link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=159 sentences=859, words=5024

1.前置知识

1.1. 初始化项目

```
npm install rollup magic-string acorn --save
```

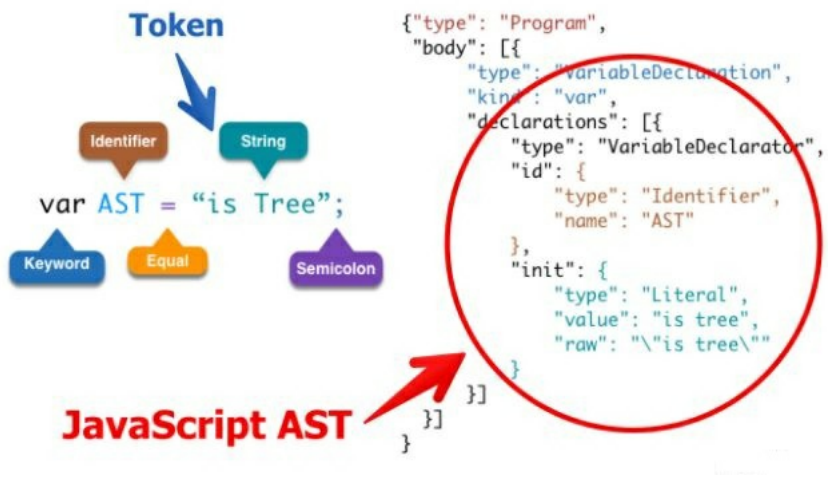
1.2. magic-string

- 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());
```

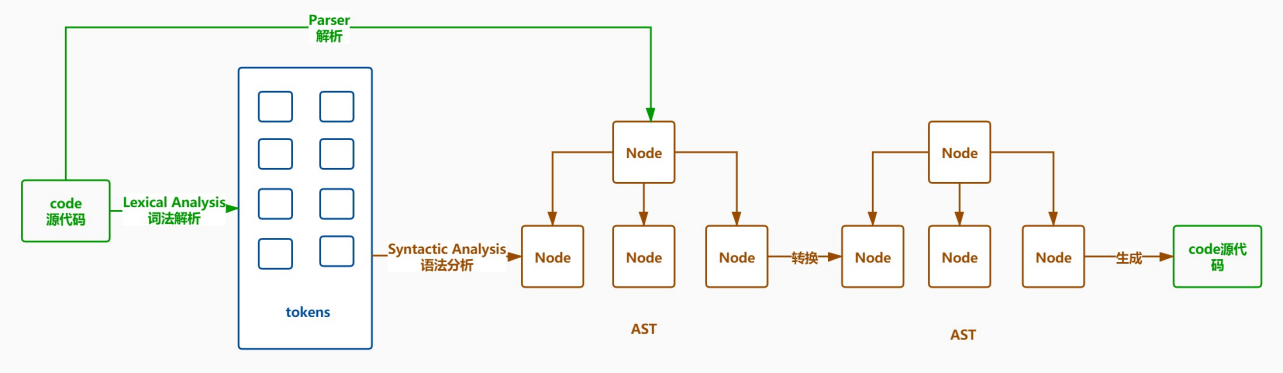
