ChatGPT

เครื่องมือ Build/Bundle สำหรับ Node.js และ Bun (เปรียบเทียบ)

ในการพัฒนาโปรเจค Node.js หรือ Bun ที่ต้องการ **build โค้ดและ bundle dependencies เป็นไฟล์เดียว** รวมถึง **สร้างไฟล์ปฏิบัติการเดียว (single executable binary)** เพื่อสะดวกต่อการแจกจ่าย (เช่น ทำ CLI, API server หรือใช้ใน Docker image) มีเครื่องมือหลายตัวที่ช่วยลดความยุ่งยากในการตั้งค่า **ESM/CommonJS** และการจัดการ dependency แบบ manual โดยแต่ละเครื่องมือมีจุดเด่นและข้อจำกัดต่างกันไป ด้านล่างคือการเปรียบเทียบเครื่องมือที่น่าสนใจ:

Bun (Runtime + Bundler ในตัว)

- **รันไทม์**: ใช้รันไทม์ Bun (รวม Bun runtime เข้าไปในไฟล์ที่ build) – สามารถรันโค้ดที่เขียนสำหรับ Node.js ได้จำนวนมาก เนื่องจาก Bun ถูกออกแบบให้เข้ากันได้กับ Node.js APIs สูง
- **คุณสมบัติเด่น**: Bun เป็นทั้ง JavaScript runtime และ bundler ในตัว ทำให้สามารถ bundle โค้ด TypeScript/JavaScript พร้อม dependency ทั้งหมดได้โดยตรงด้วยคำสั่ง `bun build` [1] นอกจากนี้ยังมี flag `--compile` สำหรับสร้างไฟล์ binary แบบ standalone ที่รันได้เลย [1]
- **ข้อดี**: ประสิทธิภาพสูงและรวดเร็ว (core สร้างด้วย Zig); **All-in-one** (ไม่ต้องใช้เครื่องมือ build ภายนอก); รองรับ **TypeScript** และโมดูล ESM/CJS โดยอัตโนมัติ; สร้างไฟล์ binary แบบพกพาได้บนหลายแพลตฟอร์ม (รองรับ macOS, Linux และ **Windows** ตั้งแต่ Bun v1.1 เป็นต้นไป [2]); รองรับการฝัง asset ไฟล์หรือ native addon (.node) ลงใน binary ด้วย (มีตัวเลือก embed files/N-API addons ในตัว); เหมาะกับทั้งแอปแบบ CLI และเซิร์ฟเวอร์ (ตัวอย่างมีการใช้ bun สร้าง web server ทั้ง backend (Elysia) และ frontend React รวมเป็น binary เดียวได้สำเร็จ [3])
- **ข้อเสีย**: Bun ยังใหม่ (v1.0 ออกปี 2023) อาจมีบักหรือฟีเจอร์ Node.js บางส่วนที่ยังไม่สมบูรณ์; การรองรับแพลตฟอร์มบางอย่างเพิ่งเริ่ม (เช่น Windows เพิ่งเสถียรใน v1.1 แต่ยังมีข้อแตกต่างเล็กน้อยเมื่อเทียบกับ macOS/Linux เช่นชุดทดสอบผ่าน ~98% [4]); ขนาดไฟล์ binary ที่ได้ค่อนข้างใหญ่ (เนื่องจากบรรจุ Bun runtime ทั้งหมด ~ไม่กี่สิบ MB คล้ายกับการพ่วง Node.js runtime ในกรณีเครื่องมืออื่น); หากโปรเจคใช้งานโมดูลพื้นฐานบางตัวที่ Bun ยังไม่รองรับ 100% (เช่น Node API เฉพาะทางมากๆ) อาจต้องทดสอบความเข้ากันได้
- **การรองรับ ESM/CommonJS**: Bun รองรับทั้ง **ESM และ CommonJS** ในตัว (Bun สามารถ import/require โมดูลได้ทั้งสองแบบ) และ bundler จะจัดการรวมโค้ดทุกไฟล์ให้เอง **โดยไม่ต้อง config การแปลงโมดูล** ผู้พัฒนา Bun ระบุว่า Bun 1.1 มีความเข้ากันกับแพ็กเกจของ Node.js ดีขึ้นมาก (สามารถใช้งานแพ็กเกจยอดนิยมอย่าง Playwright, Puppeteer, TensorFlow.js, JSONWebToken, bcrypt ได้แล้ว) [5]
- **การสร้าง Binary**: ใช้คำสั่ง `bun build --compile` เพื่อสร้างไฟล์ปฏิบัติการเดียวที่มีโค้ดและ dependency ทั้งหมด รวมถึงมี Bun runtime ฝังอยู่ภายใน ทำให้ไฟล์นี้เรียกใช้งานได้ทันที (เช่น `./myapp`) โดยไม่ต้องติดตั้ง Node.js/Bun เพิ่มเติม [1] [6] หมายเหตุ:

  "Bun's bundler implements a `--compile` flag... This bundles all imported files and packages into the executable, along with a copy of the Bun runtime. All built-in Bun and Node.js APIs are supported." [1] [6]

- **ความเหมาะสมกับ CLI/API Server**: ดีมาก – สามารถสร้าง CLI ขนาดเล็กแจกจ่ายเป็นไฟล์เดียว, หรือสร้าง backend server binary สำหรับ deploy ขึ้นเครื่องอื่นหรือใส่ Docker image แบบ minimal ได้ (กรณี Docker สามารถใช้ base image เล็กๆ แล้ว COPY ไฟล์ binary ที่สร้างไว้ไปรันได้เลย ลดขั้นตอน install Node); Bun มี performance สูงในการ

