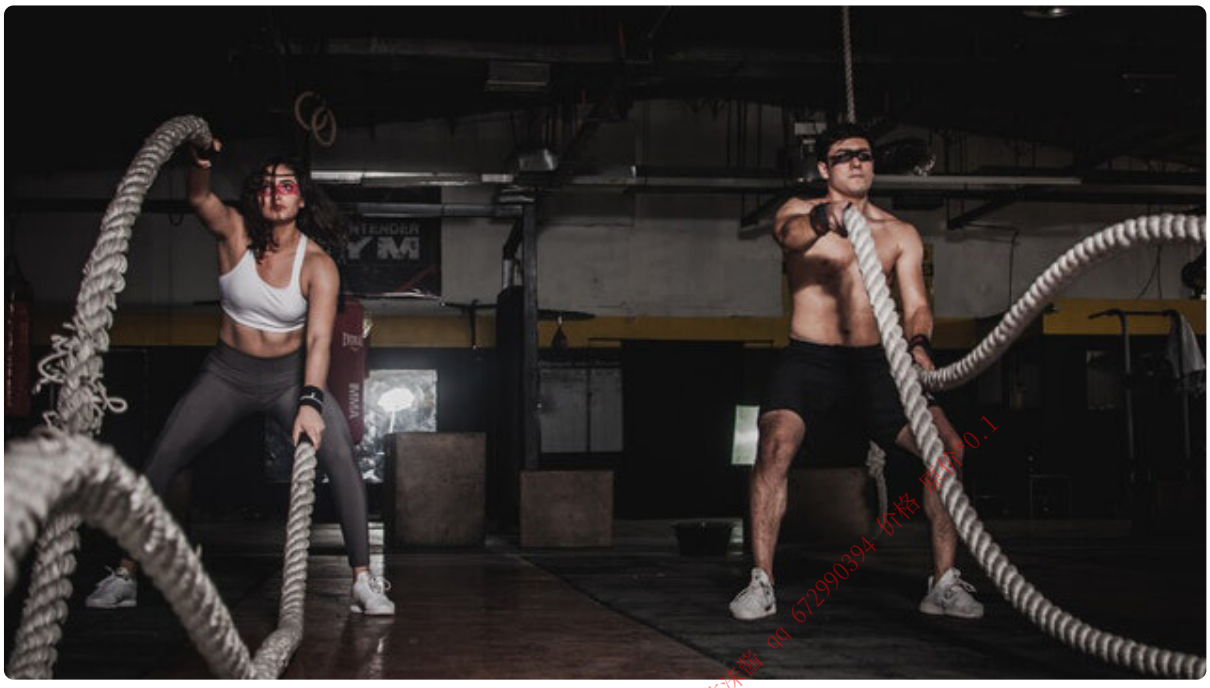
08 在 Webpack 中使用 Babel 转换 JavaScript 代码

更新时间：2019-06-24 09:25:32



> 世界上最快乐的事，莫过于为理想而奋斗。
>
> ——苏格拉底

在 webpack 中编写 JavaScript 代码，可以使用最新的 ES 语法，而最终打包的时候，webpack 会借助 Babel 将 ES6+语法转换成在目标浏览器可执行 ES5 语法。所以 Babel 是一个重要的知识点需要掌握。

什么是 Babel

Babel 是 JavaScript 的编译器，通过 Babel 可以将我们写的最新 ES 语法的代码轻松转换成任意版本的 JavaScript 语法。随着浏览器逐步支持 ES 标准，我们不需要改变代码，只需要修改 Babel 配置即可以适配新的浏览器。

举例说明，下面是 ES6 箭头函数语法的代码：

```
[1, 2, 3].map(n => n ** 2);
```

经过 Babel 处理后，可以转换为普通的 ES5 语法：

```
[1, 2, 3].map(function(n) {
    return Math.pow(n, 2);
});
```

Babel 初体验

下面来介绍下 Babel 的安装和功能及其配置文件。

使用 **babel-cli** 命令行工具

Babel 本身自己带有 CLI（Command-Line Interface，命令行界面） 工具，可以单独安装使用。下面我们在项目中安装 `@babel/cli` 和 `@babel/core`。

```
npm i -D @babel/core @babel/cli
```

