D2 前端技术论坛
D2 FRONTEND TECHNOLOGY FORUM
精心

第十六届D2前端技术论坛

用 Rust 和 Node-API 开发高性能 Node.js 模块

龙逸楠

Contents
目录

01

Node.js 中的 Native Addon

Node.js 中的 Native Addon

https://xcoder.in/2017/07/01/nodejs-addon-history/

Native Addon 的本质

- .node 格式的文件
- 是二进制文件
- 是一个动态链接库 (Windows DLL/Unix dylib/Linux so)

Node.js 中的 Native Addon

https://github.com/nodejs/node/blob/2cc7a91a5d855b4ff78f21f1bb8d4e55131d0615/lib/internal/modules/cjs/loader.js#L1172

.node 格式文件

Node.js 对 .node 格式文件的加载方式:

```
// Native extension for .node
Module._extensions['.node'] = function(module, filename) {
  if (policy?.manifest) {
    const content = fs.readFileSync(filename);
    const moduleURL = pathToFileURL(filename);
    policy.manifest.assertIntegrity(moduleURL, content);
  }
  // Be aware this doesn't use `content`
  return process.dlopen(module, path.toNamespacedPath(filename));
};
```

Node.js 中的 Native Addon

https://github.com/nodejs/node/blob/2cc7a91a5d855b4ff78f21f1bb8d4e55131d0615/lib/internal/modules/cjs/loader.js#L1172

.node 格式文件

Node.js 对 .node 格式文件的加载方式:

modules/cjs/loader.js

```
// Native extension for .node
Module._extensions['.node'] = function(module, filename) {
  if (policy?.manifest) {
    const content = fs.readFileSync(filename);
    const moduleURL = pathToFileURL(filename);
    policy.manifest.assertIntegrity(moduleURL, content);
  }
  // Be aware this doesn't use `content`
  return process.dlopen(module, path.toNamespacedPath(filename));
};
```

- Node.js 默认将 Native Addon 当作 CommonJS 格式模块加载
- 使用 process.dlopen 函数加载内容
- 目前源码包含对 ESM 的 import assets 逻辑的兼容，但实际上 Node.js 目前的最新版 (17.1.0) 只支持 json 类型的 import assert

Node.js 中的 Native Addon

https://github.com/Brooooooklyn/blake-hash/releases/tag/v1.3.0

Native Addon 文件内容

在 Linux 上使用 hexyl 查看 blake.darwin-x64.node 文件内容

```
hexyl -n 256 node_modules/@napi-rs/blake-hash-linux-x64-gnu/blake.linux-x64-gnu.node
```

00000000	7f 45 4c 46 02 01 01 00	00 00 00 00 00 00 00 00	•ELF•••0	00000000
00000010	03 00 3e 00 01 00 00 00	80 e5 01 00 00 00 00 00	•0>0•000	xx•00000
00000020	40 00 00 00 00 00 00 00	b0 59 0a 00 00 00 00 00	@0000000	xY_00000
00000030	00 00 00 00 40 00 38 00	0b 00 40 00 20 00 1f 00	0000@080	•0@0 0•0
00000040	06 00 00 00 04 00 00 00	40 00 00 00 00 00 00 00	•000•000	@0000000
00000050	40 00 00 00 00 00 00 00	40 00 00 00 00 00 00 00	@0000000	@0000000
00000060	68 02 00 00 00 00 00 00	68 02 00 00 00 00 00 00	h•000000	h•000000
00000070	08 00 00 00 00 00 00 00	01 00 00 00 04 00 00 00	•0000000	•000•000
00000080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00000000	00000000
00000090	00 00 00 00 00 00 00 00	44 d5 01 00 00 00 00 00	00000000	D×•00000
000000a0	44 d5 01 00 00 00 00 00	00 10 00 00 00 00 00 00	D×•00000	0•000000
000000b0	01 00 00 00 05 00 00 00	44 d5 01 00 00 00 00 00	•000•000	D×•00000
000000c0	44 e5 01 00 00 00 00 00	44 e5 01 00 00 00 00 00	D×•00000	D×•00000
000000d0	ec 2f 08 00 00 00 00 00	ec 2f 08 00 00 00 00 00	×/•00000	×/•00000
000000e0	00 10 00 00 00 00 00 00	01 00 00 00 06 00 00 00	0•000000	•000•000
000000f0	30 05 0a 00 00 00 00 00	30 25 0a 00 00 00 00 00	0•_00000	0%_00000

