link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=149 sentences=434, words=2721

1.抽象语法树(Abstract Syntax Tree)

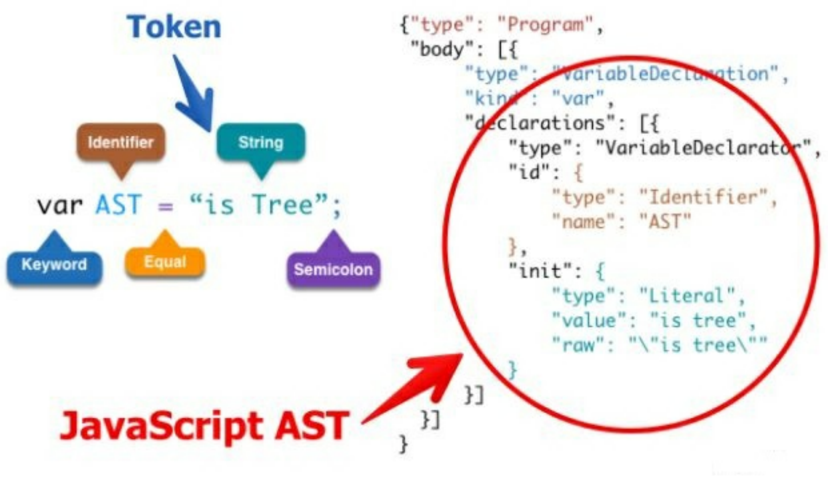
- 抽象语法树（Abstract Syntax Tree，AST）是源代码语法结构的一种抽象表示
- 它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构

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.抽象语法树定义

- 这些工具的原理都是通过 `JavaScript Parser`把代码转化为一颗抽象语法树（AST），这颗树定义了代码的结构，通过操纵这颗树，我们可以精准的定位到声明语句、赋值语句、运算语句等等，实现对代码的分析、优化、变更等操作



```
{"type": "Program",
 "body": [{
     "type": "VariableDeclaration",
     "kind": "var",
     "declarations": [{
         "type": "VariableDeclarator",
         "id": {
             "type": "Identifier",
             "name": "AST"
         },
         "init": {
             "type": "Literal",
             "value": "is tree",
             "raw": "\"is tree\""
         }
     }]
 }]
}
```

4. JavaScript Parser

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

4.1 常用的 JavaScript Parser

- SpiderMonkey
  - estree
    - esprima
    - acorn
      - babel parser

4.2 AST节点

- estree (https://github.com/estree/estree)
- spec.md (https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md)
- astexplorer (https://astexplorer.net/)
- AST节点
  - File 文件
  - Program 程序
  - Literal 字面量 NumericLiteral StringLiteral BooleanLiteral
  - Identifier 标识符
  - Statement 语句
  - Declaration 声明语句
  - Expression 表达式
  - Class 类

4.3 AST遍历

- astexplorer (https://astexplorer.net/)
- AST是深度优先遍历

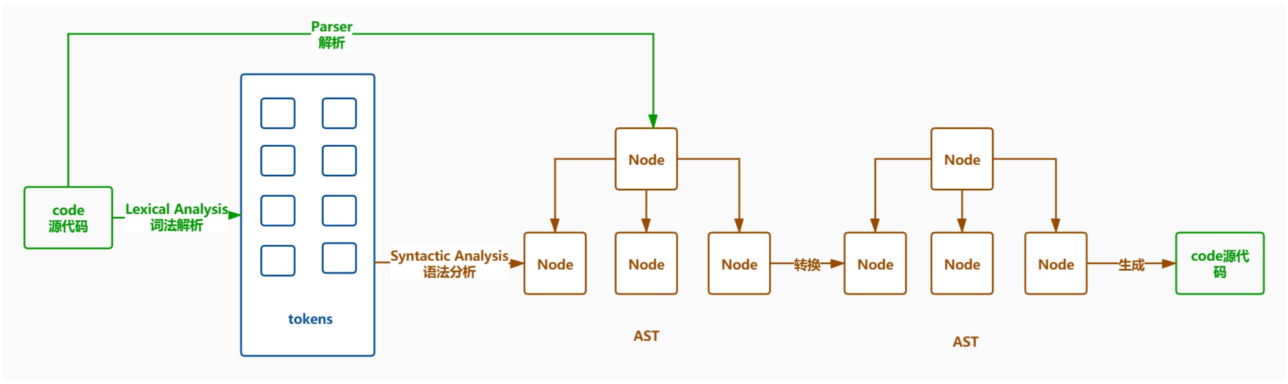
```
npm i esprima estraverse escodegen -S
```

```
let esprima = require('esprima');
let estraverse = require('estraverse');
let escodegen = require('escodegen');
let code = `function ast(){}`;
let ast = esprima.parse(code);
let indent = 0;
const padding = ()=>" ".repeat(indent);
estraverse.traverse(ast,{
    enter(node){
        console.log(padding()+node.type+'进入');
        if(node.type === 'FunctionDeclaration'){
            node.id.name = 'newAst';
        }
        indent+=2;
    },
    leave(node){
        indent-=2;
        console.log(padding()+node.type+'离开');
    }
});
```

```
Program&#x8FDB;&#x5165;
  FunctionDeclaration&#x8FDB;&#x5165;
    Identifier&#x8FDB;&#x5165;
    Identifier&#x79BB;&#x5F00;
    BlockStatement&#x8FDB;&#x5165;
    BlockStatement&#x79BB;&#x5F00;
  FunctionDeclaration&#x79BB;&#x5F00;
Program&#x79BB;&#x5F00;
```

5.babel

- Babel 能够转译 `ECMAScript 2015+` 的代码，使它在旧的浏览器或者环境中也能够运行
- 工作过程分为三个部分
  - Parse(解析) 将源代码转换成抽象语法树，树上有很多的estree节点 (https://github.com/estree/estree)
  - Transform(转换) 对抽象语法树进行转换
  - Generate(代码生成) 将上一步经过转换过的抽象语法树生成新的代码



