electron
中文文档

wangdashuaihenshuai

Published
with GitBook

目錄

vi. 构建步骤 (Windows)
vii. 构建步骤 (Linux)
viii. 在调试中使用 SymbolServer

electron-zh-document

electron 的翻译文档,直接从electron项目里面copy出来, 因为浏览跳页很麻烦,所以copy传入gitbook,更好阅读.

阅读和下载

点击这里阅读和下载

向导

教程

- 快速入门
- 桌面环境集成
- 在线/离线事件探测

快速入门

简介

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，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 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 BroswerWindow 实例创建网页。每个 BroswerWindow 实例都在自己的渲染进程里运行着一个网页。当一个 BroswerWindow 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，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 模块。并且也有一个远程进程调用风格的通讯模块 remote。

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：

```
var app = require('app');  // 控制应用生命周期的模块。
var BrowserWindow = require('browser-window');  // 创建原生浏览器窗口的模块

// 给我们的服务器发送异常报告。
require('crash-reporter').start();

// 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 完成了初始化并且准备创建浏览器窗口的时候
// 这个方法就被调用
app.on('ready', function() {
  // 创建浏览器窗口。
  mainWindow = new BrowserWindow({width: 800, height: 600});

  // 加载应用的 index.html
  mainWindow.loadUrl('file://' + __dirname + '/index.html');

  // 打开开发工具
  mainWindow.openDevTools();

  // 当 window 被关闭，这个事件会被发出
  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指导发布一个版本，并且以已经打包好的形式运行应用。

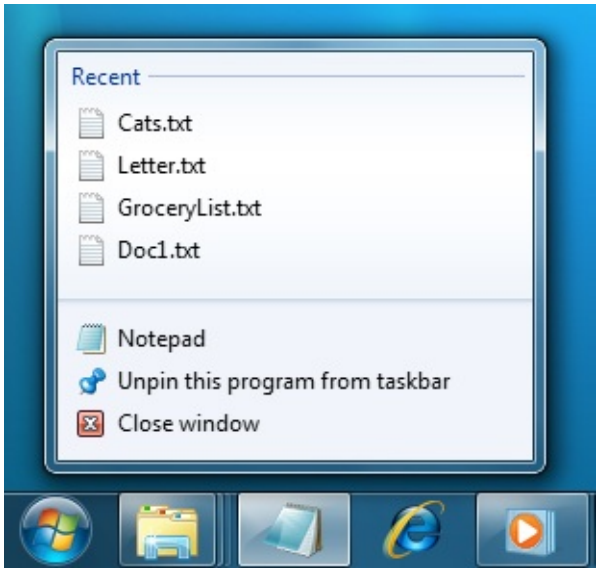
桌面环境集成

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。

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 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 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

Dock menu of Terminal.app

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

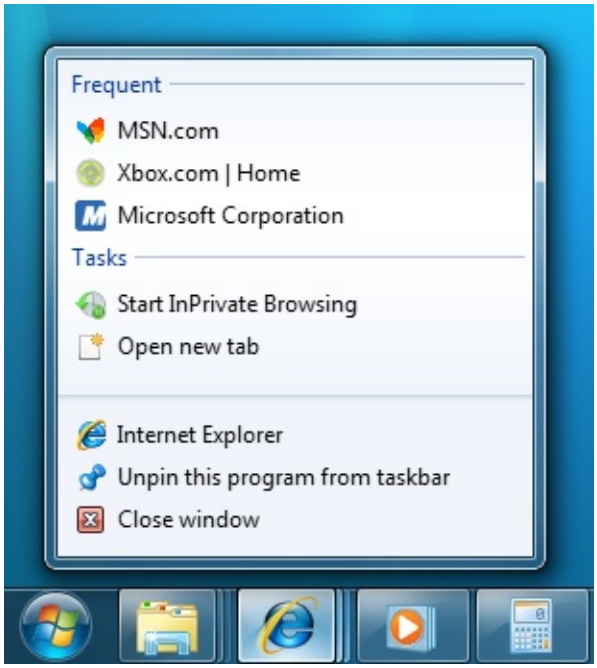
```
var app = require('app');
var Menu = require('menu');
var dockMenu = Menu.buildFromTemplate([
  { label: 'New Window', click: function() { console.log('New Window'); } },
  { label: 'New Window with Settings', submenu: [
    { label: 'Basic' },
    { label: 'Pro'}
  ]},
  { label: 'New Command...'}
]);
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 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 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."); }
  }
]);
```

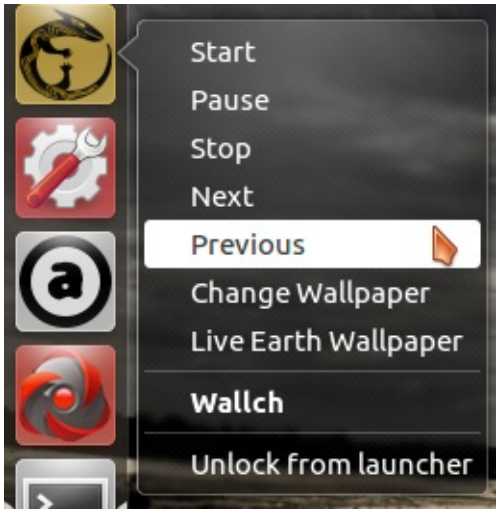
调用 `BrowserWindow.setThumbarButtons` 并传入空数组即可清空缩略图工具栏：

