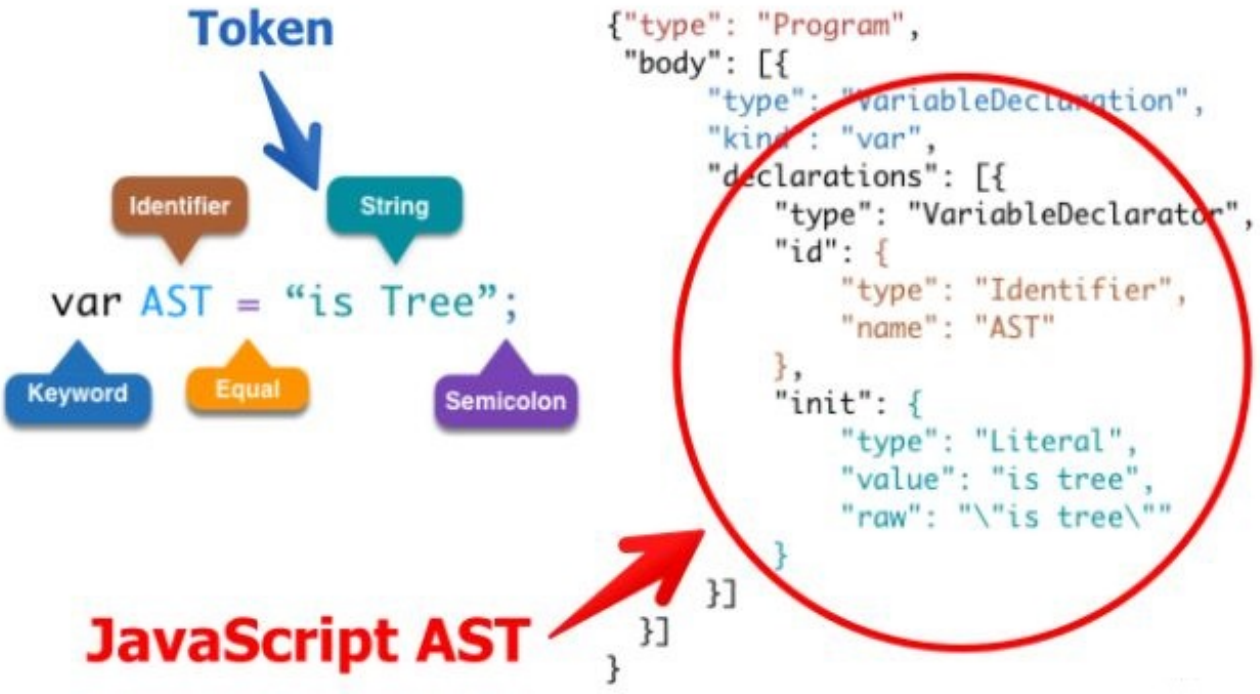
1.抽象语法树(Abstract Syntax Tree)

- 抽象语法树（Abstract Syntax Tree，AST）是源代码语法结构的一种抽象表示
- 它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构



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
- 优化变更代码，改变代码结构使达到想要的结构

3.JavaScript Parser

- `JavaScript Parser`是把JavaScript源码转化为抽象语法树的解析器
- astexplorer

4.代码转换

- 1.将代码转换成ast语法树
- 2.深度优先遍历，遍历ast抽象语法树
- 3.代码生成

```
pnpm i esprima estraverse escodegen -S
```

```
const esprima = require("esprima");
const estraverse = require("estraverse");
```

```
const escodegen = require("escodegen");
const code = `function ast(){}`;
// 1.将代码转换成ast语法树
const ast = esprima.parseScript(code);
// 2.深度优先遍历，遍历ast抽象语法树
let level = 0;
const padding = () => " ".repeat(level);
estraverse.traverse(ast, {
  enter(node) {
    console.log(padding() + "enter:" + node.type);
    level += 2;
  },
  leave(node) {
    level -= 2;
    console.log(padding() + "leave:" + node.type);
  },
});
// 3.代码生成
const result = escodegen.generate(ast);
console.log(result);
```

```
estraverse.traverse(ast, {
  enter(node) {
    if (node.type === "FunctionDeclaration") {
      node.id.name = "myFunc";
    }
  },
});
```

