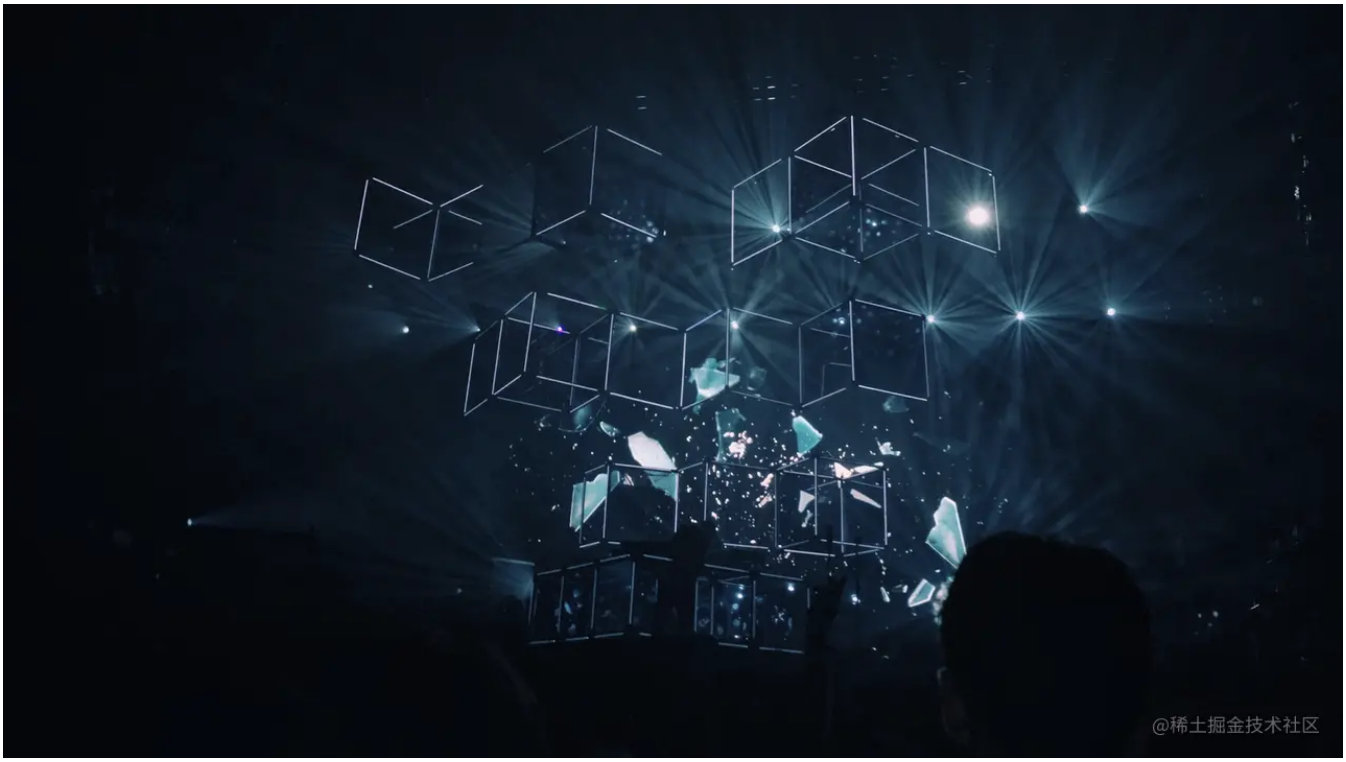
Node 最新 Module 导入导出规范

字节大力智能技术团队
2021年06月10日 11:48 · 阅读 981



字节大力智能 Lv3 研发 @ 字节跳动

> 作者：大力智能技术团队-前端 ademily

前言

Node.js 包模块最新规范提供了避免暴露特定文件、API 的能力，利于代码规范和仓库依赖治理。大力智能前端团队同学便对 Node 包模块最新规则进行了详尽的研究。本文就为大家带来最新的 Node 包模块导入导出规则，希望大家以后在工作中能够得心应手得处理这类情况。

> 原文：nodejs.org/api/package...

包模块历史变更：

版本	变更
v14.13.0	新增 `"exports"` 模式（ `"exports"` patterns）
v14.6.0	新增 `"imports"` 字段
v13.7.0、v12.16.0	条件导出不再使用特殊标识（Unflag conditional exports）
v13.6.0、v12.16.0	通过名称自引用不再使用特殊标识（Unflag self-referencing a package using its name.）
v12.7.0	引入 `"exports"` 这个 `package.json` 新字段，作为经典字段 `"main"` 的更强大的代替
v12.0.0	新增 通过 `package.json` 中 `"type"` 字段，使得 `.js` 文件会被视为 ES模块。

一. 介绍

包（package）是一个由 `package.json` 描述的文件夹树。包由包含 `package.json` 的文件夹和所有子文件夹组成，直到下一个含另一个 `package.json` 的文件夹，或者 `node_modules` 文件夹。

