Bridging CommonJS and ESM in Node.js

•••

Joyee Cheung

About me

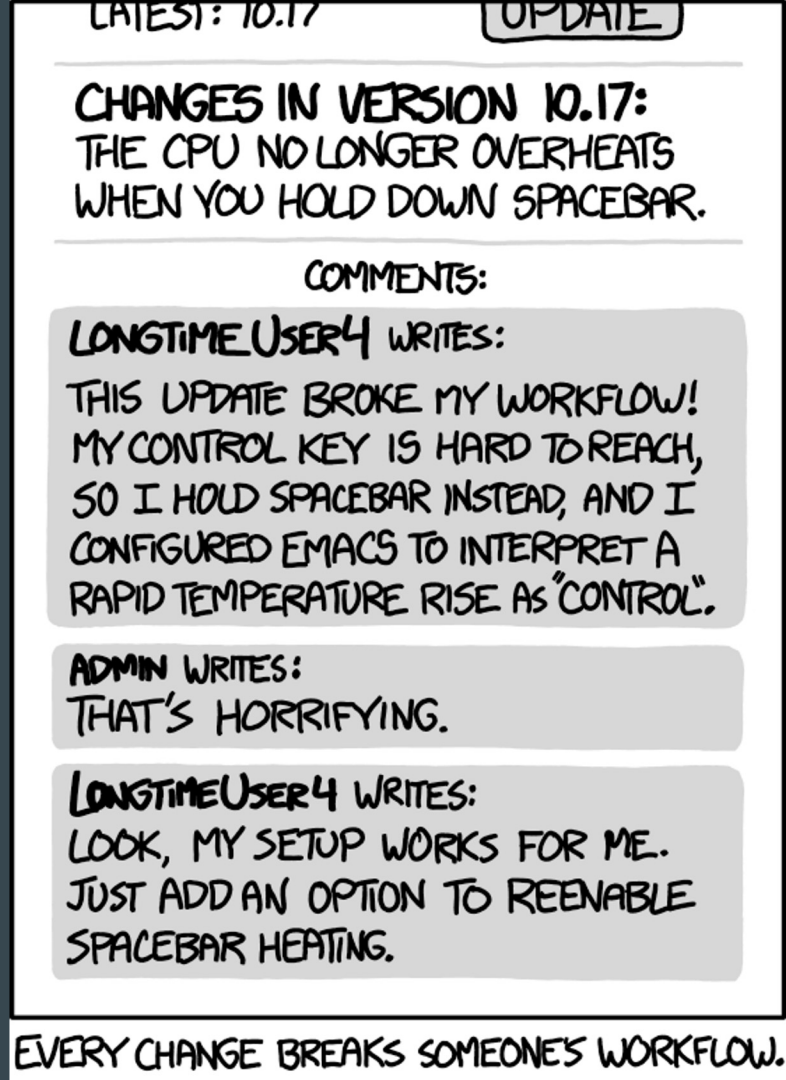
- Igalia
- Sponsored by Bloomberg on my Node.js work
- Member of Node.js TSC and V8 committer
- @joyeecheung on GitHub

It’s a story about...

- Moving an ecosystem forward by providing a path with non-breaking, incremental upgrades
  - Asking everyone in a huge ecosystem to make breaking changes to migrate is not effective - Node.js tried it for 5 years with little success

It’s a story about...

- Experimenting changes in a heavily relied upon subsystem with high compatibility risk
  - Node.js uses semver but the priority of stability make it resembles the Web
  - Every change breaks someone’s workflow
  - How do we minimize the impact?



History of ESM in Node.js

Node.js was created in 2008 and added support for the module system proposed as part of CommonJS in 2009 (CJS)

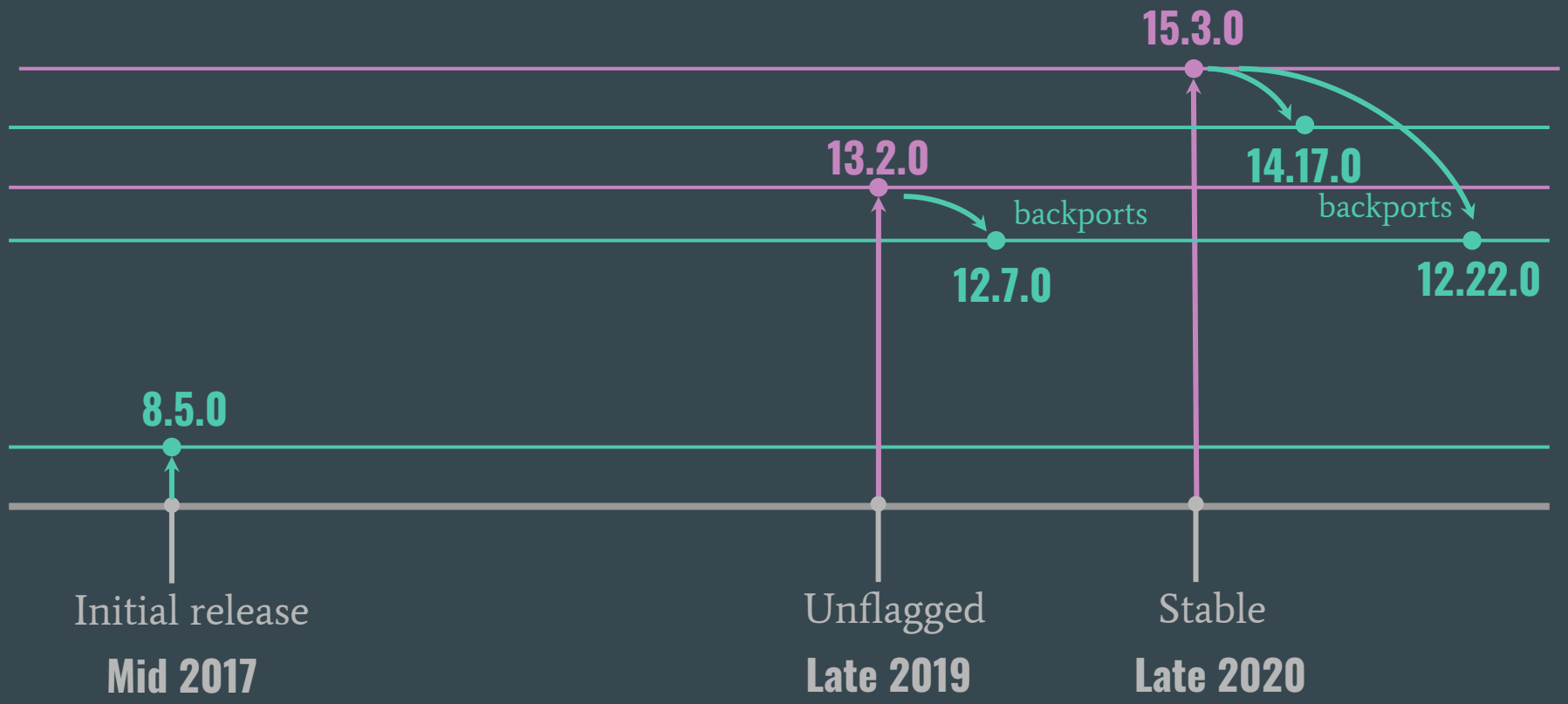
```
exports.log = function log() {}
const { log } = require('./logger.js');
```

In ES2015, JavaScript got a standardized module format – ESM (ECMAScript Modules)

```
export function log() {}
import { log } from ‘./logger.js’;
```

Proposals and initial development of Node.js ESM support started in 2015 but it took a long time to debate and develop interoperability between the two...

