目录

[基础 2](#_Toc530264158)

[总结 2](#_Toc530264159)

[脱离webpack 2](#_Toc530264160)

[.babelrc 2](#_Toc530264161)

[presets 3](#_Toc530264162)

[其他 3](#_Toc530264163)

[cli 3](#_Toc530264164)

[babel-node 4](#_Toc530264165)

[浏览器环境 6](#_Toc530264166)

[与其他工具的配合 7](#_Toc530264167)

[包 9](#_Toc530264168)

[babel-register 9](#_Toc530264169)

[babel-core 9](#_Toc530264170)

[babel-polyfill 10](#_Toc530264171)

[babel-runtime 11](#_Toc530264172)

[插件 11](#_Toc530264173)

[babel-plugin-transform-async-to-generator 11](#_Toc530264174)

[babel-preset-\* 12](#_Toc530264175)

[与react结合 12](#_Toc530264176)

[react-babel 12](#_Toc530264177)

[react热替换 12](#_Toc530264178)

[babel-plugin-react-transform 12](#_Toc530264179)

[react-transform-hmr 13](#_Toc530264180)

[babel-preset-react-hmre 13](#_Toc530264181)

[react-transform-catch-errors、redbox-react 13](#_Toc530264182)

[react热替换新 13](#_Toc530264183)

[react-hot-loader 13](#_Toc530264184)

# 基础

## 配置

.babelrc

Babel 会在正在被转录的文件的当前目录中查找一个 .babelrc 文件。 如果不存在，它会遍历目录树，直到找到一个 .babelrc 文件，或一个 package.json 文件中有 "babel": {} 。

在选项中使用 "babelrc": false 来停止查找行为，或者提供--no-babelrc CLI 标志。

不是必须的。

Babel的配置文件是.babelrc，存放在项目的根目录下。使用Babel的第一步，就是配置这个文件。

{

"presets": [],

"plugins": []

}

也可以在webpack中配置

"babel": {

"presets": [

"react-app",

"stage-0"

]

},

**也可以在loader中配置**

loaders: [

//'transform-runtime'插件告诉babel需要运行时间

//而不是内联它。

{

test: /\.js$/,

exclude: /(node\_modules|bower\_components)/,

loader: 'babel-loader',

**query:** { // 2.x以后用options,格式一样

presets: ['es2015'],

plugins: ['transform-runtime']

}

}

### presets

字段设定转码规则，官方提供以下的规则集，你可以根据需要安装。

# ES2015转码规则

$ npm install --save-dev babel-preset-es2015

# react转码规则

$ npm install --save-dev babel-preset-react 这个好用一些

npm install --save-dev babel-preset-react-app 这个使用报错

# ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个

stage向下兼容，即stage0包含stage1等

$ npm install --save-dev babel-preset-stage-0

env

## 其他

### cli

全局

$ npm install --global babel-cli

# 转码结果输出到标准输出

$ babel example.js

# 转码结果写入一个文件

# --out-file 或 -o 参数指定输出文件

$ babel example.js --out-file compiled.js

# 或者

$ babel example.js -o compiled.js

# 整个目录转码

# --out-dir 或 -d 参数指定输出目录

$ babel src --out-dir lib

# 或者

$ babel src -d lib

# -s 参数生成source map文件

$ babel src -d lib -s

项目

将babel-cli安装在项目之中。

# 安装

$ npm install --save-dev babel-cli

然后，改写package.json。

{

// ...

