# 快速入门

Electron 可以让你使用纯 JavaScript 调用丰富的原生 APIs 来创造桌面应用。你可以把它看作一个专注于桌面应用的 Node.js 的变体，而不是 Web 服务器。

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，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

在 Electron 里，运行 package.json 里 main 脚本的进程被称为**主进程**。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，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 用户拥有在网页中调用 Node.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 实例创建页面。每个 BrowserWindow 实例都在自己的渲染进程里运行页面。当一个 BrowserWindow 实例被销毁后，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页面。

由于在页面里管理原生 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，请求主进程进行相关的 GUI 操作。

在 Electron，我们提供几种方法用于主进程和渲染进程之间的通讯。像 [ipcRenderer](https://electron.org.cn/doc/api/ipc-renderer.html) 和 [ipcMain](https://electron.org.cn/doc/api/ipc-main.html) 模块用于发送消息， [remote](https://electron.org.cn/doc/api/remote.html) 模块用于 RPC 方式通讯。这些内容都可以在一个 FAQ 中查看 [how to share data between web pages](https://electron.org.cn/doc/faq.html#how-to-share-data-between-web-pages)。

# 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事件，一个典型的例子如下：

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

const path = require('path')

const url = require('url')

// 保持一个对于 window 对象的全局引用，如果你不这样做，

// 当 JavaScript 对象被垃圾回收， window 会被自动地关闭

let win

function createWindow () {

// 创建浏览器窗口。

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

// 加载应用的 index.html。

win.loadURL(url.format({

pathname: path.join(\_\_dirname, 'index.html'),

protocol: 'file:',

slashes: true

}))

// 打开开发者工具。

win.webContents.openDevTools()

// 当 window 被关闭，这个事件会被触发。

win.on('closed', () => {

// 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，

// 与此同时，你应该删除相应的元素。

win = null

})

}

// Electron 会在初始化后并准备

// 创建浏览器窗口时，调用这个函数。

// 部分 API 在 ready 事件触发后才能使用。

app.on('ready', createWindow)

// 当全部窗口关闭时退出。

app.on('window-all-closed', () => {

// 在 macOS 上，除非用户用 Cmd + Q 确定地退出，

// 否则绝大部分应用及其菜单栏会保持激活。

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

app.quit()

}

})

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

// 在这文件，你可以续写应用剩下主进程代码。

// 也可以拆分成几个文件，然后用 require 导入。

if (win === null) {

createWindow()

}

})

// 在这文件，你可以续写应用剩下主进程代码。

// 也可以拆分成几个文件，然后用 require 导入。

最后，你想展示的 index.html：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello World!</title>

</head>

<body>

<h1>Hello World!</h1>

We are using node <script>document.write(process.versions.node)</script>,

Chrome <script>document.write(process.versions.chrome)</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-prebuilt

[electron](https://github.com/electron-userland/electron-prebuilt) 是一个 npm 模块，包含所使用的 Electron 预编译版本。 如果你已经用 npm 全局安装（npm install –g –save-dev electron）了它，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：

#### macOS / Linux

$ ./node\_modules/.bin/electron .

#### Windows

$ .\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/

### macOS

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

Electron.app 里面是 Electron 发布包，你可以在 [这里](https://electron.org.cn/release.html) 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://electron.org.cn/doc/tutorial/application-distribution.html) 指导发布一个版本，并且以已经打包好的形式运行应用。

# 参照下面例子

复制并且运行这个库 [electron/electron-quick-start](https://github.com/electron/electron-quick-start)。

**注意：**运行时需要你的系统已经安装了 [Git](https://git-scm.com/) 和 [Node.js](https://nodejs.org/en/download/)（包含 [npm](https://npm.js.cn/)）。

# 克隆这仓库

$ git clone https://github.com/electron/electron-quick-start

# 进入仓库

$ cd electron-quick-start

# 安装依赖库并运行应用

$ npm install && npm start

更多 apps 例子，查看 electron 社区创建的 [list of boilerplates](https://electron.atom.io/community/#boilerplates)。

