

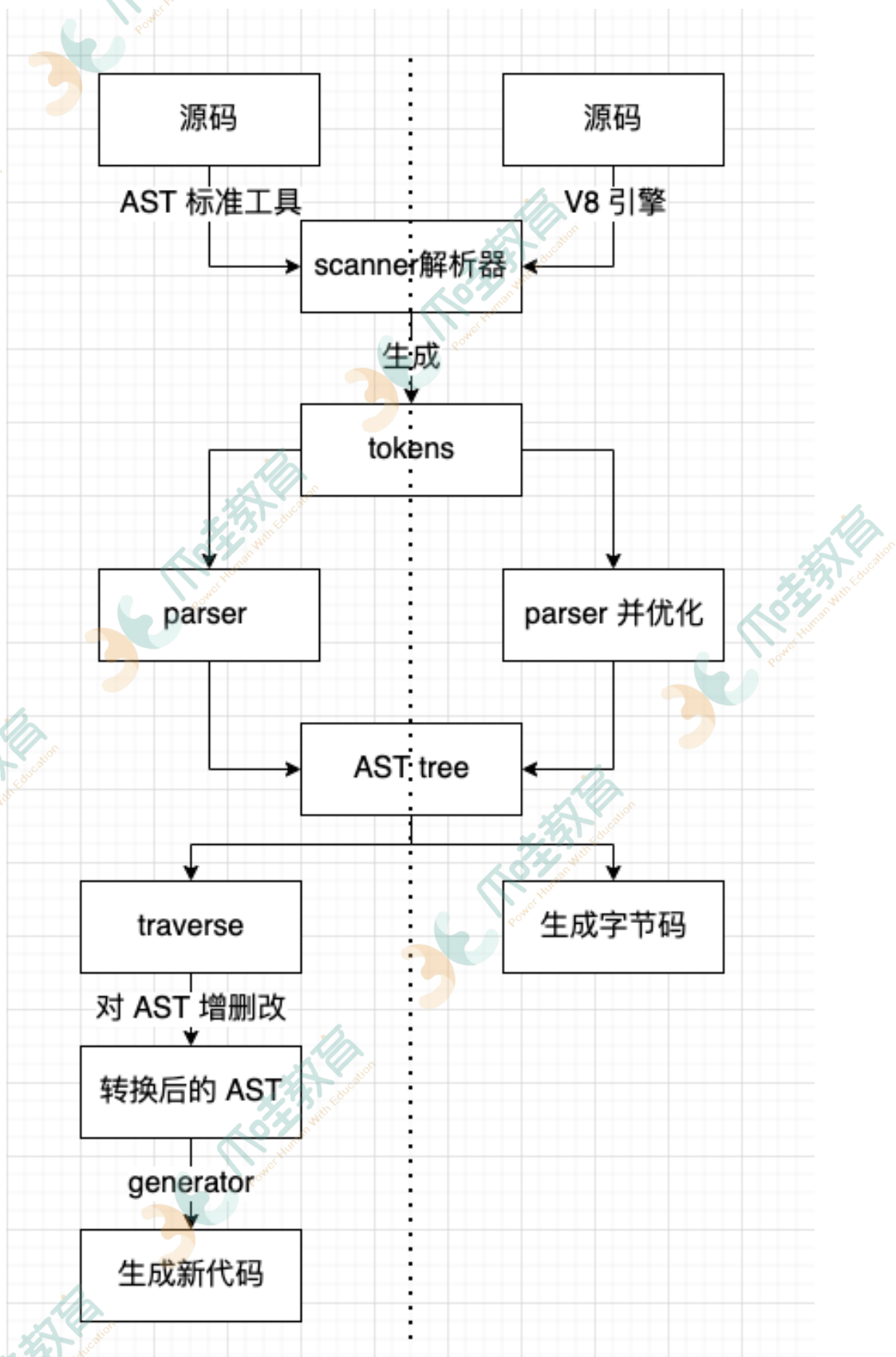
AST 简介

抽象语法树（abstract syntax trees），就是将代码转换成的一种抽象的树形结构，通常是 json 描述。AST 并不是哪个编程语言特有的概念，在前端领域，比较常用的 AST 工具如 [esprima](#)，[babel](#)（[babylon](#)）的解析模块，其他如 vue 自己实现的模板解析器。本文主要以 babel 为例对 AST 的原理进行浅析，通过实践掌握如何利用 AST 掌握代码转换的能力。

