> 技术要点：CJS、ESM、Nodemon

前言

前端工程化 完全离不开 Node ，很多需求都基于 Node 完成。一般来说 Node开发 会使用 CJS 编码，但很多 Web开发 的同学已习惯使用 ESM 编码了。

其实无需担心， Web开发 转向 Node开发 完全可继续使用 ESM 编码。因为 Web 与 Node 都是一种 JS运行环境 ，本质上只有 运行环境 不同。

当前面临的核心问题是 Node 无法直接使用 ESM 编码。本章将带领你**部署Node的ESM开发环境**，无需关注 JS运行环境 涉及模块方案的差异性，让 Node开发 也变得像 Web开发 那样丝滑。

背景：模块化是什么

首先理解 前端工程化 中一个很重要的概念：**模块化**。一起聊聊 模块化 是什么以及 JS模块化 包括哪些解决方案。当然 CJS 与 ESM 这两种 模块规范 是重点回顾对象。

模块化

JS 诞生于 1995年 ，最初设计的目的是实现一些简单的浏览器交互效果，寥寥数语就能为用户提供良好的操作体验。随着 JS 的快速发展，各种前端技术得到广泛应用，特别是 AJAX 与 Jquery 引发的前端大革命让 JS 得到质的提升，但各种问题也接踵而至。

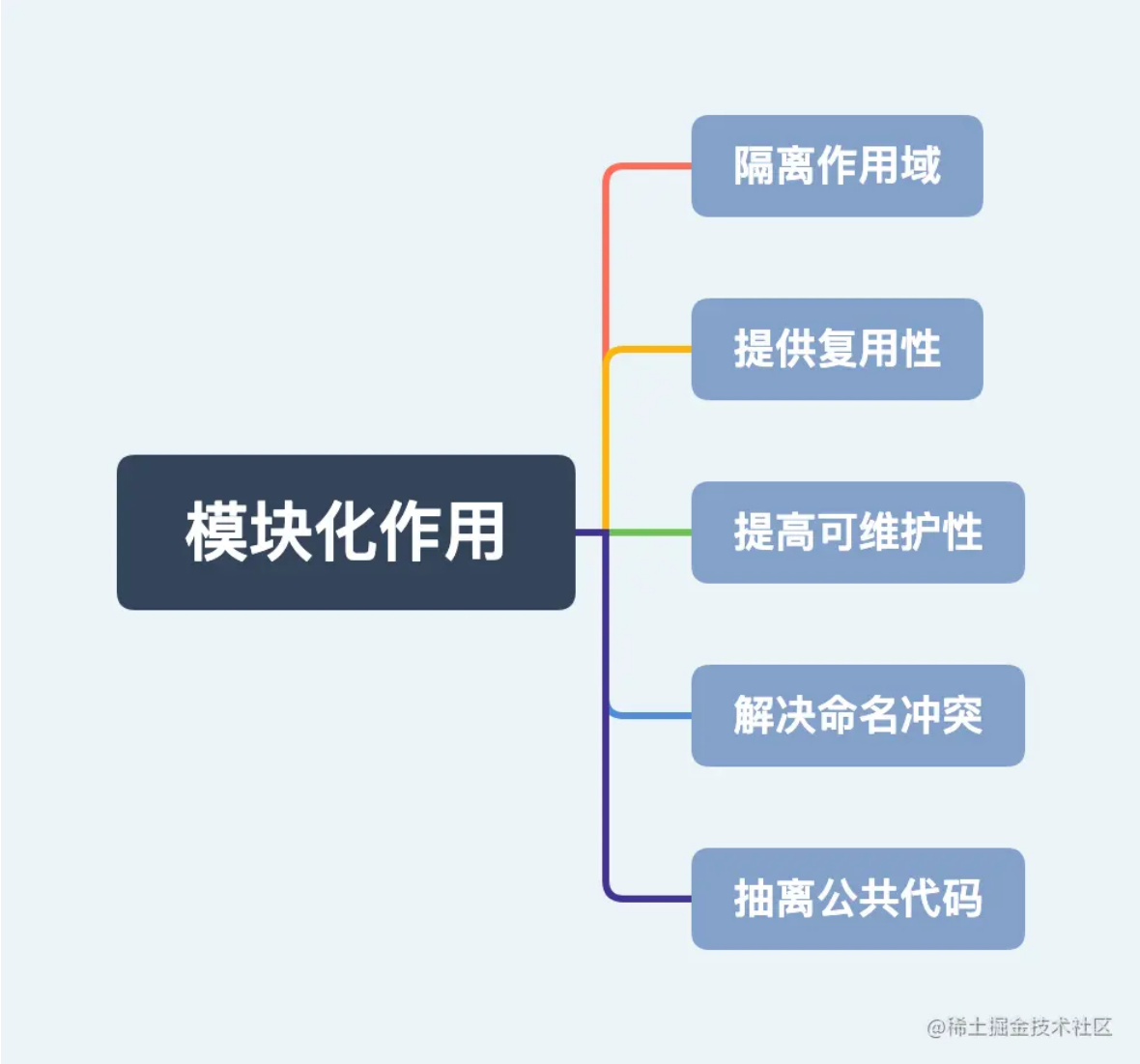
在实际开发时，经常会遇到 变量名称 或 函数名称 一样的情况。这不仅容易造成**命名冲突**，还会污染全局变量。若在应用特别复杂，存在大量相似代码，又引用很多第三方库的情况下，稍不注意就很易造成文件的**依赖混乱**。

