前端AST

一、课程目标

二、课程大纲

三、课程内容

1、AST简介

2、AST生成

如何处理AST

Literal 字面量

dentifier 标识符

Statement 语句

Declaration 声明语句

Expression 表达式

Modules ES module模块语法

Program & Directive

3、常用的AST库

4、AST的应用场景

5、Babel原理

Babel简述

工作原理

Parse

Traverse

Generate

6、Babel插件开发

Plugin的概念

Plugin原理

babel的处理步骤

parse

transform

generate

访问者模式
Plugin分类
1. syntax
2. transform
3. proposal
插件开发实战

一、课程目标

P6：

- 了解基本编译流程
- 了解AST基本结构
- 了解Babel原理

P6+ – P7

- 熟悉Babel原理和插件机制
- 能够编写Babel插件
- 理解编译原理的实际应用场景

二、课程大纲

- AST简介和使用场景
- AST应用场景
- Babel原理解析
- Babel插件开发实战

三、课程内容

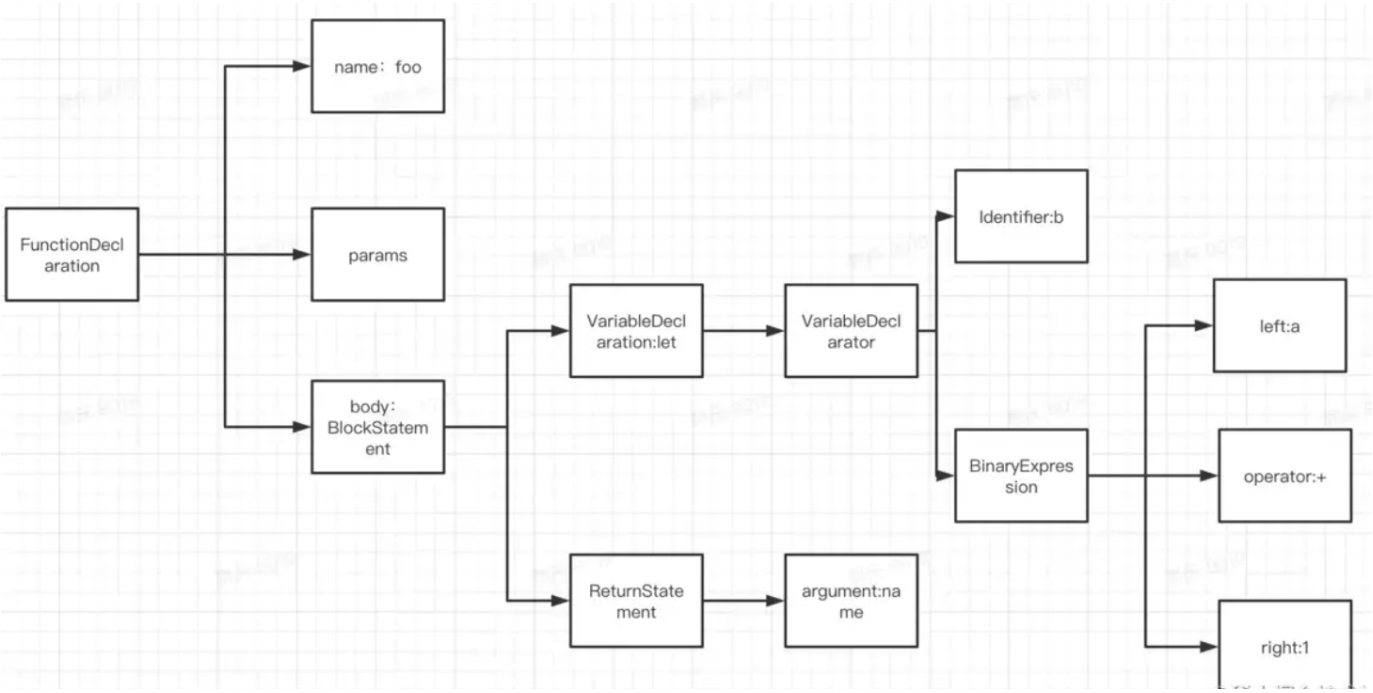
1、AST简介

抽象语法树（Abstract Syntax Tree）简称AST，顾名思义，是一棵树，用分支和节点的组合来描述代码结构。

例如如下代码：

```typescript
function foo(a) {
  const b = a + 1;
  return b;
}
```

分析，首先这是一个函数，有名字(foo)，参数(a)，函数体(body)三个基本属性。再来看body，他有两条语句，分别是声明语句和return语句。先看声明语句，他由变量b和一条表达式语句组成，表达式语句由三个元素：a,+,1组成。而return语句则由元素b组成。我们可以依照上述并按照节点与分支的组合描绘出这段代码的AST的大致结构如下。



在真正的AST中，每个节点都有自己的type以及一系列相关属性来描述它，那么真正的AST长什么样子？我们可以借助一个工具astexplorer，上述代码的AST结构如下。

2、AST生成

AST的生成是个复杂度极高过程，我们只关心一个关键概念——**编译**，以及两个关键步骤——**词法分析，语法分析**，下面对其做简单介绍。

- 什么是编译?

