rollup.js工程工具

## H2 简介

Rollup 是一个 JavaScript 模块打包器，可以将小块代码编译成大块复杂的代码，例如 library 或应用程序。Rollup 对代码模块使用新的标准化格式，这些标准都包含在 JavaScript 的 ES6 版本中，而不是以前的特殊解决方案，如 CommonJS 和 AMD。ES6 模块可以使你自由、无缝地使用你最喜爱的 library 中那些最有用独立函数，而你的项目不必携带其他未使用的代码。

## H2 与其他的打包工具的对比

| 名称 | 简介 | 开源项目 |
| --- | --- | --- |
| rollup | 是一个 JavaScript 模块打包器，可以将小块代码编译成大块复杂的代码，例如 library 或应用程序 | vue, vue-router, react |
| webpack | 是一个现代 JavaScript 应用程序的静态模块打包器(module bundler)。当 webpack 处理应用程序时，它会递归地构建一个依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。 | element-ui, mint-ui, vue-cli |

Rollup偏向应用于js库/框架等，webpack偏向应用于前端工程，UI库；如果你的应用场景中只是js代码，希望做ES转换，模块解析，可以使用Rollup。如果你的场景中涉及到css、html，涉及到复杂的代码拆分合并，建议使用webpack。

## H2 基本使用

## 安装

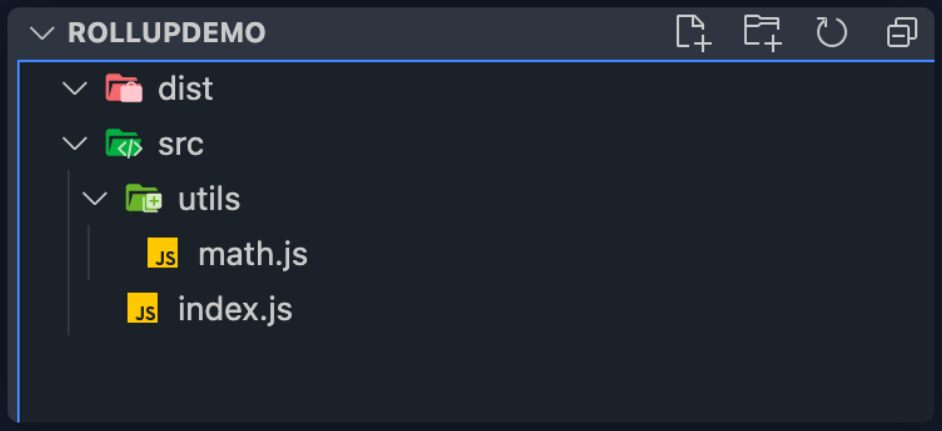
```
npm install --global rollup
```

H3

和npm 类似, 因为后期可能需要在各处用到, 所以可以直接全局进行安装, 如果仅仅只在当前项目使用, 后面会教如何在当前项目配置rollup

## 搭建项目基本结构

H3




## 基本业务代码

math.js

H3

```
function add(...args){
    return args.reduce((pre,cur)=>{
        return pre+cur;
    })
}


function multiplication(...args){
    return args.reduce((pre,cur)=>{
        return pre*cur;
    })
}

export {
    add,
    multiplication
}
```

index.js

```
import { add } from "./utils/math";

function test(...args){
    console.log(args);
    return add(args);
}

console.log(test(1,2,3,4,5));
```

## 打包

### 打包的指令详解

H3

```
rollup --input  ./src/index.js  --file ./dist/bundle.js  --format iife
## rollup --input 项目入口文件地址 --file 打包后文件的地址和名字  --format 打包模式
```

H4

在rollup中, 常用的相关指令有

```
-i, --input <filename>          要打包的文件（必须）
-o, --file <output>             输出的文件 (如果没有这个参数，则直接输出到控制台)
-f, --format <format>           输出的文件类型 (amd, cjs, esm, iife, umd)
-e, --external <ids>            将模块ID的逗号分隔列表排除
-g, --globals <pairs>           以`module ID:Global` 键值对的形式，用逗号分隔开
                                  任何定义在这里模块ID定义添加到外部依赖
-n, --name <name>               生成UMD模块的名字
-h, --help                      输出 help 信息
-m, --sourcemap                 生成 sourcemap (`-m inline` for inline map)
--amd.id                        AMD模块的ID，默认是个匿名函数
--amd.define                    使用Function来代替`define`
--no-strict                     在生成的包中省略`"use strict";`
--no-conflict                   对于UMD模块来说，给全局变量生成一个无冲突的方法
--intro                         在打包好的文件的块的内部(wrapper内部)的最顶部插入
一段内容
--outro                         在打包好的文件的块的内部(wrapper内部)的最底部插入
一段内容
--banner                        在打包好的文件的块的外部(wrapper外部)的最顶部插入
一段内容
--footer                        在打包好的文件的块的外部(wrapper外部)的最底部插入
一段内容
--interop                       包含公共的模块（这个选项是默认添加的）
```

