# 基于Chromium和Node的桌面跨平台解决方案原理

1. 前言

**1.1 跨平台应用程序开发**

桌面应用程序，又称GUI程序，是互联网早期最重要的C/S架构软件产品主流。但随着浏览器的发展C/S架构越来越多的被B/S架构所取代，虽然C/S架构的响应速度、运行效率都是优于B/S架构，但是其开发效率、迭代效率、观赏性等上都远远不如B/S架构。所以基于这些目标，桌面应用程序的开发趋势是融合两者所长，形成跨平台桌面应用程序开发的解决方案。

什么是跨平台？桌面应用软件都是运行在操作系统之上，而操作系统运行在不同的CPU架构和指令集上，因此通常Windows操作系统上的软件无法直接在Linux或者MacOS操作系统上安装运行，反之亦然。桌面应用软件的目标使用人群本身就是多操作系统，所以就需要针对不同的CPU架构版本等不同操作系统都开发相对应的软件版本。通常在跨平台应用软件开发出现之前，针对这类情况至少要维护Windows、Linux、MacOS三个主要的PC端操作系统的版本迭代。显而易见，这种软件开发方式带来的问题是，不同操作系统的软件一致性问题、开发成本、维护成本问题、开发周期迭代问题等等。

而跨平台软件开发的解决方案通过只开发一份代码，通过底层编译、架构的设计等手段实现一份代码可以编译成任意目标操作系统上的可执行文件，从而达到解决多平台开发带来的诸多问题。而跨平台软件开发方案的终极目标是为了实现跨平台目的的同时，也可以达到原生的不同平台开发的运行效率和原生体验。

常见的跨平台实现方案有JVM、Node、Electron等，这些都是针对不同的场景所设计，不过这些方案都有一个重要的通性就是都是通过增加一层抽象层去处理跨平台，对上层提供统一的API来屏蔽掉上层对操作系统的感知，从而实现跨平台的目标。

**1.2 基于Web的跨平台解决方案**

如图表1，是目前针对桌面应用跨平台开发的主要解决方案。

首先QT，是一个GUI的框架，支持跨平台，易移植，语法结构简单清晰，相比较原生开发更加容易简单。而且QT不仅仅支持C++，同时也支持Python等。但是QT学习成本比较大，涉及到协议、QML等，开发周期也会比较长。但是其开源、丰富的UI库和文档生态，也是很多桌面客户端的首选。

以Java为首的JavaFx、Swing也是一类比较重要的开发模式，优势是和Java的天然结合，但是其生态较差，并且Java运行时还要通过JVM来管理和维护Java类对象的内存分配，性能上不如C++等，开发周期又不如Web，同时可用的组件也相对较少。

本小节，主要是介绍基于Web的跨平台桌面应用开发方案。

|  |  |  |
| --- | --- | --- |
| 框架/类库 | 语言/环境 | 适用平台 |
| QT | C/C++ | 跨平台 |
| Swing/JavaFx/SWT | Java | 跨平台 |
| NW.JS | JavaScript | 跨平台 |
| Electron | JavaScript | 跨平台 |

表1 桌面应用开发的跨平台解决方案

B/S架构的兴起离不开Web的发展，Web技术开发桌面应用程序，从早期的node-webkit到NW.js再到如今的Electron，原理上基本都是基于node-runtime、chromium-runtime将用户的HTML、CSS、JavaScript渲染执行。目前很多桌面应用都是基于Electron框架开发，例如Visual Studio Code程序编辑器、Atom文本编辑器、WordPress等等。随着Chrome V8 javaScript执行引擎的出现，提高的JavaScript的执行性能，让JavaScript可以承担更复杂的应用开发。

Electron是一个基于Web构建桌面应用程序的底层工具框架。 它允许使用 Node.js 和Chromium (V8 引擎内核库)完成桌面 GUI 应用程序的开发。通过嵌入Chromium 和 Node.js 到二进制的 Electron 可以构建跨平台桌面程序。为了提供原生系统的GUI支持，Electron内置了原生应用程序接口，对调用一些系统功能，如调用系统通知、打开系统文件夹提供支持。