จัดการ I/O และ concurrency ด้วย (ใช้ JavaScriptCore engine) จึงเหมาะทั้ง CLI ที่ตอบสนองเร็ว และ server ที่ต้องการ throughput สูง

- **ความง่ายในการตั้งค่า**: **ง่ายมาก** – กรณีโปรเจค JS/TS ทั่วไป แทบไม่ต้องเขียน config ใดๆ; มีค่าเริ่มต้นฉลาด (convention-over-configuration) เช่น bundler ของ Bun จะ bundle dependencies ให้โดยอัตโนมัติ (ค่า default คือ `--packages=bundle` เพื่อ bundle dependencies ทั้งหมด ยกเว้นที่ระบุ external) และ resolve ปัญหา ESM/CJS ให้อัตโนมัติ; การสร้าง binary ก็แค่เพิ่ม `--compile` ตอน build; ทั้งนี้สามารถปรับ option เพิ่มเติมได้ตามต้องการผ่าน CLI flags หรือ API (เช่น define, minify, ฯลฯ)

Vercel pkg (เครื่องมือแพ็ค Node.js เป็น Executable)

- **รันไทม์**: ใช้ Node.js (ฝัง Node.js runtime ภายใน executable ที่สร้าง) – รองรับ Node เวอร์ชันที่กำหนดไว้ (เดิมสูงสุดถึง Node ~18.x)
- **คุณสมบัติเด่น**: `pkg` เป็น CLI ที่ช่วยแพ็คโปรเจค Node.js ให้เป็นไฟล์ .exe/.bin เดียวที่รันได้ในเครื่องที่ไม่มี Node.js (มันจะแนบ Node.js runtime + โค้ดของเราไว้ด้วยกัน) [7] ผู้ใช้สามารถกำหนดเป้าหมายแพลตฟอร์มหลายอันให้ pkg สร้าง executable ข้ามแพลตฟอร์มได้ (Windows, Linux, macOS) ในคำสั่งเดียว
- **ข้อดี**: เคยเป็นเครื่องมือยอดนิยม มี community ใช้งานกว้างขวาง; ใช้งานง่าย – **แทบไม่ต้องตั้งค่า** (สามารถรัน `pkg .` เพื่อแพ็คโปรเจคได้ทันทีถ้าโครงสร้างมาตรฐาน); รองรับการทำ **cross-platform build** (เช่น build บนแมคแต่สร้างไฟล์ .exe สำหรับ Windows ด้วย) และมีไฟล์ Node runtime ที่ build ล่วงหน้าสำหรับหลายแพลตฟอร์มให้ดาวน์โหลดใช้; ซ่อนซอร์สโค้ดภายใน executable (ช่วยแจกจ่ายเชิงพาณิชย์ได้โดยไม่ต้องเปิดเผยซอร์ส); **รวม dependencies ทั้งหมดให้อัตโนมัติ** – เมื่อตอน build มันจะวิเคราะห์โค้ด `require()` / `import` ทั้งหมด แล้ว bundle เข้าไปใน binary (สามารถรวม asset เพิ่มเติมได้โดยระบุ config); เหมาะทั้งกับ CLI tools และการแพ็คแอปเซิร์ฟเวอร์ Node ขนาดเล็ก (ช่วยลดขั้นตอนการ deploy ที่ต้องติดตั้ง Node + npm modules)
- **ข้อเสีย**: ปัจจุบัน *pkg* หยุดพัฒนาแล้ว *(deprecated)* โดยมีเวอร์ชันสุดท้าย *5.8.1 (*ประกาศเลิกซัพพอร์ตในต้นปี *2024)* [8] เนื่องจาก *Node.js 21+* จะมีฟีเจอร์ *native* สำหรับสร้าง *single executable* มาแทน [8] ดังนั้น pkg อาจไม่รองรับ Node เวอร์ชันใหม่ๆ (ล่าสุดรองรับถึง Node 18.x ตาม base binary ที่มี); **ไม่รองรับ ESM โดยตรง** – โปรเจคที่เป็นโมดูล ESM จะไม่สามารถแพ็คได้ทันที ต้อง transpile กลับเป็น CommonJS ก่อน (เช่นใช้ Babel แปลง) แล้วจึงใช้ pkg อีกที [9] ; มีข้อจำกัดกับโค้ดที่ใช้ dynamic require (ต้องใช้วิธีพิเศษเช่น config ให้ pkg รวมไฟล์ที่เกี่ยวข้อง ไม่งั้นตอนรันจะหา module ไม่เจอ); นอกจากนี้ขนาดไฟล์ output ค่อนข้างใหญ่ (เช่น ~30MB+ เพราะรวม Node runtime) และประสิทธิภาพการรันเหมือน Node ปกติ (อาจมี overhead เล็กน้อยตอน initial load เพราะต้องคลายไฟล์ที่ bundle ภายใน)
- **การรองรับ ESM/CommonJS**: **CommonJS** – pkg ถูกออกแบบมาสำหรับโค้ด CommonJS (สมัย Node ยังไม่รองรับ ESM เต็มที่) ดังนั้นหากโค้ดหรือ dependency เป็น ESM ต้องทำให้เป็น CJS ก่อน (มีผู้ใช้รายงานใน StackOverflowว่าต้อง “แปลง ESM เป็น CommonJS ก่อนแล้วค่อยใช้ pkg สร้าง exe” [9]); โดยค่า default pkg จะดึงโค้ดจาก entry ที่ระบุ (มักคือไฟล์ `index.js` หรือ script ตาม `package.json`) แล้วตาม require tree ไปรวมไฟล์ .js ทั้งหมดและ `node_modules` ที่จำเป็นเข้ามา
- **การสร้าง Binary**: ใช้ base image ของ Node ที่ pre-packaged – pkg จะนำไฟล์ Node.js runtime (ที่ปรับแก้ให้โหลดไฟล์จาก bundle ในตัว) มารวมกับโค้ดผู้ใช้เป็นไฟล์เดียว การสร้างสามารถทำ multiplatform โดยใช้ flag `-t` (target) เช่น `-t node18-linux-x64,node18-win-x64` เพื่อสร้างสองไฟล์ exe สำหรับ Linux และ Windows ในรอบเดียว; ไฟล์ binary ที่ได้สามารถรันตรงๆ โดยไม่มี dependency ภายนอก (concept เดียวกับ static linking); อย่างไรก็ตาม เนื่องจาก *pkg* หยุดพัฒนา ทาง *Vercel* ชี้แนะให้พิจารณาใช้ฟีเจอร์ *Single Executable Application* ของ *Node.js core* แทนในอนาคต [8]
- **ความเหมาะสมกับ CLI/API server**: เคยเป็นตัวเลือกยอดนิยมสำหรับ CLI Node.js (เช่นเครื่องมือ developer หลายตัวที่แจก .exe/.bin ใช้ pkg) และสำหรับแพ็คแอป Node server ที่ต้องส่งให้ลูกค้ารันโดยไม่ยุ่งกับการติดตั้ง Node; ใน Docker แม้ pkg จะสร้างไฟล์ exe ใช้เองได้ แต่กรณีใช้งานใน container ที่เราควบคุมเอง บางครั้งการใช้ Node alpine

