼眼菜单，Windows 上的用户任务表现得更像一个快捷方式，比如当用户点击一个任务，一个程序将会被传入特定的参数并且运行。

你可以使用 [app.setUserTasks](https://www.w3cschool.cn/electronmanual/electronmanual-electronapp.html) API 来设置你的应用中的用户任务：

**var** app = require('app');

app.setUserTasks([

{

program: process.execPath,

arguments: '--new-window',

iconPath: process.execPath,

iconIndex: 0,

title: 'New Window',

description: 'Create a new window'

}

]);

调用 app.setUserTasks 并传入空数组就可以清除你的任务列表：

app.setUserTasks([]);

当你的应用关闭时，用户任务会仍然会出现，在你的应用被卸载前，任务指定的图标和程序的路径必须是存在的。

**缩略图工具栏**

在 Windows，你可以在任务栏上添加一个按钮来当作应用的缩略图工具栏。它将提供用户一种用户访问常用窗口的方式，并且不需要恢复或者激活窗口。

在 MSDN，它被如是说：

This toolbar is simply the familiar standard toolbar common control. It has a maximum of seven buttons. Each button's ID, image, tooltip, and state are defined in a structure, which is then passed to the taskbar. The application can show, enable, disable, or hide buttons from the thumbnail toolbar as required by its current state.

For example, Windows Media Player might offer standard media transport controls such as play, pause, mute, and stop.

**Windows Media Player 的缩略图工具栏**



你可以使用 [BrowserWindow.setThumbarButtons](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html) 来设置你的应用的缩略图工具栏。

**var** BrowserWindow = require('browser-window');

**var** path = require('path');

**var** win = **new** BrowserWindow({

width: 800,

height: 600

});

win.setThumbarButtons([

{

tooltip: "button1",

icon: path.join(\_\_dirname, 'button1.png'),

click: **function**() { console.log("button2 clicked"); }

},

{

tooltip: "button2",

icon: path.join(\_\_dirname, 'button2.png'),

flags:['enabled', 'dismissonclick'],

click: **function**() { console.log("button2 clicked."); }

}

]);

调用 BrowserWindow.setThumbarButtons 并传入空数组即可清空缩略图工具栏：

win.setThumbarButtons([]);

**Unity launcher 快捷方式(Linux)**

在 Unity,你可以通过改变 .desktop 文件来增加自定义运行器的快捷方式，详情看 [Adding shortcuts to a launcher](https://help.ubuntu.com/community/UnityLaunchersAndDesktopFiles#Adding_shortcuts_to_a_launcher)。

**Audacious 运行器的快捷方式：**

**任务栏的进度条(Windows & Unity)**

在 Windows，进度条可以出现在一个任务栏按钮之上。这可以提供进度信息给用户而不需要用户切换应用窗口。

Unity DE 也具有同样的特性，在运行器上显示进度条。

**在任务栏上的进度条：**



**在 Unity 运行器上的进度条**



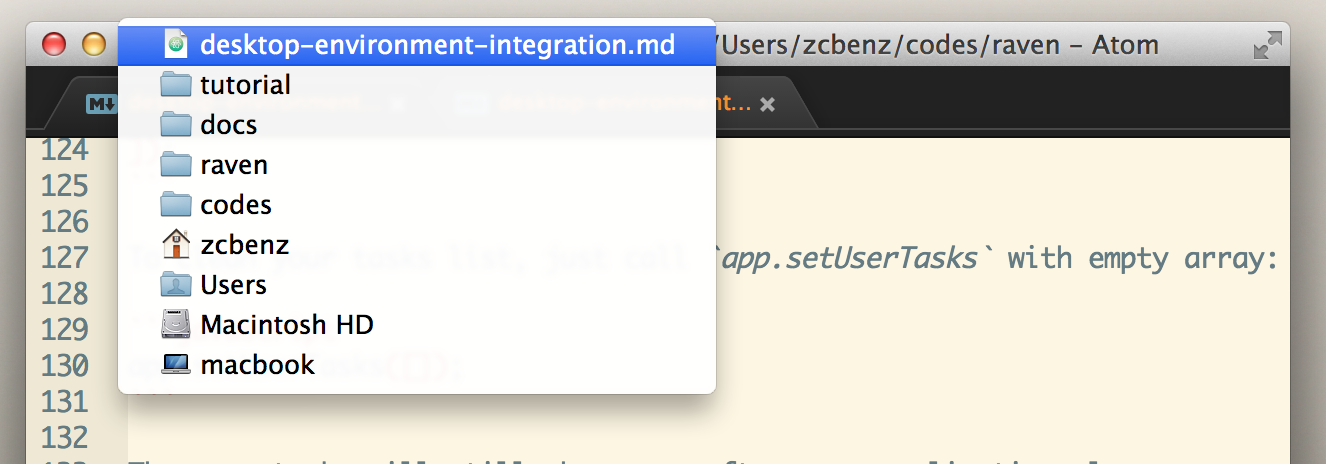
给一个窗口设置进度条，你可以调用 [BrowserWindow.setProgressBar](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html) API：

