

20 使用 Rust 开发 Node.js 模块

众所周知，我们可以用 C/C++ 语言开发 Node.js 模块，只需借助 `node-gyp` 工具链的能力，就可以把我们编写的 C/C++ 的代码编译成 Node.js 模块，可以在 Node.js 项目中充分利用 C/C++ 的性能优势。还有一种情况就是某些外部资源并没有提供 Node.js 的 SDK 或者驱动，但是提供了 C 语言版本的 SDK 或者驱动，那么就没有必要使用 JavaScript 从头再来一遍了，直接通过 `node-gyp` 封装一下，提供给 JavaScript 使用就好了，提高开发效率。

使用 Rust 开发 Node.js 模块的思路也是差不多的，只不过这里我们要使用另外的工具链了。

Neon 简介

Neon 是一个辅助你使用 Rust 开发 Node.js 模块的工具链。下面，我们使用一些列的示例代码来演示如何使用 Neon 工具链，用 Rust 开发 Node.js 模块。

创建项目

打开命令行工具，转到你平时写代码的目录，执行命令：

```
1 $ npm init neon neon-lab
```

第一次执行的时候，会自动下载 `create-neon` 包，后续的步骤和 `npm init` 一样，根据提示做下来就好。命令成功执行之后，创建了一个名为 `neon-lab` 的项目，这个目录的结构如下：

```
1  .
2  |— Cargo.toml
3  |— README.md
4  |— package.json
5  |— src
6    |— lib.rs
```

从上面的目录结构可以看出来，这个项目既包含了 Rust 的项目文件 `Cargo.toml`，也包含了 Node.js 的项目文件 `package.json`。其中，`src/lib.rs` 是这个 Rust 项目（Library Package）的入口文件。

`package.json` 文件的内容比较简单：

```
1  {
2    "name": "neon-lab",
3    "version": "0.1.0",
4    "description": "",
5    "main": "index.node",
6    "scripts": {
7      "test": "cargo test",
8      "cargo-build": "cargo build --message-format=json >
cargo.log",
9      "cross-build": "cross build --message-format=json >
cross.log",
10     "postcargo-build": "neon dist < cargo.log",
11     "postcross-build": "neon dist -m /target < cross.log",
12     "debug": "npm run cargo-build --",
13     "build": "npm run cargo-build -- --release",
14     "cross": "npm run cross-build -- --release"
15   },
16   "author": "",
17   "license": "ISC",
18   "devDependencies": {
19     "@neon-rs/cli": "0.1.68"
20   }
21 }
```

主要是 `scripts` 属性中定义了一系列的构建命令。当我们开发完成之后，需要使用相应的命令来构建项目。

`Cargo.toml` 的内容也不多：

```
1  [package]
2  name = "neon-lab"
3  version = "0.1.0"
4  license = "ISC"
5  edition = "2021"
6  exclude = ["index.node"]
7
8  [lib]
9  crate-type = ["cdylib"]
10
11 # See more keys and their definitions at https://doc.rust-
12   lang.org/cargo/reference/manifest.html
13
14 [dependencies]
15 neon = "1"
```

这里有一个需要注意的，如果我们希望准确配置这个模块兼容的 Node.js 的版本，需要在 `Cargo.toml` 中配置。默认情况下，Neon 项目会以当前系统 Node.js 的版本为准构建项目。但是，你可以手动配置支持的 Node.js 版本：

```
1  [dependencies.neon]
2  features = ["napi-6"]
```

Node API 版本对照如下表所示：

NODE-API VERSION	SUPPORTED IN
9	v18.17.0+, 20.3.0+, 21.0.0 and all later versions
8	v12.22.0+, v14.17.0+, v15.12.0+, 16.0.0 and all later versions
7	v10.23.0+, v12.19.0+, v14.12.0+, 15.0.0 and all later versions
6	v10.20.0+, v12.17.0+, 14.0.0 and all later versions
5	v10.17.0+, v12.11.0+, 13.0.0 and all later versions
4	v10.16.0+, v11.8.0+, 12.0.0 and all later versions
3	v6.14.2, 8.11.2+, v9.11.0+, 10.0.0 and all later versions
2	v8.10.0+, v9.3.0+, 10.0.0 and all later versions
1	v8.6.0+*, v9.0.0+, 10.0.0 and all later versions

