Evolving the Node.js Module Loader

Node.js のモジュールローダー
を進化させる

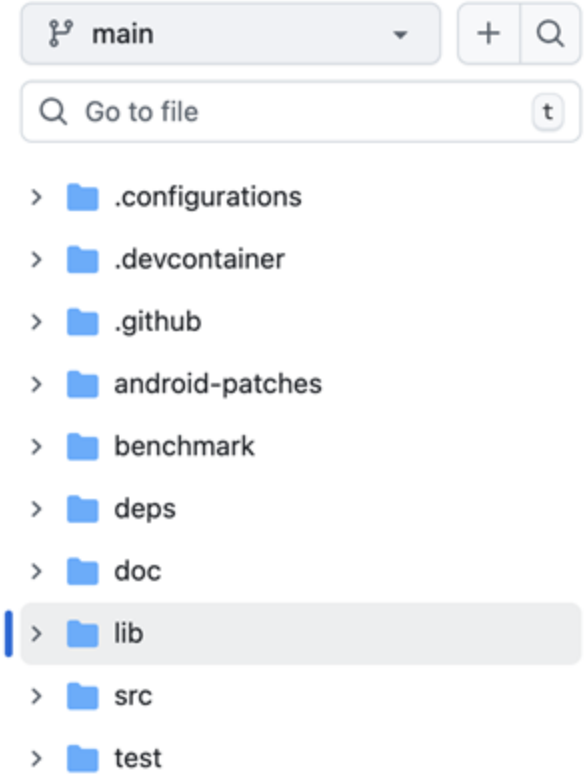
About me

- Joyee Cheung 張秋怡（中国・広州出身、スペイン・ア・コルーニャ在住）
- @joyeecheung on Github, @joyeecheung.bsky.social on BlueSky
- Igalia & Bloomberg
- Node.js TSC member & V8 committer
- Been tinkering with various parts of Node.js
- Recently been working on the Node.js module loader

This talk

- How some recent changes in Node.js module loader(s?) came about, from my perspective
- The challenges of evolving the module loader

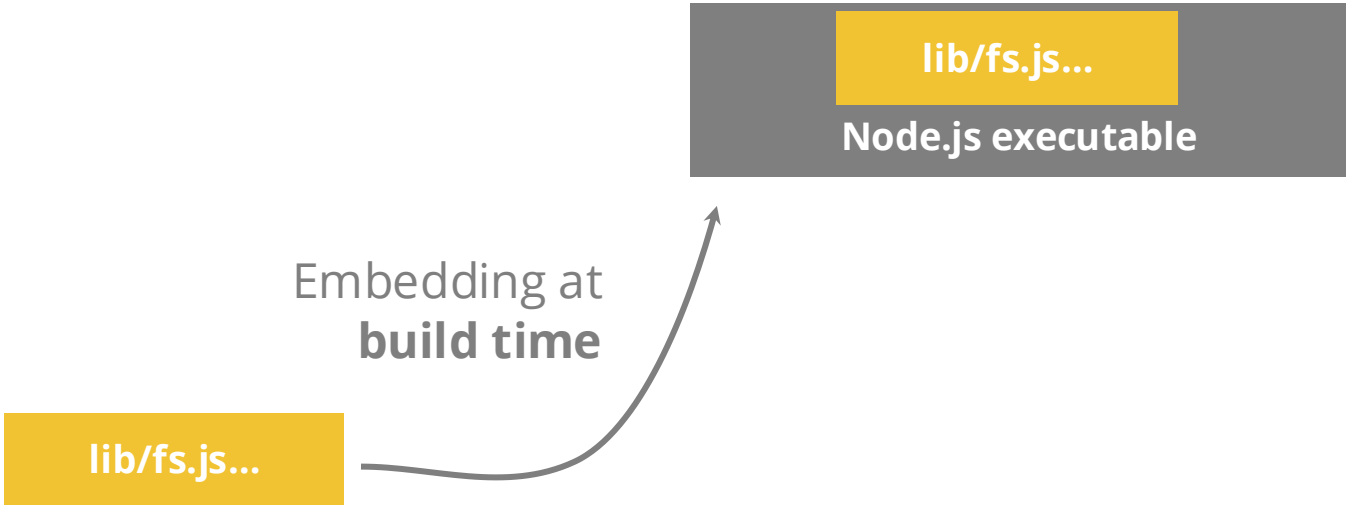
My first involvement in Node.js module loading: compile cache



- Node.js core is roughly written in half JS, half C++
- JS half in `lib/`
- C++ half in `src/`

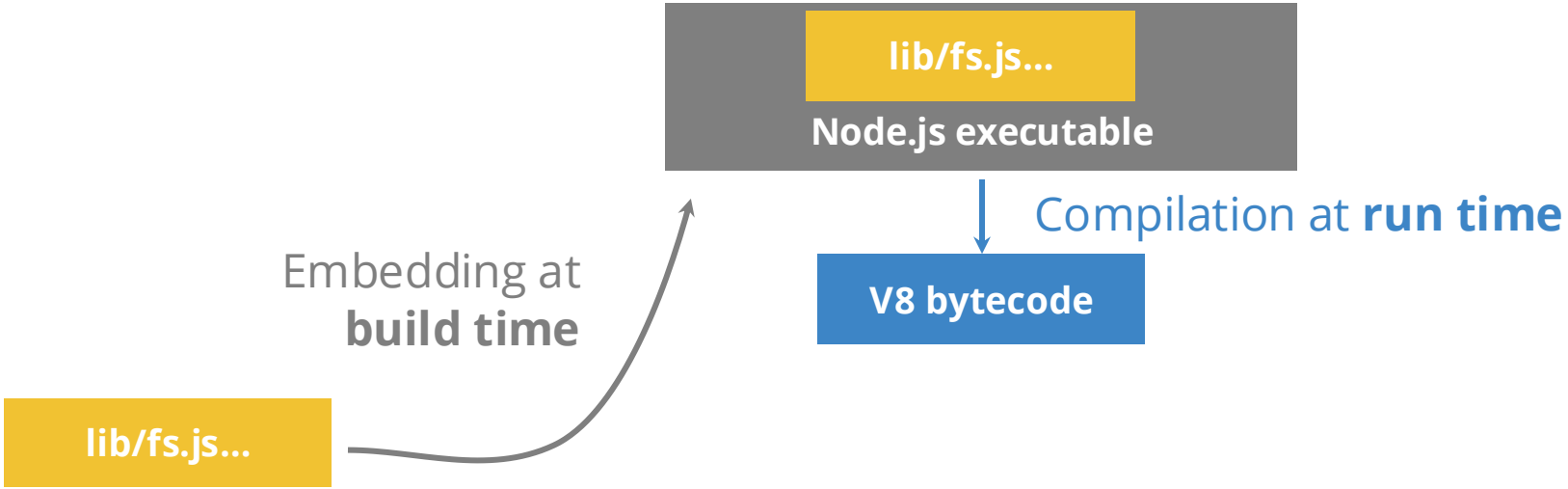
Node.js built-in module loading: ake 1

- Unlike C++, JS isn't typically compiled ahead of time
- Originally, Node.js embedded the JS into the executable



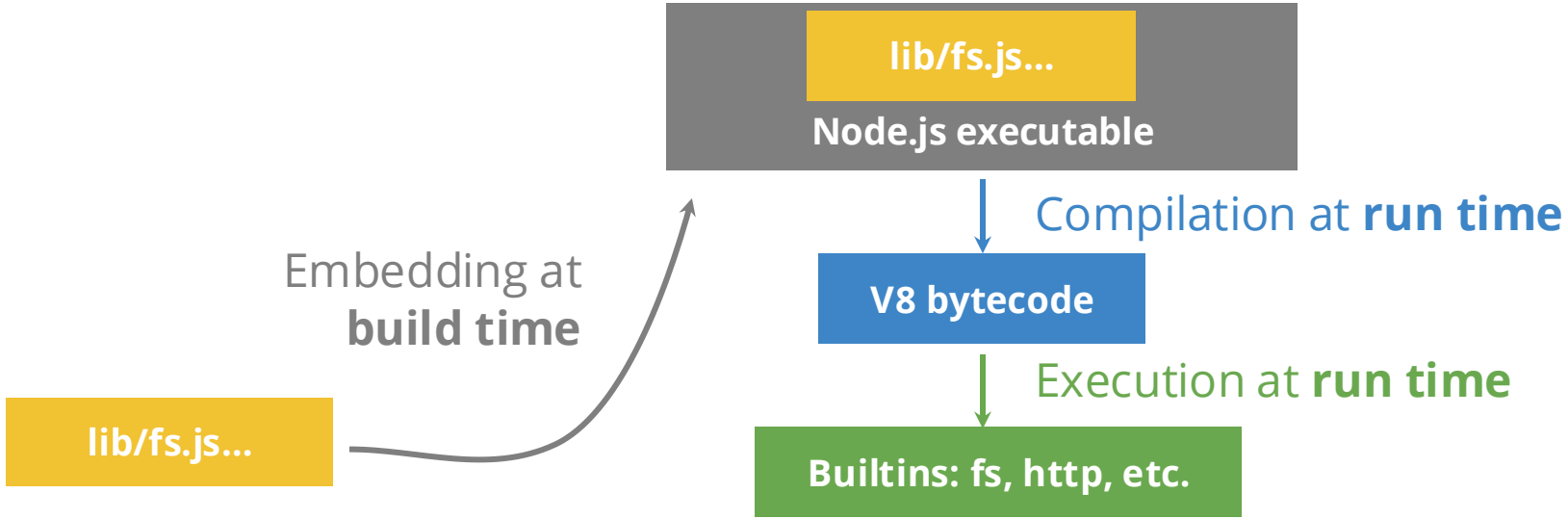
Node.js built-in module loading: take 1

The JS code was parsed & compiled at run time, usually into bytecode first (after V8 Ignition was rolled out)



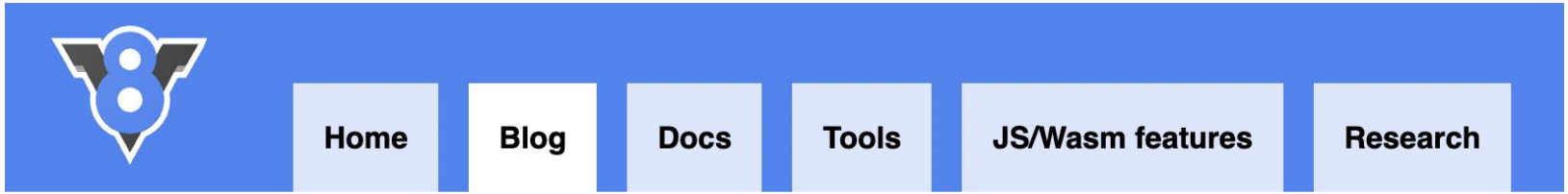
Node.js built-in module loading: take 1

The bytecode was executed at run time to initialize builtins like `fs`, `http`, etc.



Node.js built-in module loading: take 2

Proposed by **Yang Guo** (ex-V8 engineer who worked on **V8 code cache**)



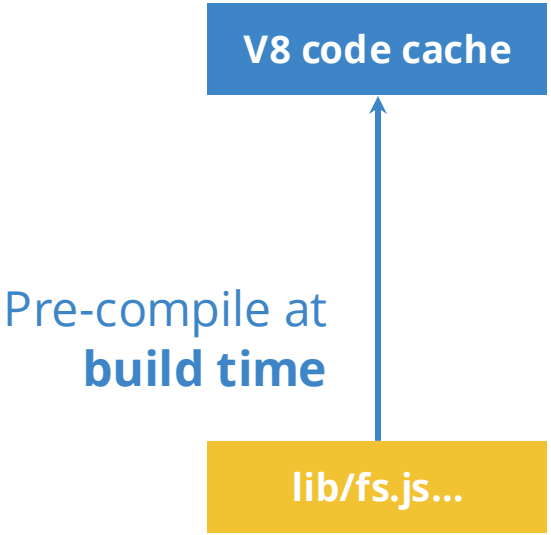
Code caching

Published 27 July 2015 · Tagged with internals

V8 uses just-in-time compilation (JIT) to execute JavaScript code. This means that immediately prior to running a script, it has to be parsed and compiled — which can cause considerable overhead. As we announced recently, code caching is a technique that lessens this overhead. When a script is compiled for the first time, cache data is produced and stored. The next time V8 needs to compile the same script, even in a different V8 instance, it can use the cache data to recreate the compilation result instead of compiling from scratch. As a result the script is executed much sooner.

