前端里说的ast是什么？

关注问题 写回答

前端开发

关注者 55 被浏览 22,496

前端里说的ast是什么？

关注问题 写回答 邀请回答 好问题 添加评论 分享 ···

查看全部 6 个回答

京东云 已认证账号 + 关注

3 人赞同了该回答

1 介绍 AST

打开前端项目中的 package.json，会发现众多工具已经占据了我们开发日常的各个角落，例如 JavaScript 转译、CSS 预处理、代码压缩、ESLint、Prettier 等。这些工具模块大都不会交付到生产环境中，但它们的存在于我们的开发而言是不可或缺的。

有没有想过这些工具的功能是如何实现的呢？没错，抽象语法树 (Abstract Syntax Tree) 就是上述工具的基石。

Babel，Webpack，Vue-cli 和 EsLint 等很多的工具和库的核心都是通过 Abstract Syntax Tree 抽象语法树这个概念来实现对代码的检查、分析等操作的。在前端当中 AST 的使用场景非常广，比如在 Vue.js 当中，我们在代码中编写的 template 转化成 render function 的过程当中第一步就是解析模版字符串生成 AST。

AST 的官方定义：

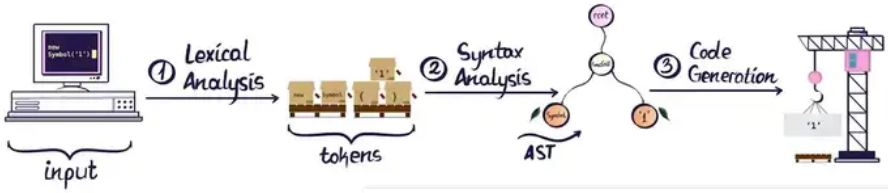
抽象语法树 (Abstract Syntax Tree，AST)，是源代码语法结构的一种抽象表示。以树状的形式表现编程语言的语法结构，每个节点都表示源代码中的一种结构。

JS 的许多语法为了给开发者更好的编程体验，并不适合程序的理解。所以需要把源码转化为 AST 来更适合程序分析，浏览器的编译器一般会把源码转化为 AST 来进行进一步的分析来进行其他操作。通过了解 AST 这个概念，对深入了解前端的一些框架和工具是很有帮助的。

那么 AST 是如何生成的？为什么需要 AST ？

了解过编译原理的同学知道计算机想要理解一串源代码需要经过 “漫长” 的分析过程：

1. 词法分析 (Lexical Analysis)
2. 语法分析 (Syntax Analysis)
3. ...
4. 代码生成 (Code Generation)