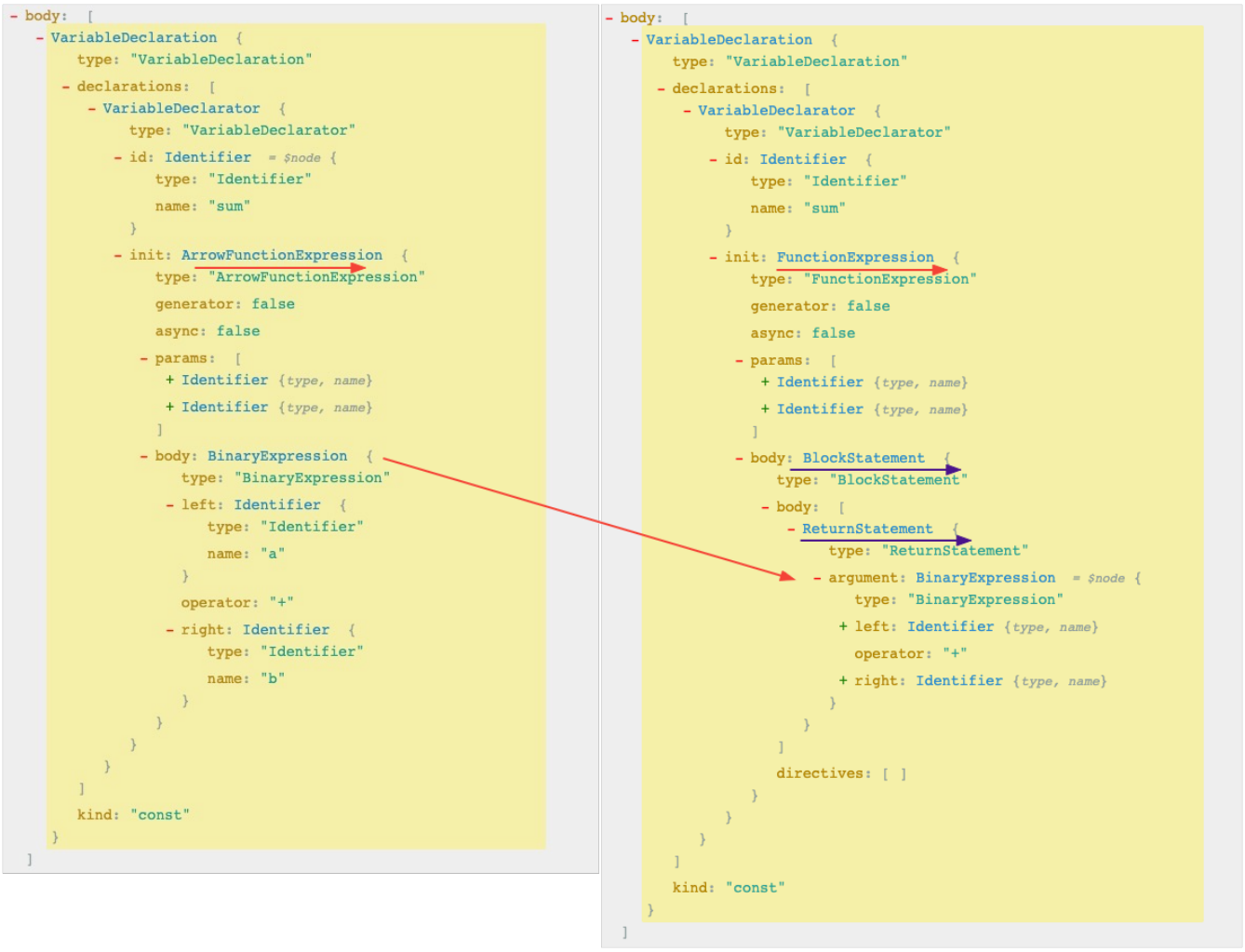
5.babel插件

5.1 转换箭头函数

- @babel/core Babel 的编译器，核心 API 都在这里面，比如常见的 transform、parse,并实现了插件功能
- @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-plugin-transform-es2015-arrow-functions 转换箭头函数插件

```
const babel = require("@babel/core");
const types = require("@babel/types");
const arrowFunctions = require("babel-plugin-transform-es2015-arrow-functions");

const code = `const sum = (a,b)=>  a+b`;
// 转化代码，通过arrowFunctions插件
const result = babel.transform(code, {
  plugins: [arrowFunctions],
});
console.log(result.code);
```




```
const arrowFunctions = {
  visitor: {
    // 访问者模式,遇到箭头函数表达式会命中此方法
    ArrowFunctionExpression(path) {
      let { node } = path;
      node.type = "FunctionExpression"; // 将节点转换成函数表达式
      let body = node.body;
      // 如果不是代码块，则增加代码块及return语句
      if (!types.isBlockStatement(body)) {
        node.body = types.blockStatement([types.returnStatement(body)]);
      }
    }
  },
};
```

> `@babel/types`主要就是对类型的判断和创建，接下来，我们继续处理箭头函数中的this问题！插件手册

```
const sum = (a,b)=> console.log(this) // 源代码