相比较原生C++等原生开发框架、QT等跨平台框架来说，Web技术跨平台桌面应用开发带来的是更加丰富的组件、更加灵活的技术架构、更快的开发周期以及更加繁荣的生态环境。但是其相比较原生，其系统性能和打包体积是其主要瓶颈。

**1.3 桌面软件**

桌面软件开发周期除了开发以外还包含那些细节？

代码签名

1. Electron

Electron是一个在2013年开源的基于Chromium和Node.js的跨操作系统的桌面应用开发框架。Electron结合了基于V8引擎的轻量浏览器内核Chromium和NodeJs丰富强大的系统层面的接口，高效地利用了操作系统的能力，使得可以通过JavaScript来创建跨平台的桌面应用。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 ) 。

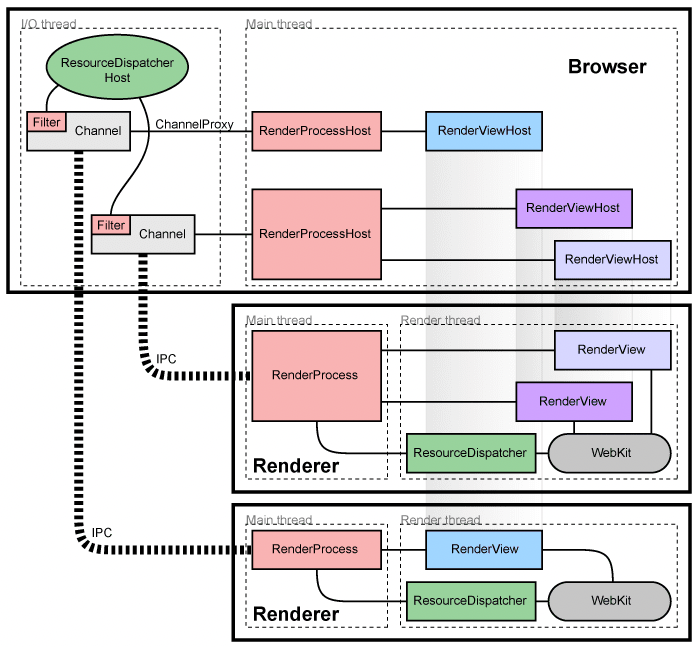
### 兼容不同操作系统（section 1 感觉也可以添加此内容）

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。

1. Chromium

3.1 多进程架构

浏览器主进程  
插件进程  
GPU进程  
渲染进程

Electron的架构模型是借鉴了Chromium浏览器内核引擎的多进程架构。多进程架构的缺点是进程之间的通讯、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。

