

浅谈前端 AST 的原理及应用

Press Space for next page →



分享人：韦帆

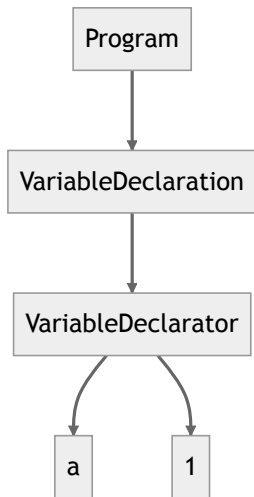
好奇心

- `IDE` 中错误提示，自动补全，代码高亮如何实现的？
- 代码压缩混淆如何实现的？
- `vue` 的 `sfc` 如何编译成浏览器认识的 `js` 的？
- `eslint`，`prettier` 如何实现的？
- `babel` 是怎么工作的？
- 各种规范之前如何互转的？`cjs-umd-amd-esm`
- `coffeescript`，`typescript`，`jsx` 等如何转换为原生 `js` 的？
- `less`，`sass`，`postcss` 等各种前置后置 `css` 处理器如何工作的？
- `webpack`，`rollup` 等各种打包器如何工作的？
- ...

什么是 AST ?

话不多说上图 [在线解析 AST](#)

抽象语法树（Abstract Syntax Tree）简称`AST`，顾名思义，它是一棵树，用分支和节点的组合来描述代码结构。他可以让计算机理解我们写的代码，从而进行一些操作。



```
AST Explorer Snippet JavaScript </> @babel/eslint-parser Transform default Parser: @babel/eslint-parser-7.13.14 4ms

1 let a = 1

Tree JSON
[Autofocus] [Hide methods] [Hide empty keys] [Hide location data] [Hide type keys]

- Program {
  start: 0
  end: 9
  + loc: {start, end, filename, identifierName}
  + range: [2 elements]
  comments: [ ]
  sourceType: "module"
  - body: [
    - VariableDeclaration {
      start: 0
      end: 9
      + loc: {start, end, filename, identifierName}
      + range: [2 elements]
      - declarations: [
        - VariableDeclarator {
          start: 4
          end: 9
          + loc: {start, end, filename, identifierName}
          + range: [2 elements]
          - id: Identifier {
            start: 4
            end: 5
            + loc: {start, end, filename}
            + range: [2 elements]
            name: "a"
          }
          - init: Literal {
            start: 8
            end: 9
            + loc: {start, end, filename, identifierName}
            + range: [2 elements]
            value: 1
            raw: "1"
          }
        }
      ]
    }
  ]
  kind: "let"
}
```

AST常见节点类型

类型名称	中文名称	描述
<code>`Program`</code>	程序主体	整段代码主体
<code>`VariableDeclaration`</code>	变量声明	声明一个变量，例如 <code>`var`</code> <code>`let`</code> <code>`const`</code>
<code>`FunctionDeclaration`</code>	函数声明	声明一个函数，例如 <code>`function`</code>
<code>`BinaryExpression`</code>	二进制表达式	通常指运算，例如 <code>`1+2`</code>
<code>`ExpressionStatement`</code>	块语句	通常是调用一个函数，例如 <code>`console.log()`</code>
<code>`BreakStatement`</code>	中断语句	通常指 <code>`break`</code>
<code>`Identifier`</code>	标识符	标识，例如声明变量时 <code>`var identi = 5`</code> 中的 <code>`identi`</code>

AST如何生成

■ 词法分析(Lexical Analysis)

词法解析是一个将字符流(char stream)转换成记号流(token stream)的过程，会将code中每个词都拆分开来，比如 ``let a = 1`` 拆分成 ``let``、``a``、``=``、``1``，并且会记录起始位置和结束位置等信息，最终生成一个 ``token`` 数组。

```
// let a = 1
[
  { type: { ... }, value: "a", start: 0, end: 1, loc: { ... } },
  { type: { ... }, value: "=", start: 2, end: 3, loc: { ... } },
  { type: { ... }, value: "1", start: 4, end: 5, loc: { ... } },
]
```

■ 语法分析(Syntactic Analysis)

通过词法分析我们拿到了 ``token`` 流，接下来我们就可以通过对 ``token`` 进行语法分析生成 ``AST``，如 ``const`` 是一个声明参数，会被标记为 ``Declaration``，数字的值会被标记为 ``NumericLiteral``，以此类推，逐步转换成一棵 ``AST`` 树。

``AST`` 总结来说就是对源码的抽象，源码中包含标识符、字面量、表达式、语句等任何东西，对应 ``AST`` 就都会有相应的映射。

为何要 `AST` 及 `AST` 编译器有哪些?

■ 为何要 `AST` ?

任何语言都是需要编译器的，也都需要将源代码解析成 `AST`，以 `JavaScript` 举例，原始的 `JavaScript` 语言计算机是无法理解的，更别说修改了，但转成 `AST` 以后，因 `AST` 是对源码的抽象，是对象结构，我们可以通过修改这个对象，进而实现修改代码

■ 常见的 `AST` 解析工具

uglify-js



Star

12.4k

esprima



Star

6.7k

acorn



Star

9.2k

@babel/parser



Star

41.9k

espre



Star

2k

typeScript



Star

88.7k

jscodeshift



Star

8.1k

gogocode



Star

4.1k

推荐使用 `@babel/parser` 以及更上层封装的 `jscodeshift`、`gogocode`。尤其 `gogocode`，类似于 `jQuery` 的链式调用，非常方便。

简单练手

代码中所有 ``let`` 改成 ``var``

- 使用 ``bable`` 进行代码转换，写法复杂
- 使用 ``jscodeshift`` 进行代码转换，写法中等
- 使用 ``gogocode`` 进行代码转换，写法最简单