基于此， JS 也引入 模块化 的概念。**早期的模块化不是真正的模块化，只是通过一些“骚操作”实现看似是模块化的效果**，例如 立即调用函数表达式 (简称 IIFE)就是一个在定义时可立即执行的函数，至于它如何实现 模块化 ，可查看MDN文档，在此不深入讲述了。**后期的模块化才算是真正的模块化**，它包括 CJS 、 AMD 、 CMD 、 UMD 和 ESM ，经过多年演变，目前 Web开发 倾向于 ESM ， Node开发 倾向于 CJS 。

模块化 让 JS 也能拥有自己的 模块化效果 ，在实际开发中，**一个模块就是一个文件。** 模块化 的核心包括以下特性，基本都是围绕如何处理文件(模块)。

- **拆分**：将代码根据功能拆分为多个可复用模块
- **加载**：通过指定方式加载模块并执行与输出模块
- **注入**：将一个模块的输出注入到另一个模块
- **管理**：因为工程模块数量众多需管理模块间的依赖关系

前端代码爆炸式增长必然会引入 前端工程化 解决问题， 模块化 作为 前端工程化 中最低成本的应用，很值得每位开发者遵守。使用 模块化 开发代码，不仅能提高代码整体可读性，也能增强项目整体维护性。不管是个人开发还是协作开发， 模块化 都能带来很多好处。



模块方案

以前引用 js文件 都会使用多个 <script> 顺序处理，曾见过一个老旧项目多达 12 个 <script> ，这会导致很多问题，例如以下情况。

- **请求过多**：每个 <script> 都有一个 src 必然会增加 HTTP 请求次数
- **依赖模糊**：每个 <script> 的摆放顺序都有可能影响前后脚本加载错误
- **难以维护**：每个 <script> 的变量命名与函数作用域都有可能互相影响

JS模块化 就很好地解决了上述问题，具体有哪些解决方案？在 JS 发展历程中，主要有六种常见模块方案，分别是 IIFE 、 CJS 、 AMD 、 CMD 、 UMD 和 ESM 。为了方便对比，通过下图展示它们各自的定义与特性，相信你有更好的理解。

分析每个模块方案的特性可知， 同步加载 包括 IIFE 与 CJS ， 异步加载 包括 AMD 、 CMD 和 ESM 。浏览器可兼容 IIFE 与 AMD ，服务器可兼容 CJS ，浏览器与服务器都兼容 CMD 、 UMD 和 ESM 。

目前只需关注 CJS 与 ESM ，那它们到底有何细微不同?

