小论文 1.0版本

基于Chromium和Node的桌面跨端解决方案原理

Sections

* 前言 叙述关于跨端的发展、桌面跨端的历史、Web跨端的解决方案的迭代、桌面开发的注意事项
* 架构 Electron整体架构和流程模型
* Chromium、Node 分别介绍两大基础，涉及Chromium的多进程架构、资源请求、IO; Node的事件循环、C++和Js的相互调用、基础模块API
* 结论 Electron的缺点、改进、展望
* 参考文献

## Section 1

什么是跨平台呢？我们常用的应用软件都是运行在操作系统之上，而操作系统同样运行在不同的CPU架构和指令集上，因此通常Windows操作系统上的软件无法在Linux或者MacOs操作系统上运行，反之亦然。所以在基于不同CPU架构和指令集的不同操作系统上开发软件，如果目标客户是多操作系统的人群，那么就需要针对不同的人群去开发平台所适应的软件。

通常如果不考虑跨平台开发，那我们需要维护Windows、MacOS、Linux三个主要PC端操作系统的版本迭代，多平台开发带来的问题就是一致性差异、资源成本的增加、维护成本等。所以跨平台的解决方案就是通过一份代码，通过编译、底层屏蔽等手段，可以跑在任意目标的操作系统上，已达到解决多平台开发带来的问题。同时跨平台的解决方案的终极目标是为了实现跨平台的同时，也在不同平台实现原生开发的运行效率。

首先，常见的跨平台应用有JVM、Node、Electron，其解决方案都是多了一层抽象层来屏蔽掉上层对操作系统的感知，从而实现跨平台的目标。

#### 代码签名？

## Section 2

Electron是一个在2013年开源的基于Chromium和NodeJs的跨操作系统的桌面应用开发框架。基于Electron开发的成功的应用程序有很多，比如VSCode 开发编辑器、Atom 文本编辑器、Messenger Facebook开发的信息软件等等。

Electron是基于nw.js发展而来的，

Electron在进程模型上参考了Chromium引擎的多进程模型，通过主进程和多个渲染进程组成。

每一个Electron应用都有且仅有一个单一的主进程，作为应用程序的入口。主进程运行在Node.js环境中，它可以利用Node.js丰富的API和强大的可扩展性，以及对底层操作系统的API封装，相当于建立在Node跨平台上的跨平台。而渲染进程同样运行在Node环境。

### 主进程

* 窗口管理

使用BrowerWindow模块创建和管理应用程序窗口。BrowerWindow类的每个实例，创建一个应用程序窗口，且在单独的渲染器进程中加载网页。BrowerWindow模块是一个EventEmitter，所以可以创建事件监听用来反馈用户的操作，比如缩放屏幕、全屏、响应快捷键等等。当一个BrowserWindow实例被销毁，与其对应的渲染器进程也会终止。

* 应用生命周期

使用App模块控制应用程序的生命周期，可以添加自定义的应用程序行为，比如修改程序坞、关于面板、菜单等等

* 原生API

多种控制原生桌面功能的模块，比如菜单、对话框、托盘图标等。

### 渲染进程

Electron会根据应用打开的BrowserWindow生成一个相对应的渲染器进程，专门用来渲染网页内容，而网页内容的渲染和交互逻辑都可以通过常规的网页开发工具和模式来编写。但是渲染器线程没办法访问操作系统的原生APi或者Node提供的底层API。

### 预加载脚本

preload脚本包含了执行在渲染器进程中，但是先于网页内容的渲染，这些脚本虽然运行在渲染器进程，但是可以访问主进程中的所有功能，所以我们可以通过在预加载脚本中定义消息通知的函数，通过这种方式来实现渲染进程对主进程的通知，从而进一步调用原生功能。

