# 如何用JS实现你的个性化语言？(AST & PARSER)

## 1. 写一门语言需要考虑哪些模块?

a. 流式读取字符

b. 将连续的字符串标计化（tokenization），即将字符转化为Token。其主要的形式和分类如下：

JSON

```
{ type: "punc", value: "(" }           // 符号: (), ,, ; {}等
{ type: "num", value: 5 }             // 数字
{ type: "str", value: "Hello World!" } // 字符串
{ type: "kw", value: "function" }        // 关键字
{ type: "var", value: "a" }             // 变量名
{ type: "op", value: "!=" }             // 操作符
```

c. 将Token数组转化为抽象语法树(AST)

d. 将AST转化为可执行代码

i. 基于JS执行

ii. 转化为JS执行

其中 a 和 b 两个步骤合起来做的事情专业术语叫做词法分析，c叫做语法分析。

## 2. 实现极其简单的中文语言

下面我们将写一门属于自己简单的中文语言。随便起个名字比如：“中语言” 吧。

### 2.1 定义语言的功能

首先我们需要定义“中语言”的功能，我们希望它能支持加减乘除的函数以及变量定义。

举几个“中语言”代码例子：（这里的阿拉伯数字就不转化为汉语了，不是重点）

JavaScript

```
const chineseAdd = `相加 为 函数（甲，乙） 甲 加 乙；打印（相加（3，4））`;
const chineseVarAdd = `定义（甲 为 2，乙 为 甲 加 7，丙 为 甲 加 乙）打印（甲 加 乙 加 丙）`;
const chineseIf = `如果（2 减 1 为 1）「打印（666）」`
```

根据第一节所总结的模块进行一一讲解，接下来就是流式读取字符。

## 2.2 流式读取字符

### 2.2.1 规则

a. 从左往右，以字符为单位读取。
b. 以空字符为结尾。
c. 为了精确错误代码地址，需要以遇到\n为标记记录行数和列数。

### 2.2.2 代码

JavaScript

```javascript
// 字符流分析器
function InputStream(input) {
    let pos = 0, col = 0, line = 1;
    return {
        // 移动到下一位
        next,
        // 返回下一位
        peek,
        // 是否到末尾
        eof,
        // 解析报错
        croak
    };
    function next() {
        const char = input.charAt(pos++);
        if (char === '\n') {
            line++;
            col = 0;
        } else {
            col++;
        }
        return char;
    }

    function peek() {
        return input.charAt(pos);
    }

    function eof() {
        return peek() === '';
    }

    function croak(msg) {
        throw new Error(`${msg} (${line}:${col})`);
    }
}
module.exports = InputStream;
```

## 2.2.3 总结

字符流分析器不是将字符串直接转化为另外的数据结构，而只是返回了操作字符串的几个方法，只有在真正需要操作字符串的时候才会移动当前“指针”。

## 2.3 将字符串转换为Token

### 2.3.1 规则

a. Token的生成顺序依赖字符流的读取顺序
b. 定义“中语言” token 分类
c. 通过正则表达式根据读取到的连续子字符串创建相应的token

JSON

```
{ type: "punc", value: "(" }          // 符号: 「」，，，；  () 等
{ type: "num", value: 5 }              // 数字
{ type: "str", value: "你好" }         // 字符串
{ type: "kw", value: "函数" }          // 关键字
{ type: "var", value: "甲" }           // 变量名
{ type: "op", value: "加" }            // 操作符
```

### 2.3.2 代码

TokenStream返回的也是一批操作，其并没有改变字符串的数据结构，但是它在每次操作后会生成当前“指针”所代表的子字符串的token数据。从结果来看，在使用时，我们可以想象其会生成一个token 数组。

JavaScript

```
// 词法解析器
function TokenStream(input) {
    let current = null;
    // 关键词数组
    const keywords = " 如果 则 否则 函数 真 假 定义 ";
    return {
        // 移动到下一个token 词组
        next,
        // 获取到下个词组
        peek,
        // 是否到了尾部
        eof,
        // 报错
        croak: input.croak
    };
```

```
    function peek() {
        return current || (current = read_next());
    }
    function next() {
        let tok = current;
        current = null;
        return tok || read_next();
    }
    function eof() {
        return peek() === null;
    }
    // 读取下一个词组
    function read_next() {
        read_while(is_whitespace);
        if (input.eof()) return null;
        const char = input.peek();
        if (char === '#') {
            skip_comment();
            return read_next();
        }
        if (char === '"') return read_string();
        if (is_digit(char)) return read_number();
        // 操作符
        if (is_op_char(char)) {
            return {
                type: 'op',
                value: read_while(is_op_char)
            }
        }
        if (is_id_start(char)) return read_ident();
        // 符号
        if (is_punc(char)) {
            return {
                type: 'punc',
                value: input.next()
            }
        }

        input.croak('无法解析的字符:' + char);
    }
    function is_keyword(x) {
        return keywords.indexOf(" " + x + " ") >= 0;
    }
    function is_digit(ch) {
        return /[0-9]/i.test(ch);
```

```
    }
    function is_id_start(ch) {
        return /[\u4e00-\u9fa5a-zλ_]/i.test(ch);
    }
    function is_id(ch) {
        return is_id_start(ch) || '?!-<>=123456789'.indexOf(ch) >= 0;
    }
    function is_op_char(ch) {
        return '加减乘除为并或等于'.indexOf(ch) >= 0;
    }
    function is_punc(ch) {
        return ", :  () 「」 【】 ; ".indexOf(ch) >= 0;
    }
    function is_whitespace(ch) {
        return " \t\n".indexOf(ch) >= 0;
    }
    // 连续读取属于predicate类型的字符
    function read_while(predicate) {
        let str = '';
        while (!input.eof() && predicate(input.peek())) {
            str += input.next();
        }
        return str;
    }

    function read_number() {
        let has_dot = false;
        let number = read_while(function (ch) {
            if (ch === '.') {
                if (has_dot) {
                    return false;
                }
                has_dot = true;
                return true;
            }
            return is_digit(ch);
        });
        return { type: 'num', value: parseFloat(number) };
    }

    function read_ident() {
        const id = read_while(is_id);
        return {
            type: is_keyword(id) ? 'kw' : 'var',
```

```
            value: id
        }
    }
    // 读取带反编译\\的字符串
    function read_escaped(end) {
        var escaped = false, str = '';
        input.next();
        while (!input.eof()) {
            let ch = input.next();
            if (escaped) {
                str += ch;
                escaped = false;
            } else if (ch === '\\') {
                escaped = true;
            } else if (ch === end) {
                break;
            } else {
                str += ch;
            }
        }
        return str;
    }

    function read_string() {
        return { type: 'str', value: read_escaped('"') };
    }

    function skip_comment() {
        read_while(function (ch) { return ch !== '\n' });
        input.next();
    }
}
module.exports = TokenStream;
```

