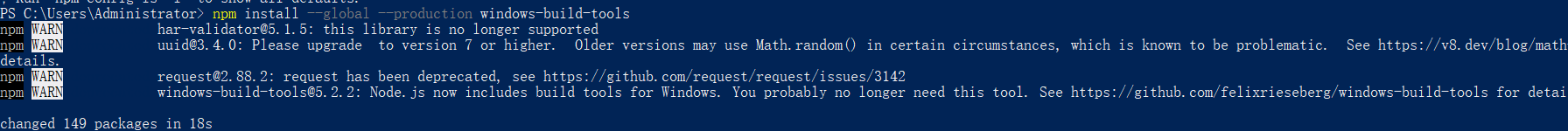
# node-ffi环境搭建(windows)

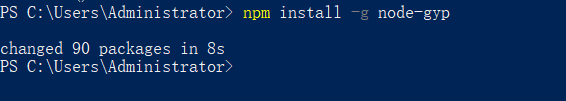
node-ffi通过Buffer类，在C代码和JS代码之间实现了内存共享，类型转换则是通过ref、ref-array、ref-struct实现。由于node-ffi/ref包含C原生代码，安装需要配置Node原生插件编译环境。

1、安装node-gyp

npm install --global --production windows-build-tools

npm install -g node-gyp

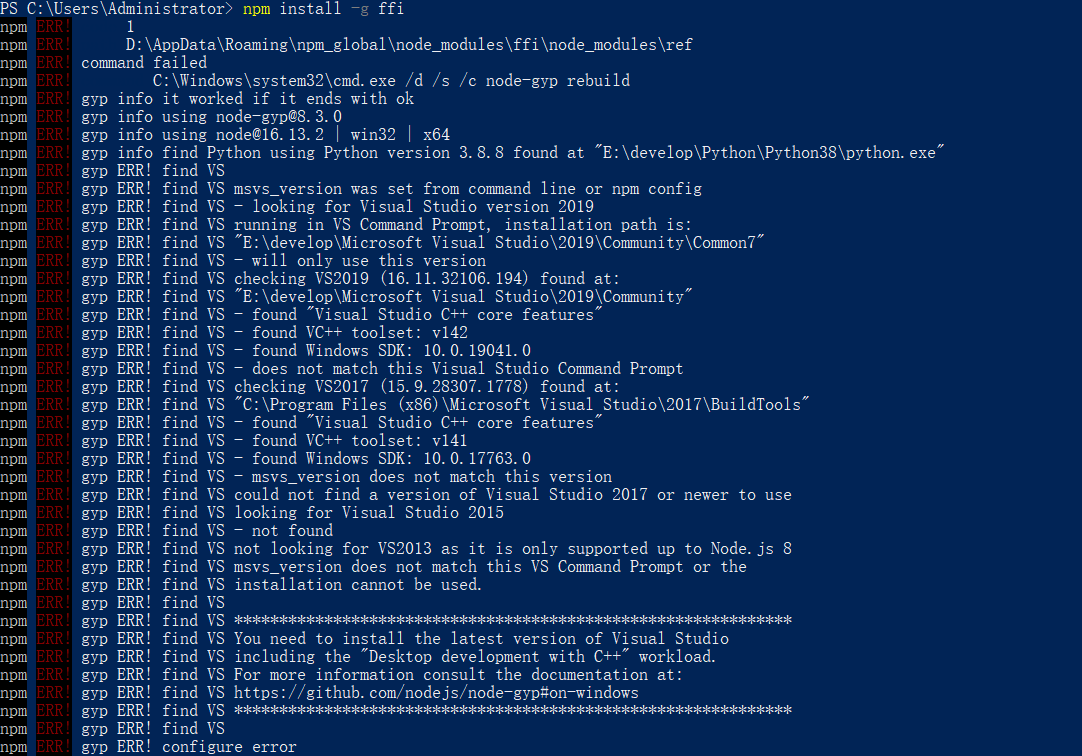




2、安装必要的依赖

npm install ffi

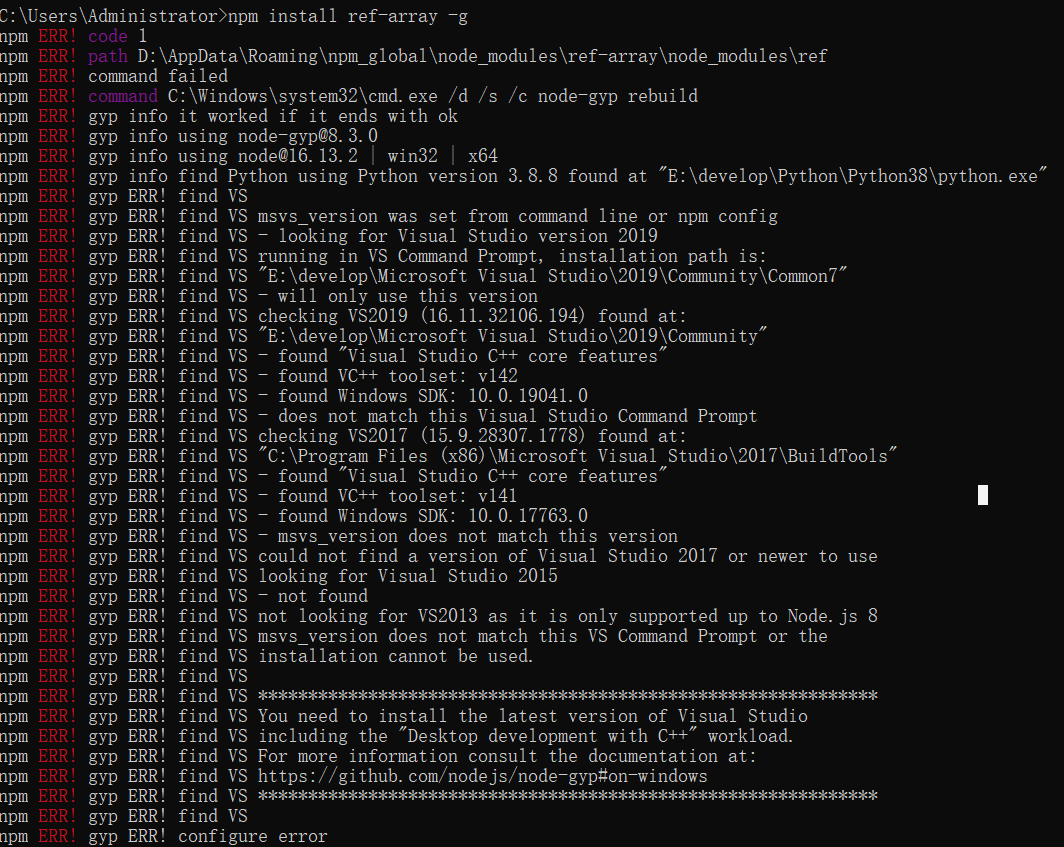
ffi安装失败



npm install ref

npm install ref-array

npm install ref-struct

上述和ffi相关的依赖安装时报同样的错误

(electron项目可选)

npm install electron-rebuild

package.json

"scripts": {

"rebuild": "cd ./node\_modules/.bin && electron-rebuild --force --module-dir=../../"