编译，就是把一门编程语言转成另一门编程语言的过程，一般是指高级语言到低级语言。

我们平时开发使用的开发语言写出的代码计算机无法直接识别，计算机能直接识别的程序语言或指令代码是机器语言。而将高级语言转化为机器语言的过程就是编译的过程，与将英语翻译成汉语是一个道理。那什么是低级语言，什么又是高级语言?

低级语言：描述指令具体在机器上的执行过程，与硬件和执行细节有关，会操作寄存器、内存，需要开发者理解熟悉计算机的工作原理，熟悉具体的执行细节，无需经过翻译，每一操作码在计算机内部都有相应的电路来完成它。

高级语言：高级语言有很多用于描述逻辑的语言特性，如分支、循环、函数、面向对象等，接近人的思维，可以让开发者快速的通过它来表达各种逻辑。比如 c++、javascript。计算机无法直接识别高级语言，它需要被编译成低级语言的指令才能被执行，这个过程就是编译。

- 编译的过程

编译的本质就是转换，而转换的前提则是要理解被转换的东西，前面提到编译器通过AST理解高级语言代码，因此编译的第一步就是解析源代码，得到AST。具体来讲这个解析的过程分为如下几步：

词法分析

何为词法？词法组成语言的单词， 是语言中最小单元。我们写的高级语言代码 ，本质上就是一段文本，只不过是按照一定的格式组织的描述逻辑的文本。 因此词法可以理解成我们代码中一系列独立的单词，var，for ，if，while等。词法分析的过程就是读取代码，识别每一个单词及其种类，将它们按照预定的规则合并成一个个的标识，也叫 token，同时，它会移除空白符，注释，等，最终产出一个token数组。即词法分析阶段把会字符串形式的代码转换为 **令牌（tokens）** 流，用一段伪代码举例：

TypeScript | 复制代码

```
const a = 10;
[
  { type: "KEYWORD_CONST", value: "const" }, { type: "VARIABLE", value: "a" },
  { type: "OPERATOR_EQUAL", value: "=" }, { type: "INTEGER", value: "10" }
  ...
]
```

语法分析

语法，是词法之间的组合方式。前面说到，我们写的源程序是按照一定的格式组织的描述逻辑的文本，而所谓描述逻辑的格式就是指语法。语法分析的任务就是用由词法分析得到的令牌流，在上下文无关文法（一般指某种程序设计语言上的语法）的约束下，生成树形的中间表示（便于描述逻辑结构），该中间表示给出了令牌流的结构表示，同时验证语法，语法如果有错的话，抛出语法错误。

经过词法、语法分析之后就产生了AST，用一棵树形的数据结构来描述源代码，从这里开始就是计算机可以理解的了。有了AST，就可以根据不用需求进行不同操作，如编译器会将AST转换成线性中间代码，生成汇编代码，最后生成机器码。解释器会将AST解释执行或转成线性的中间代码再解释执行。转译器则会将AST转换为另一个AST，再生成目标代码，例如Babel就是一个典型的Javascript转译器，其主要能力是将ES6+代码转换成兼容旧的浏览器或环境的js代码，我们今天也会利用Babel的能力进行AST操作，关于编译的后续步骤如语义分析，代码优化，代码生成等这里就不再过多讨论，接下来具体了解AST。

如何处理AST

要对AST做处理，要清楚他的基本结构，节点类型，这将是我们基于AST进行实际应用的基础。 首先我们回顾前文的AST结构。我们会注意到，AST 的每一层都拥有近乎相同的结构，都有一个type属性以及一系列描述属性，type属性用来表示节点的类型（CallExpression,Identifier,MemberExpression等等）。这样的每一层结构称为一个 **节点（Node）** 。 一个 AST 可以由单一的节点或是成百上千个节点构成。 抽象语法树有一套约定的规范：GitHub – estree/estree: The ESTree Spec，社区称为 estree。借助这个约定的 AST 规范，整个前端社区，生产类工具统一产出该格式的数据结构而无需关心下游，消费类工具统一使用该格式进行处理而无需关心上游。 AST的所有节点类型可分为以下几个大类：字面量、标识符、表达式、语句、模块语法，每个大类下又分类多个子类，下面介绍一些基本且开发常用的节点类型 ，更全面的信息可以查文档或者在ASTExplorer中具体查看。

