## **1.抽象语法树(Abstract Syntax Tree)**

webpack和Lint等很多的工具和库的核心都是通过Abstract Syntax Tree抽象语法树这个概念来实现对代码的检查、分析等操作的

* 通过了解抽象语法树这个概念，你也可以随手编写类似的工具

## **2.抽象语法树用途**

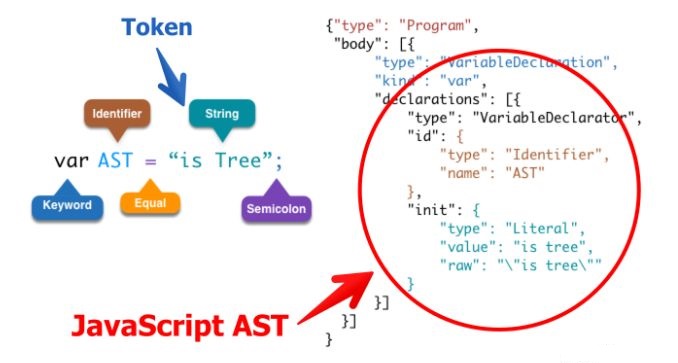
* 代码语法的检查、代码风格的检查、代码的格式化、代码的高亮、代码错误提示、代码自动补全等等
  + 如JSLint、JSHint对代码错误或风格的检查，发现一些潜在的错误
  + IDE的错误提示、格式化、高亮、自动补全等等
* 代码混淆压缩
  + UglifyJS2等
* 优化变更代码，改变代码结构使达到想要的结构
  + 代码打包工具webpack、rollup等等
  + CommonJS、AMD、CMD、UMD等代码规范之间的转化
  + CoffeeScript、TypeScript、JSX等转化为原生Javascript

## **3.抽象语法树定义**

这些工具的原理都是通过JavaScript Parser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

在计算机科学中，抽象语法树（abstract syntax tree或者缩写为AST），或者语法树（syntax tree），是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。

Javascript的语法是为了给开发者更好的编程而设计的，但是不适合程序的理解。所以需要转化为AST来使之更适合程序分析，浏览器编译器一般会把源码转化为AST来进行进一步的分析等其他操作。



## **4.JavaScript Parser**

JavaScript Parser，把js源码转化为抽象语法树的解析器。

浏览器会把js源码通过解析器转为抽象语法树，再进一步转化为字节码或直接生成机器码。

一般来说每个js引擎都会有自己的抽象语法树格式，Chrome的v8引擎，firefox的SpiderMonkey引擎等等，MDN提供了详细SpiderMonkey AST format的详细说明，算是业界的标准。

### **4.1 常用的JavaScript Parser**

* esprima
* traceur
* acorn
* shift

### **4.2 esprima**

* 通过 [esprima](https://www.npmjs.com/package/esprima) 把源码转化为AST
* 通过 [estraverse](https://www.npmjs.com/package/estraverse) 遍历并更新AST
* 通过 [escodegen](https://www.npmjs.com/package/escodegen) 将AST重新生成源码
* [astexplorer](https://astexplorer.net/) AST的可视化工具

mkdir zhufengast

cd zhufengast

cnpm i esprima estraverse escodegen- S

**let** esprima = require('esprima');**var** estraverse = require('estraverse');**var** escodegen = require("escodegen");**let** code = 'function ast(){}';**let** ast=esprima.parse(code);**let** indent=0;**function** **pad**() {

**return** ' '.repeat(indent);

}

estraverse.traverse(ast,{

enter(node) {

console.log(pad()+node.type);

**if**(node.type == 'FunctionDeclaration'){

node.id.name = 'ast\_rename';

}

indent+=2;

},

leave(node) {

indent-=2;

console.log(pad()+node.type);

}

});**let** generated = escodegen.generate(ast);console.log(generated);

