



Vite 构建基石(上)——Rollup 打包基本概念及使用

发布于 2022-05-09

Rollup 是一款基于 ES Module 模块规范实现的 JavaScript 打包工具，在前端社区中赫赫有名，同时也在 Vite 的架构体系中发挥着重要作用。不仅是 Vite 生产环境下的打包工具，其插件机制也被 Vite 所兼容，可以说是 Vite 的构建基石。因此，掌握 Rollup 也是深入学习 Vite 的必经之路。

接下来，我们将通过两小节系统学习 Rollup。本节主要围绕 Rollup 的基本概念和核心特性展开，你不仅能知道 Rollup 是如何打包项目的，还能学会 Rollup 更高阶的使用方式，甚至能够通过 JavaScript API 二次开发 Rollup。

快速上手

首先让我们用 `npm init -y` 新建一个项目，然后安装 `rollup` 依赖：

```
pnpm i rollup
```

接着新增 `src/index.js` 和 `src/util.js` 和 `rollup.config.js` 三个文件，目录结构如下所示：

```
.
├─ package.json
├─ pnpm-lock.yaml
├─ rollup.config.js
└─ src
  ├─ index.js
  └─ util.js
```

文件的内容分别如下：

```
// src/index.js
import { add } from './util';
console.log(add(1, 2));
```

```
// src/util.js
export const add = (a, b) => a + b;

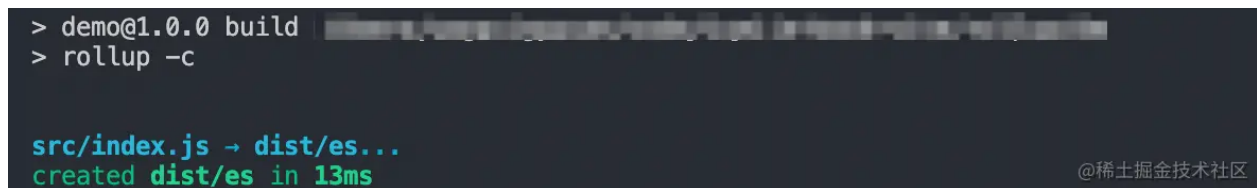
export const multi = (a, b) => a * b;
// rollup.config.js
// 以下注释是为了能使用 VSCode 的类型提示
/**
 * @type { import('rollup').RollupOptions }
 */
const buildOptions = {
  input: ["src/index.js"],
  output: {
    // 产物输出目录
    dir: "dist/es",
    // 产物格式
    format: "esm",
  },
};

export default buildOptions;
```

你可以在 `package.json` 中加入如下的构建脚本:

```
{
  // rollup 打包命令, `-c` 表示使用配置文件中的配置
  "build": "rollup -c"
}
```

接着在终端执行一下 `npm run build` , 可以看到如下的命令行信息:



```
> demo@1.0.0 build
> rollup -c

src/index.js -> dist/es...
created dist/es in 13ms
```

OK, 现在你已经成功使用 Rollup 打出了第一份产物! 我们可以去 `dist/es` 目录查看一下产物的内容:

```
// dist/es/index.js
// 代码已经打包到一起
const add = (a, b) => a + b;

console.log(add(1, 2));
```

同时你也可以发现, `util.js` 中的 `multi` 方法并没有被打包到产物中, 这是因为 Rollup 具有天然的 `Tree Shaking` 功能, 可以分析出未使用到的模块并自动擦除。

所谓 **Tree Shaking** (摇树)，也是计算机编译原理中 **DCE** (Dead Code Elimination，即消除无用代码) 技术的一种实现。由于 ES 模块依赖关系是确定的，和运行时状态无关。因此 Rollup 可以在编译阶段分析出依赖关系，对 AST 语法树中没有使用到的节点进行删除，从而实现 Tree Shaking。

常用配置解读

1. 多产物配置

在打包 JavaScript 类库的场景中，我们通常需要对外暴露出不同格式的产物供他人使用，不仅包括 **ESM**，也需要包括诸如 **CommonJS**、**UMD** 等格式，保证良好的兼容性。那么，同一份入口文件，如何让 Rollup 给我们打包出不一样格式的产物呢？我们基于上述的配置文件来进行修改：

```
// rollup.config.js
/**
 * @type { import('rollup').RollupOptions }
 */
const buildOptions = {
  input: ["src/index.js"],
  // 将 output 改造成一个数组
  output: [
    {
      dir: "dist/es",
      format: "esm",
    },
    {
      dir: "dist/cjs",
      format: "cjs",
    },
  ],
};

export default buildOptions;
```