}

之后执行npm run rebuild 操作即可完成electron的重编译。

## 变量类型

C语言中有4种基础数据类型：整型、浮点型、指针、聚合类型

整型、字符型都有分有符号和无符号两种

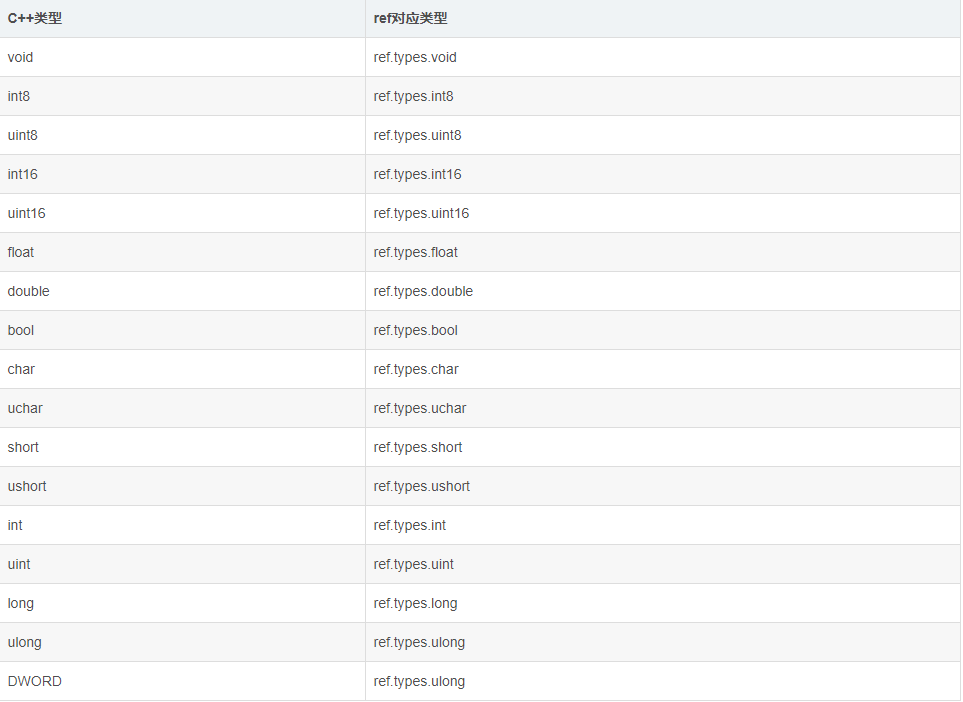


在不声明unsigned时 默认为signed型。

ref中unsigned会缩写成u, 如 uchar 对应 usigned char。

浮点型有 float、double、long double

ref库中基础类型的对应关系如下：



ffi.Library中，既可以通过ref.types.xxx的方式声明类型，也可以通过文本(如uint16)进行声明。

字符型

字符型由char构成，在GBK编码中一个汉字占2个字节，在UTF-8中占用3~4个字节。一个ref.types.char默认一字节。根据所需字符长度创建足够长的内存空间。这时候需要使用ref-array库。

const ref = require('ref')

const refArray = require('ref-array')

const CharArray100 = refArray(ref.types.char, 100) // 申明char[100]类型CharArray100

const bufferValue = Buffer.from('Hello World') // Hello World转换Buffer

// 通过Buffer循环复制

const value1 = new CharArray100()

for (let i = 0, l = bufferValue.length; i < l; i++) {

value1[i] = bufferValue[i]

}

// 使用ref.alloc初始化类型

const strArray = [...bufferValue] //需要将`Buffer`转换成`Array`

const value2 = ref.alloc(CharArray100, strArray)

在传递中文字符型时，必须预先得知DLL库的编码方式。node默认使用UTF-8编码。若DLL不为UTF-8编码则需要转码，推荐使用iconv-lite。

安装iconv-lite

npm install iconv-lite

使用：

const iconv = require('iconv-lite')

const cstr = iconv.encode(str, 'gbk')

使用encode转码后str为Buffer类，可直接作为uchar类型

iconv.encode(str.'gbk')中gbk默认使用的是unsigned char | 0 ~ 256储存。假如C代码需要的是signed char | -127 ~ 127，则需要将buffer中的数据使用int8类型转换。

const Cstring100 = refArray(ref.types.char, 100)

const cString = new Cstring100()

const uCstr = iconv.encode('农企药丸', 'gbk')

for (let i = 0; i < uCstr.length; i++) {

cString[i] = uCstr.readInt8(i)

}

C代码为字符数组char[]/char \*设置的返回值，通常返回的文本并不是定长，不会完全使用预分配的空间，末尾则会是无用的值。如果是预初始化的值，一般末尾是一大串的0x00，需要手动做trimEnd，如果不是预初始化的值，则末尾不定值，需要C代码明确返回字符串数组的长度returnValueLength。

内置简写

ffi中内置了一些简写

ref.types.int => 'int'

ref.refType('int') => 'int\*'

char\* => 'string'

建议使用'string'。

字符串虽然在js中被认为是基本类型，但在C语言中是以对象的形式来表示的，所以被认为是引用类型。所以string其实是char\* 而不是char。

聚合类型

1、多维数组

多维数组的基本类型 则需要使用ref-array进行创建

char cName[50][100] *// 创建一个cName变量储存级50个最大长度为100的名字*

const ref = require('ref')

const refArray = require('ref-array')

const CName = refArray(refArray(ref.types.char, 100), 50)

const cName = new CName()

2、结构体

结构体是C中常用的类型，需要用ref-struct进行创建

//c代码

typedef struct {

char cTMycher[100];

int iAge[50];

char cName[50][100];

int iNo;

} Class;

typedef struct {

Class class[4];

} Grade;

//js代码

const ref = require('ref')

const Struct = require('ref-struct')

const refArray = require('ref-array')

const Class = Struct({ *// 注意返回的`Class`是一个类型*

cTMycher: RefArray(ref.types.char, 100),

iAge: RefArray(ref.types.int, 50),

cName: RefArray(RefArray(ref.types.char, 100), 50)

})

const Grade = Struct({ *// 注意返回的`Grade`是一个类型*

class: RefArray(Class, 4)