- 推荐工具：[AST 在线学习](#)和 [tokens 在线分析](#)。
- 插件集合（注意，前端 AST 并不仅针对 JavaScript，CSS、HTML 一样具有相应的解析工具，JavaScript 重点关注）

ast 解析	esprima	@babel/parser	recast.parse	V8 引擎
ast 遍历	estaverse	@babel/traverse	recast.visit		
生成代码	escodegen	@babel/generator	recast.print recast.prettyPrint	生成字节码	

代码的编译流程



我们把上述过程分为三部分：解析（parse），转换（transform），生成（generate），其中 scanner 部分叫做词法（syntax）分析，parser 部分叫做语法（grammar）分析。显然，词法分析的结果是 tokens，语法分析得到的就是 AST。

例：

```
1 const esprima = require('esprima');
```

```

2
3 const code = 'const number = 10';
4 const token = esprima.tokenize(code);
5 console.log(token);
6
7 // 打印结果:
8 [
9   { type: 'Keyword', value: 'const' },
10  { type: 'Identifier', value: 'number' },
11  { type: 'Punctuator', value: '=' },
12  { type: 'Numeric', value: '10' }
13 ]

```

可见词法分析，旨在将源代码按照一定的分隔符（空格/tab/换行等）、注释进行分割，并将各个部分进行分类构造出一段 token 流。

```

1 const esprima = require('esprima');
2
3 const code = 'const number = 10';
4 const token = esprima.tokenize(code);
5 const ast = esprima.parse(code);
6 console.log(JSON.stringify(ast, null, '  '));
7
8 // 打印 AST:
9 {
10   "type": "Program",
11   "body": [{
12     "type": "VariableDeclaration",
13     "declarations": [{
14       "type": "VariableDeclarator",
15       "id": {
16         "type": "Identifier",
17         "name": "number",
18       },
19       "init": {
20         "type": "Literal",
21         "value": 10,
22         "raw": "10",
23       },
24     }],

```

```
25     "kind": "const"
26   }},
27   "sourceType": "script"
28 }
```

而语法分析，则基于 tokens 将源码语义化、结构化为一段 json 描述（AST）。反之，如果给出一段代码的描述信息，我们也是可以还原源码的。理论上，描述信息发生变化，生成的源码的对应信息也会发生变化。所以我们可以通过操作 AST 达到修改源码信息的目的，辅以文件的创建接口，这也是 babel 打包生成代码的基本原理。

了解到这一层，便能想象 ES6 => ES5、ts => js、uglifyJS、样式预处理器、eslint、代码提示等工具的工作方式了。

AST 的节点类型

在操作 AST 过程中，源码部分集中在 Program 对象的 body 属性下，每个节点有着统一固定的格式：

`@babel/core` 依赖了 `parser`、`traverse`、`generator` 模块，所以安装 `@babel/core` 即可。下文均以 babel 作为示例工具，其他工具类似，不再赘述。

```
1  const babel = require('@babel/core');
2
3  const code = `
4    import React from 'react';
5    function add(a, b) {
6      return a + b;
7    }
8    let str = 'hello';
9  `;
10
11  const ast = babel.parse(code, {
12    sourceType: 'module'
13  });
14  console.log(ast.program.body);
15  // ast.program.body 部分
16  [{
17    ...
18    "type": "ImportDeclaration",
19    "specifiers": [{ type: "ImportDefaultSpecifier", ... }],
20  }, {
21    "type": "FunctionDeclaration",
22    "async": false,
```

```

23   "body": {
24     type: "BlockStatement",
25     "body": [{
26       "type": "ReturnStatement",
27       ...
28     }]
29   }
30 }, {
31   "type": "VariableDeclaration",
32   "kind": "let",
33   "declarations": [{ type: "VariableDeclarator", init: {}, ... }]
34 }]