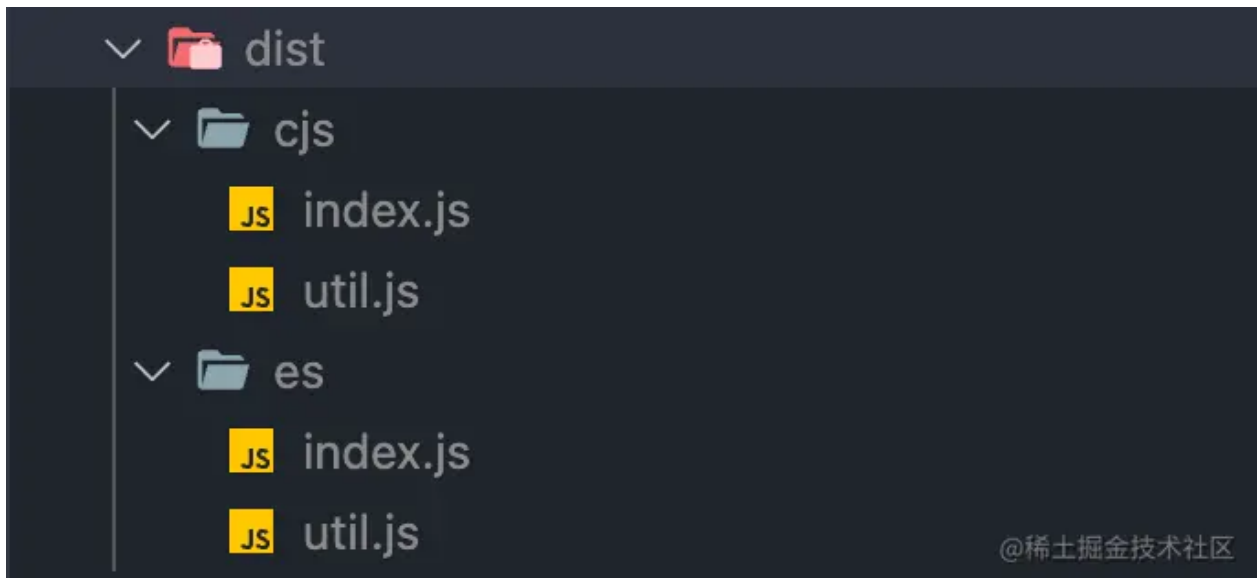
从代码中可以看到，我们将 **output** 属性配置成一个数组，数组中每个元素都是一个描述对象，决定了不同产物的输出行为。

2. 多入口配置

除了多产物配置，Rollup 中也支持多入口配置，而且通常情况下两者会被结合起来使用。接下来，就让我们继续改造之前的配置文件，将 `input` 设置为一个数组或者一个对象，如下所示：

```
{
  input: ["src/index.js", "src/util.js"]
}
// 或者
{
  input: {
    index: "src/index.js",
    util: "src/util.js",
  },
}
```

通过执行 `npm run build` 可以发现，所有入口的不同格式产物已经成功输出：



如果不同入口对应的打包配置不一样，我们也可以默认导出一个 `配置数组`，如下所示：

```
// rollup.config.js
/**
 * @type { import('rollup').RollupOptions }
 */
const buildIndexOptions = {
  input: ["src/index.js"],
  output: [
    // 省略 output 配置
  ],
};

/**
 * @type { import('rollup').RollupOptions }
 */
const buildUtilOptions = {
  input: ["src/util.js"],
  output: [
    // 省略 output 配置
  ],
};
```

```

input: ["src/util.js"],

output: [
  // 省略 output 配置
],
};

export default [buildIndexOptions, buildUtilOptions];