----transform ----

var _this = this; // 转换后的代码
const sum = function (a, b) {
  return console.log(_this);
};
```

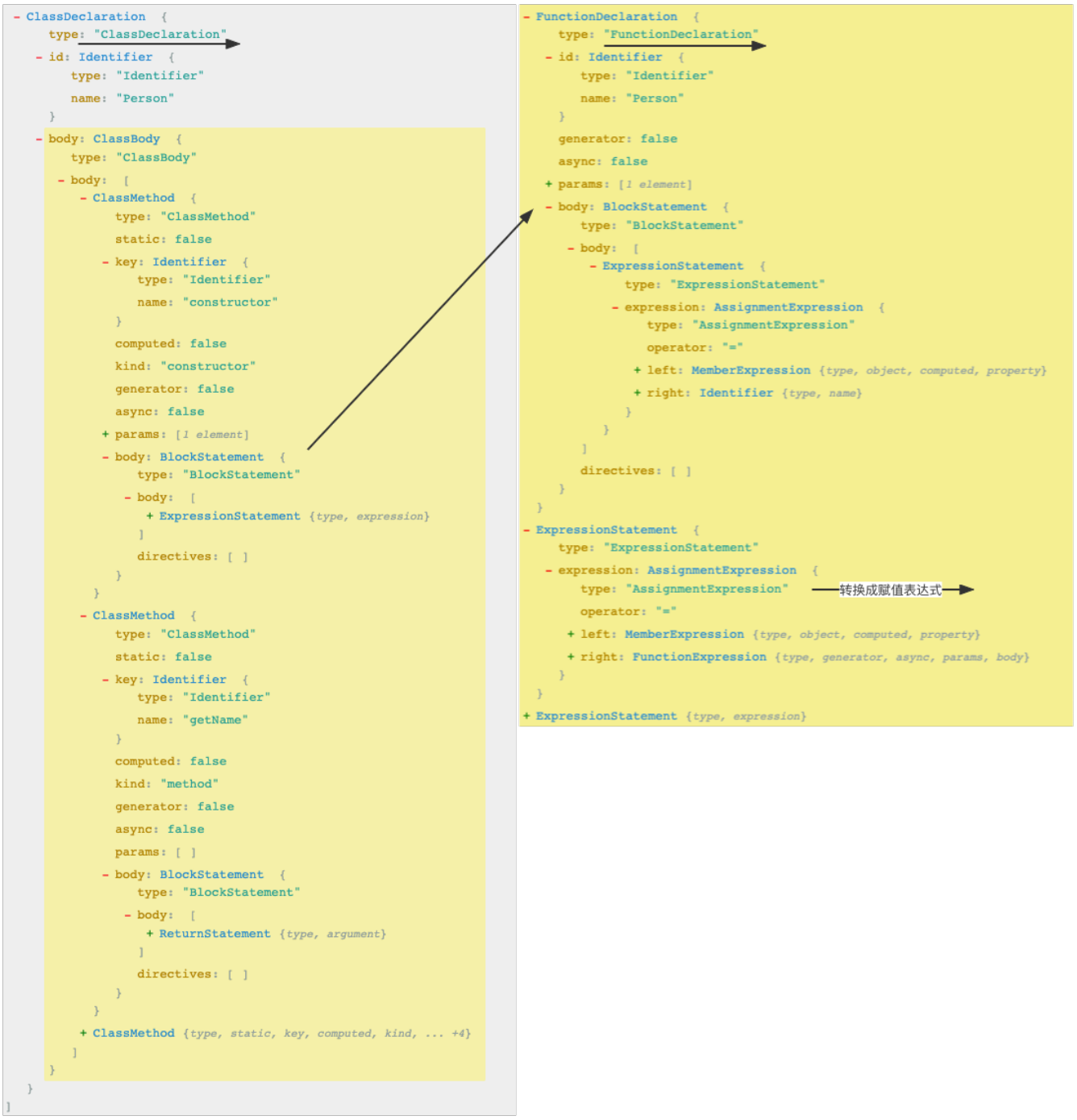
> 需要找到上级作用域增加this的声明语句

```
function getThisPaths(path) {
  const thisPaths = [];
  path.traverse({
    ThisExpression(path) {
      thisPaths.push(path);
    },
  });
  return thisPaths; // 获得所有子路径中的thisExpression
}
function hoistFunctionEnvironment(path) {
  const thisEnv = path.findParent((parent) => {
    return (
      (parent.isFunction() && !path.isArrowFunctionExpression()) ||
      parent.isProgram()
    );
  });
  const thisBindings = "_this"; // 找到this表达式替换成_this
  const thisPaths = getThisPaths(path); // 遍历子路径
  if (thisPaths.length > 0) {
    if (!thisEnv.scope.hasBinding(thisBindings)) {
      // 在作用域下增加 var _this = this
      thisEnv.scope.push({
        id: types.identifier(thisBindings),
        init: types.thisExpression(),
      });
    }
  }
  // 替换this表达式
  thisPaths.forEach((thisPath) =>
    thisPath.replaceWith(types.identifier(thisBindings))
  );
}
```