5.2 babel 插件

- @babel/parser (https://github.com/babel/babel/tree/master/packages/@babel/parser) 可以把源码转换成AST
- @babel/traverse (https://www.npmjs.com/package/babel-traverse)用于对 AST 的遍历，维护了整棵树的状态，并且负责替换、移除和添加节点
- @babel/generate (https://github.com/babel/babel/tree/master/packages/@babel/generate) 可以把AST生成源码，同时生成sourcemap
- @babel/types (https://github.com/babel/babel/tree/master/packages/babel-types) 用于 AST 节点的 Lodash 式工具库, 它包含了构造、验证以及变换 AST 节点的方法，对编写处理 AST 逻辑非常有用
- @babel/template (https://www.npmjs.com/package/@babel/template)可以简化AST的创建逻辑
- @babel/code-frame (https://www.npmjs.com/package/@babel/code-frame)可以打印代码位置
- @babel/core (https://www.npmjs.com/package/@babel/core) Babel 的编译器，核心 API 都在这里面，比如常见的 transform、parse,并实现了插件功能
- babylon (https://www.npmjs.com/package/babylon) Babel 的解析器，以前叫babel parser,是基于acorn扩展而来，扩展了很多语法,可以支持es2020、jsx、typescript等语法
- babel-types-api (https://babeljs.io/docs/en/next/babel-types.html)
- Babel 插件手册 (https://github.com/brigand/babel-plugin-handbook/blob/master/translations/zh-Hans/README.md#asts)
- babeljs.io (https://babeljs.io/en/repl.html) babel 可视化编译器
- babel-types (https://babeljs.io/docs/en/babel-types)
- 类型别名 (https://github.com/babel/babel/blob/main/packages/babel-types/src/ast-types/generated/index.ts#L2489-L2535)
- DefinitelyTyped (https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types)

5.3 Visitor

- 访问者模式 Visitor 对于某个对象或者一组对象，不同的访问者，产生的结果不同，执行操作也不同
- Visitor 的对象定义了用于 AST 中获取具体节点的方法
- Visitor 上挂载以节点 `type` 命名的方法，当遍历 AST 的时候，如果匹配上 type，就会执行对应的方法

5.3.1 path

- path (https://github.com/babel/babel/blob/main/packages/babel-traverse/src/path/index.ts)
- node 当前 AST 节点
- parent 父 AST 节点
- parentPath 父AST节点的路径
- scope 作用域
- get(key) 获取某个属性的 path
- set(key, node) 设置某个属性
- is类型(opts) 判断当前节点是否是某个类型
- find(callback) 从当前节点一直向上找到根节点(包括自己)
- findParent(callback)从当前节点一直向上找到根节点(不包括自己)
- insertBefore(nodes) 在之前插入节点
- insertAfter(nodes) 在之后插入节点
- replaceWith(replacement) 用某个节点替换当前节点
- replaceWithMultiple(nodes) 用多个节点替换当前节点
- replaceWithSourceString(replacement) 把源代码转成AST节点再替换当前节点
- remove() 删除当前节点
- traverse(visitor, state) 遍历当前节点的子节点,第1个参数是节点，第2个参数是用来传递数据的状态
- skip() 跳过当前节点子节点的遍历
- stop() 结束所有的遍历

5.3.2 scope

- scope (https://github.com/babel/babel/blob/main/packages/babel-traverse/src/scope/index.ts)
- scope.bindings 当前作用域内声明所有变量
- scope.path 生成作用域的节点对应的路径
- scope.references 所有的变量引用的路径
- getAllBindings() 获取从当前作用域一直到根作用域的集合
- getBinding(name) 从当前作用域到根使用域查找变量
- getOwnBinding(name) 在当前作用域查找变量
- parentHasBinding(name, noGlobals) 从当前父作用域到根使用域查找变量
- removeBinding(name) 删除变量
- hasBinding(name, noGlobals) 判断是否包含变量
- moveBindingTo(name, scope) 把当前作用域的变量移动到其它作用域中
- generateUid(name) 生成作用域中的唯一变量名,如果变量名被占用就在前面加下划线

5.4 转换箭头函数

- astexplorer (https://astexplorer.net/)
- babel-plugin-transform-es2015-arrow-functions (https://www.npmjs.com/package/babel-plugin-transform-es2015-arrow-functions)
- babeljs.io (https://babeljs.io/en/repl.html) babel 可视化编译器
- babel-handbook (https://github.com/jamiebuilds/babel-handbook/blob/master/translations/zh-Hans/README.md)
- babel-types-api (https://babeljs.io/docs/en/next/babel-types.html)

转换前

```
const sum = (a,b)=>{
    console.log(this);
    return a+b;
}
```

转换后

```
var _this = this;

const sum = function (a, b) {
  console.log(_this);
  return a + b;
};
```

```
npm i @babel/core @babel/types -D
```

