# Electron window.open 函数

由 425389019 创建，最后一次修改 2016-10-04

当在界面中使用 window.open 来创建一个新的窗口时候，将会创建一个 BrowserWindow 的实例（me：类似new BrowserWindow实例）返回一个标识，这个界面通过标识来对这个新的窗口进行有限的控制.

这个标识对传统的web界面来说，通过它能对子窗口进行有限的功能性兼容控制.想要完全的控制这个窗口，可以直接创建一个 BrowserWindow .

新创建的 BrowserWindow 默认为继承父窗口的属性参数，想重写属性的话可以在 features 中设置他们.

**window.open(url[, frameName][, features])**

* url String
* frameName String (可选)
* features String (可选)

创建一个新的window并且返回一个 BrowserWindowProxy 类的实例.

features 遵循标准浏览器的格式，但是每个feature 应该作为 BrowserWindow 参数的一个字段.

**window.opener.postMessage(message, targetOrigin)**

* message String
* targetOrigin String

通过指定位置或用 \* 来代替没有明确位置来向父窗口发送信息.

**Class: BrowserWindowProxy**

BrowserWindowProxy 由window.open 创建返回，并且提供了对子窗口的有限功能性控制.

**BrowserWindowProxy.blur()**

子窗口的失去焦点.

**BrowserWindowProxy.close()**

强行关闭子窗口，忽略卸载事件.

**BrowserWindowProxy.closed**

在子窗口关闭之后恢复正常.

**BrowserWindowProxy.eval(code)**

* code String

评估子窗口的代码.

**BrowserWindowProxy.focus()**

子窗口获得焦点(让其显示在最前).

**BrowserWindowProxy.postMessage(message, targetOrigin)**

* message String
* targetOrigin String

通过指定位置或用 \* 来代替没有明确位置来向子窗口发送信息.

除了这些方法，子窗口还可以无特性和使用单一方法来实现 window.opener 对象.

# 在主进程中可用的模块

**Electron app 模块**

由 425389019 创建，最后一次修改 2016-10-08

app 模块是为了控制整个应用（app）的生命周期设计的(me：在主进程中运行，因此在main.js中定义)。

下面的这个例子将会展示如何在最后一个窗口被关闭时退出应用：

var app = require('app');

app.on('window-all-closed', function() {//所有窗口关闭后会触发window-all-closed事件

app.quit();

});

**事件列表**

app 对象会触发以下的事件：

**事件：'will-finish-launching'**

当应用程序完成基础的启动的时候被触发。在 Windows 和 Linux 中， will-finish-launching 事件与 ready 事件是相同的； 在 OS X 中， 这个时间相当于 NSApplication 中的 applicationWillFinishLaunching 提示。 你应该经常在这里为 open-file 和 open-url 设置监听器，并启动崩溃报告和自动更新。

在大多数的情况下，你应该只在 ready 事件处理器中完成所有的业务。

**事件：'ready'**

当 Electron 完成初始化时被触发。

**事件：'window-all-closed'**

当所有的窗口都被关闭时触发。

这个时间仅在应用还没有退出时才能触发。 如果用户按下了 Cmd + Q， 或者开发者调用了 app.quit() ，Electron 将会先尝试关闭所有的窗口再触发 will-quit 事件， 在这种情况下 window-all-closed 不会被触发。

**事件：'before-quit'**

返回：

* event Event

在应用程序开始关闭它的窗口的时候被触发。 调用 event.preventDefault() 将会阻止终止应用程序的默认行为。

**事件：'will-quit'**

返回：

* event Event

当所有的窗口已经被关闭，应用即将退出时被触发。 调用 event.preventDefault() 将会阻止终止应用程序的默认行为。

你可以在 window-all-closed 事件的描述中看到 will-quit 事件 和 window-all-closed 事件的区别。

**事件：'quit'**

返回：

* event Event
* exitCode Integer

当应用程序正在退出时触发。

**事件：'open-file' OS X**

返回：

* event Event
* path String

当用户想要在应用中打开一个文件时触发。open-file 事件常常在应用已经打开并且系统想要再次使用应用打开文件时被触发。 open-file 也会在一个文件被拖入 dock 且应用还没有运行的时候被触发。 请确认在应用启动的时候（甚至在 ready 事件被触发前）就对 open-file 事件进行监听，以处理这种情况。

如果你想处理这个事件，你应该调用 event.preventDefault() 。 在 Windows系统中，你需要通过解析 process.argv 来获取文件路径。

