require(esm) in Node.js

Joyee Cheung

About me

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

Have you seen this error?

```
Error [ERR_REQUIRE_ESM]: require() of ES Module ...
not supported.
```

It’s now gone in v23

```
// hello.mjs
export const hello = 'world';

// index.cjs
// No more ERR_REQUIRE_ESM!
console.log(require('./hello.mjs').hello);  // world
```

Behind `--experimental-require-module` in v22 & v20, may also get unflagged in semver-minor releases soon

History of ESM/CJS interop in Node.js

- In experiment since v8.5.0 – mid 2017
- Long period of development, collaborated by a big group of community members
- Unflagged since v13.2.0 (and backported to v12) - late 2019
- Stable since v15.3.0 (and backported to v12, v14) - late 2020

History of ESM/CJS interop in Node.js

At the time of stabilization (v15.3.0):

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

// main1.mjs
import Logger from './logger.js'; // => module.exports
import { log } from './logger.js'; // Detected with static analysis
```

History of ESM/CJS interop in Node.js

At the time of stabilization (v15.3.0):

```
// main2.mjs
require('./logger.js'); // ReferenceError: require is not defined

// main3.mjs
import module from 'node:module';
const require = module.createRequire(import.meta.url);
require('./logger.js');

// main4.mjs
import Logger from './logger'; // Throws: requires extension.
await import('./logger.js'); // Top-level await works
```

History of ESM/CJS interop in Node.js

At the time of stabilization (v15.3.0):