更多信息请参考: <https://nodejs.org/api/n-api.html#node-api-version-matrix>

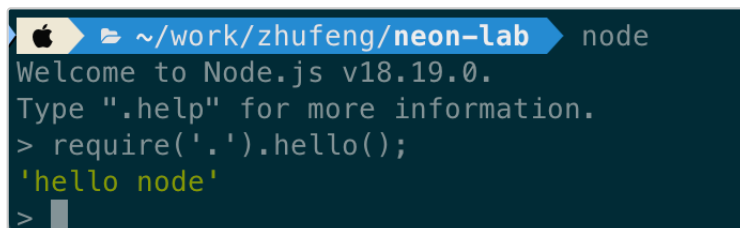
接下来在项目目录执行 `npm install` 安装所需的依赖。

Neon 为我们创建的项目中, 已经包含了一个 “Hello” 的演示。我们来看一下 `src/lib.rs` 中的代码:

```
1 use neon::prelude::*;
2
3 fn hello(mut cx: FunctionContext) -> JsResult<JsString> {
4     Ok(cx.string("hello node"))
5 }
6
7 #[neon::main]
8 fn main(mut cx: ModuleContext) -> NeonResult<()> {
9     cx.export_function("hello", hello)?;
10    Ok(())
11 }
```

虽然这是一个 Library 的包, 但是它依然是包含了一个 `main()` 函数, 不过不同的是, 这个 `main()` 函数上有一个属性宏: `#[neon::main]`, 这个属性宏是从行 1 的代码中引入的。简单的理解, 这个属性宏会使得 Neon 构建项目的时候可以找到入口, 并插入其他的代码。

行 9 的代码中，通过调用 Neon 提供的 `ModuleContext` 类型的 `export_function()` 方法，向 Node.js 导出一个函数，函数的名字叫做 `hello`，对应的是 Rust 代码中的 `hello` 函数。`hello` 函数很简单，返回一个字符串 `hello node`。接下来，我们构建一下这个项目：`npm run build`，成功构建之后，会在项目目录下生成 `index.node` 文件。我们简单的用 `node` 命令行测试一下：



```
~/work/zhufeng/neon-lab node
Welcome to Node.js v18.19.0.
Type ".help" for more information.
> require('.').hello();
'hello node'
>
```

可以看到，Rust 中的 `hello` 函数的返回值输出在命令行终端上了。

Rust 和 JavaScript 交互

前面的例子很简单，从 Rust 一侧返回一个字符串到 JavaScript。下面我们设计一个带有参数的。假定向 JavaScript 提供一个函数：

```
1  /**
2   * @param {string} bucketName
3   * @param {string} objectKey
4   * @param {number|undefined} ttlSeconds 签名有效时间，秒为单位
5   */
6  function requestStsToken(bucketName, objectKey, ttlSeconds)
```

扩展一下：对于数量参数，也就是既包含“数”又包含“量”的参数，通常在参数名中标明所使用的单位，避免调用者不知道该使用什么单位计量而传入了错误的参数。例如上面的 `ttlSeconds` 在参数名中标明了单位为“秒”。

我们使用 Rust 实现这个函数，然后导出给 JavaScript 一侧使用。

```
1
2  fn request_sts_token(mut cx: FunctionContext) ->
   JsResult<JsObject> {
```

```

3      let bucket_name = cx.argument::(&mut cx) => {
7              let nv = hv.downcast::(&mut
cx).unwrap_or(cx.number(3600));
8              let n = nv.value(&mut cx);
9              n as u32
10         },
11         _ => 3600u32
12     };
13
14     let token = cx.string("this is a dummy token");
15     let bucket_name = cx.string(bucket_name);
16     let object_key = cx.string(object_key);
17     let ttl_seconds = cx.number(ttl_seconds);
18
19     let obj = cx.empty_object();
20     obj.set(&mut cx, "token", token)?;
21     obj.set(&mut cx, "bucketName", bucket_name)?;
22     obj.set(&mut cx, "objectKey", object_key)?;
23     obj.set(&mut cx, "ttlSeconds", ttl_seconds)?;
24     Ok(obj)
25 }
26
27 #[neon::main]
28 fn main(mut cx: ModuleContext) -> NeonResult<()> {
29     cx.export_function("hello", hello)?;
30     cx.export_function("requestStsToken", request_sts_token)?;
31     Ok(())
32 }