**3.2 浏览器进程**

**3.3 渲染进程**

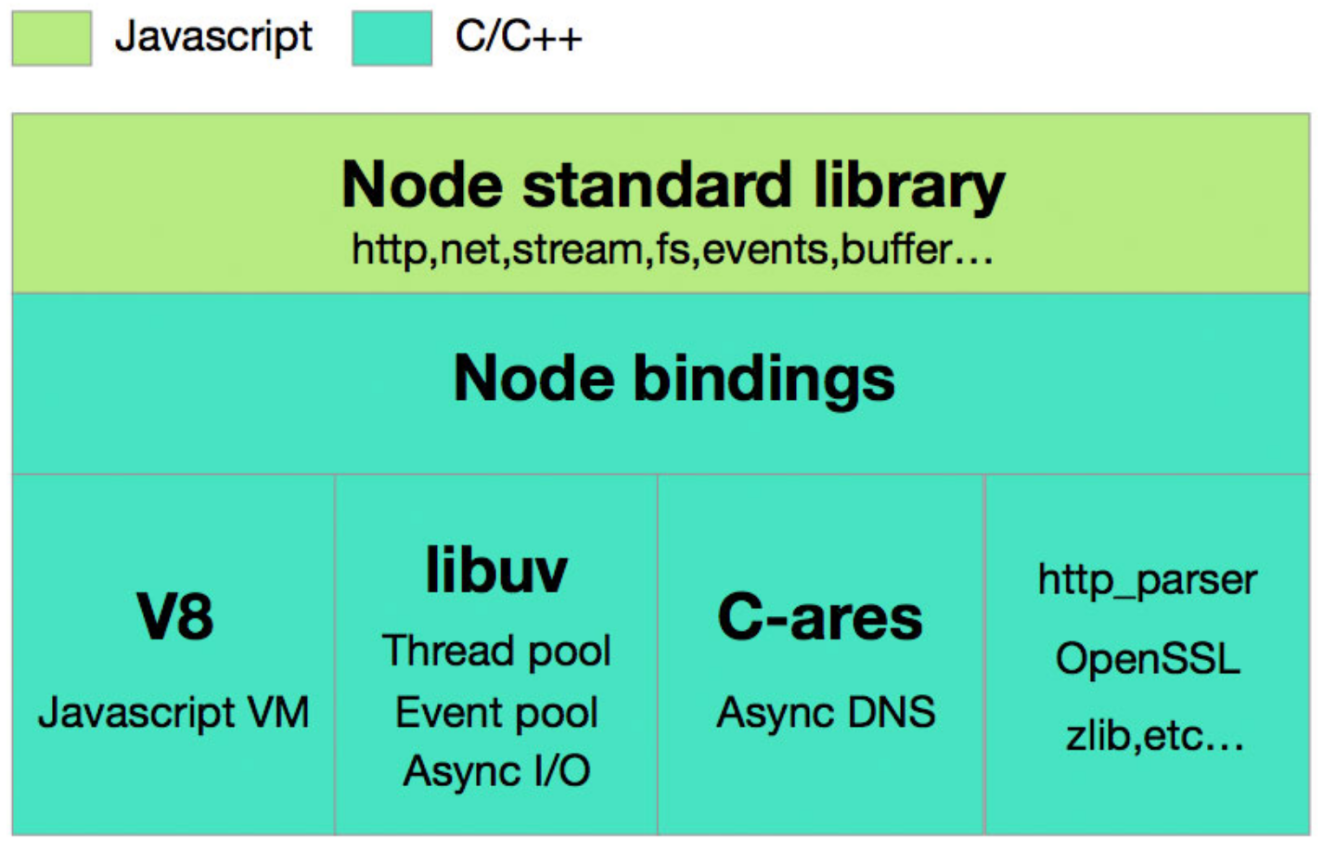
1. Node

Node是基于高性能V8的JavaScript执行引擎的一个服务端框架，还依赖了众多C++基础库，例如Libuv跨平台异步IO库、OpenSSL 网络密码库、c-ares异步DNS解析库等等，构建了一个单进程异步的高性能的JavaScript运行时框架，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JavaScript运行时，它利用V8提供的能力，极大地拓展了JS的能力，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。

这种拓展不是为了JavaScript增加了新的语言特性，而是拓展了功能模块。例如JavaScript本身并没有网络相关的功能，但是通过Node扩展，可以赋予通过JavaScript调用底层的C++模块的TCP，从而拥有了网络编程的能力。

4.1 架构模型



如图所示，Node standard library是直接暴露给用户使用的接口模块，有的是纯JavaScript语言，有的是依赖的底层C++基础库封装功能。Node bindings 是沟通JavaScript和C++之间的桥梁，封装了底层V8、Libuv等库的功能，向Node standard library层提供基础的API服务。而最底层是由众多C++库依赖组成的Node内置基础模块，是Node运行时的根本以及拥有众多能力的基石。