**var** window = **new** BrowserWindow({...});

window.setProgressBar(0.5);

在 OS X，一个窗口可以设置它展示的文件，文件的图标可以出现在标题栏，当用户 Command-Click 或者 Control-Click 标题栏，文件路径弹窗将会出现。

**展示文件弹窗菜单：**



你可以调用 [BrowserWindow.setRepresentedFilename](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html) 和 [BrowserWindow.setDocumentEdited](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html) APIs：

**var** window = **new** BrowserWindow({...});

window.setRepresentedFilename('/etc/passwd');

window.setDocumentEdited(true);

**Electron 在线/离线事件探测**

由 425389019 创建，最后一次修改 2016-10-04

使用标准 HTML5 APIs 可以实现在线和离线事件的探测，就像以下例子：

*main.js*

**var** app = require('app');

**var** BrowserWindow = require('browser-window');

**var** onlineStatusWindow;

app.on('ready', **function**() {

onlineStatusWindow = **new** BrowserWindow({ width: 0, height: 0, show: false });

onlineStatusWindow.loadURL('file://' + \_\_dirname + '/online-status.html');

});

*online-status.html*

<!DOCTYPE html>

<**html**>

<**body**>

<**script**>

**var** alertOnlineStatus = **function**() {

window.alert(navigator.onLine ? 'online' : 'offline');

};

window.addEventListener('online', alertOnlineStatus);

window.addEventListener('offline', alertOnlineStatus);

alertOnlineStatus();

</**script**>

</**body**>

</**html**>

也会有人想要在主进程也有回应这些事件的实例。然后主进程没有 navigator 对象（me：navigator对象是属于浏览器页面的）因此不能直接探测在线还是离线。使用 Electron 的进程间通讯工具（me:即使用ipc），事件就可以在主进程被使用，就像下面的例子：

*main.js*

**var** app = require('app');

**var** ipc = require('ipc');

**var** BrowserWindow = require('browser-window');

**var** onlineStatusWindow;

app.on('ready', **function**() {

onlineStatusWindow = **new** BrowserWindow({ width: 0, height: 0, show: false });

onlineStatusWindow.loadURL('file://' + \_\_dirname + '/online-status.html');

});

ipc.on('online-status-changed', **function**(event, status) {

console.log(status);

});

*online-status.html*

<!DOCTYPE html>

<**html**>

<**body**>

<**script**>

**var** ipc = require('ipc');

**var** updateOnlineStatus = **function**() {

ipc.send('online-status-changed', navigator.onLine ? 'online' : 'offline');

};

window.addEventListener('online', updateOnlineStatus);

window.addEventListener('offline', updateOnlineStatus);

updateOnlineStatus();

</**script**>

</**body**>

</**html**>

注意:如果电脑启动了虚拟机的网卡驱动,可能会出现离线探测不准确.

# Electron 进程

由 425389019 创建，最后一次修改 2016-10-04

Electron 中的 process 对象 与 upstream node 中的有以下的不同点:

* process.type String - 进程类型, 可以是 browser (i.e. main process)或 renderer.
* process.versions['electron'] String - Electron的版本.
* process.versions['chrome'] String - Chromium的版本.
* process.resourcesPath String - JavaScript源代码路径.
* process.mas Boolean - 在Mac App Store 创建, 它的值为 true, 在其它的地方值为 undefined.

## 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主进程或渲染进程的时候触发.

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

**var** \_setImmediate = setImmediate;

**var** \_clearImmediate = clearImmediate;

process.once('loaded', **function**() {

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.

## 方法

process 对象有如下方法:

### process.hang()

使当前进程的主线程挂起.

### process.setFdLimit(maxDescriptors) OS X Linux

* maxDescriptors Integer

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

# Electron 支持的 Chrome 命令行开关

由 425389019 创建，最后一次修改 2016-10-04

这页列出了Chrome浏览器和Electron支持的命令行开关. 你也可以在[app](https://www.w3cschool.cn/electronmanual/electronmanual-electronapp.html)模块的ready事件发出之前使用[app.commandLine.appendSwitch](https://www.w3cschool.cn/electronmanual/electronmanual-electronapp.html) 来添加它们到你应用的main脚本里面:

**const** app = require('electron').app;

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

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

app.on('ready', **function**() {

// 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.

## --ssl-version-fallback-min=version

设置最简化的 SSL/TLS 版本号 ("tls1", "tls1.1" or "tls1.2")，TLS 可接受回退.

## --cipher-suite-blacklist=cipher\_suites

指定逗号分隔的 SSL 密码套件 列表实效.

