Electron入门与原理介绍

传统桌面应用开发

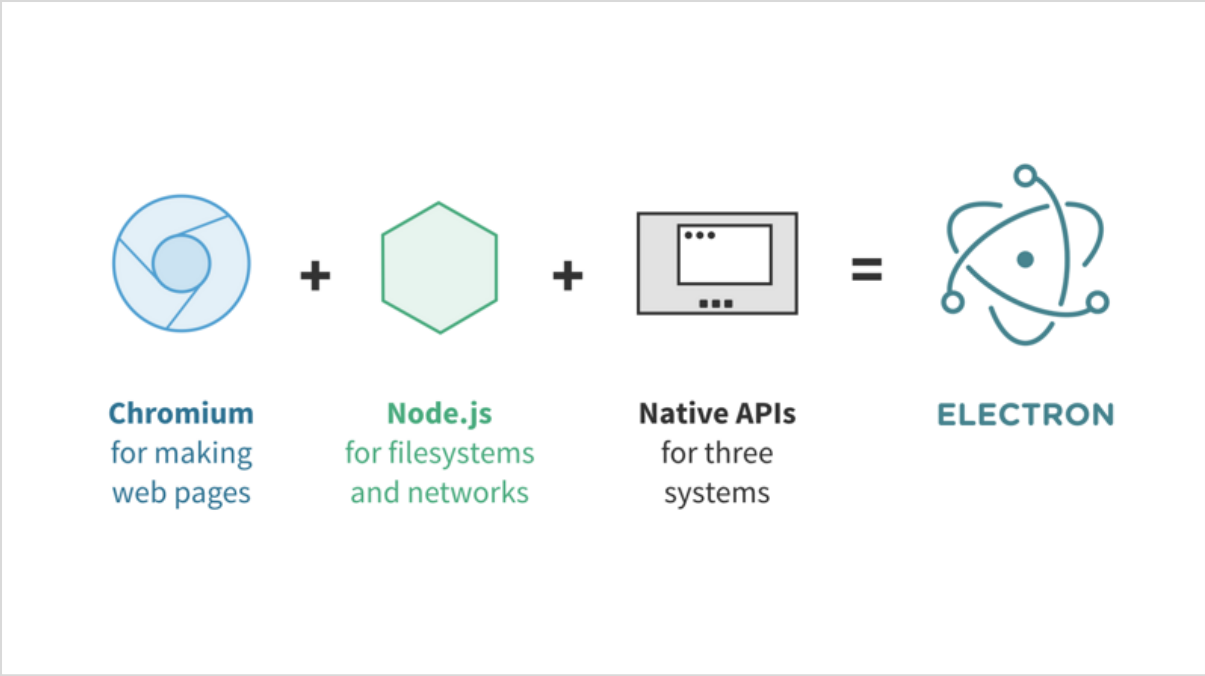
- windows应用 c++ / MFC
- Mac应用 Objective-C

直接将代码编译成可执行文件，直接调用系统API，这种原生的模式普遍的特点都是：

1. 性能好，体验好
2. 开发慢，维护难

Electron

Electron是由Github开发，用HTML，CSS和JavaScript来构建跨平台桌面应用程序的一个开源库。
Electron通过将Chromium和Node.js合并到同一个运行时环境中，并将其打包为Mac，Windows和Linux系统下的应用来实现这一目的。



