

node-gyp 实现 nodejs 调用 C++

蒋礼锐 2019-12-10 👁 9,506 ⌚ 阅读12分钟

前端轮子千千万, 但还是有些瓶颈, 公司需要在前端调用自有 tcp 协议, 该协议只有 c++ 的封装版本. 领导希望可以直接调该模块, 不要重复造轮子.

实话说我对 C 还有点印象, 毕竟也是有二级 C 语言证的人..但是已经很久没用了, 看着一大堆的 C 语言类型的定义, 让我这个常年使用隐式类型的 js 情何以堪.这是我从业以来最难实现的 `hello world` 项目.

整体介绍

Native Addon

一个 Native Addon 在 Nodejs 的环境里就是一个二进制文件, 这个文件是由低级语言, 比如 C 或 C++实现, 我们可以像调用其他模块一样 `require()` 导入 Native Addon

Native Addon 与其他.js 的结尾的一样, 会暴露出 `module.exports` 或者 `exports` 对象, 这些被封装到 node 模块中的文件也被成为 **Native Module(原生模块)**.



蒋礼锐 LV.4

前端工程师 @bytedance

19

文章

96k

阅读

54

粉丝

关注

私信

目录

收起 ^

整体

Native Addon

Native Addon 是如何工作的呢?

ABI Application Binary Interface 应...

1. 使用核心实现

2. 使用 Native Abstractions for N...

3. 使用 N-API

4. 使用 node-addon-api 模块

开始实

环境准备

binding.gyp



这里不得不说下 DLL([Dynamic Linked Library](#))动态库, 他是由 C 或 C++使用标准编译器编译而成, 在 linux 或 macOS 中也被称作 [Shared Library](#). 一个 DLL 可以被一个程序在运行时动态加载, DLL 包含源 C 或 C++代码以及可通信的 API. 有动态是否还有静态的呢? 还真有~ 可以参考[这里](#)来看这两者的区别, 简单来说静态比动态更快, 因为静态不需要再去查找依赖文件并加载, 但是动态可以颗粒度更小的修改打包的文件.

在 Nodejs 中, 当编译出 DLL 的时候, 会被导出为.node 的后缀文件. 然后可以 require 该文件, 像 js 文件一样.不过代码提示是不可能有的了.

Native Addon 是如何工作的呢?

Nodejs 其实是很多开源库的集合,可以看看他的[仓库](#), 在 package.json 中找 deps. 使用的是谷歌开源的 [V8 引擎](#)来执行 js 代码, 而 V8刚好是使用 C++写的, 不信你看 v8 的[仓库](#). 而对于像异步 IO, 事件循环和其他低级的特性则是依赖 [Libuv](#) 库.

当安装完 nodejs 之后, 实际上是安装了一个包含整个 Nodejs 以及其依赖的源代码的编译版本, 这样就不用一个一个手动安装这些依赖而. 不过Nodejs也可以由这些库的源代码编译而来. 那么跟 Native Addon 有什么关系呢? 因为 **Nodejs 是由低层级的 C 和 C++编译而成的, 所以本身就具有与 C 和 C++相互调用的能力.**

Nodejs 可以动态加载 C 和 C++的 DLL 文件, 并且使用其 API 在 js 程序中进行操作. 以上就是基本的 Native Addon 在 Nodejs 中的工作原理.

[创建 C++示例](#)[关于打包的跨平台](#)

搜索建议

[Nodejs如何调用Dll模块](#)[electron通过node-ffi调用dll文件](#)[Nodejs 调用 dll](#)[nodejs调用C++扩展模块培训_C/C++_q...](#)[NodeJS C++插件开发\(一\) - 简介](#)[NodeJS C++插件简介](#)[nodejs在Windows下c++插件引用libpq...](#)[使用node-ffi调用dll动态库](#)[java 调用c++, c++调用python](#)[electron调用第三方dll \(附带dll\) 🙌](#)[精选内容](#)[Linux完全离线RPM安装NTP\(时间同步\)...](#)

ABI 是特指应用去访问**编译好|compiled**的程序, 跟 API(Application Programming Interface)非常相似, 只不过是跟二进制文件进行交互, 而且是访问**内存地址**去查找 Symbols, 比如 numbers, objects, classes和 functions

那么这个 ABI 跟 Native Addon 有什么关系呢? 他是 Native Addon 与 Nodejs 进行通信的桥梁. DDL 文件实际上是通过 Nodejs 提供的ABI 来注册或者访问到值, 并且通过Nodejs暴露的 API和库来执行命令.

