

前言

工欲善其事，必先利其器。做好一个项目前期准备工作很重要，前期准备无非就是合理规划项目结构、按需编写构建代码、批量创建入口文件、按需封装工具函数等，而按需封装工具函数是作为一个项目基础的重中之重，是很有必要提前规划的。

这些经常重复使用的工具函数，包括但不限于浏览器类型、格式时间差、URL参数反序列化、过滤XSS等。为了避免应用开发时重复的复制粘贴操作带来不必要的麻烦，有些开发者都会将这些工具函数根据功能划分并统一封装，再发布到Npm公有仓库。每次使用时直接安装，提升开发效率，将时间用在正确的事情中。

webpack作为一个长期霸占打包领域的神器，得以其强大的生态系统，很多前端应用都基于webpack打包，因此很多类库在早期也选择webpack作为打包工具。随着前端技术的大力发展，近几年也涌现出像rollup、esbuild和parcel这样的优秀打包工具，每个工具都具备自己的特色且在某些应用场景占据一定的优势，因此webpack的地位也受到挑战，形成一超多强的局面。

虽然webpack蚕食大量打包市场份额，但也不是所有情况都适用webpack。在某些应用场景使用webpack打包代码也可能引发很多问题，例如引用大量无用代码，随着项目体积增大导致打包速度过慢、打包时间过长和打包体积过大。

基于上述原因，现在很多类库都会选择rollup打包代码，这些类库包括但不限于组件库与工具库，其共性都是包括以JS为主的代码，因此资源类型不会像应用那样复杂，所以对于单一类型单一功能的项目，使用rollup打包代码会更好，而rollup也是为JS类库打包而生。本章将带领你基于Rollup搭建类库基建模板，通过了解rollup打包思想，一步一步上手rollup，完成一个通用工具库的前端工程化模板。

在此提个醒，本课程的主题是前端工程化，因此不会展开编写业务代码，只会集中篇幅讲述前端工程化相关代

rollup 是一个 JS 模块打包器，可将小块代码编译为大块复杂代码。与 webpack 偏向于应用打包的定位不同，rollup 更专注于类库打包。像常见前端框架 react 与 vue，其源码都是基于 rollup 打包的。

为何都是打包，而像 react 与 vue 都优先选择 rollup？总体来说 webpack 与 rollup 都可基于自己特性在不同应用场景发布自身的优势作用。

webpack 内置的 分割代码、按需加载、HRM 等特性在打包应用时有着先天的优势，而 rollup 并不支持这些功能，因此在打包应用时需配合很多第三方插件才能完成，而本身 rollup 的生态系统并不强大，第三方插件不齐全，部分插件也得不到很好的维护，因此 rollup 无法支撑一个应用的打包功能。

但 rollup 也并不是一无是处，其内置的 ES6 解析、摇树优化、作用提升等特性在打包类库时有些先天的优势，若你的项目是一个组件库或工具库，而这些单一类型单一功能的项目只需输出一个或多个不同规范的 bundle 文件供其他应用引用，那使用 rollup 再也适合不过了。

特性

因为 rollup 配置简洁、性能到位和功能专一的特性，webpack 在 v2 也开始借鉴 摇树优化、作用提升等先进特性。

摇树优化

rollup 在普通情况下，必须使用 ESM 编码，而不是以前的模块规范 CJS 或 AMD。在使用 ESM 编码的基础上，rollup 会静态分析代码中的 import 并将删除所有未实际使用的代码。这允许架构在现有模块中，而不会增加额外依赖或项目体积。

因为 rollup 只引用最基本最精简的代码，所以可生成轻量快速低复杂度的类库。这种基于显式 import/export 的声明方式，远比在编译后 bundle 文件中简单地运行压缩程序自动检测未使用的变量更有效。