```
// logger.mjs
export default function log() {}

// main1.cjs
require('./logger.mjs'); // Throws ERR_REQUIRE_ESM

// main2.cjs
// Works but returns a promise and only works in async code
import('./logger.mjs');
```

Implications of lack of require(esm)

- The lack of synchronous require(esm) turns out to be a major obstacle in the CJS -> ESM transition
- If a package ships only ESM, it would break CJS users
- Users have priorities and want to migrate at their own pace
- In the open source world, many maintainers are too kind to break their CJS users, so they continue to ship CJS to maximize reach

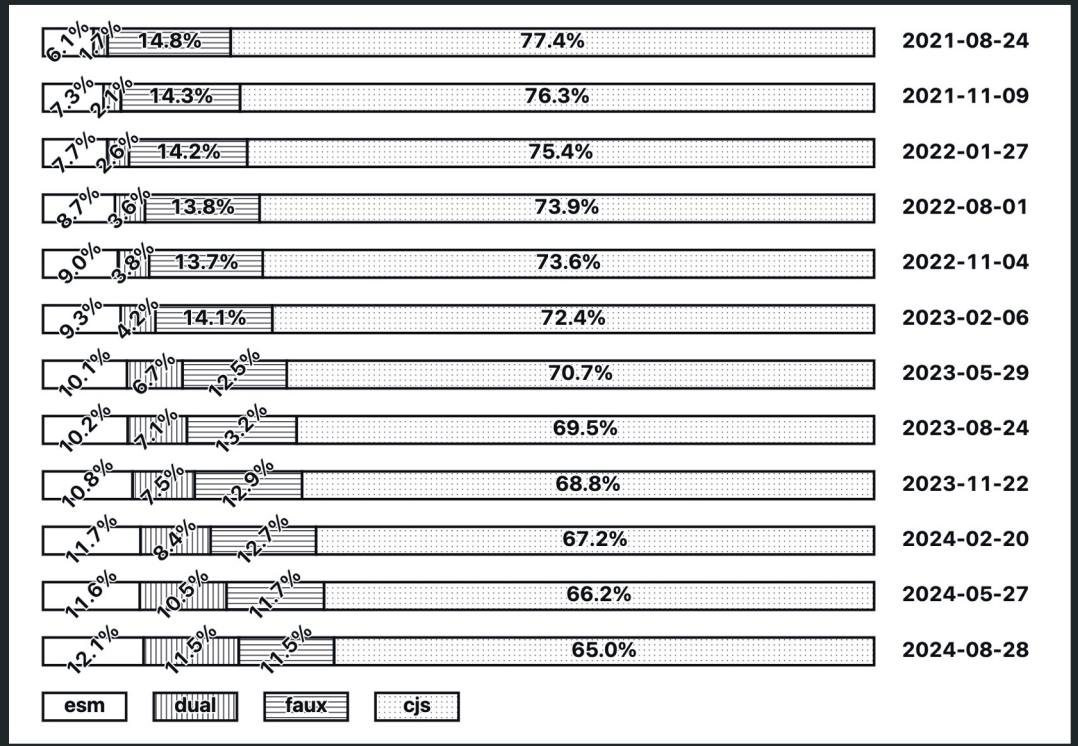
Implications of lack of require(esm)

- For framework/library maintainers:
  - **Ship ESM**: CJS users get broken, reduced reach, hurts popularity
  - **Ship CJS**: cannot support user configs/user exports in ESM, users get stuck too (if they transpile user ESM automatically, magical ERR_REQUIRE_ESM occurs for external ESM dependencies)
  - **Ship CJS+ESM**: difficult to get right, bloat node_modules with two copies, overhead from build steps

Implications of lack of require(esm)

- For end users:
  - Writing ESM != running ESM, the tools or framework they use might be transpiling it to CJS for them magically for compatibility
  - Seeing ERR_REQUIRE_ESM for loading external ESM in magically transpiled ESM code is very confusing
  - Those *running* CJS cannot upgrade dependencies that have gone ESM-only, being forced to refactoring to ESM disrupts plans, or they end up forking or getting stuck with versions that are no longer maintained
- Ripple effect: the more dependencies stick to CJS, the more users and transitive dependencies also stick to CJS

Implications of lack of require(esm)



Dual: ships CJS + ESM

Faux: ships CJS (transpiled from ESM)

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

Implications of lack of require(esm)

- Many maintainers mentioned that lack of require(esm) was the main blocker for them to ship ESM
- Maintainers should not suffer from reduced reach for adopting a standard
- Neither should they suffer from build step woes for trying to be nice and not break CommonJS users
- require(esm) is needed for ESM to be successful in Node.js

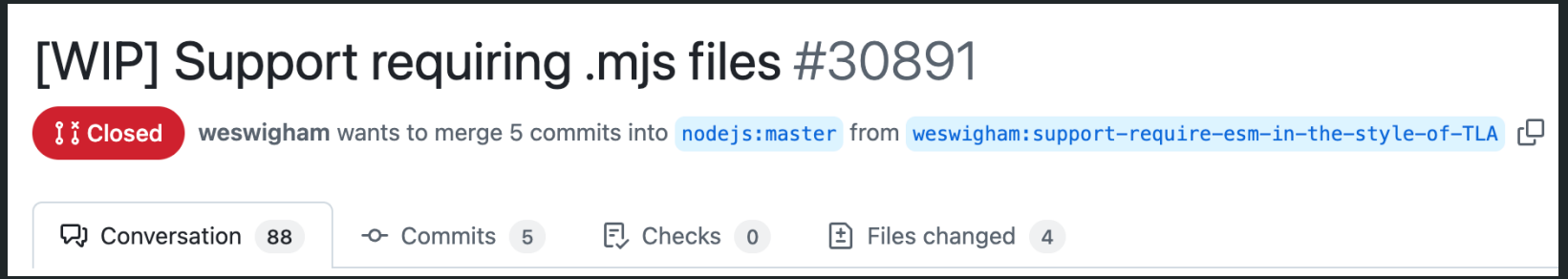
require(esm) in Node.js

Outside Node.js core…

- Bundlers have supported require(esm) with automatic transpilation.
- Popular user-land require() customizations exist: standard-things/esm
- As the statistics show before, non-buitin solutions were not enough for the ecosystem to migrate

require(esm) in Node.js

- Archaeology from the perspective of someone who wasn't involved in the initial ESM development in Node.js…(take it with a grain of salt)
- Early attempts date back to 2019
  - Tried to make top-level await work from require(esm), which was technically unsafe due to libuv event loop requirements
  - There were suggestions about excluding top-level await, but it didn't come to fruition.



require(esm) in Node.js

- Somehow a built-in solution just didn’t happen
- Archaeology:
  - No consensus on yes or no on general idea that I could find, only scepticism about details but not enough effort as made to investigate/disprove them
  - ESM in Node.js was developed by a very diverse community group: with diverse opinions, you also get diverse disagreements on ever different detail, it takes a lot of work to come out with a solution that all parties can be happy with

