link: null
title: 珠峰架构师成长计划
description: JSX代码
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=220 sentences=589, words=4715

1.需求分析

- 实现JSX语法转成JS语法的编译器
- 需求：将一段 JSX语法的代码生成一个 AST，并支持遍历和修改这个 AST，将 AST重新生成JS语法的代码

JSX代码

```
helloworld
```

JS代码

```
React.createElement("h1", {
  id: "title"
},React.createElement("span", null, "hello"), "world");
```

2.编译器工作流

- 解析(Parsing) **解析** 是将最初原始的代码转换为一种更加抽象的表示(即AST)
- 转换(Transformation) **转换** 将对这个抽象的表示做一些处理,让它能做到编译器期望它做到的事情
- 代码生成(Code Generation) 接收处理之后的代码表示,然后把它转换成新的代码

2.1 解析(Parsing)

- 解析一般来说会分成两个阶段：词法分析(Lexical Analysis)和语法分析(Syntactic Analysis)
    - 词法分析 接收原始代码,然后把它分割成一些被称为 token 的东西，这个过程是在词法分析器(Tokenizer或者Lexer)中完成的
    - Token 是一个数组，由一些代码语句的碎片组成。它们可以是数字、标签、标点符号、运算符或者其它任何东西
    - 语法分析 接收之前生成的 token，把它们转换成一种抽象的表示，这种抽象的表示描述了代码语句中的每一个片段以及它们之间的关系。这被称为中间表示(intermediate representation)或抽象语法树(Abstract Syntax Tree, 缩写为AST)
    - 抽象语法树是一个嵌套程度很深的对象，用一种更容易处理的方式代表了代码本身，也能给我们更多信息

原始 jsx代码

```
helloworld
```

tokens

```
[
    { type: 'Punctuator', value: ' },
    { type: 'JSXIdentifier', value: 'h1' },
    { type: 'JSXIdentifier', value: 'id' },
    { type: 'Punctuator', value: '=' },
    { type: 'String', value: '"title"' },
    { type: 'Punctuator', value: '>' },
    { type: 'Punctuator', value: ' },
    { type: 'JSXIdentifier', value: 'span' },
    { type: 'Punctuator', value: '>' },
    { type: 'JSXText', value: 'hello' },
    { type: 'Punctuator', value: ' },
    { type: 'Punctuator', value: '/' },
    { type: 'JSXIdentifier', value: 'span' },
    { type: 'Punctuator', value: '>' },
    { type: 'JSXText', value: 'world' },
    { type: 'Punctuator', value: ' },
    { type: 'Punctuator', value: '/' },
    { type: 'JSXIdentifier', value: 'h1' },
    { type: 'Punctuator', value: '>' }
  ]
```

抽象语法树(AST)

astexplorer (https://astexplorer.net/)

```
{
    "type": "Program",
    "body": [
        {
            "type": "ExpressionStatement",
            "expression": {
                "type": "JSXElement",
                "openingElement": {
                    "type": "JSXOpeningElement",
                    "name": {
                        "type": "JSXIdentifier",
                        "name": "h1"
                    },
                    "attributes": [
                        {
                            "type": "JSXAttribute",
                            "name": {
                                "type": "JSXIdentifier",
                                "name": "id"
                            },
                            "value": {
                                "type": "Literal",
                                "value": "title"
                            }
                        }
                    ]
                },
                "children": [
                    {
                        "type": "JSXElement",
                        "openingElement": {
                            "type": "JSXOpeningElement",
                            "name": {
                                "type": "JSXIdentifier",
                                "name": "span"
                            }
                            "attributes": []
                        },
                        "children": [
                            {
                                "type": "JSXText",
                                "value": "hello"
                            }
                        ],
                        "closingElement": {
                            "type": "JSXClosingElement",
                            "name": {
                                "type": "JSXIdentifier",
                                "name": "span"
                            }
                        }
                    },
                    {
                        "type": "JSXText",
                        "value": "world"
                    }
                ],
                "closingElement": {
                    "type": "JSXClosingElement",
                    "name": {
                        "type": "JSXIdentifier",
                        "name": "h1"
                    }
                }
            }
        }
    ]
}
```


2.2 遍历(Traversal)

- 为了能处理所有的结点，我们需要遍历它们，使用的是深度优先遍历
- 对于上面的 AST 的遍历流程是这样的

```
let esprima = require('esprima');
let code = `helloworld`;
let estraverse = require('estraverse-fb');
let ast = esprima.parseScript(code, {  tokens: true,jsx: true });
let indent = 0;
function padding(){
    return " ".repeat(indent);
}
estraverse.traverse(ast,{
    enter(node){
        console.log(padding()+node.type+'进入');
        indent+=2;
    },
    leave(node){
        indent-=2;
        console.log(padding()+node.type+'离开');
    }
});
```

```
Program进入
  ExpressionStatement进入
    JSXElement进入
      JSXOpeningElement进入
        JSXIdentifier进入
        JSXIdentifier离开
        JSXAttribute进入
          JSXIdentifier进入
          JSXIdentifier离开
          Literal进入
          Literal离开
        JSXAttribute离开
      JSXOpeningElement离开
      JSXClosingElement进入
        JSXIdentifier进入
        JSXIdentifier离开
      JSXClosingElement离开
      JSXElement进入
        JSXOpeningElement进入
          JSXIdentifier进入
          JSXIdentifier离开
        JSXOpeningElement离开
        JSXClosingElement进入
          JSXIdentifier进入
          JSXIdentifier离开
        JSXClosingElement离开
        JSXText进入
        JSXText离开
      JSXElement离开
      JSXText进入
      JSXText离开
    JSXElement离开
  ExpressionStatement离开
Program离开
```

2.3 转换(Transformation)

- 编译器的下一步就是转换,它只是把 AST 拿过来然后对它做一些修改.它可以在同种语言下操作 AST，也可以把 AST 翻译成全新的语言
- 你或许注意到了我们的 AST 中有很多相似的元素，这些元素都有 type 属性，它们被称为 AST结点。这些结点含有若干属性，可以用于描述 AST 的部分信息
- 当转换 AST 的时候我们可以添加、移动、替代这些结点，也可以根据现有的 AST 生成一个全新的 AST
- 既然我们编译器的目标是把输入的代码转换为一种新的语言，所以我们将会着重于产生一个针对新语言的全新的 AST

2.4 代码生成(Code Generation)

- 编译器的最后一个阶段是代码生成，这个阶段做的事情有时候会和转换(transformation)重叠,但是代码生成最主要的部分还是根据 AST 来输出代码
- 代码生成有几种不同的工作方式，有些编译器将会重用之前生成的 token，有些会创建独立的代码表示，以便于线性地输出代码。但是接下来我们还是着重于使用之前生成好的 AST
- 我们的代码生成器需要知道如何 打印AST 中所有类型的结点，然后它会递归地调用自身，直到所有代码都被打印到一个很长的字符串中