Literal 字面量

- StringLiteral 字符串字面量（"foo"）
- NumericLiteral 数值字面量（123）
- BooleanLiteral 布尔字面量 （true）
- TemplateLiteral 模板字面量 （${obj}）

dentifier 标识符

标识符即各种声明与引用的名字，js中的变量名，函数名，属性名等都是标识符。如下面代码中的bar,foo,num都是标识符。

```TypeScript
const bar = foo(num)
```

Statement 语句

是一段可以**独立执行**的代码。下面代码的每一行都是一条语句。

```typescript
const a = 1;
console.log(a);
export default a;
```

Statement 分为众多子类型，下面举几个例子。

```typescript
return a; // ReturnStatement
try {
  // TryStatement
} catch (error) {}
for (let index = 0; index < array.length; index++) {
  // ForStatement
  const element = array[index];
}
while (condition) {} // WhileStatement
```

Declaration 声明语句

他是一种特殊的语句，用于在作用域内声明变量、函数、class、import、export 等，同样有众多子类型。

```typescript
const a = 1; // VariableDeclaration
function b(){} // FunctionDeclaration
class C {} // ClassDeclaration
```

Expression 表达式

表达式与语句的区别是表达式执行后会有返回结果，举例：

```typescript
a = 1; // AssignmentExpression
a+b; // BinaryExpression
this; // ThisExpression
```

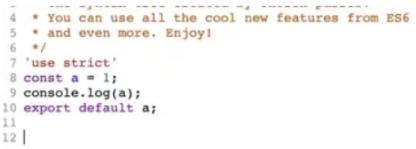
Modules ES module模块语法

```typescript
import name from 'name'; // ImportDeclaration
export const newName = 'newName'; // ExportNamedDeclaration
export default name; // ExportDefaultDeclaration
export * from 'name'; // ExportAllDeclaration