5.2 类编译为 Function

```
class Person {
  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
  setName(newName) {
    this.name = newName;
  }
}
```

```
function Person(name) {
  this.name = name;
}
Person.prototype.getName = function () {
  return this.name;
};
Person.prototype.setName = function () {
  this.name = newName;
};
```




```
const arrowFunctions = {
  visitor: {
    ClassDeclaration(path) {
      const { node } = path;
      const id = node.id;
      const methods = node.body.body; // 获取类中的方法
      const nodes = [];
      methods.forEach((method) => {
        if (method.kind === "constructor") {
          let constructorFunction = types.functionDeclaration(
            id,
            method.params,
```

```
            method.body
          );
          nodes.push(constructorFunction);
        } else {
          // Person.prototype.getName
          const memberExpression = types.memberExpression(
            types.memberExpression(id, types.identifier("prototype")),
            method.key
          );
          // function(name){return name}
          const functionExpression = types.functionExpression(
            null,
            method.params,
            method.body
          );
          // 赋值
          const assignmentExpression = types.assignmentExpression(
            "=",
            memberExpression,
            functionExpression
          );
          nodes.push(assignmentExpression);
        }
      });
      // 替换节点
      if (node.length === 1) {
        path.replaceWith(nodes[0]);
      } else {
        path.replaceWithMultiple(nodes);
      }
    },
  },
};
```

6.Eslint使用

ESLint 是一个开源的工具cli，ESLint采用静态分析找到并修复 JavaScript 代码中的问题

- ESLint 使用Espree进行 JavaScript 解析。
- ESLint 使用 AST 来评估代码中的模式。
- ESLint 是完全可插拔的，每一条规则都是一个插件，你可以在运行时添加更多。