})

const grade3 = new Grade() *// 新建实例*

3、指针

指针是一个变量，其值为实际变量的地址，即内存位置的直接地址，类似于JS中的引用对象。

C语言中使用\*来代表指针，如 int a\* 则就是 整数型a变量的指针 , &用于表示取地址。

node-ffi实现指针的原理是借助ref，使用Buffer类在C代码和JS代码之间实现了内存共享，让Buffer成为了C语言当中的指针。注意，一旦引用ref，会修改Buffer的prototype，替换和注入一些方法。

const buf = new Buffer(4) *// 初始化一个无类型的指针*

buf.writeInt32LE(12345, 0) *// 写入值12345*

console.log(buf.hexAddress()) *// 获取地址hexAddress*

buf.type = ref.types.int *// 设置buf对应的C类型，可以通过修改`type`来实现C的强制类型转换*

console.log(buf.deref()) *// deref()获取值12345*

const pointer = buf.ref() *// 获取指针的指针，类型为`int \*\*`*

console.log(pointer.deref().deref()) *// deref()两次获取值12345*

结构类型VS指针类型 *// 声明一个类的实例*

const grade3 = new Grade() *// Grade 是结构类型*

*// 结构类型对应的指针类型*

const GradePointer = ref.refType(Grade) *// 结构类型`Grade`对应的指针的类型，即指向Grade*

*// 获取指向grade3的指针实例*

const grade3Pointer = grade3.ref()

*// deref()获取指针实例对应的值*

console.log(grade3 === grade3Pointer.deref()) *// 在JS层并不是同一个对象*

console.log(grade3['ref.buffer'].hexAddress() === grade3Pointer.deref()['ref.buffer'].hexAddress()) *//但是实际上指向的是同一个内存地址，即所引用值是相同的*

可以通过ref.alloc(Object|String type, ? value) → Buffer直接得到一个引用对象

const iAgePointer = ref.alloc(ref.types.int, 18) *// 初始化一个指向`int`类的指针，值为18*

const grade3Pointer = ref.alloc(Grade) *// 初始化一个指向`Grade`类的指针*

4、回调函数

C的回调函数一般是用作入参传入

const ref = require('ref')

const ffi = require('ffi')

const testDLL = ffi.Library('./testDLL', {

setCallback: ['int', [

ffi.Function(ref.types.void, *// ffi.Function申明类型， 用`'pointer'`申明类型也可以*

[ref.types.int, ref.types.CString])]]

})

const uiInfocallback = ffi.Callback(ref.types.void, *// ffi.callback返回函数实例*

[ref.types.int, ref.types.CString],

(resultCount, resultText) => {

console.log(resultCount)

console.log(resultText)

},

)

const result = testDLL.uiInfocallback(uiInfocallback)

注意！如果CallBack是在setTimeout中调用，可能存在被GC的BUG

process.on('exit', () => {

/\* eslint-disable-next-line \*/

uiInfocallback // keep reference avoid gc

})

## 应用范例

extern "C" int \_\_declspec(dllexport)My\_Test(char \*a, int b, int c);

extern "C" void \_\_declspec(dllexport)My\_Hello(char \*a, int b, int c);

import ffi from 'ffi'

// `ffi.Library`用于注册函数，第一个入参为DLL(或so)路径，最好为文件绝对路径

const dll = ffi.Library( './test.dll', {

// My\_Test是dll中定义的函数，两者名称需要一致

// [a, [b，c....]] a是函数返回类型，[b，c]是dll函数的入参类型

My\_Test: ['int', ['string', 'int', 'int']], // 可以用文本表示类型

My\_Hello: [ref.types.void, ['string', ref.types.int, ref.types.int]] // 更推荐用`ref.types.xx`表示类型，方便类型检查

})

//同步调用

const result = dll.My\_Test('hello', 3, 2)

//异步调用

dll.My\_Test.async('hello', 3, 2, (err, result) => {

if(err) {

//todo

}

return result

})