```
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。

Audacious 运行器的快捷方式：

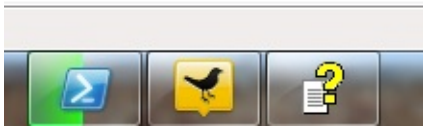


任务栏的进度条(Windows & Unity)

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：



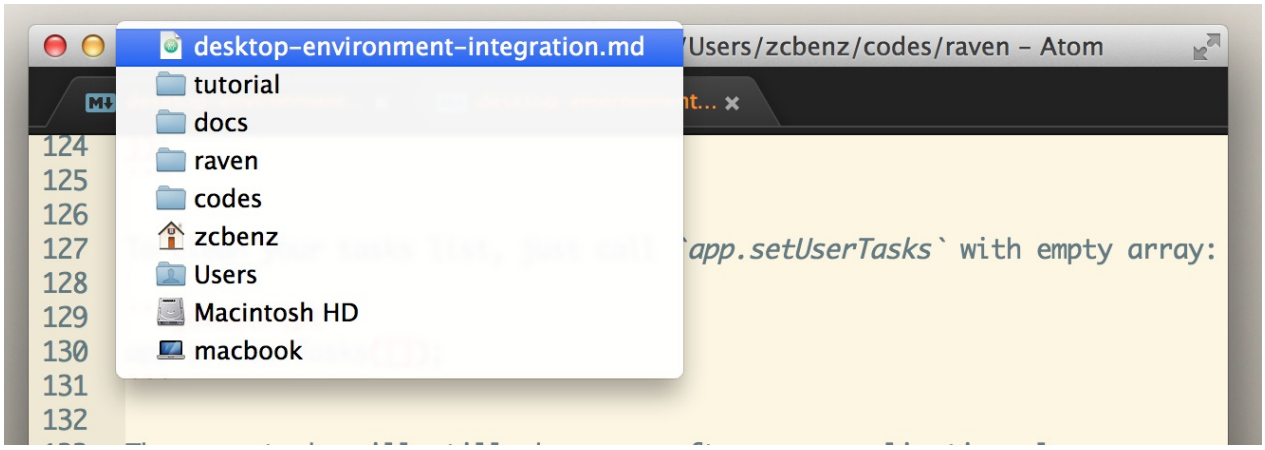
在 **Unity** 运行器上的进度条

给一个窗口设置进度条，你可以调用 BrowserWindow.setProgressBar 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 和 BrowserWindow.setDocumentEdited APIs：

```
var window = new BrowserWindow({...});
window.setRepresentedFilename('/etc/passwd');
window.setDocumentEdited(true);
```

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

```
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，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>
```

API文档

定制的DOM元素:

- `File` 对象
- `<webview>` 标签
- `window.open` 函数

主进程可用的模块:

- app
- auto-updater
- browser-window
- content-tracing
- dialog
- global-shortcut
- ipc (main process)
- menu
- menu-item
- power-monitor
- power-save-blocker
- protocol
- session
- webContents
- tray

渲染进程(网页)可用的模块:

- ipc (renderer)
- remote
- web-frame

两种进程都可用的模块:

- clipboard
- crash-reporter
- native-image
- screen
- shell

app

`app` 模块是为了控制整个应用的生命周期设计的。

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() {
  app.quit();
});
```

事件

`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` 事件

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` 事件

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'

当应用程序正在退出时触发。

事件： 'open-file'

返回：

- `event` 事件
- `path` 字符串

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()` 。

事件： 'open-url'

返回：

- `event` 事件
- `url` 字符串

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` 事件
- `hasVisibleWindows` 布尔值

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` 事件
- `window` 浏览器窗口

当一个 浏览器窗口 失去焦点的时候触发。

事件： 'browser-window-focus'

返回：

- `event` 事件
- `window` 浏览器窗口

当一个 浏览器窗口 获得焦点的时候触发。

事件： 'browser-window-created'

返回：

- `event` 事件
- `window` 浏览器窗口

当一个 浏览器窗口 被创建的时候触发。

事件： 'select-certificate'

当一个客户端认证被请求的时候被触发。

返回：

- `event` 事件
- `webContents` web组件
- `url` 字符串
- `certificateList` 对象
  - `data` PEM 编码数据
  - `issuerName` 发行者的公有名称
- `callback` 函数

```
app.on('select-certificate', function(event, host, url, list, callback) {
  event.preventDefault();
  callback(list[0]);
})
```

The `url` corresponds to the navigation entry requesting the client certificate and `callback` needs to be called with an entry filtered from the list. Using `event.preventDefault()` prevents the application from using the first certificate from the store.

事件： 'gpu-process-crashed'

当GPU进程崩溃时触发。

方法

`app` 对象拥有以下的方法：

提示: 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` 事件将会被最先触发。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。会存在一个窗口被 `beforeunload` 事件处理器返回 `false` 取消退出的可能性。

`app.getAppPath()`

返回当前应用所在的文件路径。

app.getPath(name)

- `name` 字符串

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。
- `appData` 所有用户的应用数据文件夹，默认对应：
  - `%APPDATA%` Windows 中
  - `$XDG_CONFIG_HOME` or `~/.config` Linux 中
  - `~/Library/Application Support` OS X 中
- `userData` 储存你应用程序设置文件的文件夹，默认是 `appData` 文件夹附加应用的名称。
- `cache` 所有用户应用程序缓存的文件夹，默认对应：
  - `%APPDATA%` Windows 中 (没有一个通用的缓存位置)
  - `$XDG_CACHE_HOME` 或 `~/.cache` Linux 中
  - `~/Library/Caches` OS X 中
- `userCache` 用于存放应用程序缓存的文件夹，默认是 `cache` 文件夹附加应用的名称。
- `temp` 临时文件夹。
- `userDesktop` 当前用户的桌面文件夹。
- `exe` 当前的可执行文件。
- `module` `libchromiumcontent` 库。

app.setPath(name, path)

- `name` 字符串
- `path` 字符串

重写 `path` 参数到一个特别的文件夹或者是一个和 `name` 参数有关系的文件。 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。 如果错误则抛出 `Error` 。

你只可以指向 `app.getPath` 中定义过 `name` 的路径。You can only override paths of a `name` defined in `app.getPath`.

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。

通常 `name` 字段是一个短的小写字符串，其命名规则按照 npm 中的模块命名规则。你应该单独列举一个 `productName` 字段，用于表示你的应用程序的完整名称，这个名称将会被 Electron 优先采用。

app.getLocale()

返回当前应用程序的位置。

app.resolveProxy(url, callback)

- `url` URL
- `callback` 函数

为 `url` 解析代理信息。 `callback` 在请求被执行之后将会被 `callback(proxy)` 调用。

`app.addRecentDocument(path)`

- `path` 字符串

为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()`

清除最近访问的文档列表。

`app.setUserTasks(tasks)` *Windows*

- `tasks` 由 `Task` 对象构成的数组

将 `tasks` 添加到 Windows 中 JumpList 功能的 Tasks 分类中。

`tasks` 中的 `Task` 对象格式如下：

`Task` 对象

- `program` 字符串 - 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` 字符串 - 当 `program` 执行时的命令行参数。
- `title` 字符串 - JumpList 中显示的标题。
- `description` 字符串 - 对这个任务的描述。
- `iconPath` 字符串 - JumpList 中显示的 icon 的绝对路径，可以是一个任意包含一个icon的资源文件。你通常可以通过指明 `process.execPath` 来显示程序中的icon。
- `iconIndex` 整数 - icon文件中的icon目录。如果一个icon文件包括了两个或多个icon，就需要设置这个值以确定icon。如果一个文件仅包含一个icon，那么这个值为0。

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

通过可选的参数 `value` 给 Chromium 命令行中添加一个开关。 Append a switch (with optional `value`) to Chromium's command line.

贴士: 这不会影响 `process.argv` ， 这个方法主要被开发者用于控制一些低层级的 Chromium 行为。

`app.commandLine.appendArgument(value)`

给 Chromium 命令行中加入一个参数。这个参数是当前正在被引用的。

贴士: 这不会影响 `process.argv` 。

`app.dock.bounce([type])` *OS X*

- `type` 字符串 (可选的) - 可以是 `critical` 或 `informational` 。默认下是 `informational`

当输入 `critical` 时，dock 中的 icon 将会开始弹跳直到应用被激活或者这个请求被取消。

当输入 `informational` 时，dock 中的 icon 只会弹跳一秒钟。 然而，这个请求仍然会激活，直到应用被激活或者请求被取消。

返回一个表示这个请求的 ID。

`app.dock.cancelBounce(id)` *OS X*

- `id` 整数

取消这个 `id` 对应的请求。

`app.dock.setBadge(text)` *OS X*

- `text` 字符串

设置 dock 中显示的字符。

`app.dock.getBadge()` *OS X*

返回 dock 中显示的字符。

`app.dock.hide()` *OS X*

隐藏 dock 中的 icon。

`app.dock.show()` *OS X*

显示 dock 中的 icon。

`app.dock.setMenu(menu)` *OS X*

- `menu` 菜单

设置应用的 dock 菜单.

global-shortcut

`global-shortcut` 模块可以便捷的为您设置(注册/注销)各种自定义操作的快捷键.

Note: 使用此模块注册的快捷键是系统全局的(QQ截图那种), 不要在应用模块(app module)响应 `ready` 消息前使用此模块(注册快捷键).