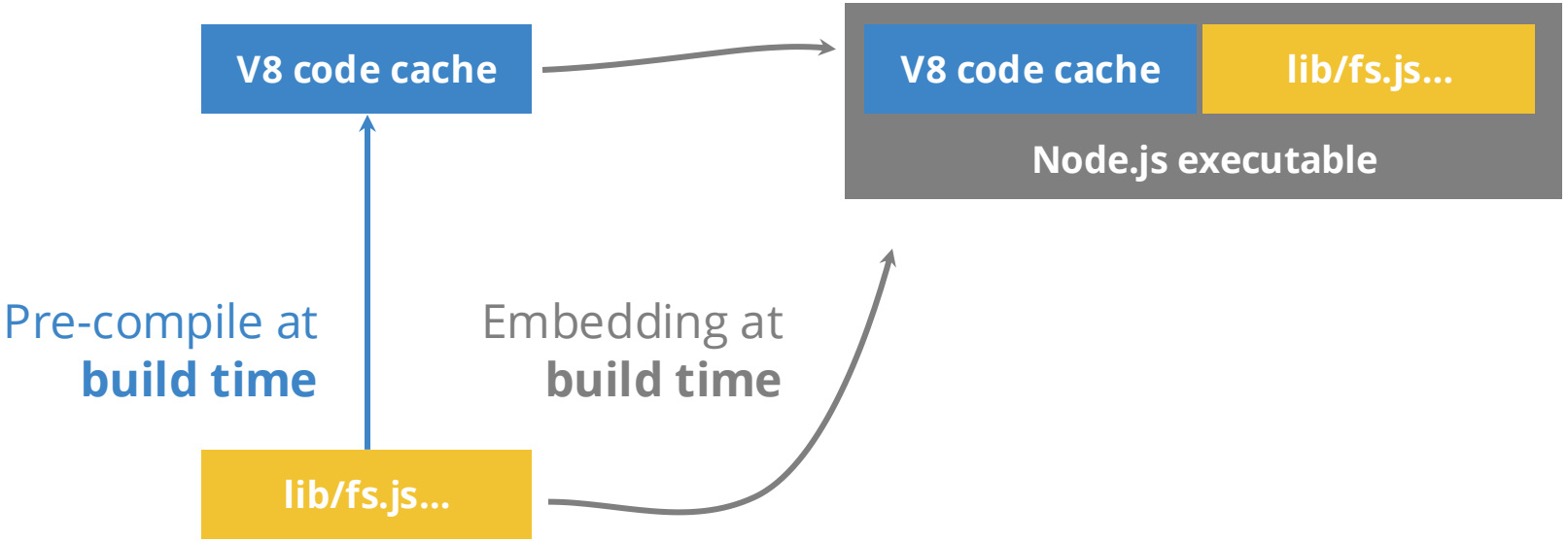
Node.js built-in module loading: take 2

At build time, Node.js pre-compile the JS, serialize the V8 code cache (bytecode + metadata)



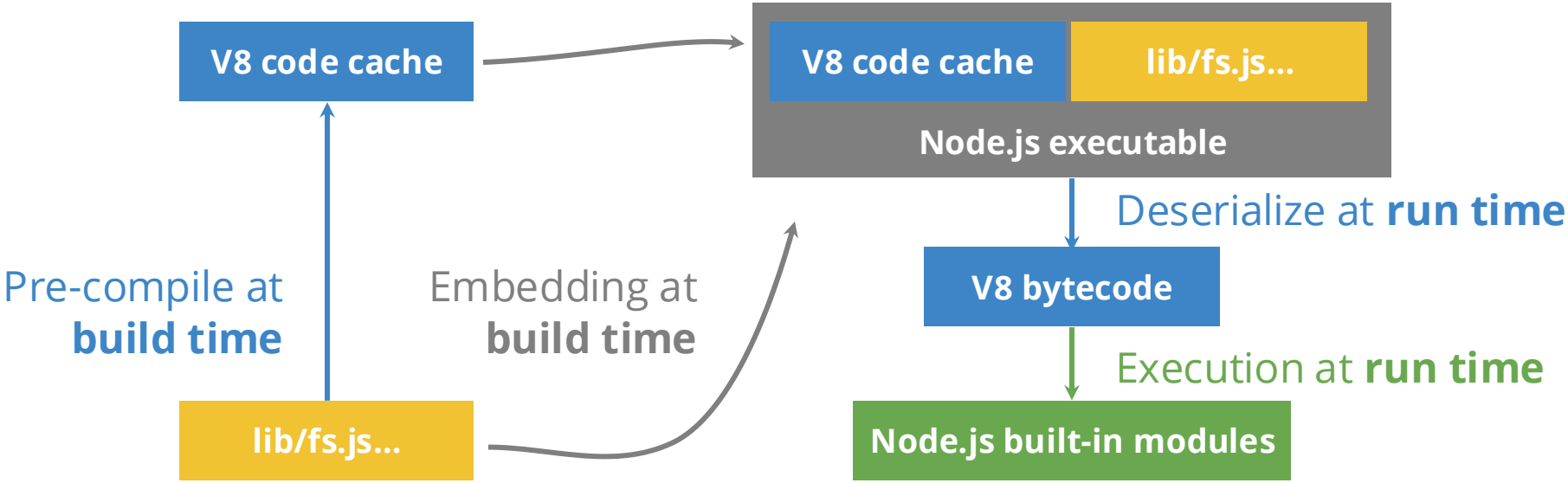
Node.js built-in module loading: take 2

- The source code and the code cache can be compiled into the executable
- The source code is still needed for debugging and better stack traces



Node.js built-in module loading: take 2

Deserialize the bytecode at run time, saving parsing and compilation time



Node.js built-in module loading: take 2

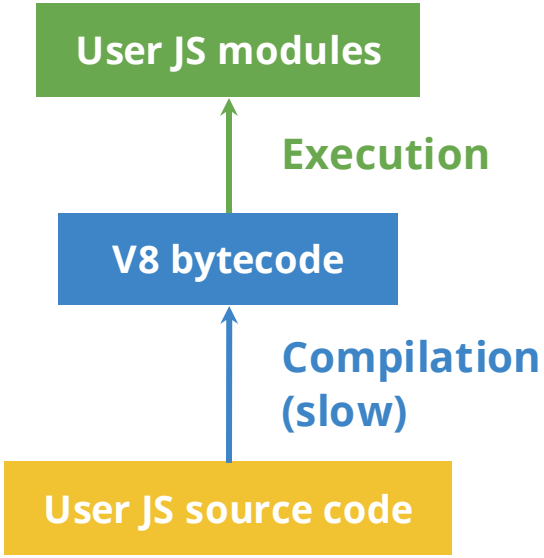
Added code cache integration for Node.js internal built-ins in 2018

```
                          confidence improvement accuracy (*)   (**)  (***)
misc/startup.js dur=1            ***     66.19 %       ±1.84% ±2.45% ±3.20%
```

	Without code cache	With code cache
Time to finish JS tests	3:00	2:25
Binary Size	37863108	38914164
RSS after startup	20365312	19210240
heapTotal after startup	6062080	5537792
heapUsed after startup	3781136	3227328

Node.js user-land module loading: take 1

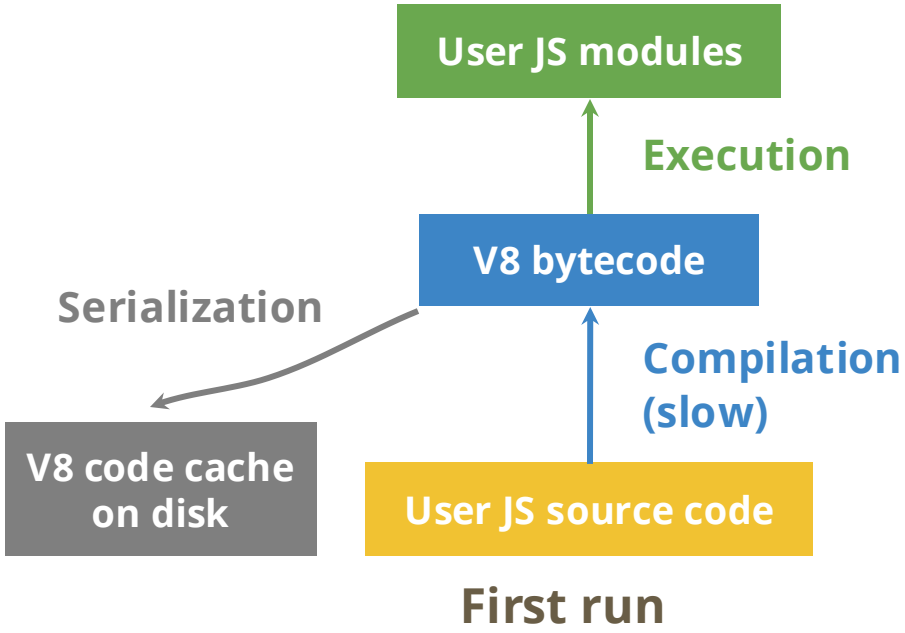
Great! Maybe it can be done for user-land modules too?



First run

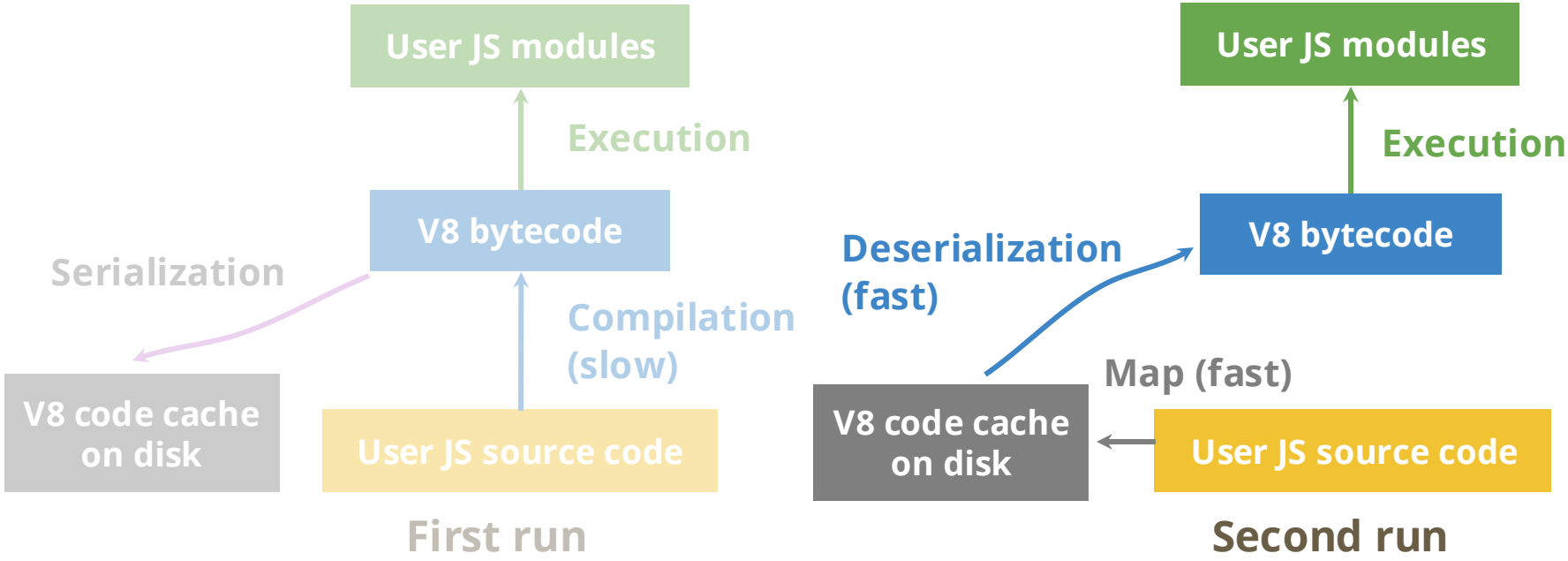
Node.js user-land module loading: take 2

Great! Maybe it can be done for user-land modules too?