require(esm) in Node.js

- Users have been requesting it, but spiritual support is only the first 1%.
- Node.js is a project collectively built by the community, not a product provided by a company.
- Some contributor, or multiple of them, need to somehow obtain the time, energy, expertise, funding…to kick it off and push it forward.
  - Otherwise, work doesn't get done by itself.
  - Especially when we are talking about 2020 – 2022 when many people were preoccupied with something else…

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

- Not that many people knew “it can be done”
- 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. Instead, use `import()` to load an ES module from a CommonJS module.

Breaking the myth

- Was reading ESM implementation in V8 to fix some memory leak
- Noticed that V8 maintains separate paths for asynchronous and synchronous module evaluation
- JS engines simply implement what the ECMAScript specification says for complex runtime behaviors…which means this may come from the specification? 🤔
- Is the Node.js documentation exaggerating things?

ESM without top-level await is synchronous

Archeology shows that it was codified in the top-level await proposal in 2019

8. Let *result* be Completion(InnerModuleEvaluation(*module*, *stack*, 0)).
9. If *result* is an abrupt completion, then
    a. For each Cyclic Module Record *m* of *stack*, do
        i. Assert: *m*.[[Status]] is EVALUATING.
        ii. Set *m*.[[Status]] to EVALUATED.
        iii. Set *m*.[[EvaluationError]] to *result*.
    b. Assert: *module*.[[Status]] is EVALUATED.
    c. Assert: *module*.[[EvaluationError]] and *result* are the same Completion Record.
    d. Perform ! Call(*capability*.[[Reject]], **undefined**, « *result*.[[Value]] »).
10. Else,
    a. Assert: *module*.[[Status]] is either EVALUATING-ASYNC or EVALUATED.
    b. Assert: *module*.[[EvaluationError]] is EMPTY.
    c. If *module*.[[AsyncEvaluation]] is **false**, then
        i. Assert: *module*.[[Status]] is EVALUATED.
        ii. Perform ! Call(*capability*.[[Resolve]], **undefined**, « **undefined** »).
    d. Assert: *stack* is empty.

V8 basically implements these step-by-step

https://tc39.es/ecma262/#sec-moduleevaluation

ESM without top-level await is synchronous

12. If *module*.[[PendingAsyncDependencies]] > 0 or *module*.[[HasTLA]] is **true**, then
    a. Assert: *module*.[[AsyncEvaluation]] is **false** and was never previously set to **true**.
    b. Set *module*.[[AsyncEvaluation]] to **true**.
    c. NOTE: The order in which module records have their [[AsyncEvaluation]] fields transition to **true** is significant. (See 16.2.1.5.3.4.)
    d. If *module*.[[PendingAsyncDependencies]] = 0, perform ExecuteAsyncModule(*module*).
13. Else,
    a. Perform ? *module*.ExecuteModule().

https://tc39.es/ecma262/#sec-innermoduleevaluation

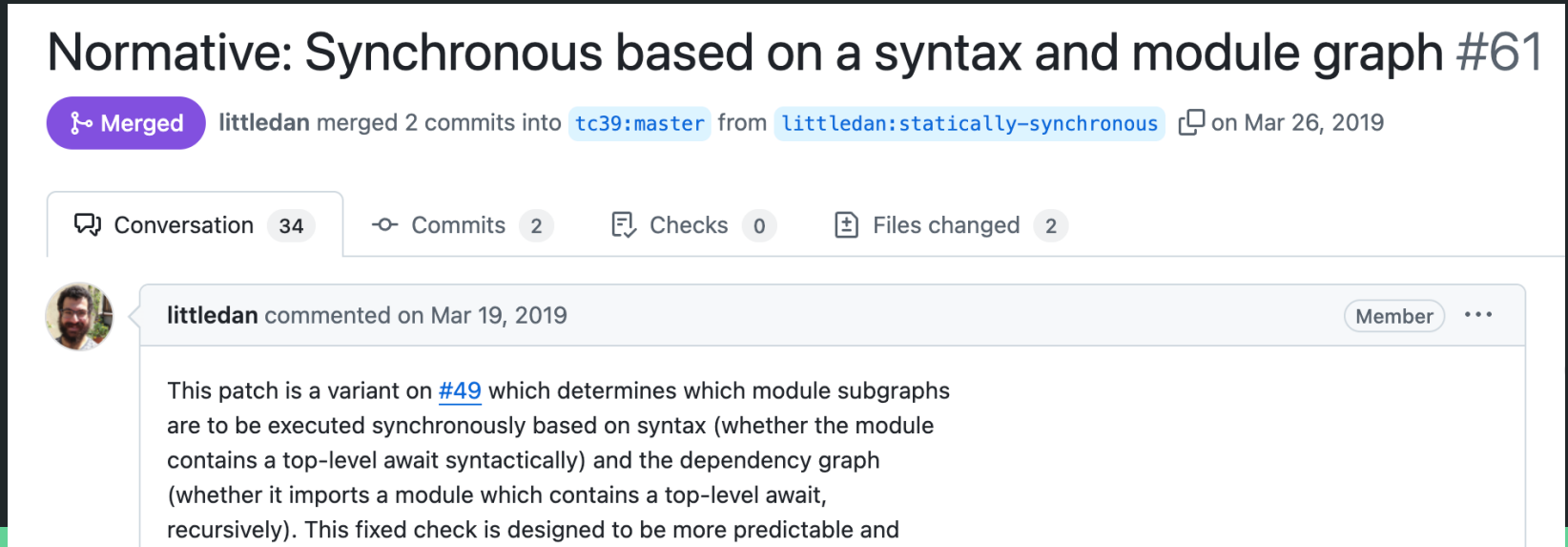
ESM without top-level await is synchronous

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

- Threaded together, this means if a module and its dependencies are free from top-level await, there is nothing that force it to finish asynchronously, and it should resolve a Promise to undefined synchronously.
- Confirmed later that this was intentional, also relied on by bundlers



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();
}
```