```

如果是比较复杂的打包场景(如 [Vite 源码本身的打包](#))，我们需要将项目的代码分成几个部分，用不同的 Rollup 配置分别打包，这种配置就很有用了。

3. 自定义 output 配置

刚才我们提到了 `input` 的使用，主要用来声明入口，可以配置成字符串、数组或者对象，使用比较简单。而 `output` 与之相对，用来配置输出的相关信息，常用的配置项如下：

```

output: {
  // 产物输出目录
  dir: path.resolve(__dirname, 'dist'),
  // 以下三个配置项都可以使用这些占位符：
  // 1. [name]: 去除文件后缀后的文件名
  // 2. [hash]: 根据文件名和文件内容生成的 hash 值
  // 3. [format]: 产物模块格式，如 es、cjs
  // 4. [extname]: 产物后缀名(带`.`)
  // 入口模块的输出文件名
  entryFileNames: `[name].js`,
  // 非入口模块(如动态 import)的输出文件名
  chunkFileNames: `chunk-[hash].js`,
  // 静态资源文件输出文件名
  assetFileNames: `assets/[name]-[hash][extname]`,
  // 产物输出格式，包括`amd`、`cjs`、`es`、`iife`、`umd`、`system`
  format: 'cjs',
  // 是否生成 sourcemap 文件
  sourcemap: true,
  // 如果是打包出 iife/umd 格式，需要对外暴露出一个全局变量，通过 name 配置变量名
  name: 'MyBundle',
  // 全局变量声明
  globals: {
    // 项目中可以直接用`$`代替`jquery`
    jquery: '$'
  }
}

```

4. 依赖 external

对于某些第三方包，有时候我们不想让 Rollup 进行打包，也可以通过 `external` 进行外部化：

```
{
  external: ['react', 'react-dom']
}
```

在 SSR 构建或者使用 ESM CDN 的场景中，这个配置将非常有用，具体细节我们会在**高级应用**这一章展开。

5. 接入插件能力

在 Rollup 的日常使用中，我们难免会遇到一些 Rollup 本身不支持的场景，比如 **兼容 CommonJS 打包**、**注入环境变量**、**配置路径别名**、**压缩产物代码** 等等。这个时候就需要我们引入相应的 Rollup 插件了。接下来以一个具体的场景为例带大家熟悉一下 Rollup 插件的使用。

虽然 Rollup 能够打包 **输出** 出 **CommonJS** 格式的产物，但对于 **输入** 给 Rollup 的代码并不支持 CommonJS，仅仅支持 ESM。你可能会说，那我们直接在项目中统一使用 ESM 规范就可以了啊，这有什么问题呢？需要注意的是，我们不光要考虑项目本身的代码，还要考虑第三方依赖。目前为止，还是有不少第三方依赖只有 CommonJS 格式产物而并未提供 ESM 产物，比如项目中用到 **lodash** 时，打包项目会出现这样的报错：

```
[!] Error: 'merge' is not exported by node_modules/.pnpm/registry.npmirror.com+lodash@4.17.21/node_modules/lodash/lodash.js, imported by src/index.js
https://rollupjs.org/guide/en/#error-name-is-not-exported-by-module
src/index.js (6:9)
4:
5: import { add, multi } from "./util";
6: import { merge } from "lodash";
   ^
7: console.log(merge);
8: console.log(add(1, 2));
Error: 'merge' is not exported by node_modules/.pnpm/registry.npmirror.com+lodash@4.17.21/node_modules/lodash/lodash.js, imported by src/index.js
@稀土掘金技术社区
```

因此，我们需要引入额外的插件去解决这个问题。

首先需要安装两个核心的插件包：

```
pnpm i @rollup/plugin-node-resolve @rollup/plugin-commonjs
```

- **@rollup/plugin-node-resolve** 是为了允许我们加载第三方依赖，否则像 **import React from 'react'** 的依赖导入语句将不会被 Rollup 识别。
- **@rollup/plugin-commonjs** 的作用是将 CommonJS 格式的代码转换为 ESM 格式

然后让我们在配置文件中导入这些插件：

```
// rollup.config.js
import resolve from "@rollup/plugin-node-resolve";
import commonjs from "@rollup/plugin-commonjs";

/**
 * @type { import('rollup').RollupOptions }
 */
export default {
  input: ["src/index.js"],
  output: [
    {
      dir: "dist/es",
      format: "esm",
    },
    {
      dir: "dist/cjs",
      format: "cjs",
    },
  ],
  // 通过 plugins 参数添加插件
  plugins: [resolve(), commonjs()],
};
```

