ChatGPT

เปรียบเทียบ Bundler/Build Tool สำหรับ Node.js (TypeScript/JavaScript)

บทนำ

ในการพัฒนาโปรเจค TypeScript/JavaScript ที่รันบน Node.js การ bundle โค้ดให้เป็นไฟล์เดียวหรือโครงสร้างที่ไม่ต้องพึ่ง `node_modules` มีข้อดีหลายประการ ไม่ว่าจะเป็นการลดขนาดไฟล์ deploy, ลดจำนวนไฟล์ I/O ในช่วง runtime (สำคัญกับ environment อย่าง AWS Lambda ที่ต้องการ cold-start เร็วขึ้น) และทำให้การนำโค้ดขึ้น **production** ง่ายขึ้น (เช่น ใส่ไฟล์เดียวลง Docker image เล็ก ๆ ได้เลย) [1] นอกจากนี้ bundler ที่ดีควรรองรับทั้ง **CommonJS (CJS)** และ **ES Module (ESM)** อย่างราบรื่น เนื่องจาก ecosystem Node.js กำลังอยู่ช่วงเปลี่ยนผ่านไป ESM ดังนั้นเครื่องมือ build ควรจัดการกับโมดูลทั้งสองรูปแบบได้โดยไม่ยุ่งยาก ผู้ใช้ยังต้องการเครื่องมือที่ **ใช้งานง่าย** แทบไม่ต้องตั้งค่าเพิ่มเติม และสามารถใช้ได้กับหลาย environment เช่น สร้างเป็นไฟล์สำหรับรันใน Docker, ทำ CLI แบบ single file, แอป backend, หรือใช้ใน serverless เช่น AWS Lambda/Azure Functions โดย output ที่ได้ควรพร้อมใช้งานใน production (production-ready) จริง ๆ

ด้านล่างนี้เป็นการเปรียบเทียบเครื่องมือ bundler/build tool ยอดนิยมที่ตอบโจทย์ข้างต้น ได้แก่ **esbuild**, **tsup**, **SWC**, **Rollup**, **Vite** เป็นต้น โดยจะกล่าวถึงจุดเด่น/จุดด้อยของแต่ละตัว, กรณีการใช้งานที่เหมาะสม รวมถึงความสามารถในการ bundle dependencies อัตโนมัติ (ไม่ต้องระบุ `externals` เอง) และการนำไปใช้จริงใน production โดยชุมชนหรือบริษัทใหญ่ๆ

Esbuild

Esbuild เป็น JavaScript bundler รุ่นใหม่ที่มีจุดเด่นเรื่องความเร็วสูงมาก (เขียนด้วยภาษา Go) และรองรับ TypeScript ในตัว โดยสามารถ bundle โค้ดทั้งหมดและ dependencies เป็นไฟล์เดียวได้ง่าย ๆ ผ่านคำสั่ง CLI (มี option เช่น `--bundle`, `--platform=node` เพื่อกำหนด target เป็น Node) ซึ่งตรงตามโจทย์ที่ต้องการ output พร้อมรันใน production ทันที [1]

- **จุดเด่น:** ทำงานรวดเร็วเป็นพิเศษ (เร็วกว่า Webpack หรือ Parcel ถึงประมาณ **10-100 เท่า** ตามบล็อกของ AWS [2]), การตั้งค่าขั้นพื้นฐานง่าย (ระบุแค่ไฟล์ input/output ก็พอ), รองรับการ bundle ทั้งโค้ดเราและ third-party dependencies อัตโนมัติ (โดยค่าเริ่มต้น esbuild จะ bundle ทุกอย่างยกเว้น Node built-in modules เช่น `fs`, `path` ซึ่งเป็นไปตามคาดหมายสำหรับ runtime Node) นอกจากนี้ esbuild ยังรองรับทั้งการ output เป็น ESM หรือ CJS ตามต้องการ และมีการ **tree-shaking** เพื่อตัดโค้ดที่ไม่ถูกใช้ออกในการ build production ทำให้ขนาดไฟล์เล็กลง (เหมือน bundler รุ่นพี่อย่าง Rollup) [3]
- **จุดด้อย:** แม้ esbuild จะมี plugin system แต่ ecosystem ของปลั๊กอินและฟีเจอร์ขั้นสูงยังน้อยกว่า Webpack/Rollup ที่มีมานาน (เช่น การ code splitting ขั้นสูงหรือการ bundle assets บางชนิดอาจต้องพึ่งปลั๊กอินเสริม) นอกจากนี้ esbuild ยังเป็นโปรเจคที่ maintainer หลักมีคนเดียว (สถานะยังเป็นรุ่น 0.x) แต่โดยภาพรวมถือว่าเสถียรพอสำหรับใช้งานทั่วไปแล้ว และมี community ติดตามมากขึ้นเรื่อย ๆ [4]
- **การใช้งานที่เหมาะสม:** esbuild เหมาะมากสำหรับการ bundle แอป Node.js ในกรณีที่ต้องการ build รวดเร็วและ output ไฟล์เดียวพร้อมรัน เช่น ใช้ในการลดขนาด Docker image หรือ bundle โค้ด Lambda ก่อน deploy (ทาง AWS ยังแนะนำการใช้ esbuild เพื่อ bundle และ minify Lambda เพื่อประสิทธิภาพที่ดีขึ้นโดยลดเวลาการอ่านไฟล์หลายๆไฟล์ใน runtime [2])
- **การใช้งานใน production:** ปัจจุบัน esbuild ได้รับความนิยมสูงใน community (ได้รับความนิยมเพราะความเร็วเหนือคู่แข่ง [4]) และถูกนำไปใช้ในเครื่องมือ/เฟรมเวิร์คใหญ่ ๆ จำนวนมาก ยกตัวอย่างเช่น *Vite* ใช้ esbuild ในการประมวลผล