在rollup中, 打包模式大致有以下几种

- **cjs** : CommonJS, 支援 Node.js
- **esm** : 作为 ES module 文件，现代浏览器中用 `<script type=module>` 标签可直接支持
- **iife** : 立即执行函数，可直接使用 `<script>` 标签。（如果你想打包你的前端应用，也可以用这种方式）
- **umd** : Universal Module Definition，通用模块定义，直接封装 amd、cjs、iife 三种方式并根据环境自动切换
- **amd** : Asynchronous Module Definition，异步模块定义，以 RequireJS 为代表
- **system** : SystemJS 的方式

Intro, outro, banner, footer等指令是为了能够在打包后的文件里面增加一些基本内容, 像是框架的基本信息等, 比如:

```
rollup --input  ./src/index.js  --file ./dist/bundle.js  --format iife
 --banner 'hello world'
```

```
(function () {
    'use strict';

    hello world

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

}());
```

```
rollup --input  ./src/index.js  --file ./dist/bundle.js  --format iife
 --intro 'hello world'
```

#### 打包后的项目目录结构

H4


bundle.js

```
(function () {
    'use strict';

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

}());
```

### rollup打包的基本特性

H3

如果你将项目拆分成小的单独文件中，这样开发软件通常会很简单，因为这通常会消除无法预知的相互影响(remove unexpected interaction)，以及显著降低了所要解决的问题的复杂度(complexity of the problem)，并且可以在项目最初时，就简洁地编写小的项目（不一定是标准答案）。不幸的是，JavaScript 以往并没有将此功能作为语言的核心功能。

除了使用 ES6 模块之外，Rollup 还静态分析代码中的 import，并将排除任何未实际使用的代码。这允许您架构于现有工具和模块之上，而不会增加额外的依赖或使项目的大小膨胀。

## 不同打包模式的比对

H2

## CommonJS

CommonJS 是同步加载模块。具有以下一些基本特性

H3

- 一个文件为一个模块
- 使用 `exports.xxx = ...` 或 `module.exports = {...}` 暴露模块
- 使用 `require(...)` 方法来引入一个模块
- `require(...)` 是同步执行

在一个模块内的变量是不能被另一个模块直接访问的。也就是说在一个文件中，定义的一个变量，在另一个文件中是不能直接访问的，如果需要访问这个变量，则需通过 module.exports 这个 API (暴露模块接口) 来暴露它，让外界能够访问这些东西

CommonJS 在 NodeJS 环境用，不适用于浏览器端. 在服务端，模块文件都存在本地磁盘，读取非常快，所以这样做不会有问题。但是在浏览器端，限于网络原因，更合理的方案是使用异步加载。如果同样采用同步加载的话, 那么界面线程就会被网络IO给阻塞, 这样的话就会导致用户在浏览界面的时候, 产生大量的时间等待

## AMD

AMD全称 Asynchronous module definition（异步模块定义）,具有以下一些基本特性

H3

- 使用 `define(...)` 定义一个模块
- 使用 `require(...)` 加载一个模块（和 CommonJS 规范是相同的方法名，注意区分 CommonJS 和 **RequireJS**）
- 依赖前置，提前执行

**RequireJS** 是 CMD 的一种实现

**代码示例**

```
define(
    // 模块名
    "alpha",
    // 依赖
    ["require", "exports", "beta"],
    // 模块输出
    function (require, exports, beta) {
        exports.verb = function() {
            return beta.verb();
            //Or:
            return require("beta").verb();
        }
    }
);
```

## CMD

全称 Common Module Definition（通用模块定义）

- 一个文件为一个模块
- 使用 `define(...)` 定义一个模块（和 AMD 相似）
- 使用 `require(...)` 加载一个模块（和 AMD 相似）
- 尽可能懒执行（和 AMD 的不同点）

H3

SeaJS 是 CMD 的一种实现

```
define(function(require, exports, module) {
    var $ = require('jquery');
    var Spinning = require('./spinning');

    // 通过 exports 对外提供接口
    // 注：不能直接对 exports 赋值，例如：exports = {...}
    exports.doSomething = ...

    // 或通过 module.exports 提供整个接口
    module.exports = ...
})
```

CMD 和 AMD 的最显著区别 “as lazy as possible”.