▼ Sandbox Info

sweet-sanderson-yoko4o


❤ 0 👁 382 📁 3

 wChenonly
韦帆

Template Vue 3

Environment vue-cli

▼ Files

 public src

▼ App.vue



Browser

Tests

Terminal



https://yoko4o.c



```
1 <template>
2   <div>babel编译: {{ babelResult }}
3
4   <div class="a">jscodeshift编译: {{ jscodeshiftResult }}
5
6   <div class="a">gogocode编译: {{ gogocodeResult }}
7 </template>
8
9 <script>
10 import { babelResult } from './babelResult'
11 import { jscodeshiftResult } from './jscodeshiftResult'
12 import { gogocodeResult } from './gogocodeResult'
13
14 export default {
15   name: "App",
16   setup() {
17     return {
18       babelResult,
```

babel编译: var a = 1; function name(d) { var c = d; console.log(c); } name(6);

jscodeshift编译: var a = 1; function name(d) { var c = d; console.log(c); } name(6)

gogocode编译: var a = 1 function name(d) { var c = d console.log(c); } name(6)

Console

1

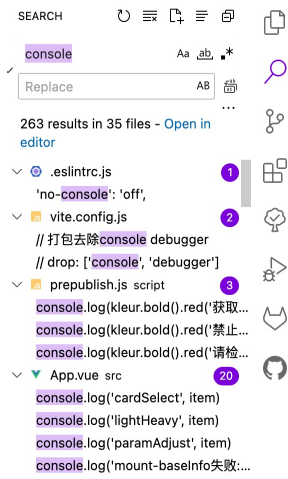
Problems

0



实际应用

出现的问题



以 `wash_api` 项目为例。

`console` 263 个，仅少数有用，大量

`console` 导致排查问题极难很快找到有效信息。

因此我们希望有办法能留下有用 `console`，而去除无用 `console`。

解决办法

- 加强代码 `review` 流程，但是有时候会疏忽
- 代码规范培训，提代码务必删除无用 `console`，但是有时候还是会疏忽
- 加强代码 `review` 流程，但是有时候会疏忽

很明显上述办法，并不能真正保证留下的都是有用 `console`。

最好的办法有工具可以在打包的时候，自动分析出哪些是有用 `console`，哪些是无用 `console`，然后删除无用 `console`

实现我们要的工具

需求:

- 可以指定删除`console`的类型, 如`log`、`warn`、`error`等
- 可以指定某个文件的`console`都不删除
- 可以根据传入的参数, 保留有用删除无用`console`, 如传入`washApi`, 则代表`console.log('washApi', ccc)`保留, `console.log('a', ccc)`删除

实现方案

`gogocode` 上场

- 问: 如何找到代码中`console`
- 答: 使用`gogocode`的`find`方法

```
import $T from "gogocode"
// 定义 `includes` 来传入需要删除的 `console` 类型, 不传默认是 `l
if (includes) {
  includes.map(type => {
    consoles.push(`console.${type}()`)
  })
} else {
  consoles = [`console.log()`]
}
const findSource = $T(source).find(consoles)
```

实现我们要的工具

- 问：找到了如何删除？
- 答：使用 ``gogocode`` 的 ``remove`` 方法

```
import $T from "gogocode"
// 定义 `includes` 来传入需要删除的 `console` 类型，不传默认是 `log`
if (includes) {
  includes.map(type => {
    consoles.push(`console.${type}()`)
  })
} else {
  consoles = [`console.log()`]
}

const findSource = $T(source).find(consoles)
return findSource.remove().generate();
```

实现我们要的工具

- 问：如何根据传入参数，删除无用，保留有用的 ``console``？
- 答：使用 ``externalValue`` 代表传入的参数，如 ``washApi``，则代表 ``console.log('washApi',ccc,ddd)`` 保留，``console.log('a', ccc)`` 删除，不传则全部删除。
难点在于我们要从 ``AST`` 读 ``console`` 的 ``value`` 的值，读到以后，用传入的参数进行正则，如果未匹配到，则删除当前 ``console``。

```
import $T from "gogocode"
const findSource = $T(source).find(conssoles);
return findSource
  .each((r: any) => {
    let eValueString = r.value.arguments
      .map((e: { value: string }) => e.value).join();
    // `eValueString` 是 `console` 中的 `value` 值，因可能是多个参数，所以用 `join` 拼接
    const pattern = new RegExp(`(${externalValue.join("|")})`), "g");
    // 如果未匹配到，则删除当前这个 `console`
    if (!pattern.test(eValueString)) return r.remove();
  })
  .root().generate();
```

实现我们要的工具

- 问：如何指定某个文件的 ``console`` 都不删除？
- 答：使用 ``external`` 代表传入的文件路径，如 ``src/ceshi.js``，则代表这个文件的 ``console`` 都不删除，难点在于我们要根据传入的参数，获取文件的绝对路径，并且要转换 ``window``，``mac``，``linux`` 对于路径格式的问题。

```
if(external?.length > 0 && 获取到的绝对路径存在){  
  // 直接返回源码，不处理  
  return source  
} else {  
  // 使用前面说的 `gogocode` 进行处理  
}
```

在打包工具中使用

上面的实现逻辑，我们还需要进一步封装成对应打包工具的插件，以便在打包工具中使用，如 `vite`，`webpack` 等。

本来这个所有的我准备写成一个 `vite` 插件用，但是发现 `github` 有轮子，但是不支持保留指定的 `console`，所以就给提了 `PR`，增加了这个功能。

下面是地址，感兴趣的可以去看看

[vite-plugin-remove-console](#)