2.3. AST

- 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

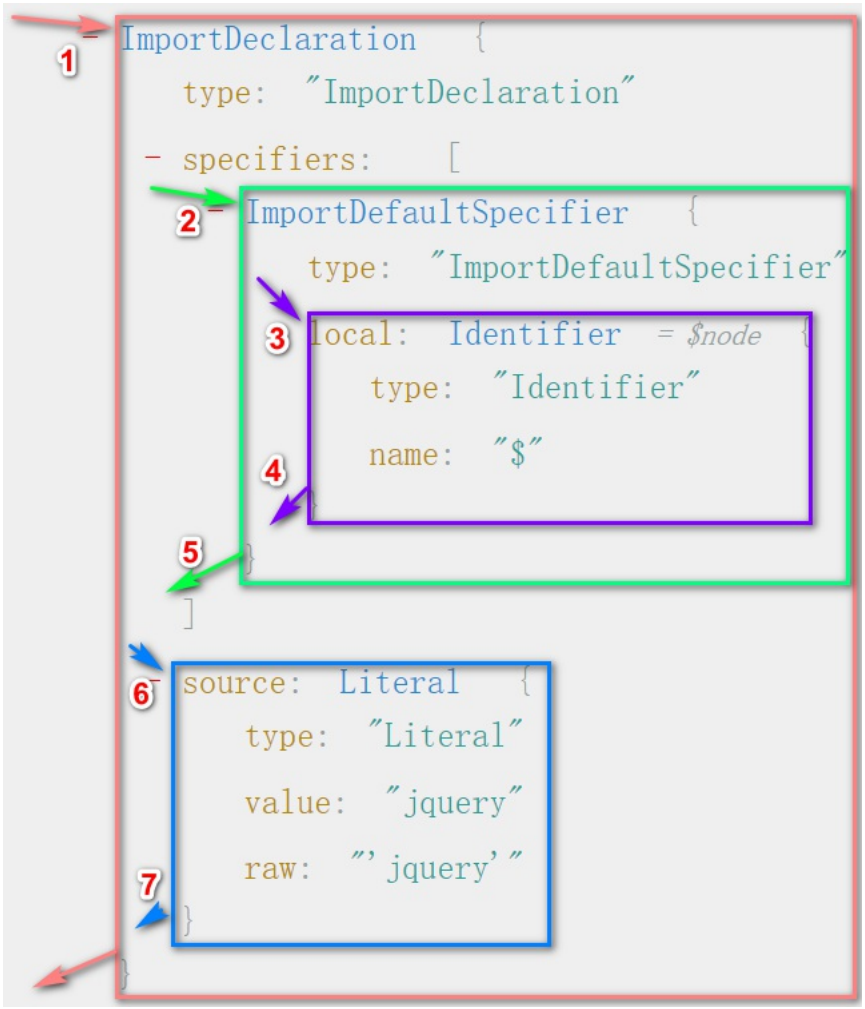
- astexplorer (https://astexplorer.net/)可以把代码转成语法树
- acorn 解析结果符合 The Estree Spec规范

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, {
 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
```