本页为大家介绍如何书写 `package.json` 。页面底部还有 Node.js 定义的 `package.json` 字段供参考。

二. 判断模块系统

当作为初始输入传递给 `node` ，或在ES模块代码中被 `import` 语句引用时，Node.js 会将以下内容视为 ESM 模块：

- 以 `.mjs` 结尾的文件
- 以 `.js` 结尾的文件，且最近的父 `package.json` 的顶层 `"type"` 值为 `"module"`
- 将标志为 `--input-type=module` 的字串，作为 `--eval` 的参数传入或通过 `STDIN` 传入 `node`

Node.js 会将所有其他形式的输入视为 CommonJS，例如 `.js` 文件们最近的父

留 CommonJS 模式，是为了向后兼容。虽然目前 Node.js 对 CommonJS 和 ESM 模块 都支持，但我们最好尽可能地明确使用哪种。

当以下情况作为初始输入启动 `node`，或在 ESM 模块代码中的使用 `import` 语句引用时，Node.js 会将其视为 CommonJS：

- 以 `.cjs` 结尾的文件
- 以 `.js` 结尾的文件，且最近的父 `package.json` 中顶层字段 `"type"` 值为 `"commonjs"`
- 将标志 `--input-type=commonjs` 作为 `--eval` 或 `--print` 的参数，或通过 `STDIN` 传递到 `node`

即使在所有源码都是 CommonJS 的情况下，包作者也应该在 `package.json` 中包含 `"type"` 字段。明确包的 `"type"`，可以在 Node.js 的默认类型发生变化的情况下对包进行未来的保护，而且也可以让构建工具和 loaders 更容易明确如何解析包内的文件。

1. package.json 和文件扩展名

在一个包中，`package.json` 的 `"type"` 字段定义了 Node.js 应该如何解析 `.js` 文件。如果 `package.json` 没有 `"type"` 字段，那么 `.js` 文件将被视为 CommonJS 模块。

当 `package.json` 的 `"type"` 值为 `"module"`，则 Node.js 会将该包中的 `.js` 文件解析为使用 ESM 模块语法。

`"type"` 字段不仅适用于初始化入口文件（`node my-app.js`），也适用于 `import` 声明和 `import()` 表达式所引用的文件。

复制代码

```
// my-app.js 被当做ES模块，因为 package.json中 "type" 为 "module"

import './startup/init.js'。
// 由于 ./startup 不包含 package.json 文件，因此继承了上一级的 "type" 值，所以作为 ES模块加载

import 'commonjs-package';
// 由于./node_modules/commonjs-package/package.json 缺乏 "type"字段或包含 `"type":"commonjs"`，所以该

import './node_modules/commonjs-package/index.js'。
// 由于./node_modules/commonjs-package/package.json 缺乏 "type"字段或包含 `"type":"commonjs"`，所以该
```

以 `.mjs` 结尾的文件总是作为 ESM 模块加载，且不受最近的父 `package.json` 的影响。

同样的，以 `.cjs` 结尾的文件总是作为 CommonJS 加载，且不受最近的父 `package.json` 的影响。

复制代码

```
import './legacy-file.cjs'。
// 以 CommonJS 的形式加载，因为 .cjs 总是以 CommonJS 的形式加载

import 'commonjs-package/src/index.mjs';
//以 ES模块加载，因为 .mjs 总是作为 ES 模块形式进行加载
```

`.mjs` 和 `.cjs` 扩展名能让我们在一个包中使用两种模式：

- 在一个 `"type": "module"` 的包中，Node.js 可以将后缀为 `.cjs` 的特定文件解释为 CommonJS（因为在 `"module"` 包中， `.js` 和 `.mjs` 文件都被视为 ES模块）。
- 在一个 `"type": "commonjs"` 的包中，Node.js 可以将后缀为 `.mjs` 的特定文件解释为 ES 模块（因为在 `"commonjs"` 包中， `.js` 和 `.cjs` 文件都被视为 CommonJS）。

2. --input-type 标志

当字串作为 `--eval` （或 `-e` ）参数或者通过 `STDIN` 传递到 `node` 时，一旦设置了 `--input-type=module` 标志，那么这些字串会被视为 ESM 模块。

复制代码

```
node --input-type=module --eval "import { sep } from 'path'; console.log(sep);"

echo "import { sep } from 'path'; console.log(sep);" | node --input-type=module
```

同样的，我们还有 `--input-type=commonjs` 用于显式地将字符串输入作为 CommonJS 运行。如果 `--input-type` 没有被指定，默认为 CommonJS 模式。

三. 包入口（Package entry points）

在 `package.json` 文件中，有两个字段可以定义包入口： `"main"` 和 `"exports"` 。所有版本的 Node.js 都支持 `"main"` 字段，但它的功能是有限的：它只定义包的主入口。