## --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.

**Electron 环境变量**

由 425389019 创建，最后一次修改 2016-10-04

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 ASAR 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\_MODE *Windows***

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\_CONSOLE *Windows***

不可使用当前控制台.

**ELECTRON\_FORCE\_WINDOW\_MENU\_BAR *Linux***

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.

**Electron 支持的平台**

由 425389019 创建，最后一次修改 2016-10-03

以下的平台是 Electron 目前支持的：

**OS X**

对于 OS X 系统仅有64位的二进制文档，支持的最低版本是 OS X 10.8。

**Windows**

仅支持 Windows 7 及其以后的版本，之前的版本中是不能工作的。

对于 Windows 提供 x86 和 amd64 (x64) 版本的二进制文件。需要注意的是ARM 版本的 Windows 目前尚不支持.

**Linux**

预编译的 ia32(i686) 和 x64(amd64) 版本 Electron 二进制文件都是在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。

预编译二进制文件是否能够运行，取决于其中是否包括了编译平台链接的库，所以只有 Ubuntu 12.04可以保证正常工作，但是以下的平台也被证实可以运行 Electron 的预编译版本：

* Ubuntu 12.04 及更新
* Fedora 21
* Debian 8

# Electron 应用部署

由 425389019 创建，最后一次修改 2016-10-03

为了使用 Electron 部署你的应用程序，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（me：可以通过拷贝app到这目录下）（在 OS X 中是指 Electron.app/Contents/Resources/，在 Linux 和 Windows 中是指 resources/）就像这样：

在 OS X 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/atom/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（me：并把其他的源文件删除，如果存在的话），然后 Electron 就会试图读取这个库文件并从中启动。如下所示：

在 OS X 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://www.w3cschool.cn/electronmanual/cexo1qkn.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。

### OS X

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。

## 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。

# Electron 向Mac App Store 应用提交向导

由 425389019 创建，最后一次修改 2016-10-03

自从 v0.34.0, Electron 就允许提交应用包到 Mac App Store(MAS) . 这个向导提供的信息有 : 如何提交应用和 MAS 构建的限制.

**注意:** 从 v0.36.0，当应用成为沙箱之后，会有一个 bug 阻止 GPU 进程开启 , 所以在这个 bug 修复之前，建议使用 v0.35.x .更多查看 [issue #3871](https://github.com/electron/electron/issues/3871) .

**注意:** 提交应用到 Mac App Store 需要参加 [Apple DeveloperProgram](https://developer.apple.com/support/compare-memberships/) , 这需要花钱.

## 如何提交

下面步骤介绍了一个简单的提交应用到商店方法.然而，这些步骤不能保证你的应用被 Apple 接受；你仍然需要阅读 Apple 的 [Submitting Your App](https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html) 关于如何满足 Mac App Store 要求的向导.

### 获得证书

为了提交应用到商店，首先需要从 Apple 获得一个证书.可以遵循 [existing guides](https://github.com/nwjs/nw.js/wiki/Mac-App-Store-%28MAS%29-Submission-Guideline#first-steps).

### App 签名

获得证书之后，你可以使用 [Application Distribution](https://www.w3cschool.cn/electronmanual/i8mc1qkl.html) 打包你的应用, 然后前往提交你的应用.这个步骤基本上和其他程序一样，但是这 key 一个个的标识 Electron 的每个依赖.

首先，你需要准备2个授权文件 .

child.plist:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<**plist** **version**="1.0">

<**dict**>

<**key**>com.apple.security.app-sandbox</**key**>

<**true**/>

<**key**>com.apple.security.inherit</**key**>

<**true**/>

</**dict**>

</**plist**>

parent.plist:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<**plist** **version**="1.0">

<**dict**>

<**key**>com.apple.security.app-sandbox</**key**>

<**true**/>

</**dict**>

</**plist**>

然后使用下面的脚本标识你的应用 :

#!/bin/bash

# Name of your app.

APP="YourApp"

# The path of you app to sign.

APP\_PATH="/path/to/YouApp.app"

# The path to the location you want to put the signed package.

RESULT\_PATH="~/Desktop/$APP.pkg"

# The name of certificates you requested.

APP\_KEY="3rd Party Mac Developer Application: Company Name (APPIDENTITY)"

INSTALLER\_KEY="3rd Party Mac Developer Installer: Company Name (APPIDENTITY)"

FRAMEWORKS\_PATH="$APP\_PATH/Contents/Frameworks"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/Electron Framework.framework/Versions/A"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/$APP Helper.app/"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/$APP Helper EH.app/"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/$APP Helper NP.app/"

**if** [ -d "$FRAMEWORKS\_PATH/Squirrel.framework/Versions/A" ]; **then**

# Signing a non-MAS build.

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/Mantle.framework/Versions/A"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/ReactiveCocoa.framework/Versions/A"

codesign --deep -fs "$APP\_KEY" --entitlements child.plist "$FRAMEWORKS\_PATH/Squirrel.framework/Versions/A"

**fi**

codesign -fs "$APP\_KEY" --entitlements parent.plist "$APP\_PATH"

productbuild --component "$APP\_PATH" /Applications --sign "$INSTALLER\_KEY" "$RESULT\_PATH"

如果你是 OS X 下的应用沙箱使用新手，应当仔细阅读 Apple 的 [Enabling App Sandbox](https://developer.apple.com/library/ios/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/EnablingAppSandbox.html) 来有一点基础,然后向授权文件添加你的应用需要的许可 keys .