### 2.3.3 总结

词法分析器会根据我们所定义的规则去生成token，如果我们写的代码不符合词法，它会提示出现无法解析的字符。

## 2.4 将Token转化为AST

### 2.4.1 规则

- 我们需要给AST定义相应的数据结构
  - 首先整个AST是树形结构
  - 每个节点可以是不同的类型，包含了不同的属性
  - 叶子节点一定是最终有具体值的节点类型，不会再嵌套AST

JSON

```
num { type: "num", value: NUMBER }
str { type: "str", value: STRING }
bool { type: "bool", value: 真 or 假 }
var { type: "var", value: NAME }
// 函数，其body是另一颗AST树
func { type: "func", vars: [ NAME... ], body: AST }
call { type: "call", func: AST, args: [ AST... ] }
if { type: "if", cond: AST, then: AST, else: AST }
assign { type: "assign", operator: "为", left: AST, right: AST }
binary { type: "binary", operator: OPERATOR, left: AST, right: AST }
prog { type: "prog", prog: [ AST... ] }
let { type: "let", vars: [ VARS... ], body: AST }
```

  - 不同的操作符有不同的权重

JavaScript

```
const PRECEDENCE = {
    "为": 1,
    "等于": 1.5,
    "或": 2,
    "并": 3,
    "加": 10, "减": 10,
    "乘": 20, "除": 20, "模": 20,
};
```

## 2.4.2 代码

JavaScript

```
const FALSE = { type: "bool", value: false };
// 操作符号权重

const PRECEDENCE = {
    "为": 1,
    "等于": 1.5,
```

```
    "或": 2,
    "并": 3,
    "加": 10, "减": 10,
    "乘": 20, "除": 20, "模": 20,
};

// 解析token
function parse(input) {
    return parse_toplevel();
    // 解析整个程序 根节点prog
    function parse_toplevel() {
        const prog = [];
        while (!input.eof()) {
            prog.push(parse_expression());
            // 表达式以分号分割
            if (!input.eof()) skip_punc("; ");
        }
        return { type: 'prog', prog }
    }
    // 解析表达式,
    // 表达式可能是调用，可能是定义。maybe_call
    // 表示式可能是多个子表达式的集合。maybe_binary
    function parse_expression() {
        return maybe_call(function () {
            return maybe_binary(parse_atom(), 0);
        })
    }
    // 解析调用表达式
    function maybe_call(exp) {
        const expr = exp();
        // 如果下一个token是（
        return is_punc(' (') ? parse_call(expr) : expr;
    }

    // 构建调用表达式类型
    function parse_call(func) {
        return {
            type: 'call',
            func,
            args: delimited(' (', ') ', ', ', parse_expression)
        }
    }
    // 解析子表达式后带操作符的表达式
    function maybe_binary(left, my_prec) {
```

```
51          const tok = is_op();
52
53          if (tok) {
54              const next_prec = PRECEDENCE[tok.value];
55              if (next_prec > my_prec) {
56                  input.next();
57                  return maybe_binary({
58                      type: tok.value === '为' ? 'assign' : 'binary',
59                      operator: tok.value,
60                      left,
61                      right: maybe_binary(parse_atom(), next_prec)
62                  }, my_prec)
63              }
64          }
65          return left;
66      }
67
68
69      // 分解原子表达式
70      function parse_atom() {
71          return maybe_call(function () {
72              if (is_punc(' (')) {
73                  input.next();
74                  const exp = parse_expression();
75                  skip_punc(') ');
76                  return exp;
77              }
78              if (is_punc(' 「')) {
79                  return parse_prog();
80              }
81              if (is_kw('定义')) return parse_let();
82              if (is_kw('如果')) {
83                  return parse_if();
84              }
85              if (is_kw('真') || is_kw('假')) return parse_bool();
86              if (is_kw('函数')) {
87                  input.next();
88                  return parse_func();
89              }
90              const tok = input.next();
91
92              if (tok.type === 'var' || tok.type === 'num' || tok.type === 'str')
    {
93                  return tok;
94              }
```

```
            unexpected();
        })
    }

    // 分解块
    function parse_prog() {
        const prog = delimited('「', '」', '; ', parse_expression);
        if (prog.length === 0) return FALSE;
        if (prog.length === 1) return prog[0];
        return { type: 'prog', prog };
    }

    // 分解if语句
    function parse_if() {
        skip_kw('如果');
        const cond = parse_expression();
        if (!is_punc('「')) skip_kw('则');
        const then = parse_expression();
        const ret = {
            type: 'if',
            cond,
            then
        }
        if (is_kw('否则')) {
            input.next();
            ret.else = parse_expression();
        }
        return ret;
    }

    // 解析bool
    function parse_bool() {
        return {
            type: 'bool',
            value: input.next().value === '真'
        }
    }

    // 解析函数
    function parse_func() {
        return {
            type: 'func',
            name: input.peek().type === 'var' ? input.next().value : null,
            vars: delimited(' (', ') ', ', ', parse_varname),
            body: parse_expression()
```

```
            body: parse_expression()
        }
    }

    // 解析参数
    function parse_varname() {
        const name = input.next();
        if (name.type !== 'var') input.croak('此处期望变量名');
        return name.value;
    }

    // 解析定义
    function parse_let() {
        skip_kw('定义');
        if (input.peek().type === 'var') {
            const name = input.next().value;
            const defs = delimited(" (", ") ", ", ", parse_vardef);
            return {
                type: 'call',
                func: {
                    type: 'func',
                    name,
                    vars: defs.map((def) => def.name),
                    body: parse_expression()
                },
                args: defs.map((def) => def.def || FALSE)
            }
        }
        return {
            type: 'let',
            vars: delimited(' (', ') ', ', ', parse_vardef),
            body: parse_expression()
        }
    }

    // 解析 定义 的变量列表
    function parse_vardef() {
        let name = parse_varname(), def;
        if (is_op('为')) {
            input.next();
            def = parse_expression();
        }
        return { name, def };
    }