`"exports"` 字段算是 `"main"` 的替代品，它既可以定义包的主入口，又封闭了包，**防止其他未**

如果同时定义了 "exports" 和 "main"， "exports" 字段优先于 "main"。 "exports" 并不是 ESM 模块 或 CommonJS 所特有的； "exports" 如果存在的话，就会覆盖 "main" 。因此， **"main"** **不能作为 CommonJS 的降级回落，但它可以作为不支持 **"exports"** 字段的 Node.js 旧版本的降级回落。**

在 "exports" 中使用"条件导出"（Conditional exports）可以为每个环境定义不同入口，包括包是通过 require 还是 import 来引用。关于如何在一个包中同时支持 CommonJS 和 ESM 模块，请参考 "双 CommonJS/ESM 模块包"部分。

注意：使用 "exports" 字段可以防止包的使用者使用其他未定义的入口点，包括 package.json （例如： require('your-package/package.json') 。**这很可能是一个重大变更。**

为了使 "exports" 的引入不具有破坏性，请确保之前支持的每个入口都被导出。最好明确指定各个入口，这样包的就有了明确的公共API定义。例如，一个项目如果之前导出了 main 、 lib 、 feature 和 package.json ，那么 package.exports 可以使用如下写法：

复制代码

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/index": "./lib/index.js",
    "./lib/index.js": "./lib/index.js",
    "./feature": "./feature/index.js",
    "./feature/index.js": "./feature/index.js",
    "./package.json": "./package.json"
  }
}
```

另外，一个项目可以选择导出整个文件夹：

复制代码

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/*": "./lib/*.js",
    "./feature": "./feature/index.js",
    "./feature/*": "./feature/*.js",
    "./package.json": "./package.json"
  }
```

在万不得已时，可以通过 `"./*": "./*"` 来导出整个包的根目录，此时"封闭"功能也就不起作用了 。 这种将包内的所有文件暴露出去的方式，以禁用封闭为代价的同时，也提供了潜在的工具便利。由于**Node.js 中的 ES M 模块加载器强制使用完整路径**，因此导出根目录而不是显式的定义入口，前者的表现力比我们之前的例子要差很多。这不仅失去了封闭，而且调用模块的人也无法直接使用 `import feature from 'my-mod/feature'` ，因为他们需要写下完整路径 `import feature from 'my-mod/feature/index.js` 。

1. 主入口的导出（Main entry point export）

要设置包的主入口，最好在 `package.json` 中同时定义 `"exports"` 和 `"main"` 。

复制代码

```
{
  "main": "./main.js",
  "exports": "./main.js"
}
```

当定义了 `"exports"` 字段，所有子路径都会被封闭，不再对使用者开放。例如，`require('pkg/subpath.js')` 会抛出 ERR_PACKAGE_PATH_NOT_EXPORTED 错误。

这种对导出的封闭不仅为工具提供了更可靠的接口保证，也为处理包的升级提供了保证。这不是一个严格意义上的封闭，因为直接 require 任何绝对路径，如 `require('/path/to/node_modules/pkg/subpath.js')` 仍然会加载 `subpath.js` 。

2. 子路径的导出（Subpath exports）

当使用 `"exports"` 字段时，可以将主入口视为 `"."` 路径，然后构造自定义路径：

复制代码

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./submodule": "./src/submodule.js"
  }
}
```

目前只有在 `"exports"` 中定义的子路径才能被导入：

复制代码

```
import submodule from 'es-module-package/submodule';
//加载./node_modules/es-module-package/src/submodule.js。
```

导入其他子路径就会报错：

复制代码

```
import submodule from 'es-module-package/private-module.js';
// 抛错 ERR_PACKAGE_PATH_NOT_EXPORTED
```

3. 子路径的导入（Subpath imports）

除了 `"exports"` 字段之外，还可以定义内部包导入映射，这些映射只适用于在包内部导入指定内容。 `imports` 字段中定义的入口必须一直以 `#` 开头，以确保它们与包的对外指定内容相区别。例如， `imports` 字段也可以做内部模块的条件导出：

复制代码

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "./dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

上述例子展示了，当处于非 node 环境中时， `import '#dep'` 并不能获取到外部包 `dep-node-native` ，而会获取本地文件 `./dep-polyfill.js` 。与 `"exports"` 字段不同， `"imports"` 字段允许映射外部包。 `imports` 字段的解析规则与 `exports` 字段类似。

4. 子路径模式（Subpath patterns）

对于 `exports` 或 `imports` 数量较少的包，我们建议明确列出每个子路径入口。但对于有大量子路径的包来说，这可能会导致 `package.json` 出现臃肿和维护问题。对于这种有大量路径的情况，我们可以用子路径导出模式来代替：

复制代码

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./features/*": "./src/features/*.js"
  },
  "imports": {
    "#internal/*": "./src/internal/*.js"
  }
}
```