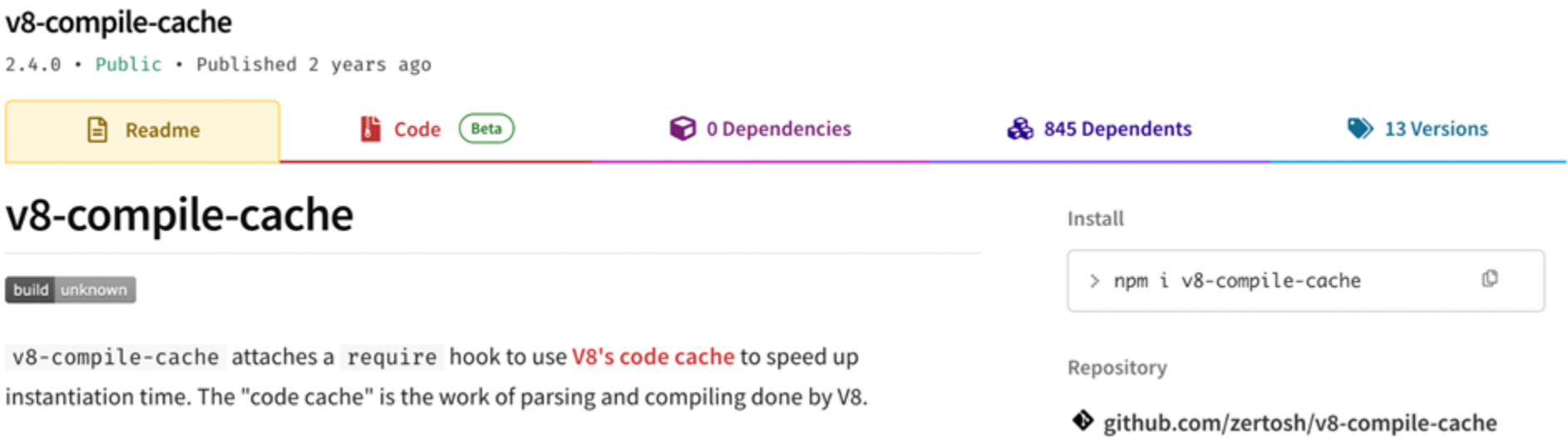
Node.js user-land module loading: take 2

Great! Maybe it can be done for user-land modules too?



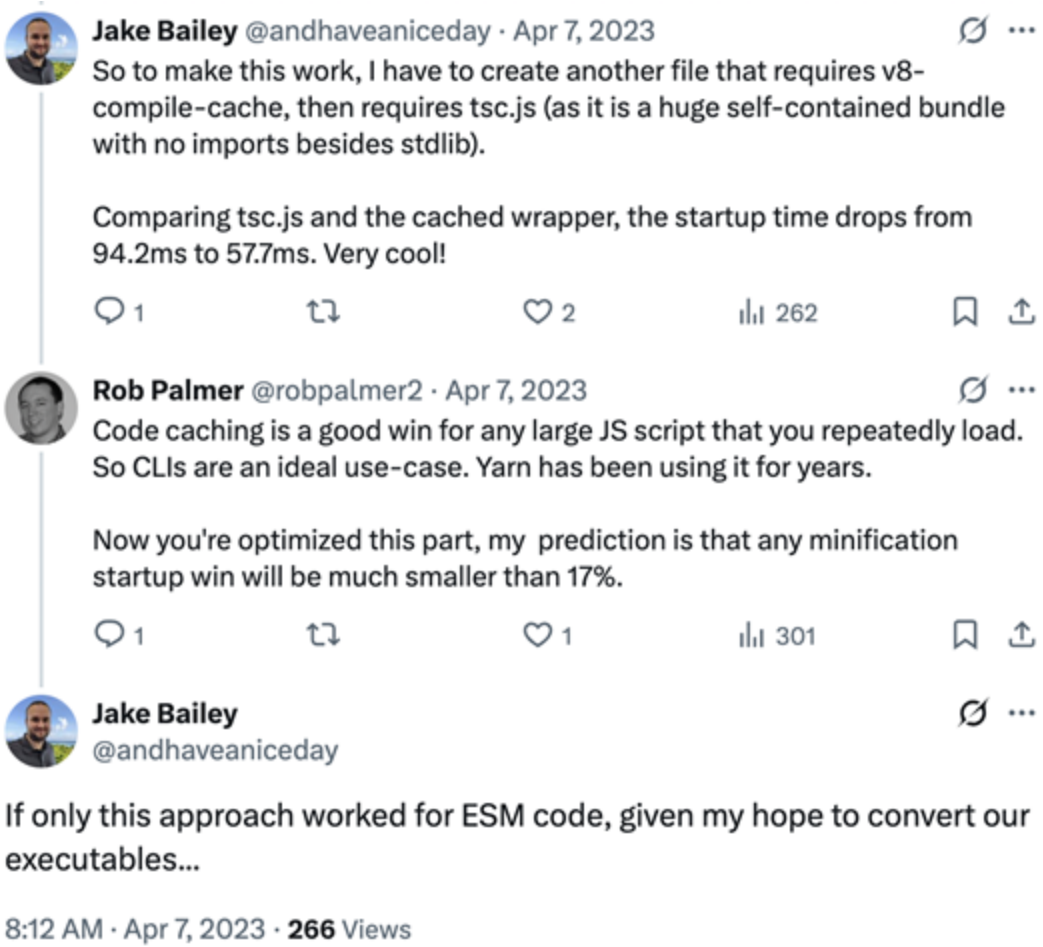
Node.js user-land module loading: take 2?

- Discussed at one of the Node.js collaboration summits
- It’s already implemented in the user-land by a package!
- No need to implement it in core?



Fast forward to 2023..

The user-land solution could affect ESM adoption?



Jake Bailey @andhaveaniceday · Apr 7, 2023

So to make this work, I have to create another file that requires v8-compile-cache, then requires tsc.js (as it is a huge self-contained bundle with no imports besides stdlib).

Comparing tsc.js and the cached wrapper, the startup time drops from 94.2ms to 57.7ms. Very cool!

1 · 2 · 262

Rob Palmer @robpalmer2 · Apr 7, 2023

Code caching is a good win for any large JS script that you repeatedly load. So CLIs are an ideal use-case. Yarn has been using it for years.

Now you're optimized this part, my prediction is that any minification startup win will be much smaller than 17%.

1 · 1 · 301

Jake Bailey
@andhaveaniceday

If only this approach worked for ESM code, given my hope to convert our executables...

8:12 AM · Apr 7, 2023 · **266** Views

What the Node.js module loader looked like

node / lib / internal / modules / cjs / **loader.js**

Code | Blame | 1596 lines (1436 loc) · 50.2 KB

```
1318  Module.prototype._compile = function(content, filename) {
1319    let moduleURL;
1320    let redirects;
1321    const manifest = policy()?.manifest;
1322    if (manifest) {
1323      moduleURL = pathToFileURL(filename);
1324      redirects = manifest.getDependencyMapper(moduleURL);
1325      manifest.assertIntegrity(moduleURL, content);
1326    }
1327
1328    const compiledWrapper = wrapSafe(filename, content, this);
1329
1330    let inspectorWrapper = null;
```

※ Code from pre ES6 era

What the Node.js module loader looked like

node / lib / internal / modules / cjs / **loader.js**

Code | Blame | 1596 lines (1436 loc) · 50.2 KB

```
1318  Module.prototype._compile = function(content, filename) {
1319
1320
1321
1322
1323    moduleURL = pathToFileURL(filename);
1324    redirects = manifest.getDependencyMapper(moduleURL);
1325    manifest.assertIntegrity(moduleURL, content);
1326  }
1327
1328  const compiledWrapper = wrapSafe(filename, content, this);
1329
1330  let inspectorWrapper = null;
```

Naming it `Module.prototype._compile` should be enough to tell users this is not public API and can break any time, right?

Being called Module.prototype._compile doesn't make it internal

node / lib / internal / modules / cjs / **loader.js**

Code | Blame | 1596 lines (1436 loc) · 50.2 KB

```
1318  Module.prototype._compile = function(content, filename) {
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328    const compiledWrapper = wrapSafe(filename, content, this);
1329
1330    let inspectorWrapper = null;
```

Hyrum's Law

With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.

And people monkey-patch it to customize compilation

v8-compile-cache / v8-compile-cache.js

Code | Blame | 373 lines (318 loc) · 10.6 KB

```
141   class NativeCompileCache {
151     install() {
154       this._previousModuleCompile = Module.prototype._compile;
155       Module.prototype._compile = function(content, filename) {
156
157
158
159
160
161
162         // https://github.com/nodejs/node/blob/v10.15.3/lib/internal/modules/cjs/helpers.js#L28
163         function resolve(request, options) {
164           return Module._resolveFilename(request, mod, false, options);
```

What if Node.js refactored and removed this internal method?

Or no longer invoked it internally? Or invoked at a different time?

Or added other logic that this is not taking care of?

Vicious loop of monkey patching

By patching internals, it's **not future-proof** when new features are integrated...

You need access to more internals to keep the patched internals work with every internal!

Vicious loop of monkey patching

- Better implement it in core to support new features e.g. ESM...
- Also, Chrome (and most browsers) implements this internally, it makes sense for Node.js to implement it internally too

Restarting work on user-land compile cache

Support for file-system based persistent code cache in user-land module loaders #47472

Closed

joyeecheung opened on Apr 8, 2023 · edited by joyeecheung Edits Member ···

This stemmed from a Twitter thread. Specifically I am wondering if there are any concerns over having something similar to what https://github.com/zertosh/v8-compile-cache does in core, the general idea is:

1. If the user enables this feature (probably should be off by default) e.g. via an environment variable, whenever we compile a module, we produce the code cache for the module, and on process exit, we store any new cache produced in a cache directory on the file system.
2. The next time the process is launched (with this feature enabled again), whenever we are loading a module, we attempt to load the cache from that directory and use it when compiling the module, in order to speed up the start up (where most of the time is usually spent on compilation).

This is also similar to what Chrome does with the V8 code cache.

Assignees

joyeeche

Labels

discuss

module

Type

No type

Restarting work on user-land compile cache

- Addressed security concerns (out of Node.js threat model - which trusts everything from the file system)
- Got support from Bloomberg to work on it (h/t Rob)
- Refactored the spaghetti internals on the **minefield** to share caching logic everywhere compilation happens
    - Tried my best not to break anyone

Minefield, you say?

node / lib / internal / modules / cjs / **loader.js**

Code | Blame | 1596 lines (1436 loc) · 50.2 KB

```
1318  Module.prototype._compile = function(content, filename) {
1319    let moduleURL;
1320    let redirects;
1321    const manifest = policy()?.manifes
```

v8-compile-cache / **v8-compile-cache.js**

Code | Blame | 373 lines (318 loc) · 10.6 K

```
141  class NativeCompileCache {
151    install() {
154      this._previousModuleCompile = Module.prototype._compile;
155      Module.prototype._compile = function(content, filename) {
156        const mod = this;
```

Changing arguments or return types of underscored methods breaks popular packages 💣

What if I want to share compilation logic with other (slightly different) code?

(ノ°□°)ノ ⌒ ┻━┻

Yay it worked! Oh wait..