-	CJS	ESM
语法类型	动态	静态
关键声明	require	export 与 import
加载方式	运行时加载	编译时加载
加载行为	同步加载	异步加载
书写位置	任何位置	顶层位置
指针指向	this 指向 当前模块	this 指向 undefined
执行顺序	首次引用时 加载模块 再次引用时 读取缓存	引用时生成 只读引用 执行时才是正式取值
属性引用	基本类型属于 复制不共享 引用类型属于 浅拷贝且共享	所有类型属于 动态只读引用
属性改动	工作空间可修改引用的值	工作空间不可修改引用的值 但可通过引用的方法修改

Node开发 习惯使用 CJS 编码，但本章将改用 ESM 编码，因此在后续编码时可能会出现一些难以解释的问题，例如 循环引用 与 前置引用 ，而了解它们的加载方式与行为可帮助你理解这些问题。

- **运行时加载**指整体加载模块生成一个对象，再从对象中获取所需的属性方法去加载。最大特性是 全部加载 ，只有运行时才能得到该对象，无法在编译时做静态优化。
- **编译时加载**指直接从模块中获取所需的属性方法去加载。最大特性是 按需加载 ，在编译时就完成模块加载，效率比其他方案高，无法引用模块本身(本身不是对象)，但可拓展 JS 高级语法(宏与类型校验)。

上述介绍了好几种模块方案，那 前端工程化 都会用到吗？当然不会，可能只需使用 CJS 或 ESM ，而目前只需使用 ESM 编码，因此接着重点关注 ESM 。

现状：ESM能否在Node环境中运行

随着主流浏览器逐步支持 ESM ，越来越多目光投注于 Node 对于 ESM 的支持上。目前 Node 使用 CJS 作为官方模块方案，虽然内置模块方案促进 Node 的流行，但也阻碍了 ESM 的引入，这一点相信接触过 Node开发 的同学一定深有感触。

原生支持ESM

2017年10月31日 ， Node 发布了 v8.9.0 ，从此只要在命令中加上 --experimental-modules ， Node 就可象征性地支持 ESM 了。

```
node --experimental-modules index.js
```

但 低版本 Node 依然无法直接支持 ESM 解析，还需在运行环境中 “做些手脚” 才行。

接着 Node 发布了 v13.2.0 带来一些新特性，正式取消 --experimental-modules 启动参数。当然并不是删除 --experimental-modules ，而是在其原有基础上实现对 ESM 的实验性支持并默认启动。

--experimental-modules 特性包括以下方面。

- 使用 type 指定模块方案
  - 在 package.json 中指定 type 为 commonjs ，则使用 CJS
  - 在 package.json 中指定 type 为 module ，则使用 ESM
- 使用 --input-type 指定入口文件的模块方案，与 type 一样
  - 命令中加上 --input-type=commonjs ，则使用 CJS
  - 命令中加上 --input-type=module ，则使用 ESM
- 支持新文件后缀 .cjs
  - 文件后缀使用 .cjs ，则使用 CJS
- 使用 --es-module-specifier-resolution 指定文件名称引用方式
  - 命令中加上 --es-module-specifier-resolution=explicit ，则引用模块时必须使用文件后缀(默认)

    - 命令中加上 `--es-module-specifier-resolution=node` ，则引用模块时无需使用文件后缀
- 使用 `main` 根据 `type` 指定模块方案加载文件
    - 在 `package.json` 中指定 `mian` 后会根据 `type` 指定模块方案加载文件

CJS/ESM判断方式

`Node` 要求使用 `ESM` 的文件采用 `.mjs` 后缀，只要文件中存在 `import/export命令` 就必须使用 `.mjs` 后缀。若不希望修改文件后缀，可在 `package.json` 中指定 `type` 为 `module` 。基于此，若其他文件使用 `CJS` ，就需将其文件后缀改成 `.cjs` 。若在 `package.json` 中未指定 `type` 或指定 `type` 为 `commonjs` ，则以 `.js` 为后缀的文件会被解析为 `CJS` 。

简而言之， `mjs文件` 使用 `ESM` 解析， `cjs文件` 使用 `CJS` 解析， `js文件` 使用基于 `package.json` 指定的 `type` 解析(`type=commonjs` 使用 `CJS` ， `type=module` 使用 `ESM`)。

当然还可通过命令参数处理，不过我认为这样做操作过多，所以就不讨论具体方法了。

刚才说了 `Node v13.2.0` 在默认情况下，会启动对 `ESM` 的实验支持，无需在命令中加上 `--experimental-modules` 参数。那 `Node` 是如何区分 `CJS` 与 `ESM` ？ 简而言之， `Node` 会将以下情况视为 `ESM` 。