### 上传你的应用并检查提交

在签名应用之后，可以使用应用 Loader 来上传到 iTunes 链接处理 , 确保在上传之前你已经 [created a record](https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/CreatingiTunesConnectRecord.html). 然后你能 [submit your app for review](https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/SubmittingTheApp.html).

## MAS构建限制

为了让你的应用沙箱满足所有条件，在 MAS 构建的时候，下面的模块被禁用了 :

* crashReporter
* autoUpdater

并且下面的行为也改变了:

* 一些机子的视频采集功能无效.
* 某些特征不可访问.
* Apps 不可识别 DNS 改变.

也由于应用沙箱的使用方法，应用可以访问的资源被严格限制了 ; 阅读更多信息 [App Sandboxing](https://developer.apple.com/app-sandboxing/) .

## Electron 使用的加密算法

取决于你所在地方的国家和地区 , Mac App Store 或许需要记录你应用的加密算法 , 甚至要求你提交一个 U.S 加密注册(ERN) 许可的复印件.

Electron 使用下列加密算法:

* AES - [NIST SP 800-38A](http://dx.doi.org/10.6028/NIST.SP.800-38A), [NIST SP 800-38D](http://dx.doi.org/10.6028/NIST.SP.800-38D), [RFC 3394](http://www.ietf.org/rfc/rfc3394.txt)
* HMAC - [FIPS 198-1](http://doi.org/10.6028/NIST.FIPS.198-1)
* ECDSA - ANS X9.62–2005
* ECDH - ANS X9.63–2001
* HKDF - [NIST SP 800-56C](http://dx.doi.org/10.6028/NIST.SP.800-56C)
* PBKDF2 - [RFC 2898](https://tools.ietf.org/html/rfc2898)
* RSA - [RFC 3447](http://www.ietf.org/rfc/rfc3447)
* SHA - [FIPS 180-4](http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf)
* Blowfish - <https://www.schneier.com/cryptography/blowfish/>
* CAST - [RFC 2144](https://tools.ietf.org/html/rfc2144), [RFC 2612](https://tools.ietf.org/html/rfc2612)
* DES - [FIPS 46-3](http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf)
* DH - [RFC 2631](https://tools.ietf.org/html/rfc2631)
* DSA - [ANSI X9.30](http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.30-1%3A1997)
* EC - [SEC 1](http://www.secg.org/sec1-v2.pdf)
* IDEA - "On the Design and Security of Block Ciphers" book by X. Lai
* MD2 - [RFC 1319](http://tools.ietf.org/html/rfc1319)
* MD4 - [RFC 6150](https://tools.ietf.org/html/rfc6150)
* MD5 - [RFC 1321](https://tools.ietf.org/html/rfc1321)
* RC2 - [RFC 2268](https://tools.ietf.org/html/rfc2268)
* RC4 - [RFC 4345](https://tools.ietf.org/html/rfc4345)
* RC5 - <http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf>
* RIPEMD - [ISO/IEC 10118-3](http://webstore.ansi.org/RecordDetail.aspx?sku=ISO%2FIEC%2010118-3:2004)

如何获取 ERN 许可, 可看这篇文章: [How to legallysubmit an app to Apple’s App Store when it uses encryption (or how to obtain anERN)](https://carouselapps.com/2015/12/15/legally-submit-app-apples-app-store-uses-encryption-obtain-ern/).

# Electron 应用打包

由 425389019 创建，最后一次修改 2016-10-03

为舒缓 Windows 下路径名过长的问题[issues](https://github.com/joyent/node/issues/6960)，也为了对 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（值得是electron.exe） 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 执行asar的list命令列出打包的文件列表

/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');

这里的内容说明了我们可以将自己的文件目录打包成asar包，然后在外面使用包中的文件从而实现内容的隐藏。

### 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');

类似发送ajax请求访问本地文件

$.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。

将package.json文件拷贝到app中，然后将main的路径修改后，将app目录整体打包如:

asar pack my-app app.asar

然后就可以使用electron app.asar启动我们的应用了，需要在全局安装了依赖的node包和electron

# Electron 使用原生模块

由 425389019 创建，最后一次修改 2016-10-03

Electron 同样也支持原生模块（me：即内置了node），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。