```

```
    // 以start开始，end 结尾，separator分割，被parser后的连续token 的AST
    function delimited(start, end, separator, parser) {
        const a = [];
        let first = true;
        // 删除开始的符号
        skip_punc(start);
        // token 数组里还有值
        while (!input.eof()) {
            // 碰到关闭符号
            if (is_punc(end)) break;
            // 开始符号后面的第一个token不会是分隔符
            if (first) {
                first = false;
            } else {
                skip_punc(separator);
            }
            if (is_punc(end)) {
                break;
            }
            // 将不是符号的token解析后塞入结果
            a.push(parser())
        }
        skip_punc(end);
        return a;
    }


    function is_punc(ch) {
        const tok = input.peek();
        return tok && tok.type === 'punc' && (!ch || tok.value === ch) && tok;
    }

    function skip_punc(ch) {
        if (is_punc(ch)) {
            input.next();
        } else {
            input.croak(`期望的符号是 "${ch}"`)
        }
    }

    function is_op(op) {
        const tok = input.peek();
        return tok && tok.type === 'op' && (!op || tok.value === op) && tok;
    }

```

```
    function is_kw(kw) {
        const tok = input.peek();
        return tok && tok.type === 'kw' && (!kw || tok.value === kw) && tok;
    }

    function skip_kw(kw) {
        if (is_kw(kw)) { input.next(); }
        else input.croak(`期望的关键字是:${kw}`)
    }
    function skip_op(op) {
        if (is_op(op)) {
            input.next();
        } else {
            input.croak(`期望的操作是${op}`)
        }
    }
    function unexpected() {
        input.croak(`不被期望出现的token` + JSON.stringify(input))
    }
}

module.exports = parse
```

### 2.4.3 总结

- 我们先定义语法规则，再根据它，在转化Token为AST时忽略一些符号Token、解析有内容的Token生成AST的叶子节点，组合成一棵AST子树等等最终生成整个AST树。
- 每次让Token流“指针”向后移动的时机在不同类型的AST节点的解析器中是不一样的，依赖于语法规则以及当前指针的位置。
- 转化为AST是静态的语法分析。在日常开发中，我们在vscode中编写代码，在文件更新后，就会触发语法解析器parse，如果写错了语法立刻就会有错误提示。


- 对于上面2.1的“中语言”例子，我们会生成什么AST呢?

JavaScript

```javascript
const chineseAdd = `相加 为 函数（甲，乙） 甲 加 乙；打印（相加（3，4））`;

const chineseVarAdd = `定义（甲 为 2，乙 为 甲 加 7，丙 为 甲 加 乙）打印（甲 加 乙 加 丙）`;
const chineseIf = `如果（2 减 1 等于 1）「打印（666）」
```

JSON

```json
// chineseAdd
{
    "type": "prog",
    "prog": [
        {
            "type": "assign",
            "operator": "为",
            "left": {
                "type": "var",
                "value": "相加"
            },
            "right": {
                "type": "func",
                "name": null,
                "vars": [
                    "甲",
                    "乙"
                ],
                "body": {
                    "type": "binary",
                    "operator": "加",
                    "left": {
                        "type": "var",
                        "value": "甲"
                    },
                    "right": {
                        "type": "var",
                        "value": "乙"
                    }
                }
            }
        },
        {
            "type": "call",
```

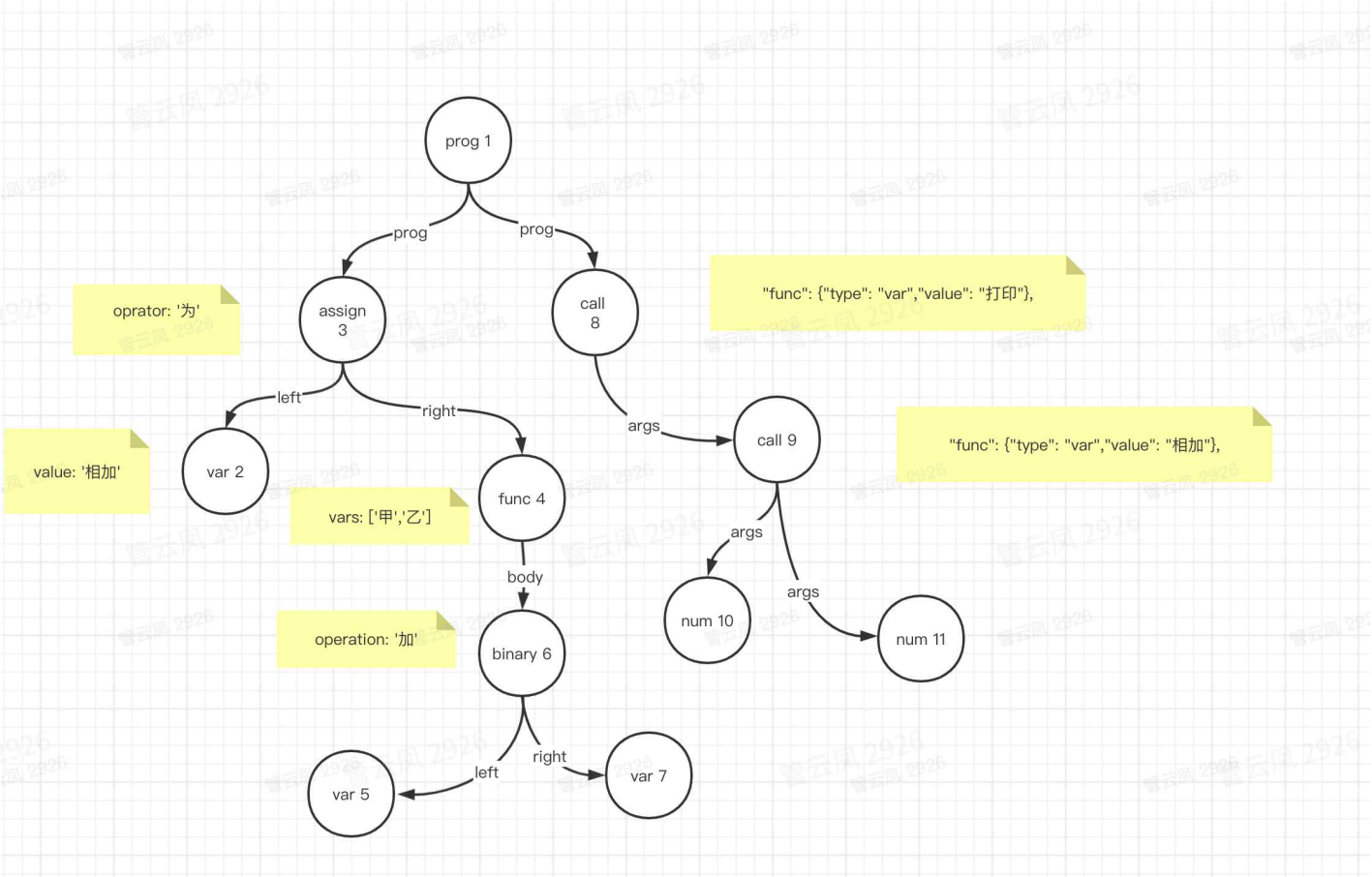
```
            "func": {
                "type": "var",
                "value": "打印"
            },
            "args": [
                {
                    "type": "call",
                    "func": {
                        "type": "var",
                        "value": "相加"
                    },
                    "args": [
                        {
                            "type": "num",
                            "value": 3
                        },
                        {
                            "type": "num",
                            "value": 4
                        }
                    ]
                }
            ]
        }
    ]
}

// chineseVarAdd
{
    "type": "prog",
    "prog": [
        {
            "type": "let",
            "vars": [
                {
                    "name": "甲",
                    "def": {
                        "type": "num",
                        "value": 2
                    }
                },
                {
                    "name": "乙",
                    "def": {
```

```
                    "type": "binary",
                    "operator": "加",
                    "left": {
                        "type": "var",
                        "value": "甲"
                    },
                    "right": {
                        "type": "num",
                        "value": 7
                    }
                }
            },
            {
                "name": "丙",
                "def": {
                    "type": "binary",
                    "operator": "加",
                    "left": {
                        "type": "var",
                        "value": "甲"
                    },
                    "right": {
                        "type": "var",
                        "value": "乙"
                    }
                }
            }
        ],
        "body": {
            "type": "call",
            "func": {
                "type": "var",
                "value": "打印"
            },
            "args": [
                {
                    "type": "binary",
                    "operator": "加",
                    "left": {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "var",
                            "value": "甲"
                        },
```

```
                            "right": {
                                "type": "var",
                                "value": "乙"
                            }
                        },
                        "right": {
                            "type": "var",
                            "value": "丙"
                        }
                    }
                ]
            }
        }
    ]
}
```