**事件：'open-url' OS X**

返回：

* event Event
* url String

当用户想要在应用中打开一个url的时候被触发。URL格式必须要提前标识才能被你的应用打开。

如果你想处理这个事件，你应该调用 event.preventDefault() 。

**事件：'activate' OS X**

返回：

* event Event
* hasVisibleWindows Boolean

当应用被激活时触发，常用于点击应用的 dock 图标的时候。

**事件：'browser-window-blur'**

返回：

* event Event
* window BrowserWindow

当一个 [BrowserWindow](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html" \t "_blank) 失去焦点的时候触发。

**事件：'browser-window-focus'**

返回：

* event Event
* window BrowserWindow

当一个 [BrowserWindow](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html" \t "_blank) 获得焦点的时候触发。

**事件：'browser-window-created'**

返回：

* event Event
* window BrowserWindow

当一个 [BrowserWindow](https://www.w3cschool.cn/electronmanual/electronmanual-browser-window.html" \t "_blank) 被创建的时候触发。

**事件：'certificate-error'**

返回：

* event Event
* webContents [WebContents](https://www.w3cschool.cn/electronmanual/electronmanual-web-contents.html" \t "_blank)
* url String - URL 地址
* error String - 错误码
* certificate Object
* data Buffer - PEM 编码数据
* issuerName String - 发行者的公有名称
* callback Function

当对 url 验证 certificate 证书失败的时候触发，如果需要信任这个证书，你需要阻止默认行为 event.preventDefault() 并且 调用 callback(true)。

session.on('certificate-error', function(event, webContents, url, error, certificate, callback) {

if (url == "https://github.com") {

// 验证逻辑。

event.preventDefault();

callback(true);

} else {

callback(false);

}

});

**事件：'select-client-certificate'**

返回：

* event Event
* webContents [WebContents](https://www.w3cschool.cn/electronmanual/electronmanual-web-contents.html" \t "_blank)
* url String - URL 地址
* certificateList [Object]
* data Buffer - PEM 编码数据
* issuerName String - 发行者的公有名称
* callback Function

当一个客户端认证被请求的时候被触发。

url 指的是请求客户端认证的网页地址，调用 callback 时需要传入一个证书列表中的证书。

需要通过调用 event.preventDefault() 来防止应用自动使用第一个证书进行验证。如下所示：

app.on('select-certificate', function(event, host, url, list, callback) {

event.preventDefault();

callback(list[0]);

})

**事件: 'login'**

返回：

* event Event
* webContents [WebContents](https://www.w3cschool.cn/electronmanual/electronmanual-web-contents.html" \t "_blank)
* request Object
  + method String
  + url URL
  + referrer URL
* authInfo Object
  + isProxy Boolean
  + scheme String
  + host String
  + port Integer
  + realm String
* callback Function

当 webContents 要做进行一次 HTTP 登陆验证时被触发。

默认情况下，Electron 会取消所有的验证行为，如果需要重写这个行为，你需要用 event.preventDefault() 来阻止默认行为，并且 用 callback(username, password) 来进行验证。

app.on('login', function(event, webContents, request, authInfo, callback) {

event.preventDefault();

callback('username', 'secret');

})

**事件：'gpu-process-crashed'**

当 GPU 进程崩溃时触发。

**方法列表**

app 对象拥有以下的方法：

**请注意** 有的方法只能用于特定的操作系统。

**app.quit()**

试图关掉所有的窗口。before-quit 事件将会最先被触发。如果所有的窗口都被成功关闭了， will-quit事件将会被触发，默认下应用将会被关闭。

这个方法保证了所有的 beforeunload 和 unload 事件处理器被正确执行。假如一个窗口的 beforeunload事件处理器返回 false，那么整个应用可能会取消退出。

**app.hide() OS X**

隐藏所有的应用窗口，不是最小化.

**app.show() OS X**

隐藏后重新显示所有的窗口，不会自动选中他们。

**app.exit(exitCode)**

* exitCode 整数

带着exitCode退出应用.

所有的窗口会被立刻关闭，不会询问用户。before-quit 和 will-quit 这2个事件不会被触发

**app.getAppPath()**

返回当前应用所在的文件路径。

**app.getPath(name)**

* name String

返回一个与 name 参数相关的特殊文件夹或文件路径。当失败时抛出一个 Error 。

你可以通过名称请求以下的路径：

* home 用户的 home 文件夹（主目录）
* appData 当前用户的应用数据文件夹，默认对应：
  + %APPDATA% Windows 中
  + $XDG\_CONFIG\_HOME or ~/.config Linux 中
  + ~/Library/Application Support OS X 中
* userData 储存你应用程序设置文件的文件夹，默认是 appData 文件夹附加应用的名称
* temp 临时文件夹
* exe 当前的可执行文件
* module libchromiumcontent 库
* desktop 当前用户的桌面文件夹
* documents 用户文档目录的路径
* downloads 用户下载目录的路径.
* music 用户音乐目录的路径.
* pictures 用户图片目录的路径.
* videos 用户视频目录的路径.