```

JavaScript 生成的所有 AST 节点类型可[在线查阅](#)。知道了节点类型可以高效地进行节点查找可编辑。上面的代码为例，如果想在函数返回语句之前加入一行语句 `console.log('函数执行完成')`，最朴素的做法是这样：

before	after	
<pre> 1 // 其他语句略 2 function add(a, b) { 3 return a + b; 4 } </pre>	<pre> 1 function add(a, b) { 2 + console.log('函数执行完成'); 3 return a + b; 4 } </pre>	<p>已知 <code>console.log('函数执行完成');</code> 的 AST 用常量 <code>CONSOLE_AST</code> 表示。</p> <pre> 1 const CONSOLE_AST = babel. 2 template.ast(` 3 console.log('函数执行完成'); 4 `); </pre>

我们参照原先的 AST 结构很容易就能实现这个需求：

```

1 const babel = require('@babel/core');
2
3 const code = `
4   import React from 'react';
5   function add(a, b) {
6     return a + b;
7   }
8   let str = 'hello';
9 `;
10
11 const ast = babel.parse(code, {
12   sourceType: 'module'
13 });

```

```

14
15 function insertConsoleBeforeReturn(body) {
16   body.forEach(node => {
17     if (node.type === 'FunctionDeclaration') { // 函数关键字声明形式
18       const blockStatementBody = node.body.body;
19       if (blockStatementBody && blockStatementBody.length) {
20         const index = blockStatementBody.findIndex(n => n.type === 'ReturnStatement');
21         if (~index) {
22           // 函数体存在语句且最后一条语句是 return (假设 return 就是最后的语句)
23           blockStatementBody.splice(index, 0, CONSOLE_AST); // 直接修改 ast, 前插一个节
24         }
25       }
26     }
27   });
28 }
29 insertConsoleBeforeReturn(ast.program.body);

```

虽然手动操作 AST 满足了当前的需求，但是诸如箭头函数，类或对象的方法、没有 return 语句或省略 return 关键字的函数、表达式声明的函数、IIFE、语句内又嵌套的函数……上述方法都是没有考虑的，所以不推荐手动实现。

由此可见，处理 AST 的过程就是对不同节点类型遍历和操作的过程，为简化操作，babel 提供了专门的接口，我们只需要提供相应类型的处理方法（visitor）即可。还是上面的需求（好一点的是所有的 return 语句都会处理，即使是嵌套的函数）：

```

1  const babel = require('@babel/core');
2
3  const code = `
4    import React from 'react';
5    function add(a, b) {
6      return a + b;
7    }
8    let str = 'hello';
9  `;
10 const CONSOLE_AST = babel.template.ast(`console.log('函数执行完成');`);
11
12 const ast = babel.parse(code, {
13   sourceType: 'module'
14 });
15

```



```

16 babel.traverse(ast, {
17   ReturnStatement(path) {
18     path.insertBefore(CONSOLE_AST);
19   }
20 });
21
22 console.log(babel.transformFromAstSync(ast).code);

```

traverse 方法帮我们处理了 ast 的遍历过程，对于不同节点的处理只需要维护一份 types 对应的方法即可。进一步的，构造 `CONSOLE_AST` 节点也有几种方式。先使用在线工具将 `console.log('函数执行完成');` 结构化（如果你已经十分熟悉这个过程，可以跳过）：

The screenshot shows the AST Explorer interface. On the left, the input code is `console.log('函数执行完成');`. On the right, the JSON representation of the AST is displayed. The structure is as follows:

```

{
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "object": {
            "type": "Identifier",
            "name": "console"
          },
          "property": {
            "type": "Identifier",
            "name": "log"
          },
          "computed": false
        },
        "arguments": [
          {
            "type": "StringLiteral",
            "value": "函数执行完成",
            "raw": "'函数执行完成'"
          }
        ],
        "extra": {
          "rawValue": "函数执行完成",
          "raw": "'函数执行完成'",
          "value": "函数执行完成"
        }
      }
    }
  ]
}