โค้ดช่วง dev, *AWS CDK* และ *SAM* รองรับการใช้ esbuild ในการ bundle Lambda, รวมถึงโครงการอื่น ๆ ของบริษัทใหญ่ (esbuild เองพัฒนาโดยวิศวกรของ Figma ซึ่งใช้งานมันจริงมาก่อน) ดังนั้น esbuild จึงถือว่า **production-ready** และผ่านการพิสูจน์ในงานจริงแล้วในระดับหนึ่ง

Tsup

Tsup เป็น build tool ที่พัฒนาขึ้นมาเพื่อทำให้การใช้ esbuild สะดวกขึ้นสำหรับโปรเจค TypeScript โดยมีแนวคิด “zero config” (ตั้งค่าน้อยมากหรือไม่ต้องตั้งค่าเลย) ผู้ใช้สามารถสั่ง `tsup` เพื่อ compile+bundle โค้ด TS ออกมาเป็นไฟล์ JavaScript ที่พร้อมใช้งานได้ทันที จุดเด่นของ tsup คือรองรับการ output ได้หลายรูปแบบ (เช่น ESM และ CJS) ภายในการรันคำสั่งครั้งเดียว, มีการสร้างไฟล์ประกาศ `.d.ts` ให้ (ถ้าต้องการ), และมีค่า default ที่เหมาะกับการทำ library **แต่**ก็สามารถปรับมาใช้ทำแอปพลิเคชันได้เช่นกัน

