link: null
title: 珠峰架构师成长计划
description: rollup.config.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=272 sentences=1214, words=6727

1. rollup实战

- rollupjs (https://rollupjs.org/guide/en/)是下一代ES模块捆绑器**1.1 背景#**
- webpack打包非常繁琐，打包体积比较大
- rollup主要是用来打包JS库的
- vue/react/angular都在用rollup作为打包工具

** 1.2 安装依赖 #**

```
cnpm i @babel/core @babel/preset-env  @rollup/plugin-commonjs @rollup/plugin-node-resolve @rollup/plugin-typescript lodash rollup rollup-plugin-babel postcss
rollup-plugin-postcss rollup-plugin-terser tslib typescript rollup-plugin-serve rollup-plugin-livereload -D
```

** 1.3 初次使用 #**

1.3.1 rollup.config.js

- Asynchronous Module Definition异步模块定义
- ES6 module是es6提出了新的模块化方案
- IIFE(Immediately Invoked Function Expression)即立即执行函数表达式，所谓立即执行，就是声明一个函数，声明完了立即执行
- UMD全称为 Universal Module Definition,也就是通用模块定义
- cjs是nodejs采用的模块化标准，commonjs使用方法 require来引入模块,这里 require()接收的参数是模块名或者是模块文件的路径

rollup.config.js

```
export default {
    input:'src/main.js',
    output:{
        file:'dist/bundle.cjs.js',
        format:'cjs',
        name:'bundleName'
    }
}
```

1.3.2 src\main.js

src\main.js

```
console.log('hello');
```

1.3.3 package.json

package.json

```
{
 "scripts": {
    "build": "rollup --config"
  },
}
```

1.3.4 dist\index.html

dist\index.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>rolluptitle>
head>
<body>
    <script src="bundle.cjs.js">script>
body>
html>
```

** 1.4 支持babel #**

- 为了使用新的语法，可以使用babel来进行编译输出

1.4.1 安装依赖

- @babel/core是babel的核心包
- @babel/preset-env是预设
- @rollup/plugin-babel是babel插件

```
cnpm install @rollup/plugin-babel @babel/core @babel/preset-env --save-dev
```

1.4.2 src\main.js

```
let sum = (a,b)=>{
    return a+b;
}
let result = sum(1,2);
console.log(result);
```

1.4.3 .babelrc

.babelrc

```
{
    "presets": [
       [
        "@babel/env",
        {
            "modules":false
        }
       ]
    ]
}
```

1.4.4 rollup.config.js

rollup.config.js

```
+import babel from '@rollup/plugin-babel';
export default {
    input:'src/main.js',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'cjs',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName'//当format为iife和umd时必须提供，将作为全局变量挂在window下
    },
+   plugins:[
+       babel({
+           exclude:"node_modules/**"
+       })
+   ]
}
```

** 1.5 tree-shaking #**

- Tree-shaking的本质是消除无用的js代码
- rollup只处理函数和顶层的import/export变量

1.5.1 src\main.js

src\main.js

```
import {name,age} from './msg';
console.log(name);
```

1.5.2 src\msg.js

src\msg.js

```
export var name = 'zhufeng';
export var age = 12;
```

** 1.6 使用第三方模块 #**

- rollup.js编译源码中的模块引用默认只支持 ES6+的模块方式 `import/export`

1.6.1 安装依赖

```
cnpm install @rollup/plugin-node-resolve @rollup/plugin-commonjs lodash  --save-dev
```

1.6.2 src\main.js

src\main.js

```
import _ from 'lodash';
console.log(_);
```

1.6.3 rollup.config.js

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/main.js',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'cjs',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName'//当format为iife和umd时必须提供，将作为全局变量挂在window下
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
+       resolve(),
+       commonjs()
    ]
}
```

** 1.7 使用CDN #**

1.7.1 src\main.js

src\main.js

```
import _ from 'lodash';
import $ from 'jquery';
console.log(_.concat([1,2,3],4,5));
console.log($);
export default 'main';
```

1.7.2 dist\index.html

dist\index.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>rolluptitle>
head>
<body>
    <script src="https://cdn.jsdelivr.net/npm/lodash/lodash.min.js">script>
    <script src="https://cdn.jsdelivr.net/npm/jquery/jquery.min.js">
```

1.7.3 rollup.config.js

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/main.js',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
+       format:'iife',//五种输出格式：amd/es6/iife/umd/cjs
+       name:'bundleName',//当format为iife和umd时必须提供，将作为全局变量挂在window下
+       globals:{
+           lodash:'_', //告诉rollup全局变量_即是lodash
+           jquery:'{{content}}#x27; //告诉rollup全局变量$即是jquery
+       }
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
        resolve(),
        commonjs()
    ],
+   external:['lodash','jquery']
}
```

** 1.8 使用typescript #**

1.8.1 安装

```
cnpm install tslib typescript @rollup/plugin-typescript --save-dev
```

1.8.2 src\main.ts

src\main.ts

```
let myName:string = 'zhufeng';
let age:number=12;
console.log(myName,age);
```

1.8.3 rollup.config.js

rollup.config.js

```
import babel from 'rollup-plugin-babel';
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
+import typescript from '@rollup/plugin-typescript';
export default {
+   input:'src/main.ts',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'iife',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName',//当format为iife和umd时必须提供，将作为全局变量挂在window下
        globals:{
            lodash:'_', //告诉rollup全局变量_即是lodash
            jquery:'{{content}}#x27; //告诉rollup全局变量$即是jquery
        }
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
        resolve(),
        commonjs(),
+       typescript()
    ],
    external:['lodash','jquery']
}
```

1.8.4 tsconfig.json

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "ESNext",
    "strict": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

** 1.9 压缩JS #**

- terser是支持ES6 +的JavaScript压缩器工具包

1.9.1 安装

```
cnpm install rollup-plugin-terser --save-dev
```

1.9.2 rollup.config.js

rollup.config.js

```
import babel from 'rollup-plugin-babel';
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
import typescript from '@rollup/plugin-typescript';
+import {terser} from 'rollup-plugin-terser';
export default {
    input:'src/main.ts',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'iife',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName',//当format为iife和umd时必须提供，将作为全局变量挂在window下
        globals:{
            lodash:'_', //告诉rollup全局变量_即是lodash
            jquery:'{{content}}#x27; //告诉rollup全局变量$即是jquery
        }
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
        resolve(),
        commonjs(),
        typescript(),
+       terser(),
    ],
    external:['lodash','jquery']
}
```

** 1.10 编译css #**

- terser是支持ES6 +的JavaScript压缩器工具包

1.10.1 安装

```
cnpm install  postcss rollup-plugin-postcss --save-dev
```

1.10.2 rollup.config.js

rollup.config.js

```
import babel from 'rollup-plugin-babel';
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
import typescript from '@rollup/plugin-typescript';
import {terser} from 'rollup-plugin-terser';
+import postcss from 'rollup-plugin-postcss';
export default {
    input:'src/main.js',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'iife',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName',//当format为iife和umd时必须提供，将作为全局变量挂在window下
        globals:{
            lodash:'_', //告诉rollup全局变量_即是lodash
            jquery:'{{content}}#x27; //告诉rollup全局变量$即是jquery
        }
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
        resolve(),
        commonjs(),
        typescript(),
        //terser(),
+       postcss()
    ],
    external:['lodash','jquery']
}
```