"devDependencies": {

"babel-cli": "^6.0.0"

},

"scripts": {

"build": "babel src -d lib"

},

}

### babel-node

babel-cli工具**自带**一个babel-node命令，提供一个支持ES6的REPL环境。它支持Node的REPL环境的所有功能，而且可以直接运行ES6代码。

它不用单独安装，而是随babel-cli一起安装。然后，执行babel-node就进入PEPL环境。

$ babel-node

> (x => x \* 2)(1)

2

babel-node命令可以直接运行ES6脚本。将上面的代码放入脚本文件es6.js，然后直接运行。

**$ babel-node es6.js**

2

babel-node也可以安装在项目中。

$ npm install --save-dev babel-cli

然后，改写package.json。

{

"scripts": {

"script-name": "babel-node script.js"

}

}

上面代码中，使用babel-node替代node，这样script.js本身就不用做任何转码处理。

### 浏览器环境

Babel也可以用于浏览器环境。但是，从Babel 6.0开始，不再直接提供浏览器版本，而是要用构建工具构建出来。如果你没有或不想使用构建工具，可以通过安装5.x版本的babel-core模块获取。

$ npm install babel-core@5

运行上面的命令以后，就可以在当前目录的node\_modules/babel-core/子目录里面，找到babel的浏览器版本browser.js（未精简）和browser.min.js（已精简）。

然后，将下面的代码插入网页。

<script src="node\_modules/babel-core/browser.js"></script>

<script type="text/babel">

// Your ES6 code

</script>

上面代码中，browser.js是Babel提供的转换器脚本，可以在浏览器运行。用户的ES6脚本放在script标签之中，但是要注明type="text/babel"。

另一种方法是使用[babel-standalone](https://github.com/Daniel15/babel-standalone)模块提供的浏览器版本，将其插入网页。

<script src="<https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.4.4/babel.min.js>"></script>

<script type="text/babel">

// Your ES6 code

</script>

注意，**网页中实时将ES6代码转为ES5，对性能会有影响。**生产环境需要加载已经转码完成的脚本。

下面是如何将代码打包成浏览器可以使用的脚本，以Babel配合Browserify为例。首先，安装babelify模块。

$ npm install --save-dev babelify babel-preset-es2015

然后，再用命令行转换ES6脚本。

$ browserify script.js -o bundle.js \

-t [ babelify --presets [ es2015 react ] ]

上面代码将ES6脚本script.js，转为bundle.js，浏览器直接加载后者就可以了。

在package.json设置下面的代码，就不用每次命令行都输入参数了。

{

"browserify": {

"transform": [["babelify", { "presets": ["es2015"] }]]

}

}

### 与其他工具的配合

许多工具需要Babel进行前置转码，这里举两个例子：ESLint和Mocha。

[ESLint](http://eslint.org/) 用于静态检查代码的语法和风格，安装命令如下。

$ npm install --save-dev eslint babel-eslint

然后，在项目根目录下，新建一个配置文件.eslint，在其中加入parser字段。

{

"parser": "babel-eslint",

"rules": {

...

}

}

再在package.json之中，加入相应的scripts脚本。

{

"name": "my-module",

"scripts": {

"lint": "eslint my-files.js"

},

"devDependencies": {

"babel-eslint": "...",

"eslint": "..."

}

}

[Mocha](http://www.ruanyifeng.com/blog/2015/12/a-mocha-tutorial-of-examples.html) 则是一个测试框架，如果需要执行使用ES6语法的测试脚本，可以修改package.json的scripts.test。

"scripts": {

"test": "mocha --ui qunit --compilers js:babel-core/register"

}

上面命令中，--compilers参数指定脚本的转码器，规定后缀名为js的文件，都需要使用babel-core/register先转码。

## 包

### babel-register

babel-register模块改写require命令，为它加上一个钩子。此后，每当使用require加载.js、.jsx、.es和.es6后缀名的文件，就会先用Babel进行转码。

这个主要用在node端。使用打包工具转码的话不需要再使用babel-register。

$ npm install --save-dev babel-register

使用时，必须首先加载babel-register。

require("babel-register");

require("./index.js");

然后，就不需要手动对index.js转码了。

需要注意的是，babel-register**只会对require命令加载的文件转码，而不会对当前文件转码。另外，由于它是实时转码，所以只适合在开发环境使用。**