3. 有限状态机

- 每一个状态都是一个机器,每个机器都可以接收输入和计算输出
- 机器本身没有状态,每一个机器会根据输入决定下一个状态

```
let WHITESPACE = /\s/;
let NUMBERS = /[0-9]/;

const Number = 'Number';
const Plus = 'Plus';

let currentToken;
let tokens = [];
function emit(token){
    currentToken = { type: "", value: "" };
    tokens.push(token);
}

function start(char){
    if(NUMBERS.test(char)){
        currentToken = {
            type:Number,
            value:''
        }
        return number(char);
    }
    throw new TypeError('函数名必须是字符 '+ char);
}
function number(char){
    if(NUMBERS.test(char)){
        currentToken.value += char;
        return number;
    }else if(char == "+"){
        emit(currentToken);
        emit({ type: Plus,  value: '+'});
        currentToken = {
            type:Number,
            value:''
        }
        return number;
    }
}
function tokenizer(input){
    let state = start;
    for(let char of input){
        state = state(char);
    }
    emit(currentToken);
}

tokenizer('10+20')
console.log(tokens);
```

```
[
  { type: 'Number', value: '10' },
  { type: 'Plus', value: '+' },
  { type: 'Number', value: '20' }
]
```

4 词法分析器

- 我们只是接收代码组成的字符串，然后把它们分割成 token 组成的数组

4.1 tokenTypes.js

src\tokenTypes.js

```
exports.LeftParentheses = 'LeftParentheses';
exports.RightParentheses = 'RightParentheses';
exports.JSXIdentifier = 'JSXIdentifier';
exports.AttributeKey = 'AttributeKey';
exports.AttributeStringValue = 'AttributeStringValue';
exports.JSXText = 'JSXText';
exports.BackSlash = 'BackSlash';
```

4.2 tokenizer.js

src\tokenizer.js

```
const LETTERS = /[a-z0-9]/;
const tokenTypes = require('./tokenTypes');
let currentToken = {type:'',value:''};
let tokens = [];
function emit(token){
    currentToken = {type:'',value:''};
    tokens.push(token);
}
function start(char){
    if(char === '<'){
        emit({type:tokenTypes.LeftParentheses,value:'<'});
        return foundLeftParentheses;
    }
    throw new Error('第一个字符必须是<');
}

function foundLeftParentheses(char){
    if(LETTERS.test(char)){
        currentToken.type = tokenTypes.JSXIdentifier;
        currentToken.value += char;
        return jsxIdentifier;
    }else if(char === '/'){
        emit({type:tokenTypes.BackSlash,value:'/'});
        return foundLeftParentheses;
    }
    throw new Error('第一个字符必须是字母');
}
function jsxIdentifier(char){
    if(LETTERS.test(char)){
        currentToken.value+=char;
        return jsxIdentifier;
    }else if(char === ' '){
        emit(currentToken);
        return attribute;
    }else if(char == '>'){
        emit(currentToken);
        emit({type:tokenTypes.RightParentheses,value:'>'});
        return foundRightParentheses;
    }
    throw new Error('第一个字符必须是字母');
}
function attribute(char){
    if(LETTERS.test(char)){
        currentToken.type = tokenTypes.AttributeKey;
        currentToken.value += char;
        return attributeKey;
    }
    throw new TypeError('Error');
}
function attributeKey(char){
    if(LETTERS.test(char)){
        currentToken.value += char;
        return attributeKey;
    }else if(char === '='){
        emit(currentToken);
        return attributeValue;
    }
    throw new TypeError('Error');
}
function attributeValue(char){
    if(char === '"'){
        currentToken.type = tokenTypes.AttributeStringValue;
        currentToken.value = '';
        return attributeStringValue;
    }
    throw new TypeError('Error');
}

function attributeStringValue(char){
    if(LETTERS.test(char)){
        currentToken.value+=char;
        return attributeStringValue;
    }else if(char === '"'){
        emit(currentToken);
        return tryLeaveAttribute;
    }
    throw new TypeError('Error');
}
```

```
function tryLeaveAttribute(char){
    if(char === ' '){
        return attribute;
    }else if(char === '>'){
        emit({type:tokenTypes.RightParentheses,value:'>'});
        return foundRightParentheses;
    }
    throw new TypeError('Error');
}
function foundRightParentheses(char){
    if(char === '<'){
        emit({type:tokenTypes.LeftParentheses,value:'<'});
        return foundLeftParentheses;
    }else{
        currentToken.type = tokenTypes.JSXText;
        currentToken.value += char;
        return jsxText;
    }
}
function jsxText(char){
  if(char === '<'){
    emit(currentToken);
    emit({type:tokenTypes.LeftParentheses,value:'<'});
    return foundLeftParentheses;
  }else{
    currentToken.value += char;
    return jsxText;
  }
}
function tokenizer(input){
    let state = start;
    debugger
    for(let char of input){
        if(state) state = state(char);
    }
    return tokens;
}

module.exports = {
    tokenizer
}
```

分词结果

```
[
  { type: 'LeftParentheses', value: '<' },
  { type: 'JSXIdentifier', value: 'h1' },
  { type: 'AttributeKey', value: 'id' },
  { type: 'AttributeStringValue', value: 'title' },
  { type: 'RightParentheses', value: '>' },
  { type: 'LeftParentheses', value: '<' },
  { type: 'JSXIdentifier', value: 'span' },
  { type: 'RightParentheses', value: '>' },
  { type: 'JSXText', value: 'hello' },
  { type: 'LeftParentheses', value: '<' },
  { type: 'BackSlash', value: '/' },
  { type: 'JSXIdentifier', value: 'span' },
  { type: 'RightParentheses', value: '>' },
  { type: 'JSXText', value: 'world' },
  { type: 'LeftParentheses', value: '<' },
  { type: 'BackSlash', value: '/' },
  { type: 'JSXIdentifier', value: 'h1' },
  { type: 'RightParentheses', value: '>' }
]
```

5.语法分析

- 语法分析的原理和递归下降算法（Recursive Descent Parsing）
- 并初步了解上下文无关文法（Context-free Grammar，CFG）

5.1 递归下降算法

- 它的左边是一个非终结符（Non-terminal）
- 右边是它的产生式（Production Rule）
- 在语法解析的过程中，左边会被右边替代。如果替代之后还有非终结符，那么继续这个替代过程，直到最后全部都是终结符（Terminal），也就是 `Token`
- 只有终结符才可以成为 `AST` 的叶子节点。这个过程，也叫做推导（Derivation）过程
- 上级文法嵌套下级文法，上级的算法调用下级的算法。表现在生成 AST 中，上级算法生成上级节点，下级算法生成下级节点。这就是 `&#x4E0B;&#x964D;`的含义