咱们可以先来总结一波Electron的优点：

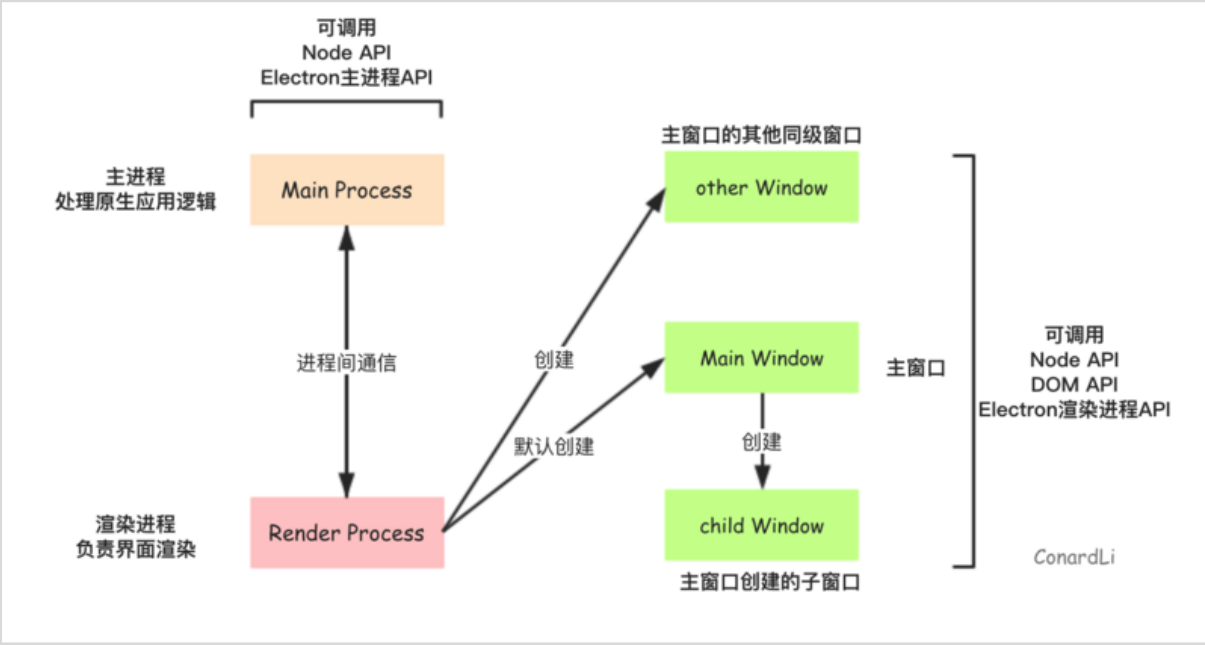
1. 使用web技术开发，成本低，在社区的帮助下，UI更丰富

2. 跨平台，一套代码可以打包windows/Mac/linux三套平台
3. 可以直接在现有web应用上拓展
4. 对前端来说，非常友好
5. 直接使用最新版本Chromium，无需再考虑浏览器兼容性
6. 可以直接使用Node.js，直接使用各种前端工具比如webpack, npm等

VScode就是使用Electron开发的，感兴趣的同学可以去看一下vscode的项目，或者打开"帮助" ➜ "开发者模式"，会有惊喜。

进程

Electron区分了两种进程：主进程和渲染进程



1. 主进程

Electron 运行 package.json 的 main 脚本的进程被称为主进程。一个 Electron 应用总是有且只有一个主进程。

职责:

创建渲染进程（可多个）
控制了应用生命周期（启动、退出APP以及对APP做一些事件监听）
调用系统底层功能、调用原生资源
可调用的API:

Node.js API
Electron提供的主进程API（包括一些系统功能和Electron附加功能）

2. 渲染进程

由于 Electron 使用了 Chromium 来展示 web 页面，所以 Chromium 的多进程架构也被使用到。 每个 Electron 中的 web页面运行在它自己的渲染进程中。