摇树优化就是 TreeShaking，可查看 [《TreeShaking实现原理》](#) 好好了解其原理，在此不深入讲述了。

模块兼容

虽然 `rollup` 在普通情况下必须使用 `ESM` 编码，但很多古老的 `Npm` 模块都是基于 `CJS` 编码，贸然引用这些 `Npm` 模块肯定会在打包流程中报错，因此官方推荐使用 [@rollup/plugin-commonjs](https://github.com/rollup/plugins/tree/master/packages/commonjs) 解决这些问题。

使用

`rollup` 可全局安装使用或局部安装使用，在项目中实现 前端工程化 会结合各种工具链做实践，所以使用局部安装会更适合。在根目录中创建 `package.json` 文件，执行 `npm i` 安装依赖。

```
{
  "name": "rollup",
  "version": "1.0.0",
  "main": "dist/index.js",
  "scripts": {
    "build": "rollup -c"
  },
  "engines": {
    "node": ">=13.2.0",
    "npm": ">=6.13.1"
  },
  "dependencies": {
    "rollup": "2.75.5"
  }
}
```

在项目中创建以下业务文件与配置文件。

- `src/a.js`
- `src/b.js`
- `src/index.js`
- `rollup.config.js`

```
// src/a.js
export function helloA() {
  const msg = "a";
  console.log(msg);
}
```

```
// src/b.js
export function helloB() {
  const msg = "a";
}
```

js

```

        console.log(msg);
    }

    // src/index.js
    import { helloA } from "./a.js";
    import { helloB } from "./b.js";

    helloA();

    // rollup.config.js
    export default {
        input: "src/index.js",
        output: {
            file: "dist/index.bundle.js",
            format: "esm"
        }
    };

```

js

执行 `npm run build` 生成 `dist/index.bundle.js` 文件。仔细观察上述代码与以下代码的区别，大致的编译流程是从 `src/index.js` 出发，分析 `import` 的两个函数 `helloA()` 与 `helloB()`，再分析 `import` 后方的代码，发现只使用过 `helloA()`，因此从 `helloA()` 引用的路径 `src/a.js` 出发，继续上述流程，直至分析完毕，最后将使用过的代码块提取出来，合并到 `index.bundle.js` 中再输出到 `dist` 文件夹。

```

function helloA() {
    const msg = "a";
    console.log(msg);
}

helloA();

```

js

还有一个细节的地方，`const` 并未转换为 `var`。若使用更多 `ESM` 的语法与 `API`，也会出现一样的情况，所以通过观察可发现 `rollup` 的打包流程是将那些使用过的代码块合并为一个或多个 `bundle` 文件，其中起到一个搬运工的作用。

若不考虑 `Web` 兼容性，`rollup` 通过入口文件收集所有使用过的代码块，最终打包为一个或多个 `bundle` 文件，浏览器可通过 `<script src="path/index.bundle.js" type="module"></script>` 引用文件。若考虑 `Web` 兼容性，在打包流程中还需加入 `babel/typescript` 这样的转译工具，将 `ESM` 转换为目标版本的 `JS` 代码。

初步看来，rollup 真的很适合打包类库，相信时间越往后，将会越多类库优先使用 rollup 打包。这样打包出来的代码在生产阶段会更好地利用 摇树优化 删除所有未实际使用的代码，最大限度减少项目体积。

原理

rollup 的初衷在于让 ESM 无处不在。如今 Web 与 Node 都在大力推行 ESM，近两年双环境版本都兼容 95% 以上的语法与 API，因此未来的 ESM 将完全取代 CJS，成为 JS 完全标准的 模块规范，而 rollup 现在就开始做这些工作。

为何 rollup 建议开发者在普通情况下必须使用 ESM 编码？高版本的 Web 与 Node 无需在其他工具的加持下就能原生支持 ESM，能在代码层面基于静态分析使用 摇树优化 删除所有未实际使用的代码，也能解决代码块循环依赖的问题。

将上述代码使用 webpack 打包一次得到以下代码。仔细观察，发现外层使用一大段 webpack 的注入代码，这些代码是 webpack 自行实现的代码垫片，目的是编译 require、modules.exports 和 export，让 Web 与 Node 可兼容 CJS 与 ESM。

```
js
"use strict";
(self.webpackChunktest_webpack = self.webpackChunktest_webpack || []).push([
  [826],
  { 973: function() { console.log("a") } },
  function(e) { var t; t = 973, e(e.s = t) }
]);
```