> 扯皮一下这些解析器的关系~~~

- esprima - 经典的解析器
- acorn - 造轮子媲美Esprima
- @babel/parser （babylon）基于acorn的
- espree 最初从Esprima中fork出来的，现在基于acorn

```
pnpm init
pnpm i eslint -D # 安装eslint
pnpm create @eslint/config  # 初始化eslint的配置文件
```

> 生成的配置文件是：

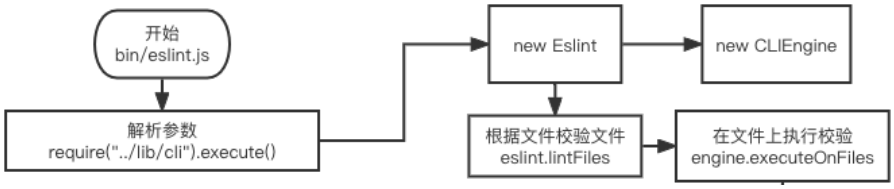
```
module.exports = {
  env: { // 指定环境
    browser: true, // 浏览器环境   document
    es2021: true,  // ECMAScript语法 Promise
    node: true,    // node环境  require
  },
  extends: "eslint:recommended",
  parserOptions: { // 支持语言的选项，支持最新js语法，同时支持jsx语法
    ecmaVersion: "latest", // 支持的语法是
    sourceType: "module", // 支持模块化
    ecmaFeatures:{
      "jsx":true
    }
  },
  rules: { // eslint规则
    "semi": ["error", "always"],
    "quotes": ["error", "double"]
  },
  globals:{ // 配置全局变量
    custom:"readonly" // readonly 、 writable
  }
};
```

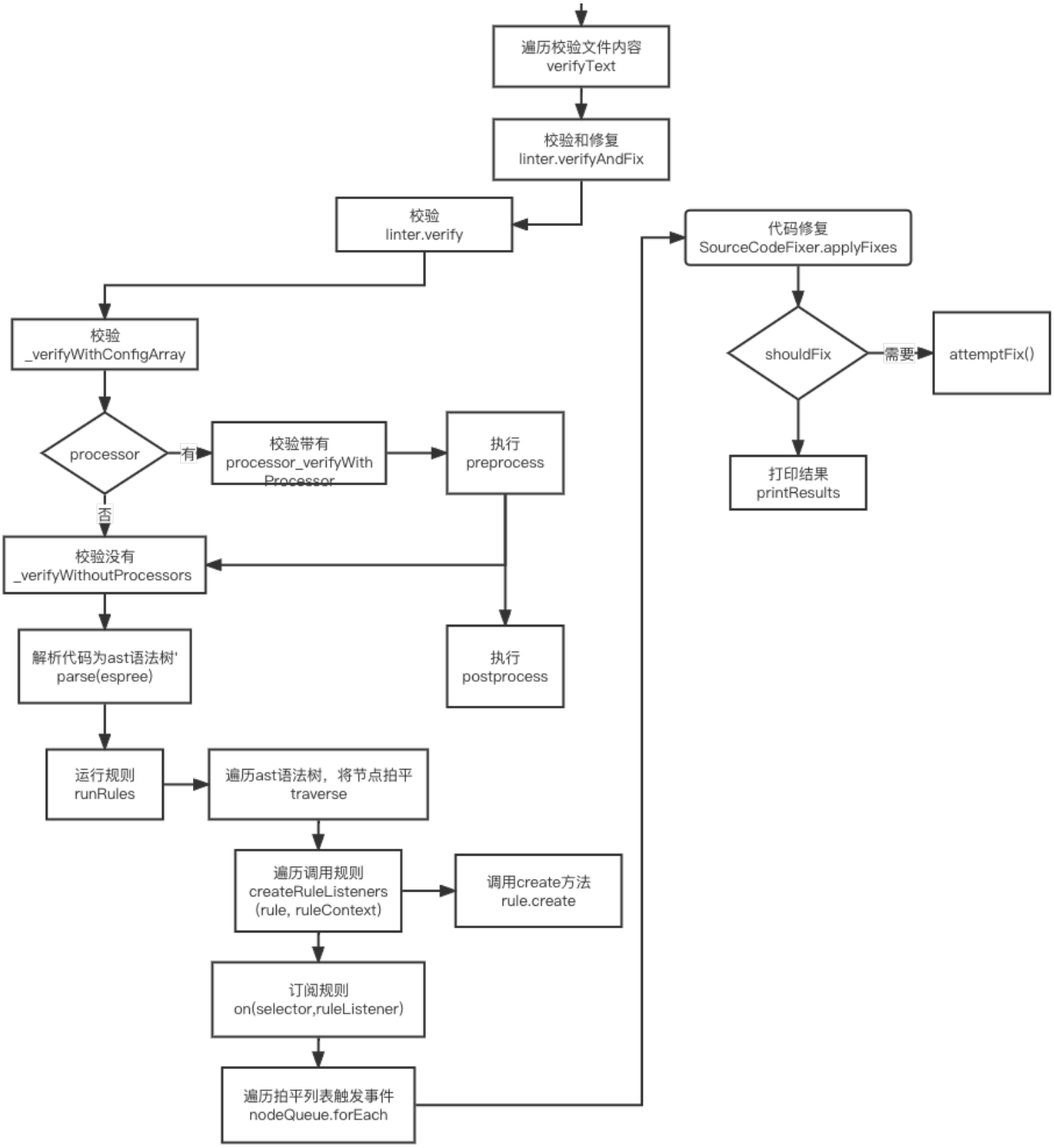
parser：可以指定使用何种parser来将code转换成estree。举例：转化ts语法

```
pnpm install @typescript-eslint/eslint-plugin@latest @typescript-eslint/parser@latest
typescript -D
```

```
"extends": [
  "eslint:recommended",
  "plugin:@typescript-eslint/recommended" // 集成规则和插件
],
"parser": "@typescript-eslint/parser" // 解析typescript
```