举个例子, 有个 Native Addon 想添加一个 **sayHello** 的方法到 **exports** 对象上, 他可以通过访问 Libuv 的 API 来创建一个新的线程,异步的执行任务, 执行完毕之后再调用回调函数. 这样 Nodejs 提供的 ABI 的工作就完成了.

通常来说, 都会将 C 或 C++编译为 DLL, 会使用到一些被称作**header 头文件**的元数据. 都是以 **.h** 结尾.当然这些头文件中, 可以是 Nodejs及node的库暴露出去的可以让 Native Addon引用的.头文件的资料可[参考](#)

一个典型的引用是使用 **#include** 比如 **#include<v8.h>**, 然后使用声明来写 Nodejs 可执行的代码.有以下四种方式使用头文件.

1. 使用核心实现

比如 **v8.h** -> v8引擎, **uv.h** -> Libuv库这两个文件都在 node 的安装目录中, 但是这样的问题就是 Native Addon 和 Nodejs 之间的依赖程度太高了.因为 Nodejs 的这

辣码甄源 · 540阅读 · 2点赞

 技术专题25期 | 性能优化的实践与经...

掘金酱 · 2.6k阅读 · 11点赞

把数据库转换成电子表格：支持实时编...

开源服务指南 · 690阅读 · 2点赞

QGIS将经纬度字符串转换为WKT和Geo...

ChatGIS · 472阅读 · 0点赞

找对属于你的技术圈子

回复「进群」加入官方微信群



2. 使用 Native Abstractions for Node(NAN)

[NAN](#) 项目最开始就是为了抽象 nodejs 和 v8 引擎的内部实现. 基本概念就是提供了一个 npm 的安装包, 可以通过前端的包管理工具 [yarn](#) 或 [npm](#) 进行安装, 他包含了 [nan.h](#) 的头文件, 里面对 nodejs 模块和 v8 进行了抽象. 但是 NAN 有以下缺点:

- 不完全抽象出了 V8 的 api
- 并不提供 nodejs **所有**库的支持
- 不是Nodejs 官方维护的库.

所以更推荐以下两种方式

3. 使用 N-API

[N-API](#)类似于 NAN 项目, 但是是由 nodejs 官方维护, 从此就不需要安装外部的依赖来导入到头文件. 并且提供了可靠的抽象层 他暴露了 [node_api.h](#) 头文件, 抽象了 nodejs 和包的内部实现, 每次 Nodejs 更新, N-API 就会同步进行优化保证 ABI 的可靠性 [这里](#)是 N-API 的所有接口文档, [这里](#)是官方对 N-API 的 ABI 稳定性的描述

N-API 同时适合于 C 和 C++, 但是 C++的 API 使用起来更加的简单, 于是, node-addon-api 就应运而生.

跟上述两个一样, 他有自己的头文件 `napi.h`, 包含了 N-API 的所有对 C++ 的封装, 并且跟 N-API 一样是由官方维护, [点这里](#) 查看仓库. 因为他的使用相较于其他更加的简单, 所以在进行 C++ API 封装的时候优先选择该方法.

开始实现 Hello World

环境准备

需要全局安装 `yarn global add node-gyp`, 因为还依赖于 Python, (GYP 全称是 Generate Your Project, 是一个用 Python 写成的[工具](#)). 具体制定 python 的环境及路径参考[文档](#).

安装完成后就有了一个生成编译 C 或 C++ 到 Native Addon 或 DLL 的模板代码的 CLI, 一顿操作猛如虎后, 会生成一个 `.node` 文件. 但是这个模板是怎么生成的呢? 就是下面这个 `binding.gyp` 文件

`binding.gyp`

`binding.gyp` 包含了模块的名字, 哪些文件应该被编译等. 模板会根据不同的平台或架构(32 还是 64) 包含必要的构建指令文件, 也提供了必要的 header 或 source 文件去编译 C 或 C++, 类似于 JSON 的格式, 详情可[点击查看](#).

