Electron 分享

Presentation slides for developers

Press Space for next page →

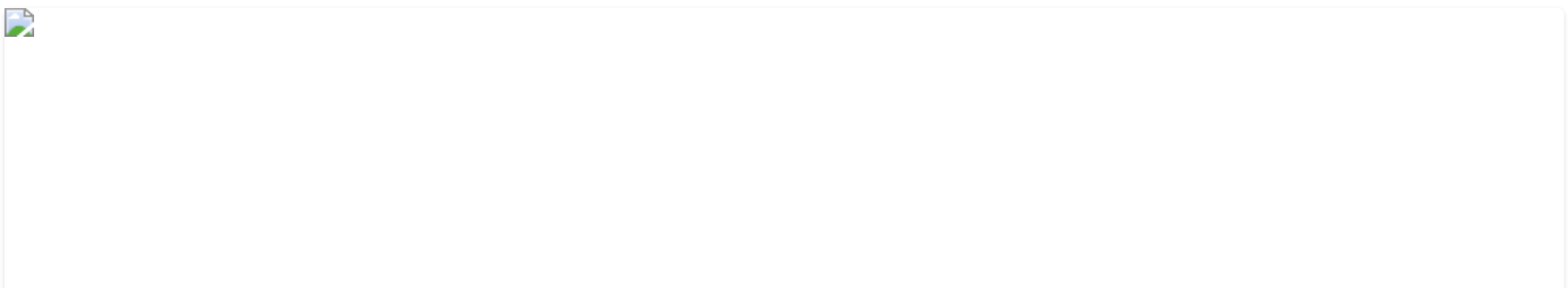
简介

Electron是一个使用 JavaScript、HTML 和 CSS 构建桌面应用程序的框架。

- Chromium 提供UI渲染能力
- Node.js 负责业务逻辑
- Native API 负责原生化能力

为什么选择Electron

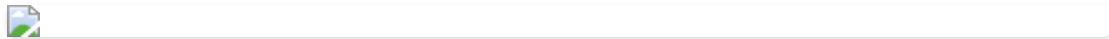
- 丰富的案例，VSCode, Notion，Microsoft Teams
- 活跃的开源社区 103k+ Star, 更新快。大部分electron相关的问题可以在社区、github issues、Stack Overflow里得到答案
- 开发方便。当不涉及一些原生化操作时，开发相较于QT这类传统跨端技术更加方便



多进程架构

- Main 主进程：通过Node, 原生API，来实现一些系统以及底层的操作。比如创建系统级别的菜单，操作剪贴板，创建APP的窗口等。
- Renderer 进程：主要通过Chromium来实现APP的图形界面
- Main进程和Renderer进程通过`ipcMain`和`ipcRenderer`来进行通信。通过事件监听和事件派发来实现两个进程通信，从而实现Main或者Renderer进程里不能实现的某些功能

Electron 模块



Main进程和Renderer进程分别包含一些模块，以及共享一些模块。

12.0.0之前的electron提供remote模块给renderer进程，以扩展渲染进程的功能

多进程之间的通信

通过事件监听和事件派发来实现两个进程通信。IPCMain 和 IPCRender。

```
// Main process
ipcMain.on('hello', (event, data)=>{
    console.log(`hello ${data}`) // hello world
})

// Renderer process
ipcRenderer.send('hello', 'world')
```

Remote模块

v12.0.0之前的electron

remote模块是electron为了让一些原本在Main进程里运行的模块也能在renderer进程里运行而创建的 eg:

- 默认浏览器里打开一个url

```
  // Renderer process
const remote = require('electron')

remote.shell.openExternal('')
```

- Remote 与 proxy object 当访问 `remoteObject.property` 属性时，实际上是做了个代理

```
-> ipcRenderer.sendSync('')
    -> ipcMain.on()
    <- event.sendReply
<- return
```

Remote模块的缺陷

1. 同步
2. 属性动态获取
3. 安全问题

为了确保能够获取到最新的值，remote底层并不会进行缓存，而是每次获取一个属性就动态到主进程中取。

```
// Main process
global.thing = {
  rectangle: {
    getBounds() { return { x: 0, y: 0, width: 100, height: 100 } }
    setBounds(bounds) { /* ... */ }
  }
}

// Renderer process
const thing = remote.getGlobal('thing')
const { x, y, width, height } = thing.rectangle.getBounds()
thing.rectangle.setBounds({ x, y, width, height: height + 100 })
```