```

- 基础方式——使用 [@babel/types](#) 来构造语句

```

1 const t = require('@babel/types');
2 const generate = require('@babel/generator').default;
3
4 const CONSOLE_AST = t.expressionStatement(
5   t.callExpression(
6     t.memberExpression(
7       t.identifier('console'),
8       t.identifier('log')
9     ),
10    [t.stringLiteral('函数执行完成')],
11  )

```

```

12  });
13
14  console.log(CONSOLE_AST, '\n\n', generate(CONSOLE_AST).code);

```

- 终极简化版——模板 API，也是上面表格提前给出来的方式：

```

1  const template = require('@babel/template').default;
2  // 或 const template = require('@babel/core').template;
3
4  const CONSOLE_AST = template.ast(
5    `console.log('函数执行完成')`
6  );
7
8  console.log(CONSOLE_AST);

```

还是那个需求：

```

1  const babel = require('@babel/core');
2
3  const code = `
4    import React from 'react';
5    const add = function (a, b) {
6      function nest () {return;}
7      return a + b;
8    }
9    let str = 'hello';
10 `;
11
12  const ast = babel.parse(code, {
13    sourceType: 'module'
14  });
15
16  babel.traverse(ast, {
17    // 仅作参考，对省略 return 的箭头函数、没有显式 return 的函数没有处理，请知悉
18    ReturnStatement(path) {
19      path.insertBefore(babel.template.ast(`console.log('函数执行完成')`));
20      // path 除了拥有当前节点的信息，还挂载着操作当前节点的各种方法、上下级节点的引用
21    },
22  });
23
24  console.log(babel.transformFromAstSync(ast).code);

```


手动构造 AST 的过程

还记得【· 基础方式——使用 @babel/types 来构造语句】那部分吧？相信所有人都会有一个疑问：那个表达式怎么来的？怎么知道一个表达式使用什么方法来构造？下面就来解决这个问题！

1. 借助网站 <https://astexplorer.net/>，输入源码 `console.log('函数执行完成')`，看到生成的 AST 结构如下：

console.log('函数执行完成')	
<pre>- Program { type: "Program" - body: [- ExpressionStatement { type: "ExpressionStatement" - expression: CallExpression { type: "CallExpression" - callee: MemberExpression { type: "MemberExpression" - object: Identifier { type: "Identifier" name: "console" } - property: Identifier { type: "Identifier" name: "log" } computed: false optional: false } - arguments: [- Literal = \$node { type: "Literal" value: "函数执行完成" raw: "'函数执行完成'" }] optional: false } }] sourceType: "module" }</pre>	<pre>t.expressionStatement(t.callExpression(t.memberExpression(t.identifier('console'), t.identifier('log'),), [t.stringLiteral('函数执行完成')]))</pre>

2. 参数顺序与含义的确定，打开 <https://babeljs.io/docs/en/babel-types>，针对上图右侧的每一个方法进行查阅，来确定参数的类型、个数。例如：

确定 expressionStatement 的参数

<https://babeljs.io/docs/en/babel-types#expressionstatement>

t.expressionStatement(expression)

<https://babeljs.io/docs/en/babel-types#expression>

t.expressionStatement(expression)

确定 **expression** 的全部类型

Expression

A cover of any Expressions.