- 我没有写chineseIf的AST，既然我们知道了生成 AST的逻辑，那我认为现在应该能看着“中语言”代码就可以在脑中生成对应的AST了。根据下面这张图的构造顺序(chineseAdd)，我们可以总结一下AST的生成规律：



- 1. 从左往右读取的流式token，token的顺序是节点解析的顺序
- 2. 利用了递归来构造ast的子树，每个节点根据其type具有特殊的属性

那么chineseIf 可以这么解析

JavaScript

```
const chineseIf = `如果（2 减 1 为 1）「打印（666）」

// 生成的token，简单描述如下
 kw: '如果'
 punc:'（',
 num: 2,
 op: 减,
 num: 1,
 op: '为',
 num: 1 ,
 punc: '）',
 punc:'「',
 var: '打印',
 punc:'（',
 num: 666 ,
 punc: '）'
 punc:'」',

// 思考一下
```

```
// 生成的AST
{
    "type": "prog",
    "prog": [
        {
            "type": "if",
            "cond": {
                "type": "binary",
                "operator": "等于",
                "left": {
                    "type": "binary",
                    "operator": "减",
                    "left": {
                        "type": "num",
                        "value": 2
                    },
                    "right": {
                        "type": "num",
                        "value": 1
                    }
                },
                "right": {
                    "type": "num",
                    "value": 1
                }
            },
            "then": {
                "type": "call",
                "func": {
                    "type": "var",
                    "value": "打印"
                },
                "args": [
                    {
                        "type": "num",
                        "value": 666
                    }
                ]
            }
        }
    ]
}
```

## 2.5 将AST表示的数据结构用JS运行

对于简单的操作符号，我们可以直接根据节点的type，value，operator等属性进行JS代码转化操作。但是对于定义、函数、函数参数等复杂一点的节点需要有作用域的概念，否则如果所有的参数都挂在一个作用域下，就会导致覆盖、参数混乱的问题。

### 2.5.1 定义作用域类

JavaScript

```
function Environment(parent) {
    this.vars = Object.create(parent ? parent.vars : null);
    this.parent = parent;
}

Environment.prototype = {
    extend: function () {
        // 返回复制当前作用域的变量、和当前作用域为父作用域的作用域
        return new Environment(this);
    },
    lookup: function (name) {
    // 在作用域链中查找
        let scope = this;
        while (scope) {
            if (Object.prototype.hasOwnProperty.call(scope.vars, name)) {
                return scope;
            }
            scope = scope.parent;
        }
    },
    // 只获取当前作用域变量
    get: function (name) {
        if (name in this.vars) {
            return this.vars[name];
        }
        throw new Error(`未定义的变量: ${name}`)
    },
    set: function (name, value) {
    // 如果在作用域链中已经有这个变量，则覆盖
        const scope = this.lookup(name);
        // 不允许在嵌套环境内定义全局变量
        if (!scope && this.parent) {
            throw new Error(`未定义的变量: ${name}`)
        }
```

```
        return (scope || this).vars[name] = value;
    },
    // 只定义为当前作用域变量
    def: function (name, value) {
        return this.vars[name] = value
    }

}


module.exports = Environment;
```

## 2.5.2 执行JS代码

- 在evaluate函数中需要传入当前需要执行的表达式和作用域
- 如果表达式的类型可以获取到具体的值就直接返回
- 如果表达式的值是比较复杂的类型就进行分解再递归执行