然后创建一个`babel.js`文件：

```
// babel.js
[1, 2, 3].map(n => n ** 2);
```

然后执行`npx babel babel.js`，则会看到输出的内容，**此时可能会看到输出的内容跟源文件内容没有区别**，这是因为没有加转换规则，下面安装`@babel/preset-env`。然后执行 CLI 的时候添加 `--presets` flag：

```
# 安装 preset-env
npm i -D @babel/preset-env
# 执行 CLI 添加--presets
npx babel babel.js --presets=@babel/preset-env
```

最终输出的代码就是转换为 ES5 的代码了：

```
'use strict';

[1, 2, 3].map(function(n) {
    return Math.pow(n, 2);
});
```

如果要输出结果到固定文件，可以使用 `--out-file` 或 `-o` 参数：`npx babel babel.js -o output.js`。

> Babel 7 使用了 @babel 命名空间来区分官方包，因此以前的官方包 `babel-xxx` 改成了 `@babel/xxx`。

配置文件

除了使用命令行配置 flag 之外，Babel 还支持配置文件，配置文件支持两种：

- 使用`package.json`的`babel`属性；
- 在项目根目录单独创建`.babelrc`或者`.babelrc.js`文件。

直接上对应的示例：

```
// package.json
{
    "name": "my-package",
    "version": "1.0.0",
    "babel": {
        "presets": ["@babel/preset-env"]
    }
}
// .babelrc
{
    "presets": ["@babel/preset-env"]
}
```

Babel会在正在被转义的文件当前目录中查找一个 `.babelrc` 文件。 如果不存在，它会向**外层目录**遍历目录树，直到找到一个 `.babelrc` 文件，或一个 `package.json` 文件中有 `"babel": {}` 。

env 选项

如果我们希望在不同的环境中使用不同的 Babel 配置，那么可以在配置文件中添加`env`选项：

```
{
  "env": {
    "production": {
      "presets": ["@babel/preset-env"]
    }
  }
}
```

env 选项的值将从 `process.env.BABEL_ENV` 获取，如果没有的话，则获取 `process.env.NODE_ENV` 的值，它也无法获取时会设置为 `"development"`。

Babel 的插件和 Preset

Babel 的语法转换是通过强大的插件系统来支持的。**Babel 的插件分为两类：转换插件和语法解析插件**。

不同的语法对应着不同的转换插件，比如我们要将箭头函数转换为 ES5 函数写法，那么可以单独安装`@babel/plugin-transform-arrow-functions`插件，转换插件主要职责是进行语法转换的，而**解析插件**则是扩展语法的，比如我们要解析`jsx`这类 React 设计的特殊语法，则需要对应的 jsx 插件。

如果不想一个个的添加插件，那么可以使用插件组合 preset（插件预设，插件组合更加好理解一些），最常见的 preset 是`@babel/preset-env`。之前的`preset`是按照`TC39`提案阶段来分的，比如看到`babel-preset-stage-1`代表，这个插件组合里面是支持 `TC39` Stage-1 阶段的转换插件集合。

> Tips：`TC39` 指的是技术委员会（Technical Committee）第 39 号。它是 ECMA 的一部分，ECMA 是「ECMAScript」规范下的 JavaScript 语言标准化的机构。ES6 出来之后，`TC39`精简了提案的修订过程，新流程设计四个 Stage 阶段：
>
> - Stage 0 - 设想（Strawman）：只是一个想法；
> - Stage 1 - 建议（Proposal）：这是值得跟进的；
> - Stage 2 - 草案（Draft）：初始规范，应该提供规范初稿；
> - Stage 3 - 候选（Candidate）：不会有太大的改变，在对外发布之前只是修正一些问题；
> - Stage 4 - 完成（Finished）：当规范的实现至少通过两个验收测试时，进入 Stage 4，会被包含在 ECMAScript 的下一个修订版中。

`@babel/preset-env`是 Babel 官方推出的插件预设，它可以根据开发者的配置按需加载对应的插件，通过`@babel/preset-env`我们可以根据代码执行平台环境和具体浏览器的版本来产出对应的 JavaScript 代码，例如可以设置代码执行在 Node.js 8.9 或者 iOS 12 版本。