新的多进程沟通方式

- 使用invoke 和 handle

```
// Main process
ipcMain.handle('my-invokable-ipc', async (event, ...args) => {
  const result = await somePromise(...args)
  return result
})


// Renderer process
async () => {
  const result = await ipcRenderer.invoke('my-invokable-ipc', arg1, arg2)
  // ...
}
```

Inter-Process Communication | Electron

contextBridge

- 不暴露ipcRender的情况

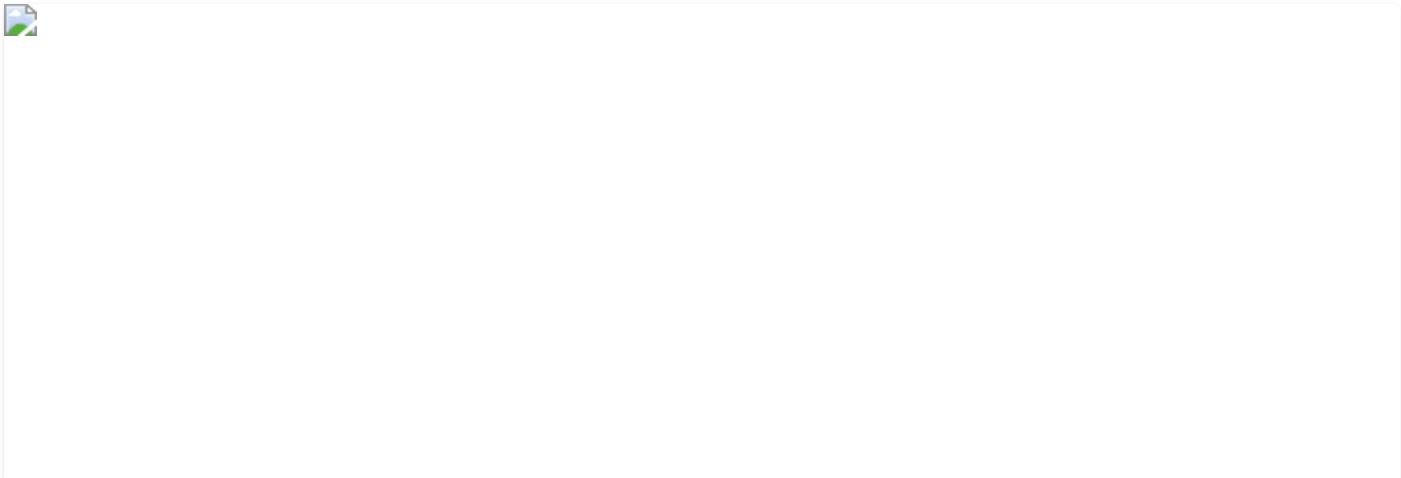
```
// preload
import { ipcRenderer, contextBridge } from 'electron';

// Adds an object 'api' to the global window object:
contextBridge.exposeInMainWorld('api', {
    doAction: async (arg) => {
        return await ipcRenderer.invoke('an-action', arg);
    }
});
```

```
 // renderer
 window.api.doAction()
```

Electron 在小布会议中的应用

小布会议中在共享屏幕/最小化时 需要多窗口交互数据的需求

IPC数据交互



基于Electron的IPC机制的，进程之间的通信的数据必须是可序列化的，比如JSON序列化。

IPC 序列化

- 原始值。例如字符串、数字、布尔值
- 数组。
- 对象。对象属性、对象的方法、以及对象的原型
- 函数。普通函数和构造方法、异常处理
- 特殊对象。Buffer、Promise