实现

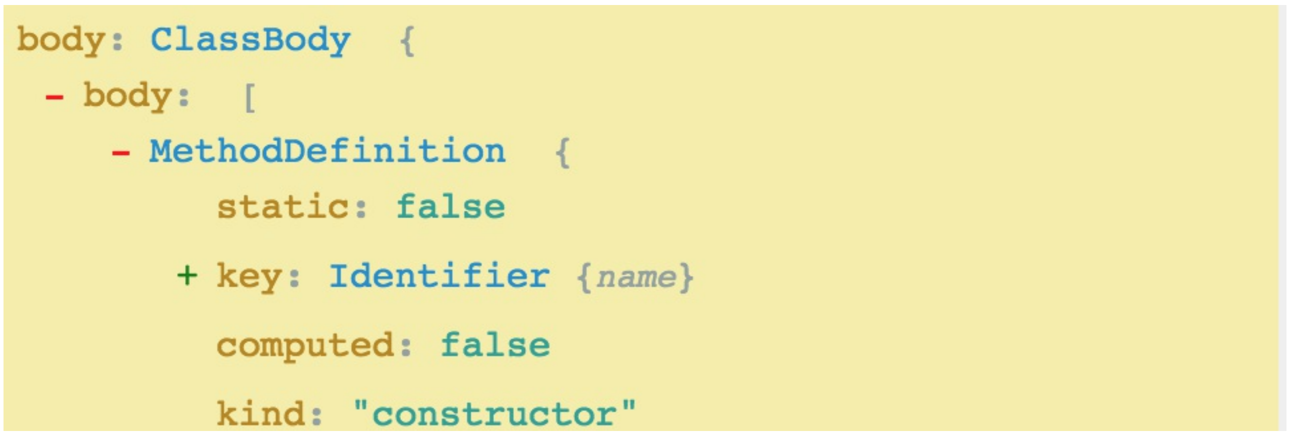
```
const core = require('@babel/core');

let types = require("@babel/types");

let arrowFunctionPlugin = {
    visitor: {

        ArrowFunctionExpression(path) {
            let { node } = path;
            hoistFunctionEnvironment(path);
            node.type = 'FunctionExpression';
            let body = node.body;

            if (!types.isBlockStatement(body)) {
                node.body = types.blockStatement([types.returnStatement(body)]);
            }
        }
    }
}

function hoistFunctionEnvironment(path) {

    const thisEnv = path.findParent(parent => {
        return (parent.isFunction() && !path.isArrowFunctionExpression()) || parent.isProgram();
    });
    let thisBindings = '_this';
    let thisPaths = getThisPaths(path);
    if (thisPaths.length>0) {

        if (!thisEnv.scope.hasBinding(thisBindings)) {
            thisEnv.scope.push({
                id: types.identifier(thisBindings),
                init: types.thisExpression()
            });
        }
    }
    thisPaths.forEach(thisPath => {

        thisPath.replaceWith(types.identifier(thisBindings));
    });
}
function getThisPaths(path){
    let thisPaths = [];
    path.traverse({
        ThisExpression(path) {
            thisPaths.push(path);
        }
    });
    return thisPaths;
}
let sourceCode = `
const sum = (a, b) => {
    console.log(this);
    const minus = (c,d)=>{
          console.log(this);
        return c-d;
    }
    return a + b;
}
`;
let targetSource = core.transform(sourceCode, {
    plugins: [arrowFunctionPlugin]
});

console.log(targetSource.code);
```

5.5 把类编译为 Function #

- @babel/plugin-transform-classes (https://www.npmjs.com/package/@babel/plugin-transform-classes)

es6

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

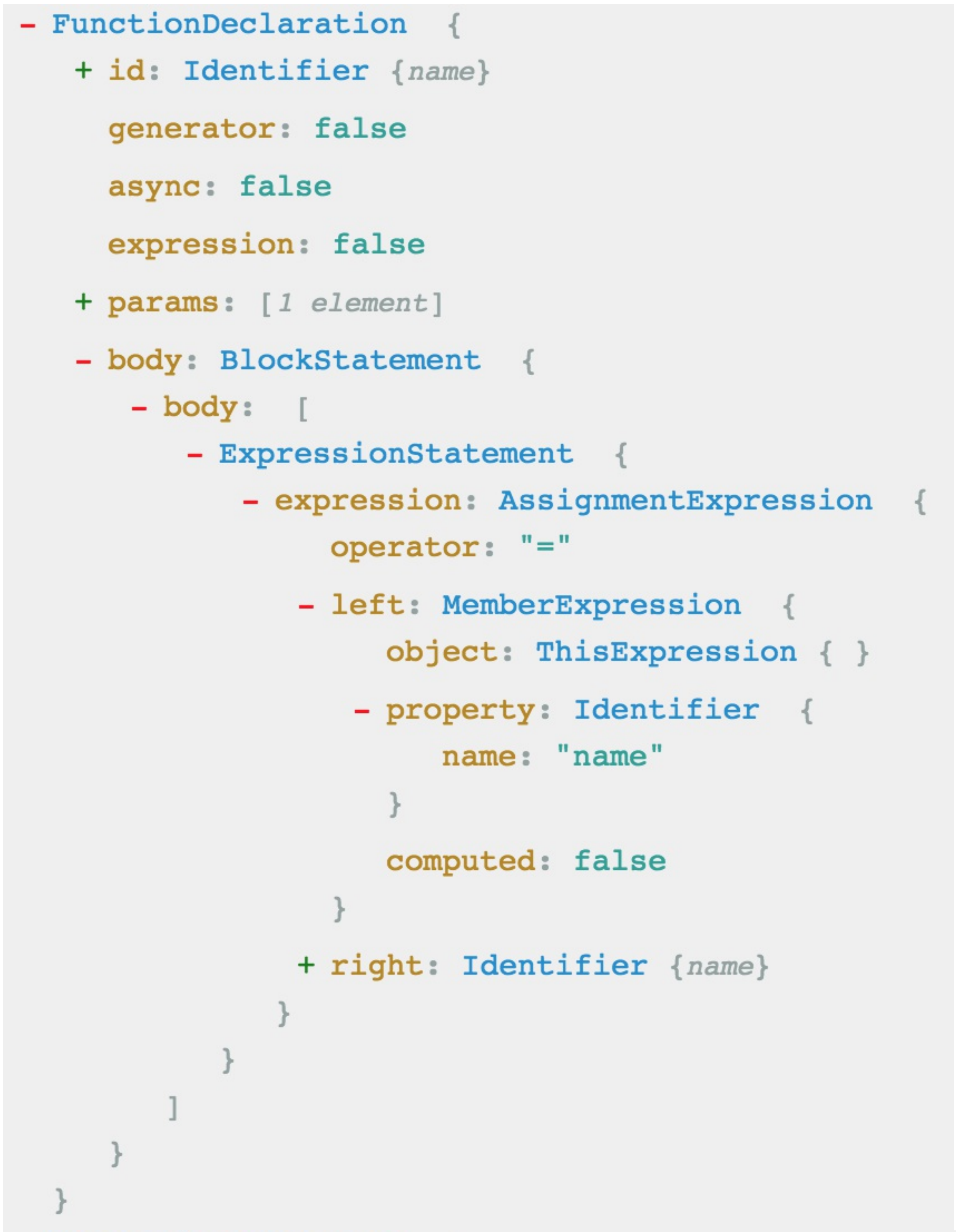


```
body: ClassBody  {
  - body:  [
      - MethodDefinition  {
            static: false
          + key: Identifier {name}
            computed: false
            kind: "constructor"
```

