New & upcoming features in the Node.js module loaders

Joyee Cheung / @joyeecheung

require(esm)

Currently under `--experimental-require-module`, expected to be unflagged in v23 and backported to LTS after more battle-testing

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

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

require(esm)

If the ESM contains top-level `await`, it throws `ERR_REQUIRE_ASYNC_MODULE`

TLA in entry points works fine, just isn't supported when you try to `require()` a module that does it

```
// hello.mjs
export const hello = await Promise.resolve('I am async');

// index.cjs
// Throws ERR_REQUIRE_ASYNC_MODULE
console.log(require('./hello.mjs').hello);
```

require(esm)

- Use `--experimental-print-required-tla` to run the code and find out where that top-level `await` is.
- The experimental prefix is expected to be removed, and may be unflagged when we figure out how to find it without running the code

```
$ node --experimental-require-module \
       --experimental-print-required-tla \
       ./index.cjs


Error: unexpected top-level await at file:///.../hello.mjs:1
export const hello = await Promise.resolve('I am async');
                     ^
```

process.getBuiltinModule(id)

Since v22.3.0 and v20.16.0

```
// Old...
try {
  // Not sure if it's Node.js, have to use TLA and dynamic import
  const os = await import('node:os');  // Do some tuning with OS info
} catch { ... }

// New: no need for TLA
if (globalThis.process?.getBuiltinModule) {
  const os = globalThis.process.getBuiltinModule('os');
  // Do some tuning with OS info
}
```

Module syntax detection in .js files

Unflagged in v22.7.0 for ESM in .js files without a package.json

Previously Node.js would ask users to specify the module type for it to parse .js as ESM.

```
$ cat esm.js
import os from 'node:os';
console.log(os.cpus().length);

$ node esm.js  # Before 22.7.0
(node:437794) Warning: To load an ES module, set "type": "module" in the package.json or use
the .mjs extension.
(Use `node --trace-warnings ...` to show where the warning was created)
/home/ubuntu/node/esm.js:1
import os from 'node:os';
^^^^^^

SyntaxError: Cannot use import statement outside a module
```

Module syntax detection in .js files

Now if it doesn’t parse as CommonJS, Node.js can try to parse the module again as an ES Module if the syntax errors indicate that it might be one.

```
$ cat esm.js
import os from 'node:os';
console.log(os.cpus().length);

$ node esm.js  # Since 22.7.0
16
```

Module syntax detection

- If you have a package.json for a .js file in ESM syntax, it's still recommended to add `"type": "module"` to your package.json
  - Tools/IDEs still consume this field for configuration
  - The file gets parsed twice: Node.js does the detection by parsing it as CommonJS first, and fallback to ESM when there is a syntax error
  - Detection result may not match your expectation or future syntax additions
- When you do have a package.json for a .js file in ESM syntax, and it's missing the type field, Node.js reminds you about it with a warning

Module compile cache

- `NODE_COMPILE_CACHE` environment variable available since v22.1.0
- `module.enableCompileCache()` JS API available since 22.8.0
- Built-in alternative to v8-compile-cache/v8-compile-cache-lib packages
  - (But also supports ESM ✨)

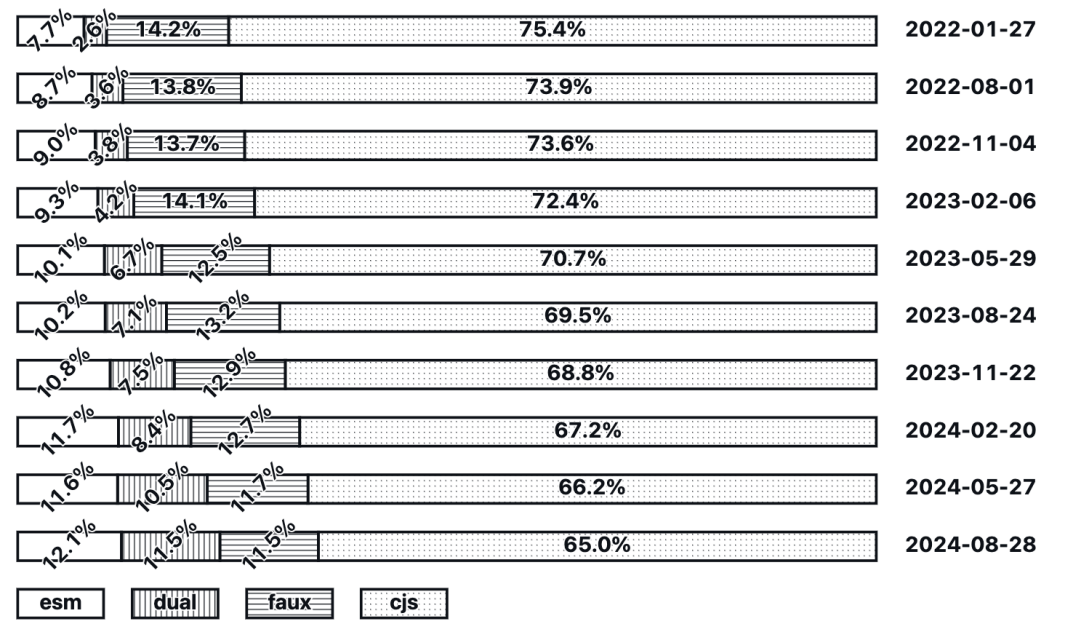
Module compile cache

- When compiling a module
  - If this is loaded for the first time on the system, compile it from source 🐢, and write the code cache (basically bytecode) to disk to speed up the next load
  - If the code cache is already there, read and deserialize it 🚀
  - Prototype in tsc made it load 2.3x faster 🚀