Fixed a V8 bug for the API that only Node.js uses to support `import()`

Merged ☆ 5401780 [compiler] reset script details in functions deserialized from code cache Cr

Change Info Show All

Submitted Apr 11, 2024
Owner Joyee Cheung
Uploader V8 LUCI CQ
Reviewers Leszek Swirski +1 V8 LUCI CQ
CC Mark Seaborn Hannes Payer
Andreas Haas Mark Mentovai
Clemens Back... almuthanna+...
and 10 more
Repo | Branch v8/v8 | main
Hashtags compiler

Submit Requirements

Reply

```
[compiler] reset script details in functions deserialized from code
cache

During the serialization of the code cache, V8 would wipe out the
host-defined options, so after a script id deserialized from the
code cache, the host-defined options need to be reset on the script
using what's provided by the embedder when doing the deserializing
compilation, otherwise the HostImportModuleDynamically callbacks
can't get the data it needs to implement dynamic import().

Change-Id: I33cc6a5e43b6469d3527242e083f7ae6d8ed0c6a
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5401780
Reviewed-by: Leszek Swirski <leszeks@chromium.org>
Commit-Queue: Joyee Cheung <joyee@igalia.com>
Cr-Commit-Position: refs/heads/main@{#93323}
```

Native compile cache is here!

- Landed initial implementation in 22.1.0
  - `NODE_COMPILE_CACHE=/path/to/cache/dir` to enable
  - Generate code cache in the specified directory in the first run, reuse code cache in second run (if the code doesn't change)
  - Supports both CommonJS and ESM ✨ transparently
- Added JS API in 22.8.0 for CLI tools
  - `module.enableCompileCache()`

Good news in the wild

Implementing it natively also allowed it to go faster than a user-land implementation

jakebailey commented on Aug 22, 2024 • edited

Member

This makes use of nodejs/node#54501 by converting our larger entrypoints (`tsc` , `tsserver` , `typingsInstaller`) into shims which call `require("node:module").enableCompileCache()` , then `require` the actual code. Our entrypoints are assumed to always be run in Node (or a compatible runtime), so this is safe.

This gives roughly a 2.5x startup time boost.

```
Benchmark 1: node ./built/local/_tsc.js --version
  Time (mean ± σ):     122.2 ms ±   1.5 ms    [User: 101.7 ms, System: 13.0 ms]
  Range (min … max):   119.3 ms … 132.3 ms    200 runs

Benchmark 2: node ./built/local/tsc.js
  Time (mean ± σ):      48.4 ms ±   1.
  Range (min … max):    45.7 ms …  52.

Summary
  node ./built/local/tsc.js --version
    2.52 ± 0.06 times faster than node
```

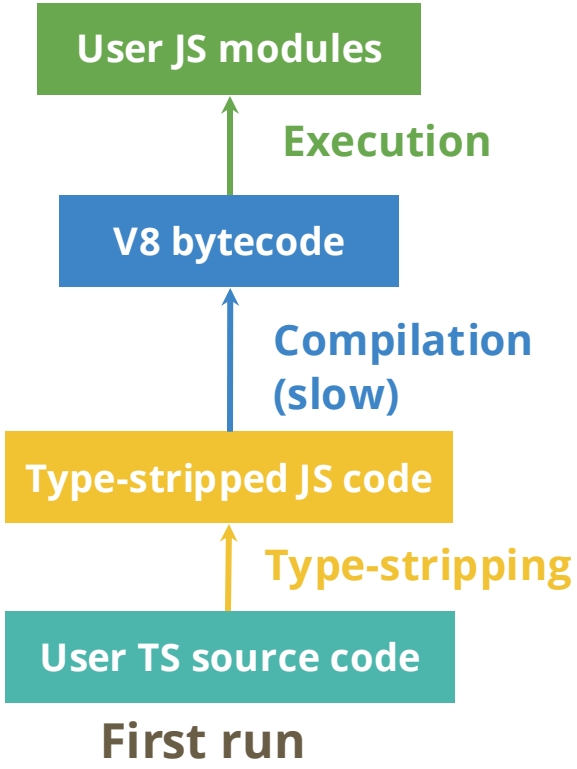
Nicholas C. Zakas
@humanwhocodes.com

ESLint enables the V8 compile cache by default in Node.js v22+. The result on my machine is a load time reduction of around 90%.

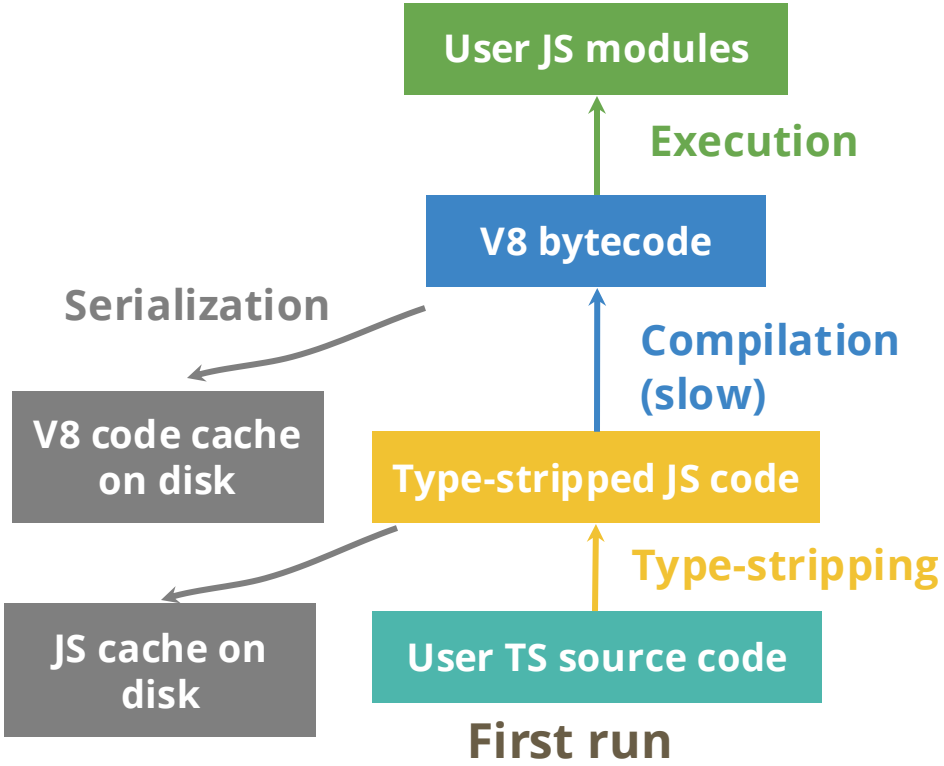
April 28, 2025 at 7:43 PM Everybody can reply

12 reposts **75** likes **2** saves

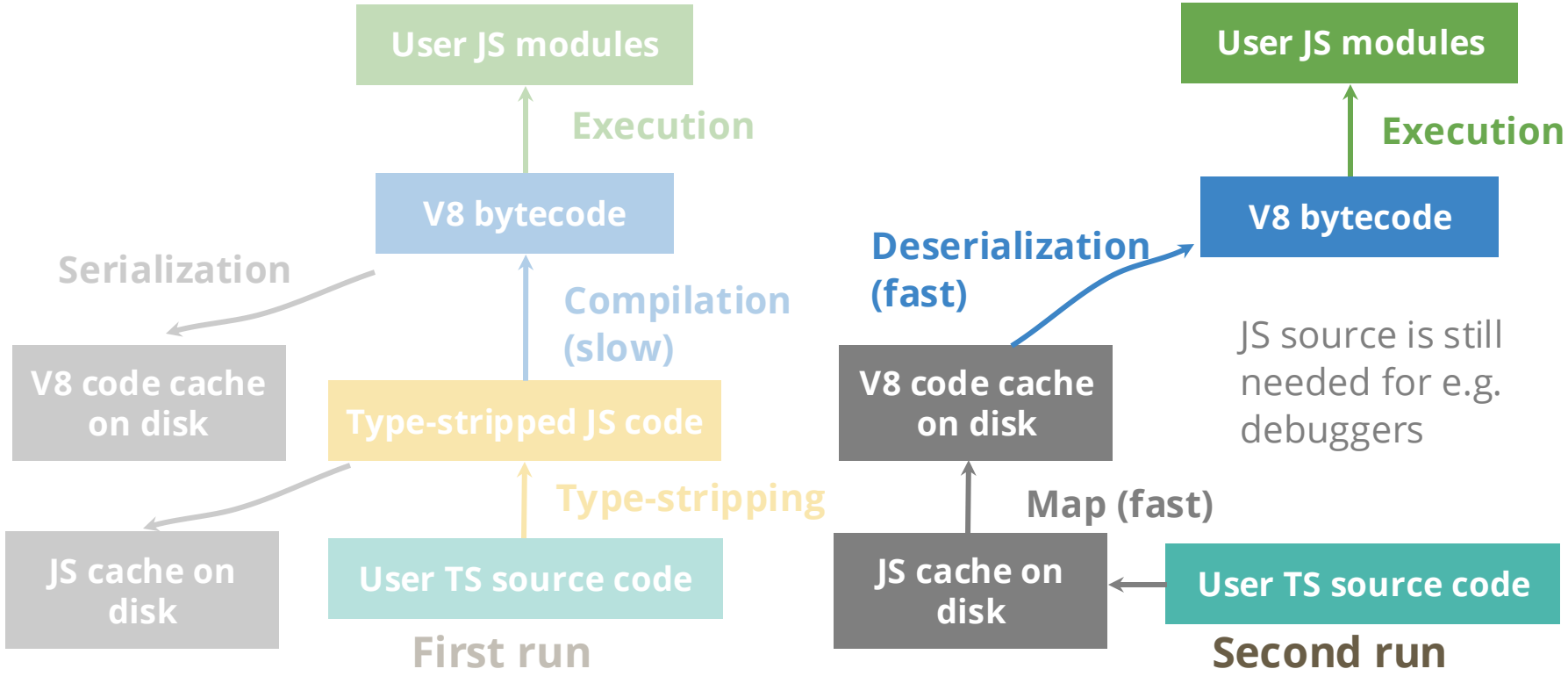
Node.js user-land module loading: take 3



Node.js user-land module loading: take 3



Node.js user-land module loading: take 3



Node.js user-land module loading: take 3

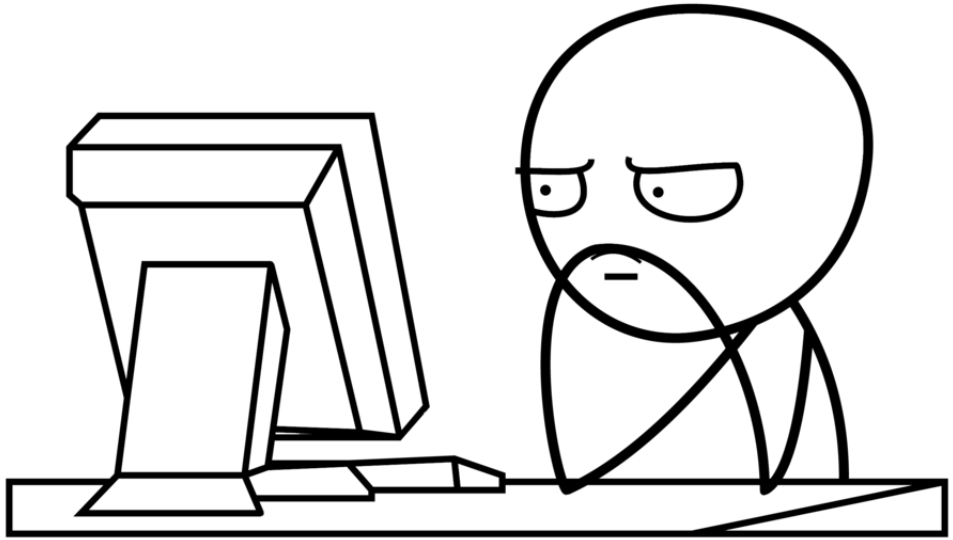
With compile cache enabled via `export NODE_COMPILE_CACHE=/tmp` :

```
                                                                confidence improvement accuracy (*)    (**)    (
:/node/benchmark/fixtures/strip-types-benchmark.js'                            2.31 %       ±3.44%  ±4.62%  ±6.
:/node/benchmark/fixtures/strip-types-benchmark.ts'                ***        65.37 %       ±3.99%  ±5.35%  ±7.03%
ects/node/benchmark/fixtures/transform-types-benchmark.js'                     0.37 %       ±1.37%  ±1.82%  ±2.37%
ects/node/benchmark/fixtures/transform-types-benchmark.ts'         ***       128.86 %       ±7.94% ±10.69% ±14.17%
```

- Still a bunch of TODOs https://github.com/nodejs/node/issues/52696
- Might possibly enable it by default when it's stable enough
  - There's a caveat of negative performance impact when the file is too small - not sure where's the sweet spot yet

Another adventure: require(esm)

I stared at the ESM module loader for too long to figure out how to refactor and integrate the compile cache..