base + โค้ด bundle ธรรมดาอาจเพียงพอ; หากต้องการลดขนาด image จนเหลือ scratch image การมี binary เดียวจาก pkg ก็ตอบโจทย์ได้

- **ความง่ายในการตั้งค่า**: **ง่าย** – เริ่มต้นแทบไม่ต้อง config (ส่วนมากปรับแค่ระบุไฟล์ `assets` หรือ `scripts` ใน `package.json` ให้ pkg รู้ว่าต้องรวมไฟล์ใดบ้างกรณีมีการโหลดไฟล์แบบ dynamic); ติดตั้ง pkg เป็น global แล้วรันคำสั่งได้เลย; เอกสารเข้าใจง่าย; ปัญหาคือเนื่องจากไม่ได้อัปเดตให้รองรับ *ESM/Node* เวอร์ชันใหม่ การใช้งานในปี *2024+* อาจต้องใช้ *workaround* เยอะ หรือหันไปใช้เครื่องมืออื่นแทน

Nexe

- **รันไทม์**: ใช้ Node.js (เลือกเวอร์ชัน Node ได้; มีทั้งแบบดาวน์โหลด binary ที่ build ไว้ หรือ build Node จากซอร์สใหม่)
- **คุณสมบัติเด่น**: Nexe เป็น CLI สำหรับ “คอมไพล์” แอป Node.js ให้เป็นไฟล์ปฏิบัติการเดียวคล้าย pkg [10] มีจุดมุ่งหมายเพื่อให้แอปเป็น self-contained (พก Node runtime ไปกับตัว) แจกจ่ายได้สะดวกขึ้น โดยรองรับ **ข้ามแพลตฟอร์ม** และสามารถปรับ pipeline การ build ได้ยืดหยุ่น (เช่นต่อพ่วงกับ bundler อื่น)
- **ข้อดี**: ยังมีการพัฒนา/ดูแล (active กว่า pkg); รองรับ Node.js หลายเวอร์ชัน – ผู้ใช้สามารถระบุ target Node version ที่ต้องการ (หากไม่มี binary สำเร็จรูป Nexe จะมีโหมด `--build` เพื่อคอมไพล์ Node ด้วยเครื่องผู้ใช้); **Cross-platform** – สร้าง exe สำหรับ Windows, macOS, Linux ได้; รวมไฟล์สคริปต์ JS และ dependencies ทั้งหมดเข้าไปใน binary (ใช้แนวทาง bundler ภายใน หรือผู้ใช้ pipe output bundler เข้าก็ได้); สามารถฝังไฟล์ static assets เพิ่มเติมได้ง่าย (flag `-r "glob/pattern"` เพื่อรวมไฟล์เข้าไป และอ่านผ่าน `fs` API ใน runtime) [11] ; ไม่มี dependency ภายนอกเมื่อรัน (ไม่ต้องติดตั้ง Node); ใช้งานได้ทั้งกับแอป CLI และ server (เคสใช้งานคล้าย pkg)
- **ข้อเสีย**: **การรองรับ ESM ยังไม่สมบูรณ์นัก** – หากโค้ดเป็นโมดูล ESM อาจต้อง transpile หรือ bundle ล่วงหน้า (เช่นใช้ Webpack/Rollup/Esbuild ก่อน แล้ว pipe เข้าคำสั่ง `nexe`) เพื่อหลีกเลี่ยงปัญหาการ require โมดูล ESM (มีรายงานปัญหาเมื่อรัน exe แล้ว Node ภายในยังตรวจพบ `package.json` นอกตัวมัน ทำให้ Error ต้องใช้วิธีแก้ เช่นย้ายโฟลเดอร์ หรือ patch พฤติกรรม Node) [12] [13] ; ขนาดไฟล์ output ใกล้เคียง pkg (เพราะคือ Node + script bundle); **ต้องการเวลาคอมไพล์นาน** กรณีที่ต้อง build Node จากซอร์ส (เช่นถ้าเลือก Node เวอร์ชันที่ไม่มีในคลัง pre-built); การ cross-compile อาจยุ่งยากหากข้าม OS (เช่นจาก Windows ไป Linux) – โดยทั่วไปต้องรันคำสั่งบน OS เดียวกับ target; Community รอบ Nexe เล็กกว่า pkg ในอดีต (documentation อาจไม่ครอบคลุมทุกเคสเท่า pkg)