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 in the peusdo code before - it was already added for Chrome to implement similar semantics for ServiceWorkers in 2020
- These had been brought up in early attempts of require(esm) in Node.js back in 2019, but somehow the work didn't happen

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

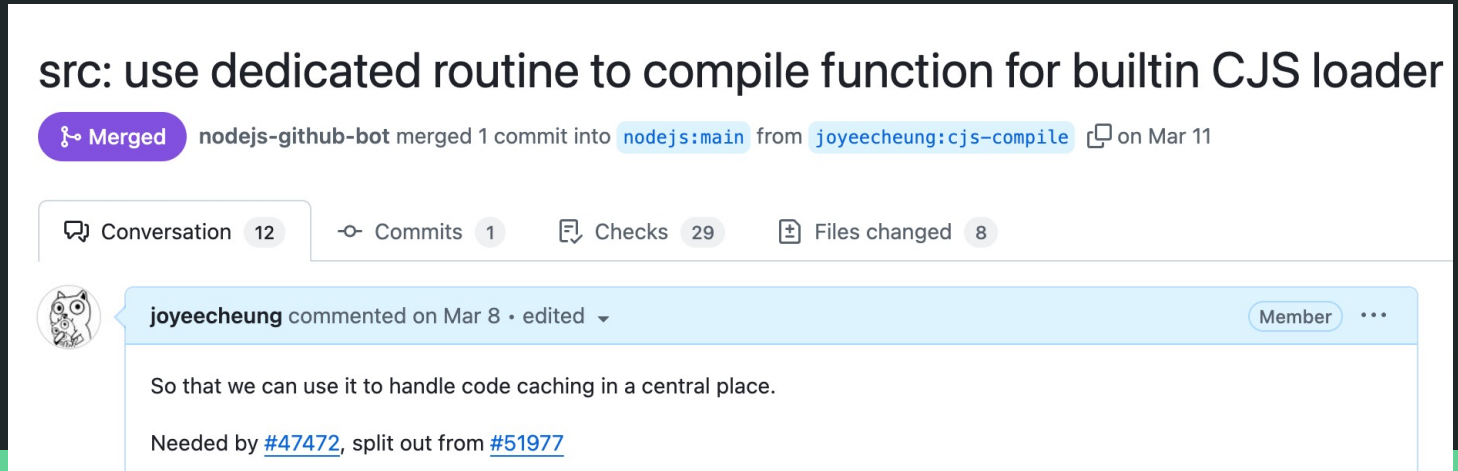
- In late 2023, I learned about the semantics when reading V8 code, discussed with other contributors who knew more about ESM in Node.js

```
// 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 JS 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 and carve a synchronous path…
- Earlier this year, working on compile cache, ended up refactoring a small part of the ESM loader code to make the compilation go through the cache, then ended up reading the whole thing…



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

- 1 afternoon to brain dump