- **จุดเด่น:** ใช้งานง่ายมาก (มักไม่ต้องมี config ไฟล์เลย), ค่าเริ่มต้นฉลาด – ใช้ **esbuild** เบื้องหลังในการ bundle ทำให้ได้ความเร็วและประสิทธิภาพเช่นเดียวกับ esbuild, รองรับทั้งการ output เป็น CommonJS และ ESM (ระบุผ่าน flag `--format` ได้), มี built-in handling หลายอย่างเช่นการ minify, watch mode, และการแยกประเภท dependency อัตโนมัติ
- **การจัดการ dependency:** โดยปกติ tsup จะถือคติแบบการทำ library คือ *bundle* เฉพาะไฟล์ที่ *import* มา แต่จะ **externalize dependencies ที่ระบุใน** `package.json` (dependencies และ peerDependencies) โดยอัตโนมัติ – นั่นหมายความว่าแพ็กเกจที่อยู่ใน dependencies จะไม่ถูกรวมเข้าไฟล์ bundle [5] (ผู้ใช้คาดว่าจะติดตั้งเองเมื่อนำ library ไปใช้) อย่างไรก็ดี เราสามารถสั่งให้ tsup รวม dependency ทั้งหมดเข้า bundle ได้โดยการตั้งค่า `--no-external` หรือระบุแพ็กเกจที่ต้องการรวมลงไปในฟิลด์ `noExternal` ใน config ซึ่งสำหรับ use case แบบแอป (เช่น ทำไฟล์เดียวไป deploy) ผู้ใช้ก็มักจะปรับตรงนี้เพื่อให้ได้ไฟล์เดียวที่มีทุก dependency ครบ
- **จุดด้อย:** เนื่องจากค่าเริ่มต้นของ tsup ออกแบบมาสำหรับการสร้าง library บน npm จึง อาจทำให้ผู้ใช้สับสนเมื่อจะนำมา bundle แอป (เช่น จู่ๆ พบว่าโค้ดบางส่วนไม่ถูก bundle เพราะ tsup มองว่าเป็น external dependency) ต้องปรับตั้งค่าเพิ่มเล็กน้อย แต่ก็ไม่ยากนัก นอกจากนี้ tsup เองไม่ได้มีระบบปลั๊กอินที่เยอะ – ถ้าต้องการฟังก์ชันพิเศษนอกเหนือจากที่ esbuild ทำได้ ก็อาจจะต้องเรียกใช้ API esbuild เพิ่มเอง หรือใช้เครื่องมืออื่นแทน
- **การใช้งานที่เหมาะสม:** เหมาะกับโปรเจค TypeScript ที่ต้องการ build รวดเร็วและสะดวก ไม่ว่าจะเป็นการทำ **CLI tool**, **Library** แจกจ่ายบน npm, หรือแม้แต่ backend service ภายในบริษัท โดยเฉพาะอย่างยิ่งถ้าโปรเจคนั้นต้องการ output หลายฟอร์แมต (เช่น ESM/CJS) หรือไฟล์ declaration – tsup จัดการให้ได้หมดในการรันคำสั่งเดียว
- **การใช้งานใน production:** มีผู้ใช้ tsup ใน community ค่อนข้างมาก โดยเฉพาะกลุ่มนักพัฒนา library (เช่นแพ็กเกจ React/Vue ส่วนมากที่เขียนด้วย TS มักใช้ tsup ช่วย build) นอกจากนี้บริษัทบางแห่งก็ใช้ tsup ในงาน production จริง เช่นทีมพัฒนา refine (framework ตัวหนึ่ง) รายงานว่าใช้ tsup (ขับเคลื่อนโดย esbuild) เพื่อ build โปรเจคของพวกเขาใน production ได้ผลลัพธ์ที่รวดเร็วมาก [6] เรียกได้ว่า tsup นั้น **production-ready** เพราะมันก็เหมือนการใช้ esbuild นั่นเองแต่เพิ่มความสะดวกขึ้น

SWC

SWC (Speedy Web Compiler) เป็นเครื่องมือ compile/transpile โค้ด JavaScript/TypeScript ที่เขียนด้วย Rust ซึ่งถูกพูดถึงมากในช่วงหลัง เนื่องจากประสิทธิภาพสูงกว่า Babel มาก และถูกนำไปใช้ในโปรเจคใหญ่อย่าง Next.js ของ Vercel (ใช้ SWC แทน Babel สำหรับการแปลงโค้ด) อย่างไรก็ตาม **SWC ไม่ได้ถูกออกแบบเป็น “bundler” โดยตรง** – มันเน้นการแปลงซอร์สโค้ด (TS,

JSX ฯลฯ) ให้เป็น JS ตามมาตรฐานที่ต้องการ แต่ในส่วนการ bundle รวมไฟล์หลาย ๆ ไฟล์เข้าเป็นไฟล์เดียวนั้น SWC มีฟีเจอร์ชื่อ `spack` อยู่ ซึ่งยังอยู่ในช่วงทดลอง