Node.js 中的 Native Addon

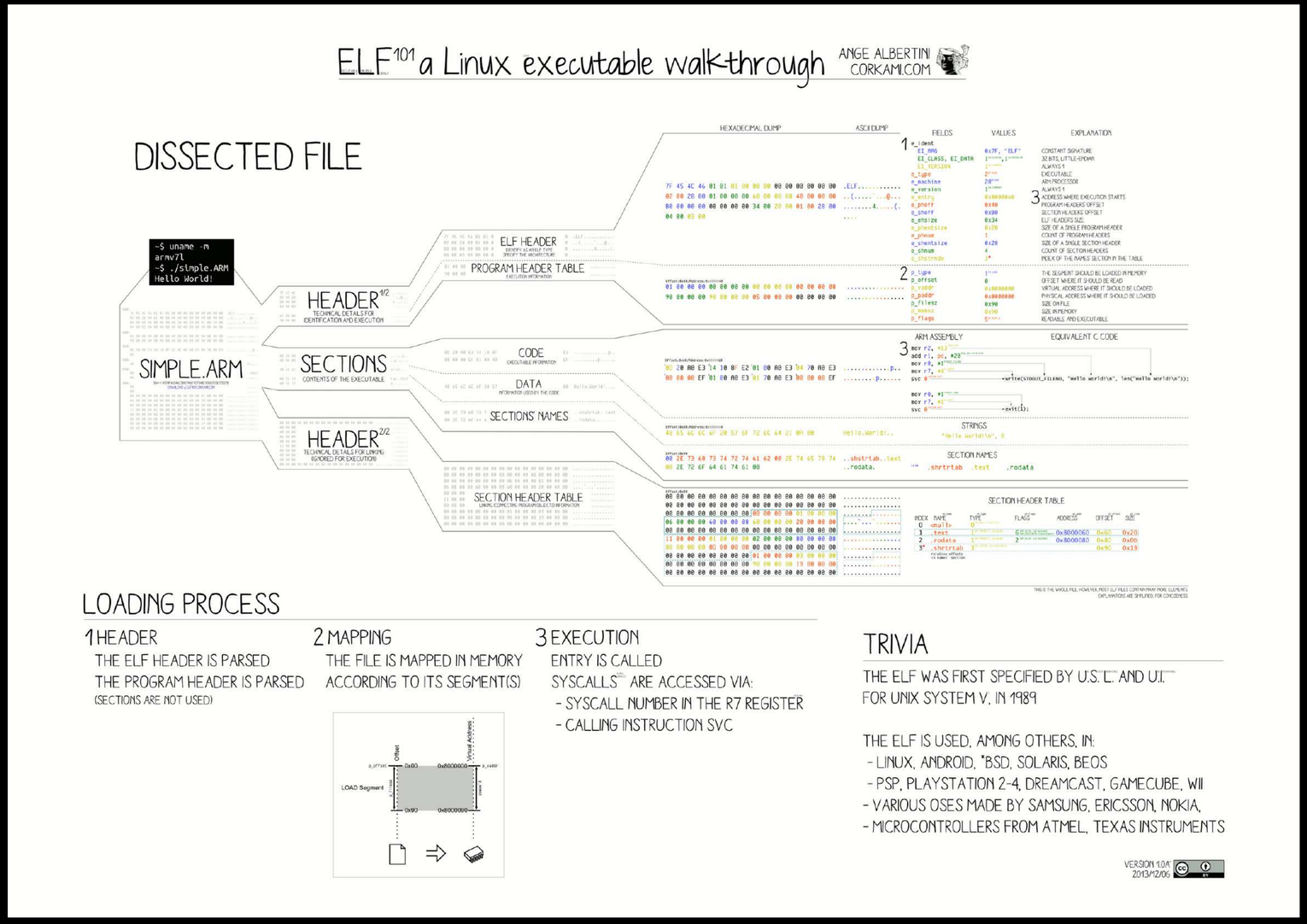
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Native Addon 文件内容

在 Linux 上使用 hexyl 查看 blake.darwin-x64.node 文件内容

Node.js 中的 Native Addon

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format#/media/File:ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png



Node.js 中的 Native Addon

https://github.com/nodejs/node/blob/v10.23.0/src/node.cc#L1232

Native Addon 文件内容

```
extern "C" {
  napi_value napi_register_module_v1(napi_env env, napi_value export);
}
```

1. process.dlopen
2. 初始化 env 与 module.exports 对象，并传入 napi_register_module_v1 函数
3. 拿到返回的 module.exports 对象，后续将 native addon 当作普通的 CommonJS 模块处理

02

Rust 在前端/Node.js

Rust 在前端/Node.js

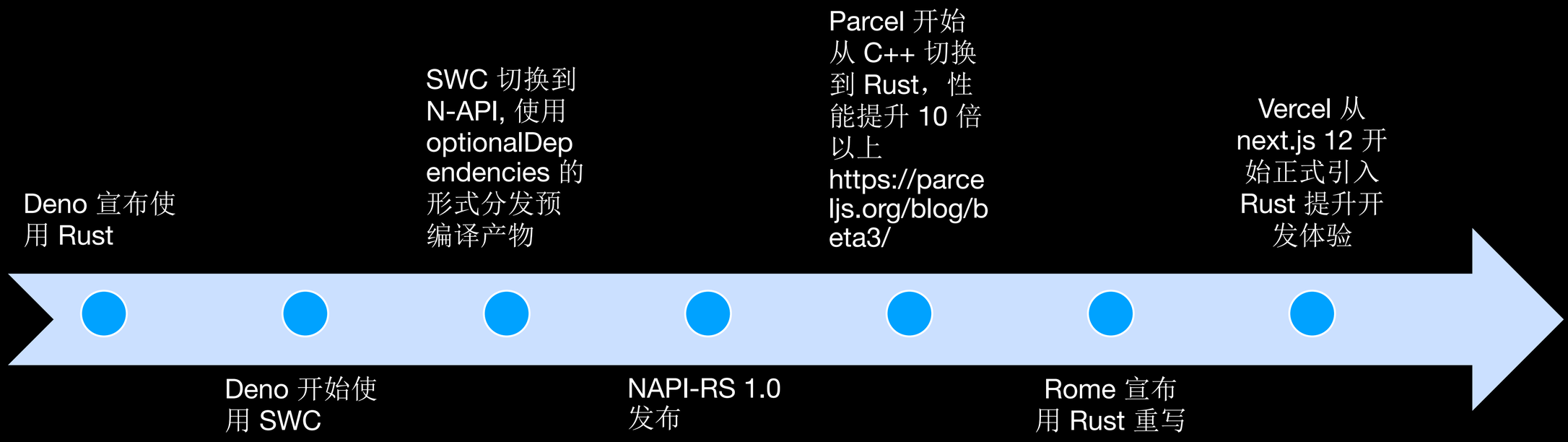
https://leerob.io/blog/rust

- Rust 在前端/Node.js 领域应用的历史
- 流行的 Rust 工具与库
- 作为工具与包的作者，如何发布 Rust 编写的前端/Node.js 工具链