Babel polyfill

Babel 只负责进行语法转换，即将 ES6 语法转换成 ES5 语法，但是如果在 ES5 中，有些对象、方法实际在浏览器中可能是不支持的，例如：`Promise`、`Array.prototype.includes`，这时候就需要用`@babel/polyfill`来做模拟处理。`@babel/polyfill`使用方法是先安装依赖，然后在对应的文件内显性的引入：

```
# 安装，注意因为我们代码中引入了 polyfill，所以不再是开发依赖（--save-dev，-D）
npm i @babel/polyfill
```

在文件内直接`import`或者`require`进来：

```
// polyfill
import '@babel/polyfill';
console.log([1, 2, 3].includes(1));
```

Bable runtime

`@babel/polyfill`虽然可以解决模拟浏览器不存在对象方法的事情，但是有以下两个问题：

- 直接修改内置的原型，造成全局污染；
- 无法按需引入，Webpack 打包时，会把所有的 Polyfill 都加载进来，导致产出文件过大。

为了解决这个问题，Babel 社区又提出了`@babel/runtime`的方案，`@babel/runtime`不再修改原型，而是采用替换的方式，比如我们用 Promise，使用`@babel/polyfill`会产生一个`window.Promise`对象，而`@babel/runtime`则会生成一个`_Promise`（示例而已）来替换掉我们代码中用到的`Promise`。另外`@babel/runtime`还支持按需引入。下面以转换`Object.assign`为例，来看下`@babel/runtime`怎么使用。

1. 安装依赖`@babel/runtime`：`npm i @babel/runtime`；
2. 安装`npm i -D @babel/plugin-transform-runtime`作为 Babel 插件；
3. 安装需要转换`Object.assign`的插件：`npm i -D @babel/plugin-transform-object-assign`

编写一个`runtime.js`文件，内容如下：

```
Object.assign({}, {a: 1});
```

执行`npx babel runtime.js --plugins @babel/plugin-transform-runtime,@babel/plugin-transform-object-assign`，最终的输出结果是：

```
import _extends from '@babel/runtime/helpers/extends';

_extends(
    {},
    {
        a: 1
    }
);
```

代码中自动引入了`@babel/runtime/helpers/extends`这个模块（**所以要添加`@babel/runtime`依赖啊**）。

`@babel/runtime`也不是完美的解决方案，由于`@babel/runtime`不修改原型，所以类似`[].includes()`这类使用直接使用原型方法的语法是不能被转换的。

> Tips：'@babel/polyfill'实际是`core-js`和`regenerator-runtime`的合集，所以如果要按需引入'@babel/polyfill'的某个模块，可以直接引入对应的 core-js 模块，但是手动引入的方式还是太费劲。

`@babel/preset-env`

铺垫了这么多，我们继续来讲`@babel/preset-env`，前面介绍了`@babel/preset-env`可以零配置的转化 ES6 代码，我们如果要精细化的使用`@babel/preset-env`，就需要配置对应的选项了，在`@babel/preset-env`的选项中，`useBuiltIns`和`target`是最重要的两个，`useBuiltIns`用来设置浏览器 polyfill，`target`是为了目标浏览器或者对应的环境（browser/node）。

`preset-env`的`useBuiltIns`

前面介绍了`@babel/polyfill`和`@babel/runtime`两种方式来实现浏览器 polyfill，两种方式都比较繁琐，而且不够智能，我们可以使用`@babel/preset-env`的`useBuildIn`选项做 polyfill，这种方式简单而且智能。

`useBuiltIns`默认为 `false`，可以使用的值有 `usage` 和 `entry`：

```
{
  "presets": [
    ["@babel/preset-env", {
      "useBuiltIns": "usage|entry|false"
    }]
  ]
}
```

1. `usage` 表示明确使用到的 Polyfill 引用。在一些 ES2015+ 语法不支持的环境下，每个需要用到 Polyfill 的引用时，会**自动加上**，例如：

```
const p = new Promise();
[1, 2].includes(1);
'foobar'.includes('foo');
```

