Babel 插件手册

这篇文档涵盖了如何创建 Babel 插件等方面的内容。.

这本手册提供了多种语言的版本，查看 自述文件 里的完整列表。

目录

介绍

Babel 是一个通用的多功能的 JavaScript 编译器。此外它还拥有众多模块可用于不同形式的静态分析。

> 静态分析是在不需要执行代码的前提下对代码进行分析的处理过程（执行代码的同时进行代码分析即是动态分析）。 静态分析的目的是多种多样的， 它可用于语法检查，编译，代码高亮，代码转换，优化，压缩等等场景。

你可以使用 Babel 创建多种类型的工具来帮助你更有效率并且写出更好的程序。

> ***在 Twitter 上关注 @thejameskyle，第一时间获取更新。***

基础

Babel 是 JavaScript 编译器，更确切地说是源码到源码的编译器，通常也叫做“转换编译器（transpiler）”。 意思是说你为 Babel 提供一些 JavaScript 代码，Babel 更改这些代码，然后返回给你新生成的代码。

抽象语法树（ASTs）

这个处理过程中的每一步都涉及到创建或是操作抽象语法树，亦称 AST。

> Babel 使用一个基于 ESTree 并修改过的 AST，它的内核说明文档可以在这里找到。.

```
function square(n) {
  return n * n;
}
```

> AST Explorer 可以让你对 AST 节点有一个更好的感性认识。 这里是上述代码的一个示例链接。

这个程序可以被表示成如下的一棵树：

```
- FunctionDeclaration:
  - id:
    - Identifier:
      - name: square
  - params [1]
    - Identifier
      - name: n
  - body:
    - BlockStatement
      - body [1]
        - ReturnStatement
          - argument
            - BinaryExpression
              - operator: *
              - left
                - Identifier
                  - name: n
              - right
                - Identifier
                  - name: n
```

或是如下所示的 JavaScript Object（对象）：

```
{
  type: "FunctionDeclaration",
  id: {
    type: "Identifier",
    name: "square"
  },
  params: [{
    type: "Identifier",
    name: "n"
  }],
  body: {
    type: "BlockStatement",
    body: [{
      type: "ReturnStatement",
      argument: {
        type: "BinaryExpression",
        operator: "*",
        left: {
          type: "Identifier",
          name: "n"
        },
        right: {
          type: "Identifier",
          name: "n"
        }
```

```
      }
    }]
  }
}
```

你会留意到 AST 的每一层都拥有相同的结构：

```
{
  type: "FunctionDeclaration",
  id: {...},
  params: [...],
  body: {...}
}
```

```
{
  type: "Identifier",
  name: ...
}
```

```
{
  type: "BinaryExpression",
  operator: ...,
  left: {...},
  right: {...}
}
```

> 注意：出于简化的目的移除了某些属性

这样的每一层结构也被叫做 **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。 它们组合在一起可以描述用于静态分析的程序语法。

每一个节点都有如下所示的接口（Interface）：

```
interface Node {
  type: string;
}
```

字符串形式的 `type` 字段表示节点的类型（如： `"FunctionDeclaration"`，`"Identifier"`，或 `"BinaryExpression"`）。 每一种类型的节点定义了一些附加属性用来进一步描述该节点类型。

Babel 还为每个节点额外生成了一些属性，用于描述该节点在原始代码中的位置。

```
{
  type: ...,
  start: 0,
  end: 38,
  loc: {
    start: {
      line: 1,
      column: 0
    },
    end: {
      line: 3,
      column: 1
    }
```

```
  },
  ...
}
```

每一个节点都会有 `start`，`end`，`loc` 这几个属性。

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）**。.

解析

解析步骤接收代码并输出 AST。 这个步骤分为两个阶段：**词法分析（Lexical Analysis）** 和 **语法分析（Syntactic Analysis）**。.

词法分析

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。.

语法分析

语法分析阶段会把一个令牌流转换成 AST 的形式。 这个阶段会使用令牌中的信息把它们转换成一个 AST 的表述结构，这样更易于后续的操作。

转换

转换步骤接收 AST 并对其进行遍历，在此过程中对节点进行添加、更新及移除等操作。 这是 Babel 或是其他编译器中最复杂的过程 同时也是插件将要介入工作的部分，这将是本手册的主要内容， 因此让我们慢慢来。

生成

代码生成步骤把最终（经过一系列转换之后）的 AST 转换成字符串形式的代码，同时还会创建源码映射（source maps）。.

代码生成其实很简单：深度优先遍历整个 AST，然后构建可以表示转换后代码的字符串。

遍历

想要转换 AST 你需要进行递归的树形遍历。

比方说我们有一个 `FunctionDeclaration` 类型。它有几个属性：`id`，`params`，和 `body`，每一个都有一些内嵌节点。

```
{
  type: "FunctionDeclaration",
  id: {
    type: "Identifier",
    name: "square"
  },
  params: [{
    type: "Identifier",
    name: "n"
  }],
  body: {
    type: "BlockStatement",
    body: [{
      type: "ReturnStatement",
      argument: {
        type: "BinaryExpression",
        operator: "*",
        left: {
          type: "Identifier",
          name: "n"
        },
        right: {
          type: "Identifier",
          name: "n"
        }
      }
    }]
  }
}
```

于是我们从 `FunctionDeclaration` 开始并且我们知道它的内部属性（即：`id`，`params`，`body`），所以我们依次访问每一个属性及它们的子节点。

接着我们来到 `id`，它是一个 `Identifier`。`Identifier` 没有任何子节点属性，所以我们继续。

之后是 `params`，由于它是一个数组节点所以我们访问其中的每一个，它们都是 `Identifier` 类型的单一节点，然后我们继续。

此时我们来到了 `body`，这是一个 `BlockStatement` 并且也有一个 `body` 节点，而且也是一个数组节点，我们继续访问其中的每一个。

这里唯一的一个属性是 `ReturnStatement` 节点，它有一个 `argument`，我们访问 `argument` 就找到了 `BinaryExpression`。.