Node.js 中的 Native Addon

https://github.com/denoland/deno/issues/208#issuecomment-395739171

Rust 在前端/Node.js 领域应用的历史



目前流行的 Rust 编写的前端/Node.js 库

- Parcel 使用 SWC 与 NAPI-RS 实现 JavaScript/TypeScript 编译与优化逻辑，如 Tree-shaking 与 Scope hoist，minify 等。source map 模块，png/jpeg 无损压缩，css 压缩与编译模块也是由 Rust + NAPI-RS 编写。
- Rome 是一个集 Linter/Compiler/Bundler/Formatter/Test Runner 为一体的工具链，目的是取代 Babel, ESLint, Webpack, Jest, Prettier 等前端工具链。 最初由 TypeScript 编写，现已全面切换到 Rust。
- Next.js 是 Vercel 出品的前后端一体化 React 框架，目前它将所有定制的 Babel custom transformer 与默认的 TypeScript/JavaScript 编译部分切换到了 Rust。
- Prisma 是下一代面向 TypeScript/JavaScript 的 ORM，Prisma 的数据库连接与查询引擎使用 Rust 与 NAPI-RS 实现。
- Ali ICE，Shopify FE 等团队使用 SWC 替换自定义 Babel plugin。

如何发布 Rust 编写的前端/Node.js 工具链

流行的 Native Addon 发布方式

- 只发布源码，通过 postinstall 脚本执行编译脚本在安装的时候编译成二进制。
- 在 CI 阶段预编译不同平台的二进制文件，在 postinstall 的时候下载下来。
- 在 CI 阶段预编译不同平台的二进制文件，将不同平台的二进制文件单独发布成包，然后通过 package.json 中的 cpu 与 os 字段限制这些包可安装的平台。最终将所有平台的独立包指定为最终发布包的 optionalDependencies，主流的包管理工具比如 npm/pnpm/yarn 就可以通过预设的 cpu 与 os 字段只下载对应平台的二进制包。

Node.js 中的 Native Addon

只发布源码

优点

- 对于库开发者来说，无需任何额外的工具链维护成本，只需要改动源码就可以轻松发布。
- 在用户的机器上用用户的工具链编译，兼容性是最好的。只要编译通过，基本上就可以运行，不会有系统组件比如 GLIBC libstdc++ 等不兼容的情况。

缺点

- 大型项目的编译时间过长，并且占用大量 CPU 资源。
- 可能需要用户安装编译相关的工具链，比如 GCC CMAKE LLVM NASM Rust 等，前端开发者一般缺乏相关经验，对于很多项目很难设置正确。
- 编译期间可能产生大量中间文件，不正确清理极其占用硬盘空间。
- 让很多 CI 的缓存逻辑无法工作，不得不重新全量下载 & 编译。
- postinstall 脚本不安全，未来可能会被默认禁用 https://github.com/npm/rfcs/pull/488

Node.js 中的 Native Addon

发布预编译产物，postinstall 的时候下载

优点

- 对于用户来说，减少了编译成本。
- 不需要考虑编译时工具链等问题。
- 没有中间编译产物，节省磁盘空间。

缺点

- postinstall 安装产物不受 lockfile 控制，会破坏很多 CI 的缓存逻辑，降低缓存复用率。
- 存放预编译产物的 CDN 无法照顾到全球的用户，比如大量项目使用 GitHub Releases 作为预编译二进制存放的服务，在国内访问困难。
- 因为要安装时下载，需要引入运行时不需要的依赖来完成下载，比如 node-pre-gyp 等。
- postinstall 脚本不安全，未来可能会被默认禁用 https://github.com/npm/rfcs/pull/488

Node.js 中的 Native Addon

https://cdn.jsdelivr.net/npm/@napi-rs/blake-hash/package.json

optionalDependencies

```
{
  "name": "@napi-rs/blake-hash",
  "version": "1.3.0",
  "main": "index.js",
  "optionalDependencies": {
    "@napi-rs/blake-hash-win32-x64-msvc": "1.3.0",
    "@napi-rs/blake-hash-darwin-x64": "1.3.0",
    "@napi-rs/blake-hash-linux-x64-gnu": "1.3.0",
    "@napi-rs/blake-hash-darwin-arm64": "1.3.0",
    "@napi-rs/blake-hash-android-arm64": "1.3.0",
    "@napi-rs/blake-hash-linux-arm64-gnu": "1.3.0",
    "@napi-rs/blake-hash-linux-arm64-musl": "1.3.0",
    "@napi-rs/blake-hash-win32-arm64-msvc": "1.3.0",
    "@napi-rs/blake-hash-linux-arm-gnueabihf": "1.3.0",
    "@napi-rs/blake-hash-linux-x64-musl": "1.3.0",
    "@napi-rs/blake-hash-freebsd-x64": "1.3.0",
    "@napi-rs/blake-hash-win32-ia32-msvc": "1.3.0"
  }
}
```

```
{
  "name": "@napi-rs/blake-hash-darwin-arm64",
  "version": "1.3.0",
  "os": [
    "darwin"
  ],
  "cpu": [
    "arm64"
  ],
  "main": "blake.darwin-arm64.node",
  "files": [
    "blake.darwin-arm64.node"
  ],
  "license": "MIT",
  "engines": {
    "node": "≥ 10"
  }
}
```