1.4 作用域

1.4.1 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);
```

1.4.2 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

1.4.2.1 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;
```

1.4.2.2 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
```

2. 实现rollup

2.1 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
```

2.2 src\main.js

src\main.js

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

2.3 debugger.js

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

2.4 rollup.js

lib\rollup.js

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

2.5 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) {
    const 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() {
    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;
```

2.6 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: 8,
      sourceType: 'module'
    })
    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;
```

2.7 analyse.js

lib\ast\analyse.js

```
function analyse(ast, code,module) {

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

3. 实现tree-shaking

3.1 基本原理

- 第一步
  - 在 module里收集 imports、 exports和 definitions
  - 在 analyse收集 _defines和 _dependsOn
- 第二步
  - 重构 expandAllStatements

3.2 基本语句

```
this.imports = {};

this.exports = {};

this.definitions = {};

this.modifications = {};

this.canonicalNames = {};

this.imports[localName] = { source, importName }

this.exports[exportName] = { localName };

statement._defines[name] = true;

this.definitions[name] = statement;

statement._dependsOn[name] = true;

module.define(name);
```

3.3 main.js

src\main.js

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

3.4 msg.js

src\msg.js

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

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) {
    //入口文件数据
    this.entryPath = path.resolve(options.entry.replace(/\.js$/, '') + '.js');
    //存放所有的模块
    this.modules = {};
  }
  build(filename) {
    const entryModule = this.fetchModule(this.entryPath);//获取模块代码
    this.statements = entryModule.expandAllStatements(true);//展开所有的语句
    const { code } = this.generate();//生成打包后的代码
    fs.writeFileSync(filename, code);//写入文件系统
  }
  fetchModule(importee, importer) {
+   let route;
+   if (!importer) {
+     route = importee;
+   } else {
+     if (path.isAbsolute(importee)) {
+       route = importee;
+     } else {
+       route = path.resolve(path.dirname(importer), importee.replace(/\.js$/, '') + '.js');
+     }
+   }
    if (route) {
      let code = fs.readFileSync(route, 'utf8');
      const module = new Module({
        code,
        path: importee,
        bundle: this
      })
      return module;
    }
  }
  generate() {
    let magicString = new MagicString.Bundle();
    this.statements.forEach(statement => {
      const source = statement._source.clone();
+     if (statement.type === 'ExportNamedDeclaration') {
+       source.remove(statement.start, statement.declaration.start);
+     }
      magicString.addSource({
        content: source,
        separator: '\n'
      })
    })
    return { code: magicString.toString() }
  }
}
module.exports = Bundle;
```

3.6 utils.js

lib\utils.js

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

3.7 module.js

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
+const { hasOwnProperty } = require('./utils');
const 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: 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 = {};
    analyse(this.ast, this.code, this);
  }
  expandAllStatements() {
    let allStatements = [];
    this.ast.body.forEach(statement => {
+     //导入的语句默认全部过滤掉
+     if (statement.type === 'ImportDeclaration') return;
      let statements = this.expandStatement(statement);
      allStatements.push(...statements);
    });
    return allStatements;
  }
  expandStatement(statement) {
    statement._included = true;
    let result = [];
+   //获取此语句依赖的变量
+   let _dependsOn = Object.keys(statement._dependsOn);
+   _dependsOn.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);
+   });
    //把此语句添加到输出列表中
    result.push(statement);
    return result;
  }
+ define(name) {
+   //先判断此变量是外部导入的还是模块内声明的
+   if (hasOwnProperty(this.imports, name)) {
+     //说明此变量不是模块内声明的，而是外部导入的,获取从哪个模块内导入了哪个变量
+     const { source, importName } = this.imports[name];
+     //获取这个模块
+     const importModule = this.bundle.fetchModule(source, this.path);
+     //从这个模块的导出变量量获得本地变量的名称
+     const { localName } = importModule.exports[importName];
+     //获取本地变量的定义语句
+     return importModule.define(localName);//name
+   } else {//如果是模块的变量的话
+     let statement = this.definitions[name];//name
+     if (statement && !statement._included) {
+       //如果本地变量的话还需要继续展开
+       return this.expandStatement(statement);
+     } else {
+       return []
+     }
+   }
+ }
}
module.exports = Module;
```

3.8 analyse.js

- 第1个循环 找出导入导出的变量
- 第2个循环 找出定义和依赖的变量

lib\ast\analyse.js