JavaScript

```
function evaluate(exp, env) {
    switch (exp.type) {
        case "num":
        case "str":
        case "bool":
            return exp.value;
            // 从作用域取得变量
        case 'var':
            return env.get(exp.value);
            // 将作用域内左边的变量名设置为右边的表达式的值
        case 'assign':
            return env.set(exp.left.value, evaluate(exp.right, env));
            // 将左边的表达式和右边的表达式用operator操作
        case 'binary':
            return apply_op(exp.operator, evaluate(exp.left, env),
evaluate(exp.right, env));
            // 将func类型的节点转化成一个函数
        case 'func':
            return make_func(env, exp);
            // 给每个变量定义创建一个新的作用域
            // {定义 甲 为 1;  {定义 乙 为 2}}
        case "let":
            exp.vars.forEach(function (v) {
                const scope = env.extend();
```

```
                scope.def(v.name, v.def ? evaluate(v.def, env) : false);
                env = scope;
            });
            return evaluate(exp.body, env);
            // 如果条件成立就执行then，不成立就执行else
        case 'if':
            const cond = evaluate(exp.cond, env);
            if (cond !== false) return evaluate(exp.then, env);
            return exp.else ? evaluate(exp.else, env) : false;
            // 根节点的每个子节点都是一个表达式
        case 'prog':
            let val = false;
            exp.prog.forEach(function (exp) {
                val = evaluate(exp, env);
            });
            return val;
            // 函数调用节点需要执行函数、其参数也可能是表达式，所有也要先递归执行参数
        case "call":
            const func = evaluate(exp.func, env);
            return func.apply(null, exp.args.map(function (arg) {
                return evaluate(arg, env);
            }));
        default:
            throw new Error("无法执行" + exp.type)
    }
}

function apply_op(op, a, b) {
    function num(x) {
        if (typeof x !== 'number') {
            throw new Error('希望获取到数字但是获取到' + x);
        }
        return x;
    }

    function div(x) {
        if (num(x) === 0) {
            throw new Error("除以0")
        }
        return x;
    }

    switch (op) {
        case "加": return num(a) + num(b);
```

```
        case "减": return num(a) - num(b);
        case "乘": return num(a) * num(b);
        case "除": return num(a) / div(b);
        case "模": return num(a) % div(b);
        case "并": return a !== false && b;
        case "或": return a !== false ? a : b;
        case '等于': return a == b;
    }
    throw new Error("不能操作的符号" + op);
}

function make_func(env, exp) {
    // 如果这个函数是有name的，则在当前作用域下定义这个name为函数本身，为了实现自己调用自己
    if (exp.name) {
        env = env.extend();
        env.def(exp.name, func);
    }
    // 新建一个函数作用域，将函数的参数名作为函数的变量
    function func() {
        const names = exp.vars;
        const scope = env.extend();
        for (let i = 0; i < names.length; i++) {
            scope.def(names[i], i < arguments.length ? arguments[i] : false);
        }
        // 最后将新的作用域用于执行函数体
        return evaluate(exp.body, scope);
    }
    return func;
}
```

## 2.6 将AST转换为JS再执行

- 对于简单的基础数据节点类型，我们只需要将数据在JSON.stringify后直接返回
- 对于复杂数据节点类型，需要转化为对应的JS代码
  - 带操作符(type: binary || assign)的表达式转化为(js(变量) 操作 js(变量))
  - 定义(表达式(type: let)转化为立即执行函数，将当前定义变量名作为参数名，后面的变量定义作为函数体；这样就可以满足“let会定义新的作用域和变量”的功能。例如：

JavaScript

```

const chineseVarAdd = `定义（甲 为 2，乙 为 甲 加 7，丙 为 甲 加 乙）打印（甲 加 乙 加
```

```
丙）`;
// 实际上
 let a = 1, b = a + 1, c = a + b;
 // 等于
 (function (a) {
  return (function(b) {
    return (function(c) {
        return c;
      }(a+b))
    }(a+1))
 }(1))

// 解析整个表达式 将let ast将转化为call ast
{
    "type": "call",
    "func": {
        "type": "func",
        "vars": [
            "甲"
        ],
        "body": {
            "type": "let",
            "vars": [
                {
                    "name": "乙",
                    "def": {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "var",
                            "value": "甲"
                        },
                        "right": {
                            "type": "num",
                            "value": 7
                        }
                    }
                },
                {
                    "name": "丙",
                    "def": {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "var",
```

```
                            "value": "甲"
                        },
                        "right": {
                            "type": "var",
                            "value": "乙"
                        }
                    }
                }
            ],
            "body": {
                "type": "call",
                "func": {
                    "type": "var",
                    "value": "打印"
                },
                "args": [
                    {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "binary",
                            "operator": "加",
                            "left": {
                                "type": "var",
                                "value": "甲"
                            },
                            "right": {
                                "type": "var",
                                "value": "乙"
                            }
                        },
                        "right": {
                            "type": "var",
                            "value": "丙"
                        }
                    }
                ]
            }
        }
    },
    "args": [
        {
            "type": "num",
            "value": 2
```

```
        }
    ]
}

// 解析乙
{
    "type": "call",
    "func": {
        "type": "func",
        "vars": [
            "乙"
        ],
        "body": {
            "type": "let",
            "vars": [
                {
                    "name": "丙",
                    "def": {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "var",
                            "value": "甲"
                        },
                        "right": {
                            "type": "var",
                            "value": "乙"
                        }
                    }
                }
            ],
            "body": {
                "type": "call",
                "func": {
                    "type": "var",
                    "value": "打印"
                },
                "args": [
                    {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "binary",
                            "operator": "加",
                            "left": {
```

```
                            "type": "var",
                            "value": "甲"
                        },
                        "right": {
                            "type": "var",
                            "value": "乙"
                        }
                    },
                    "right": {
                        "type": "var",
                        "value": "丙"
                    }
                }
            ]
        }
    }
  },
  "args": [
      {
          "type": "binary",
          "operator": "加",
          "left": {
              "type": "var",
              "value": "甲"
          },
          "right": {
              "type": "num",
              "value": 7
          }
      }
  ]
}

// 解析丙
{
    "type": "call",
    "func": {
        "type": "func",
        "vars": [
            "丙"
        ],
        "body": {
            "type": "let",
            "vars": [],
            "body": {
```

```
            "body": {
                "type": "call",
                "func": {
                    "type": "var",
                    "value": "打印"
                },
                "args": [
                    {
                        "type": "binary",
                        "operator": "加",
                        "left": {
                            "type": "binary",
                            "operator": "加",
                            "left": {
                                "type": "var",
                                "value": "甲"
                            },
                            "right": {
                                "type": "var",
                                "value": "乙"
                            }
                        },
                        "right": {
                            "type": "var",
                            "value": "丙"
                        }
                    }
                ]
            }
        }
    },
    "args": [
        {
            "type": "binary",
            "operator": "加",
            "left": {
                "type": "var",
                "value": "甲"
            },
            "right": {
                "type": "var",
                "value": "乙"
            }
        }
    ]
```

```
}

// 最终结果
(((function (甲) {return ((function (乙) {return ((function (丙) {return 打印(((甲+乙)+丙)) })((甲+乙))) })((甲+7))) })(2)))
```