Node.js 中的 Native Addon

optionalDependencies

优点

- 无感知，与安装普通 JavaScript 编写的 npm 包体验上没有区别。
- lockfile 与 cache 友好。
- 没有 postinstall 脚本。

缺点

- CI 配置复杂，发布流程繁琐。
- CI 环境链接的系统组件与用户使用环境可能存在不兼容情况，比如 GLIBC 版本。https://github.com/swc-project/swc/issues/1410

Node.js 中的 Native Addon

使用 Rust 开发 npm 包，如何选择发布模式

optionalDependencies

- Rust 工具链体积巨大，编译缓慢，中间产物庞大，不适合在用户侧编译。
- postinstall 存在潜在的安全风险，前途未卜。
- 在 postinstall 下载预编译产物对 lockfile 与构建缓存非常不友好。
- 大量 Node 应用在企业私有网络内构建，无法访问外部 CDN，postinstall 下载失败率非常高。
- CI 的复杂配置与相关工具链的维护成本被 NAPI-RS cover 了，作为包作者无需关心。

主流包含 Native 工具链的包比如 esbuild SWC 和 next.js 都采用了 optioanlDependencies 的形式发布。https://github.com/evanw/esbuild/pull/1621

03

NAPI-RS

NAPI-RS

https://napi.rs

A framework for building compiled Node.js add-ons in Rust via Node-API

- 介绍
- 使用场景
- 与 neon/node-bindgen 对比

NAPI-RS

https://napi.rs

介绍

- 简单易用的 API
- TypeScript .d.ts 文件生成
- 深度集成 Rust Tokio 异步运行时，轻松复用 Rust 异步代码
- 开箱即用的 cli，开发者无需为跨平台编译、自动化 CI 烦恼

NAPI-RS

https://github.com/napi-rs/node-rs/tree/main/packages/crc32

简单易用的 API

```rust
use crc32fast::Hasher;
use napi::bindgen_prelude::*;
use napi_derive::napi;

#[napi]
pub fn crc32(
  input_data: Either<String, Buffer>,
  initial_state: Option<u32>,
) -> u32 {
  let mut hasher = Hasher::new_with_initial(
    initial_state.unwrap_or(0)
  );
  hasher.update(match &input_data {
    Either::A(s) => s.as_bytes(),
    Either::B(b) => b.as_ref(),
  });
  hasher.finalize()
}
```

```ts
// index.d.ts in @node-rs/crc32
export function crc32(
  inputData: string | Buffer,
  initialState?: number | undefined | null
): number

// index.mjs
import { crc32 } from '@node-rs/crc32'

// index.js
const { crc32 } = require('@node-rs/crc32')
```

NAPI-RS

https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

TypeScript 类型生成

- 支持 Rust 的 struct + impl 生成 TypeScript Class 对应文件
- 联合类型与可选类型
- 异步类型生成，Rust 的 async fn 到 TypeScript 的 () => Promise<T>
- Rust 代码注释透传到 .d.ts 文件
- 为无法自动精确生成的 Rust 函数重写 TypeScript 类型

NAPI-RS

https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

TypeScript 类型自动生成

```rust
/// default enum values are continuos i32s start from 0
#[napi]
pub enum Kind {
  /// Barks
  Dog,
  /// Kills birds
  Cat,
  /// Tasty
  Duck,
}

#[napi]
pub struct Animal {
  #[napi(readonly)]
  /// Kind of animal
  pub kind: Kind,
  name: String,
}

#[napi]
impl Animal {
  /// This is the constructor
  #[napi(constructor)]
  pub fn new(kind: Kind, name: String) -> Self {
    Animal { kind, name }
  }
```

```rust
  /// This is a factory method
  #[napi(factory)]
  pub fn with_kind(kind: Kind) -> Self {
    Animal {
      kind,
      name: "Default".to_owned(),
    }
  }

  #[napi(getter)]
  pub fn get_name(&self) -> &str {
    self.name.as_str()
  }

  #[napi(setter)]
  pub fn set_name(&mut self, name: String) {
    self.name = name;
  }
```

```rust
  /// This is a
  /// multi-line comment
  /// with an emoji 🚀
  #[napi]
  pub fn whoami(&self) -> String {
    match self.kind {
      Kind::Dog => {
        format!("Dog: {}", self.name)
      }
      Kind::Cat => format!("Cat: {}", self.name),
      Kind::Duck => format!("Duck: {}", self.name),
    }
  }

  #[napi]
  /// This is static ...
  pub fn get_dog_kind() -> Kind {
    Kind::Dog
  }
}
```

NAPI-RS

https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

TypeScript 类型自动生成

```ts
/**
 * `constructor` option for `struct` requires all fields to be public,
 * otherwise tag impl fn as constructor
 * #[napi(constructor)]
 */
export class Animal {
  /** Kind of animal */
  readonly kind: Kind
  /** This is the constructor */
  constructor(kind: Kind, name: string)
  /** This is a factory method */
  static withKind(kind: Kind): Animal
  get name(): string
  set name(name: string)
  /**
   * This is a
   * multi-line comment
   * with an emoji 🚀
   */
  whoami(): string
  /** This is static... */
  static getDogKind(): Kind
}
```

```ts
/** default enum values are continuos i32s start from 0 */
export enum Kind {
  /** Barks */
  Dog = 0,
  /** Kills birds */
  Cat = 1,
  /** Tasty */
  Duck = 2
}
```