# 解决npm install安装卡住/失败问题

## 方案一：安装cnpm镜像

npm install -g cnpm --registry=https://registry.npm.taobao.org

cmd输入以上命令就可以了，然后输入

cnpm install -g @angular/cli

后面的操作跟不使用镜像的操作是差不多的。  
但是后续使用过程中可能会有一些问题，运行ng eject后发生了错误，所以放弃了这个方案，采用了方案二。

## 方案二：使用代理registry

使用淘宝提供的代理：

npm config set registry https://registry.npm.taobao.org

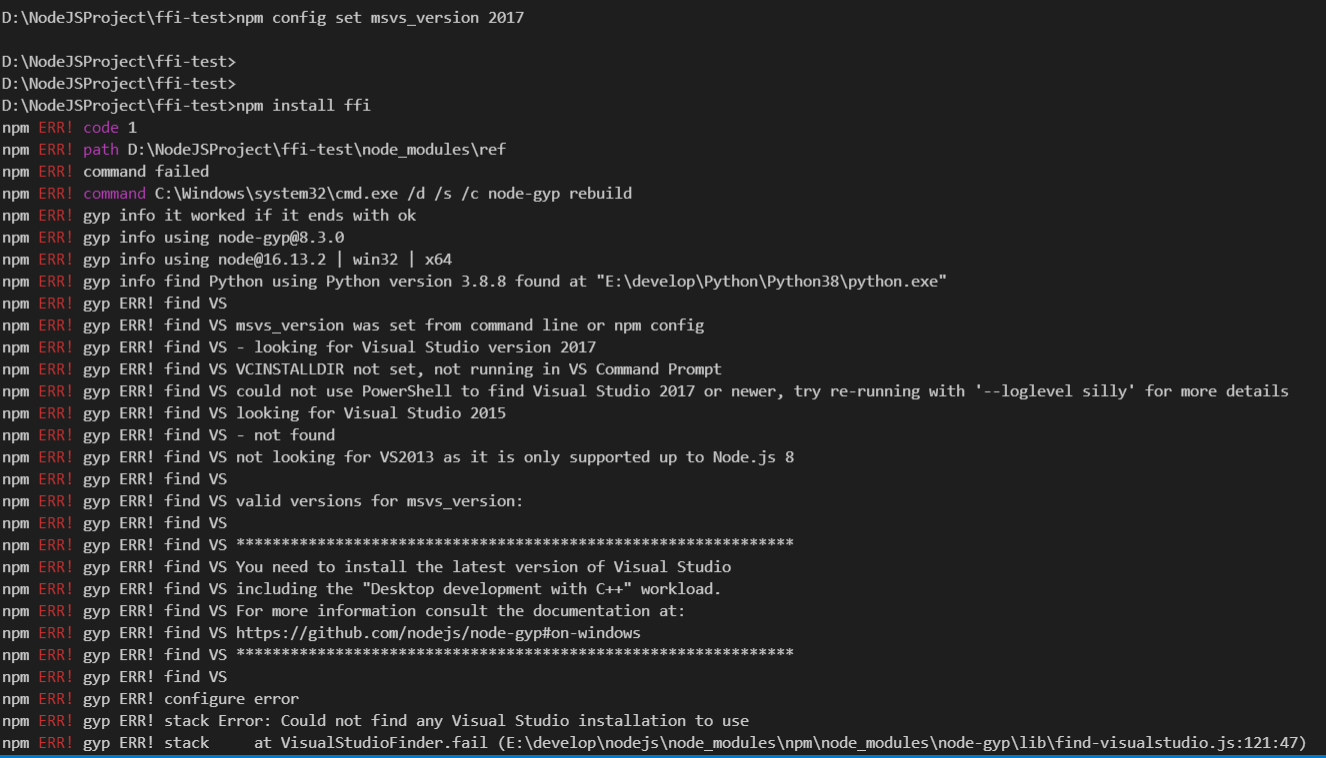
最后，设置npm使用http而非https也能提高下载时的速度

设置为false而非true。

npm config set strict-ssl false

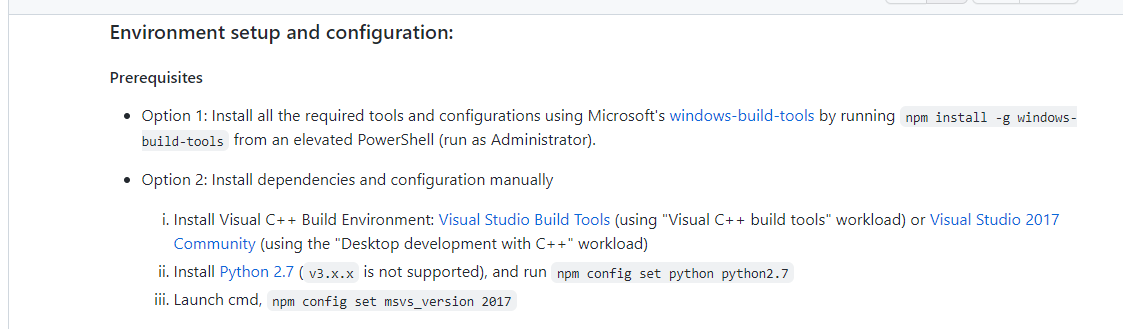
# ffi依赖安装失败处理

ffi安装失败：



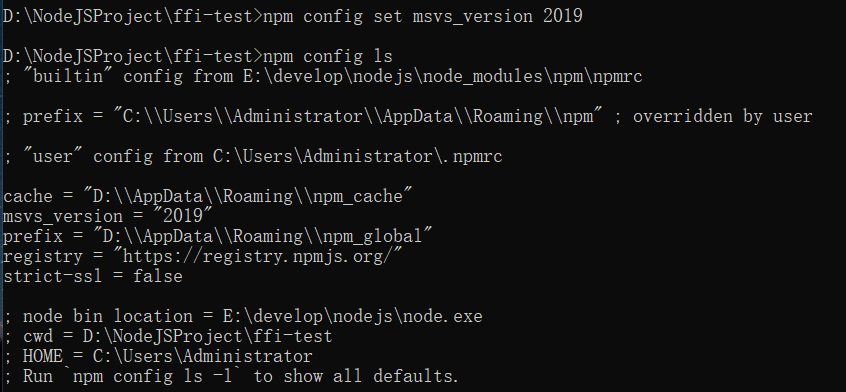
查阅github关于相关配置的介绍：

<https://github.com/Microsoft/nodejs-guidelines/blob/master/windows-environment.md#compiling-native-addon-modules>





将msvs\_version的版本设置为2017/2019/2022，均未能解决。

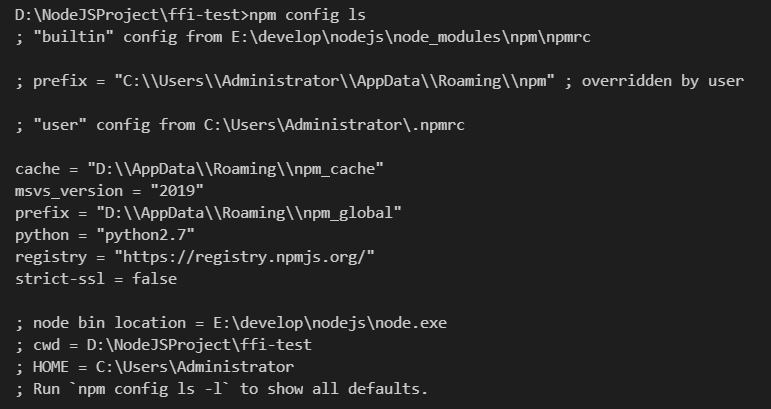


安装FFI依旧报错

继续尝试解决：

查看设置：

npm config ls