现在我们以 `lodash` 这个只有 CommonJS 产物的第三方包为例测试一下:

```
pnpm i lodash
```

在 `src/index.js` 加入如下的代码:

```
import { merge } from "lodash";
console.log(merge);
```

然后执行 `npm run build` , 你可以发现产物已经正常生成了:

```
rollup > demo > dist > es > js index.js > { } <function> > { } <function>

9  /**
10  * @license
11  * Lodash <https://lodash.com/>
12  * Copyright OpenJS Foundation and other contributors <https://openjsf.org/>
13  * Released under MIT license <https://lodash.com/license>
14  * Based on Underscore.js 1.8.3 <http://underscorejs.org/LICENSE>
15  * Copyright Jeremy Ashkenas, DocumentCloud and Investigative Reporters & Editors
16  */
17
18  (function (module, exports) {
19    (function() {
20
21      /** Used as a safe reference for `undefined` in pre-ES5 environments. */
22      var undefined$1;
23
24      /** Used as the semantic version number. */
25      var VERSION = '4.17.21';
26
27      /** Used as the size to enable large array optimizations. */
28      var LARGE_ARRAY_SIZE = 200;
29
30      /** Error message constants. */
31      var CORE_ERROR_TEXT = 'Unsupported core-js use. Try https://npms.io/search?q=ponyfill.',
32          FUNC_ERROR_TEXT = 'Expected a function',
33          INVALID_TEMPL_VAR_ERROR_TEXT = 'Invalid `variable` option passed into `_.template`';
34
35      @稀土掘金技术社区
```

在 Rollup 配置文件中，`plugins` 除了可以与 `output` 配置在同一级，也可以配置在 `output` 参数里面，如：

```
// rollup.config.js
import { terser } from 'rollup-plugin-terser'
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';

export default {
  output: {
    // 加入 terser 插件，用来压缩代码
    plugins: [terser()]
  },
  plugins: [resolve(), commonjs()]
}
```

当然，你可以将上述的 `terser` 插件放到最外层的 `plugins` 配置中。

需要注意的是，`output.plugins` 中配置的插件是有一定限制的，只有使用 `Output` 阶段相关钩子(具体内容将在下一节展开)的插件才能够放到这个配置中，大家可以去[这个站点](#)查看 Rollup 的 `Output` 插件列表。

另外，这里也给大家分享其它一些比较常用的 Rollup 插件库：

- [@rollup/plugin-json](#)：支持 `.json` 的加载，并配合 `rollup` 的 `Tree Shaking` 机制去掉未使用的部分，进行按需打包。

- [@rollup/plugin-babel](#): 在 Rollup 中使用 Babel 进行 JS 代码的语法转译。
- [@rollup/plugin-typescript](#): 支持使用 TypeScript 开发。
- [@rollup/plugin-alias](#): 支持别名配置。
- [@rollup/plugin-replace](#): 在 Rollup 进行变量字符串的替换。
- [rollup-plugin-visualizer](#): 对 Rollup 打包产物进行分析，自动生成产物体积可视化分析图。

JavaScript API 方式调用

以上我们通过 `Rollup` 的配置文件结合 `rollup -c` 完成了 Rollup 的打包过程，但有些场景下我们需要基于 Rollup 定制一些打包过程，配置文件就不够灵活了，这时候我们需要用到对应 JavaScript API 来调用 Rollup，主要分为 `rollup.rollup` 和 `rollup.watch` 两个 API，接下来我们以具体的例子来学习一下。

首先是 `rollup.rollup`，用来一次性地进行 Rollup 打包，你可以新建 `build.js`，内容如下：

```
// build.js
const rollup = require("rollup");

// 常用 inputOptions 配置
const inputOptions = {
  input: "./src/index.js",
  external: [],
  plugins: []
};

const outputOptionsList = [
  // 常用 outputOptions 配置
  {
    dir: 'dist/es',
    entryFileNames: `[name].[hash].js`,
    chunkFileNames: 'chunk-[hash].js',
    assetFileNames: 'assets/[name]-[hash][extname]',
    format: 'es',
    sourcemap: true,
    globals: {
      lodash: '_'
    }
  }
  // 省略其它的输出配置
];

async function build() {
  let bundle;
```

```

let bundle,
let buildFailed = false;
try {
  // 1. 调用 rollup.rollup 生成 bundle 对象
  bundle = await rollup.rollup(inputOptions);
  for (const outputOptions of outputOptionsList) {
    // 2. 拿到 bundle 对象，根据每一份输出配置，调用 generate 和 write 方法分别生成和写入产物
    const { output } = await bundle.generate(outputOptions);
    await bundle.write(outputOptions);
  }
} catch (error) {
  buildFailed = true;
  console.error(error);
}
if (bundle) {
  // 最后调用 bundle.close 方法结束打包
  await bundle.close();
}
process.exit(buildFailed ? 1 : 0);
}

build();