NAPI-RS

https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

TypeScript 类型自动生成

```rust
use std::thread::sleep;

use napi::bindgen_prelude::*;
use napi::Task;

struct DelaySum(u32, u32);

#[napi]
impl Task for DelaySum {
  type Output = u32;
  type JsValue = u32;

  fn compute(&mut self) -> Result<Self::Output> {
    sleep(std::time::Duration::from_millis(100));
    Ok(self.0 + self.1)
  }

  fn resolve(&mut self, _env: napi::Env, output: Self::Output) -> Result<Self::JsValue> {
    Ok(output)
  }
}

#[napi]
fn with_abort_controller(a: u32, b: u32, signal: AbortSignal) -> AsyncTask<DelaySum> {
  AsyncTask::with_signal(DelaySum(a, b), signal)
}

// .d.ts

export function withAbortController(a: number, b: number, signal: AbortSignal): Promise<number>
```

```ts
test('async task with abort controller', async (t) => {
  const ctrl = new AbortController()
  const promise = withAbortController(1, 2, ctrl.signal)
  try {
    ctrl.abort()
    await promise
    t.fail('Should throw AbortError')
  } catch (err: unknown) {
    t.is((err as Error).message, 'AbortError')
  }
})
```

NAPI-RS

https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

TypeScript 类型自动生成

```rust
use napi::bindgen_prelude::{Object, Result};

#[napi(ts_args_type = "a: { foo: number }", ts_return_type = "string[]")]
fn ts_rename(a: Object) -> Result<Object> {
  a.get_property_names()
}
```

```typescript
export function tsRename(a: { foo: number }): string[]
```

NAPI-RS

https://deno.com/blog/v1.9#native-http2-web-server

Tokio 深度集成

```rust
use futures::prelude::*;
use napi::bindgen_prelude::*;
use tokio::fs;

#[napi]
async fn read_file_async(path: String) -> Result<Buffer> {
  fs::read(path)
    .map(|r| match r {
      Ok(content) => Ok(content.into()),
      Err(e) => Err(Error::new(
        Status::GenericFailure,
        format!("failed to read file, {}", e),
      )),
    })
    .await
}
```