```
lib/internal/modules/cjs/loader.js      | 114 ++++++++++++++++++++++-----
lib/internal/modules/esm/loader.js      | 136 ++++++++++++++++++++++++++++--
lib/internal/modules/esm/module_job.js  |  67 ++++++++++---
lib/internal/modules/esm/module_map.js  |   4 +-
lib/internal/modules/esm/translators.js |  32 ++++--
5 files changed, 305 insertions(+), 48 deletions(-)
```

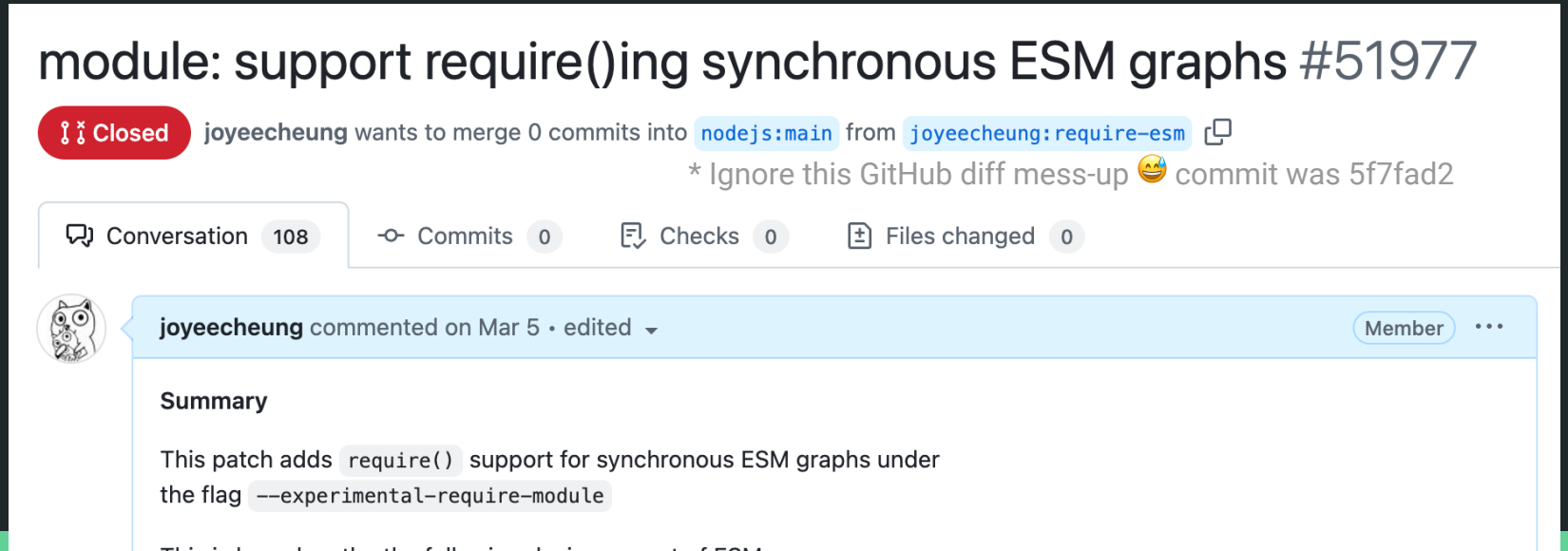
```
❯ cat test.js
const mod = require('./test/fixtures/es-module-loaders/module-named-exports.mjs');
console.log(mod);
❯ out/Release/node --experimental-require-module test.js
[Module: null prototype] { bar: 'bar', foo: 'foo' }
```

No more ERR_REQUIRE_ESM 🤩

- Benchmarked out of curiosity, turned out to be ~1.2x faster than the asynchronous ESM loader, because async handling in Node.js adds a lot of overhead

Restarting require(esm) in Node.js

- Reaction was very positive, no contributor said it shouldn't happen
- Some edges neeedd 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)
- Released to v22, backported to v20 behind --experimental-require-module flag



Does the lack of top-level await matter?

- Still fine to use them in ESM entry points, or if they are import()-ed
- Just unsupported in modules that are meant to be shared with other people who may load it with require()
- Top-level await is actually rare in packages

Does the lack of top-level await matter?

- Out of the top 5000 high-impact packages on npm, 500+ are ESM-only
  - The other 1000+ dual and faux ESM packages already don't use top-level await
  - Only 6 out of the 500+ ESM-only packages uses top-level await
  - 3 are `await fs.something()`, which can just be easily changed to `fs.somethingSync()` (and they were doing that before being coverted to ESM)
  - 2 are `try { await import('node:something') }`, in case they are not run on Node.js. We introduced `process.getBuiltinModule('node:something')` to fill the gap.
- Making it work for >99% of the packages should be enough

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

Restarting require(esm) in Node.js

>6 months collaborating with other contributors and pacakge maintainers to:

- Fix bugs in many many edge cases
- Work out more features to improve interoperability/fill gaps
- Test the ecosystem and try not to break existing code
  - 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
  - If it's semver-minor, it can be backported to 22 or even 20, so package authors can fully rely on it and start the transition around 2025 or 2026

Restarting require(esm) in Node.js

>6 months collaborating with other contributors and pacakge maintainers to:

- Try to make the dependency of bundlers go away transparently
  - If the code needed bundlers to run, it should be free of that dependency after require(esm) is enabled
  - Try to adopt existing patterns instead of asking people to modify their code...unless that would break more people, or too complex that it will delay the timeline a lot
- While keeping it reasonably performant
  - At least not too slow compared to CJS to demotivate transition

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

