ECMAScript 6 入门

作者：阮一峰

目录

其他

ECMAScript 6 简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

---

1. ECMAScript 和 JavaScript 的关系

一个常见的问题是，ECMAScript 和 JavaScript 到底是什么关系？

要讲清楚这个问题，需要回顾历史。1996 年 11 月，JavaScript 的创造者 Netscape 公司，决定将 JavaScript 提交给标准化组织 ECMA，希望这种语言能够成为国际标准。次年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是 1.0 版。

该标准从一开始就是针对 JavaScript 语言制定的，但是之所以不叫 JavaScript，有两个原因。一是商标，Java 是 Sun 公司的商标，根据授权协议，只有 Netscape 公司可以合法地使用 JavaScript 这个名字，且 JavaScript 本身也已经被 Netscape 公司注册为商标。二是想体现这门语言的制定者是 ECMA，不是 Netscape，这样有利于保证这门语言的开放性和中立性。

因此，ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的 ECMAScript 方言还有 JScript 和 ActionScript）。日常场合，这两个词是可以互换的。

---

2. ES6 与 ECMAScript 2015 的关系

ECMAScript 2015（简称 ES2015）这个词，也是经常可以看到的。它与 ES6 是什么关系呢？

2011 年，ECMAScript 5.1 版发布后，就开始制定 6.0 版了。因此，ES6 这个词的原意，就是指 JavaScript 语言的下一个版本。

但是，因为这个版本引入的语法功能太多，而且制定过程当中，还有很多组织和个人不断提交新功能。事情很快就变得清楚了，不可能在一个版本里面包括所有将要引入的功能。常规的做法是先发布 6.0 版，过一段时间再发 6.1 版，然后是 6.2 版、6.3 版等等。

但是，标准的制定者不想这样做。他们想让标准的升级成为常规流程：任何人在任何时候，都可以向标准委员会提交新语法的提案，然后标准委员会每个月开一次会，评估这些提案是否可以接受，需要哪些改进。如果经过多次会议以后，一个提案足够成熟了，就可以正式进入标准了。这就是说，标准的版本升级成为了一个不断滚动的流程，每个月都会有变动。

标准委员会最终决定，标准在每年的 6 月份正式发布一次，作为当年的正式版本。接下来的时间，就在这个版本的基础上做改动，直到下一年的 6 月份，草案就自然变成了新一年的版本。这样一来，就不需要以前的版本号了，只要用年份标记就可以了。

ES6 的第一个版本，就这样在 2015 年 6 月发布了，正式名称就是《ECMAScript 2015 标准》（简称 ES2015）。2016 年 6 月，小幅修订的《ECMAScript 2016 标准》（简称 ES2016）如期发布，这个版本可以看作是 ES6.1 版，因为两者的差异非常小（只新增了数组实例的 `includes` 方法和指数运算符），基本上是同一个标准。根据计划，2017 年 6 月发布 ES2017 标准。

因此，ES6 既是一个历史名词，也是一个泛指，含义是 5.1 版以后的 JavaScript 的下一代标准，涵盖了 ES2015、ES2016、ES2017 等等，而 ES2015 则是正式名称，特指该年发布的正式版本的语言标准。本书中提到 ES6 的地方，一般是指 ES2015 标准，但有时也是泛指“下一代 JavaScript 语言”。

---

3. 语法提案的批准流程

任何人都可以向标准委员会（又称 TC39 委员会）提案，要求修改语言标准。

一种新的语法从提案到变成正式标准，需要经历五个阶段。每一个阶段的变动都需要由 TC39 委员会批准。

- Stage 0 - Strawman（展示阶段）
- Stage 1 - Proposal（征求意见阶段）
- Stage 2 - Draft（草案阶段）
- Stage 3 - Candidate（候选人阶段）
- Stage 4 - Finished（定案阶段）

一个提案只要能进入 Stage 2，就差不多肯定会包括在以后的正式标准里面。ECMAScript 当前的所有提案，可以在 TC39 的官方网站 GitHub.com/tc39/ecma262查看。

本书的写作目标之一，是跟踪 ECMAScript 语言的最新进展，介绍 5.1 版本以后所有的新语法。对于那些明确或很有希望，将要列入标准的新语法，都将予以介绍。

---

4. ECMAScript 的历史

ES6 从开始制定到最后发布，整整用了 15 年。

前面提到，ECMAScript 1.0 是 1997 年发布的，接下来的两年，连续发布了 ECMAScript 2.0（1998 年 6 月）和 ECMAScript 3.0（1999 年 12 月）。3.0 版是一个巨大的成功，在业界得到广泛支持，成为通行标准，奠定了 JavaScript 语言的基本语法，以后的版本完全继承。直到今天，初学者一开始学习 JavaScript，其实就是在学 3.0 版的语法。

2000 年，ECMAScript 4.0 开始酝酿。这个版本最后没有通过，但是它的大部分内容被 ES6 继承了。因此，ES6 制定的起点其实是 2000 年。

为什么 ES4 没有通过呢？因为这个版本太激进了，对 ES3 做了彻底升级，导致标准委员会的一些成员不愿意接受。ECMA 的第 39 号技术专家委员会（Technical Committee 39，简称 TC39）负责制订 ECMAScript 标准，成员包括 Microsoft、Mozilla、Google 等大公司。

2007 年 10 月，ECMAScript 4.0 版草案发布，本来预计次年 8 月发布正式版本。但是，各方对于是否通过这个标准，发生了严重分歧。以 Yahoo、Microsoft、Google 为首的大公司，反对 JavaScript 的大幅升级，主张小幅改动；以 JavaScript 创造者 Brendan Eich 为首的 Mozilla 公司，则坚持当前的草案。