**app.setPath(name, path)**

* name String
* path String

重写某个 name 的路径为 path，path 可以是一个文件夹或者一个文件，这个和 name 的类型有关。 如果这个路径指向的文件夹不存在，这个文件夹将会被这个方法创建。 如果错误则会抛出 Error。

name 参数只能使用 app.getPath 中定义过 name。

默认情况下，网页的 cookie 和缓存都会储存在 userData 文件夹。 如果你想要改变这个位置，你需要在 app 模块中的 ready 事件被触发之前重写 userData 的路径。

**app.getVersion()**

返回加载应用程序的版本。如果应用程序的 package.json 文件中没有写版本号， 将会返回当前包或者可执行文件的版本。

**app.getName()**

返回当前应用程序的 package.json 文件中的名称。

由于 npm 的命名规则，通常 name 字段是一个短的小写字符串。但是应用名的完整名称通常是首字母大写的，你应该单独使用一个 productName 字段，用于表示你的应用程序的完整名称。Electron 会优先使用这个字段作为应用名。

**app.setName(name)**

* name String

重写当前应用的名字

**app.getLocale()**

返回当前应用程序的语言。

**app.addRecentDocument(path) OS X Windows**

* path String

在最近访问的文档列表中添加 path。

这个列表由操作系统进行管理。在 Windows 中您可以通过任务条进行访问，在 OS X 中你可以通过 dock 菜单进行访问。

**app.clearRecentDocuments() OS X Windows**

清除最近访问的文档列表。

**app.setUserTasks(tasks) Windows**

* tasks [Task] - 一个由 Task 对象构成的数组

将 tasks 添加到 Windows 中 JumpList 功能的 [Tasks](http://msdn.microsoft.com/en-us/library/windows/desktop/dd378460(v=vs.85).aspx#tasks) 分类中。

tasks 中的 Task 对象格式如下：

Task Object

* program String - 执行程序的路径，通常你应该说明当前程序的路径为 process.execPath 字段。
* arguments String - 当 program 执行时的命令行参数。
* title String - JumpList 中显示的标题。
* description String - 对这个任务的描述。
* iconPath String - JumpList 中显示的图标的绝对路径，可以是一个任意包含一个图标的资源文件。通常来说，你可以通过指明 process.execPath 来显示程序中的图标。
* iconIndex Integer - 图标文件中的采用的图标位置。如果一个图标文件包括了多个图标，就需要设置这个值以确定使用的是哪一个图标。 如果这个图标文件中只包含一个图标，那么这个值为 0。

**app.allowNTLMCredentialsForAllDomains(allow)**

* allow Boolean

动态设置是否总是为 HTTP NTLM 或 Negotiate 认证发送证书。通常来说，Electron 只会对本地网络（比如和你处在一个域中的计算机）发 送 NTLM / Kerberos 证书。但是假如网络设置得不太好，可能这个自动探测会失效，所以你可以通过这个接口自定义 Electron 对所有 URL 的行为。

**app.makeSingleInstance(callback)**

* callback Function

这个方法可以让你的应用在同一时刻最多只会有一个实例，否则你的应用可以被运行多次并产生多个实例。你可以利用这个接口保证只有一个实例正 常运行，其余的实例全部会被终止并退出。

如果多个实例同时运行，那么第一个被运行的实例中 callback 会以 callback(argv, workingDirectory)的形式被调用。其余的实例 会被终止。 argv 是一个包含了这个实例的命令行参数列表的数组，workingDirectory 是这个实例目前的运行目录。通常来说，我们会用通过将应用在 主屏幕上激活，并且取消最小化，来提醒用户这个应用已经被打开了。

在 app 的 ready 事件后，callback 才有可能被调用。

如果当前实例为第一个实例，那么在这个方法将会返回 false 来保证它继续运行。否则将会返回 true 来让它立刻退出。

在 OS X 中，如果用户通过 Finder、open-file 或者 open-url 打开应用，系统会强制确保只有一个实例在运行。但是如果用户是通过 命令行打开，这个系统机制会被忽略，所以你仍然需要靠这个方法来保证应用为单实例运行的。

下面是一个简单的例子。我们可以通过这个例子了解如何确保应用为单实例运行状态。

var myWindow = null;

var shouldQuit = app.makeSingleInstance(function(commandLine, workingDirectory) {