History of ESM (from ES2015) in Node.js



History of ESM in Node.js

At the time of stabilization (v15.3.0):

```
// CJS Provider: logger.js
module.exports = class Logger{};
module.exports.log = function log() {}

// ESM Consumer can load CJS via import
import Logger from './logger.js'; // => module.exports
import { log } from './logger.js'; // Detected with static analysis
```

History of ESM in Node.js

At the time of stabilization (v15.3.0):

```
// ESM consumer cannot load CJS via require()
require('./logger.js'); // ❌ ReferenceError: require is not defined

// If they have to load CJS dynamically...createRequire()
import module from 'node:module';
const require = module.createRequire(import.meta.url);
require('./logger.js');
```

History of ESM in Node.js

At the time of stabilization (v15.3.0):

```
// Unlike CJS, ESM can't import from extensionless paths
import Logger from './logger'; // ❌ Throws: requires a file extension.
await import('./logger.js'); // Top-level await works
```

History of ESM in Node.js

At the time of stabilization (v15.3.0):

```
// ESM provider
export default function log() {}

// CJS consumer cannot load ESM via require()
require('./logger.mjs'); // ❌ Throws ERR_REQUIRE_ESM!

// CJS consumer can load ESM via import(),
// but it returns a promise and only works in async code
import('./logger.mjs').then((namespace) => { namespace.log() });
```

Implications of lack of require(esm)

- CJS could not load ESM without coloring the dependency graph async
- Majority of the ecosystem still effectively run CJS
- Some providers want to use ESM without breaking users and losing popularity - they started to invent various workarounds...

Writing ESM != running ESM: Faux-ESM

- Packages, frameworks and tools transpile ESM to CJS - faux ESM
- Don't always work with real ESM
- Ripple effect

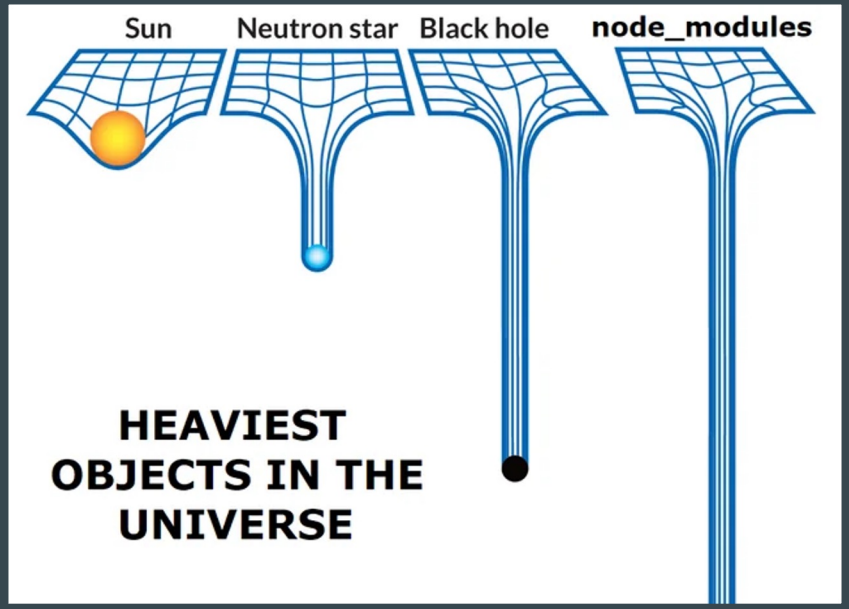
```
// Users write: handler_loaded_by_framework.ts
import { foo } from 'external_esm';
export default function handler() { return foo(); }

// Frameworks run: handler_loaded_by_framework.js
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = handler;
// Throws ERR_REQUIRE_ESM from code authored in ESM!?
const external_esm_1 = require("external_esm");
function handler() { return (0, external_esm_1.foo)(); }
```

Dual package

- Many packages ship both to support both consumers: supply ESM to ESM, CJS to CJS
- Doubles the size of node_modules...

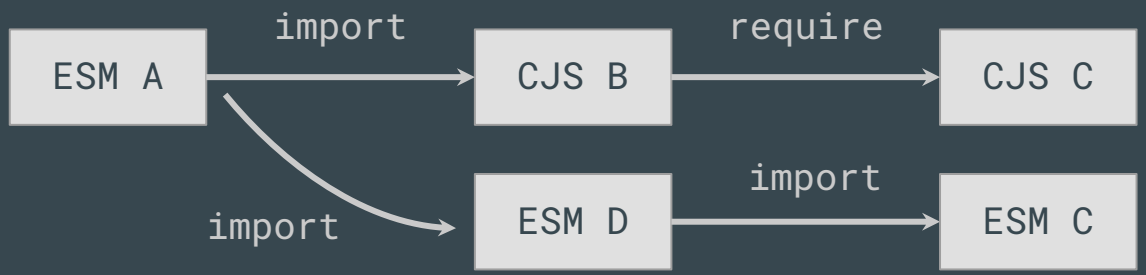
Dual package

- Many packages ship both to support both consumers: supply ESM to ESM, CJS to CJS
- Doubles the size of node_modules...