1.10.3 src\main.js

src\main.js

```
import './main.css';
```

1.10.4 src\main.css

src\main.css

```
body{
    background-color: green;
}
```

** 1.11 本地服务器 #**

1.11.1 安装

```
cnpm install rollup-plugin-serve --save-dev
```

1.11.2 rollup.config.dev.js

rollup.config.dev.js

```
import babel from 'rollup-plugin-babel';
import resolve from '@rollup/plugin-node-resolve';
import commonjs from '@rollup/plugin-commonjs';
import typescript from '@rollup/plugin-typescript';
import postcss from 'rollup-plugin-postcss';
+import serve from 'rollup-plugin-serve';
export default {
    input:'src/main.js',
    output:{
        file:'dist/bundle.cjs.js',//输出文件的路径和名称
        format:'iife',//五种输出格式：amd/es6/iife/umd/cjs
        name:'bundleName',//当format为iife和umd时必须提供，将作为全局变量挂在window下
        sourcemap:true,
        globals:{
            lodash:'_', //告诉rollup全局变量_即是lodash
            jquery:'{{content}}#x27; //告诉rollup全局变量$即是jquery
        }
    },
    plugins:[
        babel({
            exclude:"node_modules/**"
        }),
        resolve(),
        commonjs(),
        typescript(),
        postcss(),
+       serve({
+           open:true,
+           port:8080,
+           contentBase:'./dist'
+       })
    ],
    external:['lodash','jquery']
}
```

1.11.3 package.json

package.json

```
{
  "scripts": {
    "build": "rollup --config rollup.config.build.js",
    "dev": "rollup --config rollup.config.dev.js -w"
  },
}
```

2.前置知识

** 2.1. 初始化项目 #**

```
cnpm install magic-string acorn --save
```

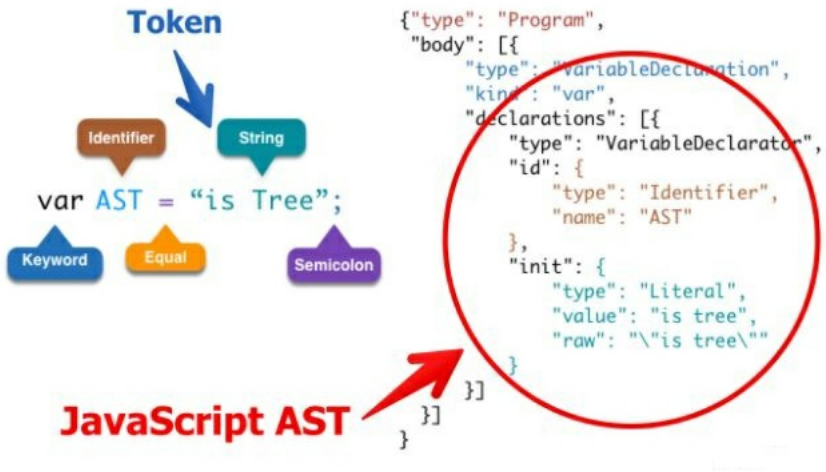
** 2.2. magic-string #**

- magic-string (https://www.npmjs.com/package/magic-string)是一个操作字符串和生成source-map的工具

```
var MagicString = require('magic-string');
let sourceCode = `export var name = "zhufeng"`;
let ms = new MagicString(sourceCode);
console.log(ms);

console.log(ms.snip(0, 6).toString());

console.log(ms.remove(0, 7).toString());

let bundle = new MagicString.Bundle();
bundle.addSource({
  content: 'var a = 1;',
  separator: '\n'
});
bundle.addSource({
  content: 'var b = 2;',
  separator: '\n'
});
console.log(bundle.toString());
```

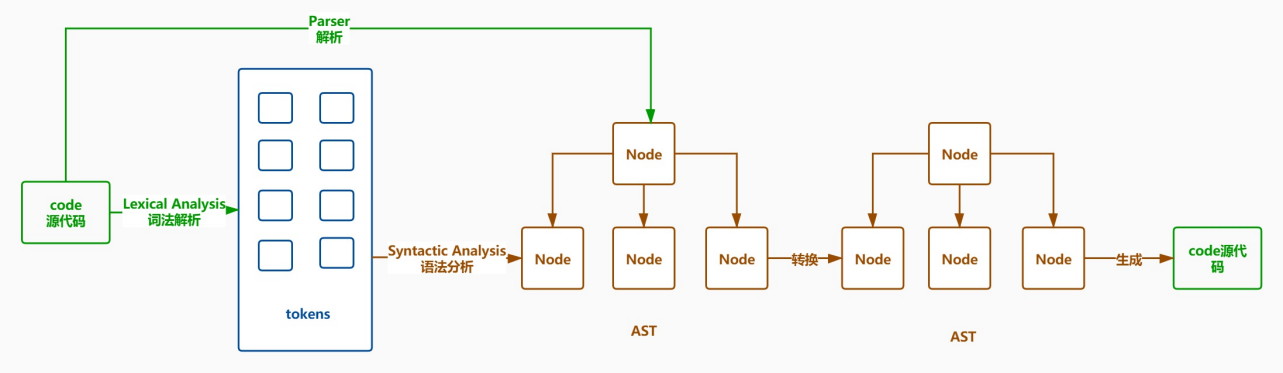
** 2.3. AST #**

- 通过 JavaScript Parser可以把代码转化为一颗抽象语法树AST,这颗树定义了代码的结构，通过操纵这颗树，我们可以精准的定位到声明语句、赋值语句、运算语句等等，实现对代码的分析、优化、变更等操作



2.3.1 AST工作流

- Parse(解析) 将源代码转换成抽象语法树，树上有很多的estree节点
- Transform(转换) 对抽象语法树进行转换
- Generate(代码生成) 将上一步经过转换过的抽象语法树生成新的代码



2.3.2 acorn

- astexplorer (https://astexplorer.net/)可以把代码转成语法树
- acorn 解析结果符合 The Estree Spec规范

2.3.2.1 walk.js

```
function walk(astNode, { enter, leave }) {
  visit(astNode, null, enter, leave);
}
function visit(node, parent, enter, leave) {
  if (enter) {
    enter.call(null, node, parent);
  }
  let keys = Object.keys(node).filter(key => typeof node[key] === 'object')
  keys.forEach(key => {
    let value = node[key];
    if (Array.isArray(value)) {
      value.forEach(val => visit(val, node, enter, leave));
    } else if (value && value.type) {
      visit(value, node, enter, leave)
    }
  });
  if (leave) {
    leave.call(null, node, parent);
  }
}

module.exports = walk;
```

2.3.2.2 use.js

```
const acorn = require('acorn');
const walk = require('./walk');
const sourceCode = 'import $ from "jquery"'
const ast = acorn.parse(sourceCode, {
  locations: true, ranges: true, sourceType: 'module', ecmaVersion: 8
});
let indent = 0;
const padding = () => ' '.repeat(indent)
ast.body.forEach((statement) => {
  walk(statement, {
    enter(node) {
      if (node.type) {
        console.log(padding() + node.type + "进入");
        indent += 2;
      }
    },
    leave(node) {
      if (node.type) {
        indent -= 2;
        console.log(padding() + node.type + "离开");
      }
    }
  });
});
```

```
ImportDeclaration进入
  ImportDefaultSpecifier进入
    Identifier进入
    Identifier离开
  ImportDefaultSpecifier离开
  Literal进入
  Literal离开
ImportDeclaration离开
```

