ECMAScript Dynamic Analysis

1. 概述

1.1 背景

现代 Web 开发中, 前端具有举足轻重的的地位. 最近几年中, 前端开发所使用的 JavaScript 语言发展突飞猛进, 涌现了大批的框架, 并且发展出了服务器端的 Node.js. 其语言标准也飞速演进了 ECMAScript6, ECMAScript7 等多个版本.

由于 JavaScript 语言的历史包袱和之前缺乏标准的原因, JavaScript 语法较为混乱, 具有许多不易察觉的易错点. 同时 JavaScript 特有的函数作用域规则以及原型继承对于有 Java 这样的 Class 继承语言经验的学习者来说理解有一定的难点.

本项目基于 JavaScript 的语法树, 实现了一个基本的 JavaScript 动态分析工具. 在对 JavaScript 语法节点进行模拟运行的基础上, 分析程序的各个作用域的层级关系. 每个作用域包含的变量和变量在该作用域中的值.

1.2 社区其他相关工作

AST 在前端有着非常重要的作用. 当前前端主流生产环境中, 利用 AST 主要有以下应用:

1. 代码整合: 社区中的 Webpack 等工具利用语法分析整合不同代码模块到一个 bundle.js 文件.
2. 风格规范: ESlint等工具通过语法分析, 禁止了一些写法(稀疏数组等)来保证代码质量.
3. 编译优化: Babel等利用语法分析技术将浏览器不支持的最新语法特性用低级特性模拟
4. 代码提示: 常见 IDE 利用静态分析和动态分析来完成精准的程序间跳跃, API 入口查询等功能.

2. 项目介绍

本项目利用开源的 ESPrima [1] 工具, 解析得到 JavaScript 的 AST, 基于 AST 解释运行其中的 Expressions, 进行动态分析.

2.1 项目特点

本工具分为两个部分 nameProcess 和 valueProcess:

1. nameProcess进行静态分析, 遍历 AST, 处理ESPrima生成的语法节点结构, 生成变量作用域结构.
2. valueProcess 动态解释 AST 中的各个 Expressions 语句, 得到各个变量的值.

本项目支持最新的 ES6/7 语法, 其中:

1. nameProcess: 支持let, const, var声明的变量, object, array, function 以及ES6的 class 和 箭头函数.
2. valueProcess: 支持绝大对数的 Expressions 语句, 具体如下
    - 基本操作符: 所有一元, 二元操作符
    - 对象属性: a.b
    - 函数: 函数调用和返回, 包括闭包
    - this: this 表达式
    - arguments: 函数内部的arguments 变量

2.2 项目用途

对 JavaScript 的动态分析在前端的许多领域具有重要的应用. 除了之前提到的内容外, 本项目可以运用在:

1. 学习 JavaScript: 初学者可以通过该工具输出作用域, 变量名, 变量值等运行内容学习 JavaScript.
2. 检测程序正确性: 利用本程序提供的接口编写脚本检验程序的正确性.
3. 开发插件: 用以开发在 Sublime, VS Code 等工具下的插件, 实现点击函数名准确跳跃到其定义处, 调试程序等辅助功能.

3. 实现过程

3.1 语法解析

JavaScript 作为一种功能强大的现代语言, 其语法树非常复杂, 如果自己来搭建词法和语法解析需要几周的时间. 这里我们基于社区最广泛使用的开源 Parser, ESPrima 得到语法树. 并且对该语法树进行修饰得到我们使用的语法节点结构.

本节简要介绍 ECMAScript 语法标准和 ESPrima 的解析结构.

3.1.1 Syntax Tree Specification

ESPrima 建立了一个节点标准, 目前被广为采用, 称为 ESTree Specification [2]. 该规范基于最初的 Mozilla Parser API [3].

最基础的结构为 Node 节点, 所有节点都继承自该结构, 其定义如下:

```
{
    type: string,
    range: [number, number],
    location: {
        start: {
            line: number,
            col: numer
        },
        end: {
            line: number,
            col: number
        }
    }
}
```

其中 range 和 location 都是可选的, 本项目中并不需要

其中种类分为三大类:

1. Expressions and Patterns: 包括了数组, 函数(特殊的对象), 对象等各个可以求值的表达式

2. Statements and Declarations: 主要是结构性语法, 包括 block, 循环, 判断等
3. Modules: 在 ES6 引入的模块化功能, 包括 import 和 export 语句.

3.1.2 Node Types

必须了解每个节点的种类, 才能根据 AST 来解释执行 JavaScript 代码.