```

Program & Directive

program 是代表整个程序的节点，它包裹了所有具体执行语句的节点，而Directive则是代码中的指令部分。



以上就是AST的一些基本结构，接下来看一下我们可以用哪些工具来生成AST，以及AST都有哪些使用场景。

3、常用的AST库

https://github.com/acornjs/acorn

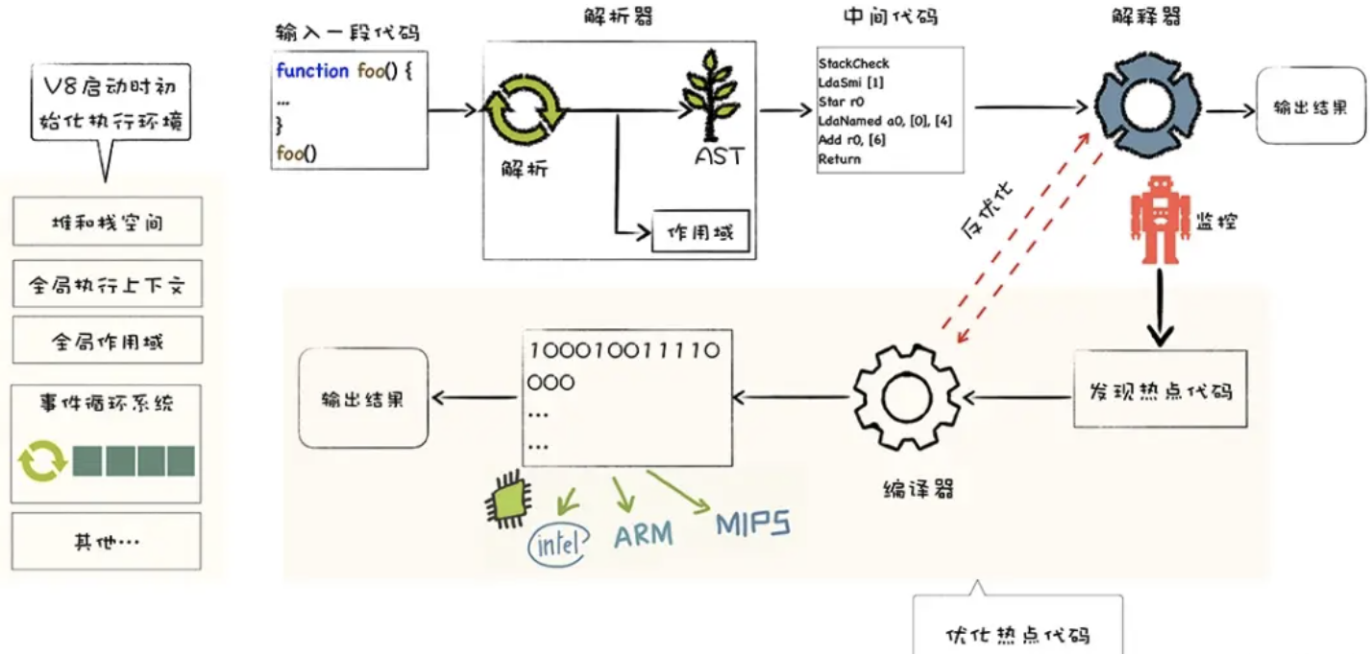
4、AST的应用场景

- 代码高亮、格式化、错误提示、自动补全等：ESlint、Prettier、Vetur等。
- 代码压缩混淆：uglifyJS等。
- 代码转译：webpack、babel、TypeScript等。

5、Babel原理

Babel简述

babel 是一个 JavaScript 编译器，具体来说，babel 是一个工具用于将 ECMAScript 2015+ 语法编写的代码转化成向后兼容的 JavaScript 语法，以便于能够运行在当前或者旧版本浏览器或其他环境中。

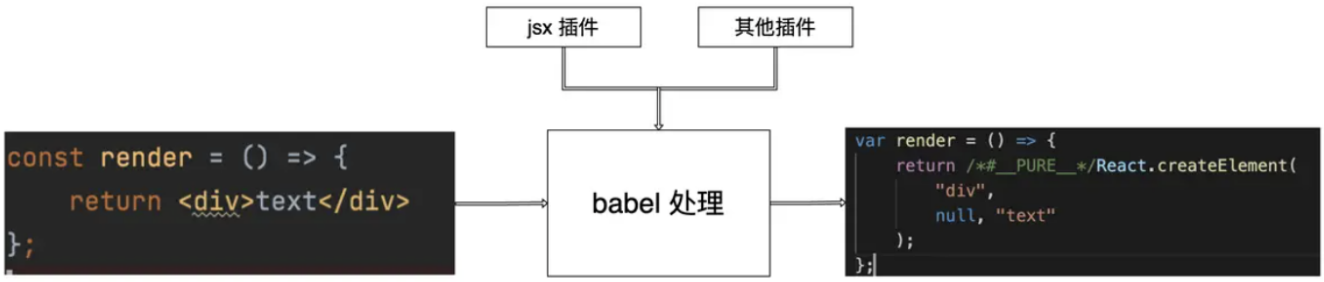


如图所示先来回顾下 v8 执行 js 代码的整体流程，有一段 js 代码经过词法分析将代码解析成几个 token，然后经过语法分析生成一棵 AST 数，再经过语义分析解析出作用域、领域等上下文环境。最终生成中间代码的形态(中间代码可能是分词表以及上下文对象等形态组成)，再放入到主线程中，加载相关的全局作用域、上下文等环境要素进行代码执行，在执行过程中还可以对代码进行优化，比如 JIT、延迟解析、隐藏类、内联缓存等技术。

但是如果 v8 解析器版本低不支持对新语法比如 class 解析，那么会在语法分析阶段报错表示解析器不知道怎么处理该语法，所以我们需要提前的将高级语法转成低级别语法，以便于 v8 解析器中识别，比如说可以将 class 转成使用基于函数原型的方式模拟类。所以说，babel 就是一种将 JavaScript 语言高级语