反观 rollup 打包的 bundle文件，除了组装形式，其余代码片段很多是原装搬运过来，因此相比 webpack 打包的 bundle文件 简洁很多，无注入代码也无太大结构变动。rollup 这样做是为了利用高版本 Web 与 Node 原生支持 ESM，而 webpack 更像一个考虑太多的孩子，自行实现代码垫片并注入，目的是同时兼容 CJS 与 ESM。

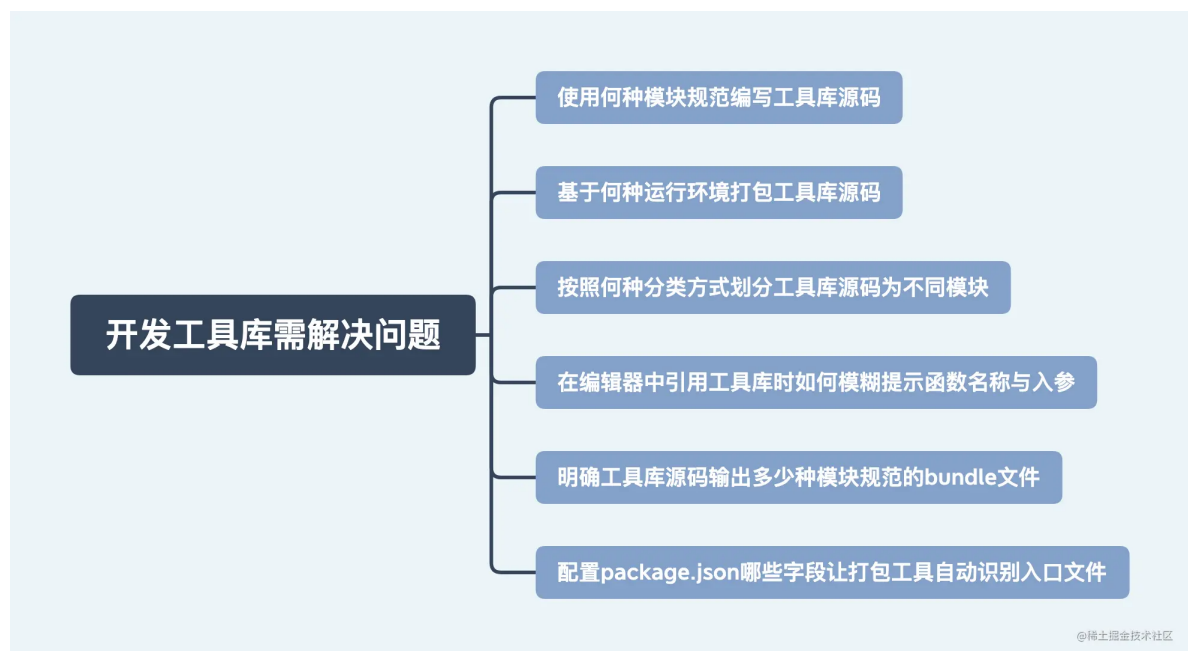
以下总结了两者对 CJS 与 ESM 的支持情况，虽然 rollup 原生不支持 CJS，但可通过[@rollup/plugin-commonjs](#)将 CJS 转换为 ESM 再让其参与到后续编译中。

打包工具	纯ESM	纯CJS	两者混用
webpack	✓(有注入)	✓(有注入)	✓(有注入)
rollup	✓(无注入)	✗	✗

通过上述分析不难得知 **rollup** 的出发点是希望开发者更多地使用 **ESM** 编码，这样适合在代码层面基于静态分析使用 **摇树优化** 删除所有未实际使用的代码，也能解决代码块循环依赖的问题，同时在 **Web** 中除了 `<script>` 外还能让 **JS** 真正拥有 **模块化** 的能力。