- 函数表达式(type:func)将会拼接作用域符合"()"和参数符号，以及递归生成body的js代码
- 其他类型表达式都是根据js语法进行字符串拼接

完整代码：

JavaScript

```
const FALSE = { type: "bool", value: false };
function make_js(exp) {
    return js(exp);

    function js(exp) {
        switch (exp.type) {
            case "num":
            case "str":
            case "bool": return js_atom(exp);
            case "var": return js_var(exp);
            case "binary": return js_binary(exp);
            case "assign": return js_assign(exp);
            case "let": return js_let(exp);
            case "func": return js_func(exp);
            case "if": return js_if(exp);
            case "prog": return js_prog(exp);
            case "call": return js_call(exp);
            default:
                throw new Error("Dunno how to make_js for " + JSON.stringify(exp));
        }
    }

    function js_atom(exp) {
        return JSON.stringify(exp.value);
    }

    function make_var(name) {
        return name;
    }

    function js_var(exp) {
```

```
        return make_var(exp.value);
    }

    function js_binary(exp) {
        return "(" + js(exp.left) + transform_operator(exp.operator) + js(exp.right) + ")";
    }

    function transform_operator(operator) {
        switch (operator) {
            case '为': return '=';
            case '加': return '+';
            case "减": return '-';
            case '乘': return '*';
            case '除': return '/';
            case '余': return '%';
            case '等于': return '==';
        }
    }

    function js_assign(exp) {
        return js_binary(exp);
    }

    function js_func(exp) {
        var code = "(function ";
        if (exp.name)
            code += make_var(exp.name);
        code += "(" + exp.vars.map(make_var).join(", ") + ") {";
        code += "return " + js(exp.body) + " })";
        return code;
    }

    function js_let(exp) {
        if (exp.vars.length == 0) {
            return js(exp.body);
        }
        var iife = {
            type: "call",
            func: {
                type: "func",
                vars: [exp.vars[0].name],
                body: {
                    type: "let",
```

```
                    vars: exp.vars.slice(1),
                    body: exp.body
                }
            },
            args: [exp.vars[0].def || FALSE]
        };
        return "(" + js(iife) + ")";
    }

    function js_if(exp) {
        return "("
            + js(exp.cond) + " !== false"
            + " ? " + js(exp.then)
            + " : " + js(exp.else || FALSE)
            + ")";
    }

    function js_prog(exp) {
        return "(" + exp.prog.map(js).join(", ") + ")";
    }
    function js_call(exp) {
        return js(exp.func) + "(" + exp.args.map(js).join(", ") + ")";
    }
}


module.exports = make_js
```

## 2.7 实践一下

*打开vscode，进行演示*

# 3. 基于JS代码执行 - 进阶

## 3.1 更高级的功能 - 回调

回调可以使得语言处理一些异步的操作，比如:

JavaScript

```
fs.readFile("file.txt", "utf8", function CC(error, data){});
// 第三个函数就是回调函数，其会让开发者在fs.readFile执行后得到错误或结果
```

如果需要我们的语言能够支持回调则需要更改evaluate函数里每个节点的返回逻辑。

- 给evaluate函数添加第三个参数，callback，callback的参数为执行结果。
- 对于简单的类型节点，直接将节点值作为callback函数的参数就可以。比如: num、str、bool、var。
- 对于部分复杂的类型节点，需要构造新的evaluate callback函数 来保证用户传入的callback得到的参数是最终答案。比如：assign、binary、if、func。
- 对于带参数的复杂类型节点，需要借用递归函数来构造新的上下文和callback。比如: let、prog、call。

**完整代码：**

JavaScript

```
function evaluate(exp, env, callback) {
    switch (exp.type) {
        case "num":
        case "str":
        case "bool":
            return callback(exp.value);
        case 'var':
            return callback(env.get(exp.value));
        case 'assign':
            if (exp.left.type != "var")
                throw new Error("Cannot assign to " + JSON.stringify(exp.left));
            return evaluate(exp.right, env, function (right) { callback(env.set
(exp.left.value, right)) });
        case 'binary':

            evaluate(exp.left, env, function (left) {
                evaluate(exp.right, env, function (right) {
                    callback(apply_op(exp.operator, left, right));
                });
            });
            return;
        case 'func':
            callback(make_func(env, exp));
            return;
```

```
        case "let":
            // 当还有let变量未定义时用loop继续定义作用域变量
            // 否则执行exp.body，exp.body执行时的scope已经记录好了所有的let变量
            (function loop(env, i) {
                if (i < exp.vars.length) {
                    var v = exp.vars[i];
                    if (v.def) evaluate(v.def, env, function (value) {
                        var scope = env.extend();
                        scope.def(v.name, value);
                        loop(scope, i + 1);
                    }); else {
                        var scope = env.extend();
                        scope.def(v.name, false);
                        loop(scope, i + 1);
                    }
                } else {
                    evaluate(exp.body, env, callback);
                }
            })(env, 0);
            return;
        case 'if':
            evaluate(exp.cond, env, function (cond) {
                if (cond !== false) evaluate(exp.then, env, callback);
                else if (exp.else) evaluate(exp.else, env, callback);
                else callback(false);
            });
            return;
        case 'prog':

            (function loop(last, i) {
                if (i < exp.prog.length) evaluate(exp.prog[i], env, function (va
l) {
                    loop(val, i + 1);
                }); else {
                    callback(last);
                }
            })(false, 0);
            return;
        case "call":

            evaluate(exp.func, env, function (func) {
                (function loop(args, i) {
                    if (i < exp.args.length) evaluate(exp.args[i], env, function
(arg) {
                        args[i + 1] = arg;
```

```
                        loop(args, i + 1);
                    }); else {
                        func.apply(null, args);
                    }
                })([callback], 0);
            });
            return;
        default:
            throw new Error("无法执行" + exp.type)
    }
}

function apply_op(op, a, b) {
    function num(x) {
        if (typeof x !== 'number') {
            throw new Error('希望获取到数字但是获取到' + x);
        }
        return x;
    }

    function div(x) {
        if (num(x) === 0) {
            throw new Error("除以0")
        }
        return x;
    }

    switch (op) {
        case "加": return num(a) + num(b);
        case "减": return num(a) - num(b);
        case "乘": return num(a) * num(b);
        case "除": return num(a) / div(b);
        case "模": return num(a) % div(b);
        case "并": return a !== false && b;
        case "或": return a !== false ? a : b;
        case '等于': return a == b;
    }
    throw new Error("不能操作的符号" + op);
}


function make_func(env, exp) {
    function func(callback) {
        const names = exp.vars;
```

```
        const scope = env.extend();
        for (let i = 0; i < names.length; i++) {
            scope.def(names[i], i + 1 < arguments.length ? arguments[i + 1] : fa
lse);
        }
        evaluate(exp.body, scope, callback);
    };
    return func;
}


module.exports = evaluate;
```