`BinaryExpression` 有一个 `operator`，一个 `left`，和一个 `right`。 Operator 不是一个节点，它只是一个值因此我们不用继续向内遍历，我们只需要访问 `left` 和 `right`。.

Babel 的转换步骤全都是这样的遍历过程。

Visitors（访问者）

当我们谈及"进入"一个节点，实际上是说我们在**访问**它们， 之所以使用这样的术语是因为有一个[访问者模式（visitor）]的概念。.

访问者是一个用于 AST 遍历的跨语言的模式。 简单的说它们就是一个对象，定义了用于在一个树状结构中获取具体节点的方法。 这么说有些抽象所以让我们来看一个例子。

```
const MyVisitor = {
  Identifier() {
    console.log("Called!");
  }
};

// 你也可以先创建一个访问者对象，并在稍后给它添加方法。
let visitor = {};
visitor.MemberExpression = function() {};
visitor.FunctionDeclaration = function() {}
```

> **注意**：`Identifier() { ... }` 是 `Identifier: { enter() { ... } }` 的简写形式。.

这是一个简单的访问者，把它用于遍历中时，每当在树中遇见一个 `Identifier` 的时候会调用 `Identifier()` 方法。

所以在下面的代码中 `Identifier()` 方法会被调用四次（包括 `square` 在内，总共有四个 `Identifier`）。).

```
function square(n) {
  return n * n;
}
```

```
path.traverse(MyVisitor);
Called!
Called!
Called!
Called!
```

这些调用都发生在**进入**节点时，不过有时候我们也可以在**退出**时调用访问者方法。.

假设我们有一个树状结构：

```
- FunctionDeclaration
  - Identifier (id)
  - Identifier (params[0])
  - BlockStatement (body)
    - ReturnStatement (body)
      - BinaryExpression (argument)
        - Identifier (left)
        - Identifier (right)
```

当我们向下遍历这颗树的每一个分支时我们最终会走到尽头，于是我们需要往上遍历回去从而获取到下一个节点。 向下遍历这棵树我们**进入**每个节点，向上遍历回去时我们**退出**每个节点。

让我们以上面那棵树为例子走一遍这个过程。

- 进入 `FunctionDeclaration`
  - 进入 `Identifier (id)`
  - 走到尽头
  - 退出 `Identifier (id)`
  - 进入 `Identifier (params[0])`
  - 走到尽头
  - 退出 `Identifier (params[0])`
  - 进入 `BlockStatement (body)`
  - 进入 `ReturnStatement (body)`
    - 进入 `BinaryExpression (argument)`
    - 进入 `Identifier (left)`
      - 走到尽头
    - 退出 `Identifier (left)`
    - 进入 `Identifier (right)`
      - 走到尽头
    - 退出 `Identifier (right)`
    - 退出 `BinaryExpression (argument)`
  - 退出 `ReturnStatement (body)`
  - 退出 `BlockStatement (body)`
- 退出 `FunctionDeclaration`

所以当创建访问者时你实际上有两次机会来访问一个节点。

```
const MyVisitor = {
  Identifier: {
    enter() {
      console.log("Entered!");
    },
    exit() {
      console.log("Exited!");
    }
  }
};
```

如有必要，你还可以把方法名用`|`分割成`Idenfifier |MemberExpression`形式的字符串，把同一个函数应用到多种访问节点。.

在flow-comments 插件中的例子如下：

```
const MyVisitor = {
  "ExportNamedDeclaration|Flow"(path) {}
};
```

你也可以在访问者中使用别名(如babel-types定义).

例如，

`Function` is an alias for `FunctionDeclaration`, `FunctionExpression`, `ArrowFunctionExpression`, `ObjectMethod` and `ClassMethod`.

```
const MyVisitor = {
  Function(path) {}
};
```

Paths（路径）

AST 通常会有许多节点，那么节点直接如何相互关联呢？ 我们可以使用一个可操作和访问的巨大可变对象表示节点之间的关联关系，或者也可以用**Paths**（路径）来简化这件事情。.

Path 是表示两个节点之间连接的对象。

例如，如果有下面这样一个节点及其子节点：

```
{
  type: "FunctionDeclaration",
  id: {
    type: "Identifier",
    name: "square"
  },
  ...
}
```

将子节点`Identifier`表示为一个路径（Path）的话，看起来是这样的：

```
{
  "parent": {
    "type": "FunctionDeclaration",
    "id": {...},
    ....
  },
  "node": {
    "type": "Identifier",
    "name": "square"
  }
}
```

同时它还包含关于该路径的其他元数据：

```
{
```

```
  "parent": {...},
  "node": {...},
  "hub": {...},
  "contexts": [],
  "data": {},
  "shouldSkip": false,
  "shouldStop": false,
  "removed": false,
  "state": null,
  "opts": null,
  "skipKeys": null,
  "parentPath": null,
  "context": null,
  "container": null,
  "listKey": null,
  "inList": false,
  "parentKey": null,
  "key": null,
  "scope": null,
  "type": null,
  "typeAnnotation": null
}
```

当然路径对象还包含添加、更新、移动和删除节点有关的其他很多方法，稍后我们再来看这些方法。

在某种意义上，路径是一个节点在树中的位置以及关于该节点各种信息的响应式 **Reactive** 表示。 当你调用一个修改树的方法后，路径信息也会被更新。 Babel 帮你管理这一切，从而使得节点操作简单，尽可能做到无状态。