NAPI-RS

https://github.com/Brooooooklyn/hns

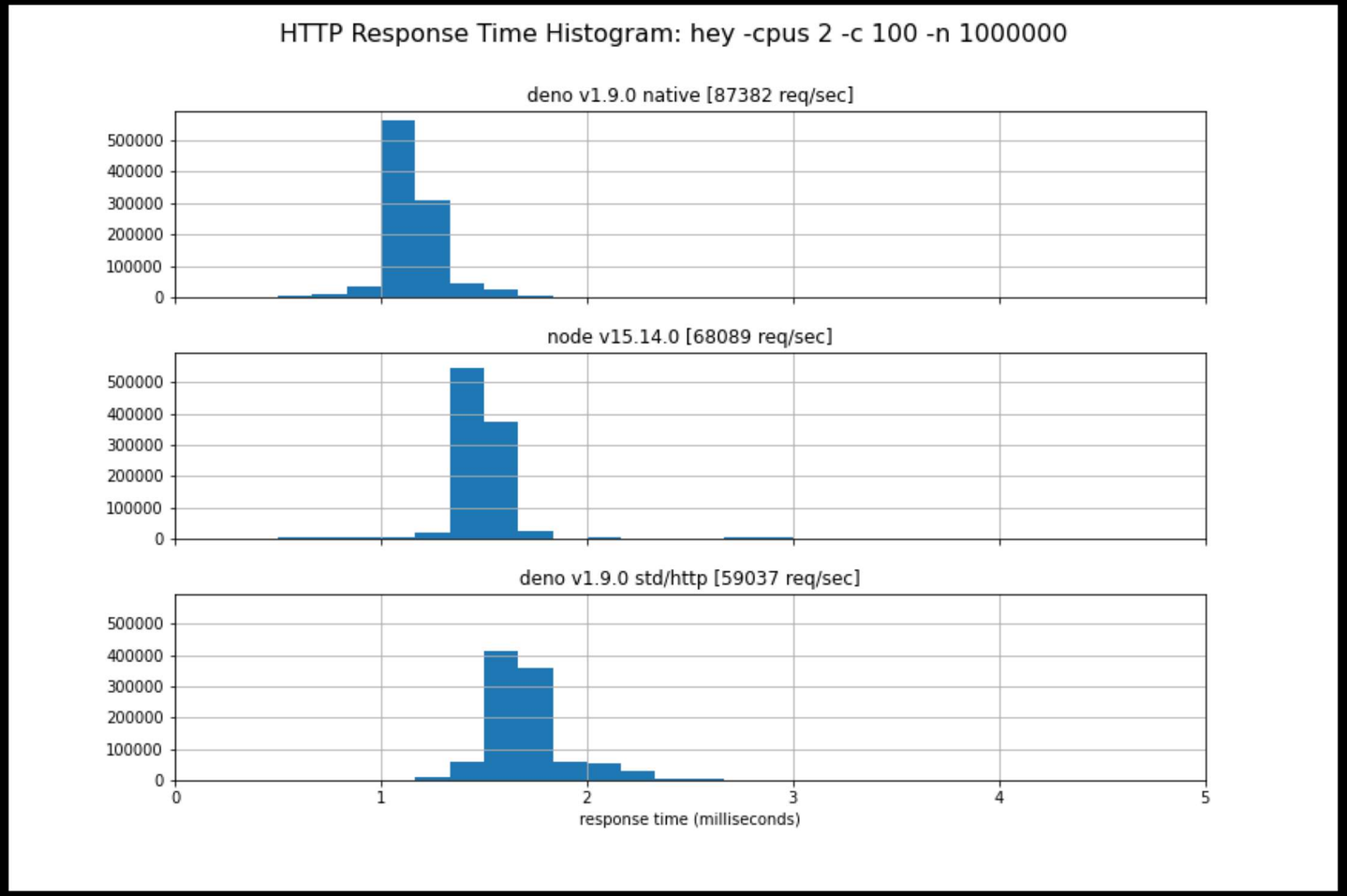
Tokio 深度集成

- 复用 Tokio 生态，比如可以像 deno 那样将 http server 层用 Rust 实现而不是用 JavaScript 实现
- 可以受益于 Tokio 强大的工具链，比如 tokio-console 与 tokio-tracing
- 提前享受 io_uring 等 libuv 未封装的内核新特性

NAPI-RS

https://deno.com/blog/v1.9#native-http2-web-server

Tokio 深度集成



NAPI-RS

https://github.com/tokio-rs/console

Tokio 深度集成

```
connection: http://127.0.0.1:6669/ (CONNECTED)
controls: ←→= select column (sort), ↑↓= scroll, ↵= task details, i = invert sort (highest/lowest), q = quit
Tasks (11) ▶Running (2) ⏸Idle (6)
ID  State Name   Total      Busy       Idle       Polls Target      Fields
 5  ▶           1923.0021s 479.1963ms 1922.5229s  1931 tokio::task kind=task spawn.location=console-subscriber/src/lib.rs:353:25
 0  ▶     blocks 1923.0024s 951.0309s  971.9715s    193 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:26:22
578 ⏸     wait   2.4045s    9.4980µs   2.4045s        1 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:68:54
  1 ⏸     task1  1923.0024s 36.1037ms  1922.9663s   383 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:50:10
  2 ⏸     task2  1923.0024s 16.2832ms  1922.9861s   182 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:53:10
569 ⏸     wait   33.4106s   11.4820µs  33.4106s       1 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:68:54
572 ⏸           24.3964s   7.5895ms   24.3888s      28 tokio::task kind=task spawn.location=<cargo>/hyper-0.14.7/src/common/exec.rs:47:21
573 ⏸           24.3915s   11.0827ms  24.3804s      51 tokio::task kind=task spawn.location=<cargo>/hyper-0.14.7/src/common/exec.rs:47:21
576 ■     wait   11.0010s   35.1450µs  11.0009s       2 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:68:54
577 ■     wait   4.0009s    33.1510µs  4.0008s        2 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:68:54
568 ■     wait   29.0011s   37.3000µs  29.0011s       2 tokio::task kind=task spawn.location=console-subscriber/examples/app.rs:68:54
```

