ES6 & ESNext 常见 API 及 babel 编译

ESnext 新增内容

新增关键词

let / const / async / await / yield

新增关键特性：

- 模板字符串
- 解构赋值 / 展开操作
- promise / ESModule
- for of
- async/await
- 迭代器接口
- 可选链

新增方法、对象等内容

- Proxy / Reflect
- Set / Map

- Array.prototype.includes / Array.prototype.flat / Array.prototype.flatMap / fill、find
- Object.assign / Object.entries / Object.values

babel 处理过程

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。.

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；

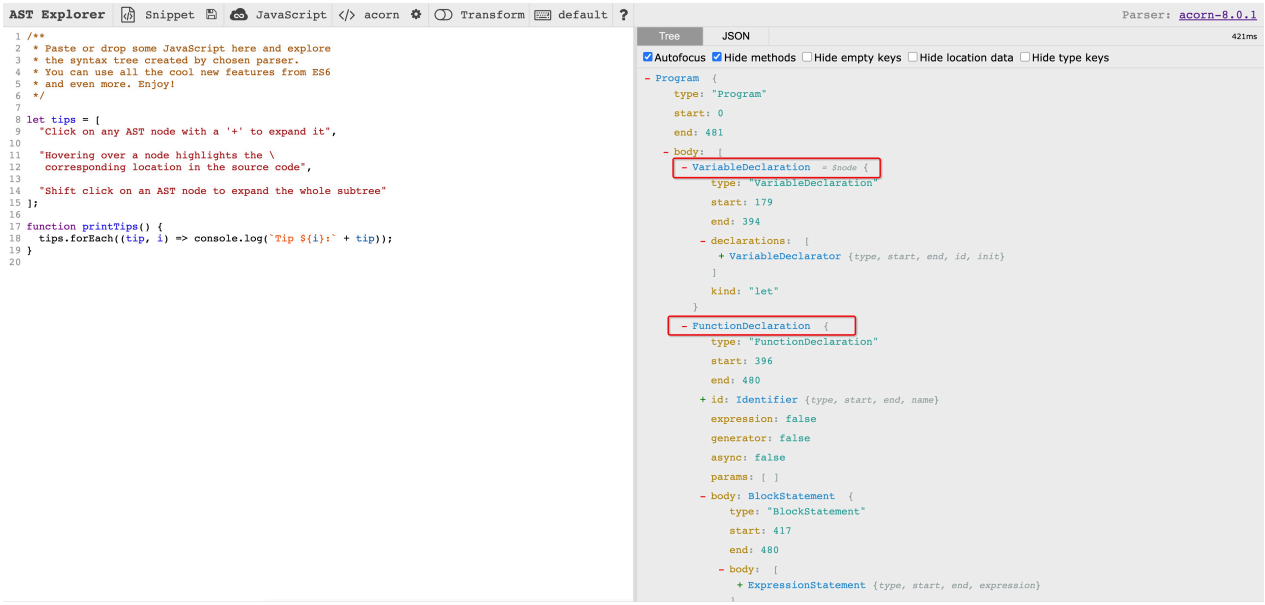
`@babel/traverse` 对 `AST` 节点进行递归遍历；

`@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；

解析为 AST

我们可以在 https://astexplorer.net/ 看到对于 JS 来说，常见的 JS 代码最终经过分词和 AST 转化之后的样子



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

转化 AST 结构

这一步也是我们插件中具体需要做的内容，在插件中，我们通过对象等方式操作 AST 中的结构，让我们的工具能够处理不同的场景。

```
export default function({ types: t }) {
  return {
    visitor: {
      BinaryExpression(path) {
        if (path.node.operator !== "===") {
          return;
```

```
        }

        path.node.left = t.identifier("a");
        path.node.right =
t.identifier("b");
      }
    }
  };
}
```

例如上面的例子中，我们通过 visitor 配置，表示当遍历 AST 结构中，遇到 BinaryExpression 这个类型时，对 AST 结构执行一定的回调函数。内部通过 path 对象，操作了当前结构下的 AST 内容。

基于 AST 结构生成代码

基于更改后的新 AST 结构，我们只需要将他们再次转化为字符串形式生成即可，这完全是第一步中解析过程的逆操作。通过字符串形式，我们就可以获得编译处理后的新代码内容了。