```
+const Scope = require('./scope');
+const walk = require('./walk');
+const { hasOwnProperty } = require('../utils');
+function analyse(ast, code, module) {
+  //第1个循环，找出导入导出的变量
+  ast.body.forEach(statement => {
+    Object.defineProperties(statement, {
       _module: { value: module }
       _source: { value: code.snip(statement.start, statement.end) },
+      _defines: { value: {} },//此节点上定义的变量say
+      _dependsOn: { value: {} }//此此节点读取了哪些变量
+    })
+    //import { name, age } from './msg';
+    if (statement.type === 'ImportDeclaration') {
+      let source = statement.source.value;// ./msg
+      statement.specifiers.forEach(specifier => {
+        let importName = specifier.imported.name;//导入的变量名
+        let localName = specifier.local.name;//本地的变量名
+        //imports.name = {source:'./msg',importName:'name'};
+        module.imports[localName] = { source, importName }
+      });
+    } else if (statement.type === 'ExportNamedDeclaration') {
+      const declaration = statement.declaration;
+      if (declaration && declaration.type === 'VariableDeclaration') {
+        const declarations = declaration.declarations;
+        declarations.forEach(variableDeclarator => {
+          const localName = variableDeclarator.id.name;//name
+          const exportName = localName;
+          //exports.name = {localName:'name'};
+          module.exports[exportName] = { localName };
+        });
+      }
+    }
+  });
+  //第2次循环创建作用域链
+  let currentScope = new Scope({ name: '全局作用域' });
+  //创建作用域链,为了知道我在此模块中声明哪些变量，这些变量的声明节点是哪个 var name = 1;
+  ast.body.forEach(statement => {
+    function addToScope(name) {
+      currentScope.add(name);//把name变量放入当前的作用域
+      //如果没有父亲，相当 于就是根作用域或者 当前的作用域是一个块级作用域的话
+      if (!currentScope.parent) {//如果没有父作用域，说明这是一个顶级作用域
+        statement._defines[name] = true;//在一级节点定义一个变量name _defines.say=true
+        module.definitions[name] = statement;
+      }
+    }
+    walk(statement, {
+      enter(node) {
+        //收集本节点上使用的变量
+        if (node.type === 'Identifier') {
+          statement._dependsOn[node.name] = true;
+        }
+        let newScope;
+        switch (node.type) {
+          case 'FunctionDeclaration':
+            addToScope(node.id.name);//say
+            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);//var
+            });
+            break;
+          default:
+            break;
+        }
+        if (newScope) {
+          Object.defineProperty(node, '_scope', { value: newScope });
+          currentScope = newScope;
+        }
+      },
+      leave(node) {
+        if (hasOwnProperty(node, '_scope')) {
+          currentScope = currentScope.parent;
+        }
+      }
+    });
+  });
+}
module.exports = analyse;
```

3.9 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;
```

3.10 lib\ast\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.包含修改语句

4.1 main.js

src\main.js

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

4.2 msg.js

src\msg.js

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

4.3 module.js

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
const { hasOwnProperty } = require('./utils');
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: 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 = {};
    //存放变量修改语句
+   this.modifications = {};
    analyse(this.ast, this.code, this);
  }
  expandAllStatements() {
    let allStatements = [];
    this.ast.body.forEach(statement => {
      if (statement.type
      let statements = this.expandStatement(statement);
      allStatements.push(...statements);
    });
    return allStatements;
  }
  expandStatement(statement) {
    statement._included = true;
    let result = [];
    //获取此语句依赖的变量
    let _dependsOn = Object.keys(statement._dependsOn);
    _dependsOn.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);
    });
    //把此语句添加到输出列表中
    result.push(statement);
+   //找到此语句定义的变量,把定义的变量和修改语句也包括进来
+   //注意要先定义再修改，所以要把这行放在push(statement)的下面
+   const defines = Object.keys(statement._defines);
+   defines.forEach(name => {
+     //找到定义的变量依赖的修改的语句
+     const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
+     if (modifications) {
+       //把修改语句也展开放到结果里
+       modifications.forEach(statement => {
+         if (!statement._included) {
+           let statements = this.expandStatement(statement);
+           result.push(...statements);
+         }
+       });
+     }
+   });
    return result;
  }
  define(name) {
    //先判断此变量是外部导入的还是模块内声明的
    if (hasOwnProperty(this.imports, name)) {
      //说明此变量不是模块内声明的，而是外部导入的,获取从哪个模块内导入了哪个变量
      const { source, importName } = this.imports[name];
      //获取这个模块
      const importModule = this.bundle.fetchModule(source, this.path);
      //从这个模块的导出变量量获得本地变量的名称
      const { localName } = importModule.exports[importName];
      //获取本地变量的定义语句
      return importModule.define(localName);//name
    } else {//如果是模块的变量的话
      let statement = this.definitions[name];//name
      if (statement && !statement._included) {
        //如果本地变量的话还需要继续展开
        return this.expandStatement(statement);
      } else {
        return []
      }
    }
  }
}
module.exports = Module;
```

4.4 analyse.js

lib\ast\analyse.js

```
const Scope = require('./scope');
const walk = require('./walk');
const { hasOwnProperty } = require('../utils');
function analyse(ast, code, module) {
  //第1个循环，找出导入导出的变量
  ast.body.forEach(statement => {
    Object.defineProperties(statement, {
      _module: { value: module },
      _source: { value: code.snip(statement.start, statement.end) },
      _defines: { value: {} },//此节点上定义的变量say
      _dependsOn: { value: {} },//此此节点读取了哪些变量
+     _modifies: { value: {} },//本语句修改的变量
    })
    //import { name, age } from './msg';
```