# 桌面环境集成

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。

## Notifications (Windows, Linux, macOS)

这三个操作系统都为用户提供了发送通知的方法。Electron 让开发人员通过 [HTML5 Notification API](https://notifications.spec.whatwg.org/) 便利的去发送通知，用操作系统自带的通知 APIs 去显示。

**Note:** 因为这是一个 HTML5 API，所以只在渲染进程中起作用.

let myNotification = new Notification('Title', {

body: 'Lorem Ipsum Dolor Sit Amet'

})

myNotification.onclick = () => {

console.log('Notification clicked')

}

尽管代码和用户体验在不同的操作系统中基本相同，但还是有一些差异。

### Windows

* 在 Windows 10 上, 通知"可以工作"。
* 在 Windows 8.1 和 Windows 8 系统下，你需要将你的应用通过一个[Application User Model ID](https://msdn.microsoft.com/en-us/library/windows/desktop/dd378459(v=vs.85).aspx)安装到开始屏幕上。需要注意的是，这不是将你的应用固定到开始屏幕。
* 在 Windows 7 以及更低的版本中，通知不被支持。不过你可以使用 [Tray API](https://electron.org.cn/doc/api/tray.html#traydisplayballoonoptions-windows) 发送一个"气泡通知"。

此外，通知支持的最大字符长度为250。Windows 团队建议通知应该保持在200个字符以下。

### Linux

通知使用 libnotify 发送，它能在任何支持[Desktop Notifications Specification](https://developer.gnome.org/notification-spec/)的桌面环境中显示，包括 Cinnamon, Enlightenment, Unity, GNOME, KDE。

### macOS

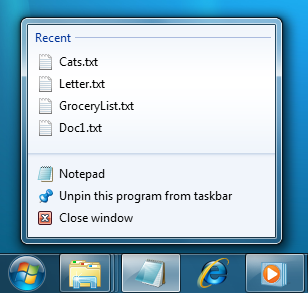
在 macOS 系统中，通知是直接转发的，你应该了解 [Apple's Human Interface guidelines regarding notifications](https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/OSXHIGuidelines/NotificationCenter.html)。

注意通知被限制在256个字节以内，如果超出，则会被截断。

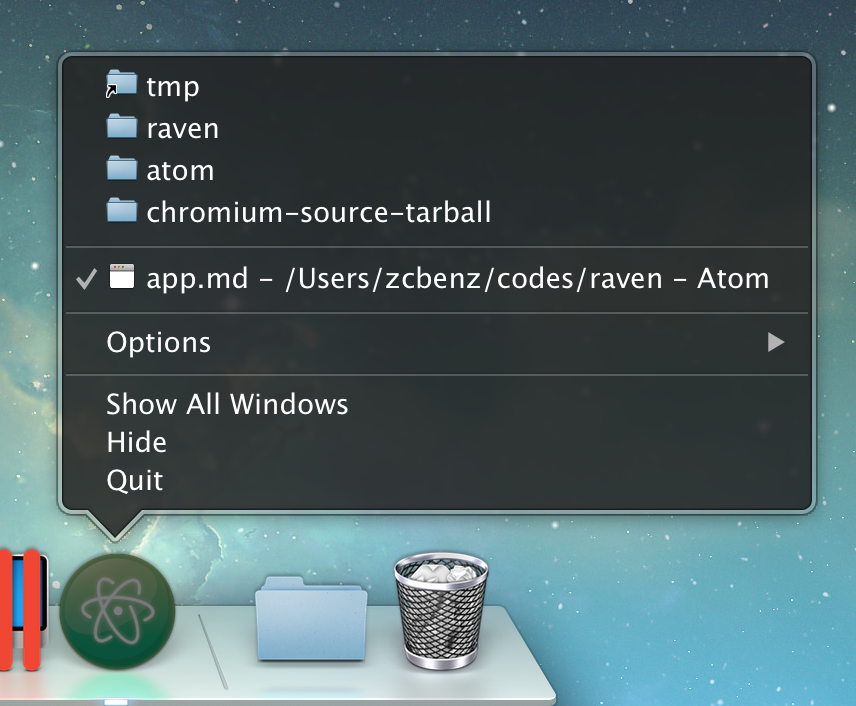
## 最近文档 (Windows & macOS)