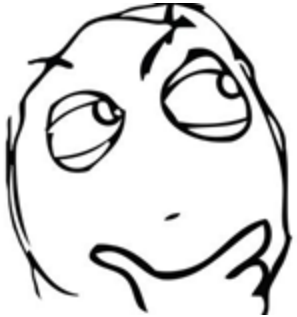
Another adventure: require(esm)

node / lib / internal / modules / esm / **module_job.js** ↑ Top

Code | Blame | 224 lines (203 loc) · 7.97 KB | Raw

```
50  class ModuleJob {
53    constructor(loader, url, importAttributes = { __proto__: null },
65      // Wait for the ModuleWrap instance being linked with all dependencies.
66      const link = async () => {
67        this.module = await this.modulePromise;
68        assert(this.module instanceof ModuleWrap);
69
70
71
72
73
74        const dependencyJobs = [];
```

Can I just write a synchronous version of this from scratch and use that to load ESM synchronously...



Implication of lack of require(esm)

Node.js ESM stabilized in 2020

```
{
  "name": "logger",
  "type": "commonjs", "module"
  "exports": "./index.js",
}
```

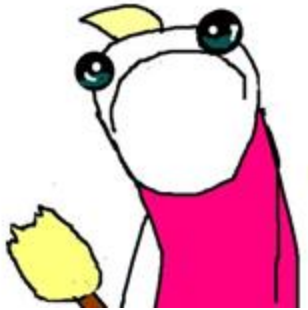
```
module.exports = class Logger {};

export default class Logger{};
```

MIGRATE ALL THE PACKAGES

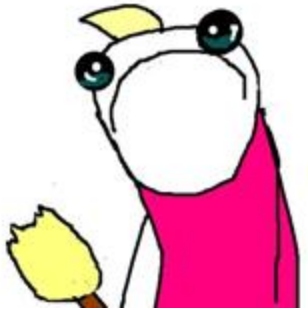
Implication of lack of require(esm)

```
// CJS consumer could no longer load it
const Logger = require('logger'); // ❌ Throws ERR_REQUIRE_ESM!
```



Implication of lack of require(esm)

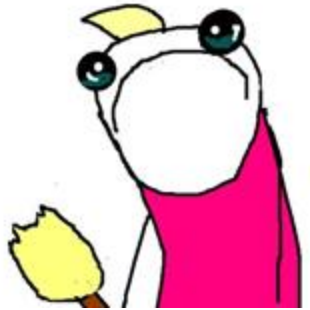
```
// Even consumers who thought they wrote ESM may no longer be able to load it
import Logger from 'logger'; // This should work, right?
```



Implication of lack of require(esm)

```
// Even consumers who thought they wrote ESM may no longer be able to load it
import Logger from 'logger'; // ⁉️ Throws ERR_REQUIRE_ESM

// What got run (magically transpiled by some tool or framework)
var _logger = _interopRequireDefault(require("logger"));// ❌ ERR_REQUIRE_ESM
function _interopRequireDefault(obj) {
  return obj && obj.__esModule ? obj : { default: obj };
}
const Logger = _logger.default;
```



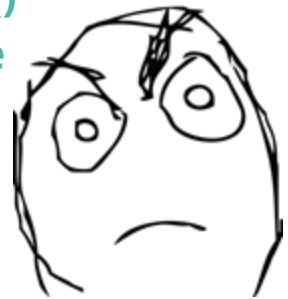
Implication of lack of require(esm)

Many packages shipped two copies to maintain backward compatibility (dual packages)

```
{
  "type": "module",
  "scripts": {
    "build": "babel src --out-dir dist"
  },
  "exports": {
    ".": {
      "require": "./dist/index.cjs",
      "import": "./src/index.js"
    }
  }
}
```

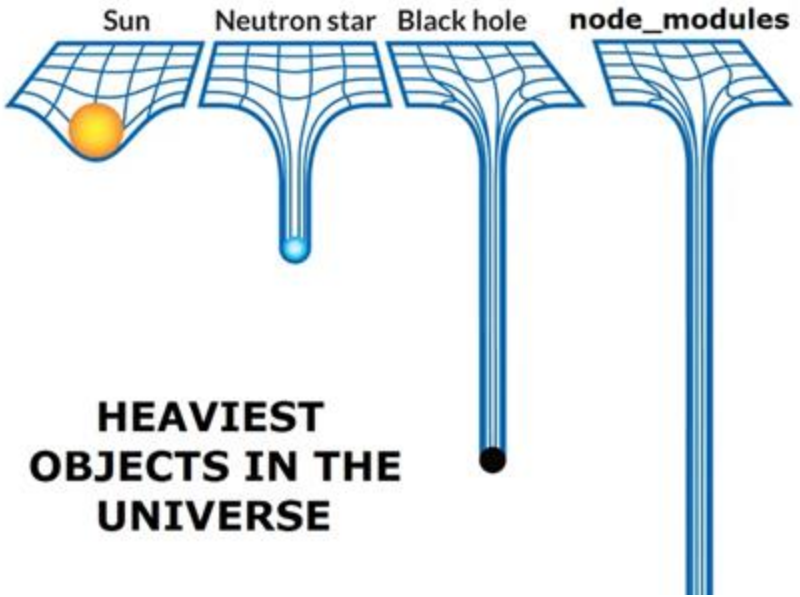
Add a transpilation step

- **Supply the CommonJS version to code loading it with require()**
- **Supply the ESM version to code loading it with import**

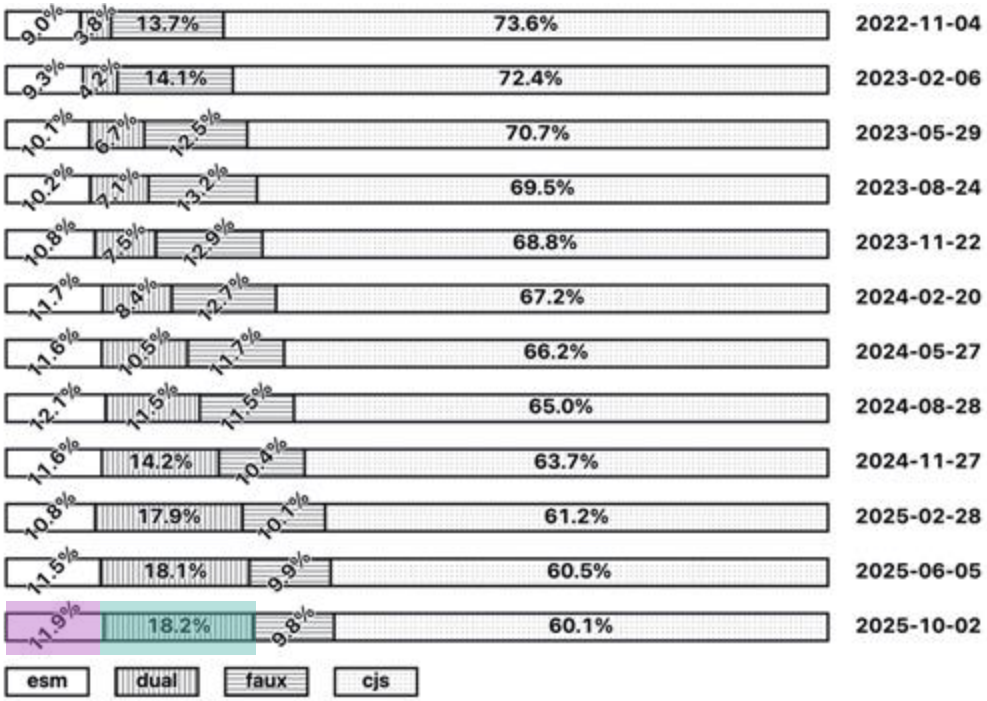


Implication of lack of require(esm)

Doubled the size of node_modules...



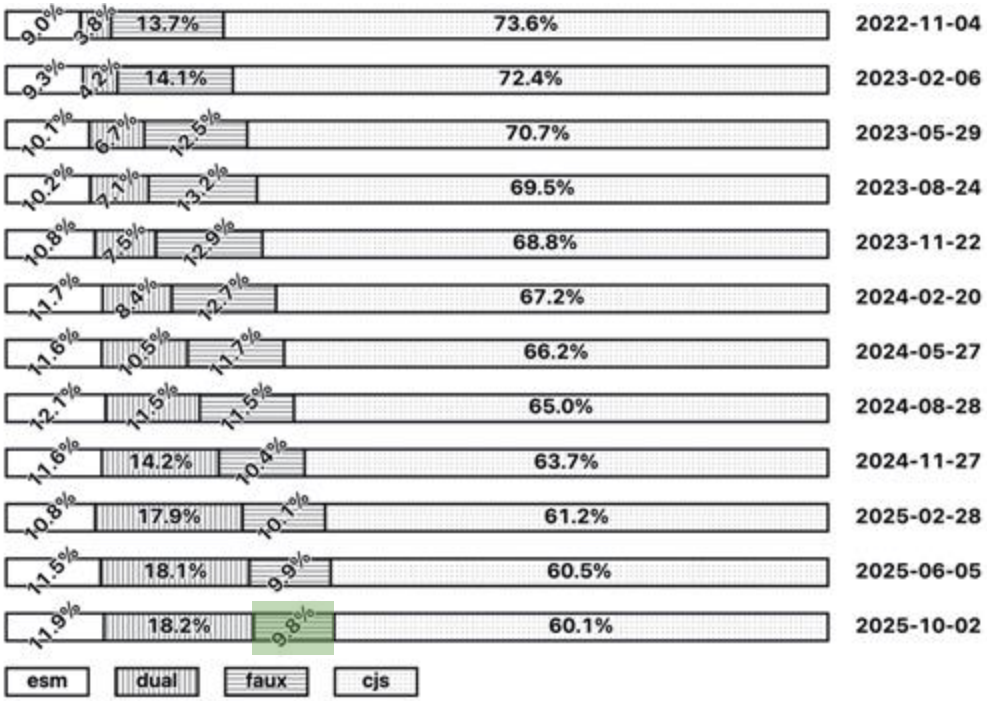
Implication of lack of require(esm)



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

- Effectively made ESM a less desirable **execution** format
- 5 years after ESM stabilization, **shipping dual** is more popular than than **shipping ESM directly**

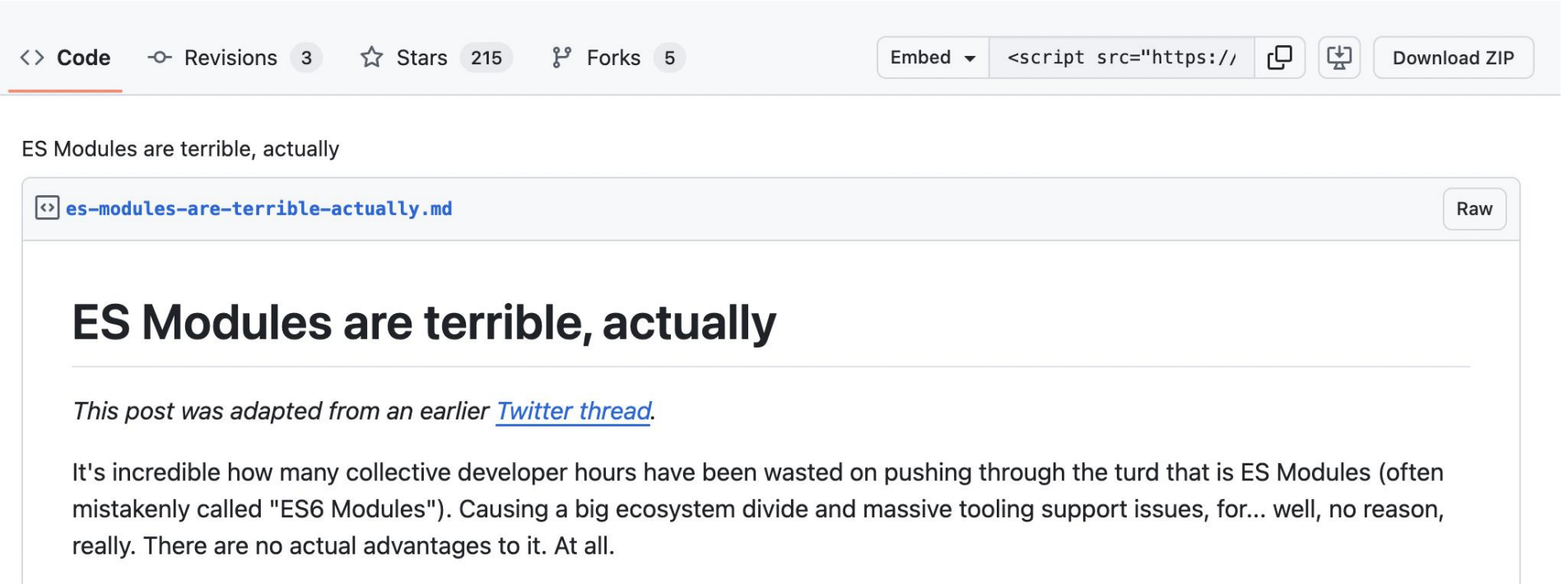
Implication of lack of require(esm)