Program

FunctionDeclaration

Identifier

Identifier

BlockStatement

BlockStatement

FunctionDeclaration

Program

## **5.babel插件**

* 访问者模式Visitor 对于某个对象或者一组对象，不同的访问者，产生的结果不同，执行操作也不同
* [@babel/core](https://www.npmjs.com/package/@babel/core) Babel 的编译器，核心 API 都在这里面，比如常见的 transform、parse
* [babylon](https://zhufeng-document.vercel.app/html/26.webpack-5-AST.html) Babel 的解析器
* [babel-types](https://github.com/babel/babel/tree/master/packages/babel-types) 用于 AST 节点的 Lodash 式工具库, 它包含了构造、验证以及变换 AST 节点的方法，对编写处理 AST 逻辑非常有用
* [babel-traverse](https://www.npmjs.com/package/babel-traverse)用于对 AST 的遍历，维护了整棵树的状态，并且负责替换、移除和添加节点
* [babel-types-api](https://babeljs.io/docs/en/next/babel-types.html)
* [Babel 插件手册](https://github.com/brigand/babel-plugin-handbook/blob/master/translations/zh-Hans/README.md" \l "asts)
* [babeljs.io](https://babeljs.io/en/repl.html) babel可视化编译器

### **5.1 转换箭头函数**

* [babel-plugin-transform-es2015-arrow-functions](https://www.npmjs.com/package/babel-plugin-transform-es2015-arrow-functions)

转换前

**const** sum = (a,b)=>a+b

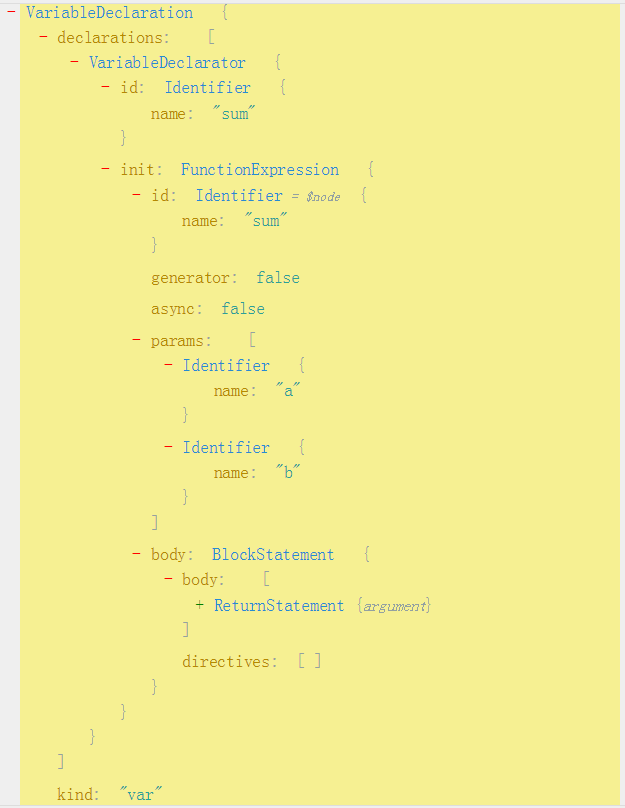


转换后

**var** sum = **function** **sum**(a, b) {

**return** a + b;

};



npm i @babel/core babel-types -D

实现

**let** babel = require('@babel/core');**let** t = require('babel-types');**const** code = `const sum = (a,b)=>a+b`;**let** transformArrowFunctions = {

visitor: {

ArrowFunctionExpression: (path) => {

**let** node = path.node;

**let** id = path.parent.id;

**let** params = node.params;

**let** body=t.blockStatement([

t.returnStatement(node.body)

]);

**let** functionExpression = t.functionExpression(id,params,body,false,false);

path.replaceWith(functionExpression);

}

}

}**const** result = babel.transform(code, {

plugins: [transformArrowFunctions]

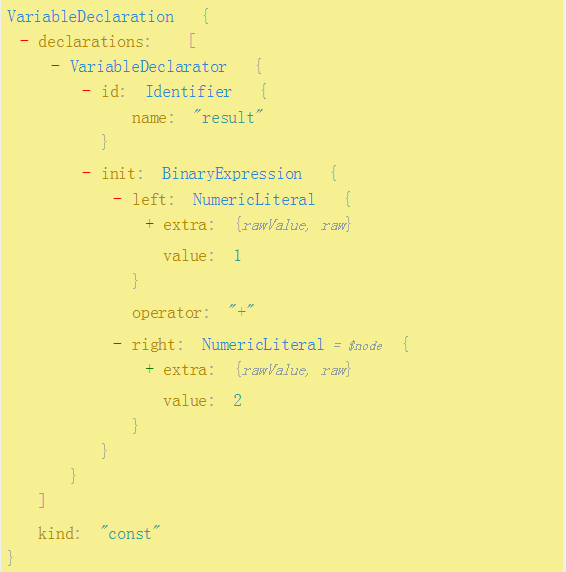
});console.log(result.code);