- [x] 文件后缀为 `.mjs`
- [x] 文件后缀为 `.js` 且在 `package.json` 中指定 `type` 为 `module`
- [x] 命令中加上 `--input-type=module`
- [x] 命令中加上 `--eval cmd`

方案：部署Node的ESM开发环境

虽然官方文档有明确的迁移方案，但很多同学还是会存在理解偏差。确实，一连串的操作会让很多未接触或很少接触 `Node开发` 的同学感觉无比混乱。为了愉快地部署 `Node` 的 `ESM开发环境` ，将实现高低两种版本的部署，规范好每种方案的实现方式，再根据自己喜好选择。

将 `Node v13.2.0` 作为高低版本分界线，当版本 `>=13.2.0` 则定为高版本，当版本 `<13.2.0` 则定为低版本。高版本使用 `Node原生部署方案` ，低版本使用 `Node编译部署方`

案。

在部署 Node 的 ESM开发环境 前需初始一个示例，以下所有代码都基于该项目进行。

在根目录中创建 package.json 并执行 npm i 安装项目依赖。

```json
{
        "name": "node-esm",
        "version": "1.0.0",
        "main": "src/index.js",
        "scripts": {
                "start": "node src/index.js"
        },
        "dependencies": {
                "@yangzw/bruce-us": "1.0.3"
        }
}
```

创建 src/index.js 文件，加入以下内容。示例引用我开源的@yangzw/bruce-us，其中 NodeType() 用于获取 Node 相关信息。

```js
import { NodeType } from "@yangzw/bruce-us/dist/node";

console.log(NodeType());
```

这一步完成就需分情况讨论了。

Node原生部署方案

假设 Node 是 v13.2.0 以上版本，执行 npm start ，输出以下信息表示运行失败。

```
(node:56155) Warning: To load an ES module, set "type": "module" in the package.json or
(Use `node --trace-warnings ...` to show where the warning was created)
/Users/.../node-esm/for-node/src/index.js:1
import { NodeType } from "@yangzw/bruce-us/dist/node";
^^^^^^

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

这是因为 Node 根据刚才提到的四种情况也无法识别出是 ESM，知道原因很快就能找出解决方案了。在保持所有文件后缀为 .js 的前提下，在 package.json 中指定 type 为 module 。

为何一定要保持所有文件后缀为 .js ？作为 前端开发者 且使用 JS 编码，那保证文件后缀为 .js 再正常不过，不想出幺蛾子就不要乱改文件后缀。

为了让 Node 支持 ESM，还需为其指定 Node/Npm 版本限制。这是为了避免预设与实际情况不同而报错，例如预设该项目在高版本运行，实际却在低版本运行。

Node 与 Npm 是成双成对地安装，可通过Node Releases查询到 Node v13.2.0 对应 Npm v6.13.1 。

```json
{
        "type": "module",
        "engines": {
                "node": ">=13.2.0",
                "npm": ">=6.13.1"
        }
}
```

重新执行 npm start，输出以下信息表示运行失败。

```
internal/process/esm_loader.js:74
internalBinding("errors").triggerUncaughtException(

Error [ERR_MODULE_NOT_FOUND]: Cannot find module ...
Did you mean to import @yangzw/bruce-us/dist/node.js?
```

根据报错提示，可知模块路径不存在，这主要是因为**显式文件名称**使用不对。

首先 高版本Node 在默认情况下，对 import命令 的文件后缀存在强制性，因此 import "./file" 并不等于 import ./file.js 。其次 CJS 的自动后缀处理行为可通过 --es-module-specifier-resolution=node 开启，但模块主入口并不会受到 ESM 的影响，例如 import Path from "path" 照样可正常运行。在命令中加上 --es-module-specifier-resolution=node 就能解决显示文件名称的问题。

```json
{
        "scripts": {
                "start": "node --es-module-specifier-resolution=node src/index.js"
```

```
        }
}
```

为何这样设计显式文件名称？这主要是想通过 Node 提供的通用解决方案鼓励开发者编写 Web 与 Node 共享的代码。

重新修改文件名称后再执行 npm start ，输出以下信息运行成功，这次就无任何问题了！

