Ast反混淆

学习AST的目的：混淆代码 -> 通过AST还原 -> 牛马能看懂的代码

什么是Ast

Ast，全称为抽象语法树，解析后的代码可以看过为一个树结构，Ast是编程语言处理中的一个重要概念。它是源代码语法结构的抽象表示，以树状结构 (理解为jSON也行) 表示源代码的语法结构。ast的使用超级广泛，但是**搞逆向的就了解逆向用法就行了**，概念就不理解太多了，不然会乱

- 官方文档：https://www.babeljs.cn/docs
- 在线转换：https://astexplorer.net

1. 常用节点汇总

序号	类型原名称	中文名称	描述
1	Program	程序主体	表示整个程序或者模块
2	VariableDeclaration	变量声明	声明一个变量，例如：var let const
3	FunctionDeclaration	函数声明	声明一个函数，例如：function
4	ExpressionStatement	表达式语句	通常是一个函数，例如：console.log()
5	BlockStatement	块语句	包括在 {} 块中的代码，例如：if(true) {var a = 1}
6	BreakStatement	中断语句	通常指break
7	ContinueStatement	持续语句	通常指 continue
8	ReturnStatement	返回语句	通常指 return
9	SwitchStatement	Switch语句	通常指Switch Case语句中的Switch
10	IfStatement	if 控制流语句	控制流语句，通常指 if(true)else{}
11	Identifier	标识符	标识符：变量名，函数名，例如变量声明时 var name = "小红" 中的 name
12	CallExpression	函数调用表达式	通常指调用一个函数，例如 console.log()
13	BinaryExpression	二元表达式	加法、乘法等，例如 1 + 2
14	MemberExpression	成员表达式	通常指对象属性访问，例如 console 对象的 log成员
15	ArrayExpression	数组表达式	通常指一个数组，例如[1,2,3,4]
16	NewExpression	New表达式	通常指使用New关键字
17	AssignmentExpression	赋值表达式	通常指函数的返回值复制给变量
18	UpdateExpression	更新表达式	通常指更新成员值，例如 i++
19	BooleanLiteral	布尔值字面量	布尔值，例如 true， false
20	Literal	字面量	字面量，例如数字，字符串，布尔值等
21	NumericLiteral	数字型字面量	数字，例如，100
22	WhileStatement	While循环语句	例如，while(true){}
23	DoWhileStatement	do While循环语句	do{}while(true)
24	ForStatement	for循环语句	for(){}
25	ForInStatement	for-in循环语句	
26	ThrowStatement	throw语句	throw("错误了")
27	TryStatement	try-catch语句	try{}catch{}

序号	类型原名称	中文名称	描述
28	CatchClause	try-catch语句中的catch	
29	DebuggerStatement	debugger语句	debugger
30	EmptyStatement	空语句	
31	UnaryExpression	一元表达式	取反、递增等
32	LogicalExpression	逻辑表达式，如逻辑与、逻辑或	
33	ObjectExpression	对象表达式	let a = {} 中的 {}
34	TemplateLiteral	模板字符串	
35	SequenceExpression	序列表达式	例如 a = 2,2

2. babel 库学习

babel 是一个JavaScript编译器，用于将ECMAscript2015+版本的代码转换为向后兼容的JavaScript语法，方便能够运行在当前和旧版本的浏览器或其他环境中，在反混淆时候我们主要使用babel提供的api

1. @babel/core: babel编译器本身，提供了babel的编译api
2. @babel/parser: 将JavaScript代码解析成AST语法树
3. @babel/traverse: 遍历，修改AST语法树的各个节点
4. @babel/generator: 将AST还原成JavaScript代码
5. @babel/types: 判断，验证节点的类型，构建新AST节点

安装命令：

- npm i @babel/core @babel/parser @babel/traverse @babel/generator @babel/types

3. babel反混淆模板

```
const fs = require("node:fs");
const process = require("node:process");
const parser = require("@babel/parser");
```

```
const traverse = require("@babel/traverse").default;
const types = require("@babel/types");

const generator = require("@babel/generator").default;

/*
* encodeFile：需要解混淆的js文件代码
* decodeFile：解混淆后的代码
* */
const encodeFile = "./encode.js";
const decodeFile = "./decode.js";
```

4. babel库path常用方法与属性

> 在ast中，path对象代表着抽象语法树中的一个节点，提供了很多的方法和属性，用于帮助我们操作和转换代码

path的主要作用：

- 节点信息：每个path对象都包含一个node属性，只想当前路径所代表的ast节点
- 遍历控制：path对象提供了各种方法来控制遍历过程，例如，跳过节点，重新访问节点等
- 节点操作：path对象允许插入，替换，删除节点
- 作用域管理：path对象拥有与作用域相关的方法scope，帮助我们处理变量和函数的作用域信息
- 上下文信息：path对象包含了关于父节点，兄弟节点，子节点的上下文信息

path.toString()