### **5.2. 预计算babel插件**

* path.parentPath 父路径

转换前

**const** result = 1 + 2;



转换后

**const** result = 3;



**let** babel = require('@babel/core');**let** t=require('babel-types');**let** preCalculator={

visitor: {

BinaryExpression(path) {

**let** node=path.node;

**let** left=node.left;

**let** operator=node.operator;

**let** right=node.right;

**if** (!isNaN(left.value) && !isNaN(right.value)) {

**let** result=eval(left.value+operator+right.value);

path.replaceWith(t.numericLiteral(result));

**if** (path.parent&& path.parent.type == 'BinaryExpression') {

preCalculator.visitor.BinaryExpression.call(null,path.parentPath);

}

}

}

}

}

**const** result = babel.transform('const sum = 1+2+3',{

plugins:[

preCalculator

]

});console.log(result.code);

### **5.3. 把类编译为Function**

* [babel-plugin-transform-es2015-classes](https://www.npmjs.com/package/babel-plugin-transform-es2015-classes)

es6

**class** **Person** {

**constructor**(name) {

**this**.name=name;

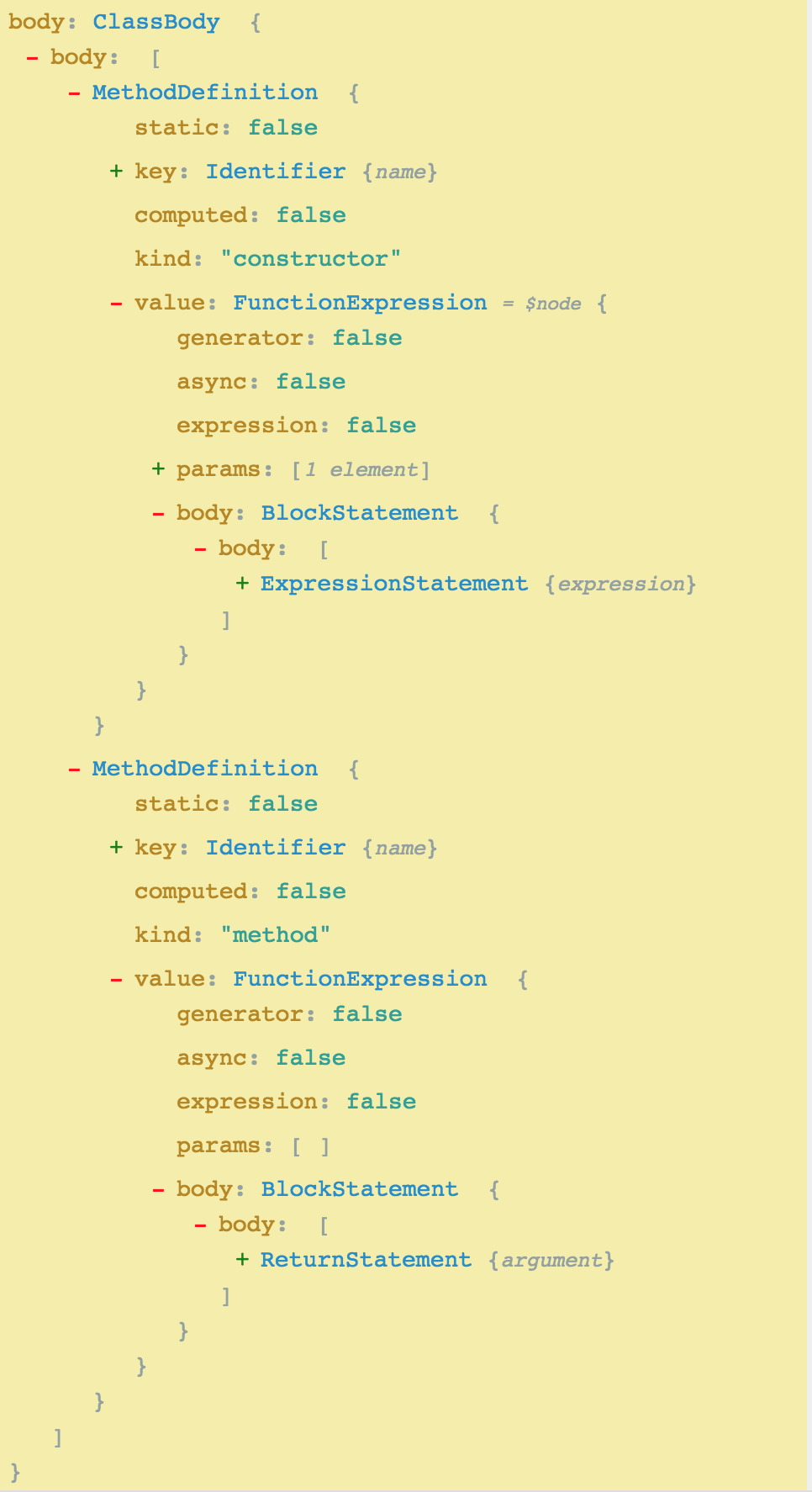
}

getName() {

**return** **this**.name;

}

}



es5

**function** **Person**(name) {

**this**.name=name;

}

Person.prototype.getName=**function** () {

**return** **this**.name;

}

实现

**let** babel = require('@babel/core');**let** t=require('babel-types');**let** source=`

class Person {

constructor(name) {

this.name=name;

}

getName() {

return this.name;

}

}

`;**let** ClassPlugin={

visitor: {

ClassDeclaration(path) {

**let** node=path.node;

**let** id=node.id;

**let** constructorFunction = t.functionDeclaration(id,[],t.blockStatement([]),false,false);

**let** methods=node.body.body;

**let** functions = [];

methods.forEach(method => {

**if** (method.kind == 'constructor') {

constructorFunction = t.functionDeclaration(id,method.params,method.body,false,false);

functions.push(constructorFunction);

} **else** {

**let** memberObj=t.memberExpression(t.memberExpression(id,t.identifier('prototype')),method.key);

**let** memberFunction = t.functionExpression(id,method.params,method.body,false,false);

**let** assignment = t.assignmentExpression('=',memberObj,memberFunction);

functions.push(assignment);