NAPI-RS

https://github.com/tokio-rs/console

Tokio 深度集成

NAPI-RS

https://napi.rs/introduction/getting-started

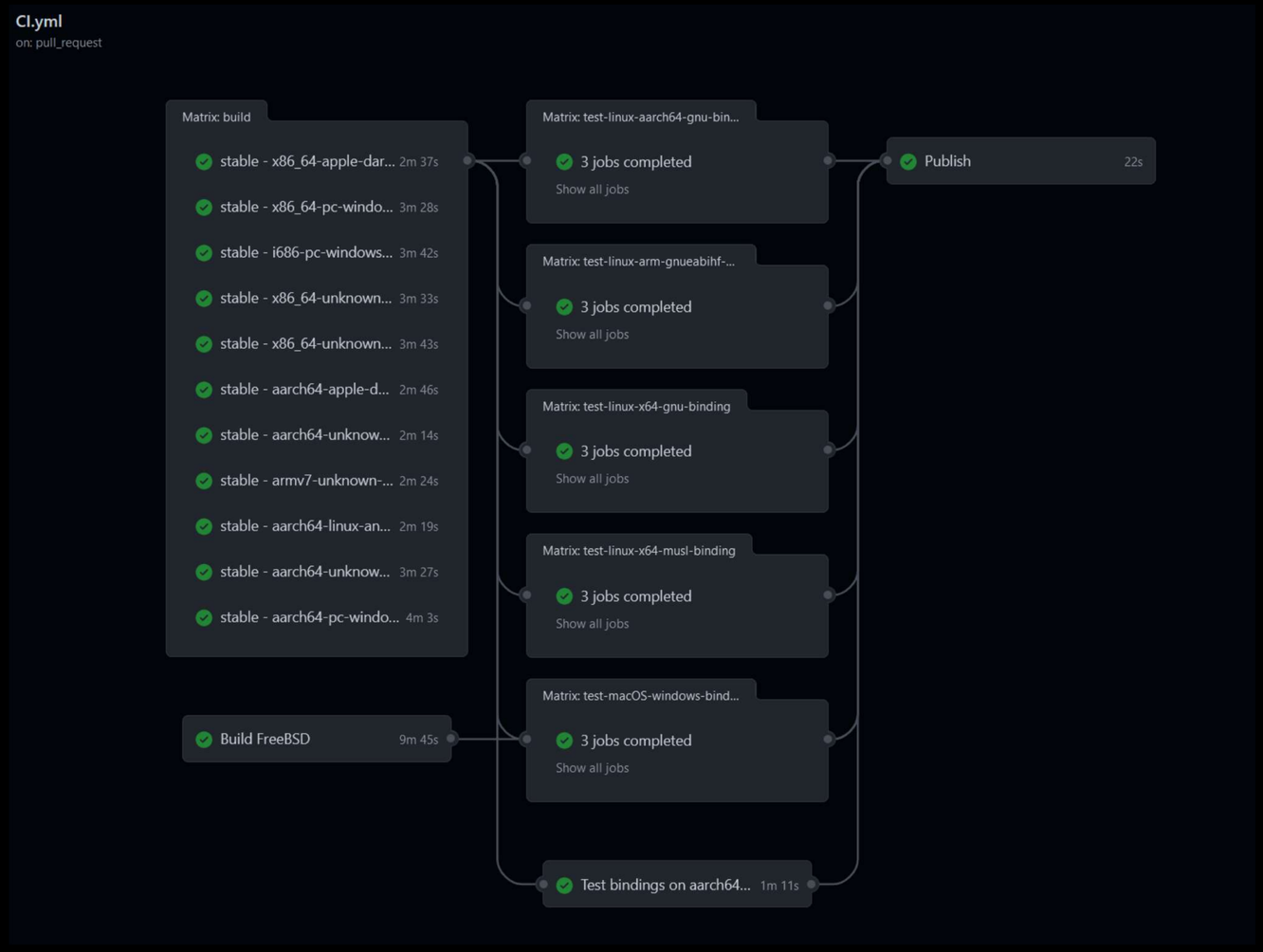
开箱即用的 CLI

- 新建项目与 GitHub Actions 模板，囊括跨平台编译与发布。
- 便捷的 build 命令，预置各种与 Node.js Addons 相关的编译参数，使用便捷。
- optionalDependencies 版本号管理，发布命令，GitHub CI Artifacts 命名与移动等功能全部封装在内，开发者无需关系底层繁琐的实现细节与维护。

NAPI-RS

https://napi.rs/introduction/getting-started

开箱即用的 CLI



NAPI-RS

https://napi.rs/introduction/getting-started

开箱即用的 CLI

Support matrix

	node12	node14	node16
Windows x64	✓	✓	✓
Windows x32	✓	✓	✓
Windows arm64	✓	✓	✓
macOS x64	✓	✓	✓
macOS arm64	✓	✓	✓
Linux x64 gnu	✓	✓	✓
Linux x64 musl	✓	✓	✓
Linux arm gnu	✓	✓	✓
Linux arm64 gnu	✓	✓	✓
Linux arm64 musl	✓	✓	✓
Android arm64	✓	✓	✓
FreeBSD x64	✓	✓	✓

NAPI-RS

https://napi.rs

使用场景

- 性能敏感的计算场景，输入输出简单。
- 需要使用 Node.js 不支持的 Native 特性，比如特定的 CPU 指令集，调用 GPU、USB 等，调用系统 API 等。
- 在 Server 端共享跨语言与框架的代码，比如 Prisma 使用 Rust 开发核心的 query engine，再封装到 Node.js Go 等。
- 利用 Rust 现代化的工具链 (包管理、跨平台编译等) 粘合现有的 C++ 代码，比如 https://github.com/Brooooooklyn/canvas

NAPI-RS

https://github.com/Brooooooklyn/rust-to-nodejs-overhead-benchmark

使用场景

调用开销

```
Running "Sum" suite...
Progress: 100%

  JavaScript:
    1 187 168 789 ops/s, ±0.53%   | fastest

  napi-rs:
    35 065 689 ops/s, ±0.25%      | 97.05% slower

  napi-rs-compact:
    35 179 682 ops/s, ±0.58%      | 97.04% slower

  neon:
    8 999 150 ops/s, ±5.50%       | 99.24% slower

  node-bindgen:
    5 395 082 ops/s, ±6.13%       | slowest, 99.55% slower

Finished 5 cases!
  Fastest: JavaScript
  Slowest: node-bindgen
```

```
Running "Hello Plus World" suite...
Progress: 100%

  JavaScript:
    1 198 402 170 ops/s, ±0.29%   | fastest

  napi-rs:
    4 697 036 ops/s, ±0.70%       | 99.61% slower

  napi-rs-compact:
    4 648 021 ops/s, ±0.42%       | 99.61% slower

  neon:
    3 552 365 ops/s, ±6.76%       | 99.7% slower

  node-bindgen:
    2 916 320 ops/s, ±0.44%       | slowest, 99.76% slower

Finished 5 cases!
  Fastest: JavaScript
  Slowest: node-bindgen
```