- path.toString()方法用于将当前AST路径转换为对应的JavaScript源代码

```
const fs = require("node:fs");
const process = require("node:process");
const { parse } = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const types = require("@babel/types");
const generator = require("@babel/generator").default;

const jsCode = `
    let vwo5Okfc = 50;
`;
const ast = parse(jsCode);

// 编写ast插件
```

path.replaceWith()

- path.replaceWith方法单节点替换，用于将当前节点替换为新的节点。这个方法对修改代码非常有用，因为它允许直接替换ast中的某个节点，从而改变代码的结构或内容

```
const fs = require("node:fs");
const process = require("node:process");
const { parse } = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const types = require("@babel/types");
const generator = require("@babel/generator").default;

const jsCode = `
    const a = 5;
`;
const ast = parse(jsCode);

// 编写ast插件
const visitor = {
```

- 以上的代码中，将变量名为a替换为变量名b

path.replaceWithMultiple()

- path.replaceWithMultiple方法多节点替换，用于将当前节点替换为新的节点，可以接收多个参数，用于替换多个节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = 'function add(a, b) { return a + b; }';
const ast = parser.parse(code);

traverse(ast, {
  BinaryExpression(path) {
    if (path.node.operator === '+') {
      const newNode1 = t.binaryExpression('*', path.node.left,
t.numericLiteral(2));
      const newNode2 = t.binaryExpression('*', path.node.right
```

path.remove()

- path.remove用于删除节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
  var unusedVar = 42;
  var usedVar = 24;
  console.log(usedVar);
`
const ast = parser.parse(code);

traverse(ast, {
  VariableDeclarator(path) {
```

path.insertBefore()

- path.insertBefore() 方法用于在当前节点前插入新的节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
  var x = 10;
  console.log(x);
`
const ast = parser.parse(code);

traverse(ast, {
  VariableDeclaration(path){
    const newNode = t.expressionStatement(t.stringLiteral('This is before
```

path.insertAfter()

- path.insertAfter() 方法用于在当前节点后插入新的节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
```

```
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
  var x = 10;
  console.log(x);
`
const ast = parser.parse(code);

traverse(ast, {
  VariableDeclaration(path){
```

path.traverse()

- path.traverse()方法用于在当前节点下遍历其他节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
    function add(a, b) {
      const sum = a + b;
      return sum;
    }
`
const ast = parser.parse(code);

traverse(ast, {
```

path.get()

- path.get() 获取当前路径下的子孙节点，返回一个

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
    let vw5Okfc = 5O;
`
const ast = parser.parse(code);

traverse(ast, {
    VariableDeclarator(path){
        // console.log(path.get("id"))
```

```
        // console.log(path.get("id").toString())
```

path.evaluate()

- path.evaluate() 用于对抽象语法树中的节点进行求值操作，通过path.evaluate()方法可以获取ast节点表达式的值
- evaluate()中的属性：
  - confident：布尔值属性，表示是否可以确信表达式的值。
    - true：confident为true，表示能够确定(计算)表达式的值
    - false：confident为false，表示无法法确定(无法计算)表达式的值
  - value：value属性包含evaluate() 方法会表达式的评估效果
    - confident为true时，value将包含表达式的计算结果
    - confident为false时，value值可能是undefined，或其他

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
   const vw50kfc = 10 + 10 + 10 + 10 + 10
`
const ast = parser.parse(code);

traverse(ast, {
    BinaryExpression(path){

        const evaluation = path.evaluate();
```

path.parent 属性

- path.parent: 获取当前节点的父节点路径，可以直接修改或替换整个父节点
- 可以通过path.parent.type属性，获取父节点的类型

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
   function add(){
     const vw50kfc = 10 + 10 + 10 + 10 + 10
   }
`
const ast = parser.parse(code);
```

```
traverse(ast, {
    VariableDeclarator(path){
```

path.skip()

- path.skip(): 跳过当前节点的子节点遍历

path.stop()

- path.stop(): 停止整个遍历过程

path.key

- path.key属性，获取当前节点的key值，key值通常用于配合path.get方法使用

path.node 属性

- node是path对象上的一个属性，表示当前节点。
- node对象包含了节点的所有属性和方法，可以在节点处理函数中使用 path.node 访问和修改当前节点处理函数 节点的属性和方法