法转化成低级语法的工具。这个转化过程是在开发打包发布代码阶段就完成的，用户浏览器中运行的代码是转化后的低级别的代码。

工作原理



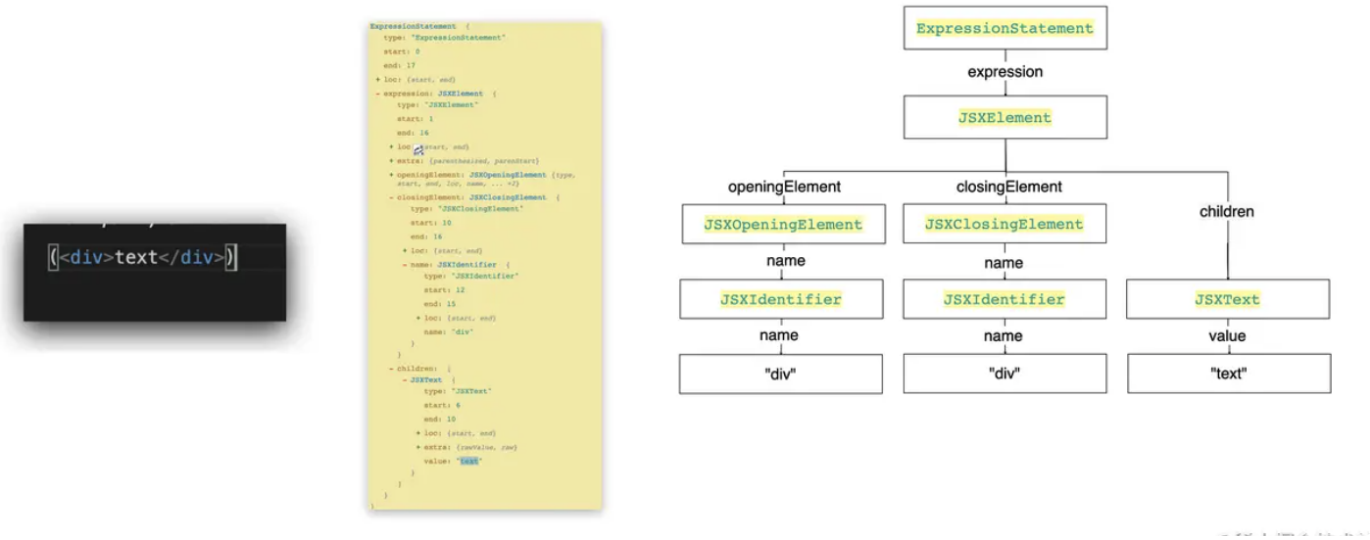
如图所示，输入的是 jsx 代码 jsx v8 是解析不了的， 输出的是可以被 v8 解析的使用 createElement 函数创建的节点。（此处可以忽略 react 相关的东西，举 jsx 例子更能凸显出 babel 各个阶段的特点）babel 核心也仅仅是提供了一个转化的框架，具体需要转化什么语法，怎么转化取决于各种的插件。

babel 大概的执行流程也是词法分析、语法分析构建出 AST 树，修改 AST 树然后转成字符串输出的过程。babel 将这个处理流程抽象出来，抽象成了 **parse、traverse、generate** 三个过程，同时对于每一个过程中都有相应的暴露钩子函数，如何对某一种语法怎么处理由插件提供，插件对钩子函数进行了实现。

Parse

parse 阶段是最基本的阶段，主要工作是把输入的源代码字符串你经过词法分析、语法分析转化成能够后续处理的 AST 中间代码表示结构，AST 遵循 estree 规范，同时也可以使用 astesplorer 网站清晰的看出