NAPI-RS

https://github.com/napi-rs/node-rs/tree/main/packages/crc32

使用场景

@node-rs/crc32

Performance

```
@node-rs/crc32 for inputs 1024B x 5,108,123 ops/sec ±1.86% (89 runs sampled)
@node-rs/crc32 for inputs 16931844B, avg 2066B x 271 ops/sec ±1.15% (85 runs sampled)
sse4_crc32c_hw for inputs 1024B x 3,543,443 ops/sec ±1.39% (93 runs sampled)
sse4_crc32c_hw for inputs 16931844B, avg 2066B x 209 ops/sec ±0.78% (76 runs sampled)
sse4_crc32c_sw for inputs 1024B x 1,460,284 ops/sec ±2.35% (90 runs sampled)
sse4_crc32c_sw for inputs 16931844B, avg 2066B x 93.50 ops/sec ±2.43% (69 runs sampled)
js_crc32c for inputs 1024B x 464,681 ops/sec ±0.46% (91 runs sampled)
js_crc32c for inputs 16931844B, avg 2066B x 28.25 ops/sec ±1.64% (51 runs sampled)
js_crc32 for inputs 1024B x 442,272 ops/sec ±2.66% (93 runs sampled)
js_crc32 for inputs 16931844B, avg 2066B x 22.12 ops/sec ±5.20% (40 runs sampled)
+--------------------+------------------+----------------------+
|                    | 1024B            | 16931844B, avg 2066B |
+--------------------+------------------+----------------------+
| @node-rs/crc32     | 5,108,123 ops/sec | 271 ops/sec          |
+--------------------+------------------+----------------------+
| sse4_crc32c_hw     | 3,543,443 ops/sec | 209 ops/sec          |
+--------------------+------------------+----------------------+
| sse4_crc32c_sw     | 1,460,284 ops/sec | 93.50 ops/sec        |
+--------------------+------------------+----------------------+
| js_crc32c          | 464,681 ops/sec   | 28.25 ops/sec        |
+--------------------+------------------+----------------------+
| js_crc32           | 442,272 ops/sec   | 22.12 ops/sec        |
+--------------------+------------------+----------------------+
```

NAPI-RS

使用场景

各种类型 Hash 计算

```rust
// Node.js trace hash function
// From ByteDance Node.js infra
// 20x faster than BigInt calculate logic in Pure JS

#[napi]
fn log_id_hash(input: String) -> u64 {
  let hash = fnv(input.as_bytes(), FNV1_64_INIT);
  linux_core_hash_u64(hash)
}
```

NAPI-RS

https://github.com/microsoft/node-pty/issues/507

使用场景

node-pty

- 封装系统调用
- 跨平台
- 预编译

NAPI-RS

https://github.com/Brooooooklyn/canvas

使用场景

@napi-rs/canvas

- 将 C++ 编写的 Skia 项目编译成 static link library，通过 Rust FFI 机制引入使用。
- 组合现有的 Rust crates 比如 cssparser, base64, avif 等。
- 用 NAPI-RS 封装成 npm 包，对比 node-canvas 0 系统依赖，支持更多系统与 CPU 架构，功能更多。

NAPI-RS

https://github.com/Brooooooklyn/canvas

使用场景

@napi-rs/canvas

```
const { writeFileSync } = require('fs')
const { join } = require('path')

const { createCanvas, GlobalFonts } = require('@napi-rs/canvas')

GlobalFonts.registerFromPath(join(__dirname, 'fonts',
'AppleColorEmoji@2x.ttf'), 'Apple Emoji')
GlobalFonts.registerFromPath(join(__dirname, 'fonts', 'COLRv1.ttf'),
'COLRv1')

const canvas = createCanvas(760, 360)
const ctx = canvas.getContext('2d')

ctx.font = '50px Apple Emoji'
ctx.strokeText('😀😃😄😁😆😅😂🤣☺️😊😊😇', 50, 150)

ctx.font = '100px COLRv1'
ctx.fillText('abc', 50, 300)

const b = canvas.toBuffer('image/png')

writeFileSync(join(__dirname, 'draw-emoji.png'), b)
```

NAPI-RS

https://napi.rs/neon

与 Neon/node-bindgen 对比

- NAPI-RS 有更好的性能/更小的调用开销。
- NAPI-RS 提供从开发到发布的全流程解决方案，而不止是仅仅在 Rust 层提供 Node-API。
- NAPI-RS 对比 Neon 提供的 Node-API 更全面，包含了所有 Node-API 的 C API 封装。
- NAPI-RS 对比 node-bindgen 提供可选的 napi 版本选择，node-bindgen 只能支持最高版本的 Node.js，而 NAPI-RS 可以通过 features 来选择支持的 Node.js 版本，最低支持到 10.0.0。
- NAPI-RS 与 node-bindgen 的异步运行时不同，NAPI-RS 支持的是 Tokio，而 node-bindgen 支持的是 fluvio

04

Rust 与 Node 的未来展望

Rust 与 Node 的未来展望

- Rust 是 JavaScript 基建的未来，Next.js Parcel Prisma Rome 仅仅是革命的开始。
- 随着生态的丰富，未来用 Rust 来编写 Native Addon 会更加便捷，各种配套的基础建设会逐步丰富。
- 除了通过 Node-API ，未来 Rust 会有更多形式为 JavaScript 生态提供服务。比如 Rome 打算嵌入 JavaScript 引擎并提供简单的 JavaScript API ，而不是依托于 Node-API 。比如 Deno 正在实现 –compact 模式 https://github.com/denoland/deno/issues/12577 。届时通过 Deno + Deno FFI + npm 包也能启动存量的 Node.js 项目。(NAPI-RS 已经计划通过 feature flag 支持无缝编译到 Deno FFI)。NAPI-RS 还计划提供无缝编译到 wasm，让包的发布和维护更简单。
- 越来越多的前端与 Node.js 业务相关的代码会出现在 Rust 的 crates 中，各种自研渲染引擎比如小程序和类 Flutter 方案更容易使用 Rust 开发。以后可能会诞生基于 Rust 生态的集运行时、构建、开发调试工具等周边的一体化解决方案 。

Thanks