5.2 上下文无关文法

- 上下文无关的意思是，无论在任何情况下，文法的推导规则都是一样的
- 规则分成两级：第一级是加法规则，第二级是乘法规则。把乘法规则作为加法规则的子规则
- 解析形成 AST 时，乘法节点就一定是加法节点的子节点，从而被优先计算
- 加法规则中还 `&#x9012;&#x5F52;`地又引用了加法规则

5.3 算术表达式

算术表达式

```
2+3*4
```

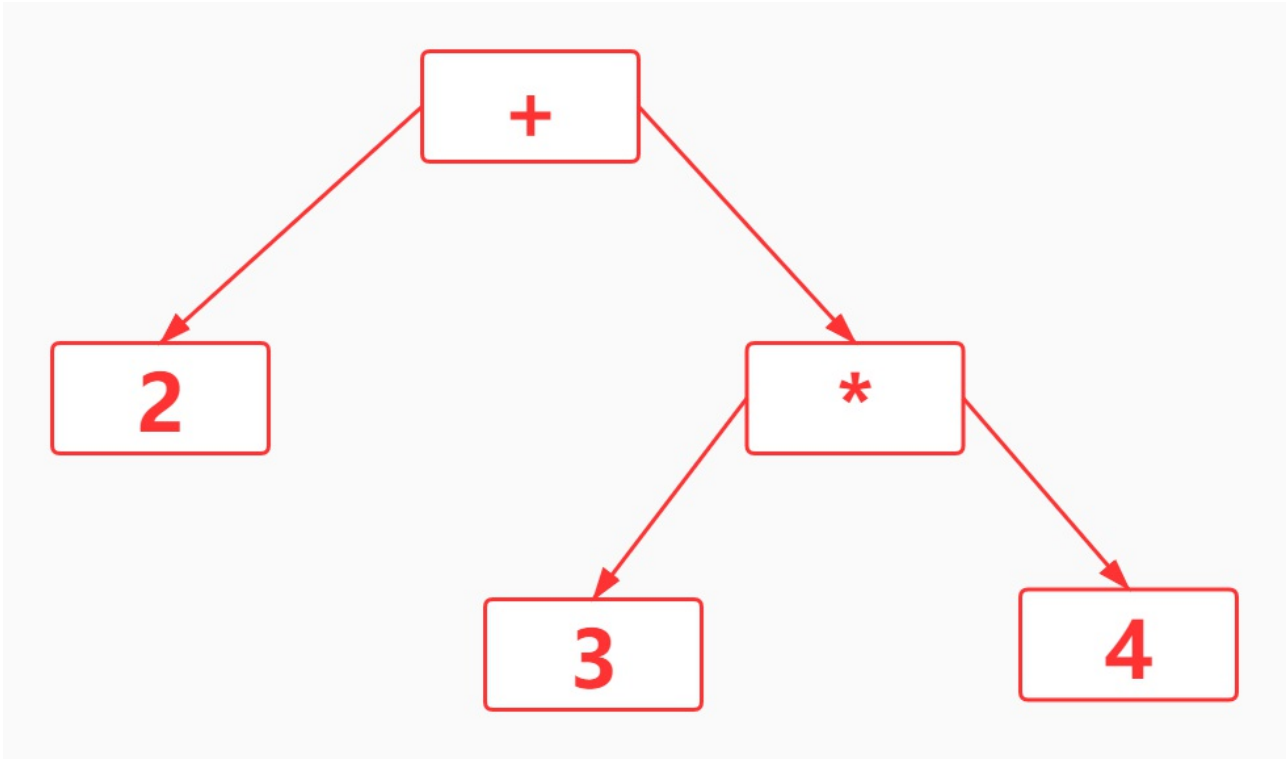
语法规则

```
add ->  multiple|multiple + add
multiple -> NUMBER | NUMBER *  multiple
```

tokens

```
[
  { type: 'NUMBER', value: '2' },
  { type: 'PLUS', value: '+' },
  { type: 'NUMBER', value: '3' },
  { type: 'MULTIPLY', value: '*' },
  { type: 'NUMBER', value: '4' }
]
```


```
- Program {
    type: "Program"
  - body: [
    - ExpressionStatement {
        type: "ExpressionStatement"
      - expression: BinaryExpression {
          type: "BinaryExpression"
          operator: "+"
        - left: Literal = $node {
            type: "Literal"
            value: 2
            raw: "2"
          }
        - right: BinaryExpression {
            type: "BinaryExpression"
            operator: "*"
          - left: Literal {
              type: "Literal"
              value: 3
              raw: "3"
            }
          - right: Literal {
              type: "Literal"
              value: 4
              raw: "4"
            }
          }
        }
      }
    ]
    sourceType: "module"
  }
```



ast

```
{
  "type": "Program",
  "children": [
    {
      "type": "Additive",
      "children": [
        {
          "type": "Numeric",
          "value": "2"
        },
        {
          "type": "Multiplicative",
          "children": [
            {
              "type": "Numeric",
              "value": "3"
            },
            {
              "type": "Numeric",
              "value": "4"
            }
          ]
        }
      ]
    }
  ]
}
```

5.4 实现

5.4.1 index.js

index.js

```
let parse = require('./parse');
let evaluate = require('./evaluate');
let sourceCode = "2+3*4";
let ast = parse(sourceCode);
console.log(JSON.stringify(ast, null, 2));
let result = evaluate(ast);
console.log(result);
```

5.4.2 parse.js

parse.js

```
let tokenize = require('./tokenize');
let toAst = require('./toAst');
function parse(script) {
    let tokens = tokenize(script);
    console.log(tokens);
    let ast = toAst(tokens);
    return ast;
}

module.exports = parse;
```

5.4.3 tokenTypes.js

tokenTypes.js

```
exports.Program = 'Program';
exports.Numeric = 'Numeric';
exports.Additive = 'Additive';
exports.Multiplicative = 'Multiplicative';
```