设置项目

复制代码

```
1  |— binding.gyp
2  |— index.js
3  |— package.json
4  |— src
5  |   |— greeting.cpp
6  |   |— greeting.h
7  |   |— index.cpp
8  |— yarn.lock
9
```

安装依赖

Native Module 跟正常的 node 模块或其他 NPM 包一样. 先 `yarn init -y` 初始化项目, 再安装node-addon-api `yarn add node-addon-api`.

创建 C++ 示例

创建 `greeting.h` 文件

复制代码

```
1  #include <string>
2  std::string helloUser(std::string name);
```

复制代码

```
1 #include <iostream>
2 #include <string>
3 #include "greeting.h"
4
5 std::string helloUser(std::string name) {
6     return "Hello " + name + "!";
7 }
```

创建 `index.cpp` 文件, 该文件会包含 `napi.h`

复制代码

```
1 #include <napi.h>
2 #include <string>
3 #include "greeting.h"
4
5 // 定义一个返回类型为 Napi String 的 greetHello 函数, 注意此处的 info
6 Napi::String greetHello(const Napi::CallbackInfo& info) {
7     Napi::Env env = info.Env();
8     std::string result = helloUser('Lorry');
9     return Napi::String::New(env, result);
10 }
11
12 // 设置类似于 exports = {key:value} 的模块导出
13 Napi::Object Init(Napi::Env env, Napi::Object exports) {
14     exports.Set(
15         Napi::String::New(env, "greetHello"), // key
16         Napi::Function::New(env, greetHello) // value
17     );
18 }
```

```
20 }  
21  
22 NODE_API_MODULE(greet, Init)
```

注意这里你看到很多的 `Napi::` 这样的书写, 其实这就是在 js 与 C++ 之间的数据格式桥梁, 定义双方都看得懂的数据类型. 这里经历了以下流程:

1. 导入 `napi.h` 头文件, 他会解析到下面会说的 `binding.gyp` 指定的路径中
2. 导入 `string` 标准头文件和 `greeting.h` 自定义头文件. 注意使用 `"` 和 `<>` 的区别, `"` 会查找当前路径, 详情请[查看](#)
3. 使用 `Napi::` 开头的都是使用的 `node-addon-api` 的头文件. `Napi` 是一个命名空间. 因为宏不支持命名空间, 所以 `NODE_API_MODULE` 前没有
4. `NODE_API_MODULE` 是一个 `node-api` (N-API) 中封装的 `NAPI_MODULE` 宏中提供的函数([宏](#)). 它将会在 js 使用 `require` 导入 Native Addon 的时候被调用.
5. 第一个参数为唯一值用于注册进 node 里表示导出模块名. 最好与 `binding.gyp` 中的 `target_name` 保持一致, 只不过这里是使用一个标签 `label` 而不是字符串的格式
6. 第二个参数是 C++ 的函数, 他会在 Nodejs 开始注册这个方法的时候进行调用. 分别会传入 `env` 和 `exports` 参数
7. `env` 值是 `Napi::env` 类型, 包含了注册模块时的环境(environment), 这个在 N-API 操作时被使用. `Napi::String::New` 表示创建一个新的 `Napi::String` 类型的值. 这样就将 `helloUser` 的 `std:string` 转换成了 `Napi::String`
8. `exports` 是一个 `module.exports` 的低级 API, 他是 `Napi::Object` 类型, 可以使用 `Set` 方法添加属性, 参考[文档](#), 该函数一定要返回一个 `exports`

创建 *binding.gyp* 文件



生成模板文件

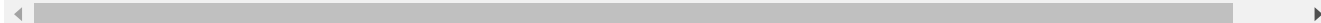
在 *binding.gyp* 同级目录下使用



将会生成一个 build 文件夹, 会包含以下文件:

[复制代码](#)

```
1  ./build
2  |— Makefile           // 包含如何构建 native 源代码到 DLL 的指令, 并且兼容 Nodejs 的运行
3  |— binding.Makefile   // 生成文件的配置
4  |— config.gypi         // 包含编译时的配置列表
5  |— greet.target.mk    // 这个 greet 就是之前配置的 target_name 和 NODE_API_MODULE 的第一个
6  |— gyp-mac-tool       // mac 下打包的python 工具
```



构建并编译

[复制代码](#)

```
1  node-gyp build
```

将会构建出一个 `.node` 文件

[复制代码](#)

```
1  ./build
2  |— Makefile
3  |— Release
4  |   |— greet.node      // 这个就是编译出来的node文件, 可直接被 js require 引用
5  |   |— obj.target
```

```
9 | └─ index.o
10 | └─ binding.Makefile
11 | └─ config.gypi
12 | └─ greet.target.mk
13 | └─ gyp-mac-tool
```

走到这一步你会发现 `.node` 文件是无法被打开的, 因为他就不是给人读的, 是一个二进制文件. 这个时候就可以尝试一波

复制代码

```
1 // index.js
2 const addon = require('./build/Release/greet.node')
3 console.log(addon.greetHello())
```

直接使用 `node index.js`

`Hello Lorry !`

内容. 真是不容易啊

仅仅到此吗? 还不够

传参

上述代码都是写死

于是之前说到的 `in`

的访问. 以下是对 `index.cpp`

[参考](#)[运算符重载](#)`greetHello`

```
2  Napi::Env env = info.Env();
3  std::string user = (std::string) info[0].ToString();
4  std::string result = helloUser(user);
5  return Napi::String::New(env, result);
6 }
```

然后使用



复制代码

```
1  node-gyp rebuild
```

在修改下引用的 index.js 文件



复制代码

```
1  const addon = require('./build/Release/greet.node')
2  console.log(addon.greetHello('张三')) // Hello 张三!
```

至此, 终于算是比较完整的实现了我们的 hello world. 别急, 还有货

如果要像其他包一样可以进行发布的话, 操作就跟正常的npm打包流程差不多了. 在 `package.json` 中的 `main` 字段中指定 `index.js`, 然后修改 `index.js` 内容为:



复制代码

包出一

把

限于

本实

nod

本地

yp

口

户疯狂,于是还有一个办法:为每个平台单独打包,然后使用不同的安装脚本实现区分安装,有一个第三方包 `node-pre-gyp`

`node-pre-gyp` 中那么复杂的配置,还可以尝试 `prebuild-install`

但是还有一个问题,我们如何实现打包出不同平台

现实. 没事,还有轮子 `prebuild`, 可以设置不同平台

PS: 这里还有一个 vscode 的坑, 在使用 C++ `extension`

`#include <napi.h>` 找不到文件,但是打包是

`binding.gyp` 里的头文件查找路径,找了很多

真的很容易写错啊!!

标签: C++

评论 8



登录 / 注册 即可发布评论!

最热 最新



abcdGJJ  前端工程师 @百度

最后的vscode插件问题能详细点么，同遇到😭

1年前  点赞  评论 ...



——1720——  资深前端开发

../src/index.cpp:8:24: error: no matching function for call to 'helloUser'

1年前  1  1 ...



小蟋蟀：你好 这个问题咋解决的 怎么调用直接用c++写的代码

1年前

点赞

回复

...



天平

好文章，感谢。

2年前

点赞

评论


...

[查看全部 8 条评论](#)

为你推荐

electron

如何通过webpack 实现在 HTML,CSS, JS 中共享变量

蒋礼锐 4年前  3.8k  5  2

SCSS

如何在 electron 中安装驱动

蒋礼锐 4年前  3.1k  9  4

Electron

从1开始学习THREE.js--光源

蒋礼锐 4年前  1.7k  4  评论

three.js

从1开始学习THREE.js--材质

蒋礼锐 4年前  1.3k  4  2


three.js

CSS中的变量

蒋礼锐 3年前  930  3  3

CSS

从1开始学习THREE.js--场景

蒋礼锐 4年前  955  3  1

three.js

SOLID的 JavaScript 实现

蒋礼锐 4年前  575  6  评论

设计模式