```

行 3 到行 12 读取从 JavaScript 一侧传入的参数。其中第一个参数 `bucketName` 和 第二个参数 `objectKey` 是必要参数，所以我们使用 `cx.argument:: 来获取；第三个参数 ttlSeconds 是可选参数，所以我们需要使用 cx.argument_opt() 方法来获取，然后判断是否是一个 JsNumber 类型，如果是，就取出来其中的值，如果不是，就给`

一个默认值。

行 14 到行 17 构造返回给 JavaScript 的数据，使用 `cx.string`、`cx.number` 构建相应的数值。行 19 使用 `cx.empty_object()` 构造一个空对象，行 20 到行 23 设置这个对象的属性。可以看到，设置属性名的时候采用 JavaScript 的命名规范。

行 30 导出这个函数：`cx.export_function("requestStsToken", request_sts_token)?`。

还是使用 Node 命令行模式简单测试一下：

```
node
Welcome to Node.js v18.19.0.
Type ".help" for more information.
> const lab = require(".");

> lab
{
  hello: [Function: neon_lab::hello],
  requestStsToken: [Function: neon_lab::request_sts_token]
}
> lab.requestStsToken("");
Uncaught TypeError: not enough arguments
> lab.requestStsToken("bucket", "path/to/my/file.ext");
{
  token: 'this is a dummy token',
  bucketName: 'bucket',
  objectKey: 'path/to/my/file.ext',
  ttlSeconds: 3600
}
> lab.requestStsToken("bucket", "path/to/my/file.ext", 1280);
{
  token: 'this is a dummy token',
  bucketName: 'bucket',
  objectKey: 'path/to/my/file.ext',
  ttlSeconds: 1280
}
> lab.requestStsToken("bucket", "path/to/my/file.ext", "ssss");
{
  token: 'this is a dummy token',
  bucketName: 'bucket',
  objectKey: 'path/to/my/file.ext',
  ttlSeconds: 3600
}
>
```

Neon 提供了一系列的方法，可以在 Rust 一侧构建返回给 JavaScript 一侧的值。

```
1 // 数值类型
2 let i: Handle<JsNumber> = cx.number(42);
3 let f: Handle<JsNumber> = cx.number(3.14);
4 let size: usize = std::mem::size_of::<u128>();
5 let n = cx.number(size as f64)
6
7 // 字符串
8 let s: Handle<JsString> = cx.string("foobar");
9
10 // 布尔
11 let b: Handle<JsBoolean> = cx.boolean(true);
12
13 // undefined
14 let u: Handle<JsUndefined> = cx.undefined();
15
16 // null
17 let n: Handle<JsNull> = cx.null();
18
19 // 数组
20 let a: Handle<JsArray> = cx.empty_array();
21
22 // 向数组中插入元素
23 let s = cx.string("hello!");
24 a.set(&mut cx, 0, s)?;
25
26 // 对象
27 let obj: Handle<JsObject> = cx.empty_object();
28
29 // 设置对象属性
30 let obj = cx.empty_object();
31 let age = cx.number(35);
32 obj.set(&mut cx, "age", age)?;
```

示例：计算文件 **SHA256** 哈希值

下面我们通过一个计算文件 SHA256 哈希值的程序来比较一下 JavaScript 代码和 Rust 编写的 Node.js 模块。

在 `neon-lab` 项目中，增加计算 SHA256 的 crate: `ring`，以及将二进制转换成十六进制字符串的 crate `hex`：

```
1 $ cargo add ring hex
```