- CMD 推崇 as lazy as possible（尽可能的懒加载，也称为延迟加载，即在需要的时候才加载)。对于依赖的模块，AMD 是提前执行，CMD 是延迟执行，两者执行方式不一样，AMD 执行过程中会将所有依赖前置执行，也就是在自己的代码逻辑开始前全部执行；而 CMD 如果 require了, 但是整个逻辑并未使用这个依赖 或 未执行到逻辑使用它的地方前不会执行。
- CMD 推崇依赖就近，AMD 推崇依赖前置。

```
// CMD
define(function(require, exports, module) {
    var a = require('./a');
    a.doSomething();
    // 此处略去 100 行
    var b = require('./b'); // 依赖可以就近书写
    b.doSomething();
    // ...
});

// AMD 默认推荐的是
define(['./a', './b'], function(a, b) { // 依赖必须一开始就写好
    a.doSomething();
    // 此处略去 100 行
    b.doSomething();
    //...
```

```
});
```

## UMD

全称 Universal Module Definition（万能模块定义），从名字就可以看出 UMD 做的是大一统的工作。Webpack 打包代码就有 UMD 这个选项。

它会做三件事情：

H3

- 判断是否支持 AMD
- 判断是否支持 CommonJS
- 如果都不支持，使用全局变量

```
// if the module has no dependencies, the above pattern can be simplified to
(function (root, factory) {

    // Environment Detection

    // 对应上述的三个步骤
    if (typeof define === 'function' && define.amd) {
        // 1.判断是否支持 AMD
        // 如果 define 这个方法是被定义 并且 define 这个方法是 AMD 的规范，那就把 factory 这个模块实体用 define 方法以 AMD 的规范 定义
        define([], factory); // [] 是依赖，factory 是模块实体
    } else if (typeof exports === 'object') {
        // 2. 判断是否支持 CommonJS
        // 如果 exports 是等于一个对象，则表明是在 Node 环境中运行，则支持 CommonJS，那就用 module.exports 暴露整个模块实体
        module.exports = factory();
    } else {
        // 3. 如果都不支持，使用全局变量
        // Browser globals (root 即是 window)
        root.returnExports = factory();
  }
}(this, function () {

    // Module Defination

    var sum = function(x, y){
        return x + y;
    }

    var sub = function(x, y){
        return x - y;
    }

    var math = {
```

```
        findSum: function(a, b){
            return sum(a,b);
        },

        findSub: function(a, b){
            return sub(a, b);
        }
    }

    return math;
}));
```

### ES Module (ESM)

全称 ECMAScript Module. ESM 现在比较流行，随着 ESM 规范的普及，开发过程越来越多地中使用 ESM 的模块化规范

ESM 规定

H3

- 一个文件为一个模块
- 引入模块用 `import` 关键字 或 `import(...)` 方法
- 暴露模块用 `export` 关键字（没有 s，注意和 CommonJS 的 `exports.xxx = ...` 或 `module.exports = {}` 区分）

```
// a.js
let helloWorld = function (msg) {
    console.log('Hello World! ' + msg);
};

export {helloWorld};

// b.js
import {helloWorld} from './a.js';
helloWorld('Date: 2018/7/3');

// b.js
import {helloWorld as myTool} from './a.js';
myTool('Date: 2018/7/3');

// b.js
import * as a from './a.js';
a.helloWorld('Date: 2018/7/3');
```

### SystemJS

SystemJS支持在浏览器端和Node动态加载之前介绍过所有格式的模块（ES6 modules, AMD, CommonJS等），通过把已加载的模块还存在"module registry"里来避免重复加载。它也同样支持转换ES6的代码至其他格式。

H3
H2
## rollup 采用不同的打包方式的执行效果

CommonJS

```
'use strict';

function add(...args){
    return args.reduce((pre,cur)=>{
        return pre+cur;
    })
}

function test(...args){
    console.log(args);
    return add(args);
}

console.log(test(1,2,3,4,5));
```

AMD

```
define(function () { 'use strict';

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

});
```

ESM

```
function add(...args){
    return args.reduce((pre,cur)=>{
        return pre+cur;
    })
}

function test(...args){
    console.log(args);
    return add(args);
}

console.log(test(1,2,3,4,5));
```