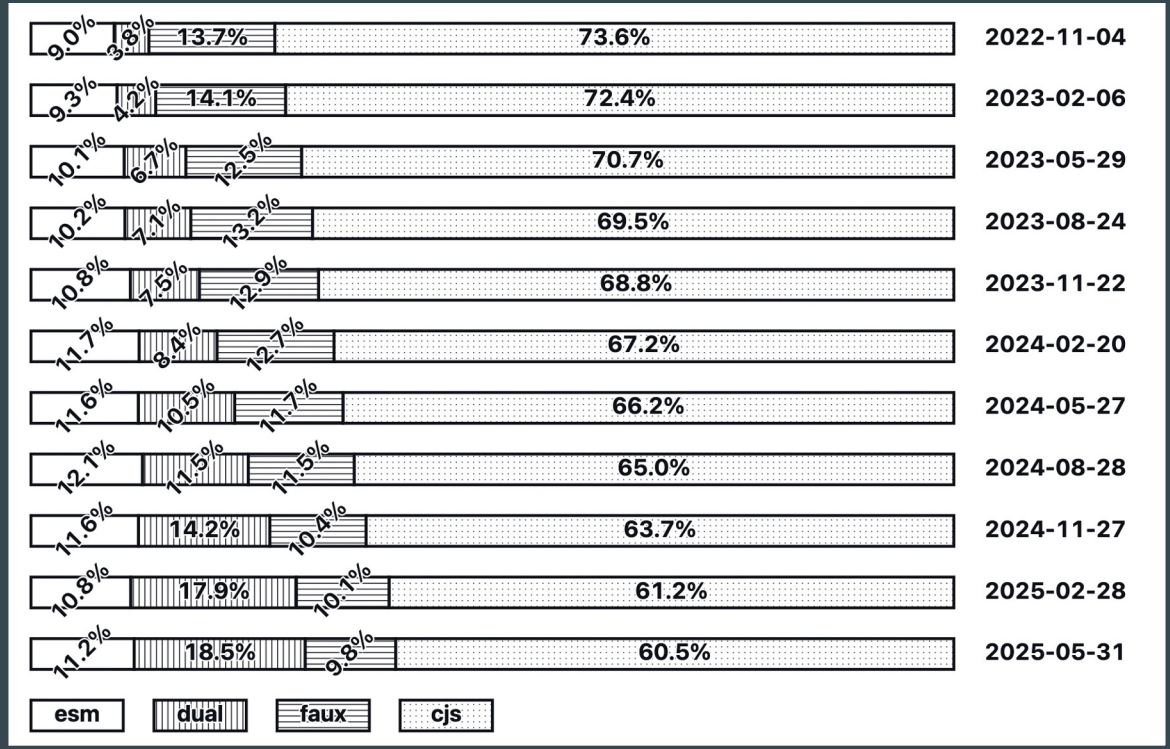
Dual package

- Dual package hazard



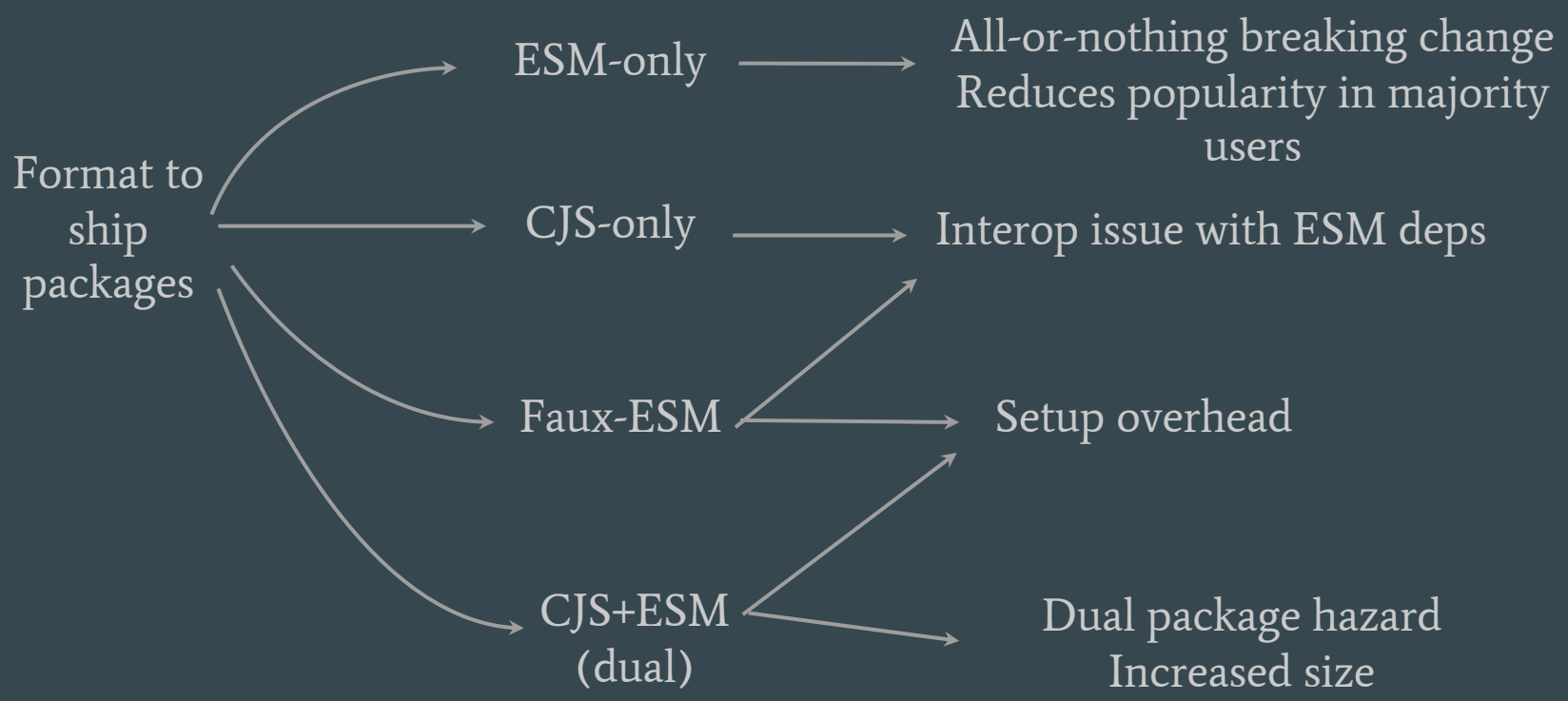
💥 Two version of the same package in the same graph!!

Implications of lack of require(esm)

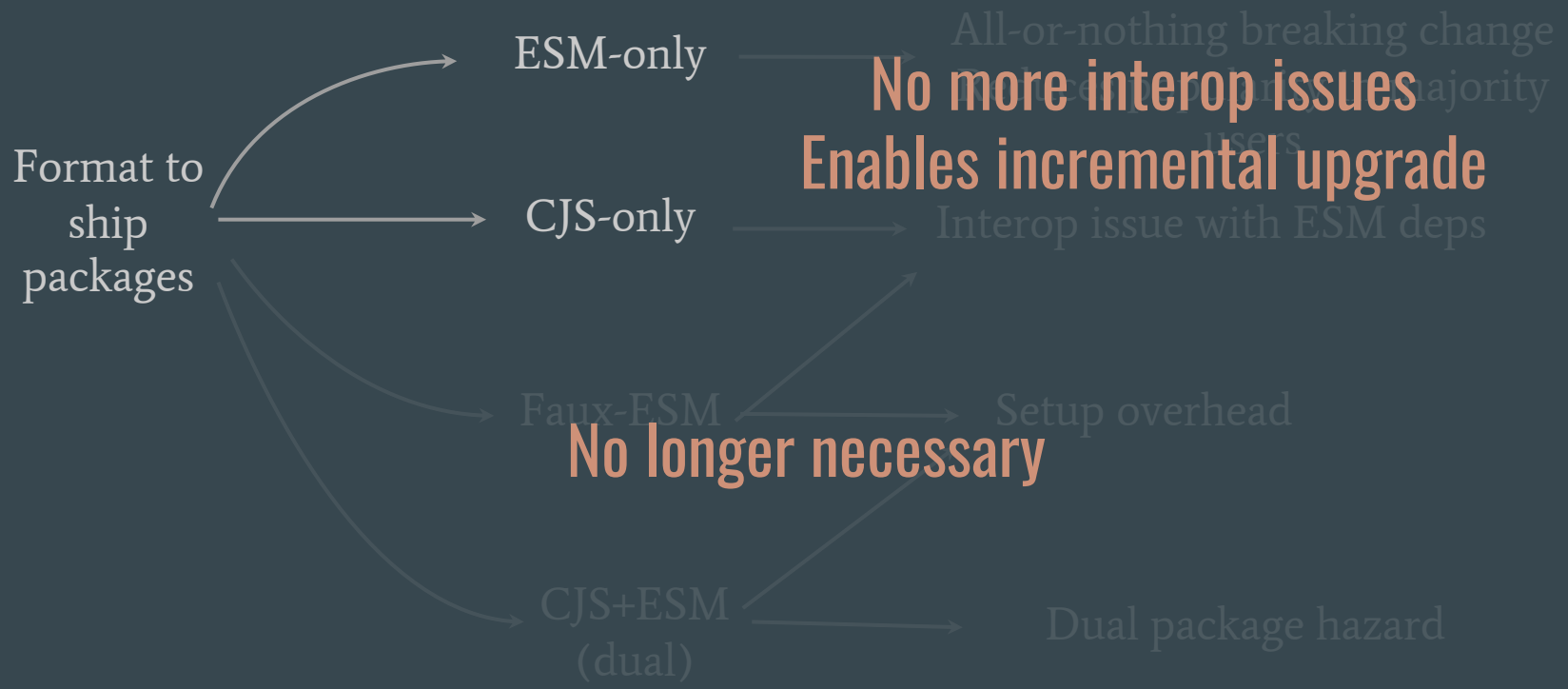


Source: https://github.com/wooorm/npm-esm-vs-cjs/

Implications of lack of require(esm)



If we have require(esm)...🤔



The myth of “ESM is async, require() is sync”

- Not that many people knew “it can be done”
- Those who did, didn’t pursue it further after initial attempt in 2019
- People involved in ESM implementation/specification knew that in the spec, ESM is only async when it contains top-level await
- Most people didn’t work on those (e.g. myself), assumed ESM is always async - even the Node.js documentation said so - and didn’t think about taking a stab at require(esm) at all

require

Sometimes, one should ignore what the documentation says..

The CommonJS module `require` always treats the files it references as CommonJS.

Using `require` to load an ES module is not supported because ES modules have asynchronous execution. Inste
use `import()` to load an ES module from a CommonJS module.

ESM without top-level await is synchronous

9. If *module*.[[HasTLA]] is **false**, then
    a. Assert: *capability* is not present.
    b. Push *moduleContext* onto the execution context stack; *moduleContext* is now the running execution context.
    c. Let *result* be Completion(Evaluation of *module*.[[ECMAScriptCode]]).
    d. Suspend *moduleContext* and remove it from the execution context stack.
    e. Resume the context that is now on the top of the execution context stack as the running execution context.
    f. If *result* is an abrupt completion, then
        i. Return ? *result*.
10. Else,
    a. Assert: *capability* is a PromiseCapability Record.
    b. Perform AsyncBlockStart(*capability*, *module*.[[ECMAScriptCode]], *moduleContext*).

https://tc39.es/ecma262/#sec-source-text-module-record-execute-module

ESM without top-level await is synchronous

Confirmed later that this was intentional, also relied on by bundlers

Normative: Synchronous based on a syntax and module graph #61

Merged **littledan** merged 2 commits into `tc39:master` from `littledan:statically-synchronous` on Mar 26, 2019

Conversation 34 | Commits 2 | Checks 0 | Files changed 2

littledan commented on Mar 19, 2019 Member

This patch is a variant on #49 which determines which module subgraphs are to be executed synchronously based on syntax (whether the module contains a top-level await syntactically) and the dependency graph (whether it imports a module which contains a top-level await, recursively). This fixed check is designed to be more predictable and analyzable.