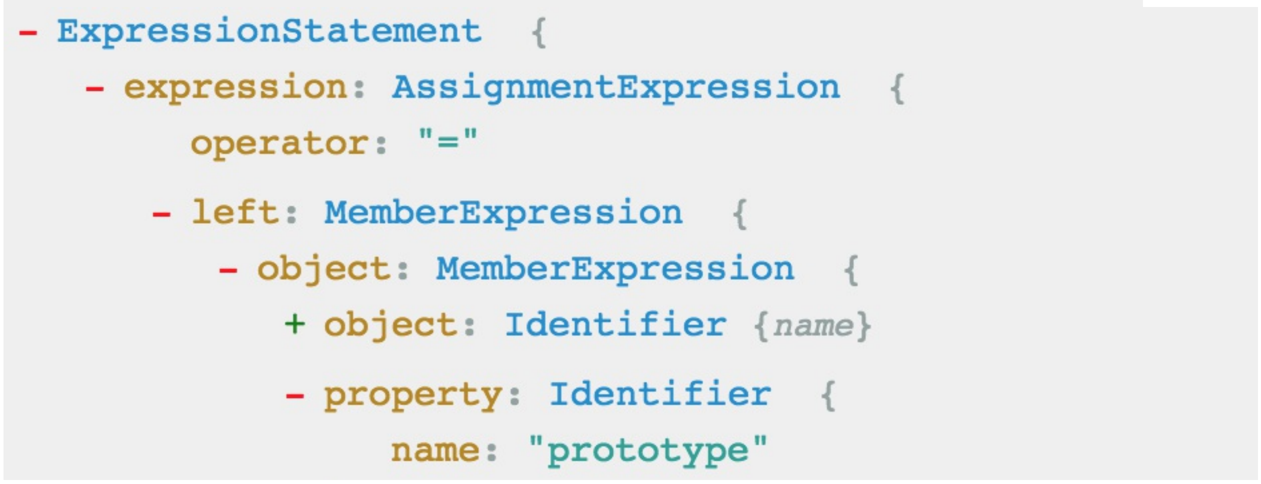
```
       - value: FunctionExpression = $node {
            generator: false
            async: false
            expression: false
          + params: [1 element]
          - body: BlockStatement {
             - body: [
                + ExpressionStatement {expression}
               ]
            }
         }
      }
    - MethodDefinition {
         static: false
       + key: Identifier {name}
         computed: false
         kind: "method"
       - value: FunctionExpression {
            generator: false
            async: false
            expression: false
            params: [ ]
          - body: BlockStatement {
             - body: [
                + ReturnStatement {argument}
               ]
            }
         }
      }
   ]
}
```

es5

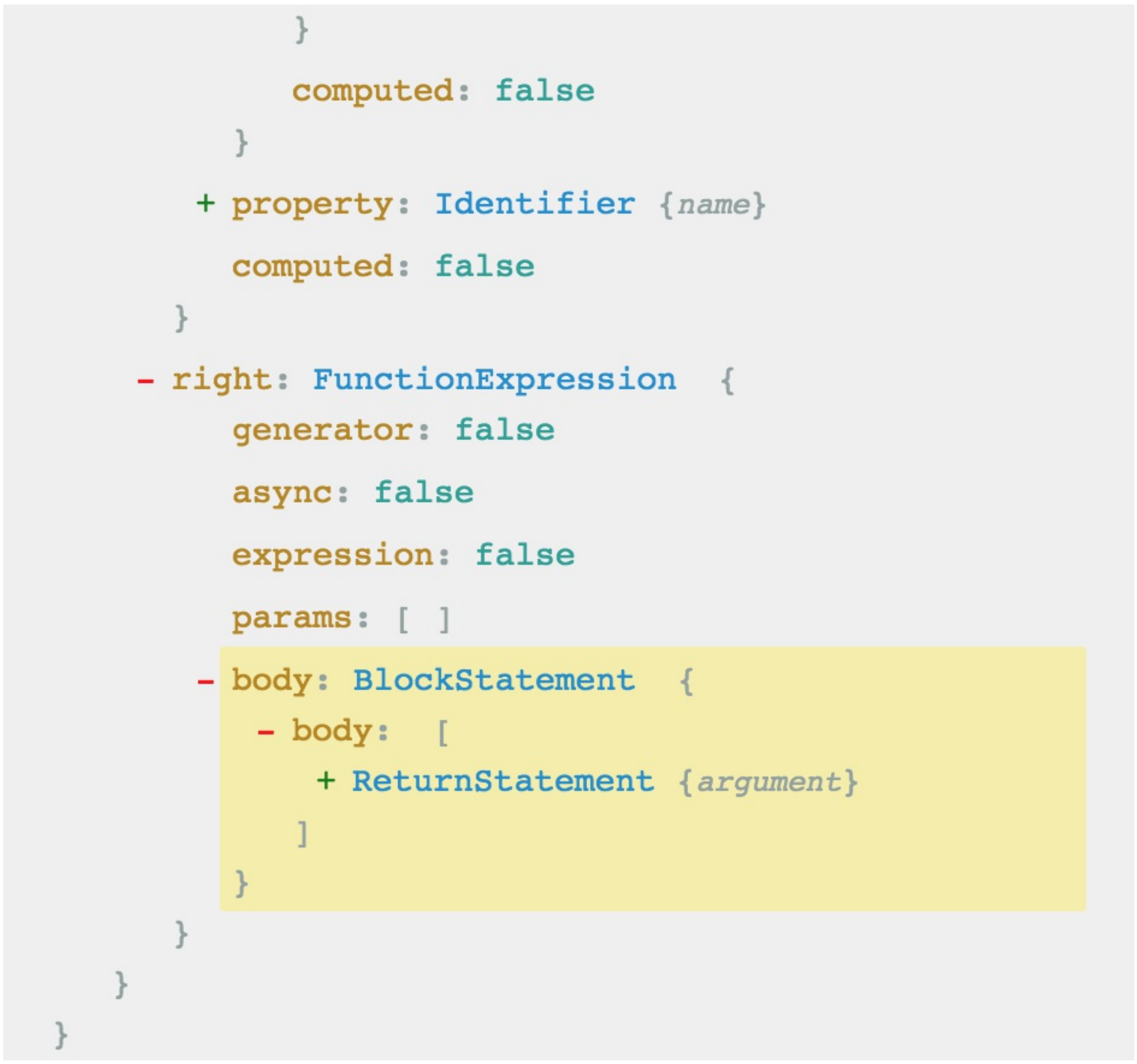
```
function Person(name) {
  this.name = name;
}
Person.prototype.getName = function () {
  return this.name;
};
```



```
- FunctionDeclaration  {
    + id: Identifier {name}
      generator: false
      async: false
      expression: false
    + params: [1 element]
    - body: BlockStatement  {
        - body:  [
            - ExpressionStatement  {
                - expression: AssignmentExpression  {
                      operator: "="
                    - left: MemberExpression  {
                          object: ThisExpression { }
                        - property: Identifier  {
                              name: "name"
                          }
                          computed: false
                      }
                    + right: Identifier {name}
                  }
              }
          ]
      }
  }
```



```
- ExpressionStatement  {
    - expression: AssignmentExpression  {
          operator: "="
        - left: MemberExpression  {
            - object: MemberExpression  {
                + object: Identifier {name}
                - property: Identifier  {
                      name: "prototype"
```



```
                  }
                  computed: false
              }
            + property: Identifier {name}
              computed: false
          }
      - right: FunctionExpression  {
              generator: false
              async: false
              expression: false
              params: [ ]
            - body: BlockStatement  {
                - body:  [
                    + ReturnStatement {argument}
                  ]
              }
          }
      }
  }
```