5.4.4 tokenize.js

tokenize.js

```
let RegExpObject = /([0-9]+)|(\+)|(\*)/g;
let tokenTypes = require('./tokenTypes');
let tokenArray = [tokenTypes.NUMBER,tokenTypes.PLUS,tokenTypes.MULTIPLY];
function* tokenizer(source){
  let result = null;
  while(true){
      result = RegExpObject.exec(source);
      if(!result) break;
      let token = {type:null,value:null};
      let index = result.findIndex((item,index)=>index>0&&!!item);
      token.type = tokenArray[index-1];
      token.value = result[0];
      yield token;
  }
}
function tokenize(script){
    let tokens = [];
    for(let token of tokenizer(script)){
        tokens.push(token);
    }
    return new TokenReader(tokens);
}
class TokenReader{
    constructor(tokens){
        this.tokens = tokens;
        this.pos = 0;
    }
    read() {
        if (this.pos < this.tokens.length) {
            return this.tokens[this.pos++];
        }
        return null;
    }
    peek() {
        if (this.pos < this.tokens.length) {
            return this.tokens[this.pos];
        }
        return null;
    }
    unread() {
        if (this.pos > 0) {
            this.pos--;
        }
    }
    getPosition() {
        return this.pos;
    }
    setPosition(position) {
        if (position >=0 && position < tokens.length){
            this.pos = position;
        }
    }
}
module.exports = tokenize;
```

5.4.5 nodeTypes.js

nodeTypes.js

```
exports.Program = 'Program';
exports.Numeric = 'Numeric';
exports.Additive = 'Additive';
exports.Multiplicative = 'Multiplicative';
```

5.4.6 ASTNode.js

ASTNode.js

```
class ASTNode{
  constructor(type,value){
      this.type = type;
      if(value)this.value = value;
  }
  addChild(child) {
    if(!this.children) this.children=[];
    this.children.push(child);
  }
}

module.exports = ASTNode;
```

5.4.7 toAst.js

toAst.js

```
let ASTNode = require('./ASTNode');
let tokenTypes = require('./tokenTypes');
let nodeTypes = require('./nodeTypes');
function toAst(tokenReader){
    let node = new ASTNode('Program');
    let child = additive(tokenReader);
    if (child != null) {
        node.addChild(child);
    }
    return node;
}

function additive(tokenReader){
    let child1 = multiplicative(tokenReader);
    let node = child1;
    let token = tokenReader.peek();
    if (child1 != null && token != null) {
        if (token.type == tokenTypes.PLUS) {
            token = tokenReader.read();
            let child2 = additive(tokenReader);
            if (child2 != null) {
                node = new ASTNode(nodeTypes.Additive);
                node.addChild(child1);
                node.addChild(child2);
            } else {
                throw new Exception("非法的加法表达式,需要右半部分");
            }
        }
    }
    return node;
}
function multiplicative(tokenReader){
    let child1 = primary(tokenReader);
    let node = child1;

    let token = tokenReader.peek();
    if (child1 != null && token != null) {
        if (token.type == tokenTypes.MULTIPLY) {
            token = tokenReader.read();
            let child2 = primary(tokenReader);
            if (child2 != null) {
                node = new ASTNode(nodeTypes.Multiplicative);
                node.addChild(child1);
                node.addChild(child2);
            } else {
                throw new Exception("非法的乘法表达式,需要右半部分");
            }
        }
    }
    return node;
}

function primary(tokenReader){
    let node = null;
    let token = tokenReader.peek();
    if (token != null) {
        if (token.type == tokenTypes.NUMBER) {
            token = tokenReader.read();
            node = new ASTNode(nodeTypes.Numeric, token.value);
        }
    }
    return node;
}

module.exports = toAst;
```

5.4.8 evaluate.js

evaluate.js

```
let nodeTypes = require('./nodeTypes');

function evaluate(node) {
    let result = 0;
    switch (node.type) {
        case nodeTypes.Program:
            for (let child of node.children) {
                result = evaluate(child);
            }
            break;
        case nodeTypes.Additive:
            result = evaluate(node.children[0]) + evaluate(node.children[1]);
            break;
        case nodeTypes.Multiplicative:
            result = evaluate(node.children[0]) * evaluate(node.children[1]);
            break;
        case nodeTypes.Numeric:
            result = parseFloat(node.value);
            break;
    }
    return result;
}
module.exports = evaluate;
```

5.5 支持减法和除法

5.5.1 index.js

```
-let sourceCode = '2+2*2+4';
+let sourceCode = '6+1-3*4/2';
```

5.5.2 tokenTypes.js

```
+exports.MINUS = 'MINUS';
+exports.MULTIPLY = 'MULTIPLY';//乘号
+exports.DIVIDE = 'DIVIDE';
```

5.5.3 tokenize.js

```
+let RegExpObject  = /([0-9]+)|(\+)|(\-)|(\*)|(\/)/g;
+let tokenNames = [tokenTypes.NUMBER,tokenTypes.PLUS,tokenTypes.MINUS,tokenTypes.MULTIPLY,tokenTypes.DIVIDE];
```

5.5.4 nodeTypes.js

```
+exports.Minus = 'Minus';//加法运算
+exports.Multiplicative = 'Multiplicative';//乘法运算
+exports.Divide = 'Divide';//加法运算
```

5.5.5 toAST.js

```
function additive(tokenReader){
     let child1 = multiple(tokenReader);
     let node = child1;
     let token = tokenReader.peek();//看看一下符号是不是加号
     if(child1 != null && token != null){
+        if(token.type === tokenTypes.PLUS||token.type === tokenTypes.MINUS){//如果后面是加号的话
            token = tokenReader.read();//把+读出来并且消耗掉
            let child2 = additive(tokenReader);
            if(child2 != null){
+                node = new ASTNode(token.type === tokenTypes.PLUS?nodeTypes.Additive:nodeTypes.Minus);
                node.appendChild(child1);
                node.appendChild(child2);
            }
        }
     }
     return node;
}
//multiple -> NUMBER | NUMBER *  multiple
function multiple(tokenReader){
    let child1 = number(tokenReader);//先配置出来NUMBER,但是这个乘法规则并没有匹配结束
    let node = child1; //node=3
    let token = tokenReader.peek();//*
    if(child1 != null && token != null){
+       if(token.type === tokenTypes.MULTIPLY||token.type === tokenTypes.DIVIDE){
         token = tokenReader.read();//读取下一个token  *
         let child2 = multiple(tokenReader);//4
         if(child2 != null){
+            node = new ASTNode(token.type === tokenTypes.MULTIPLY?nodeTypes.Multiplicative:nodeTypes.Divide);
            node.appendChild(child1);
            node.appendChild(child2);
         }
       }
    }
    return node;
}
```

5.5.6 evaluate.js

```
 switch(node.type){
    case nodeTypes.Program:
        for(let child of node.children){
            result = evaluate(child);//child  Additive
        }
        break;
    case nodeTypes.Additive://如果是一个加法节点的话,如何计算结果
        result = evaluate(node.children[0])+evaluate(node.children[1]);
        break;
+    case nodeTypes.Minus://如果是一个加法节点的话,如何计算结果
+        result = evaluate(node.children[0]) - evaluate(node.children[1]);
+        break;
    case nodeTypes.Multiplicative://如果是一个加法节点的话,如何计算结果
        result = evaluate(node.children[0]) * evaluate(node.children[1]);
        break;
+    case nodeTypes.Divide://如果是一个加法节点的话,如何计算结果
+        result = evaluate(node.children[0]) / evaluate(node.children[1]);
+        break;
    case nodeTypes.Numeric:
        result = parseFloat(node.value);
    default:
        break;
  }
```