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

- And many are still just shipping ESM transpiled into CommonJS (**faux ESM**)
- Stats include older versions; Actual CommonJS numbers likely to be even higher

Demotivated adoption of ESM



What if ESM can be loaded by require()... 🤔

```
// CJS consumers won't be broken by the migration
const Logger = require('logger'); // ✅

// Nor will faux-ESM consumers (or the frameworks/tools they use, so they
// can stop doing the magical transpilation!)
var _logger = _interopRequireDefault(require("logger"));// ✅
function _interopRequireDefault(obj) {
  return obj && obj.__esModule ? obj : { default: obj };
}
const Logger = _logger.default;
```

Shipping ESM would be simple again!

```
{
  "name": "logger",
  "type": "module",
  "scripts": {
    "build": "babel src --out-dir dist"
  },
  "exports": {
    ".": {
      "require": "./dist/index.cjs",
      "import": "./src/index.js"
    }
  }
}
```

```
{
  "name": "logger",
  "type": "module",
  "exports": "./index.js",
}
```

Going back to 2019

[WIP] Support requiring .mjs files #30891

Closed weswigham wants to merge 5 commits into `nodejs:master` from `weswigham:support-require-esm-in-the-style-of-TLA`

Conversation 88 Commits 5 Checks 0 Files changed 4

weswigham commented on Dec 10, 2019

This implements the ability to use `require` on `.mjs` files, loaded via the `esm` loader, using the same tradeoffs that top level await makes in `esm` itself.

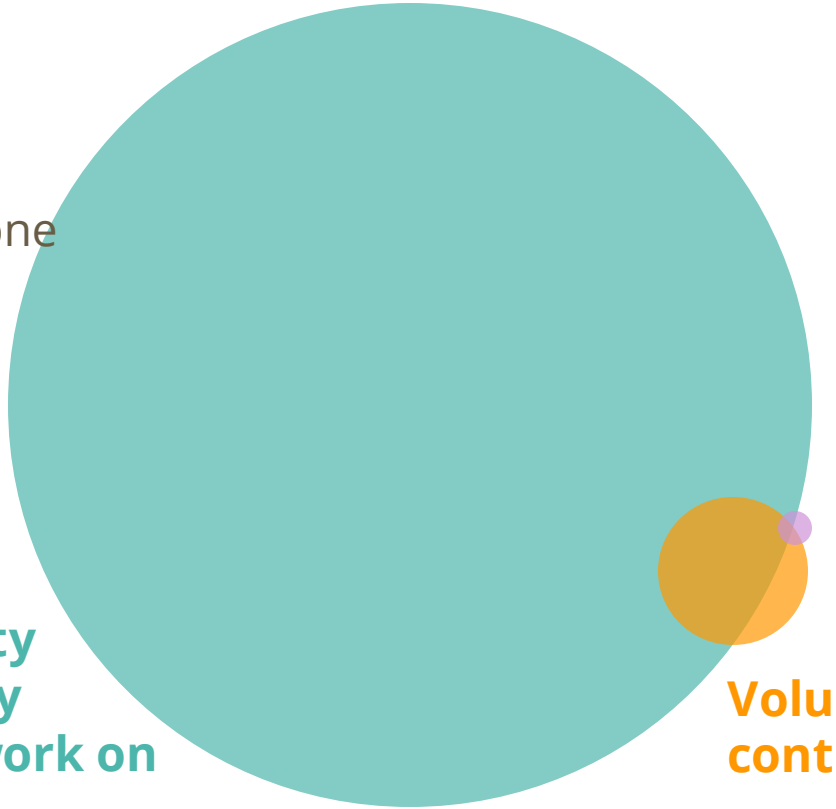
What this means: If possible, all execution and evaluation is done *synchronously*, via immediately unwrapping the execution's component promises. This means that any and all existing code should have no observable change in behavior, as there exist no asynchronous modules as of yet. The catch is that once a module which requires asynchronous execution is used, it must yield to the event loop to perform that execution, which, in turn, can allow other code to execute before the continuation after the async action, which is observable to callers of the now asynchronous module. If this matters to your callers, this means making your module execution asynchronous could be considered a breaking change to your library, however in practice, it will not matter for most callers. Moreover, as the ecosystem exists today, there are *zero* asynchronously executing modules, and so until there are, there are no downsides to this approach *at all*, as no execution is changed from what one

Going back to 2019

- **Concerns about safety** as it nested libuv event loop iteration to support top-level await, and could crash
- Lots of debates, stalled for too long, **resources ran out**
- Other contributors had **priority** to support `import cjs`, not `require(esm)`
- Contributors moved on after ESM stabilization in 2020

In reality, being a community-driven project means...

- No roadmaps
- No project management
- Work doesn't get done by itself



Work sponsored by companies & organizations

What the community wants the imaginary "Node.js team" to work on

Volunteer work done at contributors' free time

In reality, being a community-driven project means...

- 1 person saying no ≠ consensus of 100+ collaborators
- Decision making is based on consensus seeking, **no dictator**
- Navigating through disagreements to find compromise & reach consensus is also work
    - Also doesn't get done by itself!
- All these take time and can stall work

Why was require(esm) not there?

- Node.js documentation gave an answer since v12
  - Spoiler alert – it's not accurate
- Made many believe it was not practical by design and stop asking
  - Including myself before, since I didn't work with the ESM loader implementation
  - When you don't know much about it, just take what the documentation says, *right?*

require

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.

Fast forward to 2023

- Fixing a memory leak, looked at a similar leak tied to V8’ ESM implementation
- Wait a minute, V8’s code is contradicting what the Node.js docs say?

require

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.

ESM without top-level await is synchronous?!

Didn't the Node.js docs say ESM execution is async? Why is there a branch in V8?

```
for (Handle<SourceTextModule> m : exec_list) {
  if (m->has_toplevel_await()) {
    MAYBE_RETURN(ExecuteAsyncModule(isolate, m), Nothing<bool>());
  } else {
    if (!ExecuteModule(isolate, m).ToHandle(&unused_result)) {
      AsyncModuleExecutionRejected(isolate, m, exception);
    } else {
      JSPromise::Resolve(capability, isolate->factory()->undefined_value());
    }
  }
}
```

You mean, if there's no top level await, it's synchronously executed?

ESM without top-level await is synchronous?!

Mandated by spec: https://tc39.es/ecma262/#sec-innermoduleevaluation

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.

Pseudocode of a require(esm) implementation

```
// 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

Even the V8 API for checking TLA was already there

```
// 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();
}
```

V8 even already implemented this API in 2020 because ...

ESM without TLA is not that unorthodox on the Web

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`.

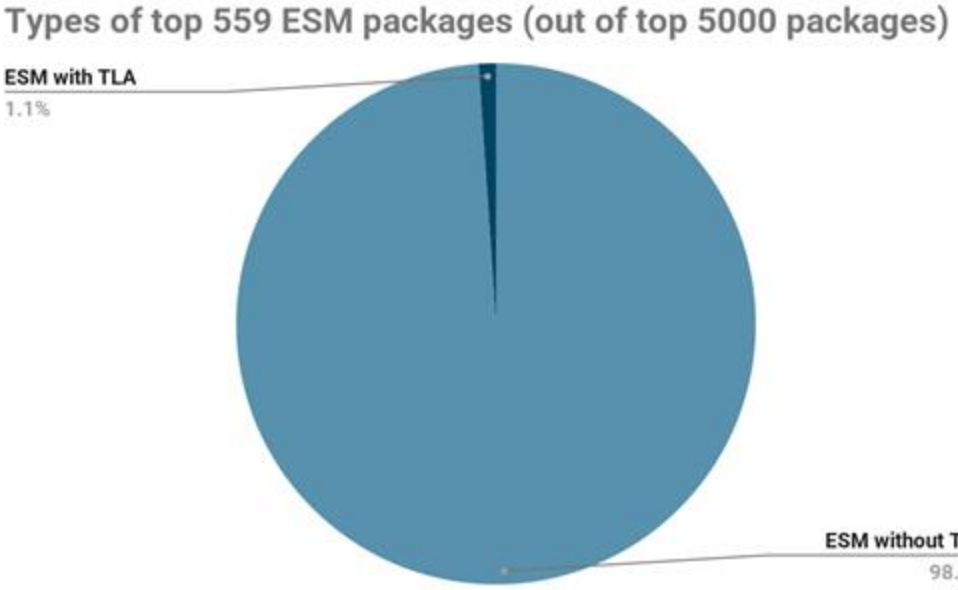
        > Note: This will do nothing if Reject Job Promise was previously invoked with "`SecurityError`" `DOMException`.

    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.

This Service Worker behaviour was brought up in the 2019 PR, but somehow remained a niche knowledge mostly known to people who actually worked on ESM

ESM *packages* using top-level await are rare



Only **6/5000** top high-impact packages have TLA, **5/6** only were not necessary and only added during migration to ESM

Top-level await is mostly used in apps/scripts/tests that import other modules, not modules shared to and loaded by external code controlled by other people

require(esm) without TLA has theoretical guarantees

```
// 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();
}
```

It should be easy to implement, then, right?

```
// 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();
}
```

Right?

It should be easy to implement, then, right?