虽然预加载脚本运行在渲染器进程，共享同一个GlobalThis或者Window全局对象，但是为了安全，Electron做了语境隔离，意味着预加载脚本于渲染器的主要运行环境是隔离开的，以避免主进程中的功能随意污染渲染进程的环境。取而代之，Electron通过contextBridge模块来安全地实现交互。

// preload.js  
const { contextBridge } = reuqire('electron');  
  
contextBridge.exposeInMainWorld('myAPI', {  
 os: 'windows'  
});  
  
  
// render.js  
cosnole.log(window.os)  
// { os: 'windows' }

这种方式，也为了渲染器进程暴露了IpcRenderer模块，使用进程间通讯（interprocess communication）来渲染器触发主进程任务（反之亦然）。

#### 沙盒化进程

进程可以在沙盒中执行。 沙盒通过限制对大多数系统资源的访问来减少恶意代码可能造成的伤害 — 沙盒化的进程只能自由使用CPU周期和内存。 为了执行需要额外权限的操作，沙盒处的进程通过专用通信渠道将任务下放给更大权限的进程。

Electron带有一个混合的沙盒环境，意味着沙盒化进程可以和有权限的进程一起运行。 默认情况下，渲染器进程未被沙盒化，但功能性进程是被沙盒化的。当 Electron 中的渲染器进程被沙盒化时，它们的行为与常规 Chrome 渲染器一样。 一个沙盒化的渲染器不会有一个 Node.js 环境。

因此，在沙盒中，渲染器进程只能透过 进程间通讯 ( inter-process communication, IPC ) 委派任务给主进程的方式， 来执行需权限的任务 ( 例如：操作档案系统，改变作业系统 或 生成子进程 ) 。

### 兼容不同操作系统（section 1 感觉也可以添加此内容）

首先针对不同操作系统可以统一的API，Electron会根据封装一层，供上游调用。但是如果出现很难统一封装的操作，那就需要针对性的写差异化的代码来分别处理。

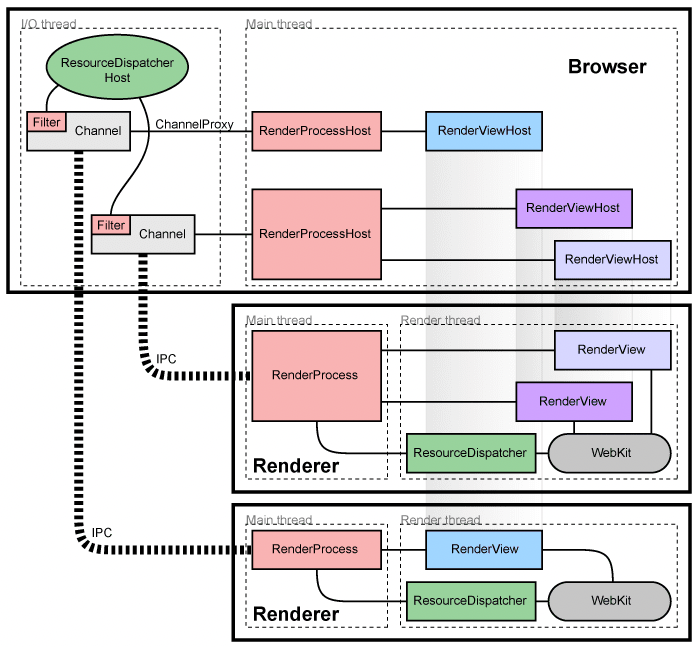
比如MacOS的关闭可能只是页面上的关闭，而Windows的关闭是直接退出

在 MacOS 和 Ubuntu, 托盘将位于屏幕右上角上，靠近你的电池和wifi 图标。 在 Windows 上，托盘通常位于右下角。

## Section 3

Electron是基于Chromium和Node之上的跨平台解决方案，它借鉴和利用了这两个底层的主要模块，而无论是chromium还是Node，又是内置了V8 javaScript执行引擎来对运行JS代码，保障了运行效率和性能。