## 原生Node模块的兼容性

当 Node 开始换新的V8引擎版本时，原生模块可能“坏”掉。为确保一切工作正常，你需要检查你想要使用的原生模块是否被 Electron 内置的 Node 支持。你可以在[这里](https://github.com/electron/electron/releases)查看 Electron 内置的 Node 版本，或者使用 process.version (参考：[快速入门](https://www.w3cschool.cn/electronmanual/p9al1qkx.html))查看（me：可以在web开发者工具中调用）。

考虑到 [NAN](https://github.com/nodejs/nan/) 可以使你的开发更容易对多版本 Node 的支持，建议使用它来开发你自己的模块。你也可以使用 [NAN](https://github.com/nodejs/nan/) 来移植旧的模块到新的 Node版本，以使它们可以在新的 Electron 下良好工作。

## 如何安装原生模块

如下三种方法教你安装原生模块：

### 最简单方式

最简单的方式就是通过 [electron-rebuild](https://github.com/paulcbetts/electron-rebuild) 包重新编译原生模块，它帮你自动完成了下载 headers、编译原生模块等步骤：

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

### 通过 npm 安装

你当然也可以通过 npm 安装原生模块。大部分步骤和安装普通模块时一样，除了以下一些系统环境变量你需要自己操作：

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

export npm\_config\_target=0.33.1

export npm\_config\_arch=x64

export npm\_config\_runtime=electron

HOME=~/.electron-gyp npm install module-name

### 通过 node-gyp 安装

你需要告诉 node-gyp 去哪下载 Electron 的 headers，以及下载什么版本：

$ cd /path-to-module/

$ HOME=~/.electron-gyp node-gyp rebuild --target=0.29.1 --arch=x64 --dist-url=https://atom.io/download/atom-shell

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

# Electron 主进程调试

由 425389019 创建，最后一次修改 2016-10-03

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（me：即在运行electron时指定启动参数）：

### --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。

## 使用 node-inspector 来调试

**备注：** Electron 目前对 node-inspector 支持的不是特别好，如果你通过 node-inspector 的 console 来检查 process 对象，主进程就会崩溃。

### 1. 确认你已经安装了 [node-gyp 所需工具](https://github.com/nodejs/node-gyp#installation)

### 2. 安装 [node-inspector](https://github.com/node-inspector/node-inspector)

$ npm install node-inspector

### 3. 安装 node-pre-gyp 的一个修订版

$ npm install git+https://git@github.com/enlight/node-pre-gyp.git#detect-electron-runtime-in-find

### 4. 为 Electron 重新编译 node-inspector v8 模块（将 target 参数修改为你的 Electron 的版本号）

$ node\_modules/.bin/node-pre-gyp --target=0.36.2 --runtime=electron --fallback-to-build --directory node\_modules/v8-debug/ --dist-url=https://atom.io/download/atom-shell reinstall

$ node\_modules/.bin/node-pre-gyp --target=0.36.2 --runtime=electron --fallback-to-build --directory node\_modules/v8-profiler/ --dist-url=https://atom.io/download/atom-shell reinstall

[How to install native modules][how-to-install-native-modules].

### 5. 打开 Electron 的调试模式

你也可以用调试参数来运行 Electron ：

$ electron --debug=5858 your/app

或者，在第一行暂停你的脚本：

$ electron --debug-brk=5858 your/app

### 6. 使用 Electron 开启 [node-inspector](https://github.com/node-inspector/node-inspector) 服务

$ ELECTRON\_RUN\_AS\_NODE=true path/to/electron.exe node\_modules/node-inspector/bin/inspector.js

### 7. 加载调试器界面

在 Chrome 中打开 <http://127.0.0.1:8080/debug?ws=127.0.0.1:8080&port=5858>

# Electron 使用 Selenium 和 WebDriver

由 425389019 创建，最后一次修改 2016-10-03

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。

为了能够使 chromedriver 和 Electron 一起正常工作，我们需要告诉它 Electron 在哪，并且让它相信 Electron 就是 Chrome 浏览器。

## 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');

**var** 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: [/\* cli arguments \*/] // 可选参数，类似：'app=' + /path/to/your/app/

}

}

};

**var** 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://www.w3cschool.cn/electronmanual/i8mc1qkl.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。

# Electron DevTools扩展

由 425389019 创建，最后一次修改 2016-10-03

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

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.md) 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。

# Electron 交互式解释器 (REPL)

由 425389019 创建，最后一次修改 2017-03-21

读取(Read)-运算(Eval)-输出(Print)-循环(Loop) (REPL) 是很简单的, 交互式的计算机编程环境，它采用单个用户输入，运算并返回结果给用户。

在这里 repl 模块提供了一个 REPL 的实现, 可以这样使用:

* 如果你的 electron 或 electron-prebuilt 已经安装为本地项目依赖项:

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

* 如果你的 electron 或 electron-prebuilt 已经为全局方式安装:

electron --interactive

这里只会为主进程创建一个REPL。 您可以使用 Dev Tools 的“控制台”选项卡来为渲染器进程获取一个REPL。

注意: electron --interactive 在 Windows 上不可用.