```
// 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 poir
  // module graph contains no top-level await.
  return linkedModule.getNamespace();
}
```

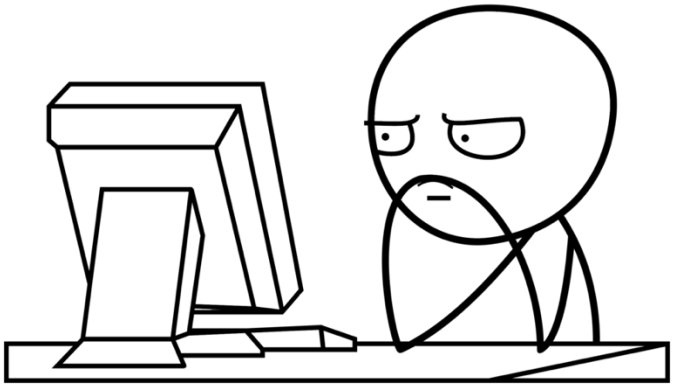
NO

The module loaders, from my perspective

- Node.js internal built-in loader: 🏰🌿
  - **400+ JS, 900+ C++(LOC)**, nicely encapsulated

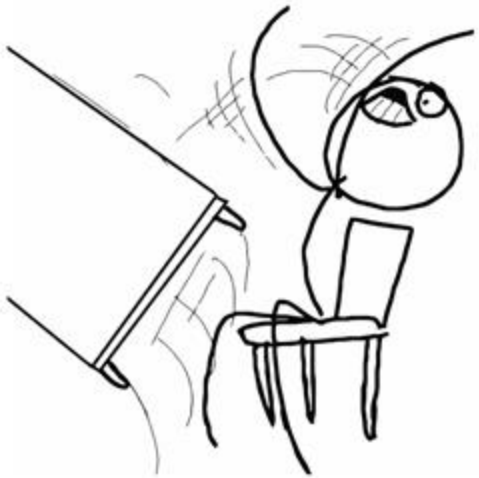
The module loaders, from my perspective

- Node.js internal built-in loader: 🏰🌿
  - 400+ JS, 900+ C++(LOC), nicely encapsulated
- Node.js user-land CommonJS loader: 🍯🐿️💣
  - **1000+ JS, 500+ C++ (LOC)**
  - Spaghetti code, monkey patched everywhere



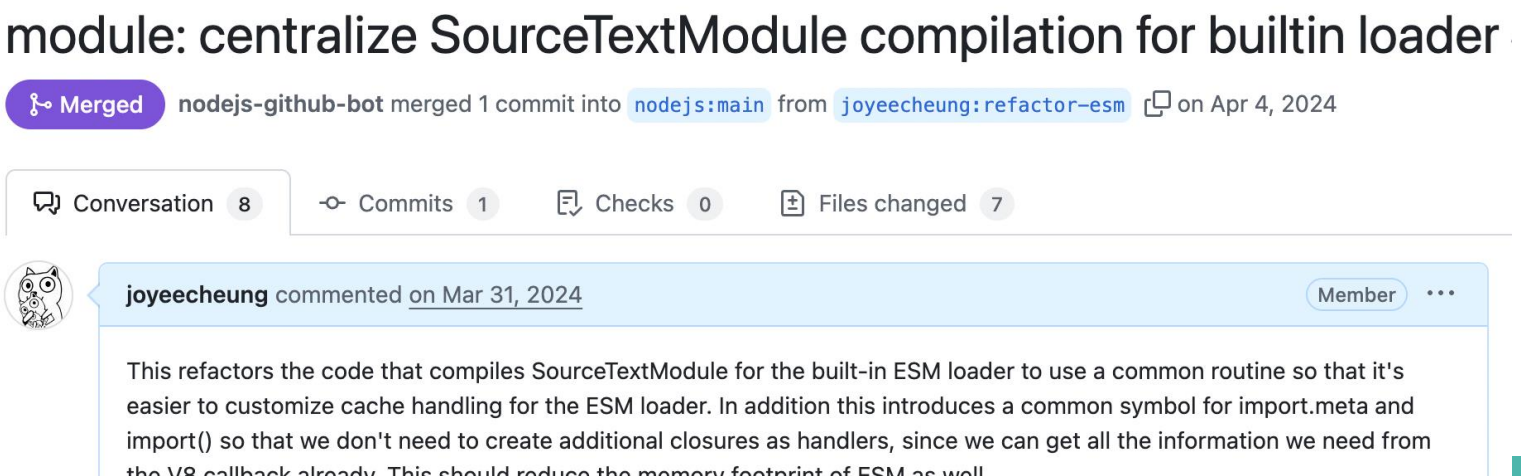
The module loaders, from my perspective

- Node.js internal built-in loader: 🏰🌿
  - 400+ JS, 900+ C++(LOC), nicely encapsulated
- Node.js user-land CommonJS loader: 🍯🐿️💣
  - 1000+ JS, 500+ C++ (LOC)
  - Spaghetti code, monkey patched everywhere
- Node.js user-land ESM loader: 😮🐟⚔️🗣️
  - **5000+ JS, 1000+ C++ (LOC)**
  - Big, convoluted code base, many early contributors no long active, past record of endless debates



Restarting require(esm) in Node.js

- Waited for other volunteers more familiar with ESM loader to refactor and carve out a synchronous path...
- Thought I only had to refactor the compilation part of the ESM loader to implement compile cache, ended up reading the entire thing...



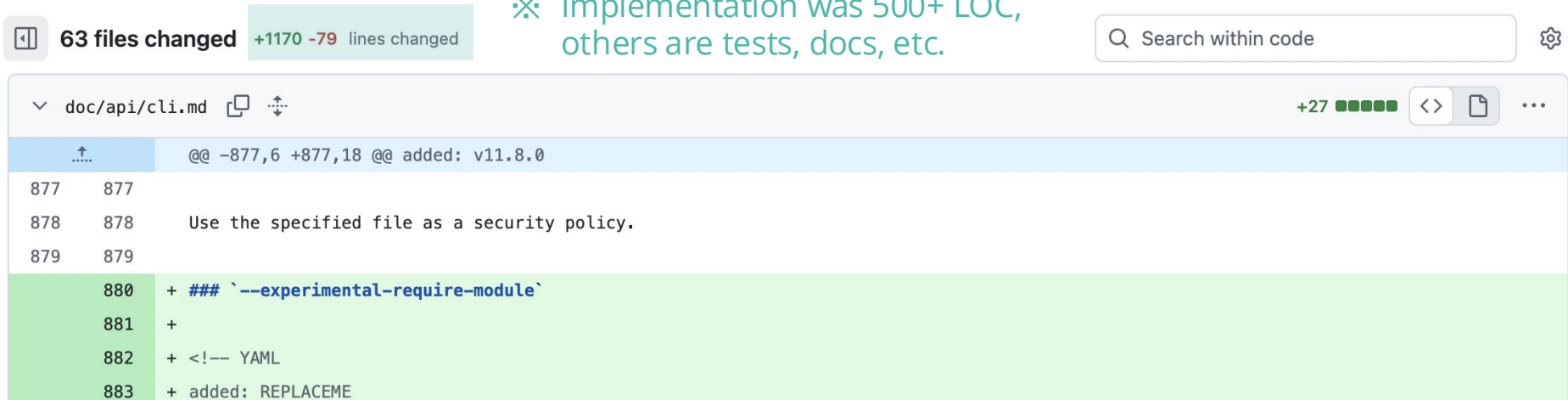
Restarting require(esm) in Node.js 

- Don’t refactor the entire ESM loader
- Write new lines to implement a synchronous and simplified ESM loading path for `require()`
- Lines added are easier to backport to older LTS than lines changed, anyway
- Implication: takes more effort to consolidate code paths later

Making require(esm) happen

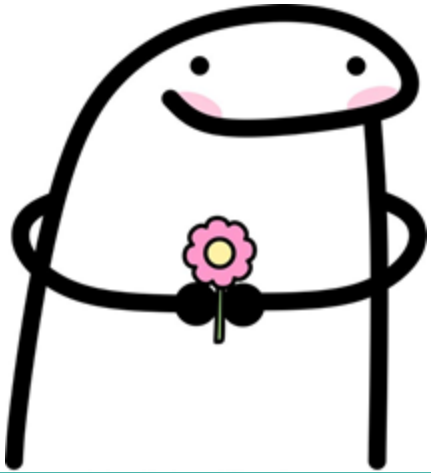
- (Re)started implementation in 2024, got support from Bloomberg
- Experimental release in 22.0.0, backported to 20
- Additional features and compat fixes to make it as non-breaking as possible
- Unflagged: ^20.19.0 || >=22.12.0

※ Implementation was 500+ LOC, others are tests, docs, etc.



require(esm) is here!

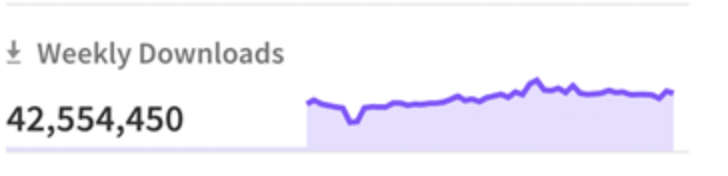
- All active LTS now supports require(esm)!
- Packages that do not support EOL Node.js versions can count on it now
- Many popular packages have started to drop dual and ship ESM-only after require(esm) was unflagged
  - Vite
  - Yargs
  - Babel
  - Storybook
  - Unjs packages..
  - Various tinylibs..
  - Various eslint plugins (h/t e18e)
  - ...



Making require(esm) customizable

pirates / lib / index.js

Code | Blame | 155 lines (138 loc) · 5.55 KB




```
87  function addHook(hook, opts = {}) {
112     loaders[ext] = Module._extensions[ext] = function newLoader(mod, filename) {
115         if (shouldCompile(filename, exts, matcher, ignoreNodeModules)) {
116           compile = mod._compile;
117           mod._compile = function _compile(code) {
118             // reset the compile immediately as otherwise we end up having the
119             // compile function being changed even though this loader might be reverted
120             // Not reverting it here leads to l
121             // addHook -> revert -> addHook ->
122             // The compile function is also any
123             mod._compile = compile;
124             const newCode = hook(code, filename
125             if (typeof newCode !== 'string') {
```

Many, many very popular packages rely on monkey patching to customize module loading

ESM loading is encapsulated and not monkey patchable

pirates / lib / index.js

Code | Blame | 155 lines (138 loc) · 5.55 KB

```
87    function addHook(hook, opts = {}) {
112       loaders[ext] = Module._extensions[ext] = function newLoader(mod, filename) {
115           if (shouldCompile(filename, exts, matcher, ignoreNodeModules)) {
116             compile = mod._compile;
117             mod._compile = function _compile(code) {
118               // reset the compile immediately as otherwise we end up having the
119               // compile function being changed ev
120               // Not reverting it here leads to lo
121               // addHook -> revert -> addHook -> r
122               // The compile function is also anyw
123               mod._compile = compile;
124               const newCode = hook(code, filename);
125               if (typeof newCode !== 'string') {
```

ESM dependencies inside this code won't be loaded through this monkey-patched path

Making require(esm) customizable

Opening more spots for monkey patching creates more vicious loops...what can we do about it?

Loader customization hooks: take 1

- `--experimental-loader` (experimental since v8.8.0 - 2017!)
- `module.register()` (experimental since v18.19.0)

```
// In register.mjs
module.register('./hooks.mjs', import.meta.url);

$ node --import ./register.mjs app.mjs
# Deprecated
$ node --experimental-loader ./hooks.mjs app.mjs
```

Loader customization hooks: take 1

```
// In hooks.mjs
export async function resolve(specifier, context, nextResolve) {
  if (specifier === 'my-custom-module')
    return { url: 'file:///path/to/custom-module.mjs', shortCircuit: true };
  return nextResolve(specifier, context);
}
export async function load(url, context, nextLoad) {
  const result = await nextLoad(url, context);
  const instrumentedSource = `console.log('instrumented');\n${result.source}`;
  return {  ...result, source: instrumentedSource };
}
// In register.mjs
module.register('./hooks.mjs', import.meta.url);
```

Customize resolution

Modify source code

Loader customization hooks: take 1

- Could not customize modules loaded through `require()` because one can't run async customization in synchronous `require()`
- What's the module loader hook for if it can only customize <~30% of the modules in the ecosystem?

~~Works for all modules~~

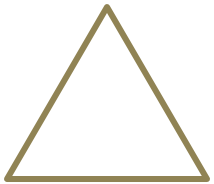
Async **In thread**

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

Loader customization hooks: take 2

- How do you run async customization from sync `require()`?
- **Answer**: run it on a different thread – breaking change in v20

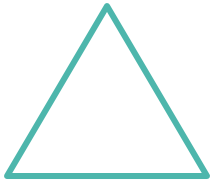
~~Works for all modules~~



Async **In thread**

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

Works for all modules*



Async ~~In thread~~

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

Loader customization hooks: take 2

specifier: “my-custom-module”

```
require('my-custom-module');
```

Main thread

Blocking using `Atomics.wait()`

```
export async function resolve(specifier, context,
                              nextResolve) {
  if (specifier === 'my-custom-module')
    return {
      url: 'file:///path/to/custom-module.mjs',
      shortCircuit: true
    };

  return nextResolve(specifier, context);
}
```

Loader hook worker thread

Loader customization hooks: take 2

```
export async function resolve(specifier, context,
                              nextResolve) {
  if (specifier === 'my-custom-module')
    return {
      url: 'file:///path/to/custom-module.mjs',
      shortCircuit: true
    };

  return nextResolve(specifier, context);
}
```

Loader hook worker thread

Main thread

Blocking using `Atomics.wait()`

Loader customization hooks: take 2

New problem: many existing hooks **need to run code on the main thread**

- Pass non-transferable objects e.g. functions between modules and hooks
- Mutate module exports using data prepared in the hooks