Paths in Visitors（存在于访问者中的路径）

当你有一个 `Identifier()` 成员方法的访问者时，你实际上是在访问路径而非节点。 通过这种方式，你操作的就是节点的响应式表示（译注：即路径）而非节点本身。

```
const MyVisitor = {
  Identifier(path) {
    console.log("Visiting: " + path.node.name);
  }
};
```

```
a + b + c;
```

```
path.traverse(MyVisitor);
Visiting: a
Visiting: b
Visiting: c
```

State（状态）

状态是抽象语法树AST转换的敌人，状态管理会不断牵扯你的精力，而且几乎所有你对状态的假设，总是会有一些未考虑到的语法最终证明你的假设是错误的。

考虑下列代码：

```
function square(n) {
  return n * n;
}
```

让我们写一个把 `n` 重命名为 `x` 的访问者的快速实现.

```
let paramName;

const MyVisitor = {
  FunctionDeclaration(path) {
    const param = path.node.params[0];
    paramName = param.name;
    param.name = "x";
  },

  Identifier(path) {
    if (path.node.name === paramName) {
      path.node.name = "x";
    }
  }
};
```

对上面的例子代码这段访问者代码也许能工作，但它很容易被打破：

```
function square(n) {
  return n * n;
}
n;
```

更好的处理方式是使用递归，下面让我们来像克里斯托佛·诺兰的电影盗梦空间那样来把一个访问者放进另外一个访问者里面。

```
const updateParamNameVisitor = {
  Identifier(path) {
    if (path.node.name === this.paramName) {
      path.node.name = "x";
    }
  }
};

const MyVisitor = {
  FunctionDeclaration(path) {
    const param = path.node.params[0];
    const paramName = param.name;
    param.name = "x";

    path.traverse(updateParamNameVisitor, { paramName });
  }
};

path.traverse(MyVisitor);
```

当然，这只是一个刻意编写的例子，不过它演示了如何从访问者中消除全局状态。

Scopes（作用域）

接下来让我们介绍**作用域（scope）**的概念。 JavaScript 支持词法作用域，在树状嵌套结构中代码块创建出新的作用域。

```
// global scope

function scopeOne() {
  // scope 1

  function scopeTwo() {
    // scope 2
  }
}
```

在 JavaScript 中，每当你创建了一个引用，不管是通过变量（variable）、函数（function）、类型（class）、参数（params）、模块导入（import）还是标签（label）等，它都属于当前作用域。

```
var global = "I am in the global scope";

function scopeOne() {
  var one = "I am in the scope created by `scopeOne()`";

  function scopeTwo() {
    var two = "I am in the scope created by `scopeTwo()`";
  }
}
```

更深的内部作用域代码可以使用外层作用域中的引用。

```
function scopeOne() {
  var one = "I am in the scope created by `scopeOne()`";

  function scopeTwo() {
    one = "I am updating the reference in `scopeOne` inside `scopeTwo`";
  }
}
```

内层作用域也可以创建和外层作用域同名的引用。

```
function scopeOne() {
  var one = "I am in the scope created by `scopeOne()`";

  function scopeTwo() {
    var one = "I am creating a new `one` but leaving reference in `scopeOne()`
alone.";
  }
}
```

当编写一个转换时，必须小心作用域。我们得确保在改变代码的各个部分时不会破坏已经存在的代码。

我们在添加一个新的引用时需要确保新增加的引用名字和已有的所有引用不冲突。 或者我们仅仅想找出使用一个变量的所有引用， 我们只想在给定的作用域（Scope）中找出这些引用。

作用域可以被表示为如下形式：

```
{
  path: path,
  block: path.node,
  parentBlock: path.parent,
  parent: parentScope,
  bindings: [...]
}
```

当你创建一个新的作用域时，需要给出它的路径和父作用域，之后在遍历过程中它会在该作用域内收集所有的引用(“绑定”)。

一旦引用收集完毕，你就可以在作用域（Scopes）上使用各种方法，稍后我们会了解这些方法。

Bindings（绑定）

所有引用属于特定的作用域，引用和作用域的这种关系被称作：**绑定（binding）**。.

```
function scopeOnce() {
  var ref = "This is a binding";

  ref; // This is a reference to a binding

  function scopeTwo() {
    ref; // This is a reference to a binding from a lower scope
  }
}
```

单个绑定看起来像这样：

```
Text for Translation
{
  identifier: node,
  scope: scope,
  path: path,
  kind: 'var',

  referenced: true,
  references: 3,
  referencePaths: [path, path, path],

  constant: false,
  constantViolations: [path]
}
```

有了这些信息你就可以查找一个绑定的所有引用，并且知道这是什么类型的绑定(参数，定义等等)，查找它所属的作用域，或者拷贝它的标识符。 你甚至可以知道它是不是常量，如果不是，那么是哪个路径修改了它。

在很多情况下，知道一个绑定是否是常量非常有用，最有用的一种情形就是代码压缩时。

```
function scopeOne() {
  var ref1 = "This is a constant binding";

  becauseNothingEverChangesTheValueOf(ref1);

  function scopeTwo() {
    var ref2 = "This is *not* a constant binding";
    ref2 = "Because this changes the value";
  }
}
```

API

Babel 实际上是一组模块的集合。本节我们将探索一些主要的模块，解释它们是做什么的以及如何使用它们。

> 注意：本节内容不是详细的 API 文档的替代品，正式的 API 文档将很快提供出来。

babylon

Babylon 是 Babel 的解析器。最初是 从Acorn项目fork出来的。Acorn非常快，易于使用，并且针对非标准特性(以及那些未来的标准特性) 设计了一个基于插件的架构。