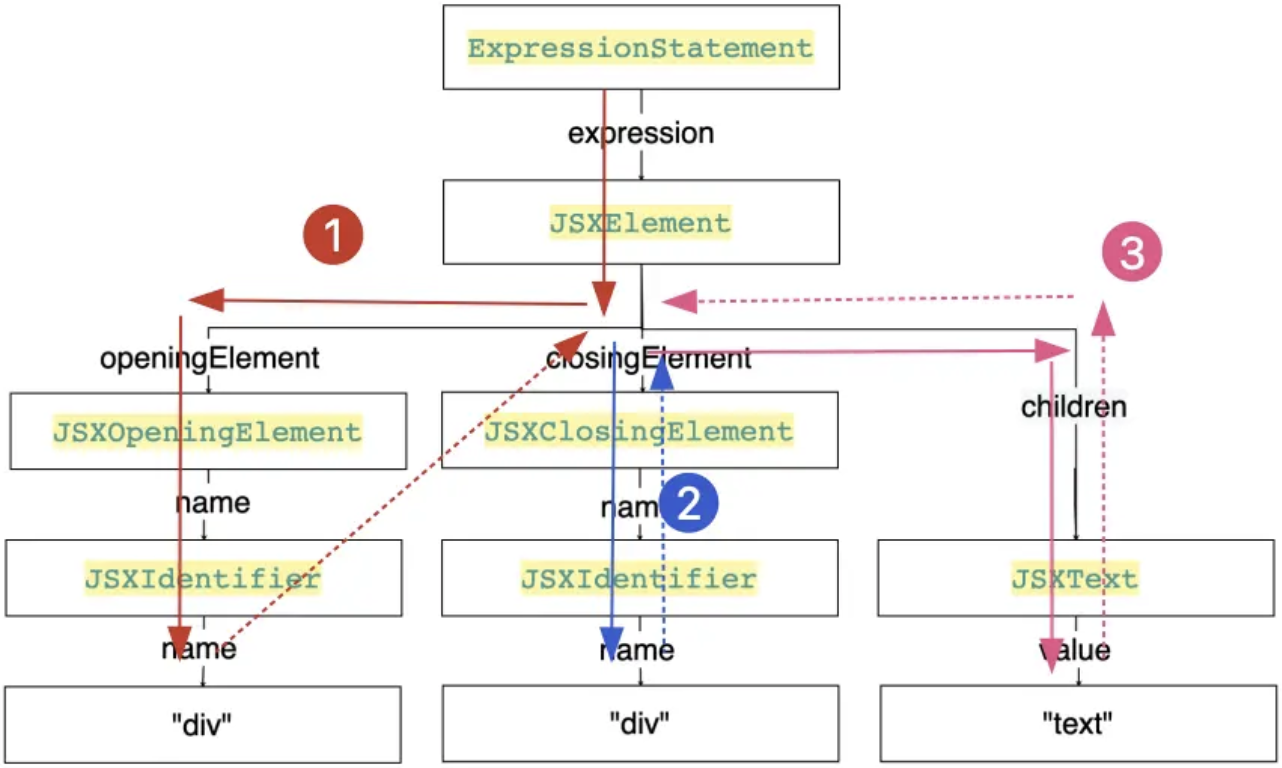
源代码解析成 ast 的结果。比如对 jsx 代码的解析：



对代码 (**< div >text</ div >**) 经过 parse 的词法分析、语法分析后得到 AST 的表示如图所示，一个完整的 AST 是有很多的单元组成的，比如图中 JSXElement、JSXOpeningElement、JSXClosingElement、JSXIdentifier、JSXText 单元。每一个单元都有一些共有属性和特有属性，共有属性有 描述类型的 type、描述词语所在文件中的开始结束位置 start、end 方便与后续代码 sourcemap 生成代码定位。特有属性不同的单元会不一样，比如 JSXElement 有开始标签 openingElement、结束标签 closingElement，以及孩子节点 children。

Traverse

traverse 阶段主要是对 parse 阶段生成的 AST 进行深度优先遍历，遍历的方式根据 type 类型决定分支是什么从而往下遍历。例如：

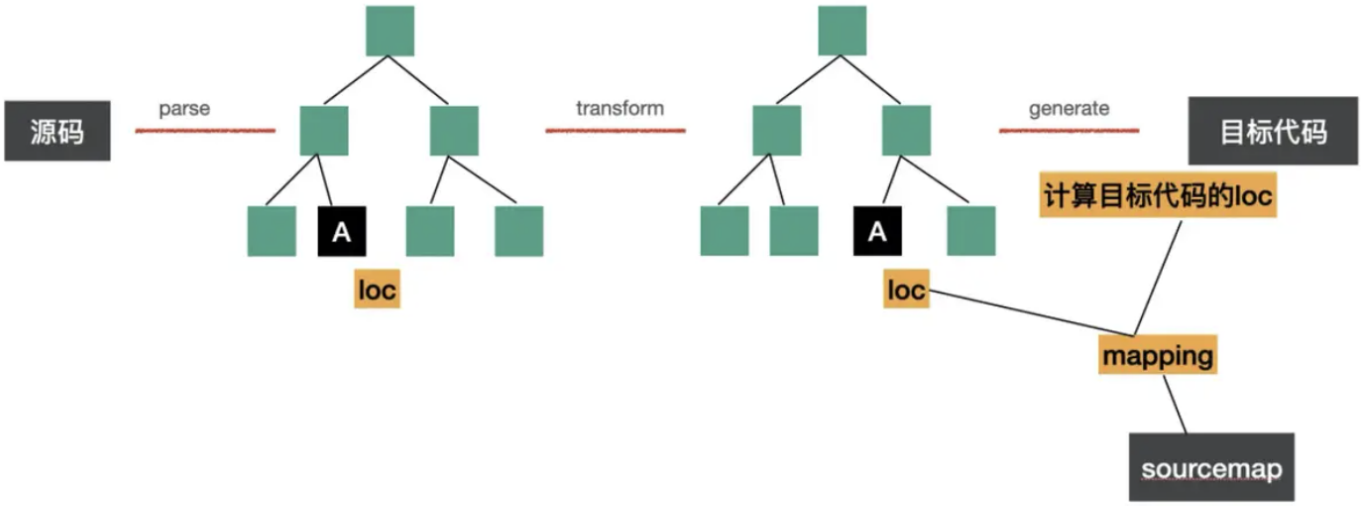


遍历到 JSXElement 单元的时候，读取 type 值是 "JSXElement" 所以知道了往下遍历的又 openingElement、closingelement、children 属性，从该节点开始深度遍历，先遍历 openingelement 单元，同理查看 type 是 "JSXOpengingElement"类型所以 name 字段是往下遍历的属性，所以继续遍历 name，依次往下遍历直到 "div"是一个字符串不是单元位置，然后回溯继续对 closingElement、children 属性进行深度遍历。