JavaScript

Copy

```
t.isExpression(node);
```

Covered nodes:

- `ArrayExpression`
- `ArrowFunctionExpression`
- `AssignmentExpression`
- `AwaitExpression`
- `BigIntLiteral`
- `BinaryExpression`
- `BindExpression`
- `BooleanLiteral`
- `CallExpression`
- `ClassExpression`
- `ConditionalExpression`
- `DecimalLiteral`
- `DoExpression`

构造函数调用表达式作为 **expression**

**t.expressionStatement(
 t.callExpression()
)**

<https://babeljs.io/docs/en/babel-types#callexpression>

callExpression

JavaScript

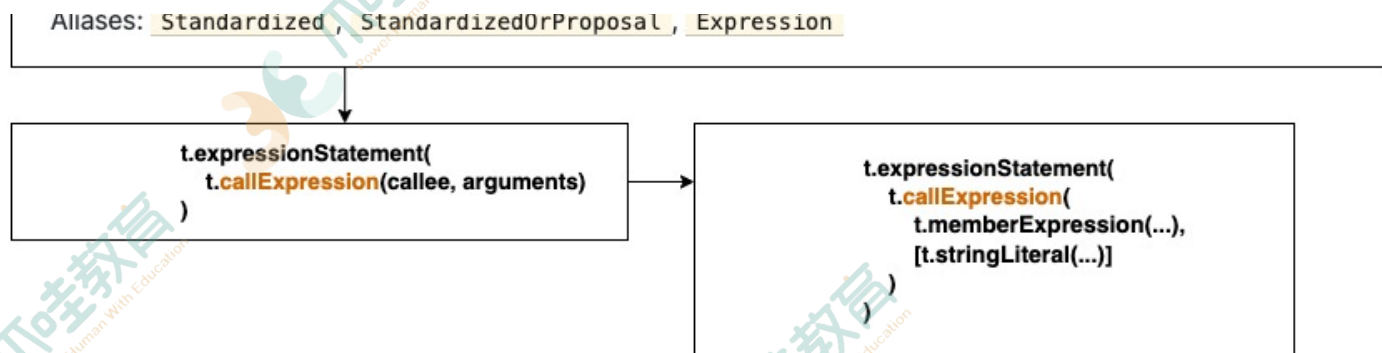
Copy

```
t.callExpression(callee, arguments);
```

See also `t.isCallExpression(node, opts)` and `t.assertCallExpression(node, opts)`.

AST Node `CallExpression` shape:

- `callee`: `Expression` | `V8IntrinsicIdentifier` (required)
- `arguments`: `Array<Expression | SpreadElement | JSXNamespacedName | ArgumentPlaceholder>` (required)
- `optional`: `true` | `false` (default: `null`, excluded from builder function)
- `typeArguments`: `TypeParameterInstantiation` (default: `null`, excluded from builder function)
- `typeParameters`: `TSTypeParameterInstantiation` (default: `null`, excluded from builder function)



手动构造 AST 的方式比较低效繁琐，但却是基于对 AST 结构的充分认识。要想深入掌握 AST、编写插件，这一过程不可忽视。建议在使用 template 之前，构造 AST 进行熟悉与实践。

AST 与 babel 插件

- 官方插件

随着 ECMAScript 的发展，不断涌出一些新的语言特性（如管道操作符、可选链操作符、控制合并操作符.....），也包括但不限于 JSX 语法等。遇到 babel 本身的解析引擎模块不能识别新特性的问题，可以由插件来处理。

```
1 ...  
2 const code = `  
3   const square = x => x ** 2;  
4   const sum = a => a + 2;  
5   const list = 5 |> square |> sum;  
6 `;  
7  
8 const ast = parser.parse(code, {  
9   sourceType: 'module'  
10 });  
11  
12 console.log(ast);
```

运行上面的代码会直接报错，源码（第 5 行）使用的管道操作符处于提案中，需要借助插件来解析：

- a. `@babel/parser` 模块 + 内联配置（记得安装 `@babel/plugin-proposal-pipeline-operator`）解析

```
1 const parser = require('@babel/core')  
2 ...  
3 const ast = parser.parse(code, {  
4   sourceType: 'module',  
5   plugins: [  
6     ['pipelineOperator', {  
7       proposal: 'hack',
```

```

8     topicToken: '^^',
9   }],
10 ]
11 });
12 ...

```

b. `@babel/core` 模块 + 文件 `babel.config.json` 解析 (babel 会自动到项目目录查找最近的 [babel 配置文件](#))

```

1 const babel = require('@babel/core')
2 ...
3 const ast = babel.parse(code, {
4   sourceType: 'module'
5 });
6 ...