** 2.4 作用域 #**

2.4.1 作用域

- 在JS中，作用域是用来规定变量访问范围的规则

```
function one() {
  var a = 1;
}
console.log(a);
```

2.4.2 作用域链

- 作用域链是由当前执行环境与上层执行环境的一系列变量对象组成的，它保证了当前执行环境对符合访问权限的变量和函数的有序访问

2.4.2.1 scope.js

scope.js

```

class Scope {
  constructor(options = {}) {

    this.name = options.name;

    this.parent = options.parent;

    this.names = options.names || [];
  }
  add(name) {
    this.names.push(name);
  }
  findDefiningScope(name) {
    if (this.names.includes(name)) {
      return this;
    } else if (this.parent) {
      return this.parent.findDefiningScope(name);
    } else {
      return null;
    }
  }
}
module.exports = Scope;
```

2.4.2.2 useScope.js

useScope.js

```
var a = 1;
function one() {
  var b = 1;
  function two() {
    var c = 2;
     console.log(a, b, c);
  }
}
let Scope = require('./scope');
let globalScope = new Scope({ name: 'global', names: [], parent: null });
let oneScope = new Scope({ name: 'one', names: ['b'], parent: globalScope });
let twoScope = new Scope({ name: 'two', names: ['c'], parent: oneScope });
console.log(
  threeScope.findDefiningScope('a').name,
  threeScope.findDefiningScope('b').name,
  threeScope.findDefiningScope('c').name
```

3. 实现rollup

** 3.1 目录结构 #**

- rollup代码仓库地址 (https://gitee.com/zhufengpeixun/rollup)

```
.
├── package.json
├── README.md
├── src
    ├── ast
    │   ├── analyse.js
    │   ├── Scope.js
    │   └── walk.js
    ├── Bundle
    │   └── index.js
    ├── Module
    │   └── index.js
    ├── rollup.js
    └── utils
        ├── map-helpers.js
        ├── object.js
        └── promise.js
```

** 3.2 src\main.js#**

src\main.js

```
console.log('hello');
```

** 3.3 debugger.js#**

```
const path = require('path');
const rollup = require('./lib/rollup');
let entry = path.resolve(__dirname, 'src/main.js');
rollup(entry,'bundle.js');
```

** 3.4 rollup.js#**

lib\rollup.js

```
const Bundle = require('./bundle')
const fs = require('fs')

function rollup(entry, filename) {
    const bundle = new Bundle({ entry });
    bundle.build(filename);
}
module.exports = rollup;
```

** 3.5 bundle.js#**

lib\bundle.js

```
let fs = require('fs');
let path = require('path');
let Module = require('./module');
let MagicString = require('magic-string');
class Bundle {
    constructor(options) {

        this.entryPath = path.resolve(options.entry.replace(/\.js$/, '') + '.js');
        this.modules = {};
    }
    build(filename) {
        let entryModule = this.fetchModule(this.entryPath);
        this.statements = entryModule.expandAllStatements(true);
        const { code } = this.generate({});
        fs.writeFileSync(filename, code);
    }
    fetchModule(importee) {
        let route = importee;
        if (route) {
            let code = fs.readFileSync(route, 'utf8');
            const module = new Module({
                code,
                path: importee,
                bundle: this
            })
            return module;
        }
    }
    generate(options) {
        let magicString = new MagicString.Bundle();
        this.statements.forEach(statement => {
            const source = statement._source.clone();
            magicString.addSource({
                content: source,
                separator: '\n'
            })

        })
        return { code: magicString.toString() }
    }
}
module.exports = Bundle;
```

** 3.6 module.js#**

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
let analyse = require('./ast/analyse');
class Module {
    constructor({ code, path, bundle }) {
        this.code = new MagicString(code, { filename: path });
        this.path = path;
        this.bundle = bundle;
        this.ast = parse(code, {
            ecmaVersion: 7,
            sourceType: 'module'
        })
        this.analyse();
    }
    analyse() {
        analyse(this.ast, this.code, this);
    }
    expandAllStatements() {
        let allStatements = [];
        this.ast.body.forEach(statement => {
            let statements = this.expandStatement(statement);
            allStatements.push(...statements);
        });
        return allStatements;
    }
    expandStatement(statement) {
        statement._included = true;
        let result = [];
        result.push(statement);
        return result;
    }
}
module.exports = Module;
```

** 3.7 analyse.js #**

lib\ast\analyse.js

```
function analyse(ast, magicString) {
    ast.body.forEach(statement => {
        Object.defineProperties(statement, {
            _source: { value: magicString.snip(statement.start, statement.end) }
        })
    });
}
module.exports = analyse;
```

4. 实现tree-shaking

```

this.imports[localName] = { localName, importName: 'default', source };

this.exports[exportName] = { node, localName, exportName, type: 'ExportSpecifier' };

statement._defines[name] = true;

this.definitions[name] = statement;

statement._dependsOn[node.name] = true;
```

** 4.1 msg.js #**

src\msg.js

```
export var name = 'zhufeng';
export var age = 12;
```

** 4.2 src\main.js #**

src\main.js

```
import {name,age} from './msg';
function say(){
    console.log('hello',name);
}
say();
```

** 4.3 bundle.js #**

lib\bundle.js

```
const path = require('path');
const fs = require('fs');
const MagicString = require('magic-string');
const Module = require('./module');
class Bundle {
  constructor(options) {
    //入口文件的绝对路径
    this.entryPath = path.resolve(options.entry);
  }
  build(filename) {
    let entryModule = this.fetchModule(this.entryPath);
    this.statements = entryModule.expandAllStatements();//this.ast.body
    const { code } = this.generate();
    fs.writeFileSync(filename, code);
  }
  //根据模块的路径，返回模块对象
  fetchModule(importee, importer) {//importee=./msg.js
+   let route;
+   if (!importer) {//如果无人导入，说明是入口模块
+     route = importee;
+   } else {
+     if (path.isAbsolute(importee)) {
+       route = importee;
+     } else {
+       route = path.resolve(path.dirname(importer), importee.replace(/\.js$/, '') + '.js')
+     }
+   }
    if (route) {
      let code = fs.readFileSync(route, 'utf8');
      let module = new Module({
        path: route,//模块的绝对路径
        code,//模块内容
        bundle: this//它所属的bundle的实例
      })
      return module;
    }
  }
  generate() {
    let bundle = new MagicString.Bundle();
    this.statements.forEach(statement => {
      const source = statement._source.clone();
+     if (statement.type === 'ExportNamedDeclaration') {
+       source.remove(statement.start, statement.declaration.start);
+     }
      bundle.addSource({
        content: source,
        separator: '\n'
      });
    });
    return { code: bundle.toString() };
  }
}
module.exports = Bundle;
```

** 4.4 scope.js #**

lib\ast\scope.js