UMD

```
(function (factory) {
    typeof define === 'function' && define.amd ? define(factory) :
    factory();
}((function () { 'use strict';

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

})));
```

SystemJS

```
System.register([], function () {
    'use strict';
    return {
        execute: function () {
```

```
            function add(...args){
                return args.reduce((pre,cur)=>{
                    return pre+cur;
                })
            }

            function test(...args){
                console.log(args);
                return add(args);
            }

            console.log(test(1,2,3,4,5));

        }
    };
});
```

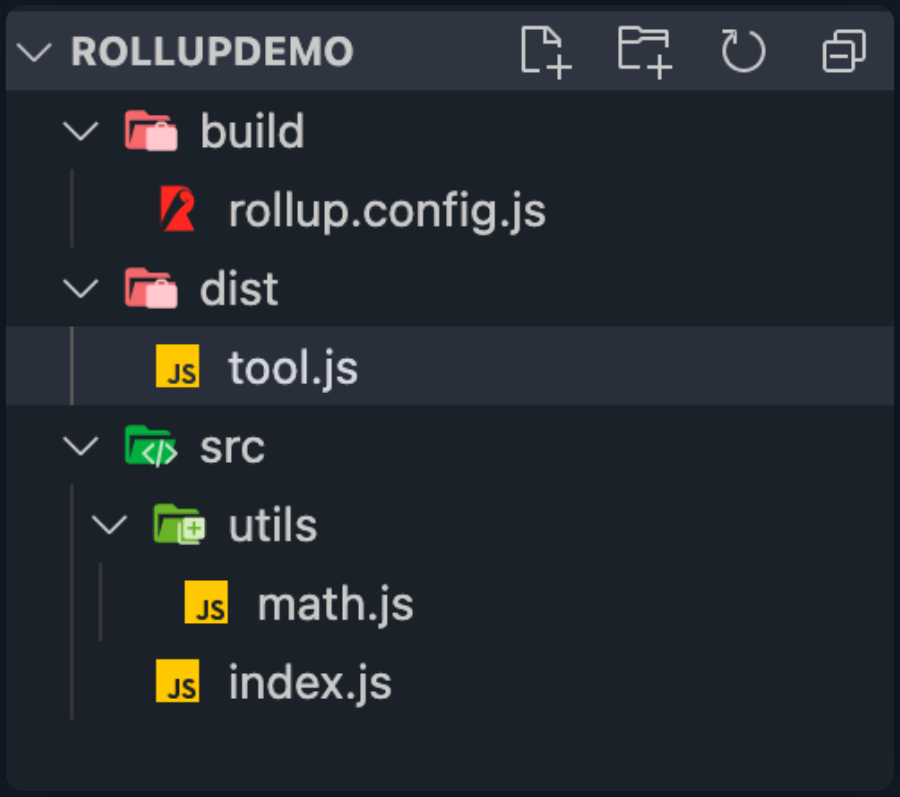
IIFE

```
(function () {
    'use strict';

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

}());
```

## H2 使用配置文件进行打包

## 项目结构

H3




把项目的配置文件数据信息, 统一都放在build的项目目录下面

## 配置信息

Rollup.config.js

H3

```
export default {
    input: 'src/index.js',// 入口文件
    output: { // 输输出文件
        file: 'dist/tool.js', // 输出文件的位置和名字
        format: 'iife', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
        banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
        footer: '/* follow me*/', // 增加注释信息在打包后的文件的头部
        intro: 'var ENVIRONMENT = "production";', // 在主文件里面插入 新的代码
    }
}
```

### 执行打包指令

```
"rollup --c ./build/rollup.config.js"
```

H3

### 打包结果

H3

```
/* my-library version version */
(function () {
    'use strict';

    var ENVIRONMENT = "production";

    function add(...args){
        return args.reduce((pre,cur)=>{
            return pre+cur;
        })
    }

    function test(...args){
        console.log(args);
        return add(args);
    }

    console.log(test(1,2,3,4,5));

}());
/* follow me */
```