对其他特殊数据比如MediaStream不支持

可能出现的内存泄露

- 无法完全模拟JavaScript对象的行为
  - 值拷贝传递给渲染进程的
  - 在给主进程传递回调时，主进程会保持回调的引用。即使你在渲染进程调用了解除事件的方法,主进程仍然会保持监听

```
// preload
contextBridge.exposeInMainWorld(
  'event',
  {
    ipcOn: (eventName,fn) => ipcRenderer.on(eventName,fn),
    ipcRemove: (eventName,fn) => ipcRenderer.removeListener(eventName,fn)
  }
)

//renderer
window.event.ipcOn('event', noop);
onUnMounted(()=>{
  window.event.ipcRemove('eventName', noop);
})
```

Transparent window

```
// main
ipcMain.handle(EIpcName.GET_CURSOR, () => screen.getCursorScreenPoint());

ipcMain.on(EIpcName.SET_IGNORE_MOUSE, (event, ...args) => {
  if (!win) return;
  win.setIgnoreMouseEvents(...args);
});
```

```
// 在循环中不断判断
const point = await window.api.getCursor();
const ele = document.elementFromPoint(
      Math.floor(point.x),
      Math.floor(point.y));
if (ele?.closest(".window")) {
    window.api.setIgnoreMouse(false);
} else {
    window.api.setIgnoreMouse(true, { forward: true });
}
```

Electron与C++的使用

为什么要使用C++模块?

- NodeJS 的原生能力无法满足需求。（对窗口的检测，控制）
- 现有的 C++类库低成本地封装成 Node.js 扩展，供 Node 生态使用

eg: `RobotJS` `node-sass`

#Node Addon NodeJs 本身是基于 V8引擎和libuv，用C++进行开发的。因此非常方便使用一些方法来进行C++进行扩展。使其能够直接通过require的方式来调用C++ 扩展里面的内容

`*.node`文件本质：实际上是**动态库**，windows中的dll，linux中的so，macos中的dylib

在Node.js 中被 require 的时候，是通过 process.dlopen() 对其进行引入的

require

开发Node Addon工具

- node-gyp (Generate Your Project)
  - 它会识别包或者项目中的 **binding.gyp**，然后根据该配置文件生成各系统下能进行编译的项目，如 Windows 下生成 Visual Studio 项目文件（***.sln**等），Unix 下生成 Makefile。
- binding.gyp

```
{
  'targets': [
    {
      'target_name': 'addon',
      'sources': [
        'src/lib/addon.c',
        'src/lib/napi_helpers.c'
      ],
      'conditions': [
        ['OS=="win"',{...}],
        ['OS=="linux"'],
        ['OS=="mac"']
      ]
    }
  ]
}
```

NAN (Native Abstractions for Node.js)

```
#include <nan.h>

void Method(const Nan::FunctionCallbackInfo<v8::Value>& info) {
  info.GetReturnValue().Set(Nan::New("world").ToLocalChecked());
}

void Init(v8::Local<v8::Object> exports) {
  v8::Local<v8::Context> context = exports->CreationContext();
  exports->Set(context,
               Nan::New("hello").ToLocalChecked(),
               Nan::New<v8::FunctionTemplate>(Method)
                   ->GetFunction(context)
                   .ToLocalChecked());
}

NODE_MODULE(hello, Init)
```

问题：

- 当Node 版本改变时，需要进行重新编译。否则版本不符的话 Node.js 无法正常载入一个 C++ 扩展。
- 暴露V8给用户，不方便编写

Napi

Napi把 Node.js 的所有底层数据结构全部黑盒化，抽象成 N-API 当中的接口。

不同版本的 Node.js 使用同样的接口，这些接口是稳定地 ABI 化的，即应用二进制接口（Application Binary Interface）。这使得在不同 Node.js 下，只要 ABI 的版本号一致，编译好的 C++ 扩展就可以直接使用，而不需要重新编译

```
#include <node_api.h>

static napi_value Method(napi_env env, napi_callback_info info) {
  napi_status status;
  napi_value world;
  status = napi_create_string_utf8(env, "world", 5, &world);
  return world;
}
```

Node-Addon-API [1]

将napi的API抽象成面向对象的模式，方便用户

```
#include <napi.h>

Napi::String Method(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();
  return Napi::String::New(env, "world");
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, "hello"),
              Napi::Function::New(env, Method));
  return exports;
}

NODE_API_MODULE(hello, Init)
```

1. nodejs/node-addon-api: Module for using Node-API from C++ ↩

Thanks