Windows 和 macOS 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][3] API:

const {app} = require('electron')

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

或者你也可以使用 [app.clearRecentDocuments][4] API 来清空最近文件列表。

const {app} = require('electron')

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](http://msdn.microsoft.com/en-us/library/windows/desktop/ee872121(v=vs.85).aspx) 里找到任何关于注册事宜的说明。

当用户点击从“跳转列表”点击一个文件，你的应用程序的新实例 将以添加为命令行参数的文件的路径启动。

## macOS 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(macOS)

macOS 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

**Dock menu of Terminal.app:**



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

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

const dockMenu = Menu.buildFromTemplate([

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

{label: 'New Window with Settings',

submenu: [

{label: 'Basic'},

{label: 'Pro'}

]

},

{label: 'New Command...'}

])

app.dock.setMenu(dockMenu)

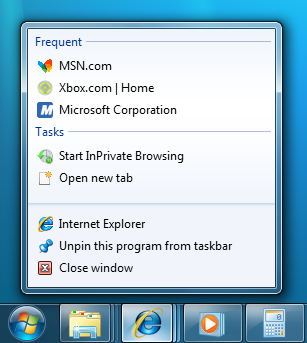
## 用户任务(Windows)

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 macOS 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://electron.org.cn/doc/api/app.html#appsetusertaskstasks-windows) API 来设置你的应用中的用户任务：

const {app} = require('electron')

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：

const {app} = require('electron')

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://electron.org.cn/doc/api/browser-window.html#winsetthumbarbuttonsbuttons-windows-7) 来设置你的应用的缩略图工具栏。

const {BrowserWindow} = require('electron')

const path = require('path')

let win = new BrowserWindow({

width: 800,

height: 600

})

win.setThumbarButtons([

{

tooltip: 'button1',

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

click () { console.log('button1 clicked') }

},

{

tooltip: 'button2',

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

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

click () { console.log('button2 clicked.') }

}

])

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

const {BrowserWindow} = require('electron')

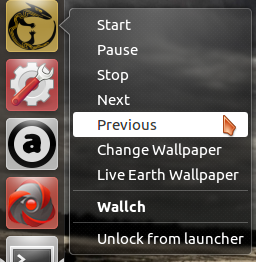
let win = new BrowserWindow()

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)。

**Launcher shortcuts of Audacious:**



### Audacious 运行器的快捷方式：

![Launcher shortcuts of Audacious][12]

## 任务栏的进度条(Windows, macOS, 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。

在 macOS，进度条将显示为 dock 图标的一部分。

Unity DE 也具有同样的特性，在运行器上显示进度条。

**Progress bar in taskbar button:**



给一个窗口设置进度条，你可以调用 [BrowserWindow.setProgressBar](https://electron.org.cn/doc/api/browser-window.html#winsetprogressbarprogress) API：

const {BrowserWindow} = require('electron')

let win = new BrowserWindow()

win.setProgressBar(0.5)

## 任务栏中的叠加层图标 (Windows)

在 Windows，任务栏按钮可以使用小型叠加层显示应用程序 状态，引用 MSDN 的文档：

Icon overlays serve as a contextual notification of status, and are intended to negate the need for a separate notification area status icon to communicate that information to the user. For instance, the new mail status in Microsoft Outlook, currently shown in the notification area, can now be indicated through an overlay on the taskbar button. Again, you must decide during your development cycle which method is best for your application. Overlay icons are intended to supply important, long-standing status or notifications such as network status, messenger status, or new mail. The user should not be presented with constantly changing overlays or animations.