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。

你可以把渲染进程想像成一个浏览器窗口，它能存在多个并且相互独立，不过和浏览器不同的是，它能调用Node API。

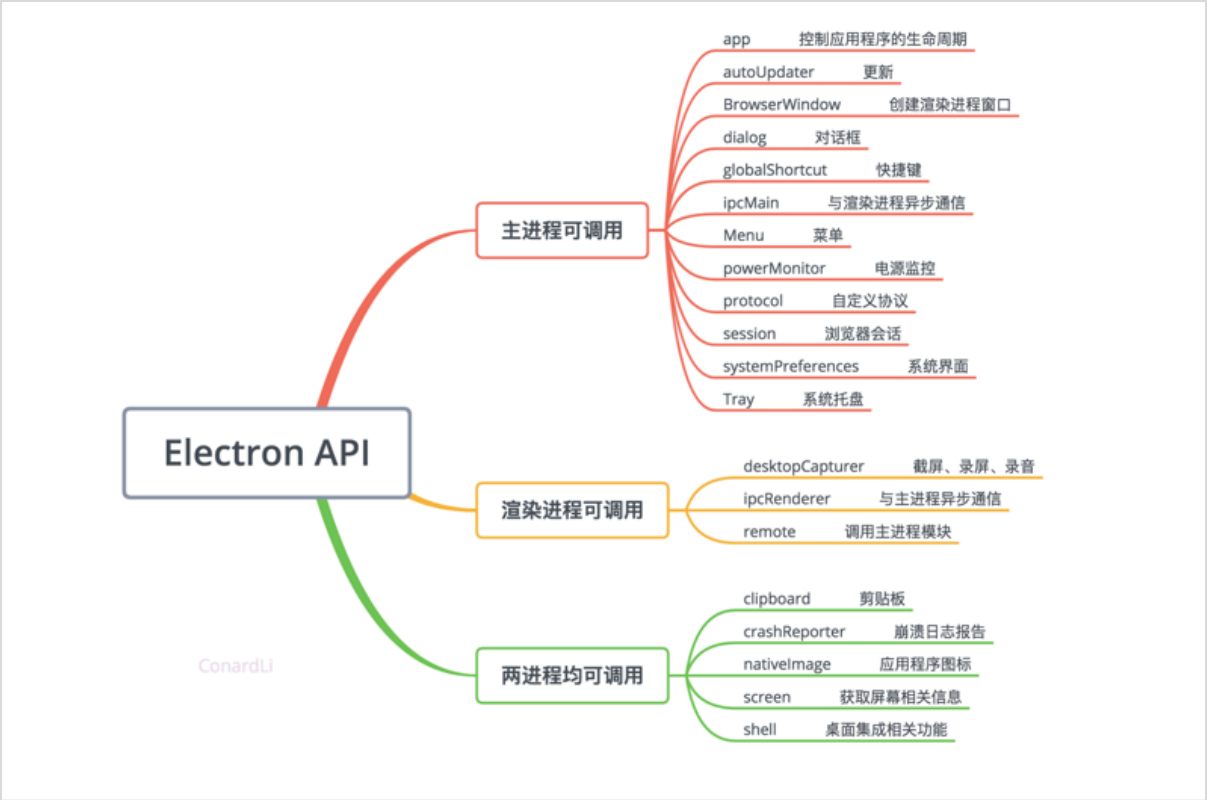
职责：

用HTML和CSS渲染界面
用JavaScript做一些界面交互

可调用的API:

DOM API
Node.js API
Electron提供的渲染进程API

Api



进程间通信

ipcMain 和 ipcRenderer 都是 EventEmitter 类的一个实例。EventEmitter 类是 NodeJS 事件的基础，它由 NodeJS 中的 events 模块导出。

EventEmitter 的核心就是事件触发与事件监听器功能的封装。它实现了事件模型需要的接口，包括 addListener，removeListener, emit 及其它工具方法. 同原生 JavaScript 事件类似， 采用了发布/订阅(观察者)的方式， 使用内部 _events 列表来记录注册的事件处理器。

我们通过 ipcMain和ipcRenderer 的 on、send 进行监听和发送消息都是 EventEmitter 定义的相关接口。

https://www.electronjs.org/docs/api/ipc-main

```
// 主进程。
const { ipcMain } = require('electron')
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.reply('asynchronous-reply', 'pong')
```

```
})

ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.returnValue = 'pong'
})

// 渲染进程
const { ipcRenderer } = require('electron')
console.log(ipcRenderer.sendSync('synchronous-message', 'ping')) // prints
"pong"

ipcRenderer.on('asynchronous-reply', (event, arg) => {
  console.log(arg) // 印出 "pong"
})
ipcRenderer.send('asynchronous-message', 'ping')
```

remote模块

https://www.electronjs.org/docs/api/remote

Remote 模块为渲染进程和主进程通信提供了一种简单方法。
在Electron中, GUI 相关的模块 (如 dialog、menu 等) 仅在主进程中可用, 在渲染进程中不可用。 为了在渲染进程中使用它们, ipc 模块是向主进程发送进程间消息所必需的。 使用 remote 模块, 你可以调用 main 进程对象的方法, 而不必显式发送进程间消息.

```
// 主进程
global.sharedObject = {
  someProperty: 'default value'
}

// 渲染进程1
require('electron').remote.getGlobal('sharedObject').someProperty = 'new value'

// 渲染进程2
```

```
console.log(require('electron').remote.getGlobal('sharedObject').someProperty)
```

不同页面间数据共享

Storage/ 进程通信/ 主进程global变量+remote

实战

实战部分会引导大家一起实现一个剪贴板应用~