将registry由淘宝镜像修改为<https://registry.npmjs.org/>

设置如下：npm config set registry=<https://registry.npmjs.org/>

继续执行安装，提示同样的错误。

windows下FFI安装失败问题未能解决。

# FFI调用C函数环境搭建（Linux）

安装ffi模块失败。

使用Java调用.so动态库。

# 使用JNA调用so

使用java调用.so，angular调用java的方式，实现angular调用so。

JNA 全称 Java Native Access，是一个建立在经典的 JNI 技术之上的 Java 开源框架 （https://github.com/twall/jna）。JNA 提供一组 Java 工具类用于在运行期动态访问系统本地 库（native library：如 Window 的 dll，Linux的so）而不需要编写任何 Native/JNI 代码。开发人员只要 在一个 java 接口中描述目标 native library 的函数与结构，JNA 将自动实现 Java 接口到 native function 的映射。

JNA 包：https://maven.java.net/content/repositories/releases/net/java/dev/jna/jna/4.0.0/ jna-4.0.0.jar

[https://maven.java.net/content/repositories/releases/net/java/dev/jna/jna-platfo rm/4.0.0/jna-platform-4.0.0.jar](https://maven.java.net/content/repositories/releases/net/java/dev/jna/jna-platfo%20rm/4.0.0/jna-platform-4.0.0.jar)