更多的内容可以在这里找到 [Node.js REPL docs](https://nodejs.org/dist/latest/docs/api/repl.html).

**Electron 使用 Pepper Flash 插件**

由 425389019 创建，最后一次修改 2016-10-03

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 OS X 和 Linux 上，你可以在 Chrome 浏览器的 chrome://plugins 页面上找到 PepperFlash 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 browser-window 的插件开关。例如：

// Specify flash path. 设置 flash 路径

// On Windows, it might be /path/to/pepflashplayer.dll

// On OS X, /path/to/PepperFlashPlayer.plugin

// On Linux, /path/to/libpepflashplayer.so

app.commandLine.appendSwitch('ppapi-flash-path', '/path/to/libpepflashplayer.so');

// Specify flash version, for example, v17.0.0.169 设置版本号

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

app.on('ready', **function**() {

mainWindow = **new** BrowserWindow({

'width': 800,

'height': 600,

'web-preferences': {

'plugins': true

}

});

mainWindow.loadURL('file://' + \_\_dirname + '/index.html');

// Something else

});

**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**>

# Electron 使用 Widevine CDM 插件

由 425389019 创建，最后一次修改 2016-10-03

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 配制许可 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 & OS X

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 OS X 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 andwidevinecdmadapter.dll, 在 OS X ，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 :

// You have to pass the filename of `widevinecdmadapter` here, it is

// \* `widevinecdmadapter.plugin` on OS X,

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

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

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

// The version of plugin can be got from `chrome://plugins` page in Chrome.

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

**var** mainWindow = null;

app.on('ready', **function**() {

mainWindow = **new** BrowserWindow({

webPreferences: {

// The `plugins` have to be enabled.

plugins: true

}

})

});

## 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是否 navigator.plugins 包含了 WidevineCDM 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 bitdash uses Widevine in your browser, 然后播放 video.

# Electron 术语表

由 425389019 创建，最后一次修改 2017-03-21

# 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 [lib chromium content](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#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（me：内部进程通信）。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://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#v8) 等）。

### main process

主进程，通常是指 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。主进程负责创建和销毁 APP 的每个渲染进程（me：每个渲染进程就是一个BrowserWindow实例）。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（该属性指定main文件的路径）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://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#process), [renderer process](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#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://github.com/electron/electron/blob/master/docs-translations/zh-CN/tutorials/mac-app-store-submission-guide.md) 。

### 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在 指定具体的 Electron 本地头文件。

参见： [Using Native Node Modules](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/tutorial/using-native-node-modules.md).

## 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://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#main-process) (主进程) 和一个或者多个 [renderer](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#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（me：所以可以在web开发者工具控制台上直接调用process的方法）。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://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#main-process), [renderer process](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#renderer-process)

### renderer process

渲染进程是你的应用内的一个浏览器窗口（me：每个窗口对应一个渲染进程）。与主进程（me：主进程就是app的运行进程）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://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#process), [main process](https://github.com/electron/electron/blob/master/docs-translations/zh-CN/glossary.md#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://github.com/electron/electron/blob/master/docs-translations/zh-CN/api/auto-updater.md) 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。

# Electron 离屏渲染

由 425389019 创建，最后一次修改 2017-03-21

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://github.com/electron/electron/blob/master/docs-translations/zh-CN/api/frameless-window.md).

## 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://github.com/electron/electron/blob/master/docs-translations/zh-CN/api/app.md#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)

})

# Electron 自动更新功能autoUpdater

由 caoyeshu 创建，最后一次修改 2017-08-27

# autoUpdater

启用应用程序自动更新。

过程： main

autoUpdater模块为[Squirrel](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/Squirrel&usg=ALkJrhhg-AAx6VAvjT9qxMj2gfoFLhvG4A)框架提供了一个界面。

您可以通过使用以下任一项目快速启动多平台发布服务器来分发应用程序：

* [坚果](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/GitbookIO/nuts&usg=ALkJrhg1z-iL6obbrjRalOPqkohVMm9EOw) ： 您的应用程序的智能版本服务器，使用GitHub作为后端。 用松鼠（Mac＆Windows）自动更新
* [电子发布服务器](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/ArekSredzki/electron-release-server&usg=ALkJrhgQhaJx-z0LM29A5pLib6JWNaCQhg) ： 一个功能齐全的自主发布的电子应用服务器，与自动更新兼容
* [squirrel-updates-server](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/Aluxian/squirrel-updates-server&usg=ALkJrhhxrygsYaDVu9F7ng9BWtEr4Mbbiw) ： 用于Squirrel.Mac和Squirrel.Windows的简单node.js服务器，它使用GitHub版本
* [squirrel-release-server](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/Arcath/squirrel-release-server&usg=ALkJrhhTAnk4wdf1OktjBFX6IU2NAh52Jw) ： 一个用于Squirrel.Windows的简单PHP应用程序，用于从文件夹读取更新。 支持增量更新。