```
class Scope {
  constructor(options = {}) {

    this.name = options.name;

    this.parent = options.parent;

    this.names = options.names || [];
  }
  add(name) {
    this.names.push(name);
  }
  findDefiningScope(name) {
    if (this.names.includes(name)) {
      return this;
    } else if (this.parent) {
      return this.parent.findDefiningScope(name);
    } else {
      return null;
    }
  }
}
module.exports = Scope;
```

** 4.5 utils.js #**

lib\utils.js

```
function hasOwnProperty(obj, prop) {
  return Object.prototype.hasOwnProperty.call(obj, prop)
}
exports.hasOwnProperty = hasOwnProperty;
```

** 4.5 module.js #**

lib\module.js

```
const MagicString = require('magic-string');
const acorn = require('acorn');
const analyse = require('./ast/analyse');
const { hasOwnProperty } = require('./utils');
class Module {
  constructor({ path, code, bundle }) {
    this.code = new MagicString(code, {
      filename: path
    })
    this.path = path;
    this.bundle = bundle;
    //把源代码转成一个语法树
    this.ast = acorn.parse(code, {
      ecmaVersion: 8,
      sourceType: 'module'
    });
+   //存放本模块的导入信息
+   this.imports = {};
+   //本模块的导出信息
+   this.exports = {};
+   //存放本模块的定义变量的语句 a=>var a = 1;b =var b =2;
+   this.definitions = {};
    //开始进行语法树的解析
    this.analyse();
  }
  analyse() {
+   this.ast.body.forEach(statement => {
+     //给this.imports赋值 分析 导入
+     if (statement.type === 'ImportDeclaration') {//main.js
+       let source = statement.source.value;//"./msg"
+       statement.specifiers.forEach(specifier => {
+         let importName = specifier.imported.name;//name 外部msg模块导出的名称 name
+         let localName = specifier.local.name;//n 导入到本模块后，本地变量的名字 n
+         this.imports[localName] = { importName, source };
+       });
+       //给this.exports赋值，分析导出
+     } else if (statement.type === 'ExportNamedDeclaration') {//msg.js
+       let declaration = statement.declaration;
+       if (declaration.type === 'VariableDeclaration') {
+         const declarations = declaration.declarations;
+         declarations.forEach((variableDeclarator) => {
+           let localName = variableDeclarator.id.name;//name
+           this.exports[localName] = { exportName: localName }
+         });
+       }
+     }
+   });
    analyse(this.ast, this.code, this);
  }
  expandAllStatements() {
    let allStatements = [];
    this.ast.body.forEach(statement => {
+     if (statement.type === 'ImportDeclaration') return;
      let statements = this.expandStatement(statement);
      allStatements.push(...statements);
    });
    return allStatements;
  }
  expandStatement(statement) {
    //为了避免语句被 重复添加，所以给每个语句放置一个变量，表示是否已经添加到结果 里了
    statement._included = true;
    let result = [];
    //获取此语句依赖的变量
    let dependencies = Object.keys(statement._dependsOn);
    dependencies.forEach(name => {
      //找到此变量定义的语句，添加到输出数组里
      let definitions = this.define(name);
      result.push(...definitions);
    });
    result.push(statement);
    return result;
  }
+ define(name) {//name
+   //说明是导入的
+   if (hasOwnProperty(this.imports, name)) {
+     //this.imports[localName]= {localName,importName,source};
+     const { importName, source } = this.imports[name];
+     let importModule = this.bundle.fetchModule(source, this.path);//msg.js
+     // this.exports[localName] = {localName, exportName: localName, declaration}
+     let { exportName } = importModule.exports[importName];//name
+     return importModule.define(exportName);//msgModule.define(name);
+   } else {//模块内声明的
+     let statement = this.definitions[name];
+     if (statement && !statement._included) {
+       return this.expandStatement(statement);
+     } else {
+       return []
+     }
+   }
+ }
}
module.exports = Module;
```

** 4.6 walk.js#**

lib\ast\walk.js

```
function walk(astNode, { enter, leave }) {
  visit(astNode, null, enter, leave);
}
function visit(node, parent, enter, leave) {
  if (enter) {
    enter.call(null, node, parent);
  }
  let keys = Object.keys(node).filter(key => typeof node[key] === 'object')
  keys.forEach(key => {
    let value = node[key];
    if (Array.isArray(value)) {
      value.forEach(val => visit(val, node, enter, leave));
    } else if (value && value.type) {
      visit(value, node, enter, leave)
    }
  });
  if (leave) {
    leave.call(null, node, parent);
  }
}
module.exports = walk;
```

** 4.7 analyse.js #**

lib\ast\analyse.js

```
+const Scope = require('./scope');
+const walk = require('./walk');
+const { hasOwnProperty } = require('../utils');
function analyse(ast, code, module) {
+ //处理作用域 构建作用域链 其实就是模块内的顶级作用域
+ let currentScope = new Scope({ name: '全局作用域' });
+ ast.body.forEach(statement => {
+   function addToScope(name) {
+     currentScope.add(name);
+     //如果说当前的作用域没有父亲 了，说明它是顶级作用域，说明name是顶级作用域中定义的变量 类似模块内的全局变量
+     if (!currentScope.parent || currentScope.isBlockScope) {
+       //此语句上定义了哪些变量
+       statement._defines[name] = true
+     }
+   }
+   //给statement语法树节点，定义属性_source=console.log('hello');
+   Object.defineProperties(statement, {
+     _source: { value: code.snip(statement.start, statement.end) },
+     _defines: { value: {} },
+     _included: { value: false, writable: true },
+     _dependsOn: { value: {} }
+   });

+   walk(statement, {
+     enter(node) {
+       //当前节点的新的作用域
+       let newScope;
+       switch (node.type) {
+         case 'FunctionDeclaration':
+           //函数的名字添加到当前作用域中
+           addToScope(node.id.name);
+           //函数的参数添加到新作用域
+           const names = node.params.map(param => param.name);
+           newScope = new Scope({ name: node.id.name, parent: currentScope, names })
+           break;
+         case 'VariableDeclaration':
+           node.declarations.forEach(declaration => {
+             addToScope(declaration.id.name);
+           });
+           break;
+         default:
+           break;
+       }
+       if (newScope) {
+         //此节点上创建了新的作用域
+         Object.defineProperty(statement, '_scope', { value: newScope });
+         currentScope = newScope;
+       }
+     },
+     leave(node) {
+       if (hasOwnProperty(node, '_scope')) {
+         currentScope = currentScope.parent;
+       }
+     }
+   });
+ });
+ //第二次循环
+ ast.body.forEach(statement => {
+   walk(statement, {
+     enter(node) {
+       if (node.type === 'Identifier') {
+         statement._dependsOn[node.name] = true;
+       }
+     }
+   });
+ });
+ ast.body.forEach(statement => {
+   //获取每个语句定义的变量
+   Object.keys(statement._defines).forEach(name => {
+     //记录模块内定义的变量，key是变量名，值是定义变量的语句
+     module.definitions[name] = statement;
+   });
+ });
}
module.exports = analyse;
```

5. 包含修改语句

** 5.1 src\index.js #**

src\index.js

```
import {name,age} from './msg';
console.log(name);
```

** 5.2 msg.js #**

src\msg.js

```
export var name = 'zhufeng';
+name += 'jiagou';
export var age = 12;
```

** 5.3 lib\module.js #**

lib\module.js