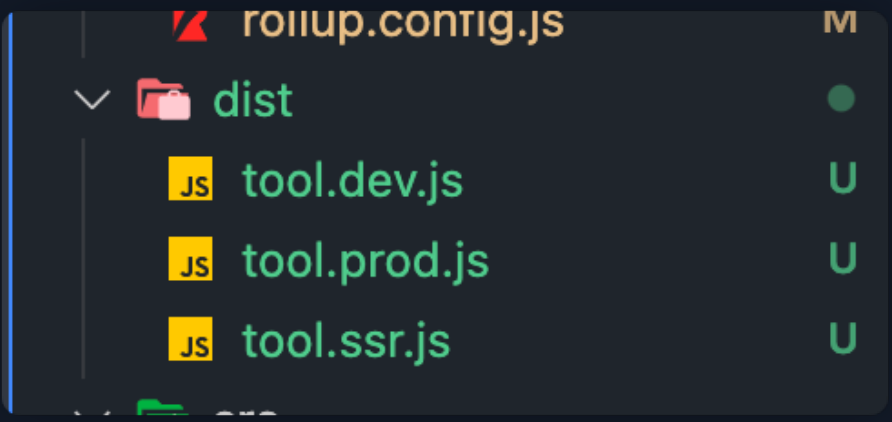
### PS: 可以配置多个输出配置, 已达到给不同使用场景构建轮子

Rollup.config.js

H3

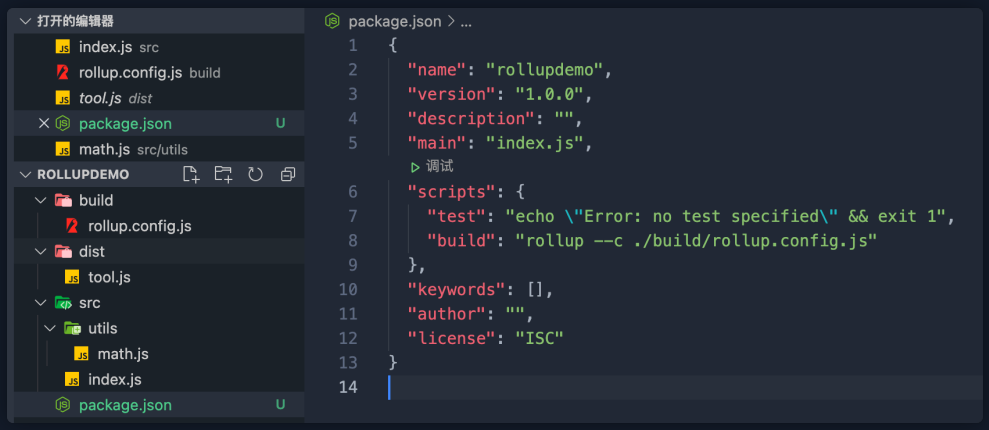
```
export default {
    input: 'src/index.js',// 入口文件
    output: [
        { // 输输出文件
            file: 'dist/tool.prod.js', // 输出文件的位置和名字
            format: 'iife', // 模块系统的格式, esm( es6的模块系统)格式或是 cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* hello 呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "production";', // 在主文件里面插入新的代码
```

```
        },
        { // 输输出文件
            file: 'dist/tool.dev.js', // 输出文件的位置和名字
            format: 'esm', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 老妹呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "devlop";', // 在主文件里面插入 新的代码
        },
        { // 输输出文件
            file: 'dist/tool.ssr.js', // 输出文件的位置和名字
            format: 'cjs', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 我两是一对啊 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "ssr";', // 在主文件里面插入 新的代码
        }
    ]
}
```




## H2 rollup与npm进行集成

在前端的日常开发中 ,我们已经习惯于用npm 来进行大型前端项目的开发与管理, 那么接下来我们就试试把npm和rollup进行集成




```
npm run build
```

这样我们就可以直接使用npm的指令来简化 打包的过程了

## H2 rollup插件系统

在rollup中使用插件, 同样需要使用npm的插件仓库来作为插件的来源

目前为止，我们通过相对路径，将一个入口文件和一个模块创建成了一个简单的 bundle。随着构建更复杂的 bundle，通常需要更大的灵活性——引入 npm 安装的模块、通过 Babel 编译代码、和 JSON 文件打交道等。