然后，在 `src/lib.rs` 中增加相应的代码：

```
1 fn digest_file(mut cx: FunctionContext) -> JsResult<JsString> {
2     let file_name = cx.argument::(<JsString>(0)?.value(&mut cx));
3     let file = File::open(&file_name).unwrap();
4     let mut reader = BufReader::new(file);
5
6     let mut ctx =
7 ring::digest::Context::new(&ring::digest::SHA256);
8
9     loop {
10         let n = reader.read(&mut buffer).unwrap();
11         if n == 0 {
12             break;
13         }
14         ctx.update(&buffer[..n]);
15     }
16
17     let result = ctx.finish();
18     let hash_hex = hex::encode(result);
19
20     Ok(cx.string(hash_hex))
21 }
22
23 #[neon::main]
24 fn main(mut cx: ModuleContext) -> NeonResult<()> {
25     cx.export_function("hello", hello)?;
26     cx.export_function("requestStsToken", request_sts_token)?;
```

```
27     cx.export_function("digestFile", digest_file)?;  
28     Ok(())  
29 }
```

在 Rust 代码中，我们定义了 `digest_file()` 的函数，然后向外导出为 `digestFile()`。接下来我们要测试一下这个模块是否工作正常。

新建一个 Node.js 项目 `neon-lab-test`，使用 `npm` 或者 `yarn` 初始化一下。然后在 `package.json` 中，将 `neon-lab` 以本地文件的形式加入到依赖项中：

```
1  {  
2    "name": "neon-lab-test",  
3    "version": "1.0.0",  
4    "main": "index.js",  
5    "license": "MIT",  
6    "dependencies": {  
7      "neon-lab": "file:../neon-lab"  
8    }  
9  }
```

然后再次执行 `npm install` 或者 `yarn install` 将 `neon-lab` 拉到 `neon-lab-test/node_modules` 中来。实际上不会真正复制文件，只是在 `node_modules` 下面创建了一个文件夹链接（可以理解成“快捷方式”）。然后在 `neon-lab-test` 项目下新建一个 `test.js`，在这个文件中，我们使用 JavaScript 实现一个计算文件 SHA256 的，然后在使用 `neon-lab` 提供的函数计算一次。比对一下不同的实现方式的性能是否有所区别。