```
    if (statement.type
      let source = statement.source.value;// ./msg
      statement.specifiers.forEach(specifier => {
        let importName = specifier.imported.name;//导入的变量名
        let localName = specifier.local.name;//本地的变量名
        //imports.name = {source:'./msg',importName:'name'};
        module.imports[localName] = { source, importName }
      });
    } else if (statement.type
      const declaration = statement.declaration;
      if (declaration && declaration.type
        const declarations = declaration.declarations;
        declarations.forEach(variableDeclarator => {
          const localName = variableDeclarator.id.name;//name
          const exportName = localName;
          //exports.name = {localName:'name'};
          module.exports[exportName] = { localName };
        });
      }
    }
  });
  //第2次循环创建作用域链
  let currentScope = new Scope({ name: '全局作用域' });
  //创建作用域链,为了知道我在此模块中声明哪些变量，这些变量的声明节点是哪个 var name = 1;
  ast.body.forEach(statement => {
+   function checkForReads(node) {
+     //如果此节点类型是一个标识符话
+     if (node.type === 'Identifier') {
+       statement._dependsOn[node.name] = true;
+     }
+   }
+   function checkForWrites(node) {
+     function addNode(node) {
+       const name = node.name;
+       statement._modifies[name] = true;//statement._modifies.age = true;
+       if (!hasOwnProperty(module.modifications, name)) {
+         module.modifications[name] = [];
+       }
+       module.modifications[name].push(statement);
+     }
+     if (node.type === 'AssignmentExpression') {
+       addNode(node.left, true);
+     } else if (node.type === 'UpdateExpression') {
+       addNode(node.argument);
+     }
+   }
    function addToScope(name) {
      currentScope.add(name);//把name变量放入当前的作用域
      //如果没有父亲，相当 于就是根作用域或者 当前的作用域是一个块级作用域的话
      if (!currentScope.parent) {//如果没有父作用域，说明这是一个顶级作用域
        statement._defines[name] = true;//在一级节点定义一个变量name _defines.say=true
        module.definitions[name] = statement;
      }
    }
    walk(statement, {
      enter(node) {
-       if (node.type === 'Identifier') {
-         statement._dependsOn[node.name] = true;
-       }
+       //收集本节点上使用的变量
+       checkForReads(node);
+       checkForWrites(node);
        let newScope;
        switch (node.type) {
          case 'FunctionDeclaration':
            addToScope(node.id.name);//say
            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);//var
            });
            break;
          default:
            break;
        }
        if (newScope) {
          Object.defineProperty(node, '_scope', { value: newScope });
          currentScope = newScope;
        }
      },
      leave(node) {
        if (hasOwnProperty(node, '_scope')) {
          currentScope = currentScope.parent;
        }
      }
    });
  });
}
module.exports = analyse;
```

4.支持块级作用域

4.1 src\main.js

src\main.js

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

5.2 lib\ast\scope.js

lib\ast\scope.js

```
class Scope {
  constructor(options = {}) {
    //作用域的名称
    this.name = options.name;
    //父作用域
    this.parent = options.parent;
    //此作用域内定义的变量
    this.names = options.names || [];
    // 是否块作用域
+   this.isBlock = !!options.isBlock
  }
+ add(name, isBlockDeclaration) {
+   if (!isBlockDeclaration && this.isBlock) {
      //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);
    }
  }
  findDefiningScope(name) {
    if (this.names.includes(name)) {
      return this;
    } else if (this.parent) {
      return this.parent.findDefiningScope(name);
    } else {
      return null;
    }
  }
}
module.exports = Scope;
```

5.3 analyse.js

lib\ast\analyse.js