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function add(aaaa){
 let vwo5Okfc = 5O;
}
`
const ast = parser.parse(code);

traverse(ast, {
```

```
traverse(ast, {
    // FunctionDeclaration节点处理函数
```

path.scope

- scope是path对象上的一个属性，表示当前节点的作用域。作用域对象包含了当前节点的变量声明和函数声明等信息
- 可以使用 path.scope 访问和修改当前节点的作用域

path.parentPath

- path.parentPath属性返回当前路径的父路径，也就是包含当前节点的上一级节点的路径
- 它允许你从当前节点访问其父节点，从而获取更多的上下文信息

5. scope 作用域相关

- scope(作用域) 代表当前路径节点的作用域。在遍历语法树时，可以使用scope属性来判断当前节点是否处于某个特定的作用域内，以及在这个作用域内声明的变量和函数
- scope提供了一些属性了方法，可以方便的查询标识符的作用域：
  - block：当前节点所在的块级作用域
  - path: 当前节点对应的路径
  - bindings: 当前作用域所有绑定(变量或函数)的信息
    - bindings 属性是一个对象
    - bindings 对象的值也是一个对象，包含以下属性：
      - kind: 变量或函数的类型，可以是var,const,let,function等
      - path: 该变量或函数的定义路径
      - constantViolations: 使用该变量或函数的非常量路径
      - referencePaths: 使用该变量或函数的所有路径
      - referenced： 是否被引用过
      - references: 引用的路径数量
- 补充知识：什么是标识符?
  - 标识符是用于，标识变量，函数，属性和参数的名称

基本使用

获取add函数中a的作用域

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;
```

```
const code = `
   const a = 1000;
   let obj = {
        name:"js逆向",
        add:function(a){
            a = 400;
        }
   }
```

scope.block

- scope.block: 获取标识符的作用域，返回一个node对象

scope.dump()

- scope.dump(): 会得到自己向上的作用域与变量信息，先输出自己当前的作用域，在输出父级作用域，在输出父级的父级的作用域，直到顶级作用域

scope.crawl()

- 重构scope，在某种情况下会报错，不过还是建议在每一个插件的最后一行加上。

scope.rename(oldName, newName, block)

- 修改当前作用域下的的指定的变量名，oldname、newname表示替换前后的变量名，为字符串。注意，oldName需要有binding，否则无法重命名。

scope.traverse(node, opts, state)

- scope.traverse()方法，用于遍历当前作用域中的节点

scope.bindings

- scope.bindings: 返回当前作用域内所有绑定的对象

在以下的代码中，在作用域中定义了a变量和foo函数，它们被存在bindings对象中，我们可以遍历bindings对象获取这些遍历和函数的信息

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
  const a = 1;
  function foo() {
    const b = 2;
    console.log(a, b);
  }
  foo();
`
const ast = parser.parse(code);
```

- 每个scope对象都有一个bingings的属性，它是一个包含当前作用域中所有绑定的对象，在JavaScript中，绑定指的是标识符和值之间的关系。
- bingings对象提供了一种获取在当前作用域中绑定的标识符和获取它们值的方法
- 以下代码中使用bindings对象获取当前作用域中所有的绑定

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
  const foo = 1;
  function bar() {}
  console.log(foo + bar());
`;

const ast = parser.parse(code);

traverse(ast, {
```

scope.hashBinding()

- scope.hashBinding: 检查标识符是否在当前作用域绑定

scope.getBinding(name)

- scope.getBinding()方法，查询特定标识符的绑定(binding)，返回一个binding对象

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
   const a = 1000;
   let obj = {
        name:"js逆向",
        add:function(a){
            a = 400;
        }
   }
`
```

输出内容：

```
Binding {
    identifier : Node {type: "Identifier", start: 83, end: 84, loc:
SourceLocation, name: "a"}, // 标识符 的 Node对象
    scope : Scope {uid: 1, path: NodePath, block: Node, labels: Map(0), inited:
true, ...},  // 对应 标识符的 scope 对象
    path : NodePath {contexts: Array(0), state: Object, opts: Object,
_traverseFlags: 0, skipKeys: null, ...}, // 对应标识符的 Path 对象
    kind : "param", // 该标识符类型 这里表示 是一个参数