```

const MagicString = require('magic-string');
const { parse } = require('acorn');
const analyse = require('./ast/analyse');
const { hasOwnProperty } = require('./utils');
class Module {
    constructor({ code, path, bundle }) {
        this.code = new MagicString(code, { filename: path });
        this.path = path;
        this.bundle = bundle;
        this.ast = parse(code, {
            ecmaVersion: 8,
            sourceType: 'module'
        });
        this.imports = {};//导入
        this.exports = {};//导出
        this.definitions = {};//此变量存放所有的变量定义的语句
+       this.modifications = {};
        this.analyse();
    }
    analyse() {
        this.ast.body.forEach(statement => {
            //1.给import赋值
            if (statement.type
                let source = statement.source.value;//./msg
                statement.specifiers.forEach(specifier => {
                    let importName = specifier.imported.name;//age
                    let localName = specifier.local.name;//age
                    //记录一下当前的这个引入的变量是从哪个模块的哪个变量导入进来的
                    this.imports[localName] = { localName, source, importName };
                });
                //2.this.exports赋值
            } else if (statement.type
                let declaration = statement.declaration;
                if (declaration.type
                    const declarations = declaration.declarations;
                    declarations.forEach(variableDeclarator => {
                        let localName = variableDeclarator.id.name;
                        //this.exports.age = {localName:'age',exportName:'age',expression:};
                        this.exports[localName] = { localName, exportName: localName, expression: declaration };
                    });
                }
            }
        });
        //1.构建了作用域 2.找到模块块依赖了哪些外部变量
        analyse(this.ast, this.code, this);
        this.ast.body.forEach(statement => {
            Object.keys(statement._defines).forEach(name => {
                //当前模块内 定义name这个变量的语句是statement
                //main.js  type   let type = 'dog';
                this.definitions[name] = statement;
            });
+           Object.keys(statement._modifies).forEach(name => {
+               if (!hasOwnProperty(this.modifications, name)) {
+                   this.modifications[name] = [];
+               }
+               this.modifications[name].push(statement);
+           })
+       });
    }
    expandAllStatements() {
        let allStatements = [];
        this.ast.body.forEach(statement => {
            if (statement.type
                return;
            }
            let statements = this.expandStatement(statement);
            //我们要把statement进行扩展，有可能一行变多行var name ='zhufeng', console.log('name'); TODO
            allStatements.push(...statements);
        });
        return allStatements;
    }
    //展开单 条语句  console.log(name,age);
    expandStatement(statement) {
        console.log(statement);
        statement._included = true;// 把当前语句标记为包含
        let result = [];
        const dependencies = Object.keys(statement._dependsOn);
        dependencies.forEach(name => {
            //获取依赖的变量对应的变量定义语句 name
            let definition = this.define(name);
            result.push(...definition);
        });
        result.push(statement);//再放进来当前的语句
+       const defines = Object.keys(statement._defines);
+       defines.forEach(name => {
```

```
+           const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
+           if (modifications) {
+               modifications.forEach(statement => {
+                   if (!statement._included) {
+                       let statements = this.expandStatement(statement);
+                       result.push(...statements);
+                   }
+               });
+           }
+       });
        return result;
    }
    define(name) {
        //说明此变量是外部导入进来的
        if (hasOwnProperty(this.imports, name)) {
            //this.imports[localName]={localName,source,importName};
            const { localName, source, importName } = this.imports[name];
            // source .msg
            let importedModule = this.bundle.fetchModule(source, this.path);
            const { localName: exportLocalName } = importedModule.exports[importName];
            //this.exports.name = {localName:'name',exportName:'name',expression:};
            return importedModule.define(exportLocalName);
        } else {//说明是在当前模块内声明的
            let statement = this.definitions[name];
            if (statement && !statement._included) {
                return this.expandStatement(statement);
            } else {
                return [];
            }
        }
    }
}
module.exports = Module;
```

** 5.4 analyse.js #**

lib\ast\analyse.js

```
let walk = require('./walk');
let Scope = require('./scope');
const {hasOwnProperty} = require('../utils');
function analyse(ast, magicStringOfAst) {
    //在遍历之前先创建作用域对象
    let scope = new Scope({ name: '全局作用域' });
    ast.body.forEach(statement => {
        function addToScope(name) {
            scope.add(name);
            if (!scope.parent || currentScope.isBlockScope) {//如果此作用域 没有父作用域,那这就是顶级变量，根作用域下的变量，可以放在_defines=tree
                //模块内的顶级变量
                statement._defines[name] = true;
            }
        }
        Object.defineProperties(statement, {
+           _modifies:{ value: {} },//修改
            _defines: { value: {} },//当前statement语法树节点声明了哪些变量
            _dependsOn: { value: {} },//当前statement语句外部依赖的变量
            _included: { value: false, writable: true },//当前的语句是否放置在结果中，是否会出现在打包结果中
            _source: { value: magicStringOfAst.snip(statement.start, statement.end) }//key是_source 值是这个语法树节点在源码中的源代码
        });
        //如何知道某个变量有没有在当前模块内定义的呢?
        //原理是这样 扫描整个模块，找到所有的定义的变量
        //构建使用域链
        walk(statement, {
            enter(node) {
                let newScope;
                switch (node.type) {
                    case 'FunctionDeclaration':
                        //函数的参数将会成为此函数子作用域内的局部变量
                        const names = node.params.map(param => param.name);
                        addToScope(node.id.name);//把node也就是say这个变量添加到当前作用内
                        //如果遇到函数声明，就会产生一个新作用域
                        newScope = new Scope({ name: node.id.name, parent: scope, params: names });
                        break;
                    case 'VariableDeclaration':
                        node.declarations.forEach((declaration) => addToScope(declaration.id.name));
                        break;
                }
                if (newScope) {//如果创建了新的作用域，那么这个作用域将会成为新的当前作用域
                    Object.defineProperty(node, '_scope', { value: newScope });
                    scope = newScope;// say这个函数作用域
                }
            },
            leave(node) {
                //当离开节点的时候 ，如果发现这个节点创建 新的作用域，就回退到使用域
                if (hasOwnProperty(node,'_scope')) {
                    scope = scope.parent;
                }
            }
        });

    });
    //2.作用域链构建完成后，再遍历一次，找出本模块定义了依赖了哪些外部变量
    ast.body.forEach(statement => {
+       function checkForReads(node) {
+           if (node.type === 'Identifier' && parent.type !== 'VariableDeclarator') {
+               const definingScope = scope.findDefiningScope(node.name);
+               if (!definingScope) {
+                   statement._dependsOn[node.name] = true;//添加外部依赖
+               }
+           }
+       }
+       function checkForWrites(node) {
+           function addNode(node) {
+               while (node.type === 'MemberExpression') {
+                   node = node.object;
+               }
+               if (node.type !== 'Identifier') {
+                   return
+               }
+               statement._modifies[node.name] = true;
+           }
+           if (node.type === 'AssignmentExpression') {// name='jiagou'
+               addNode(node.left, true)
+           } else if (node.type === 'UpdateExpression') {//name+='jiagou'
+               addNode(node.argument, true)
+           }
+       }
+       walk(statement, {
+           enter(node) {
+               if (hasOwnProperty(node,'_scope')) scope = node._scope;
+               checkForReads(node)
+               checkForWrites(node)
+           },
+           leave(node) {
+               if (node._scope) scope = scope.parent
+           }
+       })
    });