```
var app = require('app');
var globalShortcut = require('global-shortcut');

app.on('ready', function() {
  // Register a 'ctrl+x' shortcut listener.
  var ret = globalShortcut.register('ctrl+x', function() {
    console.log('ctrl+x is pressed');
  })

  if (!ret) {
    console.log('registration failed');
  }

  // Check whether a shortcut is registered.
  console.log(globalShortcut.isRegistered('ctrl+x'));
});

app.on('will-quit', function() {
  // Unregister a shortcut.
  globalShortcut.unregister('ctrl+x');

  // Unregister all shortcuts.
  globalShortcut.unregisterAll();
});
```

Methods

`global-shortcut` 模块包含以下函数:

`globalShortcut.register(accelerator, callback)`

- `accelerator` Accelerator
- `callback` Function

注册 `accelerator` 快捷键. 当用户按下注册的快捷键时将会调用 `callback` 函数.

`globalShortcut.isRegistered(accelerator)`

- `accelerator` Accelerator

查询 `accelerator` 快捷键是否已经被注册过了,将会返回 `true` (已被注册) 或 `false` (未注册).

`globalShortcut.unregister(accelerator)`

- `accelerator` Accelerator

注销全局快捷键 `accelerator`.

`globalShortcut.unregisterAll()`

注销本应用注册的所有全局快捷键.

ipc (主进程)

在主进程使用 `ipc` 模块时, `ipc` 负责捕获从渲染进程(网页)发送的同步或者是异步消息.

发送消息

主进程也可以向渲染进程发送信息,具体可以看WebContents.send.

- 当发送消息的时候,事件名字为'channel'.
- 回复一个同步消息的时候,你需要使用 `event.returnValue`
- 回复一个异步消息的时候,使用 `event.sender.send(...)`

下面是一个主进程和渲染进程的通信例子.

```
// 在主进程中.
var ipc = require('ipc');
ipc.on('asynchronous-message', function(event, arg) {
  console.log(arg);  // 打印 "ping"
  event.sender.send('asynchronous-reply', 'pong');
});

ipc.on('synchronous-message', function(event, arg) {
  console.log(arg);  // 打印 "ping"
  event.returnValue = 'pong';
});
```

```
// 在渲染进程(网页).
var ipc = require('ipc');
console.log(ipc.sendSync('synchronous-message', 'ping')); // 打印 "pong"

ipc.on('asynchronous-reply', function(arg) {
  console.log(arg); // 打印 "pong"
});
ipc.send('asynchronous-message', 'ping');
```

监听消息

`ipc` 模块有下列几种方法来监听事件.

`ipc.on(channel, callback)`

- `channel` - 事件名称.
- `callback` - 回调函数.

当事件发生的时候,会传入 `callback` `event` 和 `arg` 参数.

IPC 事件

传入 `callback` 的 `event` 对象含有下列方法.

`Event.returnValue`

在同步消息中,设置这个值将会被返回.

`Event.sender`

返回一个可以发送消息的 `WebContents` .

`Event.sender.send(channel[.arg1][,arg2][,...])`

- `channel` - 事件名称.
- `arg` (选用)

这个可以发送一个可带参数的异步消息回渲染进程.

shell

`shell` 模块提供了集成其他桌面客户端的关联功能.

在用户默认浏览器中打开URL的示例:

```
var shell = require('shell');

shell.openExternal('https://github.com');
```

Methods

`shell` 模块包含以下函数:

`shell.showItemInFolder(fullPath)`

- `fullPath` String

打开文件所在文件夹,一般情况下还会选中它.

`shell.openItem(fullPath)`

- `fullPath` String

以默认打开方式打开文件.

`shell.openExternal(url)`

- `url` String

以系统默认设置打开外部协议.(例如,mailto: somebody@somewhere.io会打开用户默认的邮件客户端)

`shell.moveItemToTrash(fullPath)`

- `fullPath` String

删除指定路径文件,并返回此操作的状态值(boolean类型).

`shell.beep()`

播放 beep 声音.

开发

编码规范

以下是 Electron 项目中代码书写规范的指导方针。

C++ 和 Python

对于 C++ 和 Python，我们追随 Chromium 的Coding Style。你可以通过 `script/cpplint.py` 来检验所有文件是否符合要求。

我们使用的 Pyhton 版本是 Python 2.7。

其中 C++ 代码中用到了许多 Chromium 的抽象和类型，我们希望你对其有所熟悉。一个好的去处是 Chromium 的重要的抽象和数据结构。这个文档提到了一些特殊的类型、域内类型（当超出作用域时会自动释放内存）、日志机制等等。

CoffeeScript

对于 CoffeeScript，我们追随 GitHub 的Style Guide 及如下规则:

- 文件不应该以换行结尾，因为我们要匹配 Google 的规范。
- 文件名应该以 `-` 作连接而不是 `_`，等等。 `file-name.coffee` 而不是 `file_name.coffee`，因为在 github/atom 模块名通常都是 `module-name` 的形式。这条规则仅应用于 `.coffee` 文件。

API 名称

当新建一个API时，我们应该倾向于 getters 和 setters 的方式，而不是 jQuery 的单函数形式。例如， `.getText()` 和 `.setText(text)` 优于 `.text([text])`。这里是相关的 讨论记录。

Electron 和 NW.js (原名 node-webkit) 在技术上的差异

备注：**Electron** 的原名是 **Atom Shell**。

与 NW.js 相似，Electron 提供了一个能通过 JavaScript 和 HTML 创建桌面应用的平台，同时集成 Node 来授予网页访问底层系统的权限。

但是这两个项目也有本质上的区别，使得 Electron 和 NW.js 成为两个相互独立的产品。

1. 应用的入口

在 NW.js 中，一个应用的主入口是一个页面。你在 `package.json` 中指定一个主页面，它会作为应用的主窗口被打开。

在 Electron 中，入口是一个 JavaScript 脚本。不同于直接提供一个URL，你需要手动创建一个浏览器窗口，然后通过 API 加载 HTML 文件。你还可以监听窗口事件，决定何时让应用退出。

Electron 的工作方式更像 Node.js 运行时。 Electron 的 APIs 更加底层，因此你可以它替代 PhantomJS 做浏览器测试。

2. 构建系统

为了避免构建整个 Chromium 带来的复杂度，Electron 通过 `libchromiumcontent` 来访问 Chromium 的 Content API。`libchromiumcontent` 是一个独立的、引入了 Chromium Content 模块及其所有依赖的共享库。用户不需要一个强劲的机器来构建 Electron。

3. Node 集成

在 NW.js，网页中的 Node 集成需要通过给 Chromium 打补丁来实现。但在 Electron 中，我们选择了另一种方式：通过各个平台的消息循环与 libuv 的循环集成，避免了直接在 Chromium 上做改动。你可以看 `node_bindings` 来了解这是如何完成的。

4. 多上下文

如果你是有经验的 NW.js 用户，你应该会熟悉 Node 上下文和 web 上下文的概念。这些概念的产生源于 NW.js 的实现方式。

通过使用 Node 的多上下文特性，Electron不需要在网页中引入新的 JavaScript 上下文。