首先，让我们安装它。

```
$ npm install --save babylon
```

先从解析一个代码字符串开始：

```
import * as babylon from "babylon";

const code = `function square(n) {
  return n * n;
}`;

babylon.parse(code);
// Node {
//   type: "File",
//   start: 0,
//   end: 38,
//   loc: SourceLocation {...},
//   program: Node {...},
//   comments: [],
//   tokens: [...]
// }
```

我们还能像下面这样传递选项给 `parse()` 方法：

```
babylon.parse(code, {
  sourceType: "module", // default: "script"
  plugins: ["jsx"] // default: []
});
```

`sourceType` 可以是 `"module"` 或者 `"script"`，它表示 Babylon 应该用哪种模式来解析。 `"module"` 将会在严格模式下解析并且允许模块定义，`"script"` 则不会。

> **注意：** `sourceType` 的默认值是 `"script"` 并且在发现 `import` 或 `export` 时产生错误。 使用 `scourceType: "module"` 来避免这些错误。

由于 Babylon 使用了基于插件的架构，因此有一个 `plugins` 选项可以开关内置的插件。 注意 Babylon 尚未对外部插件开放此 API 接口，不排除未来会开放此API。

要查看完整的插件列表，请参见 Babylon README文件。.

babel-traverse

Babel Traverse（遍历）模块维护了整棵树的状态，并且负责替换、移除和添加节点。

运行以下命令安装：

```
$ npm install --save babel-traverse
```

我们可以和 Babylon 一起使用来遍历和更新节点：

```
import * as babylon from "babylon";
import traverse from "babel-traverse";

const code = `function square(n) {
  return n * n;
}`;

const ast = babylon.parse(code);

traverse(ast, {
  enter(path) {
    if (
      path.node.type === "Identifier" &&
      path.node.name === "n"
    ) {
      path.node.name = "x";
    }
  }
});
```

babel-types

Babel Types模块是一个用于 AST 节点的 Lodash 式工具库（译注：Lodash 是一个 JavaScript 函数工具库，提供了基于函数式编程风格的众多工具函数），它包含了构造、验证以及变换 AST 节点的方法。 该工具库包含考虑周到的工具方法，对编写处理AST逻辑非常有用。

可以运行以下命令来安装它：

```
$ npm install --save babel-types
```

然后按如下所示来使用：

```
import traverse from "babel-traverse";
import * as t from "babel-types";

traverse(ast, {
  enter(path) {
    if (t.isIdentifier(path.node, { name: "n" })) {
      path.node.name = "x";
    }
  }
});
```

Definitions（定义）

Babel Types模块拥有每一个单一类型节点的定义，包括节点包含哪些属性，什么是合法值，如何构建节点、遍历节点，以及节点的别名等信息。

单一节点类型的定义形式如下：

```
defineType("BinaryExpression", {
  builder: ["operator", "left", "right"],
  fields: {
    operator: {
      validate: assertValueType("string")
    },
    left: {
      validate: assertNodeType("Expression")
    },
    right: {
      validate: assertNodeType("Expression")
    }
  },
  visitor: ["left", "right"],
  aliases: ["Binary", "Expression"]
});
```

Builders（构建器）

你会注意到上面的 `BinaryExpression` 定义有一个 `builder` 字段。.

```
builder: ["operator", "left", "right"]
```

这是由于每一个节点类型都有构造器方法builder，按类似下面的方式使用：

```
t.binaryExpression("*", t.identifier("a"), t.identifier("b"));
```

可以创建如下所示的 AST：

```
{
  type: "BinaryExpression",
  operator: "*",
  left: {
    type: "Identifier",
    name: "a"
  },
  right: {
    type: "Identifier",
    name: "b"
  }
}
```

当打印出来之后是这样的：

```
a * b
```

构造器还会验证自身创建的节点，并在错误使用的情形下会抛出描述性错误，这就引出了下一个方法类型。

Validators（验证器）

`BinaryExpression` 的定义还包含了节点的字段 `fields` 信息，以及如何验证这些字段。

```
fields: {
  operator: {
    validate: assertValueType("string")
  },
  left: {
    validate: assertNodeType("Expression")
  },
  right: {
    validate: assertNodeType("Expression")
  }
}
```

可以创建两种验证方法。第一种是 `isX`。.

```
t.isBinaryExpression(maybeBinaryExpressionNode);
```

这个测试用来确保节点是一个二进制表达式，另外你也可以传入第二个参数来确保节点包含特定的属性和值。

```
t.isBinaryExpression(maybeBinaryExpressionNode, { operator: "*" });
```

这些方法还有一种断言式的版本，会抛出异常而不是返回 `true` 或 `false`。.

```
t.assertBinaryExpression(maybeBinaryExpressionNode);
t.assertBinaryExpression(maybeBinaryExpressionNode, { operator: "*" });
// Error: Expected type "BinaryExpression" with option { "operator": "*" }
```

Converters（变换器）

> [WIP]

babel-generator

Babel Generator模块是 Babel 的代码生成器，它读取AST并将其转换为代码和源码映射（sourcemaps）。

运行以下命令来安装它：

```
$ npm install --save babel-generator
```

然后按如下方式使用：

```
import * as babylon from "babylon";
import generate from "babel-generator";

const code = `function square(n) {
  return n * n;
}`;

const ast = babylon.parse(code);

generate(ast, {}, code);
// {
//   code: "...",
//   map: "..."
// }
```

你也可以给 `generate()` 方法传递选项。.

```
generate(ast, {
  retainLines: false,
  compact: "auto",
  concise: false,
  quotes: "double",
  // ...
}, code);
```

babel-template

babel-template 是另一个虽然很小但却非常有用的模块。 它能让你编写字符串形式且带有占位符的代码来代替手动编码， 尤其是生成的大规模 AST的时候。 在计算机科学中，这种能力被称为准引用（quasiquotes）。

```
$ npm install --save babel-template
```

```
import template from "babel-template";
import generate from "babel-generator";
import * as t from "babel-types";

const buildRequire = template(`
  var IMPORT_NAME = require(SOURCE);
`);
```

```
const ast = buildRequire({
  IMPORT_NAME: t.identifier("myModule"),
  SOURCE: t.stringLiteral("my-module")
});

console.log(generate(ast).code);
```

```
var myModule = require("my-module");
```