- **จุดเด่น:** การประมวลผลโค้ด (transpile) รวดเร็วมาก (เนื่องจากเขียนด้วย Rust และใช้การประมวลผลแบบขนานเต็มประสิทธิภาพ CPU), รองรับการแปลงโค้ดทันสมัย (ESNext, JSX, TS) และการ minify ได้ในตัว, สามารถใช้แทน TypeScript compiler หรือ Babel ในงาน build เพื่อเร่งความเร็วได้ดี
- **จุดด้อย (ด้านการ bundling):** ฟีเจอร์ bundler (`spack`) ของ SWC ยังไม่สมบูรณ์และทางทีมพัฒนา **ประกาศว่าจะยกเลิกใน SWC v2** โดยแนะนำให้ไปใช้ "bundler อื่นที่สร้างบน SWC" แทน [7] เช่น Parcel v2, Turbopack, Rspack หรือ Farm ซึ่งเป็นเครื่องมือ bundler รุ่นใหม่ที่ใช้คอมไพเลอร์ความเร็วสูง (บางตัวก็ใช้ core ของ SWC) ดังนั้นการใช้ SWC ตรง ๆ เพื่อ bundle โปรเจค Node จึงอาจไม่ใช่ตัวเลือกที่ดีในตอนนี้
- **การใช้งานที่เหมาะสม:** ปัจจุบัน SWC เหมาะสำหรับใช้เป็นตัว **transpiler** ใน pipeline การ build มากกว่า (เช่นใช้ผ่านเครื่องมืออื่นอีกชั้นหนึ่ง) ตัวอย่างเช่น Next.js ใช้ SWC compile โค้ด TS/JS แล้วจึงใช้ Webpack ในการ bundle, หรือโครงการอื่น ๆ ที่ต้องการเร่งขั้นตอนการ compile สามารถใช้ @swc/cli เพื่อ transpile TS -> JS ได้รวดเร็วแทน tsc (แต่จะไม่ได้ไฟล์เดียว) ส่วนในอนาคต bundler รุ่นใหม่อย่าง **Turbopack** (ของ Vercel) หรือ **Rspack** (ทางเลือก Webpack ที่เขียนด้วย Rust) อาจกลายเป็นตัวเลือก bundler ที่รวมความเร็วระดับ SWC กับความสามารถในการรวมไฟล์แบบ Webpack ซึ่งน่าจับตามองสำหรับงาน production
- **การใช้งานใน production:** แม้ SWC ในฐานะ bundler จะยังไม่พร้อม แต่ SWC ในฐานะ compiler ถูกใช้งานจริงอย่างแพร่หลายแล้วใน production – Next.js v12+ ใช้มันกับทุกโปรเจค, เครื่องมือ build รุ่นใหม่หลายตัวก็รวม SWC เข้าไปด้วย เช่น **Parcel 2** ใช้ SWC เป็น engine ในการแปลงโค้ด, Cloudflare Workers ก็มีการทดลองใช้ SWC เพื่อแปลงโค้ด เป็นต้น กล่าวได้ว่า SWC นั้น production-ready ในส่วน compiler และกำลังมีบทบาทสูงขึ้นเรื่อย ๆ แต่ถ้าต้องการความสามารถ bundling แนะนำใช้ผ่านเครื่องมืออื่นตามที่กล่าวไป

Rollup

Rollup เป็น JavaScript bundler ยอดนิยมที่มีมานานและเป็นที่รู้จักดีในหมู่นักพัฒนา JS โดย Rollup โดดเด่นเรื่องการสร้างไฟล์ bundle ที่ **มีขนาดเล็กและสะอาด** เหมาะกับการทำ production build มาก (แนวคิดหลักคือ tree-shaking ทั้ง library และโค้ดเรา เพื่อตัดส่วนที่ไม่ใช้ทิ้ง) แตกต่างจาก Webpack ที่ output จะมีโค้ด runtime มากกว่า Rollup ทำให้ไฟล์ใหญ่กว่าในหลายกรณี [8] สำหรับ Node.js นั้น Rollup สามารถตั้งค่าให้ bundle เป็น target แบบ Node (เช่น externals สำหรับ Node built-ins) และสามารถรวมไฟล์ .js หลายไฟล์กับ dependencies เป็นไฟล์เดียวได้เช่นกัน (แต่ต้องใช้ plugin ช่วยสำหรับโมดูล CommonJS)