方案：基于Rollup搭建类库基建模板

在确定使用 **rollup** 打包类库后，接着从零到一构建一个通用工具库的 **前端工程化** 模板。该模板可用于日常搭建类库的骨架，通过命令打包输出 **bundle文件**，该模板也可作为一个 **Npm模块**，也需具备一个完整的 **Npm模块** 的工程特征。在进入开发工作前，需明确以下问题，这些问题将影响着整个开发流程与扩展思路。



上述问题将作为重点会在整个开发流程中重点讲述。那跟随我的步伐一起开撸吧。

初始项目骨架

首先为工具库起一个独一无二的包名，例如工具库的英文名字是 **Util Set**，那结合我的英文名字 **Bruce**，就成了 **bruce-us**。

包名不是起好就能用，因为最终还会发布到 **Npm公有仓库** 供其他开发者使用，那肯定会遇到包名重复、包名类似等问题导致发布不成功。执行 `npm info <pkg>`，输出以下信息则表示包名可用。可用的话就赶紧随便发一点代码占坑吧，不然被别人抢先注册就不好了。

```
npm ERR! code E404
npm ERR! 404 [NOT_FOUND] bruce-us not found : bruce-us
npm ERR! 404
npm ERR! 404 "bruce-us" is not in the npm registry.
npm ERR! 404 You should bug the author to publish it (or use the name yourself!)
npm ERR! 404
npm ERR! 404 Note that you can also install from a
npm ERR! 404 tarball, folder, http url, or git url.
```

然后初始代码仓库，代码仓库必须包括以下文件。配置 `.gitignore` 与 `.npmignore` 为了防止 `node_modules` 文件夹等自动生成的文件夹或文件提交到 Github 或 Npm，提交忽略是首要考虑的事情。接着就使用 VSCode 编码了。

```
bruce-us
├─ .gitignore
├─ .npmignore
└─ package.json
```

txt

在根目录中创建 `.gitignore` 与 `.npmignore` 文件，加入以下内容。可能有些同学不明白，`package-lock.json` 与 `yarn.lock` 用于锁版本，为何将其列入，因为涉及 `version` 相关知识，将在第20章详细讲述。

```
.DS_Store
node_modules
package-lock.json
yarn.lock
```

txt

在根目录中创建 `package.json` 文件，加入以下内容。`name` 自行定义，记得执行 `npm config <pkg>` 查重就好。`version` 与 `license` 作为重点将在第20章详细讲述。`description` 用于描述模块的功能或作用，可用一段简洁的话语表述清楚。`keywords` 用于简括模块，关键词越多越好，Npm搜索功能就是基于该字段查找相同或相似 Npm模块，因此完善更多的关键词将会提升搜索优先级别。

```
{
  "name": "bruce-us",
  "version": "1.0.0",
  "description": "A Web/Node General Util Set",
  "keywords": [],
  "license": "MIT",
  "dependencies": {},
```

json

```
"devDependencies": {}  
}
```

`dependencies` 与 `devDependencies` 是 `package.json` 中的明星字段，它俩经常被混淆。`dependencies` 又称 生产依赖，用于项目的开发环境与生产环境，为项目的运行提供稳定保障，而 `devDependencies` 又称 开发依赖，用于项目的开发环境，为项目的开发、测试、部署等流程提供稳定保障。

像 `angular/react/vue` 等框架，`day/jquery/lodash` 等类库都属于 `dependencies`，而 `webpack/rollup/esbuild/parcel` 等打包工具，`postcss/babel/typescript` 等转译工具都属于 `devDependencies`。

将 `Npm` 模块 安装到 `dependencies` 通常执行以下命令。

```
npm i <pkg>  
# 或  
npm i --save <pkg>
```

将 `Npm` 模块 安装到 `devDependencies` 通常执行以下命令。

```
npm i --save-dev <pkg>  
# 或  
npm i -D <pkg>
```