}
module.exports = analyse;
```

6. 支持块级作用域和入口treeshaking

** 6.1 src\main.js #**

src\main.js

```
if(true){
    var age = 12;
}
console.log(age);
```

```
var name = 'zhufeng';
var age = 12;
console.log(age);
```

** 6.2 scope.js #**

lib\ast\scope.js

```
class Scope {
    constructor(options = {}) {
        this.name = options.name;
        this.parent = options.parent;
        this.names = options.params || [];//存放着当前作用域内的所有的变量
+       this.isBlockScope = !!options.block// 是否块作用域
    }
+   add(name, isBlockDeclaration) {
+       if (!isBlockDeclaration && this.isBlockScope) {
            //这是一个var或者函数声明，并且这是一个块级作用域，所以我们需要向上提升
+           this.parent.add(name, isBlockDeclaration)
        } else {
            this.names.push(name)
        }
    }
    //给我一个变量，我查一下在哪个作用域中定义的这个变量
    findDefiningScope(name) {
        if (this.names.includes(name)) {//如果自己有，就返回自己这个作用
            return this;
        }
        if (this.parent) {//如果自己没有这个变量，但是有爹，问问爹有没有
            return this.parent.findDefiningScope(name)
        }
        return null;
    }
}
module.exports = Scope;
```

** 6.3 analyse.js #**

lib\ast\analyse.js

```
let walk = require('./walk');
let Scope = require('./scope');
const { hasOwnProperty } = require('../utils');
function analyse(ast, magicStringOfAst) {
    //在遍历之前先创建作用域对象
    let scope = new Scope({ name: '全局作用域' });
    ast.body.forEach(statement => {
+       function addToScope(name, isBlockDeclaration) {
+           scope.add(name, isBlockDeclaration);
            if (!scope.parent|| currentScope.isBlockScope) {//如果此作用域 没有父作用域,那这就是顶级变量，根作用域下的变量，可以放在_defines=tree
                //模块内的顶级变量
                statement._defines[name] = true;
            }
        }
        Object.defineProperties(statement, {
            _modifies: { value: {} },//修改
            _defines: { value: {} },//当前statement语法树节点声明了哪些变量
            _dependsOn: { value: {} },//当前statement语句外部依赖的变量
            _included: { value: false, writable: true },//当前的语句是否放置在结果中，是否会出现在打包结果中
            _source: { value: magicStringOfAst.snip(statement.start, statement.end) }//key是_source 值是这个语法树节点在源码中的源代码
        });
        //如何知道某个变量有没有在当前模块内定义的呢？
        //原理是这样 扫描整个模块，找到所有的定义的变量
        //构建使用域链
        walk(statement, {
            enter(node) {
                let newScope;
                switch (node.type) {
                    case 'FunctionDeclaration':
                        //函数的参数将会成为此函数子作用域内的局部变量
                        const names = node.params.map(param => param.name);
                        addToScope(node.id.name);//把node也就是say这个变量添加到当前作用内
                        //如果遇到函数声明，就会产生一个新作用域
                        newScope = new Scope({ name: node.id.name, parent: scope, params: names,
+                       block: false });
                        break;
+                   case 'BlockStatement':
+                       newScope = new Scope({
+                           parent: scope,
+                           block: true
+                       })
+                       break;
                    case 'VariableDeclaration':
                        node.declarations.forEach((declaration) => {
+                           if(node.kind === 'let'||node.kind === 'const'){
+                               addToScope(declaration.id.name,true);
+                           }else{
+                               addToScope(declaration.id.name)
                            }
                        });
                        break;
                }
                if (newScope) {//如果创建了新的作用域，那么这个作用域将会成为新的当前作用域
                    Object.defineProperty(node, '_scope', { value: newScope });
                    scope = newScope;// say这个函数作用域
                }
            },
            leave(node) {
```

```
                //当离开节点的时候 ，如果发现这个节点创建 新的作用域，就回退到使用域
                if (hasOwnProperty(node, '_scope')) {
                    scope = scope.parent;
                }
            }
        });

    });
    //2.作用域链构建完成后，再遍历一次，找出本模块定义了依赖了哪些外部变量
    ast.body.forEach(statement => {
        function checkForReads(node) {
            if (node.type
                const definingScope = scope.findDefiningScope(node.name);
-               if (!definingScope) {
                    statement._dependsOn[node.name] = true;//添加外部依赖
-               }
            }
        }
        function checkForWrites(node) {
            function addNode(node) {
                while (node.type
                    node = node.object;
                }
                if (node.type !== 'Identifier') {
                    return
                }
                statement._modifies[node.name] = true;
            }
            if (node.type
                addNode(node.left, true)
            } else if (node.type
                addNode(node.argument, true)
            }
        }
        walk(statement, {
            enter(node) {
                if (hasOwnProperty(node, '_scope')) scope = node._scope;
                checkForReads(node)
                checkForWrites(node)
            },
            leave(node) {
                if (node._scope) scope = scope.parent
            }
        })
    });

}
module.exports = analyse;
```

** 6.4 module.js#**

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
const analyse = require('./ast/analyse');
const { hasOwnProperty } = require('./utils');
+const SYSTEM_VARIABLE = ['console', 'log'];
class Module {
    constructor({ code, path, bundle }) {
        this.code = new MagicString(code, { filename: path });
        this.path = path;
        this.bundle = bundle;
        this.ast = parse(code, {
            ecmaVersion: 8,
            sourceType: 'module'
        });
        this.imports = {};//导入
        this.exports = {};//导出
        this.definitions = {};//此变量存放所有的变量定义的语句
        this.modifications = {};
        this.analyse();
    }
    analyse() {
        this.ast.body.forEach(statement => {
            //1.给import赋值
            if (statement.type
                let source = statement.source.value;//./msg
                statement.specifiers.forEach(specifier => {
                    let importName = specifier.imported.name;//age
                    let localName = specifier.local.name;//age
                    //记录一下当前的这个引入的变量是从哪个模块的哪个变量导入进来的
                    this.imports[localName] = { localName, source, importName };
                });
                //2.this.exports赋值
            } else if (statement.type
                let declaration = statement.declaration;
                if (declaration.type
                    const declarations = declaration.declarations;
                    declarations.forEach(variableDeclarator => {
                        let localName = variableDeclarator.id.name;
                        //this.exports.age = {localName:'age',exportName:'age',expression:};
                        this.exports[localName] = { localName, exportName: localName, expression: declaration };
                    });
                }
            }
        });
        //1.构建了作用域 2.找到模块块依赖了哪些外部变量
        analyse(this.ast, this.code, this);
        this.ast.body.forEach(statement => {
            Object.keys(statement._defines).forEach(name => {
                //当前模块内 定义name这个变量的语句是statement
                //main.js  type   let type = 'dog';
```

```
                this.definitions[name] = statement;
            });
            Object.keys(statement._modifies).forEach(name => {
                if (!hasOwnProperty(this.modifications, name)) {
                    this.modifications[name] = [];
                }
                this.modifications[name].push(statement);
            })
        });
    }
    expandAllStatements() {
        let allStatements = [];
        this.ast.body.forEach(statement => {
            if (statement.type
                return;
            }
+           if (statement.type === 'VariableDeclaration') return;
            let statements = this.expandStatement(statement);
            //我们要把statement进行扩展，有可能一行变多行var name ='zhufeng', console.log('name'); TODO
            allStatements.push(...statements);
        });
        return allStatements;
    }
    //展开单 条语句  console.log(name,age);
    expandStatement(statement) {
        console.log(statement);
        statement._included = true;// 把当前语句标记为包含
        let result = [];
        const dependencies = Object.keys(statement._dependsOn);
        dependencies.forEach(name => {
            //获取依赖的变量对应的变量定义语句 name
            let definition = this.define(name);
            result.push(...definition);
        });
        result.push(statement);//再放进来当前的语句
        const defines = Object.keys(statement._defines);
        defines.forEach(name => {
            const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
            if (modifications) {
                modifications.forEach(statement => {
                    if (!statement._included) {
                        let statements = this.expandStatement(statement);
                        result.push(...statements);
                    }
                });
            }
        });
        return result;
    }
    define(name) {
        console.log('name',name);
        //说明此变量是外部导入进来的
        if (hasOwnProperty(this.imports, name)) {
            //this.imports[localName]={localName,source,importName};
            const { localName, source, importName } = this.imports[name];
            // source .msg
            let importedModule = this.bundle.fetchModule(source, this.path);
            const { localName: exportLocalName } = importedModule.exports[importName];
            //this.exports.name = {localName:'name',exportName:'name',expression:};
            return importedModule.define(exportLocalName);
        } else {//说明是在当前模块内声明的
+            let statement = this.definitions[name];
+            if (statement) {
+                if(statement._included){
+                    return [];
+                }else{
+                    return this.expandStatement(statement);
+                }
+            } else if (SYSTEM_VARIABLE.includes(name)) {
+                return [];
+            } else {
+                throw new Error(`变量${name}没有既没有从外部导入，也没有在当前模块内声明！`)
+            }
        }
    }
}
module.exports = Module;
```