实现

```
const core = require('@babel/core');

let types = require("@babel/types");

let transformClassesPlugin = {
    visitor: {

        ClassDeclaration(path) {
            let node = path.node;
            let id = node.id;
            let methods = node.body.body;
            let nodes = [];
            methods.forEach(method => {
                if (method.kind === 'constructor') {
                    let constructorFunction = types.functionDeclaration(
                        id,
                        method.params,
                        method.body
                    );
                    nodes.push(constructorFunction);
                } else {
                    let memberExpression = types.memberExpression(
                        types.memberExpression(
                            id, types.identifier('prototype')
                        ), method.key
                    )
                    let functionExpression = types.functionExpression(
                        null,
                        method.params,
                        method.body
                    )
                    let assignmentExpression = types.assignmentExpression(
                        '=',
                        memberExpression,
                        functionExpression
                    );
                    nodes.push(assignmentExpression);
                }
            })
            if (nodes.length === 1) {

                path.replaceWith(nodes[0]);
            } else {

                path.replaceWithMultiple(nodes);
            }
        }
    }
}
let sourceCode = `
class Person{
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}
`;
let targetSource = core.transform(sourceCode, {
    plugins: [transformClassesPlugin]
});

console.log(targetSource.code);
```

5.6 实现日志插件

5.6.1 logger.js

```
const core = require('@babel/core');

const types = require("@babel/types");
const path = require('path');
const visitor = {
    CallExpression(nodePath, state) {
        const { node } = nodePath;
        if (types.isMemberExpression(node.callee)) {
            if (node.callee.object.name === 'console') {
                if (['log', 'info', 'warn', 'error', 'debug'].includes(node.callee.property.name)) {
                    const { line, column } = node.loc.start;
                    const relativeFileName = path.relative(__dirname, state.file.opts.filename).replace(/\\/g, '/');
                    node.arguments.unshift(types.stringLiteral(`${relativeFileName} ${line}:${column}`));
                }
            }
        }
    }
}
module.exports = function () {
    return {
        visitor
    }
}
```

5.7 自动日志插件

- babel-helper-plugin-utils (https://babeljs.io/docs/en/babel-helper-plugin-utils)
- babel-types (https://babeljs.io/docs/en/babel-types.html#api)用来生成节点和判断节点类型
- babel-helper-module-imports (https://babeljs.io/docs/en/babel-helper-module-imports)帮助插入模块
- @babel/template (https://www.npmjs.com/package/@babel/template)根据字符串模板生成AST节点
- `state` 用于在遍历过程中在AST节点之间传递数据的方式

5.7.1 use.js

```
const { transformSync } = require('@babel/core');
const autoLoggerPlugin = require('./auto-logger-plugin');
const sourceCode = `
function sum(a,b){return a+b;}
const multiply = function(a,b){return a*b;};
const minus = (a,b)=>a-b
class Calculator{divide(a,b){return a/b}}
`
const { code } = transformSync(sourceCode, {
  plugins: [autoLoggerPlugin({ libName: 'logger' })]
});
console.log(code);
```

5.7.2 auto-logger-plugin

```
const importModule = require('@babel/helper-module-imports');
const template = require('@babel/template');
const types = require('@babel/types');
const autoLoggerPlugin = (options) => {
    return {
        visitor: {
            Program: {
                enter(path, state) {
                    let loggerId;
                    path.traverse({
                        ImportDeclaration(path) {
                            const libName = path.get('source').node.value;
                            if (libName === options.libName) {
                                const specifierPath = path.get('specifiers.0');

                                if (specifierPath.isImportDefaultSpecifier()
                                    || specifierPath.isImportSpecifier()
                                    || specifierPath.isImportNamespaceSpecifier()) {
                                    loggerId = specifierPath.local.name;
                                }
                                path.stop();
                            }
                        }
                    });
                    if (!loggerId) {
                        loggerId = importModule.addDefault(path, 'logger', {
                            nameHint: path.scope.generateUid('logger')
                        }).name;
                    }

                    state.loggerNode = template.statement(`LOGGER();`)({
                        LOGGER: loggerId
                    });
                }
            },
            'FunctionExpression|FunctionDeclaration|ArrowFunctionExpression|ClassMethod'(path, state) {
                const { node } = path
                if (types.isBlockStatement(node.body)) {
                    node.body.body.unshift(state.loggerNode);
                } else {
                    const newNode = types.blockStatement([
                        state.loggerNode,
                        types.expressionStatement(node.body)
                    ]);
                    path.get('body').replaceWith(newNode);
                }
            }
        }
    }
};
module.exports = autoLoggerPlugin;
```