**所以不要在require("babel-register")的文件中写es6代码。**

### babel-core

$ npm install babel-core **–save**

var babel = require('babel-core');

**一共四个**

// 字符串转码

babel.**transform**('code();', options);

// => { code, map, ast }(返回的对象属性)

// 文件转码（异步）

babel.**transformFile**('filename.js', options, function(err, result) {

result; // => { code, map, ast }

});

// 文件转码（同步）

babel.**transformFileSync**('filename.js', options);

// => { code, map, ast }

// Babel AST转码

babel.**transformFromAst**(ast, code, options);

// => { code, map, ast }

options <http://babeljs.cn/docs/usage/options/>

例子

var es6Code = 'let x = n => n + 1';

var es5Code = require('babel-core')

.transform(es6Code, {

presets: ['es2015']

})

.code;

// '"use strict";\n\nvar x = function x(n) {\n return n + 1;\n};'

### babel-polyfill

Babel默认只转换新的JavaScript句法（syntax），而不转换新的API，比如**Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise**等全局对象，以及**一些定义在全局对象上的方法**（比如Object.assign）都不会转码。

举例来说，ES6在Array对象上新增了Array.from方法。Babel就不会转码这个方法。如果想让这个方法运行，必须使用babel-polyfill，为当前环境提供一个垫片。

Babel默认不转码的API非常多，详细清单可以查看babel-plugin-transform-runtime模块的[definitions.js](https://github.com/babel/babel/blob/master/packages/babel-plugin-transform-runtime/src/definitions.js)文件。

**使用**

在入口文件引入即可，比如index.js中。也可以在打包过程中引入。

### babel-runtime

babel-runtime 更像是分散的 polyfill 模块，我们可以在自己的模块里单独引入，比如require(‘babel-runtime/core-js/promise’)  
，它们不会在全局环境添加未实现的方法，只是，这样手动引用每个 polyfill 会非常低效。**我们借助babel-plugin-transform-runtime插件来自动化处理这一切。**

* babel-polyfill 比较适合 自己的整个工程，底层也是core-js，能使用所有的新API。
* babel-runtime 适合库和框架，针对使用到的新API 会提出打包到 build 文件中。
* runtime 适合不修改对象自身的属性和方法，比如扩展一个方法， Object.assign, 而 polyfill 是整个打补丁，适合 String.propotype.include 这样的。

那些需要修改内置api才能达成的功能，譬如：扩展String.prototype，给上面增加includes方法，就属于修改内置API的范畴。这类操作就由polyfill提供。

## 插件

### babel-plugin-transform-async-to-generator

将es7的async转成generator

"plugins": [

"transform-async-to-generator",

]

### babel-preset-\*

babel-preset 系列打包了一组插件。如 babel-preset-es2015 打包了 es6 的特性，babel-preset-stage-0 打包处于 strawman 阶段的语法

**es2015,stage-0这些都不交叉，最好都引入。比如import只在es2015模块中**

# 与react结合

## react-babel

babel-preset-react-app

"presets": [

"react-app",

],

## react热替换

**已经过时，可以直接通过webpack实现。**

### babel-plugin-react-transform

替react-hot-loader的插件，是基于Babel Plugin的。这是一个**基本的架子**，要实现热替换还要安装其他插件。

### react-transform-hmr

过时了

安装这个才能实现热替换的功能。

### babel-preset-react-hmre

让Babel知道HMR（热替换），涉及到了webpack.config.js里loader为babel（也就是js、jsx使用babel-loader地方）的配置。

### react-transform-catch-errors、redbox-react

这两个插件把catch到的错误直接显示到页面上，就不用再打开控制台看了 。不过这两个是为了看到错误方便而安装的，实际可以不安~

## react热替换新

### react-hot-loader

webpack现在可以热替换，是通过刷新网页的方式，使用这个以后可以局部更新替换。