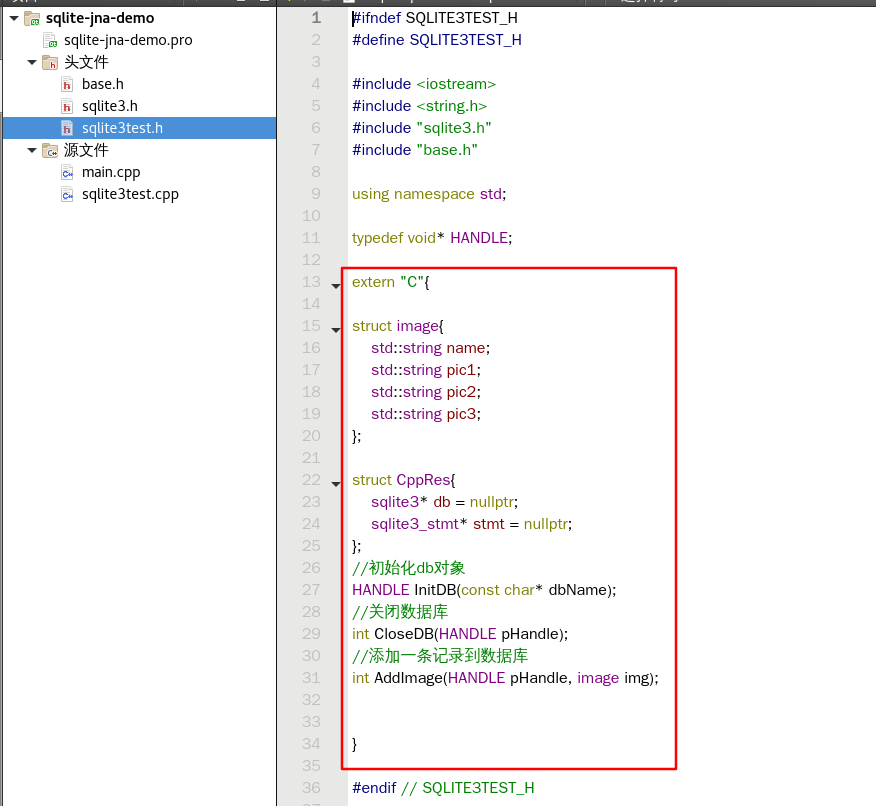
JNA 在线帮助文档：<http://java-native-access.github.io/jna/4.2.1/>