Node bindings层有大量的对于\_\_linux\_\_、\_\_POSIX\_\_、\_WIN32等关于操作系统判断的变量，而做的特殊化处理的工作。

**4.2 事件循环**

**4.3 C++ 和 JavaScript 交互**

本小节主要来讲讲如何通过 Node 如何实现 JS 调用 C++。JS 调用 C++可以分为 JS 调用 C++ 函数（全局），和调用 C++ 类。

### **数据及模板**

由于 C++ 原生数据类型与 JavaScript 中数据类型有很大差异，因此 V8 提供了 Value 类，从 JavaScript 到 C++，从 C++ 到 JavaScrpt 都会用到这个类及其子类，比如：

Handle<Value> Add(const Arguments& args){

int a = args[0]->Uint32Value();

int b = args[1]->Uint32Value();

return Integer::New(a+b);

}

Integer 即为 Value 的一个子类。

V8 中，有两个模板 (Template) 类 (并非 C++ 中的模板类)：

* 对象模板 (ObjectTemplate)
* 函数模板 (FunctionTemplate) 这两个模板类用以定义 JavaScript 对象和 JavaScript 函数。我们在后续的小节部分将会接触到模板类的实例。通过使用 ObjectTemplate，可以将 C++ 中的对象暴露给脚本环境，类似的，FunctionTemplate 用以将 C++ 函数暴露给脚本环境，以供脚本使用。

### **JS 使用 C++ 变量**

在 JavaScript 与 V8 间共享变量事实上是非常容易的，基本模板如下：

static char sname[512] = {0};

static Handle<Value> NameGetter(Local<String> name, const AccessorInfo& info) {

return String::New((char\*)&sname,strlen((char\*)&sname));

}

static void NameSetter(Local<String> name, Local<Value> value, const AccessorInfo& info) {

Local<String> str = value->ToString();

str->WriteAscii((char\*)&sname);

}

定义了 NameGetter, NameSetter 之后，在 main 函数中，将其注册在 global 上：

// Create a template for the global object.

Handle<ObjectTemplate> global = ObjectTemplate::New();

//public the name variable to script

global->SetAccessor(String::New("name"), NameGetter, NameSetter);

### **JS 调用 C++ 函数**

在 JavaScript 中调用 C++ 函数是脚本化最常见的方式，通过使用 C++ 函数，可以极大程度的增强 JavaScript 脚本的能力，如文件读写，网络 / 数据库访问，图形 / 图像处理等等，类似于 JAVA 的 jni 技术。

在 C++ 代码中，定义以下原型的函数：

Handle<Value> func(const Arguments& args){//return something}

然后，再将其公开给脚本： global->Set(String::New("func"),FunctionTemplate::New(func));

### **JS 使用 C++ 类**

如果从面向对象的视角来分析，最合理的方式是将 C++ 类公开给 JavaScript，这样可以将 JavaScript 内置的对象数量大大增加，从而尽可能少的使用宿主语言，而更大的利用动态语言的灵活性和扩展性。事实上，C++ 语言概念众多，内容繁复，学习曲线较 JavaScript 远为陡峭。最好的应用场景是：既有脚本语言的灵活性， 又有 C/C++ 等系统语言的效率。使用 V8 引擎，可以很方便的将 C++ 类” 包装” 成可供 JavaScript 使用的资源。

我们这里举一个较为简单的例子，定义一个 Person 类，然后将这个类包装并暴露给 JavaScript 脚本，在脚本中新建 Person 类的对象，使用 Person 对象的方法。 首先，我们在 C++ 中定义好类 Person：

class Person {

private:

unsigned int age;

char name[512];

public:

Person(unsigned int age, char \*name) {

this->age = age;

strncpy(this->name, name, sizeof(this->name));

}

unsigned int getAge() {

return this->age;

}

void setAge(unsigned int nage) {

this->age = nage;

}

char \*getName() {

return this->name;

}

void setName(char \*nname) {

strncpy(this->name, nname, sizeof(this->name));