ESM without top-level await is synchronous

This means as a host, Node.js could implement this:

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

Up to Node.js to make it synchronous

ESM without top-level await is synchronous

This means as a host, Node.js could implement this:

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

Check if it can be evaluated synchronously

ESM without top-level await is synchronous

This means as a host, Node.js could implement this:

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

No need to wait for anything if there's no TLA

Synchronous-only ESM on the Web

- ServiceWorkers disallows asynchronous module graphs (with top-level await)
- This saved us from having to add an API to V8 for that hasTopLevelAwaitInGraph() check the peusdocode before - it was already added for Chrome to implement similar semantics for ServiceWorkers in 2020

9. If *script* is null or Is Async Module with *script*'s record, *script*'s base URL, and « » is true, then:

    1. Invoke Reject Job Promise with *job* and `TypeError`.

        > Note: This will do nothing if Reject Job Promise was previously invoked with "`SecurityError`" `DOMException`.

    2. If *newestWorker* is null, then remove registration map[(*registration*'s storage key, serialized *scopeURL*)].
    3. Invoke Finish Job with *job* and abort these steps.

Restarting require(esm) in Node.js

In late 2023, I learned about the semantics when reading V8 code, discussed with other contributors who knew more about ESM in Node.js

```
// Pseudo code - this needs access to native V8 APIs.
function requireESM(specifier) {
  const linkedModule = fetchModuleGraphAndLinkSync(specifier);
  if (linkedModule.hasTopLevelAwaitInGraph()) {
    throw new ERR_REQUIRE_ASYNC_MODULE;
  }
  const promise = linkedModule.evaluate();
  // This is guaranteed by the ECMAScript specification.
  assert.strictEqual(getPromiseState(promise), 'fulfilled');
  assert.strictEqual(unwrapPromise(promise), undefined);
  // The namespace is guaranteed to be be fully evaluated at this point if the
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

The ESM loader only had asynchronous version of this back then, and it's ~3K lines of code that I had barely read before 😨

Restarting require(esm) in Node.js

- Wait for others who were more familiar with the ESM loader to refactor it
- A few months later, working on compile cache, ended up refactoring the compilation part of the ESM loader to make the compilation go through the cache, then ended up reading the whole thing...

src: use dedicated routine to compile function for builtin CJS loader

Merged **nodejs-github-bot** merged 1 commit into `nodejs:main` from `joyeecheung:cjs-compile` on Mar 11

Conversation 12 | Commits 1 | Checks 29 | Files changed 8

joyeecheung commented on Mar 8 • edited — Member

So that we can use it to handle code caching in a central place.

Needed by #47472, split out from #51977

Restarting require(esm) in Node.js

- 💡 : instead of refactoring that ~3K lines, maybe it's easier to just add new lines to implement a synchronous and trimmed-down ESM loading path for require()
  - Could already see it in my head
  - Lines added are easier to backport to older LTS than lines changed
  - Got support from Bloomberg to work part-time on this ❤️

Restarting require(esm) in Node.js

- Reaction was very positive
- Some edges needed more work, but we all agreed that it can be a follow-up whilst the feature is behind a flag (nothing comes out perfect at the first time anyway)

module: support require()ing synchronous ESM graphs #51977

Closed **joyeecheung** wants to merge 0 commits into `nodejs:main` from `joyeecheung:require-esm`

* Ignore this GitHub diff mess-up 😅 commit was 5f7fad2

Conversation 108 | Commits 0 | Checks 0 | Files changed 0

joyeecheung commented on Mar 5 • edited — Member

Summary

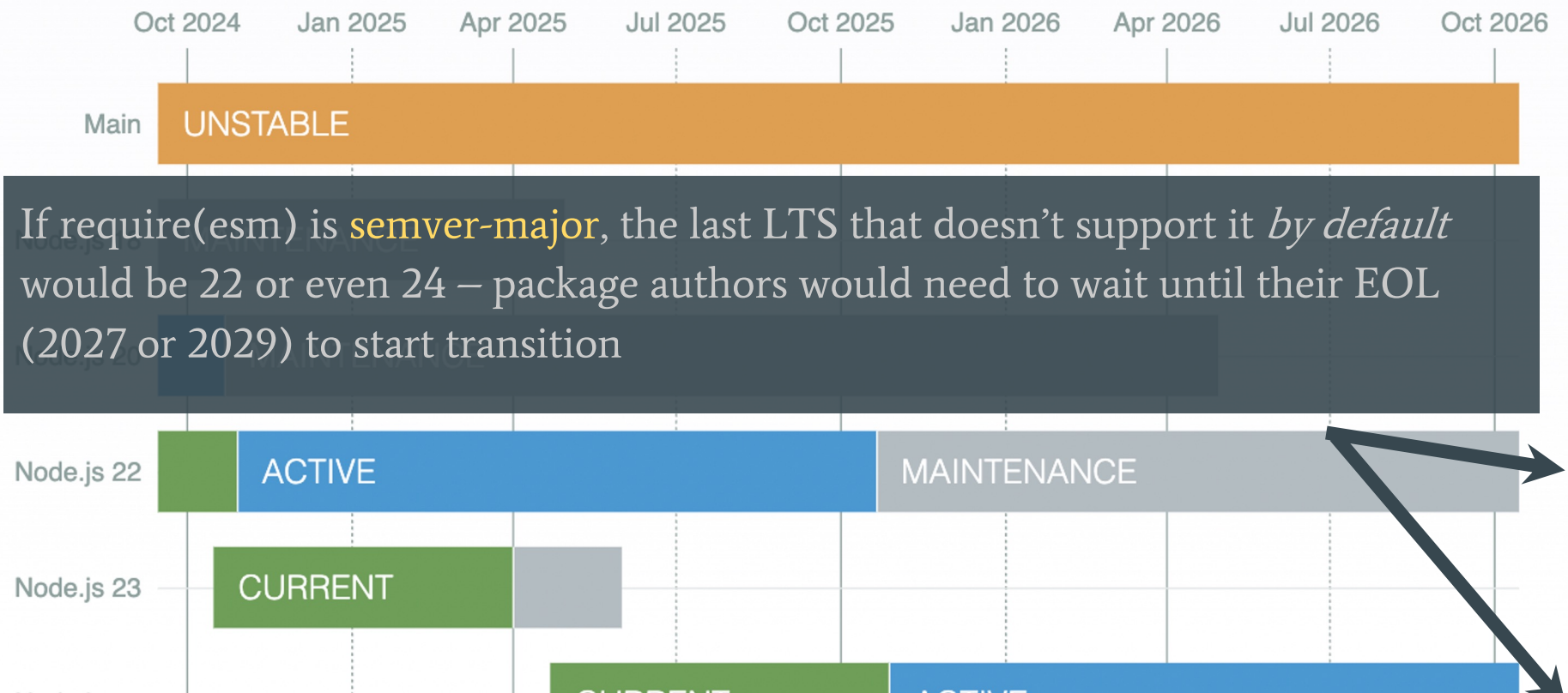
This patch adds `require()` support for synchronous ESM graphs under the flag `--experimental-require-module`

This is based on the the following design concept of ESM

Stabilization & Backporting

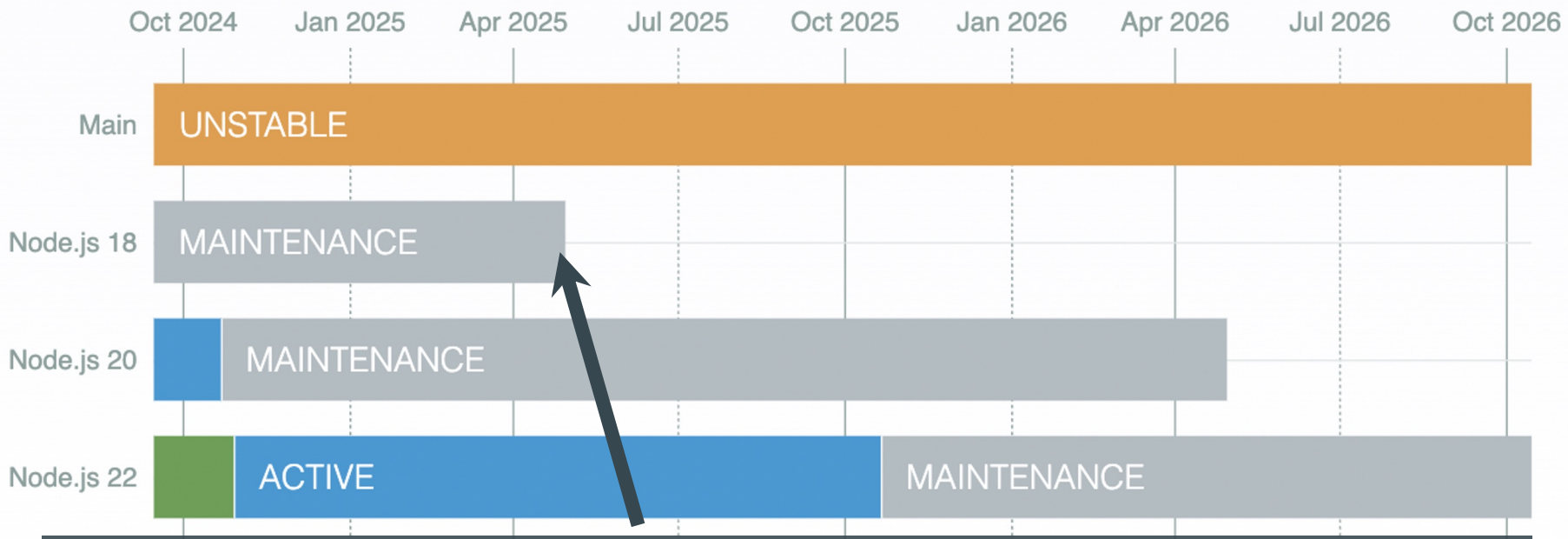
- Released to v22, unflagged in v23
- Many conventions and workarounds already existed in the ecosystem to work around the interoperability issues
- Working with package maintainers, test the ecosystem and try not to break existing code / step on their toes

Stabilization & Backporting



If require(esm) is semver-major, the last LTS that doesn't support it *by default* would be 22 or even 24 – package authors would need to wait until their EOL (2027 or 2029) to start transition

Stabilization & Backporting



If it's semver-minor, it can be backported to 22 and 20, so package authors can fully rely on it and start the transition from May 2025

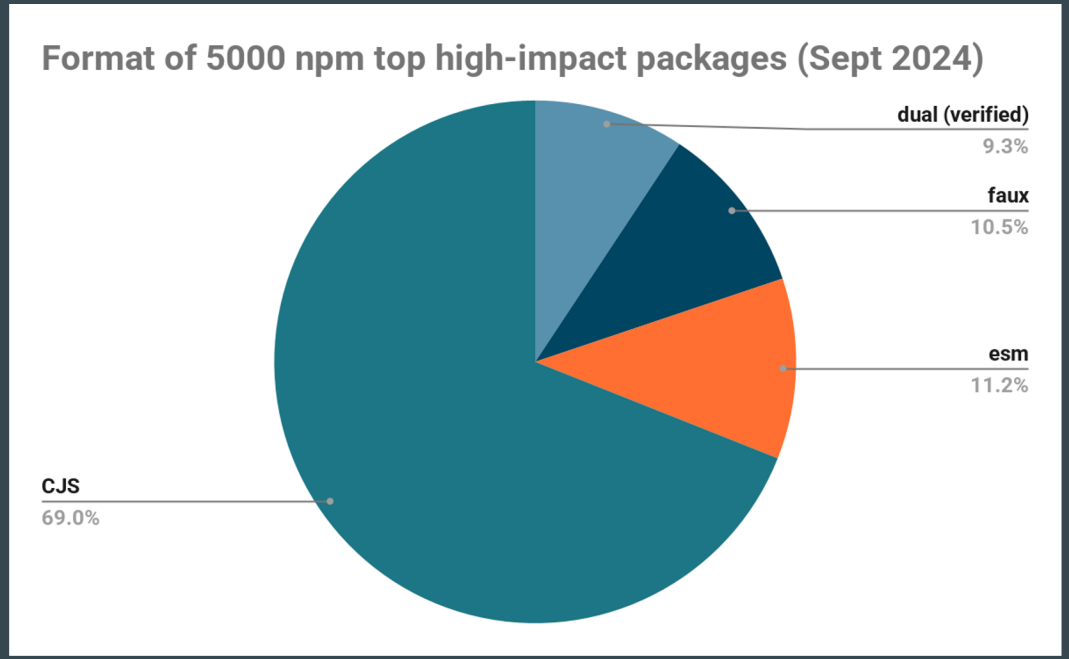
Does the lack of top-level await matter?

To understand the impact on the ecosystem, I wrote a few scripts to analyze the high-impact packages from [wooorm/npm-esm-vs-cjs]