- **จุดเด่น:** ผลลัพธ์ bundle **มีประสิทธิภาพสูง** – โค้ดส่วนเกินน้อย, runtime เบาบาง, ได้ไฟล์ที่ compact กะทัดรัด (SWC ยังอ้างถึง Rollup ว่าเป็นมาตรฐานของ output ที่ compact [3]) ทำให้เหมาะกับการนำไปใช้ใน production โดยเฉพาะกับไฟล์ขนาดใหญ่ที่จะได้ประโยชน์จากการลดขนาด นอกจากนี้ Rollup ยังมีระบบปลั๊กอินจำนวนมากให้เลือกใช้ (เช่น plugin สำหรับแก้ไขโค้ดบางอย่างหรือจัดการ asset พิเศษ) และสามารถ output หลายรูปแบบโมดูล (สร้าง bundle แบบ ESM และ CJS ได้ในการ config เดียว)
- **จุดด้อย: การตั้งค่าค่อนข้างซับซ้อน** เมื่อเทียบกับเครื่องมือรุ่นใหม่ – โดยมากการใช้ Rollup ต้องเขียนไฟล์ `rollup.config.js` เพื่อระบุ entry, output format, และกำหนดปลั๊กอินต่าง ๆ (อย่างน้อยต้องมี plugin `@rollup/plugin-node-resolve` และ `@rollup/plugin-commonjs` เพื่อให้ Rollup bundle แพ็กเกจจาก `node_modules` ที่เป็น CommonJS ได้) ทำให้ initial setup ยุ่งกว่า esbuild/tsup ที่แทบไม่ต้องตั้งค่าอะไรเลย อีกทั้งความเร็วในการ build ก็จะช้ากว่า bundler รุ่นใหม่ ๆ (เพราะ Rollup เขียนด้วย JS ล้วน ๆ)
- **การจัดการ dependency:** โดยปกติ Rollup จะ **ไม่ bundle dependencies ที่เป็น external** ถ้าเราระบุไว้ (เช่น กำหนด `external` เป็นรายการ dependencies ทั้งหมดใน config เพื่อกันไม่ให้รวมเข้ามา) ซึ่งเป็นพฤติกรรมที่แนะนำสำหรับการสร้าง library (เพื่อเลี่ยงการ duplicate dependency ในโปรเจคผู้ใช้) แต่ถ้าต้องการ bundle แอป Node ให้เป็นไฟล์เดียว เราสามารถไม่ตั้งค่า `external` เลย แล้วใช้ plugin resolution+commonjs ปล่อยให้ Rollup ดึงทุก

dependency เข้ามาใส่ bundle ได้เช่นกัน (ต้องระวังว่าหาก dependency ไหนใช้ dynamic require แปลก ๆ Rollup อาจรวมไม่สำเร็จ)

- **การใช้งานที่เหมาะสม:** Rollupเหมาะกับงานสร้าง **JavaScript library** ที่ต้องการควบคุม output อย่างละเอียดและได้ไฟล์ขนาดเล็กที่สุด นอกจากนี้ยังใช้ในงาน frontend build บ่อยครั้ง (หลายคนใช้ Rollup ทำ bundle สำหรับเว็บที่ต้องโหลดเร็ว ๆ) ในบริบท Node.js หากต้องการ bundle โค้ดสำหรับ Cloudflare Workers หรือ Lambda ที่ขนาดเล็กที่สุดเท่าที่จะทำได้ (เช่นกรณีที่มี dependency จำนวนมากแต่ใช้โค้ดแค่นิดเดียวของแต่ละตัว) Rollup อาจให้ผลลัพธ์ไฟล์เล็กกว่า esbuild ในบางกรณี เพราะ tree-shaking ลึกกว่า แต่ก็ต้องแลกกับเวลา config และเวลา build ที่นานขึ้น
- **การใช้งานใน production:** Rollup ผ่านการพิสูจน์ใน production มานานหลายปี – โครงการชื่อดังจำนวนมากใช้ Rollup ในการจัดทำ build เช่น **React** (ใช้ Rollup สร้างไฟล์ UMD), **D3.js**, **Three.js**, และอีกมาก โดยเฉพาะบรรดา framework/library ฝั่ง frontend มักนิยม Rollup ตั้งแต่ก่อนจะมี Webpack ด้วยซ้ำ สำหรับ Node ecosystem เอง ก็มีการใช้ Rollup ในการสร้างเครื่องมือ CLI หรือตัว SDK ต่าง ๆ บางตัวเช่นกัน (แต่ปัจจุบันเริ่มมีการย้ายไป esbuild เพื่อความเร็วมากขึ้น) โดยสรุป Rollup เป็น bundler ที่ **เสถียรและ production-ready สูง** อีกตัวหนึ่ง แต่เหมาะกับผู้ที่ยอมลงทุนเวลา config เพื่อผลลัพธ์ไฟล์ที่เล็กและสะอาดที่สุด

Vite

Vite เป็นเครื่องมือ build รุ่นใหม่ (พัฒนาโดย Evan You ผู้สร้าง Vue.js) ที่มุ่งเน้นประสบการณ์การพัฒนา (dev experience) ที่รวดเร็ว ด้วยการใช้ esbuild ในการเสิร์ฟโค้ดระหว่างพัฒนาแบบ HMR (Hot Module Replacement) และใช้ Rollup ในการ build production ตัว Vite เองถูกออกแบบมาสำหรับงาน **Frontend (เว็บ)** เป็นหลัก แต่มีโหมด library build ที่สามารถใช้สร้างแพ็กเกจ JavaScript ได้เช่นกัน