    constantViolations : Array(1) [NodePath], // 存放 修改了 该标识符的节点
    constant : false, // 是否是常量
    referencePaths : Array(2) [NodePath, NodePath], // 引用了 该标识符的 NodePath
数组
    referenced : true,  // 该标识符 是否被引用
```

scope.getBinding 返回的绑定对象及其属性和方法

- scope.getBinding() 方法返回的binding对象
- 属性：
    - path：表示变量声明的 AST 节点的路径。这个节点通常是一个 `VariableDeclarator` 节点或其他声明变量的节点。
    - identifier：变量的标识符，通常是一个 `Identifier` 节点，包含了变量名等信息。
    - bindings：一个数组，包含了与该变量相关的所有绑定。在大多数情况下，它只包含一个绑定，但对于一些特殊情况（如 `eval` 和 `with` 表达式中的变量），可能包含多个绑定。
    - constant：布尔值，表示变量是否是常量（即 `const` 声明的变量）。
    - constantViolations：一个数组，包含试图更改该常量的所有位置的路径。
    - referencePaths：一个数组，包含所有引用该变量的路径。

    - references：一个数组，包含所有引用该变量的引用对象。引用对象包含了引用的路径以及其他相关信息。
- 方法：
    - isVar() / isLet() / isConst()：这些方法用于检查变量的声明类型。例如，`binding.isVar()` 返回布尔值，指示变量是否是使用 `var` 关键字声明的。
    - referencePaths：获取所有引用该绑定的路径的数组。
    - referencePaths.length：获取引用该绑定的路径数量。
    - dereference()：从当前作用域中解除该绑定。

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function exampleFunction() {
    var x = 10;
    if (true) {
        var y = 20;
        console.log(x + y);
    }
    console.log(x);
    console.log(y);
```

6. binding (绑定)

- 在ast中，只有变量定义和函数定义拥有binding(绑定)，其他获取的binding都是undefined

获取binding属性

```
let scope = path.scope;
let binding = scope.getBinding(name);
```

binding.path

- binding.path: 用于定位初始拥有binding的path;

binding.constant

- binding.constant 用于**判断当前变量是否被更改**

- true表示未改变
- false 表示有更改变量值

示例代码：

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function example() {
    const a = 10;
    let b = 20;
    var c = 30;

    a = 15;
    b = 25;
    c = 35;
```

binding.referenced

- binding.referenced 用于**判断当前变量是否被引用**
  - true 表示当前作用域中有引用该变量的地方，
  - false 表示当前作用域中没有引用该变量。
  - 注意，引用和改变是分开的。

示例代码:

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function example() {
    const a = 10;
    let b = 20;
    var c = 30;

    console.log(b); // b被引用
}
`
```

binding.referencePaths

- binding.referencePaths 用于存储，获取与特定变量绑定相关联的所有引用路径的数组

binding.constantViolations

- binding.constantViolations 是binding对象的一个属性
- 作用：
  - 用于追踪代码中对常量绑定的任何修改尝试
- 示例：
  1. 例如以下的代码中，在example函数中定义了一个x的常量，这个常量绑定不应该被重新赋值。
  2. 但是在实际的代码中，可能会有意或无意的试图重新赋值这些常量，这种行为被称为 "违规"

```
function example(){
    const x = 10;
    x = 20;
}
```

  3. 在以上的例子中，x = 20 是对常量 x 的违规操作，因为在JavaScript中，常量不能被重新赋值。使用 binding.constantViolations 属性可以捕获这些违规行为
  4. **总结：**
     - constantViolations 可以理解为存储着某个标识符的违规操作数组

示例代码：

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
    function example() {
        const x = 10;
        x = 20; // 这是一个违规的操作,因为在JavaScript中常量不能被重新赋值
    }
`
const ast = parser.parse(code);

traverse(ast, {
```

7. 获取path对象或node节点的源代码

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
   const vw50kfc = 10 + 10 + 10 + 10 + 10
`
const ast = parser.parse(code);

traverse(ast, {
    VariableDeclaration(path){
        // 输出path对象的js代码
        console.log(path.toString())
```

8. enter和exit

- enter：当遍历器进入某个节点时调用
- exit：当遍历器离开某个节点时调用

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function exampleFunction() {
    var x = 10;
    if (true) {
        var y = 20;
        console.log(x + y);
    }
    console.log(x);
    console.log(y);
```

- 在以上的代码中：
  - enter钩子在遍历到每个节点时都会被调用，并打印出节点的类型，如果该节点的名称为x或y的标识符，则将其名称改为大写
  - exit狗子在离开每个节点时也会被调用，并同样打印出节点的类型

9. @babel/types

- @babel/types库，用于判断，验证节点的类型，构建新AST节点

1. 构建节点

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
   const vw50kfc = 50;
`
const ast = parser.parse(code);

traverse(ast, {
    VariableDeclaration(path){
        const {init} = path.node;
```

- 根据BinaryExpression节点的结构，自定义一个BinaryExpression结构，通过path.set将init节点修改为自定义的BinaryExpression结构

2. 节点判断

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;

const t = require('@babel/types');
const generator = require('@babel/generator').default;

const code = `
function exampleFunction() {
    var x = 10;
    if (true) {
        var y = 20;
        console.log(x + y);
    }
    console.log(x);
```

types.valueToNode

- types.valueToNode() 方法，将给定的JavaScript值value转换为对应的AST节点，会根据value的类型和内容生成一个符合ast结构的节点