```
12    const failures = [];
13    const passed = [];
14    for (let i = 0; i < packages.length; ++i) {
15      const p = packages[i];
16      try {
17        require(p);
18        passed.push(p)
19      } catch(e) {
20        failures.push({p, e});
21      }
22    }
23
```

https://github.com/joyeecheung/test-require-esm

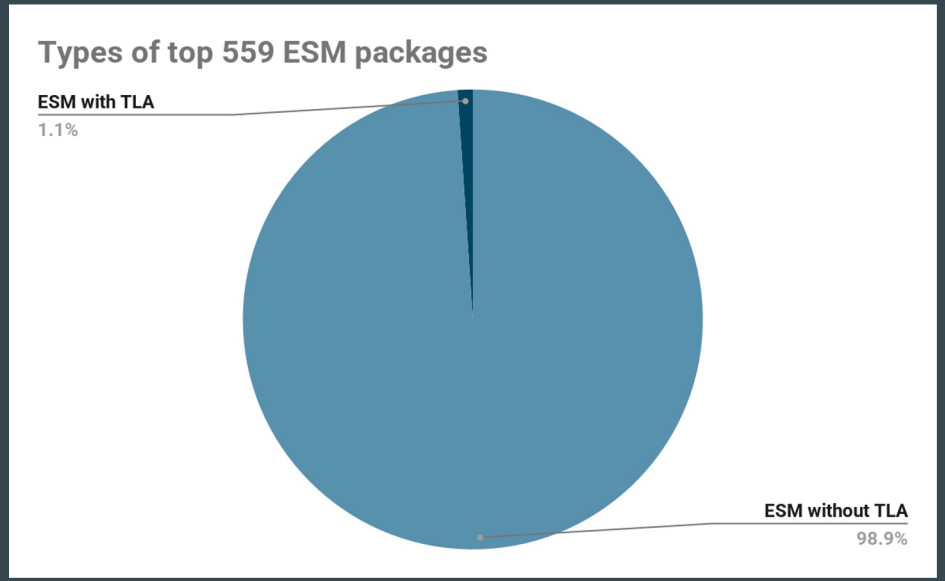
Does the lack of top-level await matter?



Out of the top 5000 high-impact packages on npm (Sept 2024)

466 dual ESM and 526 faux ESM packages already don't use top-level await and can drop CJS distribution without breaking compatibility

Does the lack of top-level await matter?



Breaking down the top 559 ESM-only packages from top 5000

- Only 6 with top-level await
- 3 were converted from `fs.somethingSync()` to `await fs.something()`
- 2 can use `process.getBuiltinModule('node:something')` to avoid using TLA for feature detection
- Only 1 might really need TLA (minified, can't tell)

Does the lack of top-level await matter?

- Top-level await is mostly intended for entry points and scripts
- It's actually rare in packages meant to be loaded by a different code base
- require() works for >99% of the high-impact packages. For the <1%, use dynamic import().
  - require(esm) do not break the usual way of loading these high-impact packages

A bunch of small features to smooth the transition..

- Let’s check out some representatives for:
  - Faux ESM -> ESM
  - CJS -> ESM
  - Dual -> ESM

Faux ESM to native ESM transition: default exports handling

Unlike CJS, ESM makes the default export a property named “default” on the module namespace object, parallel to other named exports