Generate

generate 能够将 traverse 遍历修改的 AST 转化成源码的模块，只不过在遍历 AST 过程中会根据每一个单元的 type 类型调用不用的 generate 函数输出不同的源代码。

generator 可以配置是否需要输出 sourcemap，这是 generator 另一个比较重要的点。对于 sourcemap 就是一种编译后得到的代与源码的映射，sourcemap 存在的好处一是开发的时候可以方便的快速定位源码位置，二是上线的时候通过对源代码和对应的 sourcemap 分开部署，捕获线上的错误根据 sourcemap 就可以方便的定位源码位置。



6、Babel插件开发

Plugin的概念

根据前面描述的 babel 工作流程大概能够知道 plugin 的作用。babel 经过 parse traverse generate 几个抽象的过程，将高级别源码转化成低级别源码进行了抽象，而 plugin 就是对具体语法转化的实现。下面是一个插件的例子：

TypeScript | 复制代码

```typescript
import { declare } from "@babel/helper-plugin-utils";

export default declare(api => {
  api.assertVersion(7);

  return {
    name: "syntax-jsx",

    manipulateOptions(opts, parserOpts) {
      const { plugins } = parserOpts;
      // If the Typescript plugin already ran, it will have decided whether
      // or not this is a TSX file.
      if (plugins.some(p => (Array.isArray(p) ? p[0] : p) === "typescript")) {
        return;
      }

      plugins.push("jsx");
    },
  };
});
```

插件返回的是一个被 declare 加工后的对象，比较重要的是传入的函数，该函数返回一个描述对象这里面就是对 babel 抽象处理过程中关键的钩子进行实现。name 说明该插件的唯一标识。

babel 首先经过 parse 对高级源码进行词法分析和语法分析转成 AST，在 manipulateOptions 钩子中事先表明在 parse 过程中需要将 jsx 风格的语法进行识别。在 traverse 过程中对 AST中的每一个单元做处理，处理的过程就是在 visitor 对象中，其中对象的 key 就是处理单元的 type，传入该单元的 path 描述执行对应 key 的函数就能够对高级源码转成低级别实现，然后再经过 generate 输出成字符串。

Plugin原理

@babel/parser 解析源码得到AST。

@babel/traverse 遍历 AST。

@babel/types 用于构建AST节点和校验AST节点类型；

@babel/generate 打印 AST，生成目标代码和sorucemap。

babel的处理步骤

主要有三个阶段：解析（parse），转换 （transform），生成（generate）。

parse

将源码转成 AST，用到@babel/parser模块。

transform

对AST 进行遍历，在此过程中对节点进行添加、更新及移除等操作。因此这是bebel处理代码的核心步骤，是我们的讨论重点，主要使用@babel/traverse和@babel/types模块。

generate

打印 AST 成目标代码并生成 sourcemap，用到@babel/generate模块。 接下来我们来重点了解转换这一步，上面我们提到，转换的第一步是遍历AST。说到这里就不得不提到一个设计模式——**访问者模式**。