左侧匹配模式必须都以 * 结尾。右侧所有的 * 实例都将被替换成左侧的值，包括它是否包含/分隔符。

复制代码

```
import featureX from 'es-module-package/features/x';
//加载./node_modules/es-module-package/src/features/x.js。

import featureY from 'es-module-package/features/y/y';
//加载./node_modules/es-module-package/src/features/y/y.js。

import internalZ from '#internal/z';
// 加载 ./node_modules/es-module-package/src/internal/z.js。
```

这是一个直接的静态替换，没有对文件扩展名进行任何特殊处理。在上述例子中，`pkg/features/x.json` 将在映射中被解析为 `./src/features/x.json.js` 。 `exports` 的属性是否能维持为静态可枚举，取决于导出模式如何设置，比如导出模式的右侧匹配写为 `**` 还是列出一大串文件列表。由于 `exports` 的目标中禁止写 `node_moduels` 路径，所以模式匹配只能写包本身的文件。

5. 子路径文件夹映射（Subpath folder mappings）

> 稳定性：0 - 已废弃。使用子路径模式代替。
> 由于已废弃，此处就不放译文了。

6. Exports 语法糖（Exports sugar）

当 `"."` 是唯一的导出， `"exports"` 字段为这种情况提供了语法糖。当 `"."` 导出有降级回落的数组或字符串值，那么 `"exports"` 字段可以直接设置为这个值。

复制代码

```
{
  "exports": {
    ".": "./main.js"
  }
}
```

可改写为：

复制代码

```
{
  "exports": "./main.js"
}
```

7. 条件导出（Conditional exports）

条件导出提供了一种根据特定条件映射到不同路径的方法。CommonJS 和 ES模块导入都支持条件导出。例如，一个包如果想为 `require()` 和 `import` 提供不同的 ES模块导出，可以写为：

复制代码

```
// package.json
{
  "main": "./main-require.cjs",
  "exports": {
    "import": "./main-module.js",
    "require": "./main-require.cjs"
  },
  "type": "module"
}
```

Node.js开箱即支持以下情况：

- `"import"` - 当包通过 `import` 或 `import()` 加载，或通过 ECMAScript 模块加载器的任何顶层导入或解析操作加载时，该条件就会匹配。无论目标文件的模块格式如何，都适用。*`"import"`* 总是与 *`"require"`* 互斥。
- `"require"` - 当包通过 `require()` 加载时匹配。被引用的文件应该可以用 `require()` 加载，尽管该条件与目标文件的模块格式无关。预期的格式包括 CommonJS、JSON 和本地插件，但不包括 ES模块，因为 `require()` 并不支持它们。 *`"require"`* 总是与 *`"import"`* 互斥

- `"node"` - 匹配任何 Node.js 环境。可以是 CommonJS 或 ESM 模块文件。这个条件应该总是在 `"import"` 或 `"require"` 之后。

- `"default"` - 默认的降级条件。可以是一个 CommonJS 或 ESM 模块文件。这个条件应该总是排在最后。

在 `"exports"` 对象中，键序很重要。在条件匹配过程中，排序靠前的入口具有较高的优先级。通常规则是，这些条件应该从最特殊到最不特殊来排序。

其他条件如 `"browser"`、`"electron"`、`"deno"`、`"react-native"` 等，对 Node.js 来说是不可用的，因此会被忽略。Node.js 以外的运行时或工具可以酌情使用它们。未来可能会出现更多关于条件名称的限制、定义或引导。

使用 `"import"` 和 `"require"` 条件可能会导致一些危害，这将在“CommonJS/ES 双模块包部分”进一步解释。子路径也可以使用条件导出，如下：

复制代码

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./feature": {
      "node": "./feature-node.js",
      "default": "./feature.js"
    }
  }
}
```

上面这段代码就展示了：在 Node.js 和其他 JS环境 使用 `require('pkg/feature')` 和 `import('pkg/feature')` 还有不同的实现。

当使用环境分支时，需尽量包含 `"default"` 进行兜底。提供 `"default"` 可以确保任何未知的 JS 环境都能够使用这个通用的实现，这也避免未知的 JS 环境为了支持条件导出而不得不假装成现有环境。也因为如此，使用 `"node"` 和 `"default"` 条件分支通常比使用 `"node"` 和 `"browser"` 条件分支更可取。

8. 嵌套条件（Nested conditions）

除了直接映射，Node.js 还支持嵌套条件对象。例如：定义一个只供 Node.js 使用双模式入

复制代码

```
{
  "main": "./main.js",
  "exports": {
    "node": {
      "import": "./feature-node.mjs",
      "require": "./feature-node.cjs"
    },
    "default": "./feature.mjs",
  }
}
```

条件继续按次序匹配扁平化后的条件。如果一个嵌套条件没有任何映射，它将继续检查父条件的剩余条件。在这种方式下，嵌套条件的行为类似于嵌套的 JavaScript `if` 语句。

9. 处理用户条件（Resolving user conditions）

在运行 Node.js 时，使用 `--conditions` 标志可以添加自定义条件：

复制代码