```cpp
// CJS: Logger.log is log
module.exports = class Logger{};
exports.log = function log() {};

const Logger = require('log');
Logger.log; // log

// Logger
console.log(require('log'));
```

```cpp
// ESM: Logger and log are separate
export default class Logger {};
export function log() { }

import Logger from 'log';
Logger.log; // undefined

// { default: Logger, log: log }
console.log(await import('log'));
```

Faux ESM to native ESM transition: default exports handling

Unlike CJS, ESM makes the default export a property named “default” on the module namespace object, parallel to other named exports

```cpp
// CJS: Logger.log is log
module.exports = class Logger{};
exports.log = function log() {};

const Logger = require('log');
Logger.log; // log

// Logger
console.log(require('log'));
```

```cpp
// ESM: Logger and log are separate
export default class Logger {};
export function log() { }

import Logger from 'log';
Logger.log; // undefined

// { default: Logger, log: log }
console.log(await import('log'));
```

Faux ESM to native ESM transition: default exports handling

Bundlers and transpilers have already developed the __esModule marker to work around the multiplexing

```
// Original ESM module code
export default class Logger{};
export function log() { }.
```

```
// Transpiled faux ESM module code
exports.default = class Logger{};
exports.log = function log() {}
exports.__esModule = true
```

```
// Original ESM consumer code
import Logger from 'log';
const logger = new Logger;
```

```
// Transpiled faux ESM consumer code
const _mod = require('log');
//{ default: Logger, log: log, __esModule: true }
const Logger = _mod.__esModule ? _mod.default : _mod;
const logger = new Logger;
```

Faux ESM to native ESM transition: default exports handling

When a faux ESM package is converted to native ESM, but consumer code is still transpiled, faux-ESM -> native ESM can be a breaking change if default exports are used

```
// Now directly shipped as ESM
export default class Logger{};
export function log() { }.
```

```
// Original ESM consumer code
import Logger from 'log';
const logger = new Logger;
```

```
// Transpiled faux ESM consumer code
const _mod = require('log');
// _mod looks like { default: Logger, log: log }
const Logger = _mod.__esModule ? _mod.default : _mod;
const logger = new Logger;  // ✗ Logger is undefined!
```

Faux ESM to native ESM transition: default exports handling

Solution 🧪 Node.js adopts the bundler convention and add __esModule, so that transpiled code recognize default exports in native ESM loaded by require()

```
// Now directly shipped as ESM
export default class Logger{};
export function log() { }.
```

```
// Original ESM consumer code
import Logger from 'log';
const logger = new Logger;
```

```
// Transpiled faux ESM consumer code
const _mod = require('log');
// { default: Logger, log: log, __esModule: true }
const Logger = _mod.__esModule ? _mod.default : _mod;
const logger = new Logger;  // ✅Logger is unwrapped now
```

Faux ESM to native ESM transition: default exports handling

- However...ESM namespace is not mutable - cannot just add a new __esModule property!
- Multiple ways to implement this, brainstormed with folks from different projects...
  - `Object.create(namespace, { __esModule: true })`
  - Copy over property descriptors to a new object and add __esModule
  - A proxy backed by the namespace that intercepts __esModule
  - A SourceTextModule that re-exports * from original module and also exports __esModule

```
export * from 'original';
export { default } from 'original';
export const __esModule = true;
```

Faux ESM to native ESM transition: default exports handling

Performance impact on module loading are all minimal, but impact on export access vary greatly

```
Benchmark 1: ./node_main --experimental-require-module ../test-require-esm/load.cjs
  Time (mean ± σ):     674.4 ms ±  12.6 ms    [User: 754.7 ms, System: 128.4 ms]
  Range (min … max):   657.8 ms … 693.7 ms    10 runs

Benchmark 2: ./node_proto --experimental-require-module ../test-require-esm/load.cjs
  Time (mean ± σ):     685.3 ms ±  21.8 ms    [User: 773.4 ms, System: 129.5 ms]
  Range (min … max):   661.6 ms … 729.1 ms    10 runs

Benchmark 3: ./node_desc --experimental-require-module ../test-require-esm/load.cjs
  Time (mean ± σ):     683.9 ms ±  11.9 ms    [User: 781.1 ms, System: 119.2 ms]
  Range (min … max):   665.1 ms … 698.9 ms    10 runs

Benchmark 4: ./node_stm --experimental-require-module ../test-require-esm/load.cjs
  Time (mean ± σ):     683.7 ms ±  11.5 ms    [User: 779.8 ms, System: 116.8 ms]
  Range (min … max):   669.1 ms … 705.8 ms    10 runs

Benchmark 5: ./node_proxy  --experimental-require-module ../test-require-esm/load.cjs
  Time (mean ± σ):     671.3 ms ±  10.3 ms    [User: 745.8 ms, System: 131.7 ms]
  Range (min … max):   656.1 ms … 684.9 ms    10 runs
```

Faux ESM to native ESM transition: default exports handling

```
$ node-benchmark-compare esm-proto.csv
                                                                confidence improvement accuracy (*)   (**)  (***)
esm/require-esm.js n=1000 exports='default' type='access'          ***     -7.46 %       ±1.41% ±1.87% ±2.44%
esm/require-esm.js n=1000 exports='default' type='all'             ***     -5.49 %       ±1.51% ±2.01% ±2.62%
esm/require-esm.js n=1000 exports='default' type='load'            ***     -5.44 %       ±1.40% ±1.87% ±2.43%
esm/require-esm.js n=1000 exports='named' type='access'            ***    -13.59 %       ±1.49% ±1.99% ±2.58%
esm/require-esm.js n=1000 exports='named' type='all'               ***     -6.73 %       ±1.59% ±2.11% ±2.75%
esm/require-esm.js n=1000 exports='named' type='load'              ***     -6.82 %       ±1.55% ±2.06% ±2.68%
```

```
$ node-benchmark-compare esm-proxy.csv
                                                                confidence improvement accuracy (*)   (**)  (***)
esm/require-esm.js n=1000 exports='default' type='access'          ***    -79.45 %       ±0.84% ±1.13% ±1.50%
esm/require-esm.js n=1000 exports='default' type='all'             ***     -3.00 %       ±1.45% ±1.93% ±2.52%
esm/require-esm.js n=1000 exports='default' type='load'            ***     -2.18 %       ±1.17% ±1.55% ±2.03%
esm/require-esm.js n=1000 exports='named' type='access'            ***    -84.10 %       ±1.14% ±1.54% ±2.05%
esm/require-esm.js n=1000 exports='named' type='all'               ***     -3.49 %       ±1.49% ±1.98% ±2.58%
esm/require-esm.js n=1000 exports='named' type='load'              ***     -3.81 %       ±1.26% ±1.68% ±2.19%
```

```
$ node-benchmark-compare esm-desc.csv
                                                                confidence improvement accuracy (*)   (**)  (***)
esm/require-esm.js n=1000 exports='default' type='access'          ***    -76.63 %       ±0.83% ±1.11% ±1.46%
esm/require-esm.js n=1000 exports='default' type='all'             ***     -8.88 %       ±1.03% ±1.37% ±1.78%
esm/require-esm.js n=1000 exports='default' type='load'            ***     -7.77 %       ±1.37% ±1.83% ±2.38%
esm/require-esm.js n=1000 exports='named' type='access'            ***    -81.16 %       ±1.15% ±1.55% ±2.06%
esm/require-esm.js n=1000 exports='named' type='all'               ***    -11.14 %       ±1.26% ±1.67% ±2.17%
esm/require-esm.js n=1000 exports='named' type='load'              ***     -9.58 %       ±1.46% ±1.94% ±2.53%
```