执行结果：

JavaScript

```
const InputStream = require('./InputStream');
const parse = require('./parse');
const TokenStream = require('./TokenStream');
const Environment = require('./environment');
const evaluate = require('./cps-evaluate');
const makeJs = require('./generator')


const code = `相加 为 函数（甲，乙） 甲 加 乙；打印（相加（3，4））`;
const ast = parse(TokenStream(InputStream(code)));
const globalEnv = new Environment();

globalEnv.def("打印", function (callback, txt) {
    console.log(txt);
    callback(txt);
});


evaluate(ast, globalEnv, function (result) {
    console.log("result", result);
});

// 输出
// 7
// result 7
```

## 3.2 递归栈溢出 - 清除栈

现在我们可以给函数定义名字，那是否可以实现函数自己调用自己（递归）呢?

定义一个斐波那契数列函数：

TypeScript

```
const code = `斐波那契数列 为 函数（n） 如果 n 小于 2 则 n 否则 斐波那契数列（n 减 1）加 斐波那契数列（n 减 2）；时间（函数（）打印（斐波那契数列（20））`;
```

执行函数命令行报错：maxinum call stack即执行栈溢出。


如果一个函数调用另一个函数，JS引擎就会新建一个调用帧，并且把它放在调用栈的顶部，只有这个函数运行完并且其内部变量不被其他函数引用，才会从栈顶退出，再从内存里清除。

对于一次 斐波那契数列函数 evaluate调用，其调用栈的记录如下：

Apache

```
fib(20)
## stack
 evaluate(fib(20), k0)
  evaluate(fib, K1)
    evaluate(n 小于 2, K2)
      evaluate(n, K3)
       K3(n)
        evaluate(2, K4)
         K4(2)
          evaluate(fib(19), k5)
           ## 循环上面的步骤
           ## 每个阶段都会引用到之前的fib结果


## 更简单一点的，直接到底
print(1 + 2 * 3);
## stack:
## Kn 代表一个栈帧
evaluate( print(1 + 2 * 3), K0 )
  evaluate( print, K1 )
    K1(print)  # 可以从K1中获取到print，则直接返回
      evaluate( 1 + 2 * 3, K2 )
        evaluate( 2 * 3, K3 )
          evaluate( 2, K4 )
            K4(2)  # 2 是 常量，直接返回
              evaluate( 3, K5 )  # 3 是 常量，直接返回
                K5(3)
                  K3(6)  # 返回 2 * 3
                    evaluate( 1, K6 )  # 1 是 常量，直接返回
                      K6(1)
                        K2(7)  # 返回 1+2*3
                          print(K0, 7)  # 调用print
                            K0(false)  # 结束顶级上下文
```

可以看出，evaluate 函数存在严重的性能问题。那么应该如何防止调用栈溢出呢?

**我们可以记录当前是已经调用过多少次 内部函数（evaluate），如果超过了固定的最大栈值，我们就抛出一个异常，创建一个新的函数，再重新调用，这样就会生成新的调用栈，之前的调用栈会被回收，从而不会出现栈溢出。**

**关键代码：**

JavaScript

```
var STACKLEN;
function GUARD(f, args) {
    if (--STACKLEN < 0) throw new Continuation(f, args);
}

function Continuation(f, args) {
    this.f = f;
    this.args = args;
}

function Execute(f, args) {
    while (true) try {
        STACKLEN = 200;
        return f.apply(null, args);
    } catch (ex) {
        if (ex instanceof Continuation) {
            f = ex.f, args = ex.args;
        }

        else throw ex;
    }
}

```

**完整代码**

JavaScript

```

const InputStream = require('./InputStream');
const parse = require('./parse');
const TokenStream = require('./TokenStream');
const Environment = require('./environment');

function evaluate(exp, env, callback) {
   // 记录调用
    GUARD(evaluate, arguments);
    switch (exp.type) {
        case "num":
        case "str":
        case "bool":
            callback(exp.value);
```

```
            return;

        case "var":
            callback(env.get(exp.value));
            return;

        case "assign":
            if (exp.left.type != "var")
                throw new Error("Cannot assign to " + JSON.stringify(exp.left));
            evaluate(exp.right, env, function CC(right) {
                GUARD(CC, arguments);
                callback(env.set(exp.left.value, right));
            });
            return;

        case "binary":
            evaluate(exp.left, env, function CC(left) {
                GUARD(CC, arguments);
                evaluate(exp.right, env, function CC(right) {
                    GUARD(CC, arguments);
                    callback(apply_op(exp.operator, left, right));
                });
            });
            return;

        case "let":
            (function loop(env, i) {
                GUARD(loop, arguments);
                if (i < exp.vars.length) {
                    var v = exp.vars[i];
                    if (v.def) evaluate(v.def, env, function CC(value) {
                        GUARD(CC, arguments);
                        var scope = env.extend();
                        scope.def(v.name, value);
                        loop(scope, i + 1);
                    }); else {
                        var scope = env.extend();
                        scope.def(v.name, false);
                        loop(scope, i + 1);
                    }
                } else {
                    evaluate(exp.body, env, callback);
                }
            })(env, 0);
```

```
            return;

        case "func":
            callback(make_func(env, exp));
            return;

        case "if":
            evaluate(exp.cond, env, function CC(cond) {
                GUARD(CC, arguments);
                if (cond !== false) evaluate(exp.then, env, callback);
                else if (exp.else) evaluate(exp.else, env, callback);
                else callback(false);
            });
            return;

        case "prog":
            (function loop(last, i) {
                GUARD(loop, arguments);
                if (i < exp.prog.length) evaluate(exp.prog[i], env, function CC(
val) {
                    GUARD(CC, arguments);
                    loop(val, i + 1);
                }); else {
                    callback(last);
                }
            })(false, 0);
            return;

        case "call":
            evaluate(exp.func, env, function CC(func) {
                GUARD(CC, arguments);
                (function loop(args, i) {
                    GUARD(loop, arguments);
                    if (i < exp.args.length) evaluate(exp.args[i], env, function
CC(arg) {
                        GUARD(CC, arguments);
                        args[i + 1] = arg;
                        loop(args, i + 1);
                    }); else {
                        func.apply(null, args);
                    }
                })([callback], 0);
            });
            return;