相信后续安装依赖就能一目了然了。

规划源码结构

所有工具函数源码统一存放到 `src` 文件夹中，因为使用 `rollup` 打包代码，所以这些必须使用 `ESM` 编写。

一个工具库得明确 `bundle` 文件的运行环境是 `Web`，还是 `Node`，还是双环境。`Web` 与 `Node` 的部分代码无法打包在一起，具体原因如下。

- `Web` 独有 `window` 全局变量，`Node` 独有 `global` 全局变量
- `Web` 独有 `BOM/DOM` 等对象，`Node` 独有 `Fs/Path` 等模块
- `Web` 可兼容 `IIFE` 与 `AMD`，`Node` 可兼容 `CJS`，两者都兼容 `CMD`、`UMD` 和 `ESM`

基于上述原因，在打包 `bundle` 文件时应区分开来。`Web` 代码输出一个文件，`Node` 代码输出一个文件，对于一些既能在 `Web` 中运行也能在 `Node` 中运行的代码需单独抽离出来输出一个文件。不同运行环境兼容不同模块规范，因此 `bundle` 文件要保障其在当前环境具备通用性与兼容性。结合第2章 模块规范 相关知识点，`bundle` 文件最终的 模块规范 确定为 `CJS`、`ESM` 和 `UMD`。

打包的最终目的是生成一个 `Npm` 模块 并发布到 `Npm` 公有仓库 供其他开发者使用。当一个项目安装一个 `Npm` 模块 后，引用该模块并编码可能会发现该模块的函数名称与入参都有对应模糊提示，这样更友好地提升编码体验。实现上述效果都是 `typescript` 的功劳，`VSCode` 会扫描该模块 `package.json` 的 `types` 字段，该字段引用一个 `d.ts` 声明文件，该文件描述类库代码块的相关声明信息，能为 `VSCode` 提供类型检查等特性功能。为了更好地拥有该模糊提示功能，推荐使用 `typescript` 编写工具库，在打包代码时顺便生成 `d.ts` 声明文件。

综上所述，对于一个 `Web` 工具库 或 `Node` 工具库，可将目录规划为以下结构。

```
bruce-us
├─ src
│  ├─ xxx
│  ├─ yyy
│  ├─ zzz
│  ├─ index.ts
│  └─ index.umd.ts
├─ .gitignore
├─ .npmignore
└─ package.json
```

txt

综上所述，对于一个双环境工具库，可将目录规划为以下结构。针对这种工具库，使用 `common` 文件夹封装一些在 `Web` 与 `Node` 中都能运行的工具函数，为了使其 `bundle` 文件能同时兼容双环境，这些函数所在的文件不能引用 `BOM/DOM` 等 `Web` 独有对象，也不能引用 `Fs/Path` 等 `Node` 独有模块。`node` 文件夹与 `web` 文件夹封装一些各自运行环境的工具函数。

```
bruce-us
├─ src
│  ├─ common
│  ├─ node
│  ├─ web
│  ├─ node.ts
│  ├─ web.ts
│  └─ web.umd.ts
```

txt

```
├─ .gitignore
├─ .npmignore
└─ package.json
```

本章不会讲述如何编写工具函数的业务代码，这些函数就靠你自行整理了，毕竟主题是 **前端工程化**，一切业务代码都不在考虑范围内。为了能让项目顺利进行，我准备了一个包括 **88** 个工具函数的双环境**模板工具库**，该目录结构与上述对应，请将项目的 **src** 文件夹拷贝过来跟着我继续开撸。

安装一些生产依赖与开发依赖。

```
npm i dayjs fs-extra
```

```
npm i -D rollup
```

完善打包配置

在根目录中创建 **rollup.config.js** 文件，用于配置 **rollup** 打包配置。**rollup** 整体配置相比 **webpack** 简洁明了，可查看[Rollup官网](#)，主要用到的配置选项包括 **entry**、**output** 和 **plugins**。

output 有两个重要字段，分别是 **file** 与 **format**。**file** 表示 **bundle文件** 的路径；**format** 表示 **bundle文件** 的 **模块规范**，可选 **iife/cjs/amd/umd/esm/system**。