- **การรองรับ ESM/CommonJS**: **CommonJS (เป็นหลัก)** – Nexe จะ bundle/execute โค้ดในรูปแบบที่ Node runtime ภายในรองรับ ซึ่งปัจจุบัน Node runtime นั้นรองรับ ESM อยู่แล้วหากมีไฟล์ `.mjs` หรือ `package.json` type:module; แต่ในทางปฏิบัติการรวมไฟล์แบบ ESM ที่กระจัดกระจายอาจยุ่งยาก Nexe จึงมักใช้งานร่วมกับ bundler (เช่น Webpack) ให้แปลงทุกอย่างเป็นไฟล์เดียว (UMD หรือ CJS) ก่อน; ถ้าเป็น **CommonJS** ล้วน (เช่นโปรเจค Node แบบเก่า) Nexe สามารถอ่านไฟล์ entry แล้ว include dependency ทั้งหมดให้อัตโนมัติค่อนข้างดี; โดยรวม ESM pure ยังถือว่าต้องอาศัยเครื่องมือเสริมหรือ workaround
- **การสร้าง Binary**: Nexe มีคลัง Node binary สำเร็จรูป (รองรับหลายเวอร์ชัน/แพลตฟอร์ม) ให้ดาวน์โหลด แต่ถ้าตรงกับที่ต้องการไม่ได้ ก็สามารถสั่ง `--build` เพื่อคอมไพล์ Node ด้วยเครื่องเรา (ต้องติดตั้งเครื่องมือ compile C++ เช่น Visual Studio Build Tools บน Win หรือ Xcode บน mac ฯลฯ); เมื่อ build เสร็จจะได้ไฟล์ exe ที่รวม **โค้ด JS (bundled)** เข้าในส่วน resource ของไฟล์ พร้อม stub สำหรับโหลดและรันโค้ดนั้นบน Node runtime ที่รวมมา; ผู้ใช้สามารถเพิ่ม resource อื่นๆ (ไฟล์ statics) ลงไปใน binary ได้ด้วยคำสั่ง `-r`; ตัวอย่าง*:* `nexe app.js -r 'public/**/*' -o myapp.exe` จะได้ *myapp.exe* ที่รันได้ทันทีและมีไฟล์ใน *public/* ฝังอยู่ภายใน อ่านผ่าน `fs` ได้ [11]
- **ความเหมาะสมกับ CLI/API server**: เหมาะสม – สามารถใช้แทน pkg ในการแจกจ่าย CLI (มีข้อดีที่รวม asset เพิ่มเข้าไปได้ง่ายกว่าด้วย `-r` flag) หรือจะแพ็คเป็น server executable ก็ได้; อย่างไรก็ตาม สำหรับแอป Node ขนาดใหญ่ที่มี native dependency (เช่น bcrypt, canvas) อาจต้องทดสอบว่าทำงานใน Nexe ได้ (Nexe รองรับการ include .node ไฟล์แบบ Virtual FS คล้าย pkg); ใน Docker หากต้องการ image เล็กแบบ scratch ก็ใช้แนวคิดเดียวกับ pkg คือ build binary แล้วก็อปไป; Nexe อาจเหมาะกับผู้ที่ต้องรองรับ Node เวอร์ชัน custom หรือแอปที่ pkg ใช้ไม่ได้ (เนื่องจาก pkg เลิกซัพพอร์ต)

- **ความง่ายในการตั้งค่า**: **ง่ายปานกลาง** – การใช้งานพื้นฐาน (`nexe app.js`) ตรงไปตรงมาเหมือน pkg แต่เพื่อผลลัพธ์ที่ดีมักต้องมีการเตรียมโค้ดก่อน (เช่น bundle ให้เป็นไฟล์เดียวกรณี ESM) จึงเพิ่มขั้นตอน; เอกสารของ Nexe มีตัวเลือกเยอะ (target, resource, plugin) ซึ่งอาจต้องเรียนรู้เล็กน้อย แต่โดยรวมไม่ซับซ้อนเกินไป; การแก้ปัญหาเฉพาะ (เช่น dynamic require) ต้องอ่าน FAQ/community issues; ไม่ต้องเขียน config file แยก (ทุกอย่างทำผ่าน CLI ได้เลย)

Astra (Astra Compiler)