▲ 赞同 3 ▼ 添加评论 分享 收藏 喜欢 ··· 收起

关于作者

京东云 更懂产业的云

已认证账号

回答	文章	关注者
560	1,443	12,307

关注 发私信

被收藏 3 次

开发周边 0 人关注
trto1987 创建

收藏 0 人关注
刘德华 创建

相关问题

aso的前端指什么？ 0 个回答

本人新手，想学习ASO，该怎么入手？ 0 个回答

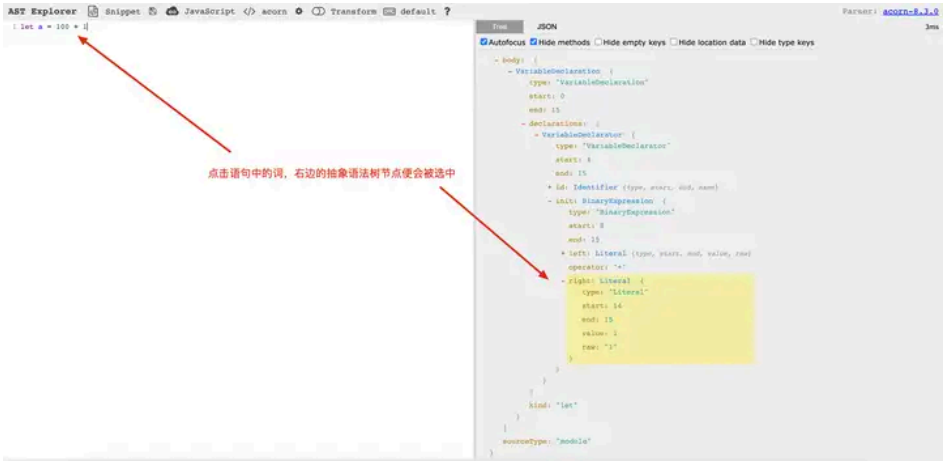
学seo了再怎么学习aso呢？ 6 个回答

ASM后台数据如何分析？ 0 个回答

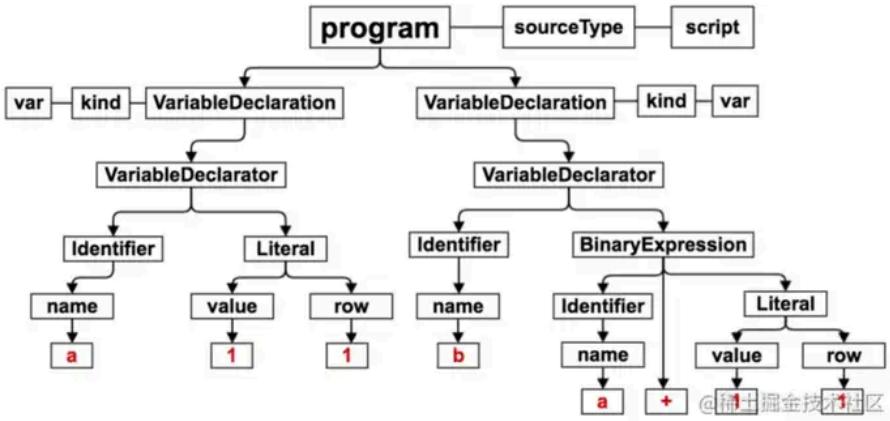
ASM的账户搭建与优化流程是哪些？ 1 个回答

前端里说的ast是什么?

边的抽象语法树节点便会被选中，如下图：



代码转化成 AST 后的格式大致如下图所示：



为了方便大家理解抽象语法树，来看看具体的例子。

```
var tree = 'this is tree'
```

js 源代码将会被转化成下面的抽象语法树：

```
{
  "type": "Program",
  "start": 0,
  "end": 25,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 25,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 25,
```

前端里说的ast是什么?


```
            "start": 4,
            "end": 8,
            "name": "tree"
          },
          "init": {
            "type": "Literal",
            "start": 11,
            "end": 25,
            "value": "this is tree",
            "raw": "'this is tree'"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "module"
}
```


可以看到一条语句由若干个词法单元组成。这个词法单元就像 26 个字母。创造出个十几万的单词，通过不同单词的组合又能写出不同内容的文章。

字符串形式的 type 字段表示节点的类型。比如”BlockStatement”，” Identifier”，” BinaryExpression” 等。 每一种类型的节点定义了一些属性来描述该节点类型，然后就可以通过这些节点来进行分析其他操作。

2 AST 如何生成

看到这里，你应该已经知道抽象语法树大致长什么样了。那么 AST 又是如何生成的呢?

以上面 var tree = ‘this is tree’ 为例:

词法分析

其中词法分析阶段扫描输入的源代码字符串，生成一系列的词法单元 (tokens)，这些词法单元包括数字，标点符号，运算符等。词法单元之间都是独立的，也即在该阶段我们并不关心每一行代码是通过什么方式组合在一起的。

大致可以看出转换之前源代码的基本构造。

语法分析阶段 —— 老师教会我们每个单词在整个句子上下文中的具体角色和含义。

- 代码生成

最后是代码生成阶段，该阶段是一个非常自由的环节，可由多个步骤共同组成。在这个阶段我们可以遍历初始的 AST，对其结构进行改造，

前端里说的ast是什么?