7.Eslint运行流程





ESLint 核心API

- lintFiles 检验文件
- lintText 校验文本
- loadFormatter 加载formatter
- calculateConfigForFile 通过文件获取配置
- isPathIgnored 此路径是否是被忽略的
- static outputFixes 输出修复的文件
- static getErrorResults 获得错误结果

CLIEngine 脚手架核心

- getRules 获取规则
- resolveFileGlobPatterns 解析文件成glob模式
- executeOnFiles 根据文件执行逻辑
- executeOnText 根据文本执行逻辑
- getConfigForFile 获取文件的配置
- isPathIgnored 此路径是否是被忽略的
- getFormatter 获取输出的格式
- static getErrorResults 获取错误结果
- static outputFixes 输出修复的结果

Linter 校验js文本

- verifyAndFix 校验和修复
- verify 校验方法
- _verifyWithConfigArray 通过配置文件校验
- _verifyWithoutProcessors 校验没有processors
- _verifyWithProcessor 校验有processors

```
ESLint 核心API
  - lintFiles 检验文件
  - lintText 校验文本
  - loadFormatter 加载formatter
  - calculateConfigForFile 通过文件获取配置
  - isPathIgnored 此路径是否是被忽略的
  - static outputFixes 输出修复的文件
  - static getErrorResults 获得错误结果
```

```
CLIEngine 脚手架核心
  - getRules 获取规则
  - resolveFileGlobPatterns 解析文件成glob模式
  - executeOnFiles 根据文件执行逻辑
  - executeOnText 根据文本执行逻辑
  - getConfigForFile 获取文件的配置
  - isPathIgnored 此路径是否是被忽略的
  - getFormatter 获取输出的格式
  - static getErrorResults 获取错误结果
  - static outputFixes 输出修复的结果
```

```
Linter 校验js文本
  - verifyAndFix 校验和修复
  - verify 校验方法
  - _verifyWithConfigArray 通过配置文件校验
  - _verifyWithoutProcessors 校验没有processors
  - _verifyWithProcessor 校验有processors
```