编写你的第一个 Babel 插件

现在你已经熟悉了 Babel 的所有基础知识了，让我们把这些知识和插件的 API融合在一起来编写第一个 Babel 插件吧。

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;
```

完美！我们的第一个 Babel 插件。

转换操作

访问

获取子节点的Path

为了得到一个AST节点的属性值，我们一般先访问到该节点，然后利用 `path.node.property` 方法即可。

```
// the BinaryExpression AST node has properties: `left`, `right`, `operator`
BinaryExpression(path) {
  path.node.left;
  path.node.right;
  path.node.operator;
}
```

如果你想访问到该属性内部的`path`，使用path对象的`get`方法，传递该属性的字符串形式作为参数。

```
BinaryExpression(path) {
  path.get('left');
}
Program(path) {
  path.get('body.0');
}
```

检查节点的类型

如果你想检查节点的类型，最好的方式是：

```
BinaryExpression(path) {
  if (t.isIdentifier(path.node.left)) {
    // ...
  }
}
```

你同样可以对节点的属性们做浅层检查：

```
BinaryExpression(path) {
  if (t.isIdentifier(path.node.left, { name: "n" })) {
    // ...
  }
}
```

功能上等价于：

```
BinaryExpression(path) {
  if (
    path.node.left != null &&
    path.node.left.type === "Identifier" &&
    path.node.left.name === "n"
  ) {
    // ...
  }
}
```

检查路径（Path）类型

一个路径具有相同的方法检查节点的类型：

```
BinaryExpression(path) {
  if (path.get('left').isIdentifier({ name: "n" })) {
    // ...
  }
}
```

就相当于：

```
BinaryExpression(path) {
  if (t.isIdentifier(path.node.left, { name: "n" })) {
    // ...
  }
}
```

检查标识符（Identifier）是否被引用

```
Identifier(path) {
  if (path.isReferencedIdentifier()) {
    // ...
  }
}
```

或者：

```
Identifier(path) {
  if (t.isReferenced(path.node, path.parent)) {
    // ...
  }
}
```

找到特定的父路径

有时你需要从一个路径向上遍历语法树，直到满足相应的条件。

对于每一个父路径调用`callback`并将其`NodePath`当作参数，当`callback`返回真值时，则将其`NodePath`返回。.

```
path.findParent((path) => path.isObjectExpression());
```

如果也需要遍历当前节点：

```
path.find((path) => path.isObjectExpression());
```

查找最接近的父函数或程序：

```
path.getFunctionParent();
```

向上遍历语法树，直到找到在列表中的父节点路径

```
path.getStatementParent();
```

获取同级路径

如果一个路径是在一个`Function`/`Program`中的列表里面，它就有同级节点。

- 使用`path.inList`来判断路径是否有同级节点，
- 使用`path.getSibling(index)`来获得同级路径,
- 使用`path.key`获取路径所在容器的索引,
- 使用`path.container`获取路径的容器（包含所有同级节点的数组）
- 使用`path.listKey`获取容器的key

> 这些API用于 babel-minify </>中使用的 transform-merge-sibling-variables </>插件.

```
var a = 1; // pathA, path.key = 0
var b = 2; // pathB, path.key = 1
var c = 3; // pathC, path.key = 2
```

```
```js
export default function({ types: t }) {
  return {
    visitor: {
      VariableDeclaration(path) {
        // if the current path is pathA
        path.inList // true
        path.listKey // "body"
        path.key // 0
        path.getSibling(0) // pathA
        path.getSibling(path.key + 1) // pathB
        path.container // [pathA, pathB, pathC]
      }
    }
  };
```
```

```
}
```

停止遍历

如果你的插件需要在某种情况下不运行，最简单的做法是尽早写回。

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

如果您在顶级路径中进行子遍历，则可以使用2个提供的API方法：

`path.skip()` skips traversing the children of the current path. `path.stop()` stops traversal entirely.

```
outerPath.traverse({
  Function(innerPath) {
    innerPath.skip(); // if checking the children is irrelevant
  },
  ReferencedIdentifier(innerPath, state) {
    state.iife = true;
    innerPath.stop(); // if you want to save some state and then stop traversal,
or deopt
  }
});
```

处理

替换一个节点

```
BinaryExpression(path) {
  path.replaceWith(
    t.binaryExpression("**", path.node.left, t.numberLiteral(2))
  );
}
```

```
  function square(n) {
-   return n * n;
+   return n ** 2;
  }
```

用多节点替换单节点

```
ReturnStatement(path) {
  path.replaceWithMultiple([
    t.expressionStatement(t.stringLiteral("Is this the real life?")),
    t.expressionStatement(t.stringLiteral("Is this just fantasy?")),
    t.expressionStatement(t.stringLiteral("(Enjoy singing the rest of the song
in your head)")),
  ]);
}
```

```
  function square(n) {
-   return n * n;
+   "Is this the real life?";
+   "Is this just fantasy?";
+   "(Enjoy singing the rest of the song in your head)";
  }
```

> **注意：</>当用多个节点替换一个表达式时，它们必须是 声明。 这是因为Babel在更换节点时广泛使用启发式算法，这意味着您可以做一些非常疯狂的转换，否则将会非常冗长。

用字符串源码替换节点

```
FunctionDeclaration(path) {
path.replaceWithSourceString(`function add(a, b) {
return a + b;
}`);
}
```