代码生成阶段 —— 我们已经弄清楚每一条句子的语法结构并知道如何写出语法正确的英文句子，通过这个基本结构我们可以把英文句子完美地转换成一个中文句子。

3 AST 的基本结构

抛开具体的编译器和编程语言，在 “AST 的世界” 里所有的一切都是节点 (Node)，不同类型的节点之间相互嵌套形成一颗完整的树形结构。

```
{
  "program": {
    "type": "Program",
    "sourceType": "module",
    "body": [
      {
        "type": "FunctionDeclaration",
        "id": {
          "type": "Identifier",
          "name": "foo"
        },
        "params": [
```

前端里说的ast是什么?

```
            name : 'x'
          }
        ],
        "body": {
          "type": "BlockStatement",
          "body": [
            {
              "type": "IfStatement",
              "test": {
                "type": "BinaryExpression",
                "left": {
                  "type": "Identifier",
                  "name": "x"
                },
                "operator": ">",
                "right": {
                  "type": "NumericLiteral",
                  "value": 10
                }
              }
            }
          ]
        }
        ...
      }
      ...
    ]
}
```

AST 的结构在不同的语言编译器、不同的编译工具甚至语言的不同版本下是各异的，这里简单介绍一下目前 JavaScript 编译器遵循的通用规范 —— ESTree 中对于 AST 结构的一些基本定义，不同的编译工具都是基于此结构进行了相应的拓展。

前端里说的ast是什么?

4 AST 的应用场景和用法

了解 AST 的概念和具体结构后，你可能不禁会问：AST 有哪些使用场景，怎么用?
代码语法的检查、代码风格[9]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

- 如 JSLint、JSHint 对代码错误或风格的检查，发现一些潜在的错误。
- IDE 的错误提示、格式化、高亮、自动补全等等。

代码混淆压缩。

- UglifyJS2 等。

优化变更代码，改变代码结构使达到想要的结构。

- 代码打包工具 webpack、rollup 等等。
- CommonJS、AMD、CMD、UMD 等代码规范之间的转化。
- CoffeeScript、TypeScript、JSX 等转化为原生 Javascript。

至于如何使用 AST ，归纳起来可以把它的使用操作分为以下几个步骤：

前端里说的ast是什么?

1. 解析 (Parsing)：这个过程由编译器实现，会经过词法分析过程和语法分析过程，从而生成 AST。
2. 读取 / 遍历 (Traverse)：深度优先遍历 AST ，访问树上各个节点的信息（Node）。
3. 修改 / 转换 (Transform)：在遍历的过程中可对节点信息进行修改，生成新的 AST。
4. 输出 (Printing)：对初始 AST 进行转换后，根据不同的场景，既可以直接输出新的 AST，也可以转译成新的代码块。

通常情况下使用 AST，我们重点关注步骤 2 和 3，诸如 Babel、ESLint 等工具暴露出来的通用能力都是对初始 AST 进行访问和修改。

这两步的实现基于一种名为访问者模式的设计模式，即定义一个 visitor 对象，在该对象上定义了对各种类型节点的访问方法，这样就可以针对不同的节点做出不同的处理。例如，编写 Babel 插件其实就是在构造一个 visitor 实例来处理各个节点信息，从而生成想要的结果。

```
const visitor = {
    CallExpression(path) {
        ...
    }
    FunctionDeclaration(path) {
        ...
    }
    ImportDeclaration(path) {
        ...
    }
    ...
}
traverse(AST, visitor)
```

5 AST 的转化流程

利用 babel-core (babel 核心库，实现核心的转换引擎) 和 babel-types (可以实现类型判断，生成 AST 节点等) 和 AST 来将

```
let sum = (a, b) => a + b
```

改成为：

```
let sum = function(a, b) {
  return a + b
```

前端里说的ast是什么?