- **จุดเด่น:** สำหรับงานเว็บ Vite ให้ประสบการณ์ “พร้อมใช้” ที่ดีมาก – แทบไม่ต้องตั้งค่า (zero config) ก็รัน dev server พร้อม HMR ได้ทันที, build production ก็ได้ optimize code splitting, minify ต่าง ๆ โดยอัตโนมัติ (ใช้ความสามารถของ Rollup) นอกจากนี้ Vite ยังรองรับการ output หลายฟอร์แมตกรณีทำ library (ESM/CJS/UMD) ได้ และจัดการกับ asset (เช่น .css, .vue, .ts) ได้สะดวกผ่านระบบ plugin ที่เข้าใจไฟล์ประเภทต่าง ๆ ตั้งแต่ต้น
- **จุดด้อย: ไม่เหมาะกับโปรเจค Node.js ที่เป็น backend-only** หรือ CLI ที่ไม่มีส่วนหน้าเว็บ – เนื่องจาก Vite ถูกออกแบบเพื่อพัฒนาเว็บแอป จึงมีสมมติฐานเรื่องการทำงานแบบมี HTML/Browser environment (เช่นตอน dev จะมีหน้าสำหรับโหลดโมดูลผ่าน `<script type=module>` เป็นต้น) หากเรานำ Vite ไปใช้ bundle โค้ด Node.js ล้วน ๆ อาจไม่ได้ประโยชน์อะไรนัก เทียบกับการใช้ esbuild ตรง ๆ ที่ง่ายกว่า นอกจากนี้ Vite เองเวลา build มันยังสร้าง output เป็นชุดไฟล์ (อาจมีหลายไฟล์กรณี code splitting) ซึ่งขัดกับโจทย์ที่อยากได้ไฟล์เดียวแบบไม่พึ่ง node_modules
- **การใช้งานที่เหมาะสม:** ใช้เมื่อโปรเจคของคุณมีส่วน frontend ที่ซับซ้อนและต้องการ dev server เร็ว ๆ – Vite จะช่วยให้รันและ HMR ได้ทันใจ และ build ออกมาพร้อม deploy สำหรับเว็บได้เลย กรณีที่เป็น **SSR (Server-Side Rendering)**, Vite ก็สามารถช่วย bundle ส่วนโค้ดที่รันบน Node (ส่วน server) ได้ด้วย โดย Vite จะมีโหมด SSR build (ใช้ esbuild ช่วย) ซึ่ง output ออกมาเป็นไฟล์สำหรับรันใน Node (แต่ยังไม่ใช่ไฟล์รวม dependency เดียวแบบ ncc) อย่างไรก็ตาม ถ้าโฟกัสหลักคือการรวมโค้ด backend Node.js ให้เป็นไฟล์เดียวเพื่อใช้ใน Lambda หรือ Docker เล็ก ๆ Vite ไม่ใช่ตัวเลือกหลัก
- **การใช้งานใน production:** Vite ได้รับความนิยมสูงมากในหมู่นักพัฒนา frontend ทันทีที่เปิดตัว และปัจจุบันถูกใช้ใน production โดย project ใหญ่ ๆ เช่น **Laravel Mix (Vite)**, **Nuxt 3 (ใช้ Vite เป็น option)**, รวมถึง framework ใหม่ๆ อย่าง **SvelteKit** ก็ใช้ Vite เป็น default dev server เป็นต้น กล่าวคือ Vite นั้น production-ready สำหรับงาน build frontend แน่นอน ส่วนการใช้ Vite ในบริบทอื่น (เช่น backend library) ก็พอทำได้แต่ไม่ค่อยมีใครใช้

อื่น ๆ ที่น่าสนใจ (Webpack, NCC, Parcel)

นอกจากเครื่องมือข้างต้น ยังมี bundler/build tool อื่น ๆ ที่ควรกล่าวถึงเพื่อความครบถ้วน:

- **Webpack:** bundler รุ่นเก๋าที่ได้รับความนิยมสูงสุดในยุคก่อน เนื่องจากยืดหยุ่นและมีปลั๊กอิน/loader มากมาย Webpack สามารถปรับโหมดให้ bundle สำหรับ Node.js (ตั้ง `target: node`) และรวมไฟล์เป็นไฟล์เดียวได้เช่นกัน จุดแข็งคือ **เสถียรภาพและ ecosystem** – แทบทุกอย่างทำได้ด้วย Webpack แต่ **จุดด้อย**คือความซับซ้อนในการตั้งค่าและประสิทธิภาพ (ช้ากว่า bundler รุ่นใหม่อย่าง esbuild/SWC มาก) อีกทั้ง output ของ Webpack มักมี runtime code แทรกค่อนข้างเยอะ ทำให้ไฟล์ใหญ่เกินจำเป็นเมื่อเทียบกับเครื่องมืออย่าง Rollup [8] ดังนั้นสำหรับ use case ที่ต้องการไฟล์เล็กและ build เร็ว ปัจจุบันคนจึงมักหันไปใช้ esbuild/rollup มากกว่า แต่ถ้าโปรเจคมีการตั้งค่า Webpack เดิมอยู่หรือมีความต้องการซับซ้อน (เช่นการ bundle asset แปลก ๆ) Webpack ก็ยังเป็นตัวเลือกที่ปลอดภัย (ใช้กันใน production มานับไม่ถ้วนโดยบริษัททุกขนาด)

- **Vercel NCC:** *Node.js Compile and Compress* หรือเรียกสั้น ๆ ว่า **ncc** เป็นเครื่องมือ CLI ที่พัฒนาโดยทีม Vercel (ผู้สร้าง Next.js) ซึ่งออกแบบมาเพื่อ **bundle โปรเจค Node.js (รวม dependencies ทั้งหมด) เป็นไฟล์เดียว** แบบไม่ต้องตั้งค่าเลย (zero-config) เหมาะกับการเอาไปใช้ใน serverless หรือแจกจ่าย CLI มากๆ [9] โดย ncc ใช้ Webpack เบื้องหลังแต่ปรับจูนค่าให้เหมาะกับ Node (เช่น built-in externals) ผู้ใช้แค่รัน `ncc build index.js -o dist` ก็จะได้ไฟล์ `dist/index.js` ไฟล์เดียวที่รวมโค้ดเรากับ dependencies ทุกตัวพร้อมรันบน Node ได้ทันที จุดเด่นคือรองรับ **TypeScript ในตัว**, จัดการ asset และ native addons ได้, handle กรณี dynamic require ได้ดีกว่าบันเดลเลอร์ทั่วไป (เพราะออกแบบมาสำหรับ Node โดยเฉพาะ) Community นิยมใช้ ncc ในการเตรียมโค้ดสำหรับรันบน **AWS Lambda หรือ CLI** ต่าง ๆ มากมาย ถือเป็นเครื่องมือที่ **production-proven** (Vercel เองก็ใช้ในระบบของตน และโปรเจคโอเพนซอร์สอย่าง Prisma, vercel/next.js repo บางส่วนก็ใช้ ncc)

- **Parcel 2:** เป็น bundler แบบ zero-config อีกตัวหนึ่ง (คล้าย Webpack แต่พยายามทำงานอัตโนมัติมากกว่า) จุดเด่นคือใช้งานง่ายและทำ **code splitting**, tree-shaking, asset bundling ได้โดยอัตโนมัติ Parcel 2 มีการใช้ SWC เป็น engine ภายในเพื่อเพิ่มความเร็ว จุดแข็งคือเหมาะกับงาน frontend ที่ไม่อยากตั้งค่าเยอะ ๆ แต่สำหรับ **Node.js** นั้น Parcel ก็สามารถ bundle ไฟล์สำหรับ Node ได้เช่นกัน (ระบุ `--target node` ตอน build) อย่างไรก็ตาม ในบริบทของการสร้างไฟล์เดียวเพื่อ deploy Lambda หรือ CLI คนมักเลือกเครื่องมือเฉพาะทางกว่าอย่าง ncc หรือ esbuild มากกว่า Parcel

สรุปและข้อแนะนำ

เมื่อต้องเลือก bundler/build tool สำหรับโปรเจค Node.js (TS/JS) ในปี 2025 นี้ **ไม่มีตัวใดที่เหมาะกับทุกกรณีโดยสมบูรณ์** แต่สามารถเลือกใช้ตามจุดแข็งดังนี้:

- **ใช้งานทั่วไป (Backend service, CLI, Lambda):** แนะนำ **esbuild** หรือ **tsup** เนื่องจากความเร็วและความง่ายในการตั้งค่า ทั้งสองสามารถ bundle โค้ดและ dependencies เป็นไฟล์เดียวได้ไม่ยุ่งยาก (tsup จะต้องปรับ noExternal เล็กน้อย) ซึ่งตอบโจทย์การนำไป deploy ใน Docker image เล็ก ๆ หรืออัปโหลดขึ้น Azure Function/AWS Lambda ได้ทันที [1] [2] Esbuild/tsup มี community รองรับมากและถูกพิสูจน์แล้วว่า production-ready