8.eslint插件开发

> Eslint官方提供了可以使用Yeoman 脚手架生成插件模板

```
npm install yo generator-eslint -g
```

8.1 模板初始化

```
mkdir eslint-plugin-zlint
cd eslint-plugin-zlint
yo eslint:plugin # 插件模板初始化
```

```
→  packages mkdir eslint-plugin-zlint
→  packages cd eslint-plugin-zlint
→  eslint-plugin-zlint yo eslint:plugin
? What is your name? zi-shui
? What is the plugin ID? zlint
? Type a short description of this plugin: 自己实现的一个eslint-plugin
? Does this plugin contain custom ESLint rules? Yes
? Does this plugin contain one or more processors? No
   create package.json
   create .eslintrc.js
   create lib/index.js
   create README.md

Changes to package.json were detected.
```

```
yo eslint:rule # 规则模版初始化
```

```
→  eslint-plugin-zlint yo eslint:rule
? What is your name? zi-shui
? Where will this rule be published? ESLint Plugin
? What is the rule ID? no-console
? Type a short description of this rule: 禁止使用console
? Type a short example of the code that will fail:
   create docs/rules/no-console.md
   create lib/rules/no-console.js
   create tests/lib/rules/no-console.js

No change to package.json was detected. No package manager install will be executed.
```

8.2 实现no-var

```
module.exports = {
    "env": {
        "browser": true,
        "es2021": true,
        "node": true
    },
    "plugins": ['zlint'],
    "rules": {
        'zlint/no-var': ['error', 'always']
    }
}
```

```
module.exports = {
  meta: {
    docs: {
      description: "代码中不能出现var",
      recommended: false,
    },
    fixable: "code",
    messages: {
      unexpectedVar: 'Unexpected var'
    }
  },
  create(context) {
    const sourceCode = context.getSourceCode();
    return {
```

```
      "VariableDeclaration:exit"(node) { // 如果类型是var，拦截var声明
        if (node.kind === 'var') {
          context.report({ // 报警告
            node,
            messageId: 'unexpectedVar',
            fix(fixer) { // --fix
              const varToken = sourceCode.getFirstToken(node, { filter: t => t.value
=== 'var' });
              // 将var 转换成 let
              return fixer.replaceText(varToken, 'let')
            }
          })
        }
      }
    };
  },
};
```

> 文档：[eslint中文]

```
const ruleTester = new RuleTester({
  parserOptions: {
    ecmaVersion: 'latest',
  },
});
ruleTester.run("no-var", rule, {
  valid: [
    {
      code: "let a = 1"
    }
  ],

  invalid: [
    {
      code: "var a = 1",
      errors: [{
        messageId: "unexpectedVar",
      }],
      output: 'let a = 1'
    },
  ],
});
```

> 单元测试

8.3 实现eqeqeq

```
module.exports = {
    env: { // 指定环境
        browser: true, // 浏览器环境
        es2021: true,  // ECMAScript2021
        node: true,    // node环境
    },
    plugins: ['zlint'],
    rules: {
        "zlint/eqeqeq": ['error', "always", { null: 'never' }]
    }
}
```

```
module.exports = {
  meta: {
    type: 'suggestion', // `problem`, `suggestion`, or `layout`
    docs: {
      description: "尽量使用三等号",
      recommended: false,
    },
    fixable: 'code',
    schema: {
      type: "array",
      items: [
        {
          enum: ["always"]
        },
        {
          type: "object",
          properties: {
            null: {
              enum: ["always", "never"]
            }
          }
        }
      ],
    },
    messages: {
      unexpected: "期望 '{{expectedOperator}}' 目前是 '{{actualOperator}}'."
    }
  },
  create(context) {
    // 处理null的情况
    const config = context.options[0] || 'always';
    const options = context.options[1] || {};
    const nullOption = (config === 'always') ? options.null || 'always' : ''
```

```
    const enforceRuleForNull = (nullOption === 'never')

    function isNullCheck(node) {
      let rightNode = node.right;
      let leftNode = node.left;
      return (rightNode.type === 'Literal' && rightNode.value === null) ||
(leftNode.type === 'Literal' && leftNode.value === null)
    }

    const sourceCode = context.getSourceCode();
    function report(node, expectedOperator) {
      const operatorToken = sourceCode.getFirstTokenBetween(node.left, node.right, {
filter: t => t.value === node.operator });

      // 左右两边是不是相同类型

      function areLiteralsAndSameType(node) {
        return node.left.type === "Literal" && node.right.type === "Literal" &&
          typeof node.left.value === typeof node.right.value;
      }
      context.report({
        node,
        loc: operatorToken.loc,
        data: { expectedOperator, actualOperator: node.operator },
        messageId: 'unexpected',
        fix(fixer) {
          if (areLiteralsAndSameType(node)) {
            return fixer.replaceText(operatorToken, expectedOperator);
          }
          return null
        }
      })
    }
    return {
      BinaryExpression(node) {
        const isNull = isNullCheck(node);
        // 如果是两等号的情况
        if (node.operator !== '==' && node.operator !== '!=') {
          if (enforceRuleForNull && isNull) { // 当遇到null的时候不进行转换
            report(node, node.operator.slice(0, -1));
          }
          return
        }
        report(node, `${node.operator}=`)
      }
    };
  },
};
```