为此，我们可以用 *插件(plugins)* 在打包的关键过程中更改 Rollup 的行为。the Rollup wiki 维护了可用的插件列表。

### 安装babel插件

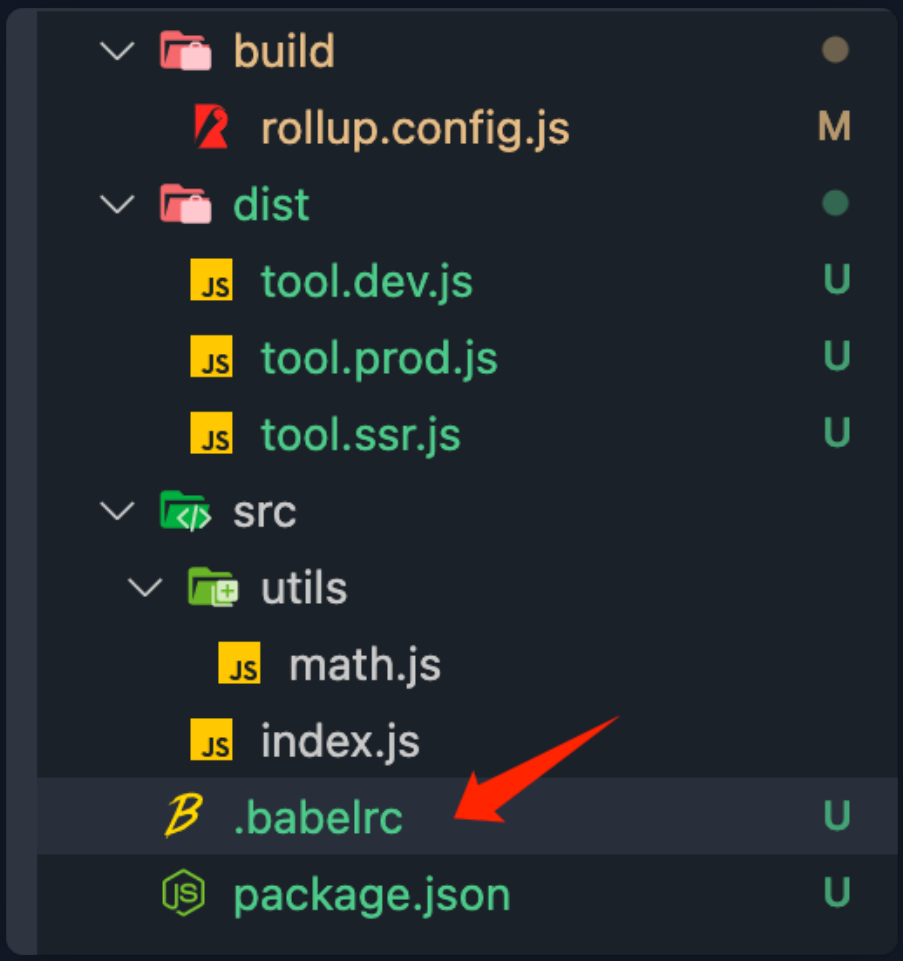
比如安装babel 来对框架的代码进行转义, 保证低版本浏览器也可以正常运行项目代码

H3

```
npm install @babel/core @babel/preset-env @babel/plugin-external-helpers rollup-plugin-babel --save-dev
```

首先安装babel的核心组件

然后在项目中进行配置babelrc的配置文件




.babelrc

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "modules": false
            }
        ]
    ],
    "plugins": ["@babel/plugin-external-helpers"]
}
```

接下来在 rollup.config.js里面引入插件 并进行配置

```
import babel from 'rollup-plugin-babel'; // 直接导入插件

export default {
    input: 'src/index.js',// 入口文件
```

```
    output: [
        { // 输输出文件
            file: 'dist/tool.prod.js', // 输出文件的位置和名字
            format: 'iife', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* hello 呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "production";', // 在主文件里面插入新的代码
        },
        { // 输输出文件
            file: 'dist/tool.dev.js', // 输出文件的位置和名字
            format: 'esm', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 老妹呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "devlop";', // 在主文件里面插入 新的代码
        },
        { // 输输出文件
            file: 'dist/tool.ssr.js', // 输出文件的位置和名字
            format: 'cjs', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 我两是一对啊 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "ssr";', // 在主文件里面插入 新的代码
        }
    ],
    plugins: [
        babel({
            exclude: 'node_modules/**'  // 排除node_module下的所有文件
          })
    ]
}
```

接下来, 我们在js代码里面使用一些ES6版本的js代码来实现效果

index.js

```
import { add } from "./utils/math";

let p = ()=>{
    console.log('吃了没');
}

function test(...args){
    console.log(args);
    return add(args);
}

console.log(test(1,2,3,4,5));
p();
```

Tool.prod.js

```
/* my-library version version */
(function () {
    'use strict';

    var ENVIRONMENT = "production";

    function add() {
      for (var _len = arguments.length, args = new Array(_len), _key =
0; _key < _len; _key++) {
        args[_key] = arguments[_key];
      }

      return args.reduce(function (pre, cur) {
        return pre + cur;
      });
    }

    var p = function p() {
      console.log('吃了没');
    };

    function test() {
      for (var _len = arguments.length, args = new Array(_len), _key =
0; _key < _len; _key++) {
        args[_key] = arguments[_key];
      }

      console.log(args);
      return add(args);
    }
```

```
    console.log(test(1, 2, 3, 4, 5));
    p();