- **กรณีที่ต้องการ Zero-Config และรวม dependency อัตโนมัติ 100%: Vercel NCC** เป็นตัวเลือกที่โดดเด่น – เพียงคำสั่งเดียวก็ได้ไฟล์พร้อมรันที่มีทุกอย่างครบ (ไม่ต้องมาคอยใส่ `external` เอง) เหมาะมากสำหรับการเตรียมโค้ดไปลง serverless หรือทำเครื่องมือ command-line แจกจ่าย [9]

- **เน้นขนาดไฟล์เล็กที่สุด/การ optimize สูงสุด: Rollup** ยังคงเป็นทางเลือกที่ดี ด้วยความสามารถ tree-shaking ขั้นสูง และ output ที่สะอาด [3] (หากยอมแลกกับเวลาคอนฟิกและ build ที่นานขึ้น) เหมาะกับการสร้าง library หรืองานที่ต้องโหลดไฟล์ขนาดใหญ่บ่อย ๆ (เช่น Cloudflare Workers ที่มีขนาดไฟล์จำกัด)

- **งาน Frontend/Fullstack ที่มีทั้งส่วนเว็บและส่วน Node: Vite** เป็นตัวเลือกที่ให้ความสะดวกในการพัฒนาส่วน frontend มาก (HMR รวดเร็ว) และสามารถใช้ build ส่วน backend (เช่น SSR) ได้ดีพอสมควร หากโปรเจคของคุณมีลักษณะ fullstack การใช้ Vite ตัวเดียวครอบทั้ง frontend/backend อาจให้ workflow ที่ง่าย อย่างไรก็ตามหากส่วน backend ต้องการ bundle ไปรันบน Lambda เป็นต้น อาจต้องใช้ esbuild หรือตัวอื่นช่วยในขั้นตอนสุดท้ายอยู่ดี

- **เทคโนโลยีอนาคต:** มี bundler รุ่นใหม่อย่าง **Turbopack** (Next.js ใหม่) และ **Rspack** (โครงการโดย Webpack Team) ที่พยายามผสานความเร็วระดับ Rust เข้ากับระบบ bundling แบบ Webpack ซึ่งสัญญาว่าจะให้ทั้งความเร็วและความยืดหยุ่น ปัจจุบันยังอยู่ในช่วงทดสอบ แต่ก็น่าจับตาสำหรับอนาคต ในขณะที่ **SWC** เองจะมุ่งเป็นคอมไพเลอร์ให้เครื่องมือเหล่านี้แทน

โดยสรุป สำหรับโปรเจค Node.js ทั่วไป **esbuild/tsup** เป็นตัวเลือกที่สมดุลสุด (ง่าย เร็ว output ใช้งานได้เลย) รองลงมาหากต้องการ zero-config แบบรวมหมดก็ใช้ **ncc** ส่วน **Rollup/Vite** จะเหมาะกับเคสเฉพาะทางหรือโปรเจคที่มีส่วน frontend มากกว่า ทั้งหมดนี้ล้วนสามารถให้ output ที่ **production-ready** ได้ ขึ้นอยู่กับว่าตรงกับกรณีการใช้งานของเราหรือไม่ และหลาย ๆ เครื่องมือก็ได้รับการพิสูจน์ในงานจริงโดยชุมชนและองค์กรใหญ่แล้วว่ามีความน่าเชื่อถือในระดับหนึ่งครับ [6] [2]

---

[1] Deploying NodeJS Apps with ESBuild and Docker | Martin C. Richards
https://www.martinrichards.me/post/building_nodejs_apps_with_esbuild_and_docker/

[2] Optimizing Node.js dependencies in AWS Lambda | AWS Compute Blog
https://aws.amazon.com/blogs/compute/optimizing-node-js-dependencies-in-aws-lambda/

[3] [7] [8] Bundling (swcpack)
https://swc.rs/docs/usage/bundling

[4] [6] esbuild - Next-generation JavaScript bundler - DEV Community
https://dev.to/refine/esbuild-next-generation-javascript-bundler-45ke?comments_sort=latest

[5] How can include all depencies in the final index.js ? · Issue #619 · egoist/tsup · GitHub
https://github.com/egoist/tsup/issues/619

[9] GitHub - vercel/ncc: Compile a Node.js project into a single file. Supports TypeScript, binary addons, dynamic requires.
https://github.com/vercel/ncc