5.6 支持括号

5.6.1 index.js

```
+ let sourceCode = '(1+2)*3*(2+2)';
```

5.6.2 tokenTypes.js

```
+ exports.LEFT_PARA = 'LEFT_PARA';
+ exports.RIGHT_PARA = 'RIGHT_PARA';
```

5.6.3 tokenize.js

```
+let RegExpObject  = /([0-9]+)|(\+)|(\-)|(\*)|(\/)|(\()|(\))/g;
let tokenTypes = require('./tokenTypes');
+let tokenNames = [tokenTypes.NUMBER,tokenTypes.PLUS,tokenTypes.MINUS,tokenTypes.MULTIPLY,tokenTypes.DIVIDE,tokenTypes.LEFT_PARA,tokenTypes.RIGHT_PARA];
```

5.6.4 toAST.js

```
additive -> multiple|multiple + additive
multiple -> primary | primary *  multiple
primary -> NUMBER | (add)

function multiple(tokenReader){
+    let child1 = primary(tokenReader);//先配置出来NUMBER,但是这个乘法规则并没有匹配结束
    let node = child1; //node=3
    let token = tokenReader.peek();//*
    if(child1 != null && token != null){
       if(token.type
         token = tokenReader.read();//读取下一个token  *
         let child2 = multiple(tokenReader);//4
         if(child2 != null){
            node = new ASTNode(token.type
            node.appendChild(child1);
            node.appendChild(child2);
         }
       }
    }
    return node;
}
+function primary(tokenReader){ //(1+2)*3
+  let node = number(tokenReader);
+  if(!node){
+    let token = tokenReader.peek();
+    if(token != null && token.type === tokenTypes.LEFT_PARA){
+      tokenReader.read();
+      node  = additive(tokenReader);
+      tokenReader.read();
+    }
+  }
+  return node;
+}
```

5.7 二元表达式

5.7.1 优先级（Priority）

- 不同的运算符之间是有优先级的
- 加减的优先级一样,乘除的优先级一样

```
2+3*4
```

5.7.2 结合性（Associativity）

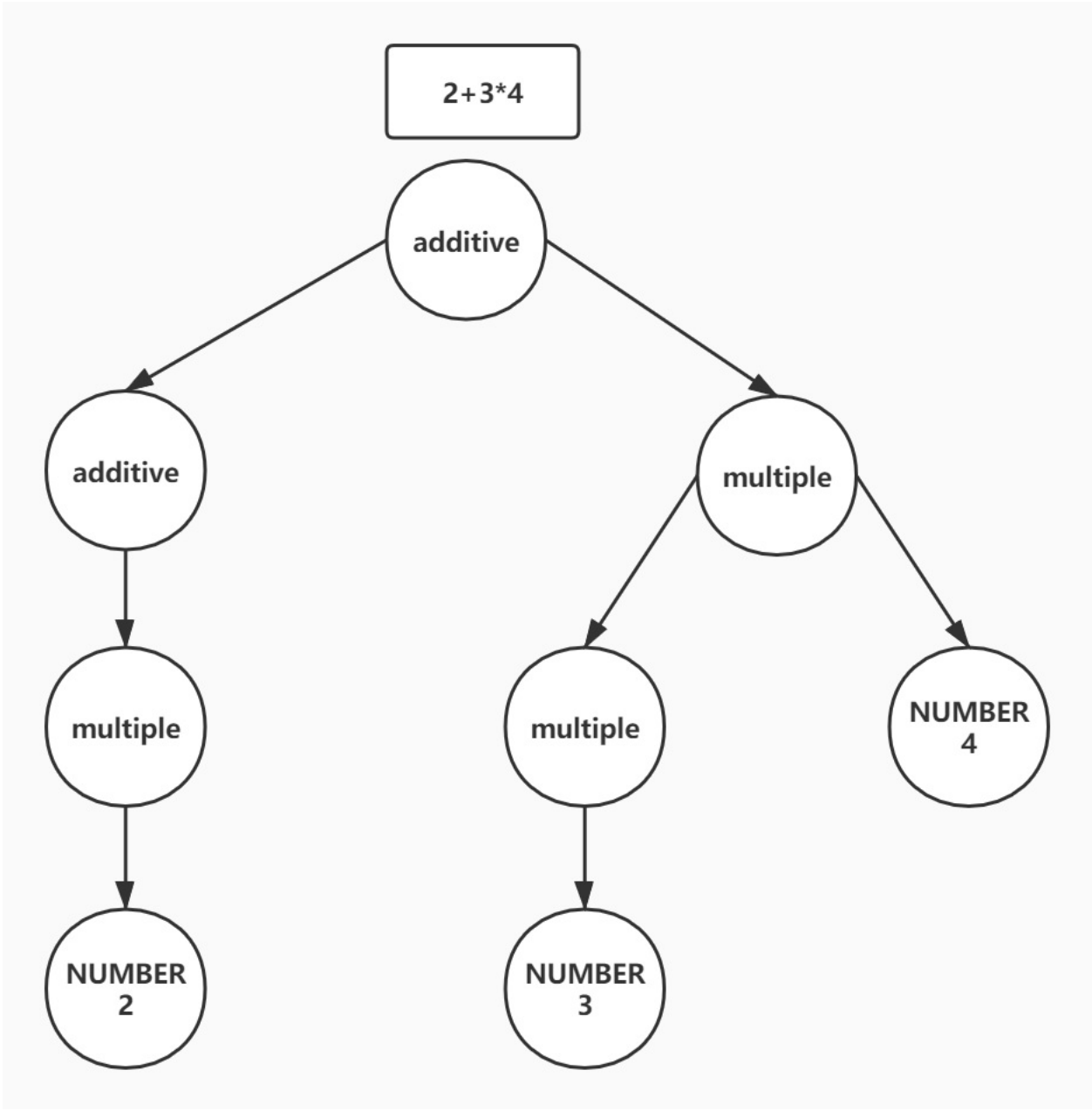
- 同样优先级的运算符是从左到右计算还是从右到左计算叫做结合性
- 加减乘除等算术运算是左结合的， .符号也是左结合的。
- 结合性是跟左递归还是右递归有关的，左递归导致左结合，右递归导致右结合

```
4+3-2-2
8/2/2
```