### Chromium



* 多进程架构

浏览器主进程  
插件进程  
GPU进程  
渲染进程

Electron借鉴了Chromium的多进程架构，为什么要多进程架构呢？多进程架构有很多缺点，比如进程之间的通讯、进程切换上下文保存、切换效率都要远低于线程。但是也正是如此，如果所有窗口都在一个进程内，那共享栈内存，如果浏览器中任何一个页面或插件的错误就可能让整个浏览器崩溃，这种绑定关系严重威胁到浏览器使用的隔离性，所以采用多进程的架构模型。

同时把渲染进程，也就是用户或者开发者的代码及脚本运行的环境，单独形成一个沙盒环境，这里所有涉及网络安全和本地存储等有安全风险的模块，都通过IPC转发到浏览器主进程中处理。而每一个渲染进程中，也有一些线程，比如IO线程和主线程，

Chromium也做了类似的设计，它把每个页面约束在单独的进程中，以保护整个浏览器不受单个页面中的故障影响。它甚至还限制了每个页面进程对其他进程和系统其他部分的访问，这极大地缓解了浏览器容易崩溃的问题。Chromium把管理页面、管理选项卡和插件的进程称为主进程。把特定于页面的进程称为渲染进程。

浏览器多进程的优势

* 避免单个page crash影响整个浏览器
* 避免第三方插件crash影响整个浏览器
* 多进程充分利用多核优势
* 方便使用沙盒模型隔离插件等进程，提高浏览器稳定性

简单理解就是：如果浏览器是单进程的，某个Tab页崩溃了，就影响了整个浏览器，体验就会很差。同理如果是单进程的，插件崩溃了也会影响整个浏览器;  
当然，内存等资源消耗也会更大，像空间换时间一样。

* 浏览器主进程

负责协调、主控。负责浏览器各个界面的管理、创建与销毁、网络资源的管理、下载

* 渲染进程

渲染进程是浏览器的主要部分，负责所有页面中的网络资源、本地存储、渲染层级树等等多个线程，通过这些线程共同支撑页面与用户的交互过程。

（1）GUI线程

（2）JS引擎线程 V8

浏览器环境和NodeJS环境只不过是维护了一套Javascript运行时能力，并赋予了一些平台的接口(对于nodejs来说是libuv模块)，就拿setTimeOut和事件 队列来说，他们都属于运行时提供的，并不属于js引擎范畴内。

### Node

NodeJs是基于高性能V8的JavaScript执行引擎的一个服务端框架，融合了很多基础库，构建了一个单进程异步。是一个集web服务器和应用服务器与一身的服务器。

Node基于V8的JS运行时，它利用V8提供的能力，极大地拓展了JS的能力，这种拓展不是为了JS增加了新的语言特性，而是拓展了功能模块。例如JS本身并没有网络相关的功能，但是通过Node扩展，可以让我们通过JS调用底层的C++模块的TCP，从而赋予网络编程的能力。

Node.js 中最核心的部分是 Libuv 和 V8，V8 不仅负责执 行 JS，还支持自定义的拓展，实现了 JS 调用 C++和 C++调用 JS 的能力。JS 层调用的所有 C、C++模块都是通过 V8 来完成的。

Node的架构底层是一些c++的基础库、libuv、SSL、c-cares等等，赋予了Node高效的系统层api。在C++和JavaScript中间定义了相互调用的胶水代码，同时利用JavaScript构建了一套包加载机制，赋予了JS项目级别的管理能力。

Node的架构底层是一些c++的基础库、libuv、SSL、c-cares等等，赋予了Node高效的系统层api。在C++和JavaScript中间定义了相互调用的胶水代码，同时利用JavaScript构建了一套包加载机制，赋予了JS项目级别的管理能力。

# 参考文献

张佳伟,张涛,周叶. 基于Electron的跨平台客户端技术[J]. 智能计算机与应用,2017,7(3):120-122.