## 不同平台差异

虽然autoUpdater为不同的平台提供了一个统一的API，但每个平台上仍然存在一些微妙的差异。

### MACOS

在macOS上， autoUpdater模块基于[Squirrel.Mac](https://github.com/Squirrel/Squirrel.Mac) ，这意味着您不需要任何特殊的设置来使其工作。 对于服务器端要求，您可以阅读[服务器支持](https://github.com/Squirrel/Squirrel.Mac#server-support) 。 请注意， [应用程序传输安全性](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html&usg=ALkJrhi_vR6UBdPqekZVHeLwkQtpOyaSMw#//apple_ref/doc/uid/TP40009251-SW35) （ATS）适用于作为更新过程一部分的所有请求。 需要禁用ATS的应用程序可以将NSAllowsArbitraryLoads键添加到其应用程序的plist中。

注意：您的应用程序必须签名才能自动更新macOS。 这是Squirrel.Mac的要求。

**windows**

在Windows上，您必须先将自己的应用程序安装到用户的机器中，然后才能使用autoUpdater ，因此建议您使用[electronic-winstaller](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron/windows-installer&usg=ALkJrhjt6tAIdVQw7DyxUA3ja365GwjFFw) ，  [electron-forge](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron-userland/electron-forge&usg=ALkJrhiq893XI7DVDT5QgiEoPbWVwFC8Ag)或[grunt-electron-installer](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron/grunt-electron-installer&usg=ALkJrhjnFTE2uxc0yb9e7qSMa_9Gh_XUwA)软件包来生成Windows安装程序。

当使用[electronic-winstaller](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron/windows-installer&usg=ALkJrhjt6tAIdVQw7DyxUA3ja365GwjFFw)或 [电子伪造时，请](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron-userland/electron-forge&usg=ALkJrhiq893XI7DVDT5QgiEoPbWVwFC8Ag)确保您[第一次运行时](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron/windows-installer&usg=ALkJrhjt6tAIdVQw7DyxUA3ja365GwjFFw#handling-squirrel-events)不尝试更新应用程序（另请参阅 [此问题以获取更多信息](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/electron/electron/issues/7155&usg=ALkJrhiDjDD-MNVSiv1f1aj-TlDqT_AbPg) ）。 还建议您使用[电子松鼠启动](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/mongodb-js/electron-squirrel-startup&usg=ALkJrhjWtJ8STSz_-du086GAqktoev0tJw)来获取您的应用程序的桌面快捷方式。

使用松鼠生成的安装程序将以com.squirrel.PACKAGE\_ID.YOUR\_EXE\_WITHOUT\_DOT\_EXE的格式创建一个[应用程序用户模型ID](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://msdn.microsoft.com/en-us/library/windows/desktop/dd378459(v%3Dvs.85).aspx&usg=ALkJrhjSLPHHjAh8yU3L8ebUO2YfGEjnZw)的快捷方式图标，示例为com.squirrel.slack.Slack和com.squirrel.code.Code 。 您必须使用app.setAppUserModelId API为您的应用使用相同的ID，否则Windows将无法在任务栏中正确引导您的应用。

与Squirrel.Mac不同，Windows可以在S3或任何其他静态文件主机上托管更新。 您可以阅读[Squirrel.Windows](https://translate.googleusercontent.com/translate_c?depth=1&hl=zh-CN&ie=UTF8&prev=_t&rurl=translate.google.com.hk&sl=en&sp=nmt4&tl=zh-CN&u=https://github.com/Squirrel/Squirrel.Windows&usg=ALkJrhin5rW4jtvXyinpbazd9JNjk7oTEg)的文档，以获取有关Squirrel.Windows如何工作的更多详细信息。

### Linux

在Linux上没有自动更新程序的内置支持，因此建议使用发行版的软件包管理器来更新应用程序。

**Events**

autoUpdater对象发出以下事件：

### 事件：'错误'

返回：

* error错误

更新时发生错误时发出。

### 事件：'检查更新'

检查更新是否已启动时发出。

### 事件：'更新可用'

当有可用更新时发出。 更新将自动下载。

### 事件：'update-not-available'

当没有可用更新时发出。

### 事件：'更新下载'

返回：

* event事件
* releaseNotes字符串
* releaseName字符串
* releaseDate Date
* updateURL String