```
const Scope = require('./scope');
const walk = require('./walk');
const { hasOwnProperty } = require('../utils');
function analyse(ast, code, module) {
  //第1个循环，找出导入导出的变量
  ast.body.forEach(statement => {
    Object.defineProperties(statement, {
      _module: { value: module },
      _source: { value: code.snip(statement.start, statement.end) },
      _defines: { value: {} },//此节点上定义的变量say
      _dependsOn: { value: {} },//此此节点读取了哪些变量
      _modifies: { value: {} },//本语句修改的变量
    })
    //import { name, age } from './msg';
    if (statement.type
      let source = statement.source.value;// ./msg
      statement.specifiers.forEach(specifier => {
        let importName = specifier.imported.name;//导入的变量名
        let localName = specifier.local.name;//本地的变量名
        //imports.name = {source:'./msg',importName:'name'};
        module.imports[localName] = { source, importName }
      });
    } else if (statement.type
      const declaration = statement.declaration;
      if (declaration && declaration.type
        const declarations = declaration.declarations;
        declarations.forEach(variableDeclarator => {
          const localName = variableDeclarator.id.name;//name
          const exportName = localName;
          //exports.name = {localName:'name'};
          module.exports[exportName] = { localName };
        });
      }
    }
  });
  //第2次循环创建作用域链
  let currentScope = new Scope({ name: '全局作用域' });
  //创建作用域链,为了知道我在此模块中声明哪些变量，这些变量的声明节点是哪个 var name = 1;
  ast.body.forEach(statement => {
    function checkForReads(node) {
      //如果此节点类型是一个标识符话
      if (node.type
        statement._dependsOn[node.name] = true;
      }
    }
    function checkForWrites(node) {
      function addNode(node) {
        const name = node.name;
        statement._modifies[name] = true;//statement._modifies.age = true;
        if (!hasOwnProperty(module.modifications, name)) {
          module.modifications[name] = [];
        }
        module.modifications[name].push(statement);
      }
      if (node.type
        addNode(node.left, true);
      } else if (node.type
        addNode(node.argument);
      }
    }
+   function addToScope(name, isBlockDeclaration) {
+     currentScope.add(name, isBlockDeclaration);//把name变量放入当前的作用域
      //如果没有父亲，相当 于就是根作用域或者 当前的作用域是一个块级作用域的话
+     if (!currentScope.parent || (currentScope.isBlock && !isBlockDeclaration)) {//如果没有父作用域，说明这是一个顶级作用域
        statement._defines[name] = true;//在一级节点定义一个变量name _defines.say=true
```

```
        module.definitions[name] = statement;
      }
    }
    walk(statement, {
      enter(node) {
        //收集本节点上使用的变量
        checkForReads(node);
        checkForWrites(node);
        let newScope;
        switch (node.type) {
          case 'FunctionDeclaration':
            addToScope(node.id.name);//say
            const names = node.params.map(param => param.name);
+           newScope = new Scope({ name: node.id.name, parent: currentScope, names, isBlock: false });
            break;
          case 'VariableDeclaration':
            node.declarations.forEach(declaration => {
+             if (node.kind === 'let' || node.kind === 'const') {
+               addToScope(declaration.id.name, true);//这是是一个块级变量
+             } else {
+               addToScope(declaration.id.name);//var
+             }
            });
            break;
+         case 'BlockStatement':
+           newScope = new Scope({ parent: currentScope, isBlock: true });
+           break;
          default:
            break;
        }
        if (newScope) {
          Object.defineProperty(node, '_scope', { value: newScope });
          currentScope = newScope;
        }
      },
      leave(node) {
        if (hasOwnProperty(node, '_scope')) {
          currentScope = currentScope.parent;
        }
      }
    });
  });