}());
/* hello 呀 */
```

我们可以看到 index.js里面的箭头函数, 解构符号被转义成了能够在ES5里面运行的代码, 但是Promise 方法并没有被转义, 原理与webpack类似, 如果要转义Promise等拓展的数据结构, 必须要引入babel polyfill

## 转义ES6新结构

```
npm install --save @babel/polyfill rollup-plugin-node-resolve rollup-plugin-commonjs
```

H3

package.json

```
{
  "name": "rollupdemo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "rollup --c ./build/rollup.config.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.13.14",
    "@babel/plugin-external-helpers": "^7.12.13",
    "@babel/preset-env": "^7.13.12",
    "rollup-plugin-babel": "^4.4.0"
  },
  "dependencies": {
    "@babel/polyfill": "^7.12.1",
    "rollup-plugin-node-resolve": "^5.2.0"
  }
}
```

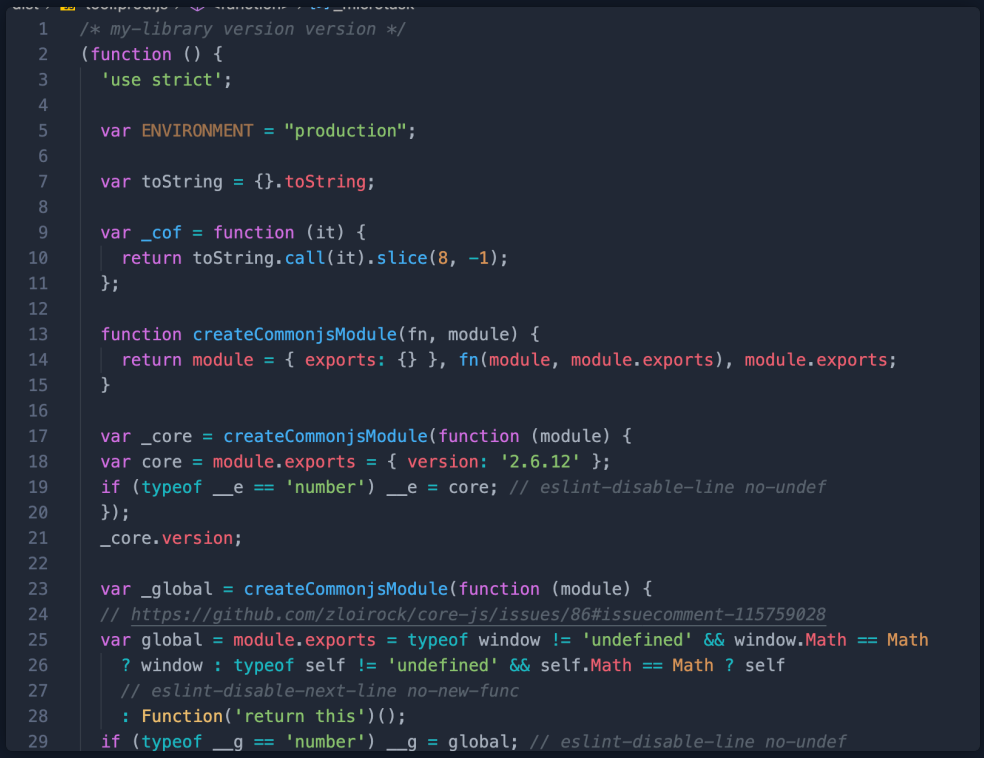
在配置文件中引入插件并调用

Rollup.config.js

```
import babel from 'rollup-plugin-babel'; //
import resolve from 'rollup-plugin-node-resolve';
import commonjs from 'rollup-plugin-commonjs';
export default {
    input: 'src/index.js',// 入口文件
    output: [
        { // 输输出文件
            file: 'dist/tool.prod.js', // 输出文件的位置和名字
            format: 'iife', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* hello 呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "production";', // 在主文件里面插入新的代码
        },
        { // 输输出文件
            file: 'dist/tool.dev.js', // 输出文件的位置和名字
            format: 'esm', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 老妹呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "devlop";', // 在主文件里面插入 新的代码
        },
        { // 输输出文件
            file: 'dist/tool.ssr.js', // 输出文件的位置和名字
            format: 'cjs', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 我两是一对啊 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "ssr";', // 在主文件里面插入 新的代码
        }
    ],
    plugins: [
        resolve({
            browser: true,
        }),
        commonjs(),
        babel({
            exclude: 'node_modules/**'  // 排除node_module下的所有文件
        })
    ]
}
```

这样在进行编译, 就可以在编译后的文件里面看到结果了( PS: 无需专门在index.js里面引入polyfill )

tool.prod.js



