AST 必知必会

AST 背景

在计算机科学中，**抽象语法树**（**A**bstract **S**yntax **T**ree，AST），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

AST 运用广泛，比如：

- 高级语言的编译、机器码的生成
- 一些高级编辑器的错误提示、代码高亮、代码自动补全；
- 对于前端来说很多工具，例如 `elint`、`pretiier` 对代码错误或风格的检查，babel、typescript 对代码的编译处理等等。

AST 转化流程

我们可以实现一个非常简单的词法分析工具，来感受一下词法分析的魅力，以及这中间我们需要处理的内容。

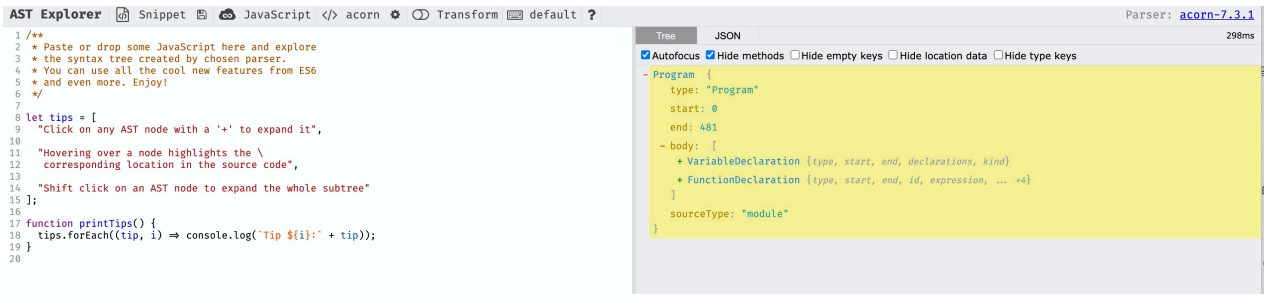
在例子中我们可以发现，我们通过读取字符串中每个元素，依次记录里面出现的内容，最终基于内容生成配置，然后再基于配置创建新的代码的结构。

整个解析过程主要分为以下两个步骤：

- 分词：将整个代码字符串分割成最小语法单元数组
- 语法分析：在分词基础上建立分析语法单元之间的关系

词法分析器里，每个关键字是一个 Token ，每个标识符是一个 Token，每个操作符是一个 Token，每个标点符号也都是一个 Token。除此之外，还会过滤掉源程序中的注释和空白字符（换行符、空格、制表符等。

我们可以通过 ast-explore 来查看代码片段转化的结果：



我们可以看到，对于左侧的代码结构，通过解析字符及对应的格式，然后序列化成为一个对象的格式，我们可以通过这个对象，来描述整体的代码的内容。

如果我们希望将 let 转化为 var，那后续我们只需要在基于配置渲染目标时，将 let 转化为 var 生成即可。

对于 AST 的类型来说，解析的过程中有这么多的类型，针对不同的语句，最终会以下面的类型进行转化：

```
ThisExpression | Identifier | Literal |
ArrayExpression | ObjectExpression | FunctionExpression |
ArrowFunctionExpression | ClassExpression |
TaggedTemplateExpression | MemberExpression | Super | MetaProperty |
NewExpression | CallExpression | UpdateExpression | AwaitExpression |
UnaryExpression |
BinaryExpression | LogicalExpression | ConditionalExpression |
YieldExpression | AssignmentExpression | SequenceExpression;
```

针对不同的工具，最终也有不同的效果：

`@babel/parser`：转化为 `AST` 抽象语法树；

`@babel/traverse` 对 `AST` 节点进行递归遍历；

`@babel/types` 对具体的 `AST` 节点进行进行修改；

`@babel/generator`： `AST` 抽象语法树生成为新的代码；

babel 插件

Babel 的处理步骤

Babel 的三个主要处理步骤分别是： **解析（parse）**，**转换（transform）**，**生成（generate）**。.

词法分析阶段把字符串形式的代码转换为 **令牌（tokens）** 流。.

你可以把令牌看作是一个扁平的语法片段数组：

```
n * n;
[
  { type: { ... }, value: "n", start: 0, end: 1, loc: { ... } },
  { type: { ... }, value: "*", start: 2, end: 3, loc: { ... } },
  { type: { ... }, value: "n", start: 4, end: 5, loc: { ... } },
  ...
]
```

每一个 `type` 有一组属性来描述该令牌：

```
{
  type: {
    label: 'name',
    keyword: undefined,
    beforeExpr: false,
    startsExpr: true,
    rightAssociative: false,
    isLoop: false,
```

```
    isAssign: false,
    prefix: false,
    postfix: false,
    binop: null,
    updateContext: null
  },
  ...
}
```

和 AST 节点一样它们也有 `start`，`end`，`loc` 属性。

对于一个 babel 插件来说，我们先从先从一个接收了当前`babel`对象作为参数的 `function` 开始。

```
export default function(babel) {
  // plugin contents
}
```

由于你将会经常这样使用，所以直接取出 `babel.types` 会更方便：（译注：这是 ES2015 语法中的对象解构，即 Destructuring）

```
export default function({ types: t }) {
  // plugin contents
}
```

接着返回一个对象，其 `visitor` 属性是这个插件的主要访问者。

```
export default function({ types: t }) {
  return {
    visitor: {
      // visitor contents
    }
  };
};
```

Visitor 中的每个函数接收2个参数：`path` 和 `state`

```
export default function({ types: t }) {
  return {
    visitor: {
      Identifier(path, state) {},
      ASTNodeTypeHere(path, state) {}
    }
  };
};
```

让我们快速编写一个可用的插件来展示一下它是如何工作的。下面是我们的源代码：

```
foo === bar;
```

其 AST 形式如下：

```
{
  type: "BinaryExpression",
  operator: "===",
  left: {
    type: "Identifier",
    name: "foo"
  },
  right: {
    type: "Identifier",
    name: "bar"
  }
}
```

我们从添加 `BinaryExpression` 访问者方法开始：

```
export default function({ types: t }) {
  return {
    visitor: {
      BinaryExpression(path) {
        // ...
      }
    }
  };
}
```

然后我们更确切一些，只关注哪些使用了 `===` 的 `BinaryExpression`。

```
visitor: {
  BinaryExpression(path) {
    if (path.node.operator !== "===") {
      return;
    }

    // ...
  }
}
```

现在我们用新的标识符来替换 `left` 属性：

```
BinaryExpression(path) {
  if (path.node.operator !== "===") {
    return;
  }

  path.node.left = t.identifier("sebmck");
  // ...
}
```

于是如果我们运行这个插件我们会得到：

```
sebmck === bar;
```

现在只需要替换 `right` 属性了。

```
BinaryExpression(path) {
  if (path.node.operator !== "===") {
    return;
  }

  path.node.left = t.identifier("sebmck");
  path.node.right = t.identifier("dork");
}
```

这就是我们的最终结果了：

```
sebmck === dork;
```

我们可以用更复杂的一些配置来完善我们的插件。。