```
```diff
- function square(n) {
-   return n * n;
+ function add(a, b) {
+   return a + b;
  }
```

> **注意：</>不建议使用这个API，除非您正在处理动态的源码字符串，否则在访问者外部解析代码更有效率。

插入兄弟节点

```
FunctionDeclaration(path) {
path.insertBefore(t.expressionStatement(t.stringLiteral("Because I'm easy come, easy go.")));
path.insertAfter(t.expressionStatement(t.stringLiteral("A little high, little low.")));
}
```

```
```diff
+ "Because I'm easy come, easy go.";
  function square(n) {
    return n * n;
  }
+ "A little high, little low.";
```

> 注意：</>这里同样应该使用声明或者一个声明数组。 这个使用了在用多个节点替换一个节点</>中提到的相同的启发式算法。.

插入到容器（container）中

如果您想要在AST节点属性中插入一个像 `body </ 0>那样的数组。`
`它与 <code> insertBefore/insertAfter` 类似, 但您必须指定 `listKey` (通常是 `正文`).

```
ClassMethod(path) {
path.get('body').unshiftContainer('body',
t.expressionStatement(t.stringLiteral('before')));
path.get('body').pushContainer('body',
t.expressionStatement(t.stringLiteral('after')));
}
```

```
```diff
 class A {
  constructor() {
+   "before"
    var a = 'middle';
+   "after"
  }
 }
```

删除一个节点

```
FunctionDeclaration(path) {
  path.remove();
}
```

```
- function square(n) {
-   return n * n;
- }
```

替换父节点

只需使用parentPath：` path.parentPath </>调用 `replaceWith </>即可`

```
BinaryExpression(path) {
  path.parentPath.replaceWith(
    t.expressionStatement(t.stringLiteral("Anyway the wind blows, doesn't really matter to me, to me."))
  );
}
`
```

```
  function square(n) {
-   return n * n;
+   "Anyway the wind blows, doesn't really matter to me, to me.";
  }
```

删除父节点

```
BinaryExpression(path) {
  path.parentPath.remove();
}
```
```

```
  function square(n) {
-   return n * n;
  }
```

Scope（作用域）

检查本地变量是否被绑定

```
FunctionDeclaration(path) {
  if (path.scope.hasBinding("n")) {
    // ...
  }
}
```

这将遍历范围树并检查特定的绑定。

您也可以检查一个作用域是否有**自己的</>绑定：

```
FunctionDeclaration(path) {
  if (path.scope.hasOwnBinding("n")) {
    // ...
  }
}
```

创建一个 UID

这将生成一个标识符，不会与任何本地定义的变量相冲突。

```
FunctionDeclaration(path) {
  path.scope.generateUidIdentifier("uid");
  // Node { type: "Identifier", name: "_uid" }
  path.scope.generateUidIdentifier("uid");
  // Node { type: "Identifier", name: "_uid2" }
}
```

提升变量声明至父级作用域

有时你可能想要推送一个` VariableDeclaration </>，这样你就可以分配给它。

```
FunctionDeclaration(path) {
  const id = path.scope.generateUidIdentifierBasedOnNode(path.node.id);
  path.remove();
  path.scope.parent.push({ id, init: path.node });
}
`
```

```
- function square(n) {
+ var _square = function square(n) {
    return n * n;
- }
+ };
```

重命名绑定及其引用

```
FunctionDeclaration(path) {
  path.scope.rename("n", "x");
}
```

```
- function square(n) {
-   return n * n;
+ function square(x) {
+   return x * x;
  }
```

或者，您可以将绑定重命名为生成的唯一标识符：

```
FunctionDeclaration(path) {
  path.scope.rename("n");
}
```

```
- function square(n) {
-   return n * n;
+ function square(_n) {
+   return _n * _n;
  }
```

插件选项

如果您想让您的用户自定义您的Babel插件的行为您可以接受用户可以指定的插件特定选项，如下所示：

```
{
  plugins: [
    ["my-plugin", {
      "option1": true,
      "option2": false
    }]
  ]
}
```

这些选项会通过`状态</>对象传递给插件访问者：

```
export default function({ types: t }) {
  return {
    visitor: {
      FunctionDeclaration(path, state) {
        console.log(state.opts);
        // { option1: true, option2: false }
      }
    }
  }
}
`
```

这些选项是特定于插件的，您不能访问其他插件中的选项。

插件的准备和收尾工作

插件可以具有在插件之前或之后运行的函数。它们可以用于设置或清理/分析目的。

```
export default function({ types: t }) {
  return {
    pre(state) {
      this.cache = new Map();
    },
    visitor: {
      StringLiteral(path) {
        this.cache.set(path.node.value, 1);
      }
    },
    post(state) {
      console.log(this.cache);
    }
  };
}
```

在插件中启用其他语法

插件可以启用babylon plugins</>，以便用户不需要安装/启用它们。 这可以防止解析错误，而不会继承语法插件。

```
export default function({ types: t }) {
  return {
    inherits: require("babel-plugin-syntax-jsx")
  };
}
```

抛出一个语法错误

如果您想用babel-code-frame和一个消息抛出一个错误：

```
export default function({ types: t }) {
  return {
    visitor: {
      StringLiteral(path) {
        throw path.buildCodeFrameError("Error message here");
      }
    }
  };
}
```

该错误看起来像：

```
file.js: Error message here
   7 |
   8 | let tips = [
>  9 |   "Click on any AST node with a '+' to expand it",
     |   ^
  10 |
  11 |   "Hovering over a node highlights the \
  12 |    corresponding part in the source code",
```

构建节点

编写转换时，通常需要构建一些要插入的节点进入AST。 如前所述，您可以使用` babel-types </>包中的 builder </>方法。

构建器的方法名称就是您想要的节点类型的名称，除了第一个字母小写。 例如，如果您想建立一个 MemberExpression 您可以使用 t.memberExpression（...）.

这些构建器的参数由节点定义决定。 有一些正在做的工作，以生成易于阅读的文件定义，但现在他们都可以在此处找到。.

节点定义如下所示：