5.8 eslint

- rules (//https://eslint.bootcss.com/docs/rules/)

5.8.1 use.js

```
const { transformSync } = require('@babel/core');
const eslintPlugin = require('./eslintPlugin');
const sourceCode = `
var a = 1;
console.log(a);
var b = 2;
`;
const { code } = transformSync(sourceCode, {
  plugins: [eslintPlugin({ fix: true })]
});
console.log(code);
```

5.8.2 eslintPlugin.js

eslintPlugin.js

```
const eslintPlugin = ({ fix }) => {
  return {
    pre(file) {
      file.set('errors', []);
    },
    visitor: {
      CallExpression(path, state) {
        const errors = state.file.get('errors');
        const { node } = path
        if (node.callee.object && node.callee.object.name === 'console') {

          errors.push(path.buildCodeFrameError(`代码中不能出现console语句`, Error));

          if (fix) {
            path.parentPath.remove();
          }
        }
      }
    },
    post(file) {
      console.log(...file.get('errors'));
    }
  }
};
module.exports = eslintPlugin;
```

5.9 uglify

5.9.1 use.js

```
const { transformSync } = require('@babel/core');
const uglifyPlugin = require('./uglifyPlugin');
const sourceCode = `
function getAge(){
  var age = 12;
  console.log(age);
  var name = '';
  console.log(name);
}
`;
const { code } = transformSync(sourceCode, {
  plugins: [uglifyPlugin()]
});
console.log(code);
```

5.9.2 uglifyPlugin.js

-
    - 类型别名 (https://github.com/babel/babel/blob/main/packages/babel-types/src/ast-types/generated/index.ts#L2174-L2191)

uglifyPlugin.js

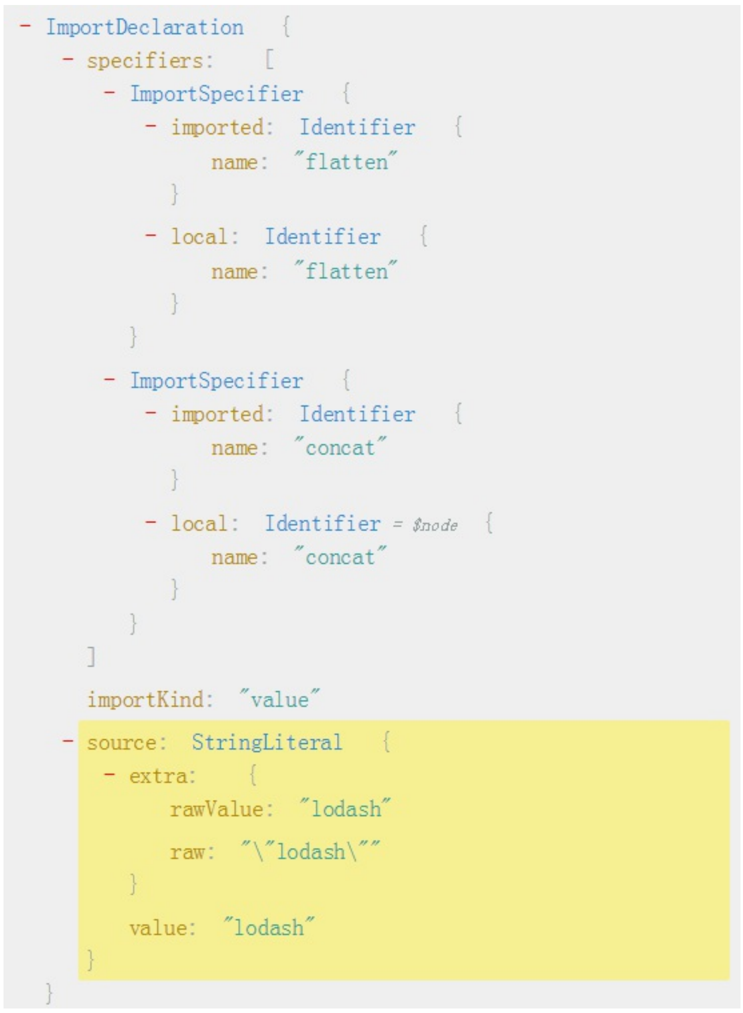
```
const uglifyPlugin = () => {
  return {
    visitor: {
      Scopable(path) {
        Object.entries(path.scope.bindings).forEach(([key, binding]) => {
          const newName = path.scope.generateUid();
          binding.path.scope.rename(key, newName)
        });
      }
    }
  }
};
module.exports = uglifyPlugin;
```