}
module.exports = analyse;
```

5.4 module.js

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
const { hasOwnProperty } = require('./utils');
const analyse = require('./ast/analyse');
+const SYSTEMS = ['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'
    })
    //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 = {};
    //存放变量修改语句
    this.modifications = {};
    analyse(this.ast, this.code, this);
  }
  expandAllStatements() {
    let allStatements = [];
    this.ast.body.forEach(statement => {
      if (statement.type
+     //默认不包含所有的变量声明语句
+     if (statement.type === 'VariableDeclaration') return;
      let statements = this.expandStatement(statement);
      allStatements.push(...statements);
    });
    return allStatements;
  }
  expandStatement(statement) {
    statement._included = true;
    let result = [];
    //获取此语句依赖的变量
    let _dependsOn = Object.keys(statement._dependsOn);
    _dependsOn.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);
    });
    //把此语句添加到输出列表中
    result.push(statement);
    //找到此语句定义的变量,把定义的变量和修改语句也包括进来
    //注意要先定义再修改，所以要把这行放在push(statement)的下面
    const defines = Object.keys(statement._defines);
    defines.forEach(name => {
      //找到定义的变量依赖的修改的语句
      const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
      if (modifications) {
        //把修改语句也展开放到结果里
        modifications.forEach(statement => {
          if (!statement._included) {
            let statements = this.expandStatement(statement);
            result.push(...statements);
          }
        });
      }
    });
    return result;
  }
  define(name) {
    //先判断此变量是外部导入的还是模块内声明的
    if (hasOwnProperty(this.imports, name)) {
      //说明此变量不是模块内声明的，而是外部导入的,获取从哪个模块内导入了哪个变量
      const { source, importName } = this.imports[name];
      //获取这个模块
      const importModule = this.bundle.fetchModule(source, this.path);
      //从这个模块的导出变量量获得本地变量的名称
      const { localName } = importModule.exports[importName];
      //获取本地变量的定义语句
      return importModule.define(localName);//name
    } else {//如果是模块的变量的话
      let statement = this.definitions[name];//name
+     if (statement) {
+       if (statement._included) {
+         return [];
+       } else {
+         return this.expandStatement(statement);
+       }
+     } else {
+       if (SYSTEMS.includes(name)) {
+         return [];
+       } else {  //如果找不到定义的变量就报错
+         throw new Error(`变量${name}既没有从外部导入，也没有在当前的模块声明`);
+       }
      }
    }
  }
}
module.exports = Module;
```

6.实现变量名重命名

6.1 src\main.js

src\main.js

```
import { age1 } from './age1.js';
import { age2 } from './age2.js';
import { age3 } from './age3.js';
console.log(age1, age2, age3);
```

6.2 src\age1.js

src\age1.js

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

6.3 src\age2.js

src\age2.js

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

6.4 src\age3.js

src\age3.js

```
const age = '年龄';
export const age3 = age + '3';
```

6.5 utils.js

lib\utils.js

```
+const walk = require('./ast/walk');
function hasOwnProperty(obj, prop) {
  return Object.prototype.hasOwnProperty.call(obj, prop)
}
exports.hasOwnProperty = hasOwnProperty;
+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.replaceIdentifiers = replaceIdentifiers;
```