本节分析了 ESTree 节点的各个种类. 本工具重点侧重于 Declaration 和 Expression

1. Programs: 代表了根节点, 具有 body 属性代表了一个完整的语法树
2. Identifier: 变量名, 属性名, 函数名都是标识符, 它可能是表达式, 或 ES6 中的解构(一种语法模式)
3. Literal: JavaScript 字面量可能为字符串, 数字, 布尔值, null 和 正则表达式. 对于正则表达式, 多出一个字段来保存正则的 pattern 和 flag.
4. Function: 函数节点, 实际上函数总是以FunctionDeclaration和FunctionExpression的形式存在.
5. Statement: Statement 是一个大的抽象分类, 具有许多具体的子类:
    - Expression: 表达式语句, 下面会详细介绍
    - Block: 语句块, 包含一个 body 数组属性
    - Empty: 空节点,比如分号
    - 控制流: 包括Return, Continue, Break, label等
    - 条件: 主要是 if 和 else 以及 switch * 循环: 包括了for, for in 和 for of 以及 while
    - 声明: 具有 kind 属性, 因为 ES6 引入了块级作用域 let 和 const
    - 其他: 比如被废弃的 with 语句
6. Expressions:
    - 常见的一元操作符, 二元操作符, 三元操作符
    - thisExpression: 表示 JavaScript 的 this

- ArrayExpression:
- ObjectExpression: 对象节点, properties 属性包含了对象的属性键值对
- FunctionExpression: 函数表达式节点

3.2 NameProcessor

通过 ESPrima 得到语法树后, 可以进行 Name Process 来得到作用域之间的层级关系和变量所处信息. 使用开源工具 ESTraverse [4] 辅助遍历语法树.

本节简单介绍本项目所采用的分析方法.

3.2.1 Scope Structure

本工具建立了一种 Scope 数据结构. 之后会对每个语法节点加入该结构. 其基本结构如下:

```
{
    this.name = name || null;
    this.scopes = [];
    this.references = {};
    this.thisRef = null;
    this.parent = null;
}
```

1. scopes 包含了内层子作用域
2. references 包含了当前作用域下的所有变量(包括函数, 数组, 对象)
3. thisRef 保存了当前的this作用域, 只在对象中指向调用者

在该结构上实现了一些基础的实例方法, 包括:

- 查询当前或者子作用域内变量: findReference, hasReference, hasOwnReference
- 增加/删除/修改当前或子作用域的变量: setReference, setOwnReference, getOwnReference等

- 增加/删除/查询子作用域: addScope, removeScope, hasScope

具体可以查看源文件 scope.js

3.2.2 Process Procedure

处理函数接受 AST 和一个初始的 Scope 作为参数. 首先设置根作用域为传入的 Scope. 之后利用ESTraverse遍历传入的 Node.

对于每一个 Node 首先确定其是否为 Block 类(包括Swith, Block, For等), 如果是则以当前作用域为父作用域新建一个 Scope. 否则给该 Node 添加 Scope 字段, 并初始化为其 Parent Node 的 Scope.(Root Node 的 Scope 指向传入的初始 Scope 参数).

之后判断节点类型, 因为 JavaScript 的历史原因, 有两种不同的作用域:

1. 对于变量声明中的 var 和函数声明, 采用函数作用域, 即沿着 Scope 链找到最近的 Function. 给该层 Scope 加入一个变量引用.
2. 对于 const 和 let 以及最新的 class(本质还是函数) 声明, 采用块级作用域, 直接加入一个新的变量引用.

运行完该过程后, 标准的语法树节点被改造为我们需求的结构, 可以进行进一步的解释执行和分析:

1. 每一个 Node 具有 Scope 属性指向它所在的作用域. 相同作用域里的节点指向相同的 Scope 引用
2. 每一个节点在遍历过程中, 加入了 parent 属性, 便于快速向上查询.
3. 得到了一个根 Scope 对象. 可以调用该对象获取作用域的层级结构.