6. webpack中使用babel插件

6.1 实现按需加载

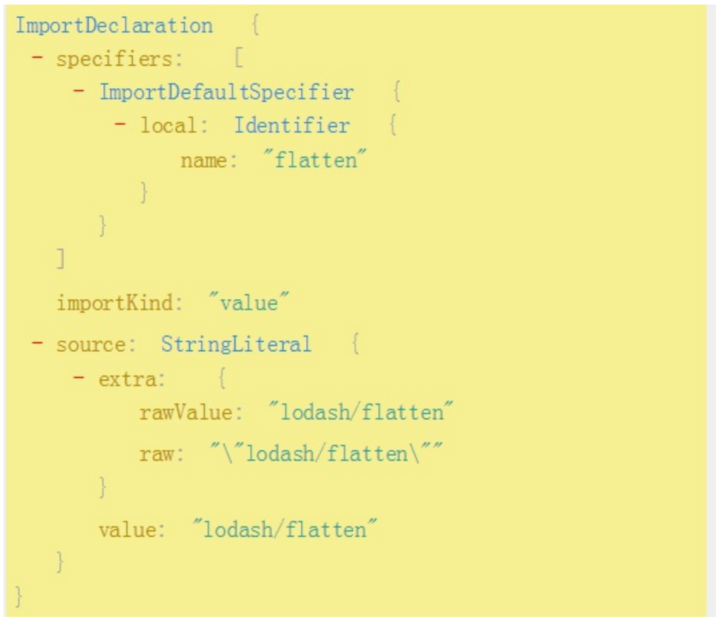
- lodashjs (https://www.lodashjs.com/docs/4.17.5.html#concat)
- babel-core (https://babeljs.io/docs/en/babel-core)
- babel-plugin-import (https://www.npmjs.com/package/babel-plugin-import)

```
import { flatten, concat } from "lodash";
```


```
- ImportDeclaration   {
    - specifiers:   [
        - ImportSpecifier   {
            - imported:  Identifier   {
                  name:  "flatten"
              }
            - local:  Identifier   {
                  name:  "flatten"
              }
          }
        - ImportSpecifier   {
            - imported:  Identifier   {
                  name:  "concat"
              }
            - local:  Identifier = $node   {
                  name:  "concat"
              }
          }
      ]
      importKind:  "value"
    - source:  StringLiteral   {
        - extra:   {
              rawValue:  "lodash"
              raw:  "\"lodash\""
          }
          value:  "lodash"
      }
  }
```

转换为

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


```
ImportDeclaration   {
  - specifiers:   [
      - ImportDefaultSpecifier   {
          - local:  Identifier   {
                name:  "flatten"
            }
        }
    ]
    importKind:  "value"
  - source:  StringLiteral   {
      - extra:   {
            rawValue:  "lodash/flatten"
            raw:  "\"lodash/flatten\""
        }
        value:  "lodash/flatten"
    }
}
```

6.1.1 webpack 配置 #

```
npm i webpack webpack-cli babel-plugin-import -D
```

```
const path = require("path");
module.exports = {
  mode: "development",
  entry: "./src/index.js",
  output: {
    path: path.resolve("dist"),
    filename: "bundle.js",
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: {
          loader: "babel-loader",
          options:{
                   plugins:[
                     [
                       path.resolve(__dirname,'plugins/babel-plugin-import.js'),
                       {
                         libraryName:'lodash'
                       }
                     ]
                   ]
                }
        },
      },
    ],
  },
};
```

```
编译顺序为首先 plugins从左往右,然后 presets从右往左
```

6.1.2 babel 插件

plugins\babel-plugin-import.js

```

const core = require('@babel/core');

let types = require("@babel/types");

const visitor = {
    ImportDeclaration(path, state) {
        const { node } = path;
        const { specifiers } = node;
        const { libraryName, libraryDirectory = 'lib' } = state.opts;

        if (node.source.value === libraryName

            && !types.isImportDefaultSpecifier(specifiers[0])) {

            const declarations = specifiers.map(specifier => {

                return types.importDeclaration(

                    [types.importDefaultSpecifier(specifier.local)],

                    types.stringLiteral(libraryDirectory ? `${libraryName}/${libraryDirectory}/${specifier.imported.name}` :
`${libraryName}/${specifier.imported.name}`)
                );
            })
            path.replaceWithMultiple(declarations);
        }
    }
}

module.exports = function () {
    return {
        visitor
    }
}
```

7. 参考

- Babel 插件手册 (https://github.com/brigand/babel-plugin-handbook/blob/master/translations/zh-Hans/README.md#asts)
- babel-types (https://github.com/babel/babel/tree/master/packages/babel-types)
- 不同的 parser 解析 js 代码后得到的 AST (https://astexplorer.net/)
- 在线可视化的看到 AST (http://resources.jointjs.com/demos/javascript-ast)
- babel 从入门到入门的知识归纳 (https://zhuanlan.zhihu.com/p/28143410)
- Babel 内部原理分析 (https://octman.com/blog/2016-08-27-babel-notes/)
- babel-plugin-react-scope-binding (https://github.com/chikara-chan/babel-plugin-react-scope-binding)
- transform-runtime (https://www.npmjs.com/package/babel-plugin-transform-runtime) Babel 默认只转换新的 JavaScript 语法，而不转换新的 API。例如，Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise 等全局对象，以及一些定义在全局对象上的方法（比如 Object.assign）都不会转译,启用插件 `babel-plugin-transform-runtime` 后，Babel 就会使用 babel-runtime 下的工具函数
- ast-spec (https://github.com/babel/babylon/blob/master/ast/spec.md)
- babel-handbook (https://github.com/jamiebuilds/babel-handbook/blob/master/translations/zh-Hans/README.md)

5.9 tsc

5.9.1 use.js

```
const { transformSync } = require('@babel/core');
const tscCheckPlugin = require('./tscCheckPlugin');
const sourceCode = `
var age:number="12";
`;

const { code } = transformSync(sourceCode, {
  parserOpts: { plugins: ['typescript'] },
  plugins: [tscCheckPlugin()]
});

console.log(code);
```

5.9.2 tscCheckPlugin.js

tscCheckPlugin.js

```
const TypeAnnotationMap = {
  TSNumberKeyword: "NumericLiteral"
}
const eslintPlugin = () => {
  return {
    pre(file) {
      file.set('errors', []);
    },
    visitor: {
      VariableDeclarator(path, state) {
        const errors = state.file.get('errors');
        const { node } = path;
        const idType = TypeAnnotationMap[node.id.typeAnnotation.typeAnnotation.type];
        const initType = node.init.type;
        console.log(idType, initType);
        if (idType !== initType) {
          errors.push(path.get('init').buildCodeFrameError(`无法把${initType}类型赋值给${idType}类型`, Error));
        }
      }
    },
    post(file) {
      console.log(...file.get('errors'));
    }
  }
};
module.exports = eslintPlugin;
```