```
{
        nodeVs: "16.14.0",
        npmVs: "8.3.1",
        system: "macos",
        systemVs: "19.6.0"
}
```

Node 使用 ESM 除了上述问题外，还存在一些特别差异。特别是 ESM 不再提供 Node 某些特性与不能灵活引用 json文件 了，因此 __dirname 、 __filename 、 require 、 module 和 exports 这几个特性将无法使用。

不过，上帝关闭一扇门的同时还会打开另一扇窗，可采用以下方式解决这些问题。

- __filename 与 __dirname 可用 import.meta 对象重建
- require 、 module 和 exports 可用 import 与 export 代替
- json文件 的引用可用 Fs模块 的 readFileSync 与 JSON.parse() 代替

```js
import { readFileSync } from "fs";
import { dirname } from "path";
import { fileURLToPath } from "url";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
console.log(__filename, __dirname);

const json = readFileSync("./info.json");
const info = JSON.parse(json);
```

CJS 的循环依赖关系已通过缓存各个模块的 module.exports 对象解决，但 ESM 用了所谓的绑定。简而言之， ESM 模块不会导出导入值而是引用值。

- **导入引用模块可访问该引用但无法修改它。**

- **导出引用模块可为引用该模块的模块重新分配值且该值由导入引用模块使用**

而 CJS 允许在任何时间点将引用分配给模块的 module.exports 对象，让这些改动仅部分反映在其他模块。

Node编译部署方案

Npm 很多模块都使用 CJS 编码，因为同时使用 require 与 export/import 会报错，所以单个模块无法切换到 ESM 。

可用 babel 将代码从 ESM 转换为 CJS ，因此使用 babel 编译 ESM 代码是低版本 Node 支持 ESM 最稳定的方案无之一。在 Node v8.9.0 前的版本无法使用 --experimental-modules 支持 ESM ，也就更需 babel 解决该问题了。

当然在任何版本中， babel 都能让新语法转换为与旧环境兼容的代码，因此在 高版本Node 中也同样适用。

接着在 v13.2.0 以下版本中部署。执行以下命令安装 babel 相关工具链到 devDependencies 。

```
npm i @babel/cli @babel/core @babel/node @babel/preset-env -D
```

这四个 babel 子包很重要， Node 能不能支持 ESM 的解析就看它们了。

- **@babel/cli**：提供支持 @babel/core 的命令运行环境
- **@babel/core**：提供转译函数
- **@babel/node**：提供支持 ESM 的命令运行环境
- **@babel/preset-env**：提供预设语法转换集成环境

安装完毕，在 package.json 中指定 babel 相关配置，将 start 命令中的 node 替换为 babel-node 。

```json
{
        "scripts": {
                "start": "babel-node src/index.js"
        },
        "babel": {
                "presets": [
                        "@babel/preset-env"
```

```
                ]
        }
}
```

执行 `npm start` ，输出以下信息表示运行成功，到此整体坑位几乎为零。

```
{
        nodeVs: "12.22.10",
        npmVs: "6.14.16",
        system: "macos",
        systemVs: "19.6.0"
}
```

当然该方案无需在 `package.json` 中指定 `engines` ，毕竟其目的还是将代码的模块方案从 `ESM` 转换为 `CJS` 。若需兼容 `更低版本Node` ，可在 `package.json` 中指定 `babel` 的 `targets` 。

json

```
{
        "babel": {
                "presets": [
                        ["@babel/preset-env", { "targets": { "node": "8.0.0" } }]
                ]
        }
}
```

监听脚本自动重启命令

每次修改脚本都需重启命令才能让脚本内容生效，这太麻烦了，所以我始终喜欢在 `Node` 中使用 `nodemon` 。**nodemon**是一个自动检测项目文件发生变化就重启服务的 `Npm模块` ，是 `Node开发` 的必备工具。

以**Node编译部署方案**的示例为例。执行 `npm i -D nodemon` 安装 `nodemon` ，在 `package.json` 中指定 `nodemonConfig` 相关配置，将 `start` 命令替换为 `nodemon -x babel-node src/index.js` 。

json

```
{
        "nodemonConfig": {
                "env": {
                        "NODE_ENV": "dev"
                },
```