5.7.3 文法规则

- 优先级是通过在语法推导中的层次来决定的，优先级越低的，越先尝试推导
- 通过文法的嵌套，实现对运算优先级的支持

```
additive : multiple|additive+multiple
multiple : NUMBER|multiple*NUMBER
```



5.7.3 左递归（Left Recursive）

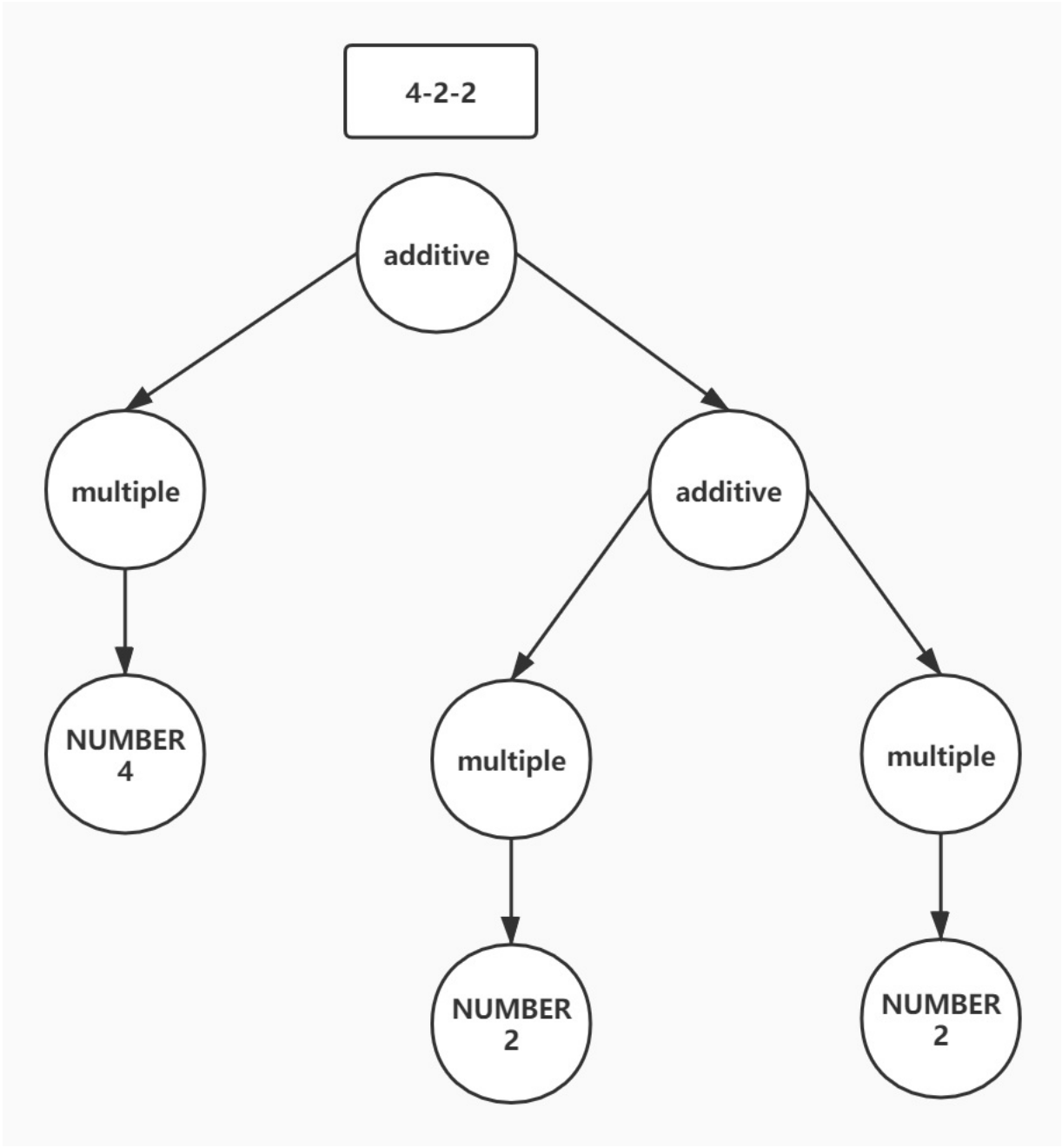
- 在二元表达式的语法规则中，如果产生式的第一个元素是它自身，那么程序就会无限地递归下去，这种情况就叫做左递归
- 巴科斯范式 以美国人巴科斯和丹麦人诺尔的名字命名的一种形式化的语法表示方法，用来描述语法的一种形式体系

```
function add(){
    add();
    multiply();
}
```

5.7.4 消除左递归

- 这样可以消除左递归,但是会带来结合性的问题

```
additive : multiple|multiple+additive
multiple : NUMBER|NUMBER*multiple
```



5.7.5 循环消除左递归

- 扩展巴科斯范式 (EBNF)会用到类似 正则表达式的一些写法
- 可以把递归改成 循环

```
additive -> multiple (+ multiple)*
multiple -> NUMBER (* NUMBER)*
```

5.7.6 实现

toAST.js

```
const ASTNode = require('./ASTNode');
let nodeTypes = require('./nodeTypes');
let tokenTypes = require('./tokenTypes');
/**
4+3-2-2
+additive -> multiple|multiple [+-] additive    包括+-
+multiple -> primary|primary [星/] multiple     包括星/
+primary -> NUMBER | (additive) 基础规则
*/
function toAST(tokenReader) {
  let rootNode = new ASTNode(nodeTypes.Program);
  //开始推导了 加法 乘法 先推导加法
  //实现的时候,每一个规则都是一个函数additive对应加法规则
  let child = additive(tokenReader);
  if (child)
    rootNode.appendChild(child);
  return rootNode;
}
//additive -> multiple|multiple [+-] additive
function additive(tokenReader) {
+  let child1 = multiple(tokenReader);
+  let node = child1;
+  if (child1 != null) {
+    while (true) {
+      let token = tokenReader.peek();//看看一下符号是不是加号
+      if (token != null && (token.type === tokenTypes.PLUS || token.type === tokenTypes.MINUS)) {
+        token = tokenReader.read();//把+-读出来并且消耗掉
+        let child2 = multiple(tokenReader);
+        node = new ASTNode(token.type === tokenTypes.PLUS ? nodeTypes.Additive : nodeTypes.Minus);
+        node.appendChild(child1);
+        node.appendChild(child2);
+        child1 = node;
+      } else {
+        break;
+      }
+    }
+  }
  return node;
}
//multiple -> primary|primary [星/] multiple
function multiple(tokenReader) {
+  let child1 = primary(tokenReader);//先配置出来NUMBER,但是这个乘法规则并没有匹配结束
+  let node = child1; //node=3
+  if (child1 != null) {
+    while (true) {
+      let token = tokenReader.peek();//*/
+      if (token != null && (token.type === tokenTypes.MULTIPLY || token.type === tokenTypes.DIVIDE)) {
+        token = tokenReader.read();//把*/读出来并且消耗掉
+        let child2 = primary(tokenReader);
+        node = new ASTNode(token.type === tokenTypes.MULTIPLY ? nodeTypes.Multiplicative : nodeTypes.Divide);
+        node.appendChild(child1);
+        node.appendChild(child2);
+        child1 = node;
+      } else {
+        break;
+      }
+    }
+  }
  return node;
}
//primary -> NUMBER | (additive) 基础规则
function primary(tokenReader) {
  let node = number(tokenReader);
  if (!node) {
    let token = tokenReader.peek();
    if (token != null && token.type
      tokenReader.read();
      node = additive(tokenReader);
      tokenReader.read();
    }
  }
  return node;
}
function number(tokenReader) {
  let node = null;
  let token = tokenReader.peek();//看看当前的这个token
  //如果能取出 token,并且token的类型是数字的话 匹配上了
  if (token != null && token.type
    token = tokenReader.read();//读取并消耗掉这个Token
    //创建一个新的语法树节点,类型是Numeric,值是2
    node = new ASTNode(nodeTypes.Numeric, token.value);
  }
  return node;
}

module.exports = toAST;
```