> 单元测试

```
onst ruleTester = new RuleTester();
ruleTester.run("eqeqeq", rule, {
  valid: [
    {
      code: '1===1'
    }
  ],

  invalid: [
    {
      code: "1==1",
      errors: [{
        data: { expectedOperator: '===', actualOperator: '==' },
        messageId: 'unexpected',
      }],
      output: "1===1"
    },
    {
      code: "null === undefined",
      options: ["always", { null: 'never' }],
      errors: [{
        data: { expectedOperator: '==', actualOperator: '===' },
        messageId: 'unexpected',
      }],
    },
  ],
});
```

8.4 实现no-console

```
rules: {
  "zlint/no-console": ['warn', { allow: ['log'] }],
  "zlint/no-var": 'error'
}
```

```
module.exports = {
  meta: {
    type: 'suggestion',
    docs: {
      description: "禁止使用console",
      recommended: false,
    },
    schema: [
      {
        type: 'object',
```

```
        properties: {
          allow: {
            type: 'array',
            items: {
              type: 'string'
            }
          }
        }
      }
    ]
  },
  create(context) {
    const allowed = context.options[0]?.allow || [];
    function isMemberAccessExceptAllowed(reference) {
      const node = reference.identifier
      const parent = node.parent;
      if (parent.type === 'MemberExpression') { // 获得父节点，看下属性名字
        const { name } = parent.property;
        return !allowed.includes(name);
      }
    }
    function report(reference) {
      const node = reference.identifier.parent; // console 表达式
      context.report({
        node,
        loc: node.loc,
        message: 'Unexpected console statement.'
      })
    }
    return {
      "Program:exit"() {
        // 1. 当前作用域
        let scope = context.getScope();
        // 2. 获得console变量
        const variable = scope.set.get('console');
        // 3. 获得references
        const references = variable.references;
        references.filter(isMemberAccessExceptAllowed).forEach(report)
      }
    };
  },
};
```

单元测试

```
const ruleTester = new RuleTester({
  env: {
    browser: true
  }
```

```
});
ruleTester.run("no-console", rule, {
  valid: [
    {
      code: "console.log('hello')",
      options: [{ allow: ['log'] }]
    }
  ],

  invalid: [
    {
      code: "console.info('hello')",
      errors: [{ message: "Unexpected console statement." }],
    },
  ],
});
```

9.extends 使用

```
module.exports = {
    rules: requireIndex(__dirname + "/rules"),
    configs: {
        // 导出自定义规则 在项目中直接引用
        recommended: {
            plugins: ['zlint'], // 引入插件
            rules: {
                // 开启规则
                'zlint/eqeqeq': ['error', "always", { null: 'never' }],
                'zlint/no-console': ['error', { allow: ['log'] }],
                'zlint/no-var': 'error'
            }
        }
    },
    processors: {
        '.vue': {
            preprocess(code) {
                console.log(code)
                return [code]
            },
            postprocess(messages) {
                console.log(messages)
                return []
            }
        }
    }
};
```

```
module.exports = {
    "env": {
        "browser": true,
        "es2021": true,
        "node": true
    },
    extends: [
        'plugin:zlint/recommended' // 直接集成即可
    ]
}
```