```
                "execMap": {
                        "js": "node --harmony"
                },
                "ext": "js json",
                "ignore": [
                        "dist/"
                ],
                "watch": [
                        "src/"
                ]
        }
}
```

修改 `src/index.js` 内容， `nodemon` 就能快速响应改动并重启命令。 `nodemon配置` 可查看Nodemon官网，在此不深入讲述了。

总结

很多同学接触 `Node开发` 都会使用 `CJS` 编码，但通过本章学习，相信能在 `Node环境` 中使用 `ESM` 编码有一个更充分的实践。所有 `Node项目` 都能通过上述探讨的解决方案改造为 `ESM` 形式，其余改动可能就是业务代码中 `require()` 与 `import/export` 的转换了。

每种解决方案都可能有自身的局限，我根据以往开发经验对上述解决方案做一个应用场景的总结。考虑到实用性与稳定性，最终还是推荐使用第三种解决方案：**监听脚本自动重启命令。**

- 基于**Node原生部署方案**改造 `Node项目` ，适合在 `高版本Node环境` 中使用，点击查看源码
- 基于**Node编译部署方案**改造 `Node项目` ，适合在 `低版本Node环境` 或 `任何版本Node环境` 中使用，点击查看源码
- 基于**监听脚本自动重启命令**改造 `Node项目` ，与 `Node编译部署方案` 一样的使用条件，中使用，点击查看源码

从 `前端工程化` 的角度来看，设计任何解决方案都要准备一些备用方案甚至是兜底方案，这些方案必须纳入考虑范围并使其具备可行性。

本章内容到此为止，希望能对你有所启发，欢迎你把自己的学习心得打到评论区！

- 示例项目：fe-engineering
- 正式项目：bruce

留言

输入评论（Enter换行，Ctrl + Enter发送）

发表评论

全部评论（51）

Samaritan JY.3 前端 24天前

基于监听脚本自动重启命令改造Node项目，index文件引入是省略.js还是会报错怎么解决呀

点赞 回复

lilywhite JY.3 1月前

使用main根据type指定模块方案加载文件
在package.json中指定mian后会根据type指定模块方案加载文件

指定mian中【mian】好像错了?

点赞 回复

qingkooo JY.3 前端开发 1月前

"若需兼容更低版本Node，可在package.json中指定babel的targets。"
请问大家，要版本低到什么程度才写该配置呢？node v8及以下吗?

1 1

qingkooo 1月前

官方没有找到低版本nodejs需要指定targets的文档，请老师明示出处吧。 babeljs.io

点赞 回复

卡尔好好玩 JY.3 前端 @ 兴盛优选 1月前

请教一个问题。在对一个老项目进行 cjs 转 esm 的过程，发现有一些开源的模块是 cjs 的方式，这就导致整个项目无法进行 esm 的转换。这种怎么办呢？提issue让作者迭代出 esm 的版本么？

点赞 1

qingkooo 1月前

那就采用Node编译部署方案喽

点赞 回复

伽蓝 JY.3 前端开发 2月前

CJS的循环依赖关系已通过缓存各个模块的module.exports对象解决，但ESM用了所谓的绑定。简而言之，ESM模块不会导出导入值而是引用值。

导入引用模块可访问该引用但无法修改它。
导出引用模块可为引用该模块的模块重新分配值且该值由导入引用模块使用
而CJS允许在任何时间点将引用分配给模块的module.exports对象，让这些改动仅部分...

展开

2 回复

Quintus Peng JY.3 2月前

老师，有没有ts+ node的加餐啊

点赞 回复

前端胖子 JY.2 web前端 2月前

"readFileSync与JSON.parse()代替"这段，后面的示例代码，直接readFileSync("./info.json")是会报错的。
应该在path模块上再导入一个join, 然后使用readFileSync(join(__dirname, "./info.json"))。

点赞 2

静听花亦落 2月前

老哥很棒，怪不得我老报错

点赞 回复

JowayYoung （作者） 2月前

是的，我的代码示例简化了这个内容，后续我补充上去

点赞 回复

前端胖子 JY.2 web前端 2月前

(node:12888) ExperimentalWarning: The ESM module loader is experimental.
file:///D:/fe-engineering/node-esm/for-node/src/index.js:1