- **รันไทม์**: ใช้ Node.js (อิงฟีเจอร์ **Single Executable Applications (SEA)** ของ Node core) – ปัจจุบันรองรับ Node v20/21 (ใช้ฟีเจอร์ SEA ที่ experimental ใน Node)
- **คุณสมบัติเด่น**: Astra เป็นโปรเจคใหม่ที่พัฒนาเพื่อแก้จุดอ่อนของ pkg/nexe โดยเฉพาะเรื่อง **ESM** – ตัวมันจะใช้ **esbuild** ในการ bundle โค้ด JavaScript/TypeScript ทั้งหมดให้เป็นไฟล์เดียว แล้วใช้ความสามารถ SEA ของ Node.js ในการฝังโค้ดนี้ลงใน binary ของ Node อีกที (ผ่านเครื่องมือเสริมที่ชื่อ *postject*) [14] เป้าหมายหลักคือให้ developer ที่เขียนด้วยโมดูล ESM หรือ TypeScript *"compile"* โปรเจคออกมาเป็นไฟล์ .exe/.bin เดียวได้ง่ายๆ (DX ที่ดี)
- **ข้อดี**: **รองรับ ESM และ TypeScript อย่างสมบูรณ์** – Astra ใช้ esbuild แปลงโค้ดทันทีจึงจัดการเรื่อง import/export ได้หมด (ไม่ต้องแก้เป็น CJS เอง); ใช้ฟีเจอร์ Node SEA ซึ่งเป็นกลไกทางการของ Node (ทำให้ติดตาม Node เวอร์ชันใหม่ๆ ได้ทัน, ไม่ต้องรอแก้แพตช์เองแบบ pkg ต้องทำ) – ผู้สร้างระบุว่าที่ทำ Astra เพราะไม่ชอบปัญหา pkg/nexe ที่ไม่รองรับ ESM ดีพอ [15] ; **ใช้งานง่าย** – มี CLI ที่ออกแบบ Developer Experience ดี (syntax คำสั่งกระชับ, มี UX ที่ดี) ตามที่ผู้พัฒนาแจ้ง; เช่นคำสั่งอาจเป็น `astra compile` คล้าย *pkg (*รายละเอียดขึ้นกับ *implementation* แต่เป้าคือให้ใช้ง่าย*)*; เหมาะสำหรับสร้าง CLI tools หรือ server binaries เช่นเดียวกับ pkg/nexe (โฟกัสหลักคือ CLI/Servers ไม่รวม Electron เป็นต้น [16])
- **ข้อเสีย**: เป็นโครงการใหม่ **ยังไม่สมบูรณ์** – ปัจจุบัน (ข้อมูลปลายปี 2024) Astra **รองรับเฉพาะ Windows** ในการ output (อยู่ระหว่างพัฒนาการรองรับ Linux/Mac) [17] ; ยังมีปัญหากับ dependency ที่เป็น native binaries (เช่น bcrypt) ซึ่งยัง compile รวมไม่สำเร็จในตอนนี้ [17] ; เอกสารยังน้อย (ตามความเห็นผู้ใช้ใน HN); โครงการอยู่ในสถานะ early development อาจเจอบักหรือฟีเจอร์ที่ยังไม่รองรับ; ชุมชนผู้ใช้ยังเล็ก
- **การรองรับ ESM/CommonJS**: **ESM first** – Astra โดดเด่นที่รองรับ ESM/TypeScript ได้ดี (bundler ภายในจัดการ import/export แทบทุกอย่าง) ในขณะที่ก็สามารถ bundle โค้ด CommonJS ได้ด้วยเช่นกัน (esbuild รองรับ CJS module); การรันไฟล์ใน runtime Node ที่ embed มาก็สามารถใช้ require (CJS) ได้ตามปกติเพราะ Node runtime สนับสนุนทั้งสองแบบ; แต่เนื่องจาก Node SEA ณ ปัจจุบันยังจำกัดที่การรันสคริปต์แบบ CJS เดียว (Astra จึงน่าจะ bundle ทั้งหมดออกมาเป็นไฟล์ CJS เดียวแล้ว embed) – ผู้ใช้แทบไม่ต้องสนใจส่วนนี้ เพราะ Astra จัดการให้เบื้องหลัง
- **การสร้าง Binary**: ใช้ Node SEA + Node binary – เบื้องหลัง Astra จะเรียก Node runtime ให้สร้าง snapshot/blob ของโค้ด (SEA blob) แล้วใช้เครื่องมือเช่น postject ฉีด blob นี้เข้าไปในตัว binary ของ Node (ทำให้ Node binary ที่ได้ พอรันจะตรวจพบ blob และ execute โค้ดของเราทันที) [18] [19] ; การ build ตอนนี้จำกัด OS (Win) แต่ผู้พัฒนากำลังพัฒนาให้ cross-compile ได้; ขนาดไฟล์ที่ได้จะประมาณ Node binary ปกติ (~30-40MB) บวกโค้ดเรา (ซึ่งมักเล็กเมื่อ bundle แล้ว)
- **ความเหมาะสมกับ CLI/API server**: หากเน้น **ESM/TS** เป็นหลัก Astra น่าสนใจมาก เพราะลดความยุ่งยากเรื่อง module format; สำหรับ CLI ที่กลุ่มเป้าหมายใช้ Windows เป็นหลัก Astra ตอบโจทย์ (สร้าง .exe ได้) ส่วน Linux/Mac อาจต้องรอการรองรับ; สำหรับ server-side กรณีต้องการใช้ Node เวอร์ชันใหม่ล่าสุด (เช่น 20+ ที่ pkg ยังไม่รองรับ) Astra ที่พึ่ง Node core ก็นับว่าแนวทางทันสมัย – อย่างไรก็ดี ณ ตอนนี้ Astra อาจยังไม่พร้อมใช้ใน production ที่กว้างขวางเพราะข้อจำกัด platform
- **ความง่ายในการตั้งค่า**: **ง่าย** – เป้าหมายโครงการคือลด config ลงให้น้อยที่สุด (คล้ายปรัชญา pkg ที่ใช้ง่าย) – ผู้ใช้ส่วนใหญ่แค่มี Node v20+ และติดตั้ง Astra ก็ compile ได้เลย (รายละเอียดการใช้งานจริงอาจดูจาก README ของโครงการ); อย่างไรก็ตามเพราะ Astra ยังใหม่ การ debug หรือ config เพิ่มเติมในเคสซับซ้อนอาจต้องดูโค้ดหรือตั้ง issue ถามผู้พัฒนาโดยตรง

Node-compiler (nodec)