**Overlay on taskbar button:**

Overlay on taskbar button

要设置窗口的叠加层图标，可以使用 [BrowserWindow.setOverlayIcon](https://electron.org.cn/doc/api/browser-window.html#winsetoverlayiconoverlay-description-windows-7) API:

const {BrowserWindow} = require('electron')

let win = new BrowserWindow()

win.setOverlayIcon('path/to/overlay.png', 'Description for overlay')

## 突出显示框架 (Windows)

在 Windows，你可以突出显示任务栏按钮，以获得用户的关注。 这类似于在 macOS 上弹出 dock 图标。 在 MSDN，它如是说：

Typically, a window is flashed to inform the user that the window requires attention but that it does not currently have the keyboard focus.

要在 BrowserWindow 的任务栏按钮突出显示，可以使用 [BrowserWindow.flashFrame](https://electron.org.cn/doc/api/browser-window.html#winflashframeflag) API:

const {BrowserWindow} = require('electron')

let win = new BrowserWindow()

win.once('focus', () => win.flashFrame(false))

win.flashFrame(true)

不要忘记在调用 flashFrame 方法后，设置 false 来关闭突出显示。 在 上面的例子，它是在窗口进入焦点时调用的，但你可能会 使用超时或某些其他事件来禁用它。

## 展示文件窗口 (macOS)

在 macOS，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。

您还可以设置窗口的编辑状态，以便文件图标可以指示 该窗口中的文档是否已修改。

**Represented file popup menu:**

要设置展示文件窗口，可以使用 [BrowserWindow.setRepresentedFilename](https://electron.org.cn/doc/api/browser-window.html#winsetrepresentedfilenamefilename-os-x) 和 [BrowserWindow.setDocumentEdited](https://electron.org.cn/doc/api/browser-window.html#winsetdocumenteditededited-os-x) APIs：

const {BrowserWindow} = require('electron')

let win = new BrowserWindow()

win.setRepresentedFilename('/etc/passwd')

win.setDocumentEdited(true)

## 将文件拖出窗口

对于某些操作文件的应用程序， 将文件从 Electron 拖动到其他应用程序是很重要的能力。要在 app 的实现此功能 ，你需要在 ondragstart 事件上调用 webContents.startDrag(item) API。

在网页端：

<a href="#" id="drag">item</a>

<script type="text/javascript" charset="utf-8">

document.getElementById('drag').ondragstart = (event) => {

event.preventDefault()

ipcRenderer.send('ondragstart', '/path/to/item')

}

</script>

在主进程：

const {ipcMain} = require('electron')

ipcMain.on('ondragstart', (event, filePath) => {

event.sender.startDrag({

file: filePath,

icon: '/path/to/icon.png'

})

})

在线/离线事件探测

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*

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

let onlineStatusWindow

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

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

onlineStatusWindow.loadURL(`file://${\_\_dirname}/online-status.html`)

})

*online-status.html*

<!DOCTYPE html>

<html>

<body>

<script>

const alertOnlineStatus = () => {

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，事件就可以在主进程被使用，就像下面的例子：

*main.js*

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

let onlineStatusWindow

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

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

onlineStatusWindow.loadURL(`file://${\_\_dirname}/online-status.html`)

})

ipcMain.on('online-status-changed', (event, status) => {

console.log(status)

})

*online-status.html*

<!DOCTYPE html>

<html>

<body>

<script>

const {ipcRenderer} = require('electron')

const updateOnlineStatus = () => {

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

}

window.addEventListener('online', updateOnlineStatus)

window.addEventListener('offline', updateOnlineStatus)

updateOnlineStatus()

</script>

</body>

</html>

**注意：** 如果 Electron 无法连接到局域网（LAN）或 一个路由器，它被认为是离线的; 所有其他条件返回 true。 所以，虽然你可以假设 Electron 是离线的，当 navigator.onLine 返回一个 `false' 值，你不能假设一个 'true' 值必然 意味着 Electron 可以访问互联网。你可能会获得虚假连接，例如在计算机运行虚拟化软件的情况下，具有始终“连接”的虚拟以太网适配器。 因此，如果您真的想确定 Electron 的互联网访问状态， 你应该开发额外的检查手段。

# 交互式解释器 (REPL)

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).