6. 语法分析器

- 语法分析器接受 token 数组，然后把它转化为 AST

6.1 nodeTypes.js

src\nodeTypes.js

```
exports.Program = 'Program';
exports.ExpressionStatement = 'ExpressionStatement';

exports.JSXElement = 'JSXElement';

exports.JSXOpeningElement = 'JSXOpeningElement';

exports.JSXAttribute = 'JSXAttribute';

exports.JSXIdentifier = 'JSXIdentifier';

exports.AttributeKey='AttributeKey';

exports.JSXClosingElement='JSXClosingElement';

exports.StringLiteral = 'StringLiteral';

exports.JSXText = 'JSXText';

exports.MemberExpression = 'MemberExpression';
exports.ObjectExpression = 'ObjectExpression';
exports.ObjectProperty = 'ObjectProperty';
exports.CallExpression = 'CallExpression';
exports.Identifier = 'Identifier';
exports.NumberLiteral = 'NumberLiteral';
exports.StringLiteral = 'StringLiteral';
exports.NullLiteral = 'NullLiteral';
```

6.2 parser.js

src\parser.js

```
const { tokenizer } = require('./tokenizer');
const tokenTypes = require('./tokenTypes');
const nodeTypes = require('./nodeTypes');
function parser(code) {
    let tokens = tokenizer(code);
    let current = 0;
    function walk(parent) {
        let token = tokens[current];
        let next = tokens[current + 1];

        if (token.type === tokenTypes.LeftParentheses && next.type === tokenTypes.JSXIdentifier) {
            let node = {
                type: nodeTypes.JSXElement,
                openingElement: null,
                closingElement: null,
                children: []
            }
            token = tokens[++current];
            node.openingElement = {
                type: nodeTypes.JSXOpeningElement,
                name: {
                    type: nodeTypes.JSXIdentifier,
                    name: token.value
                },
                attributes: []
            }

            token = tokens[++current];

            while (token.type === tokenTypes.AttributeKey) {
                node.openingElement.attributes.push(walk());
                token = tokens[current];
            }
            token = tokens[++current];
            next = tokens[current+1];

            while (
                token.type !== tokenTypes.LeftParentheses
                ||( token.type === tokenTypes.LeftParentheses
                    && next.type !== tokenTypes.BackSlash)) {
                node.children.push(walk());
                token = tokens[current];
                next = tokens[current+1];
            }
            node.closingElement = walk(node);
            return node;
        }

        else if(parent && token.type === tokenTypes.LeftParentheses && next.type === tokenTypes.BackSlash){
            current++;
            token = tokens[++current];
            current++;
            current++;
            return parent.closingElement = {
                type: nodeTypes.JSXClosingElement,
                name: {
                    type: nodeTypes.JSXIdentifier,
                    name: token.value
                }
            }
        }
        else if(token.type === tokenTypes.AttributeKey){
            let next = tokens[++current];
            let node = {
                type: nodeTypes.JSXAttribute,
                name: {
                    type: nodeTypes.JSXIdentifier,
                    name: token.value
                },
                value: {
                    type: nodeTypes.StringLiteral,
                    value: next.value
                }
            }
            current++;
            return node;
        }
        else if(token.type === tokenTypes.JSXText){
            current++;
            return {
                type: nodeTypes.JSXText,
                value: token.value
            }
        }
        throw new TypeError(token.type);
    }
    var ast = {
        type: nodeTypes.Program,
        body: [
            {
                type: nodeTypes.ExpressionStatement,
                expression: walk()
            }
        ]
    };
    return ast;
}

module.exports = {
    parser
}
```

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "JSXElement",
        "openingElement": {
          "type": "JSXOpeningElement",
          "name": {
            "type": "JSXIdentifier",
            "name": "h1"
          },
          "attributes": [
            {
              "name": {
                "type": "JSXIdentifier",
                "name": "id"
              },
              "value": {
                "type": "StringLiteral",
                "value": "title"
              }
            }
          ]
        },
        "closingElement": {
          "type": "JSXClosingElement",
          "name": {
            "type": "JSXIdentifier",
            "name": "h1"
          }
        },
        "children": [
          {
            "type": "JSXElement",
            "openingElement": {
              "type": "JSXOpeningElement",
              "name": {
                "type": "JSXIdentifier",
                "name": "span"
              },
              "attributes": []
            },
            "closingElement": {
              "type": "JSXClosingElement",
              "name": {
                "type": "JSXIdentifier",
                "name": "span"
              }
            },
            "children": [
              {
                "type": "JSXText",
                "value": "hello"
              }
            ]
          },
          {
            "type": "JSXText",
            "value": "world"
          }
        ]
      }
    }
  ]
}
```


7. 遍历语法树

src\traverse.js