```

```
        default:
            throw new Error("I don't know how to evaluate " + exp.type);
    }
}
function apply_op(op, a, b) {
    function num(x) {
        if (typeof x !== 'number') {
            throw new Error('希望获取到数字但是获取到' + x);
        }
        return x;
    }

    function div(x) {
        if (num(x) === 0) {
            throw new Error("除以0")
        }
        return x;
    }

    switch (op) {
        case "加": return num(a) + num(b);
        case "减": return num(a) - num(b);
        case "乘": return num(a) * num(b);
        case "除": return num(a) / div(b);
        case "模": return num(a) % div(b);
        case "并": return a !== false && b;
        case "或": return a !== false ? a : b;
        case '等于': return a == b;
        case "小于": return a < b;
    }
    throw new Error("不能操作的符号" + op);
}
function make_func(env, exp) {
    if (exp.name) {
        env = env.extend();
        env.def(exp.name, func);
    }
    function func(callback) {
        GUARD(func, arguments);
        var names = exp.vars;
        var scope = env.extend();
        for (var i = 0; i < names.length; ++i)
            scope.def(names[i], i + 1 < arguments.length ? arguments[i + 1] : fa
lse);
        evaluate(exp.body, scope, callback);
```

```
        evaluate(exp.body, scope, callback);
    }
    return func;
}

var STACKLEN;
function GUARD(f, args) {
    if (--STACKLEN < 0) throw new Continuation(f, args);
}

function Continuation(f, args) {
    this.f = f;
    this.args = args;
}

function Execute(f, args) {
    while (true) try {
        STACKLEN = 200;
        return f.apply(null, args);
    } catch (ex) {
        if (ex instanceof Continuation) {
            f = ex.f, args = ex.args;
        }

        else throw ex;
    }
}
const globalEnv = new Environment();



const code = `斐波那契数列 为 函数（n） 如果 n 小于 2 则 n 否则 斐波那契数列（n 减 1）加 斐波那契数列（n 减 2）；时间（函数（）打印（斐波那契数列（20））`
const token = TokenStream(InputStream(code));
const ast = parse(token);

globalEnv.def("打印", function (k, val) {
    console.log(val)
    k(false);
});

globalEnv.def("时间", function (k, func) {
    console.time("time");
    func(function (ret) {
        console.timeEnd("time");
```

```
        k(ret);
    });
});


console.log('ast', JSON.stringify(ast));
Execute(evaluate, [ast, globalEnv, function (result) {
    console.log("*** Result:", result);
}]);

```

**运行结果：**

```
guanyunfeng@C02XH711JG5L chinese-language % node cc
6765
time: 148.924ms
*** Result: false
```

## 4. 完整的AST

到目前为止比较完整的AST节点类型可以参考我们都在用的babel-parser ，其包括了：

- Identifier
- PrivateName
- Literals
  - RegExpLiteral
  - NullLiteral
  - StringLiteral
  - BooleanLiteral
  - NumericLiteral
- Programs
- Functions
- Statements
  - ExpressionStatement
  - BlockStatement
  - EmptyStatement

- DebuggerStatement
- WithStatement
- Control flow
  - ReturnStatement
  - LabeledStatement
  - BreakStatement
  - ContinueStatement
- Choice
  - IfStatement
  - SwitchStatement
    - SwitchCase
- Exceptions
  - ThrowStatement
  - TryStatement
    - CatchClause
- Loops
  - WhileStatement
  - DoWhileStatement
  - ForStatement
  - ForInStatement
  - ForOfStatement

- Declarations
  - FunctionDeclaration
  - VariableDeclaration
    - VariableDeclarator
- Misc
  - Decorator
  - Directive
  - DirectiveLiteral
- Expressions
  - Super
  - Import

- ThisExpression
- ArrowFunctionExpression
- YieldExpression
- AwaitExpression
- ArrayExpression
- ObjectExpression
  - ObjectMember
    - ObjectProperty
    - ObjectMethod
- FunctionExpression
- Unary operations
  - UnaryExpression
    - UnaryOperator
  - UpdateExpression
    - UpdateOperator
- Binary operations
  - BinaryExpression
    - BinaryOperator
  - AssignmentExpression
    - AssignmentOperator
  - LogicalExpression
    - LogicalOperator
  - SpreadElement
  - MemberExpression
  - BindExpression
- ConditionalExpression
- CallExpression
- NewExpression
- SequenceExpression
- DoExpression

- Template Literals
  - TemplateLiteral

    - TaggedTemplateExpression
    - TemplateElement
- Patterns
    - ObjectPattern
    - ArrayPattern
    - RestElement
    - AssignmentPattern
- Classes
    - ClassBody
    - ClassMethod
    - ClassPrivateMethod
    - ClassProperty
    - ClassPrivateProperty
    - ClassDeclaration
    - ClassExpression
    - MetaProperty
- Modules
    - ModuleDeclaration
    - ModuleSpecifier
    - Imports
        - ImportDeclaration
        - ImportSpecifier
        - ImportDefaultSpecifier
        - ImportNamespaceSpecifier
    - Exports
        - ExportNamedDeclaration
        - ExportSpecifier
        - ExportDefaultDeclaration
        - ExportAllDeclaration

**更复杂的节点类型意味着在parse中需要对不同type的token进行更复杂处理。**

**具体的处理方式感兴趣的可以参考babel-parser的源码。**

## 5. 参考链接

- How to implement a programming language (tutorial for beginners)
- ES6 入门教程
- babel/babylon

## 6. 分享结束，谢谢大家

- 所有讲解，完整代码参考。