```
/* my-library version version */
(function () {
  'use strict';

  var ENVIRONMENT = "production";

  var toString = {}.toString;

  var _cof = function (it) {
    return toString.call(it).slice(8, -1);
  };

  function createCommonjsModule(fn, module) {
    return module = { exports: {} }, fn(module, module.exports), module.exports;
  }

  var _core = createCommonjsModule(function (module) {
  var core = module.exports = { version: '2.6.12' };
  if (typeof __e == 'number') __e = core; // eslint-disable-line no-undef
  });
  _core.version;

  var _global = createCommonjsModule(function (module) {
  // https://github.com/zloirock/core-js/issues/86#issuecomment-115759028
  var global = module.exports = typeof window != 'undefined' && window.Math == Math
    ? window : typeof self != 'undefined' && self.Math == Math ? self
    // eslint-disable-next-line no-new-func
    : Function('return this')();
  if (typeof __g == 'number') __g = global; // eslint-disable-line no-undef
```

## H2 rollup外链

在有些时候 ,我们创造的轮子需要依赖于某些已经存在的插件或是工具, 但是我们又不想直接全部打包到一个文件里面, 那么这个时候, 我们就可以引入外链的方式来引入这个文件。需要使用external属性

Rollup.config.js

```
import babel from 'rollup-plugin-babel'; //
import resolve from 'rollup-plugin-node-resolve';
import commonjs from 'rollup-plugin-commonjs';
export default {
    input: 'src/index.js',// 入口文件
    output: [
        { // 输输出文件
            file: 'dist/tool.prod.js', // 输出文件的位置和名字
            format: 'iife', // 模块系统的格式, esm( es6的模块系统)格式或是 cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* hello 呀 */', // 增加注释信息在打包后的文件的头部
```

```
            intro: 'var ENVIRONMENT = "production";', // 在主文件里面插入新的代码
        },
        { // 输输出文件
            file: 'dist/tool.dev.js', // 输出文件的位置和名字
            format: 'esm', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 老妹呀 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "devlop";', // 在主文件里面插入 新的代码
        },
        { // 输输出文件
            file: 'dist/tool.ssr.js', // 输出文件的位置和名字
            format: 'cjs', // 模块系统的格式, esm( es6的模块系统)格式或是cjs (common js), amd(), umd 通用模块, iife,
            banner: '/* my-library version ' + 'version' + ' */', // 增加注释信息在打包后的文件的头部
            footer: '/* 我两是一对啊 */', // 增加注释信息在打包后的文件的头部
            intro: 'var ENVIRONMENT = "ssr";', // 在主文件里面插入 新的代码
        }
    ],
    plugins: [
        resolve({
            browser: true
        }),
        commonjs(),
        babel({
            exclude: 'node_modules/**'  // 排除node_module下的所有文件
        })
    ],
    global: {
        jquery: '$'
    },
    external: [
        'jquery'
    ]
}
```

index.js

```
import $ from "jquery";
import { add } from "./utils/math";
let p = new Promise(()=>{
    console.log('吃了吗?');
})

function test(...args){
    console.log(args);
    return add(args);
}

console.log(test(1,2,3,4,5));
console.log($);
```

打包后的代码文件

```
/* my-library version version */
(function ($) {
    'use strict';

    var ENVIRONMENT = "production";

    function _interopDefaultLegacy (e) { return e && typeof e === 'object' && 'default' in e ? e : { 'default': e }; }

    var $__default = /*#__PURE__*/_interopDefaultLegacy($);

    function add() {
      for (var _len = arguments.length, args = new Array(_len), _key = 0; _key < _len; _key++) {
        args[_key] = arguments[_key];
      }

      return args.reduce(function (pre, cur) {
        return pre + cur;
      });
    }

    new Promise(function () {
      console.log('吃了吗?');
    });

    function test() {
      for (var _len = arguments.length, args = new Array(_len), _key = 0; _key < _len; _key++) {
        args[_key] = arguments[_key];
```

```
      }

      console.log(args);
      return add(args);
    }

    console.log(test(1, 2, 3, 4, 5));
    console.log($__default['default']);

}($));
/* hello 呀 */
```