- **รันไทม์**: ใช้ Node.js (embedding Node runtime + JS bundle) – default มากับ Node v20.12.0 แต่ผู้ใช้สามารถเลือกเวอร์ชัน Node ได้ตามต้องการ
- **คุณสมบัติเด่น**: Node-compiler (หรือ `nodec`) เป็นเครื่องมือ CLI แบบไม่เป็นทางการที่ทำงานคล้าย pkg/nexe แต่ใช้แนวทางสมัยใหม่ โดย **ใช้ ESBuild** ในการ bundle โค้ดของโปรเจค และใช้ภาษา **Go** ในการคอมไพล์รวม Node runtime กับโค้ดที่ bundle แล้วออกมาเป็นไฟล์ executable เดียว (Cross-compile ข้าม OS ได้) [20] [21] โปรเจคนี้เริ่มโดย community หลังจาก pkg ไม่ตอบโจทย์ Node เวอร์ชันใหม่ๆ โดยเน้นให้รองรับ ESM และ Node รุ่นใหม่
- **ข้อดี**: **รองรับ ESM/CJS และ TypeScript** – เพราะใช้ esbuild เหมือน Astra, การ bundle โค้ดโมดูลสมัยใหม่ทำได้เนียน; **Cross-Platform Build** – สามารถสร้างหลาย target OS/ARCH ในคำสั่งเดียว (เช่น `--target linux-x64 --target win-x64` เป็นต้น จะได้ binary หลายไฟล์) [22] ; เลือกเวอร์ชัน Node ที่จะฝังได้ (option `--nodeVersion` เพื่อระบุ Node เวอร์ชันที่ใช้) [23] ทำให้ยืดหยุ่นเรื่อง compatibility; ผลงานที่ได้คือไฟล์ binary ที่รันได้เลยเหมือน pkg/nexe; เหมาะกับแอป CLI หรือ server เช่นกัน; การจัดการ dependency เป็นแบบ bundle ทั้งหมด (ไม่ต้องห่วงเรื่อง externals); ช่วยล็อก Node runtime เวอร์ชันที่แอปใช้ไปในตัว (ไม่ต้องสนว่าเครื่องผู้ใช้มี Node เวอร์ชันอะไร)
- **ข้อเสีย**: โครงการนี้ยังอยู่ในช่วง **ทดลอง (experimental)** มีคำเตือนในเอกสารว่าอาจยังมีบั๊ก [20] ; **ต้องมี Go compiler** ในระบบ (เพิ่ม prerequisite เล็กน้อยนอกเหนือจาก Node); ชุมชนผู้ใช้งานยังไม่มากและรุ่นปัจจุบันเป็น 0.x (แสดงว่ายังไม่ stable เต็มที่); เคยมีโครงการลักษณะเดียวกัน (เช่น node-packer) ในอดีตที่ concept ดีแต่ไม่ได้ไปต่อยาวนัก ผู้ใช้จึงควรประเมินความเสี่ยงในการใช้ระยะยาว; การรวม native modules (.node ไฟล์) – โดยหลักการ Node SEA สามารถฝังไฟล์เพิ่มเติมได้ แต่ nodec อาจยังไม่มี API ง่ายๆ สำหรับกรณีนี้ (ต้องตรวจสอบเพิ่มเติม); นอกจากนี้ output binary ขนาดใหญ่ตาม Node runtime (เช่น ~30MB+)
- **การรองรับ ESM/CommonJS**: **รองรับทั้งสองแบบ** – Nodec สามารถ compile โค้ด **รูปแบบ ESM หรือ CJS ก็ได้** (มี option `--format` ให้เลือก output จะเอาเป็น cjs หรือ esm ก่อนแพ็คได้ด้วย [24] แต่ดีฟอลต์จะพยายามใช้ ESM ตาม target Node); ไม่ว่าต้นทางจะเขียนด้วยโมดูลระบบใด esbuild จะจัดการ bundle ให้เป็นไฟล์เดียว พร้อมแก้ import ให้; ตัว Node runtime ภายในรองรับ CJS อยู่แล้ว และ ESM ก็รองรับหากเรียกผ่าน loader ที่ esbuild เตรียมไว้; เรียกได้ว่า developer ไม่ต้องกังวลเรื่อง ESM/CJS เมื่อใช้ nodec – เขียนตามปกติได้เลย
- **การสร้าง Binary**: เบื้องหลัง nodec จะดาวน์โหลด Node.js runtime เวอร์ชันที่ระบุ (หรือเวอร์ชัน default ถ้าไม่ระบุ) มา จากนั้นใช้ esbuild bundle โค้ด JS/TS ทั้งหมดเป็นไฟล์ JS ก้อนเดียวที่เหมาะกับ Node เวอร์ชันนั้น (รวม polyfill ตาม target) [25] แล้วใช้ Go สร้าง binary โดยฝัง Node + ไฟล์ JS bundle เข้าไว้ (concept เดียวกับ Node SEA); สามารถระบุหลายแพลตฟอร์มในคำสั่งเพื่อสร้างหลายไฟล์ต่อรอบได้สะดวก [26] (Go จะ cross-compile ให้อัตโนมัติถ้ามีเวอร์ชัน Node ที่ตรงกัน); เวลา runtime เริ่มทำงาน Node จะโหลดโค้ดจากส่วนที่ฝังแล้วรันทันที (คล้าย PKG/Nexe)
- **ความเหมาะสมกับ CLI/API server**: เป็นตัวเลือกที่ดีอีกตัวสำหรับการสร้าง CLI tools (โดยเฉพาะอย่างยิ่งถ้าต้องรองรับ Node v19/v20 ซึ่ง pkg ไม่ทำแล้ว) และใช้ได้กับ server เช่นกัน; ข้อดีคือ cross-compile หลายแพลตฟอร์มง่าย เหมาะกับการเตรียม release artifact หลาย OS พร้อมกัน; อย่างไรก็ตามเพราะ nodec ยังใหม่ ผู้ใช้ควรทดสอบแอปที่ build ออกมาว่าทำงานถูกต้องครบถ้วน; ใน Docker environment ก็ใช้หลักการเดียวกันคือเอา binary ไป run บน base image เล็กๆ ได้
- **ความง่ายในการตั้งค่า**: **ค่อนข้างง่าย** – มี CLI ตรงไปตรงมา (`npx nodec --entry index.ts --name myapp` แล้วได้ไฟล์ `myapp` ในไม่กี่ขั้นตอน [21]); ไม่ต้องสร้าง config file แยก (option ผ่าน CLI ทั้งหมด); แต่ต้องติดตั้ง Go ก่อนหนึ่งครั้ง; โดยรวม developer ที่คุ้นกับการใช้ CLI bundle อย่าง tsup หรือ pkg จะเรียนรู้ nodec ได้เร็ว นอกจากนี้ community มีตัวอย่างใน README ให้ดู; ความยากอาจอยู่ที่เวลา debug ถ้า binary ใช้งานไม่ได้ (เพราะเครื่องมือนี้ยังใหม่, ต้องดู error ที่ runtime ซึ่งอาจ debug ยากกว่ารันผ่าน Node ตรงๆ)