```
node --conditions=development main.js
```

当恰当地解析现有的 `"node"`、 `"default"`、 `"import"` 和 `"require"` 条件时，上面这段代码会在包的导入和导出中解析 `"development"` 条件。可以添加任意多的自定义条件，只需要一直添加重复的 `--conditions` 即可。

10. 使用包名称进行自我引用（Self-referencing a package using its name）

在包内部， `package.json` 中 `"exports"` 定义的值可以被用作包内引用的名称。假设 `package.json` 如下：

复制代码

```
// package.json
{
  "name": "a-package",
  "exports": {
    ".": "./main.mjs",
    "./foo": "./foo.js"
  }
}
```

然后，_该包中_的任何模块都可以在包内被引用：

复制代码

```
// ./a-module.mjs
import { something } from 'a-package'; // Imports "something" from ./main.mjs.
```

自引用只在 `package.json` 有 `"exports"` 的情况下才能使用，并且只允许导入该 `"exports"` （在 `package.json` 中）允许的内容。所以按照上例中 `package.json` 的定义，下面的代码会产生一个运行时错误：

复制代码

```
// ./another-module.mjs。

// Imports "another" from ./m.mjs。失败的原因是
// "package.json" 中的 "exports" 字段
// 没有提供名为 "./m.mjs" 的导出。
import { another } from 'a-package/m.mjs';
```

当使用 `require` 时自引用也是可行的，无论是在 ES模块 还是在 CommonJS 模块中。例如：

复制代码

```
// ./a-module.js
const { something } = require('a-package/foo'); // Loads from ./foo.js.
```

四. 双 CommonJS/ESM 模块包（Dual CommonJS/ES Module packages）

在 Node.js 支持 ESM 模块之前，包作者在包内同时放入 CommonJS 和 ESM 模块源码是一种常见的模式，方式为：在 `package.json` 的 `"main"` 字段指定 CommonJS 入口，在 `"module"` 字段指定ES模块入口。这使得 Node.js 能够正常运行 CommonJS 入口，而其他打包构建工具等则使用 ES模块入口，因为 Node.js 会忽略（并且仍然忽略）顶层的 `"module"` 字段。

Node.js 现在可以运行 ESM 模块 入口，一个包可以同时包含 CommonJS 和 ESM 模块 入口（可以通过单独的指定内容如 `"pkg"` 和 `"pkg/es-module"` ，也可以通过"条件导出"将两者放在同一个指定内容上）。与 `"module"` 只被打包程序使用，或是在 Node.js 执行前 ESM 模块文件被转化为 CommonJS 的情况不同，ESM 模块入口 引用的文件是直接作为 ESM 模块 使用的。

双包危害（Dual package hazard）

当一个应用使用一个同时提供 CommonJS 和 ES模块源 的包时，如果这两个版本的包都被加载，那么就有可能出现某些bug。这种潜在的风险来自这样一个事实：由 `const pkgInstance = require('pkg')` 创建的 `pkgInstance` 与由 `import pkgInstance from 'pkg'` （或像 `'pkg/module'` 这样的替代主路径）创建的 `pkgInstance` 并不相同。同一个包的两个版本在同一运行时环境中被加载产生的影响，被称为 "双包危险"。虽然应用程序或包不太可能故意直接加载两个版本，但常见的情况是：应用程序加载一个版本，而该应用程序的依赖加载了另一个版本。因为 Node.js 支持 CommonJS 和 ES模块 的相互混合，所以这种危害可能会发生，并导致意外行为。

如果包主要导出的是一个构造函数，那么两个版本创建的实例的 `instanceof` 的比较会返回 `false` 。另外，如果导出的是一个对象，为一个版本的对象添加的属性（比如 `pkgInstance.foo = 3` ）在另一个版本上是不存在的。这与 `import` 和 `require` 语句分别在全 CommonJS 或全 ESM 模块环境中的作用方式不同，因此会让使用者感到惊讶。这也与大家通过 Babel 或 esm 等工具编译时熟悉的行为不同。

当编写双包时避免或减少危害（Writing dual packages while avoiding or minimizing hazards)

首先，当一个包同时包含 CommonJS 和 ESM 模块源，并且这两个源都给 Node.js 使用时就会发生上一节中描述的危险，不论是通过单独的主入口或导出的路径。相反，如果编写一个包，任何版本的 Node.js 都只接收 CommonJS 源，而包中任何单独的 ESM 模块源只用于其他环境，比如只作用于浏览器。这样的包就可以被任何版本的 Node.js 使用，因为 `import` 可以引用 CommonJS 文件，只是它不会提供使用 ES 模块语法的任何优势。

一个包也可能会在一次大版本更新中从 CommonJS 切换到 ESM 模块语法。这会有一个缺点，即包的最新版本只能在支持 ESM 模块的 Node.js 版本中使用。

每种模式都需要权衡，但有两种方法可以满足以下情况：

1. 包可以通过 `require` 和 `import` 两种方式使用
2. 包既可以在当前的 Node.js中 使用，也可以在缺乏 ESM 模块 支持的旧版本 Node.js 中使用
3. 包的主入口，例如 `'pkg'` 既可以通过 `require` 解析到 CommonJS 文件，也可以通过 `import` 解析到 ESM 模块文件。(同样，导出的路径也是如此，例如 `'pkg/feature'`)
4. 包提供了命名导出，例如 `import { name } from 'pkg'` ，而不是 `import pkg from 'pkg'; pkg.name`

6. 上一节中描述的双包危害应被避免或最小化了

方法#1：使用 ESM 模块封装器（Approach #1: Use an ES module wrapper）

用 CommonJS 编写包或将 ESM 模块源码 转译为 CommonJS 的方式编写包，并创建一个带有命名导出的 ESM 模块封装文件。使用“条件导出”，将 ESM 模块封装器用于 `import`，CommonJS 入口用于 `require`。

复制代码

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    "import": "./wrapper.mjs",
    "require": "./index.cjs"
  }
}
```