```
// The loader code run on a separate thread now.
globalThis.__instrument = {};
export async function load(url, context, nextLoad) {
 const result = await nextLoad(url, context);
  globalThis.__instrument[url] = () => { /* instrumentation code */ }
  // The module runs on the main thread and don't get the same globalThis.__instrument!
  const instrumentedSource =
    `globalThis.__instrument[${JSON.stringify(url)}]();\n${result.source}`;
  return {  ...result, source: instrumentedSource };
}
```

Problems from being off-thread

pirates / lib / index.js

Code | Blame | 155 lines (138 loc) · 5.55 KB

```
87   function addHook(hook, opts = {}) {
112      loaders[ext] = Module._extensions[ext] = function newLoader(mod, filename) {
115          if (shouldCompile(filename, exts, matcher, ignoreNodeModules)) {
116            compile = mod._compile;
117            mod._compile = function _c
118              // reset the compile imm
119              // compile function beir
120              // Not reverting it here
121              // addHook -> revert ->
122              // The compile function                                          sec
123              mod._compile = compile;
124              const newCode = hook(code, filename);
125              if (typeof newCode !== 'string') {
```

hook is a function, need to be run on the same thread the modules are run - how do you transfer it to another thread?

Loader customization hooks: take 2

- Mixing User messages and locks with Node.js internal ones: **prone to deadlocks**
- Inter-thread communication comes with a **performance overhead**

```
let portInLoaderThread; // Must use ports and locks to talk to the thread running the modules
export async function initialize({ port }) {
  portInLoaderThread = port;
}
export async function load(url, context, nextLoad) {
 const result = await nextLoad(url, context);
 if (portInLoaderThread) { portInLoaderThread.postMessage({ type: 'instrument', url });}
 const instrumentedSource =
   `portInMainThread.on('message', (msg) => { /* runs on main thread */ });\n${result.source}`;
  return { ...result, source: instrumentedSource };
}
```

Loader customization hooks: take 2

- The `require()` on the main thread, once customized to block on loader thread, have many **quirks** and do not work like normal `require()`

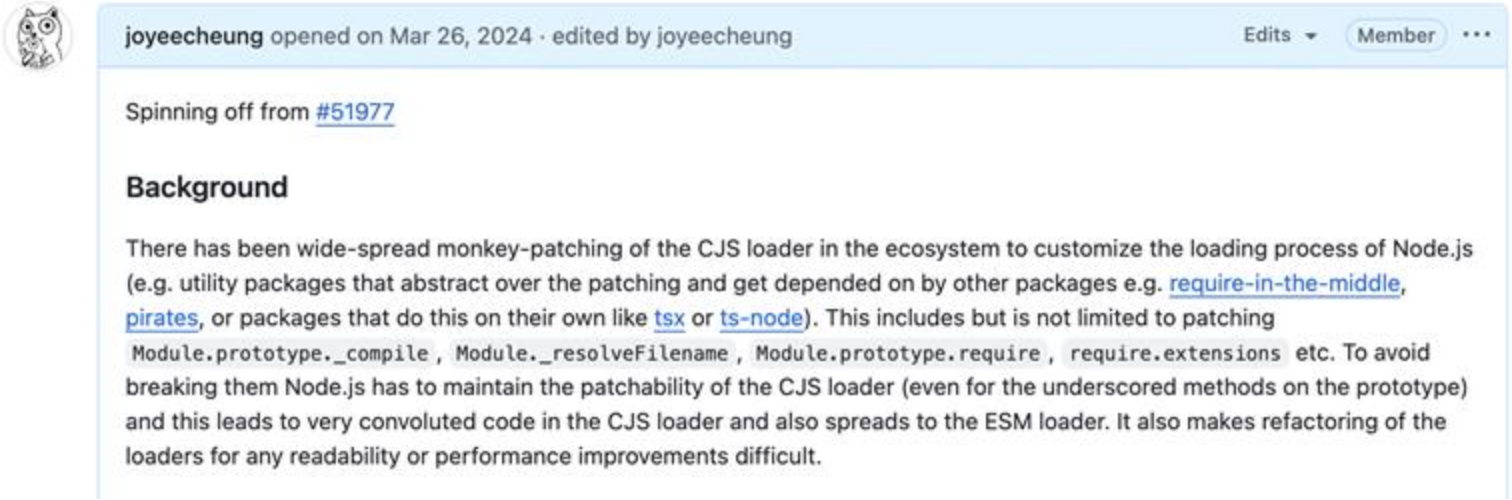
- Many users of `--experimental-loader` & `module.register()` ended up keeping both hooks and still monkey patching CommonJS when the entry point is CommonJS
    - Did not really reduce the monkey-patching in the wild

Loader customization hooks: take 3

Proposal for a simple, universal module loader hooks API to replace require() monkey-patching #52219

Closed

Just one more API…one more



joyeecheung opened on Mar 26, 2024 · edited by joyeecheung Edits Member

Spinning off from #51977

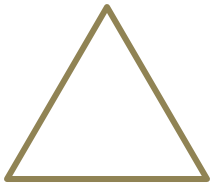
Background

There has been wide-spread monkey-patching of the CJS loader in the ecosystem to customize the loading process of Node.js (e.g. utility packages that abstract over the patching and get depended on by other packages e.g. require-in-the-middle, pirates, or packages that do this on their own like tsx or ts-node). This includes but is not limited to patching `Module.prototype._compile`, `Module._resolveFilename`, `Module.prototype.require`, `require.extensions` etc. To avoid breaking them Node.js has to maintain the patchability of the CJS loader (even for the underscored methods on the prototype) and this leads to very convoluted code in the CJS loader and also spreads to the ESM loader. It also makes refactoring of the loaders for any readability or performance improvements difficult.

Loader customization hooks: take 3

New API: Just provide a simpler API that allows customizing ESM/CJS/everything on the main thread **synchronously**

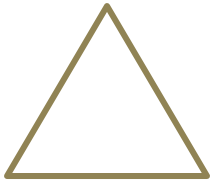
~~Works for all modules~~



Async **In thread**

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

Works for all modules*



Async ~~In thread~~

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

Works for all modules

~~Async~~ **In thread**

New API: `module.registerHooks()`

Loader customization hooks: take 3

```
// In hooks.mjs
export async function resolve(specifier, context, nextResolve) {
  if (specifier === 'my-custom-module')
    return { url: 'file:///path/to/custom-module.mjs', shortCircuit: true };
  return nextResolve(specifier, context);
}
export async function load(url, context, nextLoad) {
  const result = await nextLoad(url, context);
  const instrumentedSource = `console.log('instrumented');\n${result.source}`;
  return {  ...result, source: instrumentedSource };
}
// In register.mjs, run on a different thread
module.register('./hooks.mjs', import.meta.url);
```

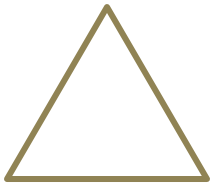
Loader customization hooks: take 3

```
// In register.mjs
                function resolve(specifier, context, nextResolve) {
  if (specifier === 'my-custom-module')
    return { url: 'file:///path/to/custom-module.mjs', shortCircuit: true };
  return nextResolve(specifier, context);
}
                function load(url, context, nextLoad) {
  const result =       nextLoad(url, context);
  const instrumentedSource = `console.log('instrumented');\n${result.source}`;
  return {  ...result, source: instrumentedSource };
}
// Same file because it's run on the same thread
module.registerHooks({ resolve, load });
```

Loader customization hooks: take 3

- Async handling is the least needed out of the three
- Many existing hooks are only doing work synchronously (because they also monkey-patch CommonJS)

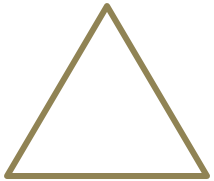
~~Works for all modules~~



Async ~~In thread~~

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

Works for all modules*



Async ~~In thread~~

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

Works for all modules

~~Async~~ **In thread**

New API:
`module.registerHooks()`

Loader customization hooks: take 3

babel / packages / babel-register / src / **worker-client.cts**

Code | Blame | 102 lines (80 loc) · 2.75 KB

```
38  class WorkerClient extends Client {
58    constructor() {
61        const subChannel = new WorkerClient.#worker_threads.MessageChannel();
62
63        this.#worker.postMessage(
64          { signal: this.#signal, port: subChannel.port1, action, payload },
65          [subChannel.port1],
66        );
67
68        Atomics.wait(this.#signal, 0, 0);
69        const { message } = WorkerClient.#worker_threads.receiveMessageOnPort(
70          subChannel.port2,
```

If they really need to do something async, users hook can and already do spawn their own worker thread to de-async

Loader customization hooks: take 3

babel / packages / babel-register / src / **worker-client.cts**

Code | Blame | 102 lines (80 loc) · 2.75 KB

```
38  class WorkerClient extends Client {
58    constructor() {
61      const subChannel = new WorkerClient.#worker_threads.MessageChannel();
62
63      this.#worker.postMessage(
64        { signal: this.#signal, port: subChannel.port1, action, payload },
65        [subChannel.port1],
66      );
67
68      Atomics.wait(this.#signal, 0, 0);
69      const { message } = WorkerClient.#worker_threads.receiveMessageOnPort(
70        subChannel.port2,
```

Giving hook authors more more control & not mixing the messages & locks in the de-async worker also helps avoiding deadlocks

Loader customization hooks: take 3

For hooks that do not need async handling, the new API is easier to use

pirates / lib / index.js

Code | Blame | 155 lines (138 loc) · 5.55 KB

```
87  function addHook(hook, opts = {}) {
112     loaders[ext] = Module._extensions[ext] =
115         if (shouldCompile(filename, exts, matcher, ignoreNodeModules)) {
116           compile = mod._compile;
117           mod._compile = function _compile(code)
118             // reset the compile immediately as
119             // compile function being changed even though this loader might be reverted
120             // Not reverting it here leads to long useless compile chains when doing
121             // addHook -> revert -> addHook -> revert -> ...
122             // The compile function is also anyway created new when the loader is called a second ti
123             mod._compile = compile;
124             const newCode = hook(code, filename);
```

Filter extensions (patching internals)

Register customizations (patching internals)

Invoke user hook function

Loader customization hooks

```
 8  function addHook(hook, options) {
 9    function load(url, context, nextLoad) {
10      const result = nextLoad(url, context);
11      const index = url.lastIndexOf('.');
12      const ext = url.slice(index);
13      if (!options.exts.includes(ext)) {
14        return result;
15      }
16      const filename = fileURLToPath(url);
17      if (!options.matcher(filename)) {
18        return result;
19      }
20      return { ...result, source: hook(result.source.toString(), filename) }
21    }
22
23    const registered = registerHooks({ load });
```

Filter extensions (using public API)

Invoke user hook function

Register customizations (public API)

Loader customization hooks

```
8  function addHook(hook, options) {
9    function load(url, context, nextLoad) {
10     const result = nextLoad(url, context);
11     const index = url.lastIndexOf('.');
12     const ext = url.slice(index);
13     if (!options.exts.includes(ext)) {
14       return result;
15     }
16     const filename = fileURLToPath(url);
17     if (!options.matcher(filename)) {
18       return result;
19     }
20     return { ...result, source: hook(result.source.toString(), filename) }
21   }
22
23   const registered = registerHooks({ load });
```

This is run on the same thread the modules are run, and works transparently with ESM

Filter extensions (using public API)

Invoke user hook function

Register customizations (public API)

Where are we now?

- Implemented and landed after a lot of refactoring (again)
- `module.registerHooks()` is available from v22
- Fixed most bugs based on user feedback, now mostly complete and has fewer bugs/caveats than `module.register()`

Where are we now?

- Consider migrating to `module.registerHooks()` from `--experimental-loader` / `module.register()` if/when you can drop support for older Node.js and the hook does not really need to be async
- The old APIs has many bugs/caveats that are not solvable by design and stabilization will be further away.
  - Also, async overhead in Node.js is non-trivial
- If you are still monkey patching CommonJS loader for some reason..migrate to `module.registerHooks()`

Thanks!