若工具库只输出一种 **模块规范** 的 **bundle文件**，可直接 **export default** 一个对象。

```
export default {                                     js
  input: "src/index.ts",
  output: {
    file: "dist/index.js",
    format: "cjs"
  }
};
```

若工具库输出多种 **模块规范** 的 **bundle文件**，可直接 **export default** 一个数组。上述确定输出 **CJS**、**ESM** 和 **UMD**，那 **rollup.config.js** 的配置如下。输出 **UMD** 还需

配置一个 `name` , 配置后可通过 `window.BruceUs` 调用全部工具函数。

```
export default [{  
  input: "src/web.ts",  
  output: { file: "dist/web.js", format: "cjs" }  
}, {  
  input: "src/web.ts",  
  output: { file: "dist/web.esm.js", format: "esm" }  
}, {  
  input: "src/web.umd.ts",  
  output: { file: "dist/web.umd.js", format: "umd", name: "BruceUs" }  
}, {  
  input: "src/node.ts",  
  output: { file: "dist/node.js", format: "cjs" }  
}, {  
  input: "src/node.ts",  
  output: { file: "dist/node.esm.js", format: "esm" }  
}];
```

在 `package.json` 中指定 `scripts` , 加入打包命令。

```
{  
  "scripts": {  
    "build": "rollup -c"  
  }  
}
```

执行 `npm run build` 打包工具库, 输出以下信息。这是因为无法识别 `ts`文件 导致编译不成功。

```
src/web.ts → dist/web.js...  
[!] Error: Could not resolve "../common/index" from src/web.ts  
Error: Could not resolve "../common/index" from src/web.ts
```

安装[@rollup/plugin-typescript](#), 使用该插件编译 `typescript` 。使用该插件编译 `typescript` 就无需配置 `babel` , 配置相关编译配置就能将代码输出为 `ES5` 。

```
npm i -D @rollup/plugin-typescript tslib typescript  
npm i -D @types/fs-extra @types/node
```

在根目录中创建 `tsconfig.json` 文件，加入以下内容。 `typescript` 的编译配置很多，可查看 [《掌握tsconfig.json》](#) 与 [《tsconfig.json配置解析》](#)，在此不深入讲述了。

json

```
{
  "compilerOptions": {
    "allowJs": true,
    "allowSyntheticDefaultImports": true,
    "baseUrl": ".",
    "declaration": true,
    "declarationDir": "dist",
    "downlevelIteration": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "forceConsistentCasingInFileNames": true,
    "jsx": "react",
    "lib": [
      "DOM",
      "DOM.Iterable",
      "ES2015",
      "ES2016",
      "ES2017",
      "ES2018",
      "ES2019",
      "ES2020",
      "ES2021",
      "ESNext"
    ],
    "module": "ES6",
    "moduleResolution": "node",
    "removeComments": true,
    "paths": {
      "#/*": ["*"],
      "@/*": ["src/*"]
    },
    "sourceMap": true,
    "strict": true,
    "target": "ES5",
    "typeRoots": [
      "node_modules/@types",
      "typings"
    ]
  },
  "exclude": [
    "dist",
    "node_modules",
    "rollup.config.js"
  ]
}
```

```
    ]  
  }  
}
```

改造 `rollup.config.js` 代码，为每个打包配置加入 `plugins`，`plugins` 使用 `@rollup/plugin-typescript` 编译 `typescript`。因为每个打包配置都包括统一的 `plugins`，所以使用一个变量 `PLUGINS` 包括这些插件实例。

```
import TypescriptPlugin from "@rollup/plugin-typescript";  
  
const PLUGINS = [  
  TypescriptPlugin()  
];  
  
export default [{  
  input: "src/web.ts",  
  output: { file: "dist/web.js", format: "cjs" },  
  plugins: PLUGINS  
}, {  
  input: "src/web.ts",  
  output: { file: "dist/web.esm.js", format: "esm" },  
  plugins: PLUGINS  
}, {  
  input: "src/web.umd.ts",  
  output: { file: "dist/web.umd.js", format: "umd", name: "BruceUs" },  
  plugins: PLUGINS  
}, {  
  input: "src/node.ts",  
  output: { file: "dist/node.js", format: "cjs" },  
  plugins: PLUGINS  
}, {  
  input: "src/node.ts",  
  output: { file: "dist/node.esm.js", format: "esm" },  
  plugins: PLUGINS  
}];
```