}

});

**if** (functions.length ==0) {

path.replaceWith(constructorFunction);

}**else** **if** (functions.length ==1) {

path.replaceWith(functions[0]);

} **else** {

path.replaceWithMultiple(functions);

}

}

}

}

**const** result = babel.transform(source,{

plugins:[

ClassPlugin

]

});console.log(result.code);

## **6. webpack babel插件**

**var** babel = require("@babel/core");**let** { transform } = require("@babel/core");

### **6.1 实现按需加载**

* [lodashjs](https://www.lodashjs.com/docs/4.17.5.html" \l "concat)
* [babel-core](https://babeljs.io/docs/en/babel-core)
* [babel-plugin-import](https://www.npmjs.com/package/babel-plugin-import)

**import** { flatten,concat } **from** "lodash"



转换为

**import** flatten **from** "lodash/flatten";**import** concat **from** "lodash/flatten";



### **6.2 webpack配置**

cnpm i webpack webpack-cli -D

**const** path=require('path');module.exports={

mode:'development',

entry: './src/index.js',

output: {

path: path.resolve('dist'),

filename:'bundle.js'

},

module: {

rules: [

{

test: /\.js$/,

use: {

loader: 'babel-loader',

options: {

plugins:[['import',{library:'lodash'}]]

}

}

}

]

}

}

编译顺序为首先plugins从左往右,然后presets从右往左

### **6.3 babel插件**

* babel-plugin-import.js放置在node\_modules目录下

**let** babel = require('@babel/core');**let** types = require('babel-types');**const** visitor = {

ImportDeclaration:{

enter(path,state={opts}){

**const** specifiers = path.node.specifiers;

**const** source = path.node.source;

**if**(state.opts.library == source.value && !types.isImportDefaultSpecifier(specifiers[0])){

**const** declarations = specifiers.map((specifier,index)=>{

**return** types.ImportDeclaration(

[types.importDefaultSpecifier(specifier.local)],

types.stringLiteral(`${source.value}/${specifier.local.name}`)

)

});

path.replaceWithMultiple(declarations);

}

}

}

}module.exports = **function**(babel){

**return** {

visitor

}

}

## **9. AST**

### **9.1 解析过程**

AST整个解析过程分为两个步骤

* 分词：将整个代码字符串分割成语法单元数组
* 语法分析：建立分析语法单元之间的关系

### **9.2 语法单元**

Javascript 代码中的语法单元主要包括以下这么几种

* 关键字：const、let、var等
* 标识符：可能是一个变量，也可能是 if、else 这些关键字，又或者是 true、false 这些常量
* 运算符
* 数字
* 空格
* 注释