6.6 bundle.js

lib\bundle.js

```
let fs = require('fs');
let path = require('path');
let Module = require('./module');
let MagicString = require('magic-string');
+const { hasOwnProperty, replaceIdentifiers } = require('./utils');
class Bundle {
  constructor(options) {
    //入口文件数据
    this.entryPath = path.resolve(options.entry.replace(/\.js$/, '') + '.js');
    //存放所有的模块
    this.modules = {};
  }
  build(filename) {
    const entryModule = this.fetchModule(this.entryPath);//获取模块代码
    this.statements = entryModule.expandAllStatements(true);//展开所有的语句
+   this.deconflict();
    const { code } = this.generate();//生成打包后的代码
    fs.writeFileSync(filename, code);//写入文件系统
    console.log('done');
  }
+ deconflict() {
+   const defines = {};//定义的变量
+   const conflicts = {};//变量名冲突的变量
+   this.statements.forEach(statement => {
+     Object.keys(statement._defines).forEach(name => {
+       if (hasOwnProperty(defines, name)) {
+         conflicts[name] = true;
+       } else {
+         defines[name] = [];//defines.age = [];
+       }
+       //把此声明变量的语句，对应的模块添加到数组里
+       defines[name].push(statement._module);
+     });
+   });
+   Object.keys(conflicts).forEach(name => {
+     const modules = defines[name];//获取定义此变量名的模块的数组
+     modules.pop();//最后一个模块不需要重命名,保留 原来的名称即可 [age1,age2]
+     modules.forEach((module, index) => {
+       let replacement = `${name}${modules.length - index}`;
+       debugger
+       module.rename(name, replacement);//module age=>age$2
+     });
+   });
+ }
  fetchModule(importee, importer) {
    let route;
    if (!importer) {
      route = importee;
    } else {
      if (path.isAbsolute(importee)) {
        route = importee;
      } else {
        route = path.resolve(path.dirname(importer), importee.replace(/\.js$/, '') + '.js');
      }
    }
    if (route) {
      let code = fs.readFileSync(route, 'utf8');
      const module = new Module({
        code,
        path: importee,
        bundle: this
      })
      return module;
    }
  }
  generate() {
    let magicString = new MagicString.Bundle();
    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;
+       });
      const source = statement._source.clone();
      if (statement.type
        source.remove(statement.start, statement.declaration.start);
      }
+     replaceIdentifiers(statement, source, replacements);
      magicString.addSource({
        content: source,
        separator: '\n'
      })
    })
    return { code: magicString.toString() }
  }
}
module.exports = Bundle;
```

6.7 module.js

lib\module.js

```
const MagicString = require('magic-string');
const { parse } = require('acorn');
const { hasOwnProperty } = require('./utils');
const analyse = require('./ast/analyse');
const SYSTEMS = ['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'
    })
    //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 = {};
    //存放变量修改语句
    this.modifications = {};
+   this.canonicalNames = {};
    analyse(this.ast, this.code, this);
  }
  expandAllStatements() {
    let allStatements = [];
    this.ast.body.forEach(statement => {
      if (statement.type
      if (statement.type
      let statements = this.expandStatement(statement);
      allStatements.push(...statements);
    });
    return allStatements;
  }
  expandStatement(statement) {
    statement._included = true;
    let result = [];
    //获取此语句依赖的变量
    let _dependsOn = Object.keys(statement._dependsOn);
    _dependsOn.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);
    });
    //把此语句添加到输出列表中
    result.push(statement);
    //找到此语句定义的变量,把定义的变量和修改语句也包括进来
    //注意要先定义再修改，所以要把这行放在push(statement)的下面
    const defines = Object.keys(statement._defines);
    defines.forEach(name => {
      //找到定义的变量依赖的修改的语句
      const modifications = hasOwnProperty(this.modifications, name) && this.modifications[name];
      if (modifications) {
        //把修改语句也展开放到结果里
        modifications.forEach(statement => {
          if (!statement._included) {
            let statements = this.expandStatement(statement);
            result.push(...statements);
          }
        });
      }
    });
    return result;
  }
  define(name) {
    //先判断此变量是外部导入的还是模块内声明的
    if (hasOwnProperty(this.imports, name)) {
      //说明此变量不是模块内声明的，而是外部导入的,获取从哪个模块内导入了哪个变量
      const { source, importName } = this.imports[name];
      //获取这个模块
      const importModule = this.bundle.fetchModule(source, this.path);
      //从这个模块的导出变量量获得本地变量的名称
      const { localName } = importModule.exports[importName];
      //获取本地变量的定义语句
      return importModule.define(localName);//name
    } else {//如果是模块的变量的话
      let statement = this.definitions[name];//name
      if (statement) {
        if (statement._included) {
          return [];
        } else {
          return this.expandStatement(statement);
        }
      } else {
        if (SYSTEMS.includes(name)) {
          return [];
        } else {
          throw new Error(`变量${name}既没有从外部导入，也没有在当前的模块声明`);
        }
      }
    }
  }
+ rename(name, replacement) {
+   this.canonicalNames[name] = replacement;
+ }
+ getCanonicalName(name) {
+   return this.canonicalNames[name] || name;
+ }
}
module.exports = Module;
```