使用`useBuiltIns='usage'`编译之后，上面代码变成，真正的做到了按需加载，而且类似`[].includes()`这类直接使用原型方法的语法是能被转换的：

```
'use strict';
require('core-js/modules/es.array.includes');
require('core-js/modules/es.object.to-string');
require('core-js/modules/es.promise');
require('core-js/modules/es.string.includes');
var p = new Promise();
[1, 2].includes(1);
'foobar'.includes('foo');
```

2. `entry` 表示替换 `import "@babel/polyfill";`（新版本的 Babel，会提示直接引入 `core-js`或者`regenerator-runtime/runtime`来代替`@babel/polyfill`）的全局声明，然后根据`targets`中浏览器版本的支持，将 polyfill 拆分引入，仅引入有浏览器不支持的 polyfill，所以`entry` 相对`usage`使用起来相对麻烦一些，首先需要手动显性的引入`@babel/polyfill`，而且根据配置`targets`来确定输出，这样会导致代码实际用不到的 polyfill 也会被打包到输出文件，导致文件比较大。

一般情况下，个人建议直接使用`usage`就满足日常开发了。

需要提一下的是，polyfill 用到的`core-js`是可以指定版本的，比如使用 `core-js@3`，则首先安装依赖`npm i -S core-js@3`，然后在 Babel 配置文件`.babelrc`中写上版本。

```
//.babelrc
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage",
                "corejs": 3
            }
        ]
    ]
}
```

`preset-env` 的 `target`

假设希望代码中使用 ES6 的模板字面量\``语法，但是实际执行代码的宿主浏览器是 IE 10 却不支持，那么我们可以使用 target`指定目标浏览器了。

```
{
  "presets": [
    ["@babel/preset-env", {
      "targets": {
        "browsers": "IE 10"
      }
    }]
  ]
}
```

如果我们代码是在 Node.js 环境执行的，则可以指定 Node.js 的版本号：

```
{
  "presets": [
    ["env", {
      "@babel/preset-env": {
        "node": "8.9.3"
      }
    }]
  ]
}
```

1. `targets.browsers` 需要使用 browserslist 的配置方法，但是其设置会被 `targets.[chrome, opera, edge, firefox, safari, ie, ios, android, node, electron]` 覆盖；
2. `targets.node` 设置为 `true` 或 `"current"` 可以根据当前 Node.js 版本进行动态转换。也可以设置为具体的数字表示需要支持的最低 Node.js 版本；
3. `targets.esmodules` 设置使用 ES Modules 语法，最新浏览器支持，这个在后面 Webpack 插件章节会详细介绍如何实现 Modern Mode。

> Tips：browserslist 介绍和配置规则本文最后部分会介绍。

在 Webpack 中使用 Babel

通过上面的内容，我们已经掌握了 Babel 的基本用法，下面在 webpack 中使用 Babel 就变得很简单了，首先安装 npm 依赖，然后修改 `webpack.config.js`。

安装依赖包：

```
# 安装开发依赖
npm i webpack babel-loader webpack-cli @babel/core @babel/preset-env @babel/plugin-transform-runtime -D
# 将 runtime 作为依赖
npm i @babel/runtime -S
```

第二步创建`webpack.config.js`文件，内容如下：

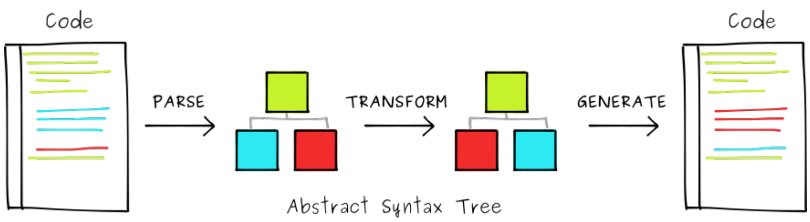
```js
// webpack.config.js
module.exports = {
    entry: './babel.js',
    mode: 'development',
    devtool: false,
    module: {
        rules: [
            {
                test: /\.js$/,
                use: [
                    {
                        loader: 'babel-loader',
                        options: {
                            presets: [
                                [
                                    '@babel/preset-env',
                                    {
                                        useBuiltIns: 'usage'
                                    }
                                ]
                            ]
                        }
                    }
                ]
            }
        ]
    }
};
```