}

};

Person 类的结构很简单，只包含两个字段 age 和 name，并定义了各自的 getter/setter. 然后我们来定义构造器的包装：

Handle<Value> PersonConstructor(const Arguments& args){

Handle<Object> object = args.This();

HandleScope handle\_scope;

int age = args[0]->Uint32Value();

String::Utf8Value str(args[1]);

char\* name = ToCString(str);

Person \*person = new Person(age, name);

object->SetInternalField(0, External::New(person));

return object;

}

从函数原型上可以看出，构造器的包装与上一小节中，函数的包装是一致的，因为构造函数在 V8 看来，也是一个函数。需要注意的是， 从 args 中获取参数并转换为合适的类型之后，我们根据此参数来调用 Person 类实际的构造函数，并将其设置在 object 的内部字段中。紧接着，我们需要包装 Person 类的 getter/setter：

Handle<Value> PersonGetAge(const Arguments& args){

Local<Object> self = args.Holder();

Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));

void \*ptr = wrap->Value();

return Integer::New(static\_cast<Person\*>(ptr)->getAge());

}

Handle<Value> PersonSetAge(const Arguments& args) {

Local<Object> self = args.Holder();

Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));

void\* ptr = wrap->Value();

static\_cast<Person\*>(ptr)->setAge(args[0]->Uint32Value());

return Undefined();

}

而 getName 和 setName 的与上例类似。在对函数包装完成之后，需要将 Person 类暴露给脚本环境： 首先，创建一个新的函数模板，将其与字符串”Person” 绑定，并放入 global：

Handle<FunctionTemplate> person\_template = FunctionTemplate::New(PersonConstructor);

person\_template->SetClassName(String::New("Person"));

global->Set(String::New("Person"), person\_template);

然后定义原型模板：

Handle<ObjectTemplate> person\_proto = person\_template->PrototypeTemplate();

person\_proto->Set("getAge", FunctionTemplate::New(PersonGetAge));

person\_proto->Set("setAge", FunctionTemplate::New(PersonSetAge));

person\_proto->Set("getName", FunctionTemplate::New(PersonGetName));

person\_proto->Set("setName", FunctionTemplate::New(PersonSetName));

最后设置实例模板：

Handle<ObjectTemplate> person\_inst = person\_template->InstanceTemplate();

person\_inst->SetInternalFieldCount(1);

### **C++ 调用 JS 函数**

我们直接看下 src/timer\_wrap.cc 的例子，V8 编译执行 timer.js, 构造了 Timer 对象。

static void OnTimeout(uv\_timer\_t\* handle) {

TimerWrap\* wrap = static\_cast<TimerWrap\*>(handle->data);

Environment\* env = wrap->env();

HandleScope handle\_scope(env->isolate());

Context::Scope context\_scope(env->context());

wrap->MakeCallback(kOnTimeout, 0, nullptr);

}

inline v8::Local<v8::Value> AsyncWrap::MakeCallback(uint32\_t index, int argc, v8::Local<v8::Value>\* argv) {

v8::Local<v8::Value> cb\_v = object()->Get(index);

CHECK(cb\_v->IsFunction());

return MakeCallback(cb\_v.As<v8::Function>(), argc, argv);

}

TimerWrap 对象通过数组的索引寻址，找到 Timer 对象索引 0 的对象，而对其赋值的是在 lib/timer.js 里面的 list.\_timer[kOnTimeout] = listOnTimeout; 。这边找到的对象是个 Function, 后面忽略 domains 异常处理等，就是简单的调用 Function 对象的 Call 方法, 并且传人上文提到的 Context 和参数。

Local<Value> ret = callback->Call(recv, argc, argv);

这就实现了 C++ 对 JS 函数的调用。

**4.4 底层系统API**

1. 总结

**5.1 Electron前景与缺点**

**5.2 其他的可能和新的创新**

1. 参考文献

[1] 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.