```cjs
// 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'));
```

```esm
// 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
- Bundlers and transpilers has 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 user code
import Logger from 'log’;
const logger = new Logger;
```

```
// Transpiled faux ESM user code
const _mod = require('log’);
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 using transpiler, if the namespace is returned as-is, they won't work together, making faux-ESM -> native ESM a breaking change if default exports are used

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

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

```
// Transpiled faux ESM user 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

Adopt the bundler convention and add __esModule when default exports are present, 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 user code
import Logger from 'log';
const logger = new Logger;
```

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

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

```
// CommonJS module code
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

```
// Migrate 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
- But when they do, Node.js needs a hint from package authors to customize what `require(esm)` should return.
  - Cannot just unwrap default export without hint, in case named exports get lost.
  - Cannot unwrap by __esModule, because that would break existing faux-ESM consumer that expect require() to return faux-ESM namespace as-is.
- 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

- require(esm) allows dual packages to go ESM-only and reduce the duplication. But it takes time for Node.js changes to roll out to all active LTS, which packages usually support.
- What if they want to be ESM-first on newer versions of Node.js?
- 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, so that both the ESM and the CJS consumers in the same graph get
    // the same version to avoid the having two versions of the same package
    // conflicting with each other a.k.a. package hazard.
    "node": "./dist/index.cjs”,
    // On any other environment, use the ESM version.
    "default": "./index.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 will transpile and produce cleaner code.

```
{
  "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 Node.js, where "module" and require(esm) were not supported,
      // use the transpiled CJS version to avoid dual-package hazard
      "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

- 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 using these non-Node.js resolution rules in their ESM code

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

Implemented “module-sync” for dual packages that want to take advantage of require(esm) earlier while there are still Node.js LTS release lines that don’t support require(esm)

```
{
  "type": "module",
  "exports": {
    "node": {  // Packages can drop this special case as they drop support for older Node.js
      // 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

- It’s a tool for easier semver planning during the transition period when some Node.js LTS support require(esm) and some don’t
- Package authors can drop all the conditions when require(esm) is available on all Node.js versions they support

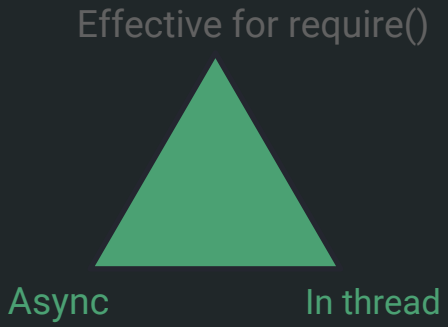
```
{
  "type": "module",
  // When the package no longer supports Node.js versions without require(esm),
  // just bump major version and get rid of the conditions.
  "main": "index.js"
}
```

CJS-only tooling transition: universal customization hooks

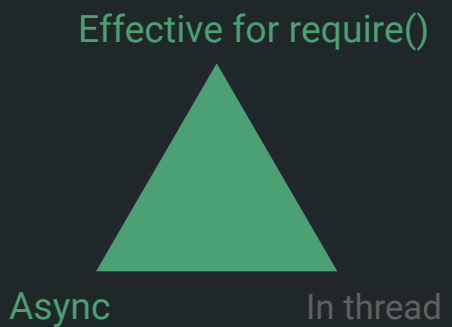
- Many high-impact tools in the ecosystem customize module loading by monkey-patching internals of `module.Module` or `require()` for mocking, transpiling & instrumentation
- Poses another underlying dependency on the CJS loader itself – your code could be migrated to ESM, but the tools you use will stop working if you do
- To help the tools migrate/support both CJS and ESM, we need a customization API that work for both ESM and CJS, and have an execution model close to what existing customizations operate on.
- Needed for customizing both the ESM being loaded by require(esm) and the CJS that calls require(esm)