上面的例子使用了显式扩展名 `.mjs` 和 `.cjs`。`"type": "module"` 将把 `.js` 文件视为 ES模块，就像 `"type": "commonjs"` 会把 `.js` 文件视为 CommonJS 一样。

复制代码

```
// ./node_modules/pkg/index.cjs
exports.name = 'value';


// ./node_modules/pkg/wrapper.mjs
import cjsModule from './index.cjs';
export const name = cjsModule.name;
```

在上面的例子中，`import { name } from 'pkg'` 中的 `name` 和 `const { name } = require('pkg')` 中的 `name` 是同一个东西。因此 `===` 在比较两个 `name` 时返回 `true`，避免了不同指定符造成的危害。

如果模块不是简单的命名导出列表，而是包含了独特的函数或对象导出，比如 `module.exports = function () { … }`，或如果希望在封装器中支持 `import pkg from 'pkg'`，那么封装器将被写成导出默认的同时也导出其他命名值：

```
export default cjsModule;
```

这种方法适合于以下任何一种用例：

- 包目前是用 CommonJS 编写的，作者不希望将其重构为 ESM 模块语法，但希望为 ESM 模块使用者提供命名导出
- 这个包还有其他依赖它的包，终端用户可能会同时安装这个包和一些依赖它的包。例如 `utilities` 包直接被使用在应用程序中，而 `utilities-plus` 包则为 `utilities` 增加了一些功能。因为封装器导出了底层的 CommonJS 文件，所以不管 `utilities-plus` 是用 CommonJS 还是 ES模块语法写的，这都可以正常使用。
- 包中存储了内部状态，包作者不打算重构包来隔离其状态管理。参见下一节"方法#2"。

这种不要求使用者进行条件导出的另一个方式是添加一个导出，例如：`"./module"`，指向一个全 ES模块语法版本的包。如果使用者确信 CommonJS 版本不会在应用程序中的任何地方被加载，比如通过依赖关系；或者 CommonJS 版本可以被加载但不影响 ES模块版本（举个例子，因为包是无状态的），那么就可以通过 `import 'pkg/module'` 来使用。

复制代码

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./wrapper.mjs"
  }
}
```

方法#2：隔离状态（Approach #2: Isolate state）

`package.json` 文件可以直接定义 CommonJS 和 ESM 模块的独立入口：

复制代码

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
```

```
  }
}
```

如果包的 CommonJS 和 ESM 模块版本是等价的，例如因为一个是另一个的转译过来的；并且包的状态管理被仔细隔离（或者包是无状态的），就可以做到这一点。

之所以状态是一个问题，是因为包的 CommonJS 和 ESM 模块版本都可能在应用程序内被使用；例如，用户的应用程序代码可能 `import` ESM 模块版本，而应用中的一个依赖 `require` 了 CommonJS 版本。如果发生这种情况，内存中会加载这两个版本，因此会出现两种不同的状态。这很可能会产生难以解决的 bug。

除了编写一个无状态的包（假如 JavaScript 的 `Math` 是一个包，由于它所有方法都是静态的，那么它就是无状态的），还有一些方法可以隔离状态，使得状态在可能加载的 CommonJS 和 ESM 模块实例之间被共享：

1. 如果可能的话，将所有的状态都包在一个实例化的对象中。例如，JavaScript 的 `Date` 需要被实例化来包含状态；如果它是一个包，就会被这样使用：

复制代码

```
import Date from 'date';
const someDate = new Date();
// someDate 包含状态; Date 无状态
```