上面的`webpack.config.js`文件直接将 Babel 的配置写到了`options`中，还可以在项目根目录下创建`.babelrc`或者使用`package.json`的 babel 字段。

Babel 原理

了解了 Babel 的使用方法，接下来简单看下 Babel 的原理。Babel 是一个 JavaScript 的静态分析编译器，所谓静态分析指的是在不需要执行代码的前提下对代码进行分析和处理的过程（执行时进行代码分析叫动态分析）。要实现 Babel 从一个语法转换成另外一个语法，需要经过三个主要步骤：解析（Parse），转换（Transform），生成（Generate）。

- 解析：指的是首先将代码经过词法解析和语法解析，最终生成一颗 AST（抽象语法树），在 Babel 中，语法解析器是`Babylon`；
- 转换：得到 AST 之后，可以对其进行遍历，在此过程中对节点进行添加、更新及移除等操作，Babel 中 AST 遍历工具是`@babel/traverse`
- 生成：经过一系列转换之后得到的一颗新树，要将树转换成代码，就是生成的过程，Babel 用到的是`@babel/generator`



> Tips：Babel 的语法解析器 Babylon 目前已经放到 @babel/parser 维护，除了 Babylon， JavaScript 解析器比较著名的还有acorn、Esprima。

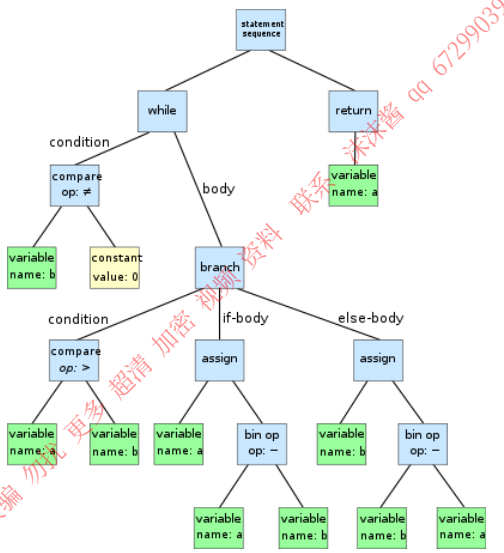
AST

> 在计算机科学中，抽象语法树（Abstract Syntax Tree，AST），或简称语法树（Syntax tree），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。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 `if-condition-then` 这样的条件跳转语句，可以使用带有两个分支的节点来表示。 ——维基百科

Babylon 解析 JavaScript 得到的 AST 是符合 ESTree。AST 是经过**词法解析**和**语法解析**两个步骤解析出来，组织成与程序结构对应的树状结构表示。

例如下面的代码，可以用下图中的树来表示：

```
while (b !== 0) {
    if (a > b) {
        a -= b;
    } else {
        b -= a;
    }
}
```



也可以使用 JavaScript 对象来表示，例如下面代码（放入 `source.js` 文件）：

```
// source.js
function square(n) {
    return n * n;
}
```

对应的 JavaScript 对象为：

```
{
  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"
        }
      }
    }]
  }
}
```

在上面的对象中，AST 的每一层都有相同结构的树分支。

> Tips：AST Explorer 可以在线解析 JavaScript 代码的 AST 结构，还可以在线编写转换函数，学习 AST 的好帮手。

现在开始动手使用 `@babel/parse` 来生成下 AST，将 `square` 函数代码放入 `source.js`，然后新建一个 js 文件（index.js），内容如下：

```
// index.js
const fs = require('fs');
const babel = require('@babel/core');
const traverse = require('@babel/traverse').default;
// 读取 source.js内容
let source = fs.readFileSync('./source.js');

// 使用 babel.parse方法
babel.parse(source, (err, ast) => {
    // ast就是树
    console.log(ast);
});
```