// 当另一个实例运行的时候，这里将会被调用，我们需要激活应用的窗口

if (myWindow) {

if (myWindow.isMinimized()) myWindow.restore();

myWindow.focus();

}

return true;

});

// 这个实例是多余的实例，需要退出

if (shouldQuit) {

app.quit();

return;

}

// 创建窗口、继续加载应用、应用逻辑等……

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

});

**app.setAppUserModelId(id) Windows**

* id String

改变当前应用的 [Application User Model ID](https://msdn.microsoft.com/en-us/library/windows/desktop/dd378459(v=vs.85).aspx) 为 id.

**app.isAeroGlassEnabled() Windows**

如果 [DWM composition](https://msdn.microsoft.com/en-us/library/windows/desktop/aa969540.aspx)(Aero Glass) 启用 了，那么这个方法会返回 true，否则是 false。你可以用这个方法来决定是否要开启透明窗口特效，因为如果用户没开启 DWM，那么透明窗 口特效是无效的。

举个例子：

let browserOptions = {width: 1000, height: 800};

// 只有平台支持的时候才使用透明窗口

if (process.platform !== 'win32' || app.isAeroGlassEnabled()) {

browserOptions.transparent = true;

browserOptions.frame = false;

}

// 创建窗口

win = new BrowserWindow(browserOptions);

// 转到某个网页

if (browserOptions.transparent) {

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

} else {

// 没有透明特效，我们应该用某个只包含基本样式的替代解决方案。

win.loadURL('file://' + \_\_dirname + '/fallback.html');

}

**app.commandLine.appendSwitch(switch[, value])**

通过可选的参数 value 给 Chromium 中添加一个命令行开关。

**注意** 这个方法不会影响 process.argv，我们通常用这个方法控制一些底层 Chromium 行为。

**app.commandLine.appendArgument(value)**

给 Chromium 中直接添加一个命令行参数，这个参数 value 的引号和格式必须正确。

**注意** 这个方法不会影响 process.argv。

**app.dock.bounce([type]) OS X**

* type String - 可选参数，可以是 critical 或 informational。默认为 informational。

当传入的是 critical 时，dock 中的应用将会开始弹跳，直到这个应用被激活或者这个请求被取消。

当传入的是 informational 时，dock 中的图标只会弹跳一秒钟。但是，这个请求仍然会激活，直到应用被激活或者请求被取消。

这个方法返回的返回值表示这个请求的 ID。

**app.dock.cancelBounce(id) OS X**

* id Integer

取消这个 id 对应的请求。

**app.dock.setBadge(text) OS X**

* text String

设置应用在 dock 中显示的字符串。

**app.dock.getBadge() OS X**

返回应用在 dock 中显示的字符串。

**app.dock.hide() OS X**

隐藏应用在 dock 中的图标。

**app.dock.show() OS X**

显示应用在 dock 中的图标。

**app.dock.setMenu(menu) OS X**

* menu [Menu](https://www.w3cschool.cn/electronmanual/electronmanual-menu.html" \t "_blank)

设置应用的 dock 菜单.

**app.dock.setIcon(image) OS X**

* image [NativeImage](https://www.w3cschool.cn/electronmanual/electronmanual-native-image.html" \t "_blank)

设置应用在 dock 中显示的图标。

**Electron ipcMain 模块**

由 425389019 创建，最后一次修改 2016-10-08

ipcMain 模块是类 [EventEmitter](https://nodejs.org/api/events.html" \t "_blank) 的实例（me:ipcMain是事件触发的实例,可以在代码中new 一个 EventEmitter实例来模拟这些事件触发）.当在主进程中使用它的时候，它控制着由渲染进程(web page)发送过来的异步或同步消息.从渲染进程发送过来的消息将触发监听的事件.

使用electron .的方式启动应用，主进程在控制台监听。

调用同步发送消息方法才能收到同步回应消息，而只要channel一样就能够被监听到

**发送消息**

同样也可以从主进程向渲染进程发送消息，查看更多 [webContents.send](https://www.w3cschool.cn/electronmanual/electronmanual-web-contents.html" \t "_blank) .

* 发送消息，事件名为 channel.
* 回应同步消息, 你可以设置 event.returnValue（me：默认使用的是发送消息使用的channel，类似长连接，发送方法返回的结果就是回应方event.retrueValue的值）.
* 回应异步消息, 你可以使用 event.sender.send(...)（me：需要自己设置发送消息的channel，相当于接收方有成为了一个普通的发送方）.