重新执行 `npm run build` 打包工具库，输出以下信息。这是因为在普通情况下，`rollup` 只会解析相对模块ID，意味着导入语句 `import Xyz from "xyz"` 不会让 `Npm` 模块 应用到 `bundle` 文件中。

```
src/node.ts → dist/node.esm.js...  
(!) Unresolved dependencies  
https://rollupjs.org/guide/en/#warning-treating-module-as-external-dependency  
dayjs (imported by src/common/date.ts)  
fs (imported by src/node/fs.ts)
```

```
path (imported by src/node/fs.ts)
process (imported by src/node/fs.ts)
fs-extra (imported by src/node/fs.ts)
os (imported by src/node/os.ts, src/node/type.ts)
child_process (imported by src/node/process.ts)
```

若要让 **Npm** 模块 应用到 **bundle** 文件中，需告知 **rollup** 如何找到它。安装 [@rollup/plugin-node-resolve](#)，使用该插件自动寻找引用到的 **Npm** 模块。还有一个细节需注意，有些 **Npm** 模块 在引用时导入的 **bundle** 文件的 模块规范 可能是 **CJS**，例如 **jquery**、**day** 等，而 **rollup** 在普通情况下无法解析 **CJS**，贸然引用这些 **Npm** 模块 肯定会在打包流程中报错。安装 [@rollup/plugin-commonjs](#)，使用该插件将 **CJS** 转换为 **ESM** 再让其参与到后续编译中。

```
npm i -D @rollup/plugin-commonjs @rollup/plugin-node-resolve
```

改造 **rollup.config.js** 代码，将上述插件实例加入到 **PLUGINS** 中。

```
import CommonjsPlugin from "@rollup/plugin-commonjs";
import NodeResolvePlugin from "@rollup/plugin-node-resolve";

const PLUGINS = [
  NodeResolvePlugin(),
  CommonjsPlugin(),
  TypescriptPlugin()
];
```

js

重新执行 **npm run build** 打包工具库，无任何报错信息就代表打包成功了。打开 **dist** 文件夹可发现5个 **bundle** 文件，这些文件都是该工具库的工程文件，它们都是在生产环境中被使用 **Npm** 模块 主体文件。

稍等，发现基于 **tsconfig.json** 文件生成的 **d.ts** 声明文件都是根据 **src** 文件夹中的目录结构生成，这么多声明文件也增加了 **VSCode** 的解析负担，那可将这些声明文件合并成一个声明文件吗？安装 [rollup-plugin-dts](#)，使用该插件合并声明文件。在此有一个小技巧，在 **src** 文件夹中创建 **index.ts**，收集所有 **Web** 与 **Node** 的工具函数，可查看 [index.ts](#)，因为类型检查不涉及不同环境独有的对象或模块，因此可合并在一起。

```
npm i -D rollup-plugin-dts
```

改造 `rollup.config.js` 代码，为 `src/index.ts` 增加一份打包配置。

```
import DtsPlugin from "rollup-plugin-dts";  
  
export default [{  
  // ...  
}, {  
  input: "src/index.ts",  
  output: { file: "dist/index.d.ts", format: "esm" },  
  plugins: [DtsPlugin()]  
}];
```

重新执行 `npm run build` 打包工具库，最终只生成一份声明文件。到此整个打包流程已完成，可输出多种 模块规范 的 `bundle` 文件 了。

其实还有两个细节可继续优化。多次打包可能都是一个逐渐完善配置的过程，该过程会产生很多不理想的 `bundle` 文件，因此每次打包前需清理 `dist` 文件夹。打包出来的 `bundle` 文件 还会保持很多的源码结构，因此压缩 `bundle` 文件 还能让文件体积更小。安装[rollup-plugin-cleandir](#)与[rollup-plugin-terser](#)，使用这些插件分别在打包前清理 `dist` 文件夹，压缩输出代码。

```
npm i -D rollup-plugin-cleandir rollup-plugin-terser
```