```
defineType("MemberExpression", {
  builder: ["object", "property", "computed"],
  visitor: ["object", "property"],
  aliases: ["Expression", "LVal"],
  fields: {
    object: {
      validate: assertNodeType("Expression")
    },
    property: {
      validate(node, key, val) {
        let expectedType = node.computed ? "Expression" : "Identifier";
        assertNodeType(expectedType)(node, key, val);
      }
    },
    computed: {
      default: false
    }
  }
});
`
```

在这里你可以看到关于这个特定节点类型的所有信息，包括如何构建它，遍历它，并验证它。

通过查看 生成器 属性，可以看到调用生成器方法所需的3个参数 （t. 情况）.

```
生成器: ["object", "property", "computed"],
```

> 请注意，有时在节点上可以定制的属性比``构建器</>数组包含的属性更多。 这是为了防止生成器有太多的参数。 在这些情况下，您需要手动设置属性。 一个例子是 ClassMethod </>.

```
// Example
// because the builder doesn't contain `async` as a property
var node = t.classMethod(
  "constructor",
  t.identifier("constructor"),
  params,
  body
)
// set it manually after creation
node.async = true;
``
```

> You can see the validation for the builder arguments with the fields object.

```
  fields: {
  object: {
  validate: assertNodeType("Expression")
  },
  property: {
  validate(node, key, val) {
   let expectedType = node.computed ? "Expression" : "Identifier";
   assertNodeType(expectedType)(node, key, val);
  }
  },
  computed: {
  default: false
  }
  }
```

````
You can see that `object` needs to be an `Expression`, `property` either needs to be an `Expression` or an `Identifier` depending on if the member expression is `computed` or not and `computed` is simply a boolean that defaults to `false`.

So we can construct a `MemberExpression` by doing the following:

```js
t.memberExpression(
  t.identifier('object'),
  t.identifier('property')
  // `computed` is optional
);
````

Which will result in:

```
object.property
```

However, we said that `object` needed to be an `Expression` so why is `Identifier` valid?

Well if we look at the definition of `Identifier` we can see that it has an `aliases` property which states that it is also an expression.

```
aliases: ["Expression", "LVal"],
```

So since `MemberExpression` is a type of `Expression`, we could set it as the `object` of another `MemberExpression`:

```
t.memberExpression(
  t.memberExpression(
    t.identifier('member'),
    t.identifier('expression')
  ),
  t.identifier('property')
)
```

Which will result in:

```
member.expression.property
```

It's very unlikely that you will ever memorize the builder method signatures for every node type. So you should take some time and understand how they are generated from the node definitions.
```

You can find all of the actual definitions here and you can see them documented here

---

最佳实践

Create Helper Builders and Checkers

It's pretty simple to extract certain checks (if a node is a certain type) into their own helper functions as well as extracting out helpers for specific node types.

```
function isAssignment(node) {
  return node && node.operator === opts.operator + "=";
}

function buildAssignment(left, right) {
  return t.assignmentExpression("=", left, right);
}
```

尽量避免遍历抽象语法树（AST）

Traversing the AST is expensive, and it's easy to accidentally traverse the AST more than necessary. This could be thousands if not tens of thousands of extra operations.

Babel optimizes this as much as possible, merging visitors together if it can in order to do everything in a single traversal.

及时合并访问者对象

When writing visitors, it may be tempting to call `path.traverse` in multiple places where they are logically necessary.

```
path.traverse({
  Identifier(path) {
    // ...
  }
});

path.traverse({
  BinaryExpression(path) {
    // ...
  }
});
```

However, it is far better to write these as a single visitor that only gets run once. Otherwise you are traversing the same tree multiple times for no reason.

```
path.traverse({
  Identifier(path) {
    // ...
  },
  BinaryExpression(path) {
    // ...
  }
});
```

可以手动查找就不要遍历

It may also be tempting to call `path.traverse` when looking for a particular node type.

```
const nestedVisitor = {
  Identifier(path) {
    // ...
  }
};

const MyVisitor = {
  FunctionDeclaration(path) {
    path.get('params').traverse(nestedVisitor);
  }
};
```

However, if you are looking for something specific and shallow, there is a good chance you can manually lookup the nodes you need without performing a costly traversal.

```
const MyVisitor = {
  FunctionDeclaration(path) {
    path.node.params.forEach(function() {
      // ...
    });
  }
};
```

优化嵌套的访问者对象

当您嵌套访问者（visitor）时，把它们嵌套在您的代码中可能是有意义的。

```
const MyVisitor = {
  FunctionDeclaration(path) {
    path.traverse({
      Identifier(path) {
        // ...
      }
    });
  }
};
```

但是，每当调用`FunctionDeclaration()</>时都会创建一个新的访问者对象。 That can be costly, because Babel does some processing each time a new visitor object is passed in (such as exploding keys containing multiple types, performing validation, and adjusting the object structure). Because Babel stores flags on visitor objects indicating that it's already performed that processing, it's better to store the visitor in a variable and pass the same object each time.