`neon-lab-test/test.js` 的内容如下：

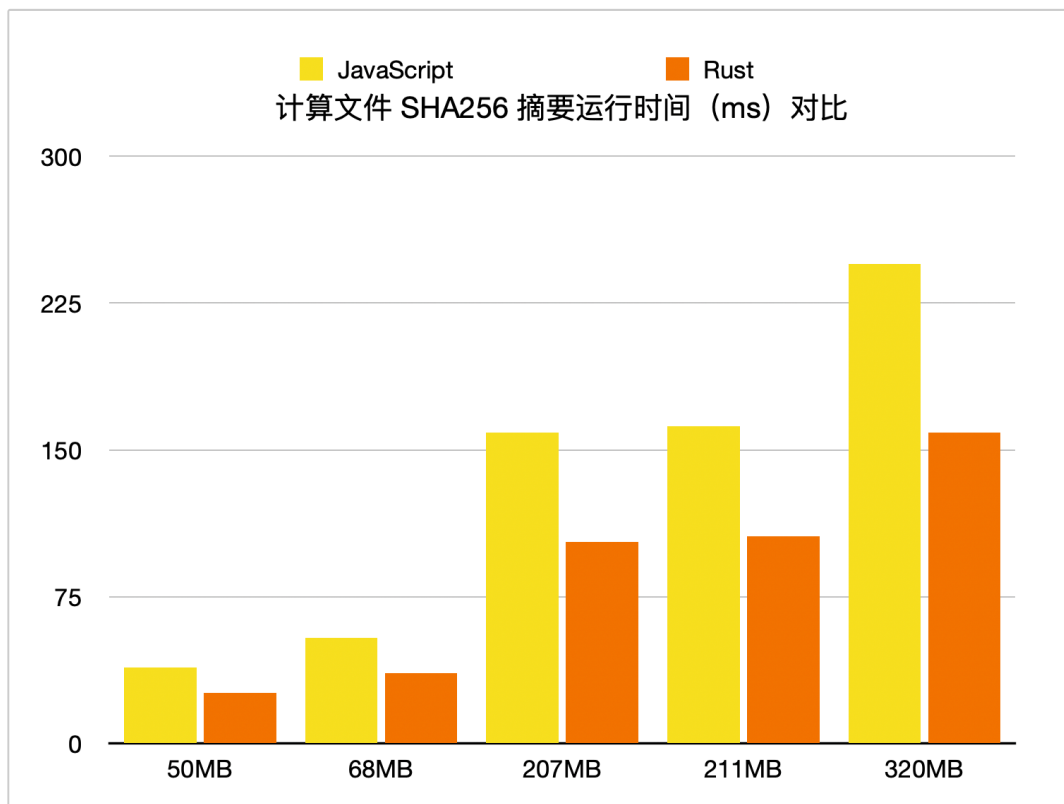
```
1  const crypto = require("node:crypto");  
2  const fsPromises = require("fs/promises");  
3  const neonLab = require("neon-lab");  
4  
5  async function digestFile(filename) {  
6    const hasher = crypto.createHash("SHA256");  
7    const fin = await fsPromises.open(filename, "r");  
8    const buf = new Buffer.alloc(1024 * 64);
```

```
9     for (;;) {
10         const { bytesRead } = await fin.read(buf, 0, buf.length,
null);
11         if (bytesRead === 0) {
12             break;
13         }
14
15         hasher.update(buf.subarray(0, bytesRead));
16     }
17     await fin.close();
18     return hasher.digest("hex");
19 }
20
21 (async function() {
22     const filenames = [
23         "/path/to/file1.zip",
24         "/path/to/file2.zip",
25         "/path/to/file3.zip",
26         "/path/to/file4.zip",
27         "/path/to/file5.zip",
28     ];
29
30     console.log("calculate file hash using JavaScript");
31
32     for (let f of filenames) {
33         const stat = await fsPromises.stat(f);
34         const startMs = Date.now();
35         const ret = await digestFile(f);
36         const endMs = Date.now();
37         const speed = (stat.size / 1024 / 1024) * 1000 / (endMs
- startMs);
38         console.log(`file size: ${stat.size} bytes, time used:
${Date.now() - startMs} ms, speed: ${speed.toFixed(2)} MB/s,
ret: ${ret}`);
39     }
40
41     console.log("calculate file hash using Rust");
42     for (let f of filenames) {
43         const stat = await fsPromises.stat(f);
```

```
44     const startMs = Date.now();
45     const ret = neonLab.digestFile(f);
46     const endMs = Date.now();
47     const speed = (stat.size / 1024 / 1024) * 1000 / (endMs
- startMs);
48     console.log(`file size: ${stat.size} bytes, time used:
    ${Date.now() - startMs} ms, speed: ${speed.toFixed(2)} MB/s,
    ret: ${ret}`);
49 }
50 }());
```