7. 实现变量名重命名

** 7.1 src\main.js #**

src\main.js

```
import {age1} from './age1.js';
import {age2} from './age2.js';
console.log(age1,age2);
```

** 7.2 src\age1.js #**

src\age1.js

```
const age = '年龄';
export const age1 = age+'1';
```

** 7.3 src\age2.js #**

age2.js

```
const age = '年龄';
export const age2 = age+'2';
```

** 7.4 bundle.js #**

```
const _age = '年龄';
const age1 = _age+'1';
const age = '年龄';
const age2 = age+'2';
console.log(age1,age2);
```

** 7.5 lib\utils.js #**

lib\utils.js

```
const walk = require('./ast/walk');
function hasOwnProperty(obj, prop) {
    return Object.prototype.hasOwnProperty.call(obj, prop);
}
function keys(obj) {
    return Object.keys(obj);
}
function replaceIdentifiers(statement, source, replacements) {
    walk(statement, {
        enter(node) {
            if (node.type === 'Identifier') {
                if(node.name && replacements[node.name]){
                    source.overwrite(node.start, node.end, replacements[node.name]);
                }
            }
        }
    })
}
exports.hasOwnProperty=hasOwnProperty;
exports.keys=keys;
exports.replaceIdentifiers=replaceIdentifiers;
```

** 7.6 analyse.js #**

lib\ast\analyse.js

```
let walk = require('./walk');
let Scope = require('./scope');
const { hasOwnProperty } = require('../utils');
+function analyse(ast, magicStringOfAst,module) {
    //在遍历之前先创建作用域对象
    let scope = new Scope({ name: '全局作用域' });
    ast.body.forEach(statement => {
        function addToScope(name, isBlockDeclaration) {
            scope.add(name, isBlockDeclaration);
            if (!scope.parent|| currentScope.isBlockScope) {//如果此作用域 没有父作用域,那这就是顶级变量，根作用域下的变量，可以放在_defines=tree
                //模块内的顶级变量
                statement._defines[name] = true;
            }
        }
        Object.defineProperties(statement, {
+           _module: { value: module },
            _modifies: { value: {} },//修改
            _defines: { value: {} },//当前statement语法树节点声明了哪些变量
            _dependsOn: { value: {} },//当前statement语句外部依赖的变量
            _included: { value: false, writable: true },//当前的语句是否放置在结果中，是否会出现在打包结果中
            _source: { value: magicStringOfAst.snip(statement.start, statement.end) }//key是_source 值是这个语法树节点在源码中的源代码
        });
        //如何知道某个变量有没有在当前模块内定义的呢?
        //原理是这样 扫描整个模块，找到所有的定义的变量
        //构建使用域链
        walk(statement, {
            enter(node) {
                let newScope;
                switch (node.type) {
                    case 'FunctionDeclaration':
                        //函数的参数将会成为此函数子作用域内的局部变量
                        const names = node.params.map(param => param.name);
                        addToScope(node.id.name);//把node也就是say这个变量添加到当前作用内
                        //如果遇到函数声明，就会产生一个新作用域
                        newScope = new Scope({ name: node.id.name, parent: scope, params: names, block: false });
                        break;
                    case 'BlockStatement':
                        newScope = new Scope({
                            parent: scope,
                            block: true
                        })
                        break;
                    case 'VariableDeclaration':
                        node.declarations.forEach((declaration) => {
                            if(node.kind
                                addToScope(declaration.id.name,true);
                            }else{
                                addToScope(declaration.id.name)
                            }
                        });
                        break;
                }
                if (newScope) {//如果创建了新的作用域，那么这个作用域将会成为新的当前作用域
                    Object.defineProperty(node, '_scope', { value: newScope });
                    scope = newScope;// say这个函数作用域
                }
            },
            leave(node) {
                //当离开节点的时候 ，如果发现这个节点创建 新的作用域，就回退到使用域
                if (hasOwnProperty(node, '_scope')) {
                    scope = scope.parent;
                }
            }
        });

    });
    //2.作用域链构建完成后，再遍历一次，找出本模块定义了依赖了哪些外部变量
    ast.body.forEach(statement => {
        function checkForReads(node) {
```

```
            if (node.type
                const definingScope = scope.findDefiningScope(node.name);
                //-if (!definingScope) {
                    statement._dependsOn[node.name] = true;//添加外部依赖
                //-}
            }
        }
        function checkForWrites(node) {
            function addNode(node) {
                while (node.type
                    node = node.object;
                }
                if (node.type !== 'Identifier') {
                    return
                }
                statement._modifies[node.name] = true;
            }
            if (node.type
                addNode(node.left, true)
            } else if (node.type
                addNode(node.argument, true)
            }
        }
        walk(statement, {
            enter(node) {
                if (hasOwnProperty(node, '_scope')) scope = node._scope;
                checkForReads(node)
                checkForWrites(node)
            },
            leave(node) {
                if (node._scope) scope = scope.parent
            }
        })
    });

}
module.exports = analyse;
```