JNA 入门示例： <https://github.com/twall/jna/blob/master/www/GettingStarted.md>

## 封装C项目生成.so文件

<https://github.com/Angeliau/sqlite-c-demo.git>

C工程结构(开发工具QtCreator)如下：



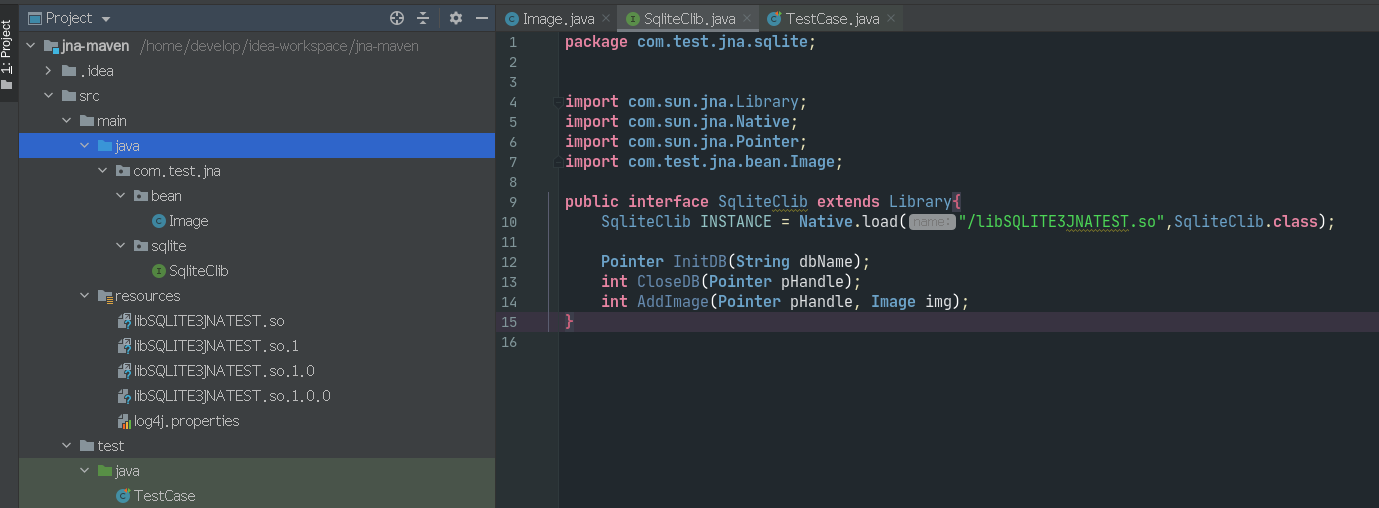


编译该项目，生成so

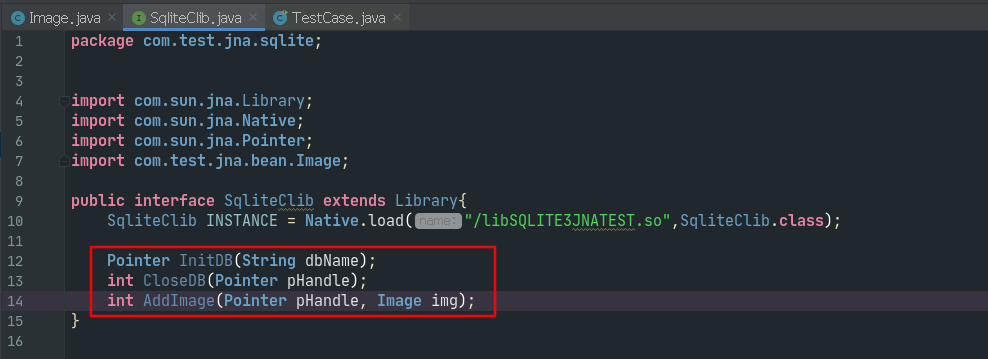


## Java调用so

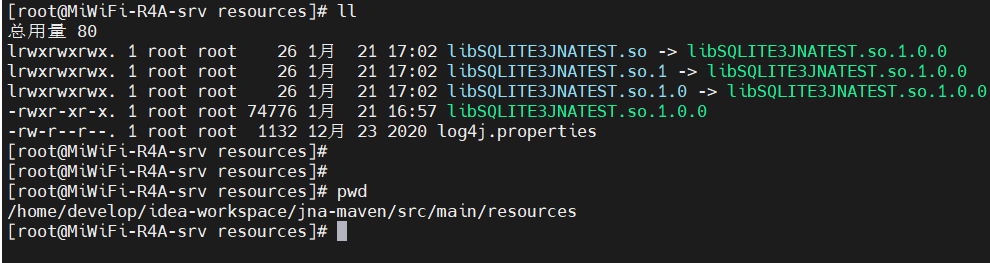
Java工程目录结构如下

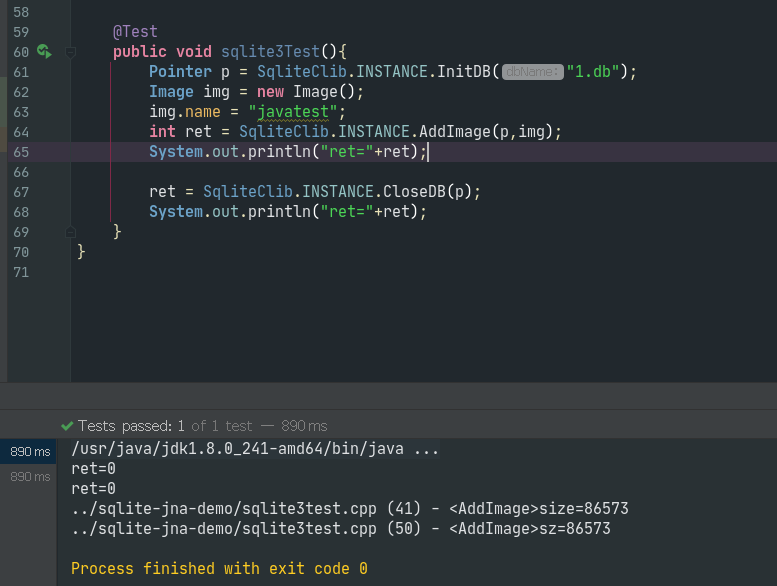