实现代码如下：

```
// babel核心库，实现核心的转换引擎
let babel = require('babel-core');
// 可以实现类型判断，生成AST节点等
let types = require('babel-types');

let code = `let sum = (a, b) => a + b`;
// let sum = function(a, b) {
//   return a + b
// }

// 这个访问者可以对特定类型的节点进行处理
let visitor = {
  ArrowFunctionExpression(path) {
    console.log(path.type);
    let node = path.node;
    let expression = node.body;
    let params = node.params;
    let returnStatement = types.returnStatement(expression);
    let block = types.blockStatement([
        returnStatement
    ]);
    let func = types.functionExpression(null,params, block,false, false);
    path.replaceWith(func);
  }
}

let arrayPlugin = { visitor }
// babel内部会把代码先转成AST， 然后进行遍历
let result = babel.transform(code, {
  plugins: [
    arrayPlugin
  ]
})
console.log(result.code);
```

分词将整个代码字符串分割成最小语法单元数组，生成 AST 抽象语法树，经过转化 transformer 生成新的 AST 树，遍历生成最终想要的结果 genrator：

前端里说的ast是什么?

- 通过 esprima 生成 AST
- 通过 estraverse 遍历和更新 AST
- 通过 escodegen 将 AST 重新生成源码

我们可以来做一个简单的例子：
1. 先新建一个 test 的工程目录。
2. 在 test 工程下安装 esprima、estraverse、escodegen 的 npm 模块

```
npm i esprima estraverse escodegen --save
```

3. 在目录下面新建一个 test.js 文件，载入以下代码：

```
const esprima = require('esprima');
let code = 'const a = 1';
const ast = esprima.parseScript(code);
console.log(ast);
```

将会看到输出结果：

```
Script {
  type: 'Program',
  body:
   [ VariableDeclaration {
       type: 'VariableDeclaration',
       declarations: [Array],
       kind: 'const' } ],
  sourceType: 'script' }
```

4. 再在 test 文件中，载入以下代码：

```
const estraverse = require('estraverse');

estraverse.traverse(ast, {
    enter: function (node) {
        node.kind = "var";
    }
});

console.log(ast);
```

5. 最后在 test 文件中，加入以下代码：

```
const escodegen = require("escodegen");
const transformCode = escodegen.generate(ast)

console.log(transformCode);
```

输出的结果：

```
var a = 1;
```

前端里说的ast是什么?

6 实际应用

利用 AST 实现预计算的 Babel 插件，实现代码如下：

```
// 预计算简单表达式的插件
let code = `const result = 1000 * 60 * 60`;
let babel = require('babel-core');
let types= require('babel-types');

let visitor = {
  BinaryExpression(path) {
    let node = path.node;
    if (!isNaN(node.left.value) && ! isNaN(node.right.value)) {
      let result = eval(node.left.value + node.operator + node.right.value);
      result = types.numericLiteral(result);
      path.replaceWith(result);
      let parentPath = path.parentPath;
      // 如果此表达式的parent也是一个表达式的话，需要递归计算
      if (path.parentPath.node.type == 'BinaryExpression') {
        visitor.BinaryExpression.call(null, path.parentPath)
      }
    }
  }
}

let cal = babel.transform(code, {
  plugins: [
    {visitor}
  ]
});
```

> 作者：京东物流 李琼
> 来源：京东云开发者社区 自猿其说 Tech

发布于 2023-07-20 22:51 · IP 属地北京

更多回答

kerry

Esprima: Parser

展开阅读全文

看一些demo就有直观的感受了

▲ 赞同 19 ▼ 添加评论 分享 收藏 喜欢 ···

匿名用户

我猜是指抽象语法树（Abstract Syntax Tree）。

▲ 赞同 3 ▼ 添加评论 分享 收藏 喜欢 ···

前端里说的ast是什么?