然后执行 `node index.js` 就可以看到对应的 JavaScript 对象表示的 AST 结构了。

在 Babel 中，除了基本的 JavaScript 语法，还支持扩展支持了很多其他的语法，例如 jsx、flow 等。

遍历

如果想处理 AST 那么我们就需要进行树的遍历，学过算法的应该知道树的遍历包括深度优先和广度优先。。。慢点，这里 Babel 提供了 `@babel/traverse`，可以直接来遍历，不需要我们手动来写遍历代码。

按照上一小节的代码，继续修改 `index.js` 文件：

```
const fs = require('fs');
const babel = require('@babel/core');
const traverse = require('@babel/traverse').default;

let source = fs.readFileSync('./source.js');

babel.parse(source, (err, ast) => {
    // console.log(ast)
    let indent = '';
    traverse(ast, {
        // 进入节点
        enter(path) {
            console.log(indent + '<' + path.node.type + '>');
            indent += '  ';
        },
        // 退出节点
        exit(path) {
            indent = indent.slice(0, -2);
            console.log(indent + '<' + '/' + path.node.type + '>');
        }
    });
});
```

执行上面代码，最终我们得到一个类似 Html 结构的树形结构：

```
<Program>
    <FunctionDeclaration>
        <Identifier> </Identifier>
        <Identifier> </Identifier>
        <BlockStatement>
            <ReturnStatement>
                <BinaryExpression>
                    <Identifier> </Identifier>
                    <Identifier> </Identifier>
                </BinaryExpression>
            </ReturnStatement>
        </BlockStatement>
    </FunctionDeclaration>
</Program>
```

遍历的时候，我们**进入（Enter）某个节点时会调用对应的`enter`函数，当退出（Exit）某个节点时，会调用`exit`函数。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。我们还可以针对某个类型的节点进行遍历，如下面代码：


```
const vistors = {
    FunctionDeclaration(path) {
        const param = path.node.params[0];
        paramName = param.name;
        param.name = 'x';
    },
    Identifier: {
        enter() {
            console.log('Entered!');
        },
        exit() {
            console.log('Exited!');
        }
    }
};
traverse(ast, vistors);
```

每次进入和退出函数都会接收一个`path`的参数，`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"
  }
}
```

`path.parent` 为当前节点的父节点信息，`path.node` 则是当前节点的信息。通过操作 `path` 对象就可以对 AST 产生影响（对象引用类型）。

生成

生成是使用了 `@babel/generator`，比较简单，直接看下面例子：

```
const fs = require('fs');
const babel = require('@babel/core');
const traverse = require('@babel/traverse').default;
const gen = require('@babel/generator').default;

let source = fs.readFileSync('./source.js');

babel.parse(source, (err, ast) => {
    // console.log(err, ast)
    let indent = '';
    traverse(ast, {
        // 一顿操作猛如虎。。
    });
    // 生成新的 ast，然后使用generator生成 code
    console.log(gen(ast).code);
});
```

Babel 插件编写

在 Babel 中，代码会由 Babel 先行解析成 AST，Babel 插件做的事情就是写 `vistor` 而已，所以 Babel 插件固定的模板如下：

```
module.exports = () => {
    return {
        name: 'example-plugin',
        visitor: {
            Identifier(path, state) {
                // 一顿操作猛如虎
            }
        }
    };
};
```

> Tips：如果想继续深入了解 Babel 的原理和插件编写相关知识，可以查看《Babel 插件手册》这个 Github 项目。

Browserslist

在本节内容的最后，介绍下在 Browserslist 相关知识和配置。实际开发项目中，我们肯定知道自己的项目运行在什么浏览器内，比如我们做移动开发，不可能需要兼容 IE10 以下的浏览器，所以如果我们做了很多兼容 IE10 以下浏览器的工作，那么就是无用功。通过设置目标浏览器，可以让我们的代码更有针对性的输出兼容性代码（包括 CSS 前缀、JS 的 Polyfill 等），而不是无脑的全部兼容。

Browserslist就是帮助我们来设置目标浏览器的工具。Browserslist 被广泛的应用到 Babel、postcss-preset-env、autoprefixer 等开发工具上。

browserslist 实际上就是声明了一段浏览器的集合，我们的工具可以根据这段集合描述，针对性的输出兼容性代码。

配置

Browserslist 的配置可以放在 `package.json` 中，也可以单独放在配置文件 `.browserslistrc` 中。所有的工具都会主动查找 browserslist 的配置文件，根据 browserslist 配置找出对应的目标浏览器集合。

在 `package.json` 中的配置是增加一个 `browserslist` 数组属性：