在Java中声明了与C函数中对应的函数：



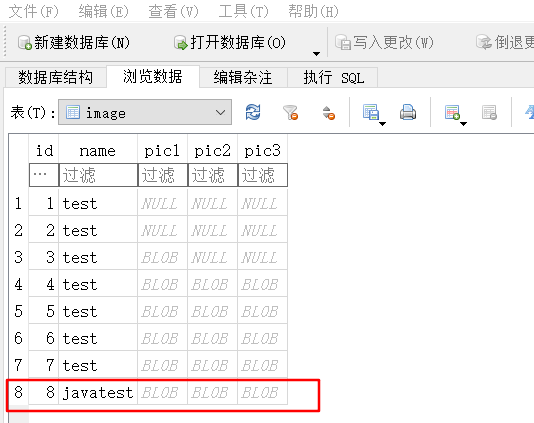
将生成的so文件放在Java项目的src/main/resources目录下，执行测试：





jna项目：<https://github.com/Angeliau/sqlite-jna-demo.git>

查看db中的数据：



最后一条记录是使用Java调用插入的数据。

# Angular项目Linux下运行验证-通过

1、在linux上安装nodejs

2、下载源码到linux

3、在源码文件夹下，执行npm install安装必要依赖

4、执行npm start启动运行

5、在浏览器访问localhost:4200/