ฟีเจอร์ Single Executable ของ Node.js (SEA)

- **รันไทม์**: Node.js (ฟีเจอร์ builtin ใน Node core ตั้งแต่ v18.16.0 เป็นต้นมา – ยัง experimental)

- **คุณสมบัติเด่น**: Node.js เพิ่มความสามารถ **Single Executable Applications (SEA)** เพื่อให้แจกจ่ายแอป Node ได้สะดวกโดยไม่ต้องติดตั้ง Node ในระบบปลายทาง [27] แนวคิดคือ **"ฝัง"** (inject) โค้ดหรือ asset ที่เตรียมไว้เข้าไปในไฟล์ binary `node` แล้วเมื่อรันไฟล์นั้น Node จะเช็คว่ามี payload ฝังมาหรือไม่ ถ้ามีก็จะรันสคริปต์นั้นทันที [18] [19] (หากไม่มี payload ก็จะทำงานเป็น Node ปกติ)
- **ข้อดี**: เป็นโซลูชันทางการ – ได้รับการดูแลโดยทีม Node.js core เอง (ทำให้ตามเวอร์ชันใหม่ได้รวดเร็วและเชื่อถือได้ในระยะยาว); ไม่ต้องพึ่งเครื่องมือ third-party ที่อาจเลิกพัฒนา; สามารถใช้งานได้ตั้งแต่ Node 18.16+ (ต้องเปิด experimental flag) และจะเสถียรขึ้นใน Node เวอร์ชันอนาคต; **ความเข้ากัน**: เพราะคือ Node แท้ๆ runtime ภายในจึงไม่มีปัญหาความเข้ากันกับแพ็กเกจใดๆ (ตรงข้ามกับ Bun ที่อาจมี compatibility gap เล็กน้อย)
- **ข้อเสีย**: **ยังอยู่ระหว่างพัฒนา** (Stability Index 1 – Active development [28]) – ปัจจุบันรองรับแค่การรันไฟล์สคริปต์หลักไฟล์เดียวแบบ CommonJS เท่านั้น (โมดูลย่อยต้องถูก bundle มาก่อน) [19] ; การ *embed assets* หลายไฟล์หรือ Directory ยังไม่สะดวก (ต้องรวมเป็น blob เอง) และ **ยังไม่รองรับ ESM โดยตรง** ในการฝัง (ต้อง bundle เป็น CJS ก่อน) [19] ; ขั้นตอนการสร้างค่อนข้าง manual: ผู้ใช้ต้องใช้ Node สร้าง blob (ผ่านคำสั่ง `node --experimental-sea-config`) แล้วใช้เครื่องมืออย่าง `postject` ในการฉีด blob เข้ากับไฟล์ Node binary [29] [30] ; เอกสารยังเป็นเชิงเทคนิค (สำหรับนักพัฒนาที่คุ้นเคยระบบ build)
- **การรองรับ ESM/CommonJS**: **CommonJS (ในขั้นปัจจุบัน)** – SEA อนุญาตให้ฝัง *startup snapshot* ของโค้ด JS ซึ่งตอนนี้รองรับเฉพาะการรันโค้ดแบบ CommonJS script เมื่อ Node start (ทีม Node กำลังพัฒนาการรองรับ ESM snapshot ในอนาคต); ซึ่งหมายความว่าหากแอปใช้ ESM ก็จะต้องทำการ bundle/แปลงเป็น CJS หนึ่งไฟล์ก่อนสร้าง SEA blob; อย่างไรก็ดี Node runtime ที่รวมมานั้นสามารถ `import` ESM ได้ถ้าโค้ดของเราทำการ import runtime (แต่การ embed เข้า snapshot ยังทำไม่ได้)
- **การสร้าง Binary**: ผู้ใช้ต้องมี Node เวอร์ชันที่รองรับ (เช่น v22+) และทำตามขั้นตอนที่กำหนดในเอกสาร SEA [29] [30] คือ: (1) เตรียมไฟล์โค้ดหลัก เช่น `index.js` ; (2) สร้างไฟล์ config (JSON) ระบุ main script และชื่อไฟล์ blob output; (3) รันคำสั่ง Node เพื่อสร้าง blob (`node --experimental-sea-config config.json` ได้ไฟล์ `sea-prep.blob`); (4) ทำสำเนาไฟล์ `node` binary และ (บน macOS/Windows ต้องลบ signature ด้วย codesign/signtool เพื่อให้ postject เขียนข้อมูลได้); (5) ใช้เครื่องมือ `postject` ฝัง blob เข้าไปใน binary Node ที่ copy มา; ผลลัพธ์คือไฟล์ Node executable ที่ฝังโค้ดพร้อมรัน; หมายเหตุ: ขั้นตอนเหล่านี้โดยรวมยังไม่เหมาะกับการใช้งานของนักพัฒนาทั่วไป จึงมีเครื่องมืออย่าง *Astra, nodec* มาช่วย *automate*
- **ความเหมาะสมกับ CLI/API server**: ในอนาคตฟีเจอร์นี้จะเหมาะสมอย่างยิ่งเพราะลดการพึ่งพาเครื่องมือนอก; ปัจจุบันในโปรดักชันบางที่ก็เริ่มทดลองใช้ (ด้วยความระมัดระวังและอาจใช้เครื่องมือช่วย) – สำหรับ CLI ที่โค้ดไม่ใหญ่ สามารถ bundle แล้ว SEA ได้เลย; สำหรับ server ถ้าต้องการ slim deployment ก็ทำได้เช่นกัน แต่ยังต้องพิจารณาข้อจำกัดเรื่อง asset และ ESM ในตอนนี้
- **ความง่ายในการตั้งค่า**: **ค่อนข้างซับซ้อน (ณ ตอนนี้)** – เพราะยัง experimental และไม่มี CLI friendly; Developer ทั่วไปมักจะใช้เครื่องมือ wrapper (เช่น Astra, nodec) มากกว่าจะมาทำเองทีละขั้นตอน; แต่เมื่อฟีเจอร์ mature ขึ้น อาจมีการผนวกคำสั่งหรือ API ที่ทำให้สร้าง SEA ได้ง่ายขึ้นในอนาคต; โดยสรุป ถ้าต้องการเสถียรตอนนี้ แนะนำใช้เครื่องมืออย่างข้างต้น (Astra/nodec) ที่ใช้ SEA เบื้องหลังแทน