`new` 关键字并不是必须的，包的函数也可以返回一个新的对象，或者修改一个传入的对象，以保持包的外部状态。

2. 将在 CommonJS 和 ESM 模块版本的包中共享的一个或多个 CommonJS 文件，做状态隔离。例如，如果 CommonJS 和 ESM 模块 的入口分别是 `index.cjs` 和 `index.mjs`：

复制代码

```
// ./node_modules/pkg/index.cjs
const state = require('./state.cjs');
module.exports.state = state;


// ./node_modules/pkg/index.mjs
import state from './state.cjs';
export {
  state
};
```

即使 `pkg` 在应用程序中分别通过 `require` 和 `import` 两种方式使用 (例如，在应用程序代码中使用 `import` 和在其他依赖模块中使用 `require`)， `pkg` 的每个引用都将包含相同的状态；并且从任一模块系统中修改该状态都将应用于这两个模块。

任何与包的单例相关的插件都需要分别应用于 CommonJS 和 ESM 模块的单例上。这种方法适用于以下任意一种用例：

- 包目前是用 ESM 模块语法编写的，而且包作者希望在支持该语法的地方使用该版本
- 包是无状态的，或它的状态可以比较容易得被隔离
- 包不太可能被其他公共包所依赖；或者如果有被依赖的话，这个包是无状态的，或它的状态不需要在依赖者之间或在整个应用程序中被共享。

即使是隔离状态，在包的 CommonJS 和 ESM 版本之间可能仍需要执行额外代码。与前一种方法相同，这种不要求使用者做条件导出的另一个方法是：添加一个导出。例如 `"./module"` ，以指向包的全 ESM 版本：

复制代码

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./index.mjs"
  }
}
```

五. Node.js `package.json` 字段定义

本节描述 Node.js 运行时使用的字段。其他工具（如 npm）使用了额外的字段，这些字段被 Node.js 忽略了，在此不做说明。 `package.json` 中的以下字段是被 Node.js 使用的：

- "name" - 当在包内使用命名导入时相关。也被包管理器用作包的名称。
- "main" - 如果没有指定 `exports` ，或在无法使用 `exports` 的老 Node.js 版本中， `"main"` 就是加载包时的默认模块。
- "type" - 这个字段决定了将 `.js` 文件作为 CommonJS 还是 ES模块 进行加载

- "exports" - 包导出和条件导出。当 `"exports"` 有值时，会限制可以被加载的子模块为哪些。
- "imports" - 包的导入，供包内模块本身使用。

1. "name"

历史变更：

版本	变更
v13.1.0、v12.16.0	在 v13.1.0、v12.16.0 版本中新增该属性
v13.6.0、v12.16.0	移除 `--experimental-resolve-self` 选项

- Type: `<string>`

复制代码

```
{ "name": "package-name" }
```

`"name"` 字段定义了包的名称。发布到 *npm* 上需要一个满足特定要求的名字。

`"name"` 可以搭配 `"exports"` 字段，来实现自引用。

2. "main"

在 v0.4.0 新增该属性

- Type: `<string>`

复制代码

```
{ "main": "./main.js" }
```

当通过 `require()` 加载包目录时, 就是使用 `"main"` 字段定义的内容。 `"main"` 的值是一个路径。

复制代码

```
require('./path/to/directory'); // 这将被解析为 ./path/to/directory/main.js.
```

3. "type"

历史变更：

版本	变更
v13.2.0、v12.17.0	不再使用 `--experimental-modules` 标志
v12.0.0	在 v12.0.0 版本中新增该属性

- Type: `<string>`

`"type"` 字段定义了 Node.js 对所有以该 `package.json` 为最近的父文件的 `.js` 文件所使用的模块格式。

当最近的父 `package.json` 的顶层字段 `"type"` 为 `"module"` 时，以 `.js` 结尾的文件将作为ES模块加载。

最近的父 `package.json` ：被定义为在当前文件夹中搜索时找到的第一个 `package.json` ，或在该文件夹的父包找到的第一个 `package.json` 以此类推，直到到达 `node_modules` 文件夹或根目录。

复制代码

```
// package.json
{
  "type": "module"
}
```

复制代码

```
# 与上面的 package.json 在同一文件夹中
node my-app.js # 以 ESM 模块的形式运行，因为 package.json里定义了 "module" type
```

如果最近的父 `package.json` 缺少 `"type"` 字段，或包含 `"type": "commonjs"` ，则 `.js` 文件将被视为 CommonJS。如果直到根目录也没找到 `package.json` ， `.js` 文件将被视为CommonJS。

如果最近的父 `package.json` 中包含 `"type": "module"` ，那么在 `.js` 文件被 `import` 时会被作为ES模块处理：

复制代码