```
$ node-benchmark-compare esm-stm.csv
                                                                confidence improvement accuracy (*)   (**)  (***)
esm/require-esm.js n=1000 exports='default' type='access'          ***    -16.69 %       ±2.27% ±3.02% ±3.94%
esm/require-esm.js n=1000 exports='default' type='all'             ***    -21.00 %       ±1.07% ±1.43% ±1.86%
esm/require-esm.js n=1000 exports='default' type='load'            ***    -21.62 %       ±1.16% ±1.54% ±2.02%
esm/require-esm.js n=1000 exports='named' type='access'            ***    -14.63 %       ±1.42% ±1.90% ±2.48%
esm/require-esm.js n=1000 exports='named' type='all'               ***    -22.57 %       ±1.15% ±1.53% ±2.00%
esm/require-esm.js n=1000 exports='named' type='load'              ***    -23.04 %       ±1.13% ±1.50% ±1.95%
```

- Suggested by Bun
- Breaks enumerability of the returned objects

- Exported names are still enumerable
- Looks very similar to the original namespace

Faux ESM to native ESM transition: default exports handling

Optimizing SourceTextModule facade approach

- Caching facade module compilation – module record is constant, just needs relinking
- Only add it when the original module contains default export
- Micro-optimizations

```
                                                          confidence improvement accuracy (*)   (**)   (***)
esm/require-esm.js n=1000 exports='default' type='access'        ***      8.57 %       ±2.44% ±3.26% ±4.20%
esm/require-esm.js n=1000 exports='default' type='all'           ***     -7.40 %       ±0.71% ±0.95% ±1.25%
esm/require-esm.js n=1000 exports='default' type='load'          ***     -8.81 %       ±0.61% ±0.81% ±1.05%
esm/require-esm.js n=1000 exports='named' type='access'          ***     10.39 %       ±1.75% ±2.33% ±3.03%
esm/require-esm.js n=1000 exports='named' type='all'             ***     -2.36 %       ±0.52% ±0.69% ±0.90%
esm/require-esm.js n=1000 exports='named' type='load'            ***     -2.57 %       ±0.45% ±0.60% ±0.78%
```

CJS -> ESM transition: default exports handling (again)

The default exports multiplexing problem happens again to packages that are originally authored in CJS, and want to migrate to ESM

```
// When the package is provided through CJS
module.exports = class Logger {};
module.exports.log = function log() {}

// ESM user gets..
import { log } from 'log';
import Logger from 'log’;

// CJS user gets..
const { log } = require(‘log');
const Logger = require(‘log');
```

CJS -> ESM transition: default exports handling (again)

The default exports multiplexing problem happens again to packages that are originally authored in CJS, and want to migrate to ESM

```
// If the package migrates to ESM..
export default class Logger{};
export function log() { }
Logger.log = log;

// ESM user gets..
import { log } from 'log';
import Logger from 'log';

// In ESM, default export is placed separately from named exports 🤔
// CJS user gets..
const { log } = require('log');
const Logger = require('log'); // ❌ Oops, it's now { default: Logger, log: log }!
const Logger = require('log').default; // Have to unwrap it from .default..
```

CJS -> ESM transition: default exports handling (again)

- Not a problem if module doesn't have default exports
  - ~36% of the high-impact ESM packages have only a default export
  - ~16% have both named and default exports
  - ~48% have no default exports
- When they do, Node.js needs a hint from package authors to customize what should be returned.
- Can't unwrap default based on __esModule because existing faux-ESM code would need that to be left to them.

CJS -> ESM transition: default exports handling (again)

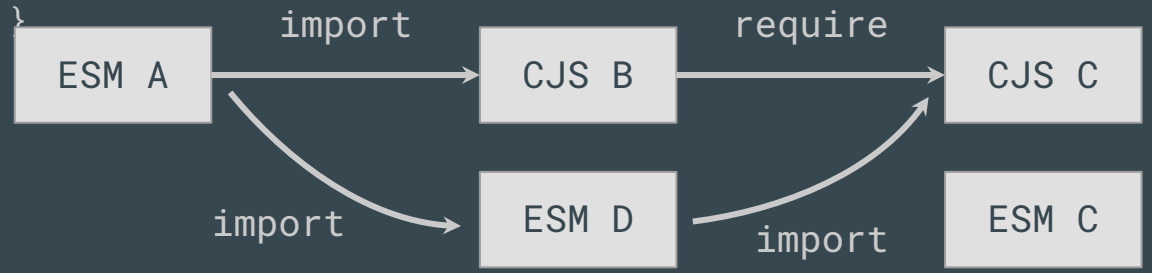
Solution 🧪 Use another marker, "`module.exports`", which will be written by human instead of being generated

```
// Migrate to ESM
export default class Logger{};
export function log() { }
Logger.log = log;
export { Logger as 'module.exports' }; // Customize for require(esm) in  Node.js

// CJS user gets the same as before
const { log } = require('log');
const Logger = require('log');
```

Dual -> ESM transition: prioritize ESM on newer Node.js version

Common shipping pattern for dual packages: CJS-first on Node.js, ESM in other environments

```
{
  "type": "module",
  "exports": {
    ".": {
      // On Node.js, provide a CJS version of the package transpiled from the original
      // ESM version
      "node": "./dist/index.cjs",
      // On any other environment, use the ESM version.
      "default": "./index.js"
    }
  }
}
```



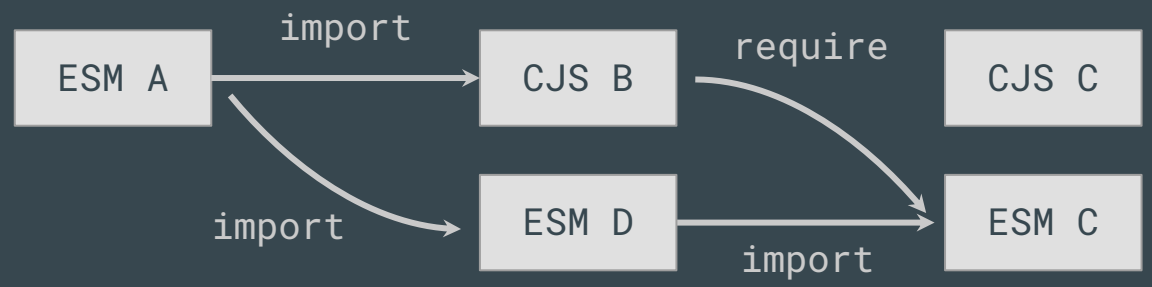
Always use the CJS version on Node.js, ignore the ESM one

Dual -> ESM transition: prioritize ESM on newer Node.js version

require(esm) allows dual packages to go ESM-only and reduce the duplication.

```
{
  "type": "module",
  "exports": {
    // Always use the ESM version.
    ".": "./index.js"
  }
}
```

🤔 What if they still want to keep the CJS distribution for older versions of Node.js for some time?



Now they can always use the ESM version on Node.js!

Dual -> ESM transition: prioritize ESM on newer Node.js version

Bundlers already have a convention “module” for require() to pick up ESM, which they use to transpile and produce cleaner code. Can we reuse it to avoid “one more condition”? 🤔

```
{
  "type": "module",
  "exports": {
    ".": {
      "node": {
        // When the package is bundled, bundlers will pick up “module”, which contains
        // original ESM code, for both import and require() to produce cleaner code.
        "module": "./index.js",
        // On older versions of Node.js, use the transpiled CJS
        "default": "./dist/index.cjs"
      },
      // On any other environment, use the ESM version.
      "default": "./index.js”
    }
  }
}
```

Dual -> ESM transition: prioritize ESM on newer Node.js version

- Implemented it, tested it on high impact packages...unfortunately, bundlers also have resolution rules that differ from Node.js ESM for ESM bundles
- Existing high-impact packages using the “module” condition (including many high-impact packages) are also expecting these non-Node.js resolution rules to work in their ESM code

```
{
  "type": "module",
  "exports": {
    ".": {
      "module": "./dist-es/index.js",
    }
  }
}
```

Breaks @aws-sdk/core, @sentry/core, etc. 😬

```
// @aws-sdk/core/dist-es/index.js
export * from "./submodules/client/index"; // Only supported by bundlers
export * from "./submodules/httpAuthSchemes/index";
export * from "./submodules/protocols/index";
```

Dual -> ESM transition: prioritize ESM on newer Node.js version

Solution 🧪 Adding one more “module-sync” condition for dual packages that still need to support EOL Node.js versions (temporarily, hopefully)