สรุป: เครื่องมือแต่ละตัวมีจุดเด่นต่างกัน – หากต้องการโซลูชันครบวงจรง่ายๆ สำหรับโปรเจคใหม่ **Bun** เป็นตัวเลือกที่น่าสนใจเพราะ bundler เร็วและ output เป็น binary ได้เลย (เหมาะกับงานที่ยอมใช้ Bun runtime แทน Node) แต่ถ้าต้องการ runtime Node.js แท้ๆ และรองรับ ESM สมัยใหม่ ควรพิจารณา **Astra** หรือ **nodec** ซึ่งใช้ความสามารถ Node SEA รุ่นใหม่ (แม้จะยังใหม่อยู่ก็ตาม) ในขณะที่ **nexe** เป็นตัวเลือกที่ดีสำหรับใครที่ยังต้องการแพ็ค Node app ในยุคที่ pkg หยุดพัฒนา – nexe มีความยืดหยุ่นและยังใช้งานได้แม้กับ Node เวอร์ชันล่าสุด (แต่ต้องจัดการ ESM เอง) ส่วน **pkg** นั้นเหมาะกับโปรเจคเดิมที่ยังติดอยู่กับ Node เวอร์ชันเก่าและ CommonJS (หรือเพื่อเปรียบเทียบให้เห็นข้อดีข้อเสีย) แต่ไม่แนะนำใช้กับงานใหม่เพราะหยุดซัพพอร์ต. โดยภาพรวมแนวโน้มอนาคตจะไปทางการใช้ **Node.js SEA** โดยตรงร่วมกับเครื่องมือช่วย เพื่อให้การสร้าง single executable **ง่าย**, **รองรับเวอร์ชันใหม่**, และ **ไม่ต้อง config ยุ่งยาก** เหมือนในอดีตครับ

อ้างอิง: [1] [6] [2] [5] [3] [8] [9] [12] [13] [10] [11] [14] [17] [20] [21] [18] [19]

[1] [6] Single-file executable – Runtime | Bun Docs
https://bun.sh/docs/bundler/executables

[2] [4] [5] Bun 1.1 released with Windows support, stable WebSocket client and more • DEVCLASS
https://devclass.com/2024/04/02/bun-1-1-released-with-windows-support-stable-websocket-client-and-more/

[3] Bundling your Node.js web app into a single executable using Bun | Hiddentao Labs
https://hiddentao.com/archives/2024/11/16/bundling-your-nodejs-web-app-into-a-single-executable-using-bun/

[7] [8] GitHub - vercel/pkg: Package your Node.js project into an executable
https://github.com/vercel/pkg

[9] use ES6 import modules with pkg - javascript - Stack Overflow
https://stackoverflow.com/questions/58392391/use-es6-import-modules-with-pkg

[10] [11] GitHub - nexe/nexe: create a single executable out of your node.js apps
https://github.com/nexe/nexe

[12] [13] ESM Support · Issue #815 · nexe/nexe · GitHub
https://github.com/nexe/nexe/issues/815

[14] [15] [16] [17] Show HN: Astra – a new js2exe compiler | Hacker News
https://news.ycombinator.com/item?id=44042343

[18] [19] [27] [28] [29] [30] Single executable applications | Node.js v24.1.0 Documentation
https://nodejs.org/api/single-executable-applications.html

[20] [21] [22] [23] [24] [25] [26] @better-builds/nodec - npm
https://www.npmjs.com/package/%40better-builds%2Fnodec