改造 `rollup.config.js` 代码，将上述插件实例加入到 `PLUGINS` 中，最终的 `rollup.config.js` 如下。

```
import CommonjsPlugin from "@rollup/plugin-commonjs";  
import NodeResolvePlugin from "@rollup/plugin-node-resolve";  
import TypescriptPlugin from "@rollup/plugin-typescript";  
import { cleandir as CleandirPlugin } from "rollup-plugin-cleandir";  
import DtsPlugin from "rollup-plugin-dts";  
import { terser as TerserPlugin } from "rollup-plugin-terser";  
  
const PLUGINS = [  
  NodeResolvePlugin(),  
  CommonjsPlugin(),  
  TypescriptPlugin(),  
  TerserPlugin({  
    compress: { drop_console: false },  
    format: { comments: false }  
  })  
];
```

```

export default [{
  input: "src/web.ts",
  output: { file: "dist/web.js", format: "cjs" },
  plugins: [...PLUGINS, CleandirPlugin("dist")]
}, {
  input: "src/web.ts",
  output: { file: "dist/web.esm.js", format: "esm" },
  plugins: PLUGINS
}, {
  input: "src/web.umd.ts",
  output: { file: "dist/web.umd.js", format: "umd", name: "BruceUs" },
  plugins: PLUGINS
}, {
  input: "src/node.ts",
  output: { file: "dist/node.js", format: "cjs" },
  plugins: PLUGINS
}, {
  input: "src/node.ts",
  output: { file: "dist/node.esm.js", format: "esm" },
  plugins: PLUGINS
}, {
  input: "src/index.ts",
  output: { file: "dist/index.d.ts", format: "esm" },
  plugins: [DtsPlugin()]
}
];

```

重新执行 `npm run build` 打包工具库，输出更小体积的 `bundle` 文件。在 `package.json` 中指定 `main` 与 `types`，分别指向 默认入口文件 与 默认声明文件。当引用或打包该 `Npm` 模块 时会直接使用 `main` 指向的入口文件，当 `VSCode` 开启类型检验时会直接使用 `types` 指向的声明文件。

总结

本章从零到一完成一个工具库的打包流程，完整代码可查看[bruce-us](#)。组件库也同样是基于上述流程操作，然后多了一些样式处理。通过一步一步上手 `rollup`，完成一个通用工具库的 `前端工程化` 模板，除了 `src` 文件夹中的业务代码，其余代码基本上可移植到新的工具库中。

当然还有些细节性的问题会放到后续章节中讲述，慢慢将 `前端工程化` 应用起来，是作为开发者的终极目标。

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 (7)

qingkooo  前端开发 1月前

src/index.ts 通过手工方式把所有模块都罗列进去, 会不会有点傻呀? 弱问这是最佳实践吗?

👍 点赞 🗨 回复

qingkooo  前端开发 1月前

章节“初始项目骨架”中描述“name自行定义, 记得执行npm config <pkg>查重就好。”我在官方文档没有找到这种用法, 请问这是在做什么查重呢? 参考: docs.npmjs.com

👍 点赞 🗨 回复

qingkooo  前端开发 1月前

在“原理”章节中, “同时在Web中除了<script>外还能让JS真正拥有模块化的能力。

”

这句话怎么理解, 请同学和老师帮忙答疑解惑下吧?

👍 点赞 🗨 回复



lelecao   前端 3月前

👍 点赞 🗨 回复



seer2  前端开发 4月前

很不错 每篇都有总结

👍 点赞 🗨 1



JowayYoung (作者) 4月前

感谢支持😊

👍 点赞 💬 回复



涵信



前端工程师 @ 个推 4月前

第一个😁

👍 1 💬 回复