### **9.3 词法分析**

**let** jsx = `let element=<h1>hello</h1>`;

**function** **lexical**(code) {

**const** tokens=[];

**for** (**let** i=0;i<code.length;i++){

**let** char=code.charAt(i);

**if** (char == '=') {

tokens.push({

type: 'operator',

value:char

});

}

**if** (char=='<') {

**const** token={

type: 'JSXElement',

value:char

}

tokens.push(token);

**let** isClose = false;

**for** (i++;i<code.length;i++){

char=code.charAt(i);

token.value+=char;

**if** (char=='>') {

**if** (isClose) {

**break**;

} **else** {

isClose=true;

}

}

}

**continue**;

}

**if** (/[a-zA-Z\$\\_]/.test(char)) {

**const** token={

type: 'Identifier',

value:char

}

tokens.push(token);

**for** (i++;i<code.length;i++){

char=code.charAt(i);

**if** (/[a-zA-Z\$\\_]/.test(char)) {

token.value+=char;

} **else** {

i--;

**break**;

}

}

**continue**;

}

**if** (/\s/.test(char)) {

**const** token={

type: 'whitespace',

value:char

}

tokens.push(token);

**for** (i++;i<code.length;i++){

char=code.charAt[i];

**if** (/\s/.test(char)) {

token.value+=char;

} **else** {

i--;

**break**;

}

}

**continue**;

}

}

**return** tokens;

}**let** result=lexical(jsx);console.log(result);

[

{ type: 'Identifier', value: 'let' },

{ type: 'whitespace', value: ' ' },

{ type: 'Identifier', value: 'element' },

{ type: 'operator', value: '=' },

{ type: 'JSXElement', value: '<h1>hello</h1>' }

]

### **9.4 语法分析**

* 语义分析则是将得到的词汇进行一个立体的组合，确定词语之间的关系
* 简单来说语法分析是对语句和表达式识别，这是个递归过程

// babylon7 https://astexplorer.net/// babylon7 https://astexplorer.net/**function** **parse**(tokens) {

**const** ast={

type: 'Program',

body: [],

sourceType:'script'

}

**let** i=0;//标示当前位置

**let** currentToken;//当前的符号

**while** ((currentToken = tokens[i])) {

**if** (currentToken.type == 'Identifier' && (currentToken.value == 'let'||currentToken.value == 'var')) {

**const** VariableDeclaration={

type: 'VariableDeclaration',

declarations:[]

}

i+=2;

currentToken=tokens[i];

**let** VariableDeclarator = {

type: 'VariableDeclarator',

id: {

type: 'Identifier',

name:currentToken.value

}

};

VariableDeclaration.declarations.push(VariableDeclarator);

i+=2;

currentToken=tokens[i];

**if** (currentToken.type=='JSXElement') {

**let** value=currentToken.value;

**let** [,type,children]=value.match(/([^<]+?)>([^<]+)<\/\1>/);

VariableDeclarator.init={

type: 'JSXElement',

openingElement:{

type:'JSXOpeningElement',

name:{

type:'JSXIdentifier',

name:'h1'

}

},

closingElement:{

type:'JSXClosingElement',

name:{

type:'JSXIdentifier',

name:'h1'

}

},

name: type,

children:[

{

type:'JSXText',

value:'hello'

}

]

}

} **else** {

VariableDeclarator.init={

type: 'Literal',

value:currentToken.value

}

}

ast.body.push(VariableDeclaration);

}

i++;

}

**return** ast;

}

**let** tokens=[

{type: 'Identifier',value: 'let'},

{type: 'whitespace',value: ' '},

{type: 'Identifier',value: 'element'},

{type: 'operator',value: '='},

{type: 'JSXElement',value: '<h1>hello</h1>'}

];**let** result = parse(tokens);console.log(result);console.log(JSON.stringify(result));

{

"type": "Program",

"body": [{

"type": "VariableDeclaration",

"declarations": [{

"type": "VariableDeclarator",

"id": {

"type": "Identifier",

"name": "element"

},

"init": {

"type": "JSXElement",

"openingElement": {

"type": "JSXOpeningElement",

"name": {

"type": "JSXIdentifier",

"name": "h1"

}

},

"closingElement": {

"type": "JSXClosingElement",

"name": {

"type": "JSXIdentifier",

"name": "h1"

}

},

"name": "h1",

"children": [{

"type": "JSXText",

"value": "hello"

}]

}

}]

}],

"sourceType": "script"

}