```
{
    "browserslist": ["last 2 version", "> 1%", "maintained node versions", "not ie < 11"]
}
```

或者在项目的根目录下创建一个 `.browserslistrc` 文件：

```
# 注释是这样写的，以#号开头
# 每行一个浏览器集合描述
last 2 version
> 1%
maintained node versions
not ie < 11
```


常见集合范围说明

范围	说明
`last 2 versions`	caniuse.com网站跟踪的最新两个版本，假如 iOS 12 是最新版本，那么向后兼容两个版本就是 iOS 11 和 iOS 12
`> 1%`	全球超过 1%人使用的浏览器，类似 `> 5% in US` 则指代美国 5%以上用户
`cover 99.5%`	覆盖 99.5%主流浏览器
`chrome > 50` `ie 6-8`	指定某个浏览器版本范围
`unreleased versions`	说有浏览器的 beta 版本
`not ie < 11`	排除 ie11 以下版本不兼容
`since 2013` `last 2 years`	某时间范围发布的所有浏览器版本
`maintained node versions`	所有被 node 基金会维护的 node 版本
`current node`	当前环境的 node 版本
`dead`	通过 `last 2 versions` 筛选的浏览器中，全球使用率低于 `0.5%` 且官方声明不在维护或者事实上已经两年没有再更新的版本
defaults	默认配置， `> 0.5%` `last 2 versions` `Firefox ESR` `not dead`

浏览器名称列表（大小写不敏感）

- `Android`：安卓 webview 浏览器；
- `Baidu`： 百度浏览器；
- `BlackBerry` / `bb`：黑莓浏览器；
- `Chrome`：chrome 浏览器；
- `ChromeAndroid` / `and_chr`：chrome 安卓移动浏览器；
- `Edge`：微软 Edge 浏览器；
- `Electron` ：Electron；
- `Explorer` / `ie`：ie 浏览器；
- `ExplorerMobile` / `ie_mob`：ie 移动浏览器；
- `Firefox` / `ff`：火狐浏览器； *`FirefoxAndroid` / `and_ff`：火狐安卓浏览器；
- `iOS` / `ios_saf`：iOS Safari 浏览器；
- `Node`：nodejs；
- `Opera`：opera 浏览器；
- `OperaMini` / `op_mini`：operaMini 浏览器；
- `OperaMobile` / `op_mob`：opera 移动浏览器；
- `QQAndroid` / `and_qq`：QQ 安卓浏览器；
- `Samsung`：三星浏览器；
- `Safari`：桌面版本 Safari；
- `UCAndroid` / `and_uc`：UC 安卓浏览器

整个目标浏览器的集合是取并集，即满足上面的全部条件。

Browserslist 的环境变量

我们还可以为不同的环境配置不同的目标浏览器。通过设置`BROWSERSLIST_ENV` 或者 `NODE_ENV`可以配置不同的环境变量。默认情况下会优先从 `production` 对应的配置项加载。在配置文件中，可以通过设置对应的环境目标浏览器：

```
// package.json写法
{
    "browserslist": {
        "production": ["> 1%", "ie 10"],
        "development": ["last 1 chrome version", "last 1 firefox version"]
    }
}
```

或者使用`.browserslistrc`：

```
[production staging]
> 1%
ie 10

[development]
last 1 chrome version
last 1 firefox version
```

> Tips：Browserslist 配置不仅仅 Babel 会用到，其他编译工具也可能会用到，例如后面章节介绍 Webpack CSS 相关配置使用到 postcss 的`autoprefixer`插件也会用到。

Babel Polyfill 的最佳实践

通过上面的介绍，我们可能觉得`useBuiltIns: 'usage'`可以完美的解决我们的 Polyfill 问题，它是按需引入模块，根据`.browserslist`+业务实际代码来设置引入 Polyfill，不会多余的引入。但是在我们构建的时候发现实际还是有问题的：