```

babel.config.json:

```

1 {
2   "plugins": [
3     ["@babel/plugin-proposal-pipeline-operator", {
4       "proposal": "hack",
5       "topicToken": "^^"
6     }]
7   ]
8 }

```

同理, 其他插件通过相同的方式使用。

- 当项目需要支持的语言特性越来越多, `plugins` 需要逐一添加, 为了解决插件的管理与依赖问题, 通过[预设 \(presets\)](#) 提供常用的环境配置。因此 [babel 配置文件](#) 总能看到这样的配置 (react 项目):

```

1 {
2   "presets": ["@babel/preset-env", "@babel/preset-react"]
3   "plugin": []
4 }

```

1. 先执行完所有 `plugins`, 再执行 `presets`。
2. 多个 `plugins`, 按照声明次序顺序执行。
3. 多个 `presets`, 按照声明次序逆序执行。

- 自定义插件 ([在线指南](#): 注意链接中的 # 号, 如果被转义打不开就手动替换一下)

以下面的源码为例, 实现变量标识的重命名, 源码及转换逻辑:

```

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

```

```

2
3 const code = `
4   const square = x => x ** 2, ddd = 0;
5   const sum = a => a + 2;
6   const list = 5 |> square(^^)|> sum(^^);
7 ;
8
9 console.log(babel.transform(code).code)

```

补充内容:

```

1 // ./my-plugin.js
2 module.exports = function (babel) {
3   return {
4     visitor: {
5       VariableDeclaration(path, state) {
6         path.node.declarations.forEach(each => {
7           path.scope.rename(
8             each.id.name,
9             path.scope.generateUidIdentifier("uid").name
10          );
11        });
12      }
13    },
14  }
15 }
16
17 //babel.config.json
18 {
19   "plugins": [
20     ["@babel/plugin-proposal-pipeline-operator", {
21       "proposal": "hack",
22       "topicToken": "^^"
23     }],
24     ["./my-plugin"]
25   ]
26 }

```

输出结果:

```

1 const _uid = x => x ** 2, _uid2 = 0;

```

```
2 const _uid3 = a => a + 2;
3 const _uid4 = _uid3(_uid(5));
```

试想，如果当前作用域内，生成 uid 的方法换作最简化的不重复标识的算法，是不是就有代码压缩的效果了呢？最后，关于 path 参数上操作 ast 的一系列方法，可以[在线学习](#)（注意链接中的 # 号，如果被转义打不开就手动替换一下）。

复杂的插件可以借助一些外部工具、插件参数来实现，安装预设 `@babel/preset-env`，查看部分插件源码。如 `@babel/plugin-transform-classes`（ES6 的 class 转换）的实现（关注第二种类型）：

```
1 var _core = require("@babel/core");
2
3 // _helperPluginUtils.declare 是插件声明的工具方法
4 var _default = (0, _helperPluginUtils.declare)((api, options) => {
5   // api 是定义插件函数的第一个参数，能够访问到一些 babel 环境和方法
6   return {
7     visitor: {
8       ExportDefaultDeclaration(path) { // 默认导出的 class
9         if (!path.get("declaration").isClassDeclaration()) return;
10        (0, _helperSplitExportDeclaration.default)(path);
11      },
12      ClassDeclaration(path) { // class 关键字声明的类
13        const { node } = path;
14        const ref = node.id || path.scope.generateUidIdentifier("class");
15        path.replaceWith(_core.types.variableDeclaration("let", [
16          _core.types.variableDeclarator(ref, _core.types.toExpression(node))
17        ]));
18      },
19      ClassExpression(path, state) {} // 类表达式
20    };
21  });
22
23 exports.default = _default
```

可见，以 `ClassDeclaration` 类型声明的 class，将被替换为一条 `let` 语句，这里依赖了 `@babel/core` 模块构造 AST 节点。当然，这里的 `_core.types` 也可以从插件的首个参数解构出来。