import { NodeType } from "@yangzw/bruce-us/dist/node";
^^^^^^^^

SyntaxError: The requested module '@yangzw/bruce-us/dist/node' does not ...

展开

1 2

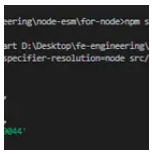
前端胖子 2月前

Node原生部署方案最后一步出现上面错误了，就是Node版本问题了。原文说“假设Node是v13.2.0以上版本”，我就用nvm装了一个这个版本的包，但最后一步会报上面错误。
试了一下node版本>=14.13.0才可以

点赞 回复

前端胖子 2月前

。



点赞 回复

hovey JY.3 前端开发 3月前

要是有typescript的部分就更好了

点赞 4

JowayYoung （作者） 3月前

感觉在Node中写TS多此一举，毕竟不像浏览器那样要编辑才能运行，有Eslint就行了

点赞 回复

斯蒂芬大狗 回复 JowayYoung 3月前

这个不太理解，可以详细说一下吗？node 写 TS 多此一举与浏览器对比的主要差别在哪里？

“感觉在Node中写TS多此一举，毕竟不像浏览器那样要编辑才能运行，...”

点赞 回复

查看更多回复

codermao JY.4 web实习 @ 小红书 3月前

打卡打卡

点赞 回复

isea JY.3 4月前

“浏览器只兼容IIFE与AMD，服务器只兼容CJS，浏览器与服务器都兼容CMD、UMD和ESM。”这句话表述上有问题吧?

1 1

JowayYoung （作者） 4月前

修复表述了

1 回复

梨木 JY.3 4月前

"scripts": {
"start": "node --es-module-specifier-resolution=node src/index.js"
}
配置完这一步：
运行 npm start
还是报这个错误...

展开

点赞 2

JowayYoung （作者） 4月前

要改成from '@yangzw/bruce-us/dist/node.js';

默认是指向/dist/web.js，现在是在Node环境中运行，所以要指向/dist/node.js

1 回复

多肉小子 回复 JowayYoung 3月前

那为什么不直接指向dist/node，而是指向dist/web呢?

“要改成from '@yangzw/bruce-us/dist/node.js';... ”

点赞 回复

小萌新入场 JY.4 IT 4月前

import { readFileSync } from 'fs'
const json = readFileSync('./info.json')
const info = JSON.parse(json)
console.log(info)
这里怎么打印报错？ type: module🙄

1 3

JowayYoung （作者） 4月前

package.json的type有设置为module吗

点赞 回复

小萌新入场 回复 JowayYoung 4月前

设置了的，不知道啥原因，可能是node 版本问题吧

“package.json的type有设置为module吗”

点赞 回复

查看更多回复

furfurJiang JY.4 前端 @ 程序江 4月前

模块总结图和表做得很棒，看过内容最全的了

点赞 1

JowayYoung （作者） 4月前

谢谢支持

1 回复

HING JY.3 前端 4月前

在把自己的项目改造成esm时，postcss-loader报错，Error [ERR_REQUIRE_ESM]: Must use import to load ES Module， 后面我把postcss.config.js后缀改成.cjs就好了

2 1

JowayYoung （作者） 4月前

是的，目前还有部分的工具的配置文件xxx.config.js都不能改成ESM，我统计了下常用的几个工具：
可兼容ESM：webpack、rollup
不兼容ESM，只能将文件后缀改为cjs：postcss、babel、stylelint、eslint

点赞 回复

前端小鹿 JY.2 4月前

总结的好啊

点赞 1

JowayYoung （作者） 4月前

谢谢支持

点赞 回复

aComputerGeek JY.4 逗比一枚~ 4月前

从css的文章开始， 就感觉作者的总结能力相当厉害， 不知道能不能出一个 你学习总结的小册或分享

2 1

JowayYoung （作者） 4月前

有机会出一个视频或者文章哈

1 回复

空城以北 JY.3 web前端工程师 @ 百腾... 4月前

请问有了vuecli还需要模块化吗

点赞 1

JowayYoung （作者） 4月前

需要的

点赞 回复

小白兔学前端 JY.5 小白兔 @ 国家富强 4月前

太帅了，我点了

点赞 回复

大沙杯 JY.3 蠢蛋一个 4月前

拼写 错误npm i -D nodemen ->npm i -D nodemon

1 1

JowayYoung （作者） 4月前

修复了

点赞 回复

查看全部 51 条回复