```

主要的执行步骤如下:

- 通过 `rollup.rollup` 方法，传入 `inputOptions`，生成 `bundle` 对象；
- 调用 `bundle` 对象的 `generate` 和 `write` 方法，传入 `outputOptions`，分别完成产物和生成和磁盘写入。
- 调用 `bundle` 对象的 `close` 方法来结束打包。

接着你可以执行 `node build.js`，这样，我们就可以完成了以编程的方式来调用 Rollup 打包的过程。

除了通过 `rollup.rollup` 完成一次性打包，我们也可以通过 `rollup.watch` 来完成 `watch` 模式下的打包，即每次源文件变动后自动进行重新打包。你可以新建 `watch.js` 文件，内容入下:

```

// watch.js
const rollup = require("rollup");

const watcher = rollup.watch({
  // 和 rollup 配置文件中的属性基本一致，只不过多了`watch`配置
  input: "./src/index.js",
  output: [
    {
      dir: "dist/es",
      format: "esm",
    },
    {

```

```

    dir: "dist/cjs",

    format: "cjs",
  },
],
watch: {
  exclude: ["node_modules/**"],
  include: ["src/**"],
},
});

// 监听 watch 各种事件
watcher.on("restart", () => {
  console.log("重新构建...");
});

watcher.on("change", (id) => {
  console.log("发生变动的模块id: ", id);
});

watcher.on("event", (e) => {
  if (e.code === "BUNDLE_END") {
    console.log("打包信息:", e);
  }
});

```

现在你可以通过执行 `node watch.js` 开启 Rollup 的 watch 打包模式，当你改动一个文件后可以看到如下的日志，说明 Rollup 自动进行了重新打包，并触发相应的事件回调函数：

```

发生变动的模块id: /xxx/src/index.js
重新构建...
打包信息: {
  code: 'BUNDLE_END',
  duration: 10,
  input: './src/index.js',
  output: [
    // 输出产物路径
  ],
  result: {
    cache: { /* 产物具体信息 */ },
    close: [AsyncFunction: close],
    closed: false,
    generate: [AsyncFunction: generate],
    watchFiles: [
      // 监听文件列表
    ],
    write: [AsyncFunction: write]
  }
}

```

基于如上的两个 JavaScript API 我们可以很方便地在代码中调用 Rollup 的打包流程，相比于配置文件有了更多的操作空间，你可以在代码中通过这些 API 对 Rollup 打包过程进

行定制，甚至是二次开发。

小结

好，本文的内容就到这里了。恭喜你，学习了完了 Rollup 基础使用篇的内容，在本小节中，你需要重点掌握 **Rollup 基本配置项的含义**和 **JavaScript API 的使用**。

首先，我们通过一个简单的示例学习了 Rollup 一般的使用方法，用 Rollup 打包出了第一份产物，然后我们针对 Rollup 中常用的配置项进行介绍，包括 `input`、`output`、`external`、`plugins` 等核心配置，并以一个实际的打包场景为例带你在 Rollup 中接入插件功能。接着，我给你介绍了 Rollup 更高级的使用姿势——通过 JavaScript API 使用，分别介绍了两个经典的 API: `rollup.rollup` 和 `rollup.watch`，并带你在实战中使用这些 API，完成了更为复杂的打包操作。

在下一小节中，我们将继续深入 Rollup 的学习，一起剖析 Rollup 优秀的插件机制并进行代码实战，大家加油！

上一篇：得力的性能推手: Esbuild 功能使用与插件开发实战

下一篇：Vite 构建基石(下)——深入理解 Rollup 的插件机制