5.9.3 赋值

```
const babel = require('@babel/core');
function transformType(type){
    switch(type){
        case 'TSNumberKeyword':
        case 'NumberTypeAnnotation':
            return 'number'
        case 'TSStringKeyword':
        case 'StringTypeAnnotation':
            return 'string'
    }
}
const tscCheckPlugin = () => {
    return {
        pre(file) {
            file.set('errors', []);
        },
        visitor: {
            AssignmentExpression(path,state){
              const errors = state.file.get('errors');
              const variable = path.scope.getBinding(path.get('left'));
              const variableAnnotation = variable.path.get('id').getTypeAnnotation();
              const variableType = transformType(variableAnnotation.typeAnnotation.type);
              const valueType = transformType(path.get('right').getTypeAnnotation().type);
              if (variableType !== valueType){
                  Error.stackTraceLimit = 0;
                  errors.push(
                      path.get('init').buildCodeFrameError(`无法把${valueType}赋值给${variableType}`, Error)
                  );
              }
            }
        },
        post(file) {
            console.log(...file.get('errors'));
        }
    }
}

let sourceCode = `
  var age:number;
  age = "12";
`;

const result = babel.transform(sourceCode, {
    parserOpts:{plugins:['typescript']},
    plugins: [tscCheckPlugin()]
})
console.log(result.code);
```

5.9.4 泛型

```
const babel = require('@babel/core');
function transformType(type){
    switch(type){
        case 'TSNumberKeyword':
        case 'NumberTypeAnnotation':
            return 'number'
        case 'TSStringKeyword':
        case 'StringTypeAnnotation':
            return 'string'
    }
}
const tscCheckPlugin = () => {
    return {
        pre(file) {
            file.set('errors', []);
        },
        visitor: {
            CallExpression(path,state){
              const errors = state.file.get('errors');
              const trueTypes = path.node.typeParameters.params.map(param=>transformType(param.type));
              const argumentsTypes = path.get('arguments').map(arg=>transformType(arg.getTypeAnnotation().type));
              const calleePath = path.scope.getBinding(path.get('callee').node.name).path;
              const genericMap=new Map();
              calleePath.node.typeParameters.params.map((item, index) => {
                genericMap[item.name] = trueTypes[index];
              });
              const paramsTypes =  calleePath.get('params').map(arg=>{
                const typeAnnotation = arg.getTypeAnnotation().typeAnnotation;
                if(typeAnnotation.type === 'TSTypeReference'){
                    return genericMap[typeAnnotation.typeName.name];
                }else{
                    return transformType(type);
                }
              });
              Error.stackTraceLimit = 0;
              paramsTypes.forEach((type,index)=>{
                  console.log(type,argumentsTypes[index]);
                if(type !== argumentsTypes[index]){
                    errors.push(
                        path.get(`arguments.${index}`).buildCodeFrameError(`实参${argumentsTypes[index]}不能匹配形参${type}`, Error)
                    );
                }
              });
            }
        },
        post(file) {
            console.log(...file.get('errors'));
        }
    }
}

let sourceCode = `
  function join(a:T,b:T):string{
      return a+b;
  }
  join(1,'2');
`;

const result = babel.transform(sourceCode, {
    parserOpts:{plugins:['typescript']},
    plugins: [tscCheckPlugin()]
})
console.log(result.code);
```

5.9.5 类型别名

```
const babel = require('@babel/core');
function transformType(type){
    switch(type){
        case 'TSNumberKeyword':
        case 'NumberTypeAnnotation':
            return 'number'
        case 'TSStringKeyword':
        case 'StringTypeAnnotation':
            return 'string'
        case 'TSLiteralType':
            return 'liternal';
        default:
        return type;
    }
}
const tscCheckPlugin = () => {
    return {
        pre(file) {
            file.set('errors', []);
        },
        visitor: {
            TSTypeAliasDeclaration(path){
                const typeName  = path.node.id.name;
                const typeInfo = {
                    typeParams:path.node.typeParameters.params.map(item =>item.name),
                    typeAnnotation:path.getTypeAnnotation()
                }
                path.scope.setData(typeName,typeInfo)
            },
            CallExpression(path,state){
              const errors = state.file.get('errors');
              const trueTypes = path.node.typeParameters.params.map(param=>{

                if(param.type === 'TSTypeReference'){
                    const name = param.typeName.name;
                    const {typeParams,typeAnnotation} = path.scope.getData(name);
                    const trueTypeParams = typeParams.reduce((memo, name, index) => {
                        memo[name] = param.typeParameters.params[index].type;
                        return memo;
                    },{});
                    const {checkType,extendsType,trueType,falseType} = typeAnnotation;
                    let check=checkType.type;
                    if(check === 'TSTypeReference'){
                        check = trueTypeParams[checkType.typeName.name]
                    }
                    if (transformType(check) === transformType(extendsType.type)) {
                        return transformType(trueType.type);
                    } else {
                        return transformType(falseType.type);
                    }
                }else{
                    return  transformType(param.type);
                }
              });
              const argumentsTypes = path.get('arguments').map(arg=>transformType(arg.getTypeAnnotation().type));
              const calleePath = path.scope.getBinding(path.get('callee').node.name).path;
              const genericMap=new Map();
              calleePath.node.typeParameters.params.map((item, index) => {
                genericMap[item.name] = trueTypes[index];
              });
              const paramsTypes =  calleePath.get('params').map(arg=>{
                const typeAnnotation = arg.getTypeAnnotation().typeAnnotation;
                if(typeAnnotation.type === 'TSTypeReference'){
                    return genericMap[typeAnnotation.typeName.name];
                }else{
                    return transformType(type);
                }
              });
              Error.stackTraceLimit = 0;
              paramsTypes.forEach((type,index)=>{
                if(type !== argumentsTypes[index]){
                    errors.push(
                        path.get(`arguments.${index}`).buildCodeFrameError(`实参${argumentsTypes[index]}不能匹配形参${type}`, Error)
                    );
                }
              });
            }
        },
        post(file) {
            console.log(...file.get('errors'));
        }
    }
}

let sourceCode = `
    type Infer = K extends 'number' ? number : string;
    function sum(a: T, b: T) {

    }
    sum>(1, 2);
`;

const result = babel.transform(sourceCode, {
    parserOpts:{plugins:['typescript']},
    plugins: [tscCheckPlugin()]
})
console.log(result.code);
```