```
const nestedVisitor = {
  Identifier(path) {
    // ...
  }
};
```

```
const MyVisitor = {
  FunctionDeclaration(path) {
    path.traverse(nestedVisitor);
  }
};
`
```

如果您在嵌套的访问者中需要一些状态，像这样：

```
const MyVisitor = {
  FunctionDeclaration(path) {
    var exampleState = path.node.params[0].name;

    path.traverse({
      Identifier(path) {
        if (path.node.name === exampleState) {
          // ...
        }
      }
    });
  }
};
```

您可以将它作为状态传递给`traverse()</ 0>方法，并有权访问<code>this`在访问者中。

```
const nestedVisitor = {
  Identifier(path) {
    if (path.node.name === this.exampleState) {
      // ...
    }
  }
};

const MyVisitor = {
  FunctionDeclaration(path) {
    var exampleState = path.node.params[0].name;
    path.traverse(nestedVisitor, { exampleState });
  }
};
```

留意嵌套结构

有时候在考虑给定的转换时，可能会忘记给定的转换结构可以是嵌套的。

例如，想象一下，我们想要查找`构造函数` `ClassMethod` `Foo` `ClassDeclaration`.

```
class Foo {
  constructor() {
    // ...
  }
}
```

```
const constructorVisitor = {
  ClassMethod(path) {
    if (path.node.name === 'constructor') {
      // ...
    }
  }
}
```

```
const MyVisitor = {
  ClassDeclaration(path) {
    if (path.node.id.name === 'Foo') {
      path.traverse(constructorVisitor);
    }
  }
}
```

我们忽略了类可以嵌套的事实，使用遍历的话，上面我们也会得到一个嵌套的`构造函数</>：

```
class Foo {
  constructor() {
    class Bar {
      constructor() {
        // ...
      }
    }
  }
}
`
```

单元测试

有几种主要的方法来测试babel插件：快照测试，AST测试和执行测试。 对于这个例子，我们将使用 jest </>，因为它支持盒外快照测试。 我们在这里创建的示例是托管在这个 repo</>.

首先我们需要一个babel插件，我们将把它放在src / index.js中。

```
<br />module.exports = function testPlugin(babel) {
  return {
    visitor: {
      Identifier(path) {
        if (path.node.name === 'foo') {
          path.node.name = 'bar';
        }
      }
    }
  };
};
```

快照测试

接下来，用`` npm install --save-dev babel-core jest </>安装我们的依赖关系，
那么我们可以开始写我们的第一个测试：快照。 快照测试允许我们直观地检查我们的babel插件的输出。 我们给它一个输入，告诉它一个快照，并将其保存到一个文件。 我们检查快照到git中。 这允许我们来看看我们什么时候影响了我们任何试用例子测试的输出。 它也给出了使用差异在拉请求的时候。 当然，您可以用任何测试框架来做到这一点，但是要更新一下快照就像 jest -u </>一样简单.

```
 // src/__tests__/index-test.js
const babel = require('babel-core');
const plugin = require('../');

var example = var foo = 1;
if (foo) console.log(foo); ;
```

```
it('works', () => {
  const {code} = babel.transform(example, {plugins: [plugin]});
  expect(code).toMatchSnapshot();
});
```

这给了我们一个快照文件在`` src / __ tests __ / __ snapshots __ / index-test.js.snap </>.

```
exports[`test works 1`] = `
"
var bar = 1;
if (bar) console.log(bar);"
`;
```

如果我们在插件中将“bar”更改为“baz”并再次运行，则可以得到以下结果：

```
接收到的值与存储的快照1不匹配。

    - Snapshot
    + Received

    @@ -1,3 +1,3 @@
     "
    -var bar = 1;
    -if (bar) console.log(bar);"
    +var baz = 1;
    +if (baz) console.log(baz);"
```

我们看到我们对插件代码的改变如何影响了我们插件的输出 如果输出看起来不错，我们可以运行`jest -u </>来更新快照。

AST 测试

除了快照测试外，我们还可以手动检查AST。 这是一个简单但是脆弱的例子。 对于更多涉及的情况，您可能希望利用Babel-遍历。 它允许您用访问者键指定一个对象，就像您使用插件本身。

```
it('contains baz', () => {
  const {ast} = babel.transform(example, {plugins: [plugin]});
  const program = ast.program;
  const declaration = program.body[0].declarations[0];
  assert.equal(declaration.id.name, 'baz');
  // or babelTraverse(program, {visitor: ...})
});
`
```

Exec Tests

在这里，我们将转换代码，然后评估它的行为是否正确。 请注意，我们在测试中没有使用``assert</>。 这确保如果我们的插件做了奇怪的操作，如意外删除断言线，但测试仍然失败。

```
it('foo is an alias to baz', () => {
  var input = `
    var foo = 1;
    // test that foo was renamed to baz
    var res = baz;
  `;
  var {code} = babel.transform(input, {plugins: [plugin]});
  var f = new Function(`
    ${code};
    return res;
  `);
  var res = f();
  assert(res === 1, 'res is 1');
});
```

Babel核心使用类似的方法</>去获取快照和执行测试。

babel-plugin-tester

这个包使测试插件更容易。 如果您熟悉ESLint的 RuleTester您应该对这是熟悉的。 您可以看看the docs去充分了解可能的情况，但这里有一个简单的例子：

```
import pluginTester from 'babel-plugin-tester';
import identifierReversePlugin from '../identifier-reverse-plugin';

pluginTester({
  plugin: identifierReversePlugin,
  fixtures: path.join(__dirname, '__fixtures__'),
  tests: {
    'does not change code with no identifiers': '"hello";',
    'changes this code': {
      code: 'var hello = "hi";',
      output: 'var olleh = "hi";',
    },
    'using fixtures files': {
      fixture: 'changed.js',
      outputFixture: 'changed-output.js',
    },
    'using jest snapshots': {
      code: `
        function sayHi(person) {
          return 'Hello ' + person + '!'
        }
      `,
      snapshot: true,
    },
  },
});
```

> ***对于将来的更新，请跟随 @thejameskyle </>和 @babeljs </> 的Twitter。