

# 实现小型打包工具

原本小册计划中是没有这一章节的，Webpack 工作原理应该是上一章节包含的内容。但是考虑到既然讲到工作原理，必然需要讲解源码，但是 Webpack 的源码很难读，不结合源码干巴巴讲原理又没有什么价值。所以在这一章节中，我将会带大家来实现一个几十行的迷你打包工具，该工具可以实现以下两个功能

- 将 ES6 转换为 ES5
- 支持在 JS 文件中 import CSS 文件

通过这个工具的实现，大家可以理解到打包工具的**原理**到底是什么。

## 实现

因为涉及到 ES6 转 ES5，所以我们首先需要安装一些 Babel 相关的工具

```
yarn add babylon babel-traverse babel-core babel-preset-env
```

接下来我们将这些工具引入文件中

```
const fs = require('fs')
const path = require('path')
const babylon = require('babylon')
const traverse = require('babel-traverse').default
const { transformFromAst } = require('babel-core')
```

首先，我们先来实现如何使用 Babel 转换代码

```

function readCode(filePath) {
  // 读取文件内容
  const content = fs.readFileSync(filePath, 'utf-8')
  // 生成 AST
  const ast = babylon.parse(content, {
    sourceType: 'module'
  })
  // 寻找当前文件的依赖关系
  const dependencies = []
  traverse(ast, {
    ImportDeclaration: ({ node }) => {
      dependencies.push(node.source.value)
    }
  })
  // 通过 AST 将代码转为 ES5
  const { code } = transformFromAst(ast, null, {
    presets: ['env']
  })
  return {
    filePath,
    dependencies,
    code
  }
}

```

- 首先我们传入一个文件路径参数，然后通过 fs 将文件中的内容读取出来
- 接下来我们通过 babylon 解析代码获取 AST，目的是为了分析代码中是否还引入了别的文件
- 通过 dependencies 来存储文件中的依赖，然后再将 AST 转换为 ES5 代码
- 最后函数返回了一个对象，对象中包含了当前文件路径、当前

## 文件依赖和当前文件转换后的代码

接下来我们需要实现一个函数，这个函数的功能有以下几点

- 调用 readCode 函数，传入入口文件
- 分析入口文件的依赖
- 识别 JS 和 CSS 文件

```
function getDependencies(entry) {
  // 读取入口文件
  const entryObject = readCode(entry)
  const dependencies = [entryObject]
  // 遍历所有文件依赖关系
  for (const asset of dependencies) {
    // 获得文件目录
    const dirname = path.dirname(asset.filePath)
    // 遍历当前文件依赖关系
    asset.dependencies.forEach(relativePath => {
      // 获得绝对路径
      const absolutePath = path.join(dirname,
relativePath)
      // CSS 文件逻辑就是将代码插入到 `style` 标签中
      if (/\.css$/.test(absolutePath)) {
        const content =
fs.readFileSync(absolutePath, 'utf-8')
        const code = `
          const style =
document.createElement('style')
          style.innerText =
${JSON.stringify(content).replace(/\r\n/g, '')}
          document.head.appendChild(style)
        `
        dependencies.push({
```

```

        filePath: absolutePath,
        relativePath,
        dependencies: [],
        code
    })
} else {
    // JS 代码需要继续查找是否有依赖关系
    const child = readCode(absolutePath)
    child.relativePath = relativePath
    dependencies.push(child)
}
})
}
return dependencies
}

```

- 首先我们读取入口文件，然后创建一个数组，该数组的目的是存储代码中涉及到的所有文件
- 接下来我们遍历这个数组，一开始这个数组中只有入口文件，在遍历的过程中，如果入口文件有依赖其他的文件，那么就会被 push 到这个数组中
- 在遍历的过程中，我们先获得该文件对应的目录，然后遍历当前文件的依赖关系
- 在遍历当前文件依赖关系的过程中，首先生成依赖文件的绝对路径，然后判断当前文件是 CSS 文件还是 JS 文件
  - 如果是 CSS 文件的话，我们就不能用 Babel 去编译了，只需要读取 CSS 文件中的代码，然后创建一个 style 标签，将代码插入进标签并且放入 head 中即可
  - 如果是 JS 文件的话，我们还需要分析 JS 文件是否还有别的依赖关系
  - 最后将读取文件后的对象 push 进数组中

现在我们已经获取到了所有的依赖文件，接下来就是实现打包的功能了

```
function bundle(dependencies, entry) {
  let modules = ''
  // 构造函数参数，生成的结构为
  // { './entry.js': function(module, exports,
  // require) { 代码 } }
  dependencies.forEach(dep => {
    const filePath = dep.relativePath || entry
    modules += `'{filePath}': (
      function (module, exports, require) {
        ${dep.code}
      },
    `
  })
  // 构建 require 函数，目的是为了获取模块暴露出来的内容
  const result = `
    (function(modules) {
      function require(id) {
        const module = { exports : {} }
        modules[id](module, module.exports,
        require)
        return module.exports
      }
      require('${entry}')
    })([${modules}])
  `

  // 当生成的内容写入到文件中
  fs.writeFileSync('./bundle.js', result)
}
```

这段代码需要结合着 Babel 转换后的代码来看，这样大家就能理解为什么需要这样写了

```
// entry.js
var _a = require('./a.js')
var _a2 = _interopRequireDefault(_a)
function _interopRequireDefault(obj) {
  return obj && obj.__esModule ? obj : {
default: obj }
}
console.log(_a2.default)
// a.js
Object.defineProperty(exports, '__esModule', {
  value: true
})
var a = 1
exports.default = a
```

Babel 将我们 ES6 的模块化代码转换为了 CommonJS（如果你不熟悉 CommonJS 的话，可以阅读这一章节中关于 [模块化的知识点](https://juejin.im/book/5bdc715fe51d454e755f75ef/section) (<https://juejin.im/book/5bdc715fe51d454e755f75ef/section>) 的代码，但是浏览器是不支持 CommonJS 的，所以如果这段代码需要在浏览器环境下运行的话，我们需要自己实现 CommonJS 相关的代码，这就是 bundle 函数做的大部分事情。

接下来我们再来逐行解析 bundle 函数

- 首先遍历所有依赖文件，构建出一个函数参数对象
- 对象的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数 module、exports、require
  - module 参数对应 CommonJS 中的 module
  - exports 参数对应 CommonJS 中的 module.export
  - require 参数对应我们自己创建的 require 函数
- 接下来就是构造一个使用参数的函数了，函数做的事情很简

- 单，就是内部创建一个 `require` 函数，然后调用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.export` 的方式让外部获取到
- 最后再将打包出来的内容写入到单独的文件中

如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码

```

;(function(modules) {
  function require(id) {
    // 构造一个 CommonJS 导出代码
    const module = { exports: {} }
    // 去参数中获取文件对应的函数并执行
    modules[id](module, module.exports, require)
    return module.exports
  }
  require('./entry.js')
})(
  './entry.js': function(module, exports,
require) {
    // 这里继续通过构造的 require 去找到 a.js 文件对应的函数
    var _a = require('./a.js')
    console.log(_a2.default)
  },
  './a.js': function(module, exports, require) {
    var a = 1
    // 将 require 函数中的变量 module 变成了这样的结构
    // module.exports = 1
    // 这样就能在外部取到导出的内容了
    exports.default = a
  }
  // 省略
})

```

## 小结

虽然实现这个工具只写了不到 100 行的代码，但是打包工具的核心原理就是这些了



1. 找出入口文件所有的依赖关系
2. 然后通过构建 CommonJS 代码来获取 exports 导出的内容

如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。