3.3 ValueProcessor

valueProcessor 可以根据语法树, 解释执行 JavaScript 代码, 得到各个变量在作用域中的值. 运行 valueProcessor 前必须首先运行 nameProcessor.

该部分是一个小型的, 仅支持表达式的 JavaScript 解释器, 基本支持 JavaScript 中的常见 Expression s. 通过解释执行 AST 节点, 获取各个变量的值, 存储在 Scope 结构内.

对于 Statement 语句, 比如控制流 if, for, while 以及其他上文语法分析时提到的语句, 本项目暂时不支持.

这里简要介绍解释执行的实现

因为只处理 Expression, 遍历语法树节点, 筛选节点类型, 对于 Expressions 节点,判断类型:

1. **一元操作符**: 包括 +, -, !, ~, typeof 和 delete, 正常解释, 递归运行其后的 Expression 即可.
2. **Update操作符**: 语法树将 -- 和 ++ 单独作为一类节点, 并且在 token 中用 prefix 属性表明其和被操作的 Expression 的前后关系. 因此 Update 时要判断该 token 的 prefix 来完成解释.
3. **二元操作符**: 正常运算即可, 特殊的有 in 操作符, 需要遍历右值(Object)的属性来返回结果.
4. **Array & Object**: 数组字面量和对象字面量在 JavaScript 中属于 Expression, 因为在 ES6 中允许用变量来初始化对象属性. 对于对象/数组, 遍历所有成员, 递归解析 Expression, 将其成员和值记录在 token.obj 域中.
5. **左值是对象属性**: 如a.b = ..., 首先从 Scope 里取得 b 的引用再调用赋值.
6. **Function & ArrowFunction**: 递归遍历函数的节点即可, 对于函数内部, 依然只能够处理Expressions 节点. 特别的加入了对 Return Statement 的识别, 可以将最终结果表达式返回出来.

具体的实现可以参看lib/valueResolver.js源文件

4. 运行示例

4.1 运行方法

运行该工具需要 Node.js 环境.

1. 安装依赖包 ESPrima 和 ESTraverse.

```
    npm install
```

2. 在文件中导入依赖包
3. 创建初始作用域 Scope
4. 语法分析, 得到 AST
5. 进行 NameProcess 和 ValueProcess

4.2 基本示例

如下所示是给出的一个示例程序, 程序的输出标出在注释中

```
let estel = require('./src/index');
let parser = require('esprima');

let scope = estel.createScope();
let ast = parser.parse(`
    let obj = {
        a: false
    };
    let fn = function () {
        return 1;
    };
    let number = fn();
    let number2 = obj.a;
`);


estel.processNames(ast, scope);
estel.processValues(ast);

let numberRef = scope.getReference('number');
let number2Ref = scope.getReference('number2');
let objRef = scope.getReference('obj');

console.log(scope.getReferenceNames()); // [ 'obj',
'fn', 'number', 'number2' ]
console.log(numberRef.value);       // 1
console.log(number2Ref.value);      // false
```

```
console.log(objRef.value);          // { a: false }
```

4.3 高级语法示例

下面的示例给出了对 ES6 特性 Arrow Function 以及 this 和 arguments 表达式的支持.

```
let estel = require('./src/index');
let parser = require('esprima');

let scope = estel.createScope();
let ast = parser.parse(`
    function fn(a, b) {
        return () => this.someProp + arguments[0] +
arguments[1];
    };
     let obj = { someProp: 10, fn };
     let closure = obj.fn(1, 2);
     let result = closure();
`);


estel.processNames(ast, scope);
estel.processValues(ast);

let resultRef = scope.getReference('result');

console.log(scope.getReferenceNames()); // [ 'fn',
'obj', 'closure', 'result' ]
console.log(resultRef.value); // 13
```

5. 参考资料

1. *ESPrima* http://esprima.org/
2. *ESTree Specification* https://github.com/estree/estree

3. *Mozilla Parser API* https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
4. *ESTraverse* https://github.com/estools/estraverse