```
"node": {
  // On new version of Node.js, both require() and import get the ESM version
  "module-sync": "./index.js",
  // Supply ESM to bundlers for better generated code
  "module": "./index.js",
  // On older version of Node.js, where "module" and require(esm) are not supported,
  // use the transpiled CJS version to avoid dual-module hazard.
  "default": "./index.js”
},
// On any other environment, use the ESM version.
"default": "./index.js"
```

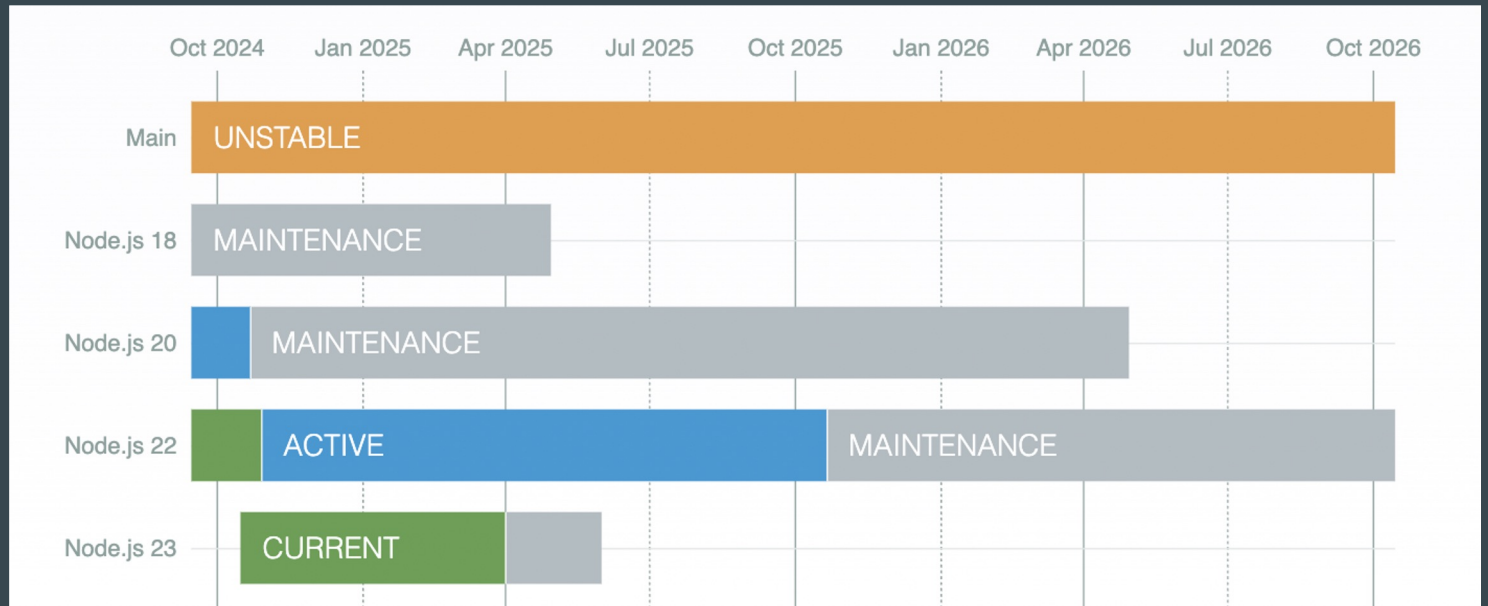
Dual -> ESM transition: prioritize ESM on newer Node.js version

Package authors can drop all the conditions when require(esm) is available on all Node.js versions they support

```
{
  "type": "module",
  // When the package no longer supports Node.js versions without require(esm),
  // just bump major version and get rid of the conditions.
  "exports": {
      ".": "./index.js”
  }
}
```

Backporting

- Developed on the main branch during v22-v23, unflagged in v23
- Backporting to v22 was relatively easy
- Backporting to v20 was..challenging



Backporting

- Tried “backporting all related module loader commits” to v20: 119 in total, some semver-major ones difficult to be made non-breaking 🥲
- “Only backport the essential commits”: took some time to triage 33 out of 119 ✅
- Skipping commits means a lot of modification needs to be made..
- Wrote a script to “diff the diffs” to make sure commits adapted to v20.x do not have unintentional behavior differences and I did not leave out important bits

https://gist.github.com/joyeecheung/7889b89265dc66a6889f7f7167efb89f

Backporting

diffs > 0002-module-detect-ESM-syntax-by-trying-to-recompile-as-SourceTextModule....

```
142  *** 187,205 ****
143
144          if (that->SetPrivate(context,
145                               realm->isolate_data()->host_defined_option_symbol(),
146  !         module = Module::CreateSyntheticModule(isolate, url, export_names,
147  !           SyntheticModuleEvaluationStepsCallback);
148          } else {
149  -         ScriptCompiler::CachedData* cached_data = nullptr;
150  +         // When we are compiling for the default loader, this will be
151  +         // std::nullopt, and CompileSourceTextModule() should use
152  ! +       // on-disk cache (not present on v20.x).
153  +         std::optional<v8::ScriptCompiler::CachedData*> user_cached_data;
154  +         if (id_symbol !=
155  +             realm->isolate_data()->source_text_module_default_hdo()) {
156  +           user_cached_data = nullptr;
157  +         }
158            if (args[5]->IsArrayBufferView()) {
159  +           CHECK(!can_use_builtin_cache);  // We don't use this option internally.
160              Local<ArrayBufferView> cached_data_buf = args[5].As<ArrayBufferView>();
161              uint8_t* data =
162  --- 173,194 ----
163
164          if (that->SetPrivate(context,
165                               realm->isolate_data()->host_defined_option_symbol(),
166  !         module = Module::CreateSyntheticModule(
167  !             isolate, url, span, SyntheticModuleEvaluationStepsCallback);
168          } else {
169  -         ScriptCompiler::CachedData* cached_data = nullptr;
```

summary.md

1 - https://github.com/nodejs/node/pull/52093: Adapted to the lack of reader rewrite in v20.x.

2 - https://github.com/nodejs/node/pull/52413: Adapted to the absence cache support in the C++ layer, borrowing some lines from https://gi nodejs/node/pull/52535

3 - https://github.com/nodejs/node/pull/52058: Do not freeze `module. dependencySpecifiers` because it's semver-major. Added a `FromV8Arra polyfill for v20.x which does not have the new V8 API.

4 - https://github.com/nodejs/node/pull/52047: `pkg?.data.type` -> `pk because we are not backporting package reader rewriting

5 - https://github.com/nodejs/node/pull/52868: no modifications

6 - https://github.com/nodejs/node/pull/53050: no modifications

7 - https://github.com/nodejs/node/pull/51711: Changed location of `co in test harness

8 - https://github.com/nodejs/node/pull/52658: Adapted to the lack of naming changes

9 - https://github.com/nodejs/node/pull/53573: No need to take care of cache since we are not backporting it to v20

10 - https://github.com/nodejs/node/pull/52166: Adapted to the lack of naming changes. Also v8::ScriptOrigin takes an isolate on v20.

11 - https://github.com/nodejs/node/pull/53872: no modifications

12 - https://github.com/nodejs/node/pull/53619: process.env.TEST_PARALL in tests is not backported to v20. Doesn't hurt to check it though s the python test runner in v20.

13 - https://github.com/nodejs/node/pull/54045: Work around absense of tripping, taking a bit of the typeless package.json warning helper c https://github.com/nodejs/node/pull/53725

14 - https://github.com/nodejs/node/pull/54868: Removed TypeScript test not on v20.x

Status of require(esm)

- Release candidate
- Available without flags in v24, v22, v20 (backported)
- Stabilization soon after it's more battle tested
- Many popular packages have started shipping/planning to ship ESM-only targeting v20.x and above after v18.x EOL in April 2025
  - vite
  - babel
  - yargs
  - graphql
  - various eslint plugins..
  - unjs packages..
  - various tinylibs..

What's next for ESM in Node.js?

- Consolidate the internal loader paths to eliminate races from async linking vs synchronous linking
  - Mostly fixed for user land, theoretically still possible
- Stabilize & complete in-thread synchronous loader hooks
  - Reduce (widespread) ecosystem dependency on CJS loader internals/monkey-patching of the CJS loader itself
  - Improve instrumentation
- Improve ESM performance
  - Even require(esm) was 1.2x faster than import esm.
  - CJS loading can be 2-2.5x faster than ESM
  - https://gist.github.com/bengl/c6d3beae27be0d76bce8312881ae8f2d

Thanks

•••