访问者模式

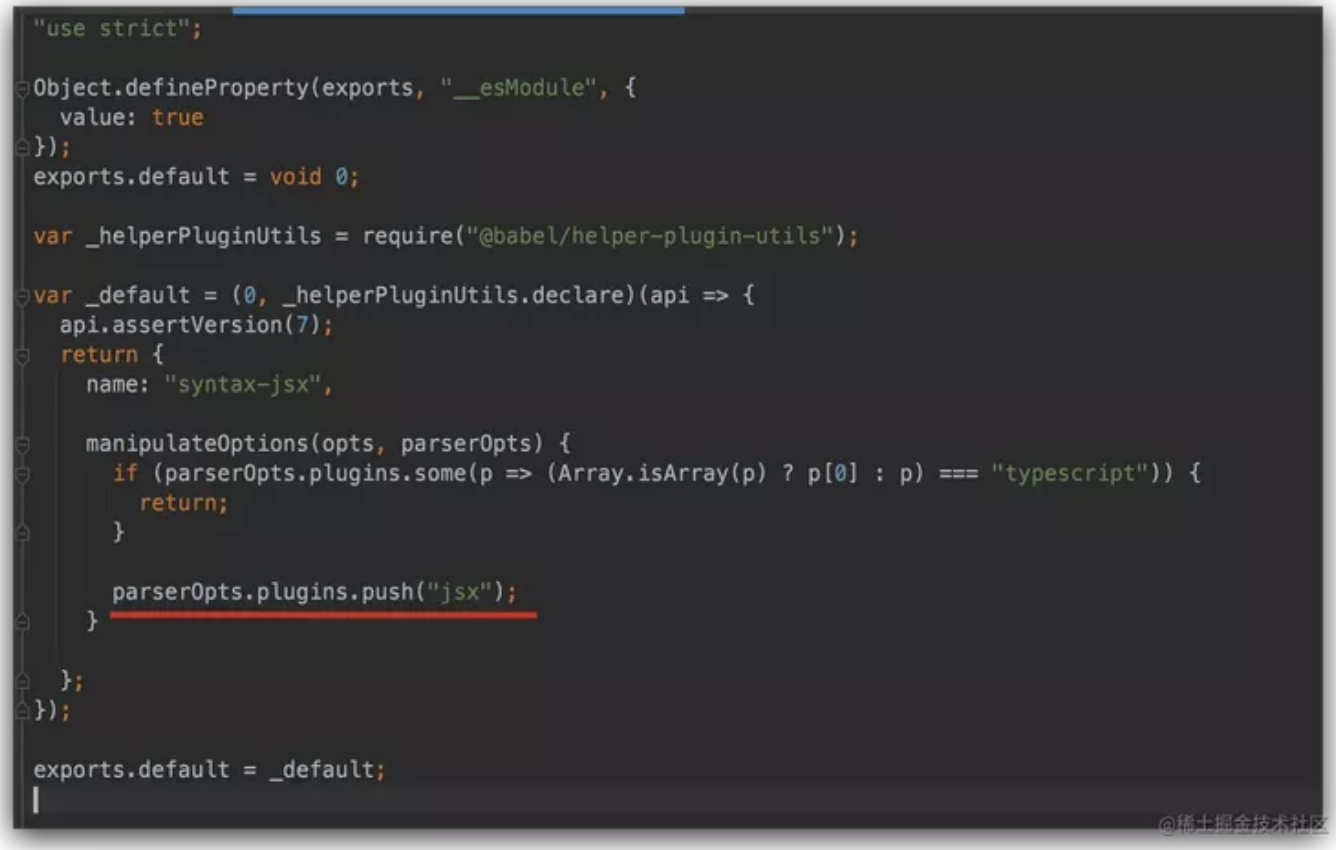
在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了目标元素的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。而在当前场景下，访问者即是一个用于 AST 遍历的模式， 简单的说它就是一个对象，定义了用于在一个树状结构中获取具体节点的方法。当访问者把它用于遍历中时，每当在树中遇见一个对应类型，都会调用该类型对应的方法。

Plugin分类

babel 根据解析过程，以及预言特性的分类，将 plugin分为了三大类型分别是 syntax、transform、proposal。如果使用过 babel 会在项目的 node_modules @babel 文件夹中发现一些 plugin-syntax-*、*plugin-transform-**、plugin-proposal-** 开头的插件，下面会介绍这些插件的不同。

1. syntax

syntax 插件只是对 manipulateOptions 钩子函数的实现，目的就是让 babel 在解析过程中能够对特定语法的支持，避免解析不出来报错。比如 parse 章节中介绍的



在 manipulateOptions中 parserOpts 对象中 plugins 属性放入 jsx 标识，在 parse过程中就会对 <> 这种类似的语法解析成标签而不是 小于号运算符从而 babel 能够解析通过而不报错。当然配置 syntax 可以与 具体 traverse 中使用的 visitor 实现是分开的，因为语法识别出来也可以不需要解析成 低级语法形式。或许你运行的环境就支持 jsx 语法呢~

2. transform

transform plugin 中大多是是对高级语法的转化实现，比如各种 es20xx 语言特性、typescript、jsx 等语言特性等。在这种 plugin 中是对 visitor 对象不同的 AST 单元的具体转化动作实现，所以前提是 parse 阶段需要对解析语法识别，所以一般 @babel/plugin-transform-** 都会引用继承 @babel/plugin-syntax-**。

3. proposal

未加入语言标准的特性的 AST 转换插件叫 proposal plugin，其实他也是 transform plugin，但是为了和标准特性区分，所以这样叫。
完成 proposal 特性的支持，有时同样需要 综合 syntax plugin 和 proposal plugin，比如 function bind（:: 操作符）就需要同时使用 @babel/plugin-syntax-function-bind 和 @babel/plugin-proposal-function-bind。

插件开发实战