一个例子，在主进程和渲染进程之间发送和处理消息:

// In main process.

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

ipcMain.on('asynchronous-message', **function**(event, arg) {

console.log(arg); // prints "ping"

event.sender.send('asynchronous-reply', 'pong');

});

ipcMain.on('synchronous-message', **function**(event, arg) {

console.log(arg); // prints "ping"

event.returnValue = 'pong';

});

// In renderer process (web page).

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

console.log(ipcRenderer.sendSync('synchronous-message', 'ping')); // prints "pong"，需要调用同步发送才能收到返回的同步回应消息

ipcRenderer.on('asynchronous-reply', **function**(event, arg) {

console.log(arg); // prints "pong"

});

ipcRenderer.send('asynchronous-message', 'ping');

**监听消息**

ipcMain 模块有如下监听事件方法:

**ipcMain.on(channel, listener)**

* channel String
* listener Function

监听 channel, 当新消息到达，将通过 listener(event, args...) 调用 listener.

**ipcMain.once(channel, listener)**

* channel String
* listener Function

为事件添加一个一次性用的listener 函数.这个 listener 只有在下次的消息到达 channel 时被请求调用，之后就被删除了.

**ipcMain.removeListener(channel, listener)**

* channel String
* listener Function

为特定的 channel 从监听队列中删除特定的 listener 监听者.

**ipcMain.removeAllListeners([channel])**

* channel String (可选)

删除所有监听者，或特指的 channel 的所有监听者.

一般不要移除所有channel上的listeners，否则应用将不能正常工作了

**事件对象**

传递给 callback 的 event 对象有如下方法:

**event.returnValue**

将此设置为在一个同步消息中返回的值.

**event.sender**

返回发送消息的 webContents ，你可以调用 event.sender.send 来回复异步消息，更多信息 [webContents.send](https://www.w3cschool.cn/electronmanual/electronmanual-web-contents.html" \t "_blank).

# 在渲染进程中（网页内）可用的模块Electron ipcRenderer 模块

由 425389019 创建，最后一次修改 2016-10-08

ipcRenderer 模块是一个 [EventEmitter](https://nodejs.org/api/events.html" \t "_blank) 类的实例（me：与ipcMain一样）. 它提供了有限的方法，你可以从渲染进程向主进程发送同步（sendSync）或异步(send)消息. 也可以收到主进程的相应(同步回应和异步回应).

查看 [ipcMain](https://www.w3cschool.cn/electronmanual/electronmanual-ipc-main.html" \t "_blank) 代码例子.

**消息监听**

ipcRenderer 模块有下列方法来监听事件:

**ipcRenderer.on(channel, listener)**

* channel String
* listener Function

监听 channel, 当有新消息到达，使用 listener(event, args...) 调用 listener .

**ipcRenderer.once(channel, listener)**

* channel String
* listener Function

为这个事件添加一个一次性 listener 函数.这个 listener 将在下一次有新消息被发送到 channel 的时候被请求调用，之后就被删除了.

**ipcRenderer.removeListener(channel, listener)**

* channel String
* listener Function

从指定的 channel 中的监听者数组删除指定的 listener .

**ipcRenderer.removeAllListeners([channel])**

* channel String (optional)

删除所有的监听者，或者删除指定 channel 中的全部.

**发送消息**

ipcRenderer 模块有如下方法来发送消息:

**ipcRenderer.send(channel[, arg1][, arg2][, ...])**

* channel String
* arg (可选)

通过 channel 向主进程发送异步消息，也可以发送任意参数.参数会被JSON序列化，之后就不会包含函数或原型链.

主进程通过使用 ipcMain 模块来监听 channel，从而处理消息.

**ipcRenderer.sendSync(channel[, arg1][, arg2][, ...])**

* channel String
* arg (可选)

通过 channel 向主进程发送同步消息，也可以发送任意参数.参数会被JSON序列化，之后就不会包含函数或原型链.

主进程通过使用 ipcMain 模块来监听 channel，从而处理消息, 通过 event.returnValue 来响应同步消息.

**注意:** 发送同步消息将会阻塞整个渲染进程,除非你知道你在做什么，否则就永远不要用它，等待主进程回应 .

**ipcRenderer.sendToHost(channel[, arg1][, arg2][, ...])**

* channel String
* arg (可选)

类似 ipcRenderer.send ，但是它的事件将发往 host page 的 <webview> 元素，而不是主进程.

IpcRenderer只能发消息给ipcMain，即只能被ipcMain注册的事件监听器监听到；反之，ipcMain发布的消息只能被ipcRenderer注册的事件监听器监听到