```
// my-app.js，这是上例的一部分内容
import './startup.js'; // 由于package.json，这会作为ES模块被加载。
```

无论 `"type"` 字段的值是什么， `.mjs` 文件总是被视为 ES模块，而 `.cjs` 文件总是被视为 CommonJS。

4. "exports"

历史变更：

版本	变更
v14.13.0	新增支持 `exports` 模式
v13.7.0、v12.16.0	实现条件导出的逻辑顺序
v13.7.0、v12.16.0	移除 `--experimental-conditional-exports` 选项
v13.2.0、v12.16.0	实现条件导出
v12.7.0	在 v12.7.0 版本中新增该属性

- Type: `<Object>` | `<string>` | `<string[]>`

复制代码

```
{ "exports": "./index.js" }
```

`"exports"` 字段可以定义包的入口，而包则可以通过 `node_modules` 查找或自引用导入。Node.js 12+ 开始支持 `"exports"` ，作为 `"main"` 的替代，它既支持定义子路径导出和条件导出，又封闭了内部未导出的模块。

条件导出也可以在 `"exports"` 中使用，以定义每个环境中不同的包入口，包括这个包是通过 `require` 还是通过 `import` 引入。

所有在 `"exports"` 中定义的路径必须是以 `./` 开头的相对路径URL。

5. "imports"

在 v14.6.0 版本中新增该属性

- Type: `<Object>`

复制代码

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "./dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

`imports` 字段中的入口必须是以 `#` 开头的字符串。

导入映射允许映射外部包。这个字段定义了当前包的子路径导入。

六. 附录：webpack 功能校验

以上 Node.js 支持的功能，用 webpack 也检测了一下：

下图为使用 webpack 在浏览器端的验证：

上述**测试用例的 repo** 请访问：github.com/zEmily/webp...

七. 总结

查看了几个常用的库，发现它们并没有在使用 `exports` 新特性。或许包作者目前不想做破坏性的更新迭代，但未来预计新特性都会慢慢更新上去。

虽然外部的包没有在使用 `exports`，但是我们可以在内部先使用起来。比如书写通用 lib 包时，使用 `exports` 仅暴露预期内的接口，而不是将所有包内容都让外部随意访问；如果这个包

仅在 node 环境下使用的话，可以使用上面提到的 **“条件导出”** 来做限制，避免出现不必要的 bug。

分类: 前端 标签: Node.js 前端

评论

输入评论（Enter换行，Ctrl + Enter发送）

全部评论 2

最新 最热

lianjianwei 后端菜鸟 @ 无 1月前

这个 ESM 在 node 14.8.x 版本可以正式使用了吗？以前都是用 babel 去转的。

点赞 1

字节大力智能 Lv3 （作者） 1月前

是可以正式使用的

点赞 回复

相关推荐

iwhao 3月前 前端 Node.js

给女友写的，每日自动推送暖心消息

1.6w 357 181

2.3w 324 44

Alone381 3年前 Node.js 前端

前端的焦虑，你想过30岁以后的前端路怎么走吗?

4.0w 623 332

卡不通 1年前 Node.js

node导出pdf

668 4 2

Plum23535 4年前 ECMAScript 6

ES6是如何解决js中功能模块导入导出问题的

1599 33 评论

KDDA_ 9月前 React.js

如何做到修改了node_module中的包，却不受重新安装的影响

2590 93 21

muwoo 3年前 Node.js Vue.js 前端

厌倦了写活动页？快来撸一个页面生成器吧！

2.0w 570 41

小炼_ 9月前 Node.js

看了就会的 Node.js 三个模块常用 API

568 8 评论

gavin103 3年前 Node.js JavaScript 前端

用JS搞了一个自动翻译，从此不再头疼看英文书了

1.2w 258 36

FESKY 1年前 Node.js JavaScript

CommonJS 和 ES6 Module 究竟有什么区别?

1.1w 149 9

只非鱼 4年前 Node.js JavaScript 服务器

小张不慌张 4年前 Node.js JavaScript 面试

如何通过饿了么 Node.js 面试

2.6w 894 19

kledwu 3年前 Node.js 前端 koa

接口咋整？前端数据药神来也

1.5w 531 59

ahabhgk 2年前 Node.js

Node.js HTTP服务器中不依赖第三方模块的文件、图片上传

121 1 评论

ClausClaus 1年前 Node.js

NodeJs获取文件流并导出到本地的三种方案

1815 4 1

王、先生 4月前 前端 Node.js

Node 中Module的require

139 2 评论

胡子大哈 4年前 JavaScript Node.js

在 Node.js 中引入模块：你所需要知道的一切都在这里

4301 88 2

zxg_神说要有光 1月前 前端 JavaScript Node.js

JS 的 6 种打断点的方式，你用过几种?

2.2w 325 31

众成翻译 11月前 Node.js

更新 Node 模块的正确姿势 - Jama Software

97 1 评论

小彩笔 2年前 Node.js

node相关草稿