CJS-only tooling transition: universal customization hooks

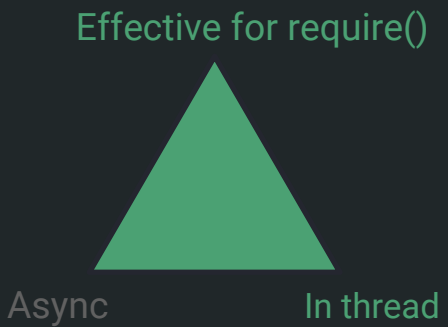
- Existing require() customizations generally need to run on the main thread to pass functions between the customized modules and the user configurations, or mutating the exports directly
- module.register() and --experimental-loader don't fit this model very well – they cling on built-in async handling



`--experimental-loader` & `module.register()` before v20



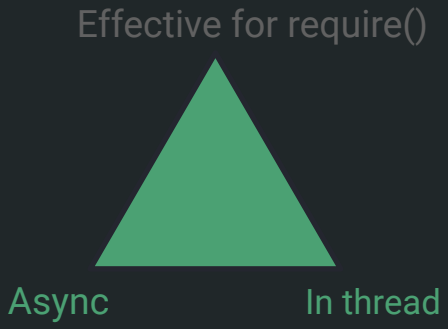
`--experimental-loader` & `module.register()` after v20



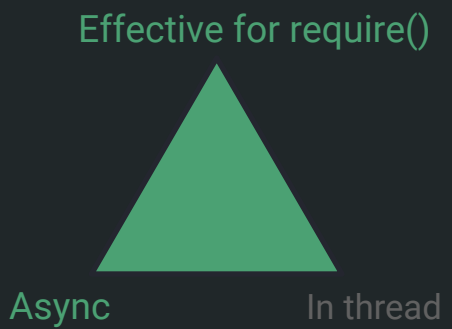
What existing require() customizations need for the migration

CJS-only tooling transition: universal customization hooks

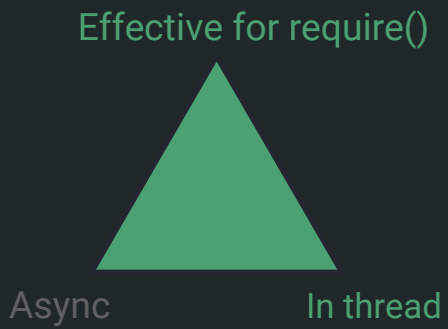
- Existing require() customization are already synchronous, or they already de-async specific asynchronous operations with their own worker threads
- Need a new universal, in-thread, synchronous API that works on both require() and import



`--experimental-loader` & `module.register()` before v20



`--experimental-loader` & `module.register()` after v20



New module.registerHooks() API provides

WIP: module.registerHooks({ resolve, load })

- Fills the gap left by module.register()
- Works for require() as well as import/import()
- Allows existing tools based on CJS monkey-patching to migrate and reuse code to support ESM

```
module.registerHooks({
  // Replace all the 'foo' in resolved paths with 'bar'
  resolve(specifier, context, nextResolve) {
    const result = nextResolve(specifier, context); // Use the default resolution
    return { ...result, url: result.url.replace('foo', 'bar') };
  },
  // Expand all the '@@foo' in source code to `console.log('foo')`, similar
  // to the "pirates" package example on npm.
  load(url, context, nextLoad) {
    const loaded = nextLoad(specifier, context); // Use the default load steps
    return {
      ...loaded,
      source: loaded.source.toString().replace('@@foo', 'console.log(\'foo\');')
    };
  }
});
```

https://github.com/nodejs/node/pull/55698/

Status of require(esm)

- 🚀 Unflagged in v23, still emits an experimental warning when invoked for the first time.
- 🧪 Available behind `--experimental-require-module` on v22 and v20
- 🗒️ Plan to backport the unflagging to v22, and maybe v20 in semver-minor releases
- 🎯 Hopefully stabilize and remove warning by v24 after customization is supported

Thanks