下载更新时发出。

在Windows上，只有releaseName可用。

## 方法

autoUpdater对象具有以下方法：

### autoUpdater.setFeedURL(url[, requestHeaders])

* url String
* requestHeaders对象macOS （可选） - HTTP请求头。

设置url并初始化自动更新程序。

### autoUpdater.getFeedURL()

返回String - 当前的更新Feed URL。

### autoUpdater.checkForUpdates()

请求服务器是否有更新。 在使用此API之前，您必须调用setFeedURL 。

### autoUpdater.quitAndInstall()

重新启动应用程序，并在安装更新后下载。 只有在update-downloaded后才应该调用它。

注意： autoUpdater.quitAndInstall()将首先关闭所有应用程序窗口，然后才在app之后发出before-quit事件。 这与正常退出事件序列不同。

# Electron 编码规范

由 425389019 创建，最后一次修改 2016-10-08

以下是 Electron 项目的编码规范。

## C++ 和 Python

对于 C++ 和 Python，我们遵循 Chromium 的编码规范。你可以使用 script/cpplint.py 来检验文件是否符合要求。

我们目前使用的 Pyhton 版本是 Python 2.7。

C++ 代码中用到了许多 Chromium 中的接口和数据类型，所以希望你能熟悉它们。Chromium 中的重要接口和数据结构就是一篇不错的入门文档，里面提到了一些特殊类型、域内类型（退出作用域时自动释放内存）、日志机制，等等。

## CoffeeScript

对于 CoffeeScript，我们遵循 GitHub 的编码规范 及以下规则:

* 文件**不要**以换行符结尾，我们要遵循 Google 的编码规范。
* 文件名使用 - 而不是 \_ 来连接单词，比如 file-name.coffee 而不是 file\_name.coffee，这是沿用 [github/atom](https://github.com/github/atom) 模块的命名方式（module-name）。这条规则仅适用于 .coffee 文件。

## API 命名

当新建一个 API 时，我们倾向于使用 getters 和 setters 而不是 jQuery 单函数的命名方式，比如 .getText() 和 .setText(text) 而不是 .text([text])。

# Electron 源码目录结构

由 425389019 创建，最后一次修改 2016-10-08

Electron 的源代码主要依据 Chromium 的拆分约定被拆成了许多部分。

为了更好地理解源代码，您可能需要了解一下 [Chromium 的多进程架构](http://dev.chromium.org/developers/design-documents/multi-process-architecture)。

## 源代码的结构

Electron

├──atom - Electron 的源代码

| ├── app - 系统入口代码

| ├── browser - 包含了主窗口、UI 和其他所有与主进程有关的东西，它会告诉渲染进程如何管理页面

| | ├── lib - 主进程初始化代码中 JavaScript 部分的代码

| | ├── ui - 不同平台上 UI 部分的实现

| | | ├── cocoa - Cocoa 部分的源代码

| | | ├── gtk - GTK+ 部分的源代码

| | | └── win - Windows GUI 部分的源代码

| | ├── default\_app - 在没有指定 app 的情况下 Electron 启动时默认显示的页面

| | ├── api - 主进程 API 的实现

| | | └── lib - API 实现中 Javascript 部分的代码

| | ├── net - 网络相关的代码

| | ├── mac - 与 Mac 有关的 Objective-C 代码

| | └── resources - 图标，平台相关的文件等

| ├── renderer - 运行在渲染进程中的代码

| | ├── lib - 渲染进程初始化代码中 JavaScript 部分的代码

| | └── api - 渲染进程 API 的实现

| | └── lib - API 实现中 Javascript 部分的代码

| └── common - 同时被主进程和渲染进程用到的代码，包括了一些用来将 node 的事件循环

| | 整合到 Chromium 的事件循环中时用到的工具函数和代码

| ├── lib - 同时被主进程和渲染进程使用到的 Javascript 初始化代码

| └── api - 同时被主进程和渲染进程使用到的 API 的实现以及 Electron 内置模块的基础设施

| └── lib - API 实现中 Javascript 部分的代码

├── chromium\_src - 从 Chromium 项目中拷贝来的代码

├── docs - 英语版本的文档

├── docs-translations - 各种语言版本的文档翻译

├── spec - 自动化测试

├── atom.gyp - Electron 的构建规则

└── common.gypi - 为诸如 `node` 和 `breakpad` 等其他组件准备的编译设置和构建规则

## 其他目录的结构

* **script** - 用于诸如构建、打包、测试等开发用途的脚本
* **tools** - 在 gyp 文件中用到的工具脚本，但与 script 目录不同， 该目录中的脚本不应该被用户直接调用
* **vendor** - 第三方依赖项的源代码，为了防止人们将它与 Chromium 源码中的同名目录相混淆， 在这里我们不使用 third\_party 作为目录名
* **node\_modules** - 在构建中用到的第三方 node 模块
* **out** - ninja 的临时输出目录
* **dist** - 由脚本 script/create-dist.py 创建的临时发布目录
* **external\_binaries** - 下载的不支持通过 gyp 构建的预编译第三方框架