上面的代码并不难理解（请无视我两次查询文件大小的啰嗦代码 :P），我们直接执行，看结果：

```
1 calculate file hash using JavaScript
2 file size: 53023930 bytes, time used: 39 ms, speed: 1296.60
  MB/s, ret:
  316932411a0e96262f0093ee732a271313820c43a0dd03978a34d67c7bc4741b
3 file size: 71466438 bytes, time used: 54 ms, speed: 1262.14
  MB/s, ret:
  cdf342010cf818d36a860bce6238ef204c7112a8ad711757125e977d3fcba632
4 file size: 217418549 bytes, time used: 159 ms, speed: 1304.07
  MB/s, ret:
  7c9b34402b4d26fbc80c8acae9f82c52708bdf2c547135e74d499a3d4aed5124
5 file size: 222027171 bytes, time used: 162 ms, speed: 1307.05
  MB/s, ret:
  9d24d15fbf7482ea9a9629a1d0677aa47baa21ab3c3b15be7cba950074d7f7e8
6 file size: 335982861 bytes, time used: 245 ms, speed: 1307.83
  MB/s, ret:
  ff5218ed0aff42da570e4459cd95fa5fe103da86e6e727ab88414379db2d89e9
7 calculate file hash using Rust
8 file size: 53023930 bytes, time used: 26 ms, speed: 1944.91
  MB/s, ret:
  316932411a0e96262f0093ee732a271313820c43a0dd03978a34d67c7bc4741b
9 file size: 71466438 bytes, time used: 36 ms, speed: 1893.21
  MB/s, ret:
  cdf342010cf818d36a860bce6238ef204c7112a8ad711757125e977d3fcba632
10 file size: 217418549 bytes, time used: 103 ms, speed: 2013.07
  MB/s, ret:
  7c9b34402b4d26fbc80c8acae9f82c52708bdf2c547135e74d499a3d4aed5124
11 file size: 222027171 bytes, time used: 106 ms, speed: 1997.56
  MB/s, ret:
  9d24d15fbf7482ea9a9629a1d0677aa47baa21ab3c3b15be7cba950074d7f7e8
12 file size: 335982861 bytes, time used: 159 ms, speed: 2015.21
  MB/s, ret:
  ff5218ed0aff42da570e4459cd95fa5fe103da86e6e727ab88414379db2d89e9
```



从上图对比来看，Rust 还是要比 JavaScript 快。但是如果我们观察绝对值，320MB 的文件 JavaScript 的运行时间也不过才 245ms，如果不是压力特别大的场景，这个也不是不能接受。换句话说，Node.js 运行时已经很高效率了。另外，还需要指出的是，并不是 Rust 一定会比 JavaScript 快。最一开始，我在 Rust 中使用的是 `sha2 crate`，运行效率远远低于 JavaScript 的代码。上网查了一些资料，据说是因为这个 `crate` 对安全要求较高导致性能下降。后来换成了 `ring crate`。所以，很多时候，我们纠结语言本身的性能问题，实际上还要关注不同的库的性能差异。

扩展知识：在面试的时候，我有时候会问：你知道的加密算法有哪些？有很多同学回答说：MD5，还有的说：base64。这种回答很不严谨甚至离谱。在信息处理领域，有 3 个概念，他们是独立的，但是又经常一起出现，但是不可以混为一谈：

1. 加密/解密 (Encrypt/Decrypt)。加解密必须是需要密钥的，必须是无损的。也就是说，解密后的信息必须和原始输入的信息要完全一样。加解密的目的是保护信息安全。从密钥类型可以分为两类：对称加密和非对称加密。
 - a. 对称加密是指加密用的密钥和解密用的密钥是一样的，例如常见的对称加密算法有：AES，DES，3DES，Blowfish 等。

- b. 非对称加密是指加密的密钥和解密的密钥不一样，这一对儿钥匙叫做公钥和私钥，公钥用来加密数据，私钥用来解密数据。常见的非对称加密算法有：RSA，DSA，ECC等。
- 2. 摘要（Digest）。摘要是提取一段信息中的独特特征，它是不可逆的。也就是说，你不能从摘要的输出结果反推摘要的输入信息。同一个摘要算法，无论输入的信息长度是多少，输出的结果都是一样长度的。所以它是有损的，不可逆的。经常被用来检查信息的完整性，因为一旦信息被篡改哪怕一个字节，摘要的结果都是天壤之别。常见的摘要算法有：MD5，SHA-1，SHA-2（SHA-224, SHA-256, SHA-384, SHA-512）等。
- 3. 编码/解码（Encoding/Decoding）。编解码不需要密钥，且是无损的，算法都是公开的。编解码的目的是为了便于信息传递。例如二进制的数字很难在 URL 中携带，所以可以使用 base64 算法将数据转换成可见 ASCII 字符之后再传输。常见的编解码算法有：base64，hex 等。

回顾

- 1. 使用 Rust 开发 Node.js 模块的工具链：Neon
- 2. 如何在 Rust 中接收参数和返回不同类型的值
- 3. Rust 和 JavaScript 的性能对比