```
const asyncFun = async () => {
    await new Promise(setTimeout, 2e3);

    return '2s之后才返回该字符串';
};
export default asyncFun;
```

根据上述的 useBuiltIns：'usage' 配置编译后：

```
import 'core-js/modules/es6.promise';
import 'regenerator-runtime/runtime';

function asyncGeneratorStep(gen, resolve, reject, _next, _throw, key, arg) {
    // asyncGeneratorStep
}

function _asyncToGenerator(fn) {
    // _asyncToGenerator 源码
}

var asyncFun =
    /*#__PURE__*/
    (function() {
        var _ref = _asyncToGenerator(
            /*#__PURE__*/
            regeneratorRuntime.mark(function _callee() {
                return regeneratorRuntime.wrap(
                    function _callee$(_context) {
                        while (1) {
                            switch ((_context.prev = _context.next)) {
                                case 0:
                                    _context.next = 2;
                                    return new Promise(setTimeout, 2000);

                                case 2:
                                    return _context.abrupt('return', '2s之后才返回该字符串');

                                case 3:
                                case 'end':
                                    return _context.stop();
                            }
                        }
                    },
                    _callee,
                    this
                );
            })
        );

        return function asyncFun() {
            return _ref.apply(this, arguments);
        };
    })();

export default asyncFun;
```


通过上述的构建之后的代码，我们发现`asyncGeneratorStep`和 `_asyncToGenerator` 这两个函数是被内联进来，而不 是 `import` 进来的。如果这样的话，在多个文件中用到了`async`和`await`关键字，那么每个文件都会编译出一遍`asyncGeneratorStep`和 `_asyncToGenerator`函数。这样的代码明显是重复了，我们再解决了这个问题，Babel polyfill 的方案就完美了，要解决这个问题，需要用到`@babel/plugin-transform-runtime`这个 Babel 插件。

我们知道 Babel 在每个需要转换的代码前面都会插入一些`helpers`代码，这可能会导致多个文件都会有重复的`helpers`代码。`@babel/plugin-transform-runtime`的`helpers`选项就可以把这些代码抽离出来。

所以 Babel 的 Polyfill 的最佳实践是如下的 Babel 配置：

```
// .babelrc
{
    "plugins": [
        [
            "@babel/plugin-transform-runtime",
            {
                "corejs": false, // 默认值，可以不写
                "helpers": true, // 默认，可以不写
                "regenerator": false, // 通过 preset-env 已经使用了全局的 regeneratorRuntime，不再需要 transform-runtime 提供的 不污染全局的 regeneratorRuntime
                "useESModules": true // 使用 es modules helpers，减少 commonJS 语法代码
            }
        ]
    ],
    "presets": [
        [
            "@babel/preset-env",
            {
                "targets": {}, // 这里是targets的配置，根据实际browserslist设置
                "corejs": 3, // 添加core-js版本
                "modules": false, // 模块使用 es modules ，不使用 commonJS 规范
                "useBuiltIns": "usage" // 默认 false，可选 entry , usage
            }
        ]
    ]
}
```

小结

在本小节中，我们不仅仅学习了 Webpack 怎么配置 Babel，还介绍了 Babel 的实现原理，Babel 是先使用Babylon 的解释器，将 JavaScript 语法解析成 AST，然后通过遍历处理这颗树实现代码转换的。在 Babel 中，我们可以通过配置 Browserslist 来针对不同的浏览器组合，生成不同的适配代码。在文章最后，介绍了Babel Polyfill 的最佳实践，这个最佳实践可以直接在我们现有的项目中使用。

> 本小节 Webpack 相关面试题：
>
> 1. Babel 的 preset-env 是什么？
> 2. 懂得 Babel 的原理吗？你会手写 Babel 插件吗？
> 3. Babel 怎么做 Polyfill，Polyfill 的最佳实践是什么？
> 4. Babel 怎么针对不同的浏览器打包不同的适配代码

专栏代码已经整理好给大家共享出来：
https://github.com/ksky521/webpack-tutorial

07 Webpack 中的模块化开发

09 Webpack 中使用 TypeScript 开发项目