Upcoming (work in progress) features

WIP: “module” exports condition

Package authors have been shipping dual-module packages to maximize compatibility



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

WIP: “module” exports condition

To avoid the dual package hazard (both CommonJS and ESM version of the same package get loaded in the graph, conflicting with each other), package authors have been shipping CJS-first for Node.js.

```
{
  "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"
  }
}
```

WIP: “module” exports condition

https://github.com/nodejs/node/pull/54648

Support new patterns to help dual package authors prioritize ESM on newer versions of Node.js

```
{
  "type": "module",
  "exports": {
    "node": {
      // On new version of Node.js, both require() and import get the ESM version
      "module": "./index.js", /* This name is still under discussion! */

      // On older version of Node.js, where "module" and require(esm) are not supported,
      // use the transpiled CJS version to avoid dual-module hazard. Library authors
      // can drop all these when they think it's time to drop support for older
      // versions of Node.js
      "default": "./dist/index.cjs"
    },
    // On any other environment, use the ESM version.
    "default": "./index.js"
  }
}
```

WIP: Special default export for require(esm)

https://github.com/nodejs/node/pull/54563

Address CJS -> ESM default export disparity for package authors

```
// CommonJS library code
module.exports = function foo () { ... };
// ESM user gets..
import foo from 'foo';
// CJS user gets..
const foo = require('foo');

// Upgrade to ESM
export default function foo() { ... }

// In ESM, default export is placed separately from named exports 🤔
// ESM user gets..
import foo from 'foo';
// CJS user gets..
const foo = require('foo').default;
// Node.js can't just unwrap 'default', because additional `export named` would get lost
```

WIP: Special default export for require(esm)

Not a problem if module doesn't have default exports, but when they do, we need a hint from package authors to customize what `require(esm)` should return.

```
// Upgrade to ESM
export default function foo() { ... }
export { foo as 'module.exports' } /* This is still under discussion! */

// CommonJS user gets..
const foo = require('foo'); // No need to change the code to get default exports 🤓
```

WIP: synchronous module loader hooks

Fills the gap left by `module.register(specifier, parentURL)`

- Synchronous, light-weight and in-thread
- Allows mutating the main context and passing functions around in loader code
- Easier debugging
- Works for all `require()` as well as `import/import()`
- Allows replacing existing code monkey-patching `require` and `module.Module.prototype`
  - And allow them to support ESM out of the box ✨

WIP: synchronous module loader hooks

https://github.com/nodejs/loaders/pull/198

```
/* Early draft! */
// Porting from the example of the pirates package on npm:
// Expanding @foo in the source code of every loaded user module to console.log('foo')
module.registerHooks({
  load(url, context, nextLoad) { // Notice that it's not async
    const loaded = nextLoad(specifier, context);
    if (!url.endsWith('.js')) { return loaded; }
    const originalSource = loaded.source.toString();
    loaded.source = originalSource.replace('@@foo', 'console.log(\'foo\');');
    return loaded;
  }
});
```

Unblocks future additions..

- More specialized hooks coming up to mutate `require()` results directly (i.e. what the `require-in-the-middle` npm package does)
- Customize ESM after it's compiled/parsed.

WIP: noderc - auto-loaded runtime configuration file

- Sometimes a tool may want their users to preload some of their code to prepare the environment before any other user code is run
  - `--import`/`--require` must be passed down by the user code spawning workers ro processes
- It can get difficult for users to pass additional Node.js command line flags if the Node.js process is launched via another executable that takes control over the flags
  - `NODE_OPTIONS` might be overridden if the code is not careful
- UX would be nicer if these configurations can be specified with a file & can be edited by tools

WIP: noderc - auto-loaded runtime configuration file

https://github.com/nodejs/loaders/pull/225

```
// Early draft!
// package.json
{
  "noderc": "./.noderc.json"
}

// .noderc.json
{
  "schema": 0,
  // For all the processes and their child workers launched from this directory,
  // preload a telemetry script to monitor the process/report errors remotely.
  "import": [ "./telemetry.js" ],
  // Loads the environment variables from a local file
  "env-file": [ "./.env.local" ]
}
```

WIP: noderc - auto-loaded on-disk runtime configuration

More ideas…

- Structural representation of the configurations
- Reuse/overrides/extension of common configurations for dev/prod/test environments

Discussions here!
https://github.com/nodejs/loaders/pull/225

Thanks