```
const { parser } = require('./parser');
const replace = (parent,oldNode,newNode)=>{
    if(parent){
        for (const key in parent) {
            if (parent.hasOwnProperty(key)) {
                if(parent[key] === oldNode){
                    parent[key] = newNode;
                }
            }
        }
    }
}
function traverse(ast, visitor) {
    function traverseArray(array, parent) {
        array.forEach(function (child) {
            traverseNode(child, parent);
        });
    }

    function traverseNode(node, parent) {
        let replaceWith  = replace.bind(node,parent,node);
        var method = visitor[node.type];
        if (method) {
            if (typeof method === 'function') {
                method({node,replaceWith}, parent)
            } else if (method.enter) {
                method.enter({node,replaceWith}, parent);
            }
        }
        switch (node.type) {
            case 'Program':
                traverseArray(node.body, node);
                break;
            case 'ExpressionStatement':
                traverseNode(node.expression, node);
                break;

            case 'JSXElement':
                traverseNode(node.openingElement, node);
                traverseNode(node.closingElement, node);
                traverseArray(node.children, node);
                break;
            case 'JSXOpeningElement':
                traverseNode(node.name, node);
                traverseArray(node.attributes, node);
                break;
            case 'JSXAttribute':
                traverseNode(node.name, node);
                traverseNode(node.value, node);
                break;
            case 'JSXClosingElement':
                traverseNode(node.name, node);
                break;
            case 'JSXIdentifier':
                break;
            case 'StringLiteral':
                break;
            case 'JSXText':
                break;

            case 'CallExpression':
                traverseNode(node.callee, node);
                traverseArray(node.arguments, node);
                break;
            case 'MemberExpression':
                traverseNode(node.object, node);
                traverseNode(node.property, node);
                break;
            case 'Identifier':
                break;
            case 'ObjectExpression':
                traverseArray(node.properties, node);
                break;
            case 'ObjectProperty':
                traverseNode(node.key, node);
                traverseNode(node.value, node);
                break;
            case 'NullLiteral':
                break;
            default:
                throw new TypeError(node.type);
        }
        if (method && method.exit) {
            method.exit({node,replaceWith}, parent);
        }
    }

    traverseNode(ast, null);
}

module.exports = {
    traverse
};
```

8. 转换器

src\transformer.js

```
const { traverse } = require('./traverse');
const { parser } = require('./parser');
const nodeTypes = require('./nodeTypes');
class t {
```

```
    static nullLiteral() {
        return {
            type: nodeTypes.NullLiteral
        }
    }

    static memberExpression(object, property) {
        return {
            type: nodeTypes.MemberExpression,
            object,
            property
        }
    }

    static identifier(name) {
        return {
            type: nodeTypes.Identifier,
            name
        }
    }

    static stringLiteral(value) {
        return {
            type: nodeTypes.StringLiteral,
            value
        }
    }

    static objectExpression(properties) {
        return {
            type: nodeTypes.ObjectExpression,
            properties
        }
    }

    static objectProperty(key, value) {
        return {
            type: nodeTypes.ObjectProperty,
            key,
            value
        }
    }

    static callExpression(callee, _arguments) {
        return {
            type: nodeTypes.CallExpression,
            callee,
            arguments: _arguments
        }
    }

    static isJSXText(node) {
        return node.type ===nodeTypes.JSXText
    }

    static isJSXElement(node) {
        return node.type === nodeTypes.JSXElement;
    }
}

function transformer(ast) {
    traverse(ast, {
        JSXElement(nodePath) {
            const transform = (node) => {debugger
                if (!node) return t.nullLiteral();

                if (t.isJSXElement(node)) {

                    let memberExpression = t.memberExpression(
                        t.identifier("React"),
                        t.identifier("createElement")
                    );

                    let _arguments = [];

                    let stringLiteral = t.stringLiteral(node.openingElement.name.name);

                    let objectExpression = node.openingElement.attributes.length
                        ? t.objectExpression(
                            node.openingElement.attributes.map((attr) =>
                                t.objectProperty(t.identifier(attr.name.name), attr.value)
                            )
                        )
                        : t.nullLiteral();
                    _arguments = [stringLiteral, objectExpression];

                    _arguments.push(...node.children.map((item) => transform(item)));
                    return t.callExpression(memberExpression, _arguments);
                } else if (t.isJSXText(node)) {

                    return t.stringLiteral(node.value);
                }
            };
            let targetNode = transform(nodePath.node);
            nodePath.replaceWith(targetNode);
        },
    });
}

module.exports = {
    transformer
}
```

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "object": {
            "type": "Identifier",
            "name": "React"
          },
          "property": {
            "type": "Identifier",
            "name": "createElement"
          }
        },
        "arguments": [
          {
            "type": "StringLiteral",
            "value": "h1"
          },
          {
            "type": "ObjectExpression",
            "properties": [
              {
                "type": "ObjectProperty",
                "key": {
                  "type": "Identifier",
                  "name": "id"
                },
                "value": {
                  "type": "StringLiteral",
                  "value": "title"
                }
              }
            ]
          },
          {
            "type": "CallExpression",
            "callee": {
              "type": "MemberExpression",
              "object": {
                "type": "Identifier",
                "name": "React"
              },
              "property": {
                "type": "Identifier",
                "name": "createElement"
              }
            },
            "arguments": [
              {
                "type": "StringLiteral",
                "value": "span"
              },
              {
                "type": "NullLiteral"
              },
              {
                "type": "StringLiteral",
                "value": "hello"
              }
            ]
          },
          {
            "type": "StringLiteral",
            "value": "world"
          }
        ]
      }
    }
  ]
}
```


9. 生成器

- 先序遍历抽象语法树，将特定的节点类型转换成对应的js代码即可

src\codeGenerator.js

```
const nodeTypes = require('./nodeTypes');
function codeGenerator(node) {
    switch (node.type) {
        case nodeTypes.Program:
            return node.body.map(codeGenerator).join('\n');
        case nodeTypes.ExpressionStatement:
            return (
                codeGenerator(node.expression) + ';'
            );

        case nodeTypes.MemberExpression:
            return (
                codeGenerator(node.object) +
                '.' +
                codeGenerator(node.property)
            )

        case nodeTypes.ObjectExpression:
            return (
                '{' +
                node.properties.map(codeGenerator).join(', ') +
                '}'
            )

        case nodeTypes.ObjectProperty:
            return (
                codeGenerator(node.key) +
                ':' +
                codeGenerator(node.value)
            )

        case nodeTypes.CallExpression:
            return (
                codeGenerator(node.callee) +
                '(' +
                node.arguments.map(codeGenerator).join(', ') +
                ')'
            );

        case nodeTypes.Identifier:
            return node.name;

        case nodeTypes.NumberLiteral:
            return node.value;

        case nodeTypes.StringLiteral:
            return '"' + node.value + '"';

        case nodeTypes.NullLiteral:
            return 'null'

        default:
            throw new TypeError(node.type);
    }
}

module.exports = {
    codeGenerator
}
```

src\index.js

```
const { parser } = require('./parser');
const { transformer } = require('./transformer');
const { codeGenerator } = require('./codeGenerator');
let code = 'helloworld';
let ast = parser(code);
transformer(ast);
console.log(JSON.stringify(ast, null, 2));
let result = codeGenerator(ast);
console.log(result);
```