** 7.7 module.js #**

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
const analyse = require('./ast/analyse');
const { hasOwnProperty } = require('./utils');
const SYSTEM_VARIABLE = ['console', 'log'];
class Module {
    constructor({ code, path, bundle }) {
        this.code = new MagicString(code, { filename: path });
        this.path = path;
        this.bundle = bundle;
        this.ast = parse(code, {
            ecmaVersion: 8,
            sourceType: 'module'
        });
        this.imports = {};//导入
        this.exports = {};//导出
        this.definitions = {};//此变量存放所有的变量定义的语句
        this.modifications = {};
+       this.canonicalNames = {};
        this.analyse();
    }
    analyse() {
        this.ast.body.forEach(statement => {
            //1.给import赋值
            if (statement.type
                let source = statement.source.value;//./msg
                statement.specifiers.forEach(specifier => {
                    let importName = specifier.imported.name;//age
                    let localName = specifier.local.name;//age
                    //记录一下当前的这个引入的变量是从哪个模块的哪个变量导入进来的
                    this.imports[localName] = { localName, source, importName };
                });
                //2.this.exports赋值
            } else if (statement.type
                let declaration = statement.declaration;
                if (declaration.type
                    const declarations = declaration.declarations;
                    declarations.forEach(variableDeclarator => {
                        let localName = variableDeclarator.id.name;
                        //this.exports.age = {localName:'age',exportName:'age',expression:};
                        this.exports[localName] = { localName, exportName: localName, expression: declaration };
                    });
                }
            }
        });
        //1.构建了作用域 2.找到模块块依赖了哪些外部变量
        analyse(this.ast, this.code, this);
        this.ast.body.forEach(statement => {
            Object.keys(statement._defines).forEach(name => {
                //当前模块内 定义name这个变量的语句是statement
                //main.js  type   let type = 'dog';
                this.definitions[name] = statement;
            });
            Object.keys(statement._modifies).forEach(name => {
                if (!hasOwnProperty(this.modifications, name)) {
                    this.modifications[name] = [];
                }
                this.modifications[name].push(statement);
            })
        });
    }
```

```
    expandAllStatements() {
        let allStatements = [];
        this.ast.body.forEach(statement => {
            if (statement.type
                return;
            }
            if (statement.type
            let statements = this.expandStatement(statement);
            //我们要把statement进行扩展，有可能一行变多行var name ='zhufeng', console.log('name'); TODO
            allStatements.push(...statements);
        });
        return allStatements;
    }
    //展开单 条语句  console.log(name,age);
    expandStatement(statement) {
        console.log(statement);
        statement._included = true;// 把当前语句标记为包含
        let result = [];
        const dependencies = Object.keys(statement._dependsOn);
        dependencies.forEach(name => {
            //获取依赖的变量对应的变量定义语句 name
            let definition = this.define(name);
            result.push(...definition);
        });
        result.push(statement);//再放进来当前的语句
        const defines = Object.keys(statement._defines);
        defines.forEach(name => {
            const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
            if (modifications) {
                modifications.forEach(statement => {
                    if (!statement._included) {
                        let statements = this.expandStatement(statement);
                        result.push(...statements);
                    }
                });
            }
        });
        return result;
    }
    define(name) {
        console.log('name',name);
        //说明此变量是外部导入进来的
        if (hasOwnProperty(this.imports, name)) {
            //this.imports[localName]={localName,source,importName};
            const { localName, source, importName } = this.imports[name];
            // source .msg
            let importedModule = this.bundle.fetchModule(source, this.path);
            const { localName: exportLocalName } = importedModule.exports[importName];
            //this.exports.name = {localName:'name',exportName:'name',expression:};
            return importedModule.define(exportLocalName);
        } else {//说明是在当前模块内声明的
            let statement = this.definitions[name];
            if (statement) {
                if(statement._included){
                    return [];
                }else{
                    return this.expandStatement(statement);
                }
            } else if (SYSTEM_VARIABLE.includes(name)) {
                return [];
            } else {
                throw new Error(`变量${name}没有既没有从外部导入，也没有在当前模块内声明！`)
            }
        }
    }
+   rename(name, replacement) {
+     this.canonicalName[name] = replacement;
+   }
+   getCanonicalName(name) {
+     return this.canonicalName[name] || name;
+   }f
}
module.exports = Module;
```

** 7.8 lib\bundle.js #**

lib\bundle.js

```
const MagicString = require('magic-string');
const path = require('path');
const fs = require('fs');
const Module = require('./module');
+let { hasOwnProperty, keys, replaceIdentifiers } = require('./utils');
class Bundle {
    constructor(entry) {
        this.entryPath = path.resolve(entry);//得到入口文件的绝对路径
        this.modules = {};//存放着本次打包的所有的模块
    }
    //负责编译入口文件，然后把结果写入outputFile
    build(outputFile) {
        //先获取入口模块的实例
        let entryModule = (this.entryModule = this.fetchModule(this.entryPath));
        //展开入口模块语句
        this.statements = entryModule.expandAllStatements();
+       this.deConflict();
        const transformedCode = this.generate();
        fs.writeFileSync(outputFile, transformedCode);
    }
+   deconflict() {
+     const defines = {};//定义的变量
+     const conflicts = {};//冲突的变量
+     this.statements.forEach(statement => {
+       //循环此语句上定义的所有的变量
+       keys(statement._defines).forEach(name => {
```

```
+         //判断此变量name是否已经定义过了
+         if (hasOwnProperty(defines, name)) {
+           //此变量已经出现过了，就标记为此变量冲突
+           conflicts[name] = true;
+         } else {
+           defines[name] = [];
+         }
+         defines[name].push(statement._module);
+       })
+     })
+     Object.keys(conflicts).forEach(name => {
+       const modules = defines[name];
+       //最后一个模块不需要重命名
+       modules.pop();
+       modules.forEach((module, index) => {
+         const replacement = `${name}${modules.length - index}`;
+         module.rename(name, replacement);
+       });
+
+     });
+   }
    /**
     * 根据模块的绝对路径返回模块的实例
     * @param {*} importee 被 导入的模块的绝对路径也可能是相对
     * @param {*} importer 导入的模块绝对路径
     */
    fetchModule(importee, importer) {
        let route;
        if (!importer) {
            route = importee;
        } else {
            if (path.isAbsolute(importee)) {
                route = importee;
            } else {
                route = path.resolve(path.dirname(importer), importee);
            }
        }
        if (route) {
            let code = fs.readFileSync(route, 'utf8');
            const module = new Module({
                code,
                path: route,
                bundle: this
            });
            return module;
        }
    }
    generate() {
        let transformedCode = new MagicString.Bundle();//字符串包
        //this.statements只有入口模块里所有的顶层节点
        this.statements.forEach(statement => {
+           let replacements = {};
+           Object.keys(statement._dependsOn)
+               .concat(Object.keys(statement._defines))
+               .forEach(name => {
+                   const canonicalName = statement._module.getCanonicalName(name)
+                   if (name !== canonicalName) {
+                       replacements[name] = canonicalName;
+                   }
+               })
            //我如何向statement这个顶级节点上添加_source属性，值是magicString实例
            const content = statement._source.clone();
            if (/^Export/.test(statement.type)) {
                if (statement.type
                    content.remove(statement.start, statement.declaration.start);
                }
            }
+           replaceIdentifiers(statement, content, replacements);
            transformedCode.addSource({
                content,
                separator: '\n'
            });
        });
        return transformedCode.toString();
    }
}
module.exports = Bundle;
//webpack Compiler
```