2008 年 7 月，由于对于下一个版本应该包括哪些功能，各方分歧太大，争论过于激烈，ECMA 开会决定，中止 ECMAScript 4.0 的开发，将其中涉及现有功能改善的一小部分，发布为 ECMAScript 3.1，而将其他激进的设想扩大范围，放入以后的版本，由于会议的气氛，该版本的项目代号起名为 Harmony（和谐）。会后不久，ECMAScript 3.1 就改名为 ECMAScript 5。

2009 年 12 月，ECMAScript 5.0 版正式发布。Harmony 项目则一分为二，一些较为可行的设想定名为 JavaScript.next 继续开发，后来演变成 ECMAScript 6；一些不是很成熟的设想，则被视为 JavaScript.next.next，在更远的将来再考虑推出。TC39 委员会的总体考虑是，ES5 与 ES3 基本保持兼容，较大的语法修正和新功能加入，将由 JavaScript.next 完成。当时，JavaScript.next 指的是 ES6，第六版发布以后，就指 ES7。TC39 的判断是，ES5 会在 2013 年的年中成为 JavaScript 开发的主流标准，并在此后五年中一直保持这个位置。

2011 年 6 月，ECMAScript 5.1 版发布，并且成为 ISO 国际标准（ISO/IEC 16262:2011）。

2013 年 3 月，ECMAScript 6 草案冻结，不再添加新功能。新的功能设想将被放到 ECMAScript 7。

2013 年 12 月，ECMAScript 6 草案发布。然后是 12 个月的讨论期，听取各方反馈。

2015 年 6 月，ECMAScript 6 正式通过，成为国际标准。从 2000 年算起，这时已经过去了 15 年。

---

5. 部署进度

各大浏览器的最新版本，对 ES6 的支持可以查看kangax.github.io/compat-table/es6/。随着时间的推移，支持度已经越来越高了，超过 90%的 ES6 语法特性都实现了。

Node 是 JavaScript 的服务器运行环境（runtime）。它对 ES6 的支持度更高。除了那些默认打开的功能，还有一些语法功能已经实现了，但是默认没有打开。使用下面的命令，可以查看 Node 已经实现的 ES6 特性。

```
// Linux & Mac
$ node --v8-options | grep harmony

// Windows
$ node --v8-options | findstr harmony
```

我写了一个工具 ES-Checker，用来检查各种运行环境对 ES6 的支持情况。访问ruanyf.github.io/es-checker，可以看到您的浏览器支持 ES6 的程度。运行下面的命令，可以查看你正在使用的 Node 环境对 ES6 的支持程度。

```
$ npm install -g es-checker
$ es-checker

========================================
Passes 24 feature Detections
Your runtime supports 57% of ECMAScript 6
========================================
```

---

6. Babel 转码器

Babel 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在现有环境执行。这意味着，你可以用 ES6 的方式编写程序，又不用担心现有环境是否支持。下面是一个例子。

```
// 转码前
input.map(item => item + 1);

// 转码后
input.map(function (item) {
  return item + 1;
});
```

上面的原始代码用了箭头函数，Babel 将其转为普通函数，就能在不支持箭头函数的 JavaScript 环境执行了。

下面的命令在项目目录中，安装 Babel。

```
$ npm install --save-dev @babel/core
```

配置文件 .babelrc

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。

该文件用来设置转码规则和插件，基本格式如下。

```
{
  "presets": [],
  "plugins": []
}
```

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。

```
# 最新转码规则
$ npm install --save-dev @babel/preset-env

# react 转码规则
$ npm install --save-dev @babel/preset-react
```

然后，将这些规则加入 .babelrc 。

```
{
  "presets": [
    "@babel/env",
    "@babel/preset-react"
  ],
  "plugins": []
}
```

注意，以下所有 Babel 工具和模块的使用，都必须先写好 .babelrc 。

命令行转码

Babel 提供命令行工具 @babel/cli ，用于命令行转码。

它的安装命令如下。

```
$ npm install --save-dev @babel/cli
```

基本用法如下。

```
# 转码结果输出到标准输出
$ npx 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
$ npx babel example.js --out-file compiled.js
# 或者
$ npx babel example.js -o compiled.js

# 整个目录转码
# --out-dir 或 -d 参数指定输出目录
$ npx babel src --out-dir lib
# 或者
$ npx babel src -d lib

# -s 参数生成source map文件
$ npx babel src -d lib -s
```

---

babel-node

@babel/node 模块的 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。

首先，安装这个模块。

```
$ npm install --save-dev @babel/node
```

然后，执行 `babel-node` 就进入 REPL 环境。

```
$ npx 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。

```
# es6.js 的代码
# console.log((x => x * 2)(1));
$ npx babel-node es6.js
2
```

@babel/register 模块

`@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。

```
$ npm install --save-dev @babel/register
```

使用时，必须首先加载 `@babel/register`。

```
// index.js
require('@babel/register');
require('./es6.js');
```

然后，就不需要手动对 `index.js` 转码了。

```
$ node index.js
2
```

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。

---

babel API

如果某些代码需要调用 Babel 的 API 进行转码，就要使用 `@babel/core` 模块。

```
var babel = require('@babel/core');

// 字符串转码
babel.transform('code();', options);
// => { code, map, ast }

// 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 }
```

配置对象 `options`，可以参看官方文档http://babeljs.io/docs/usage/options/。

下面是一个例子。

```
var es6Code = 'let x = n => n + 1';
var es5Code = require('@babel/core')
  .transform(es6Code, {
    presets: ['@babel/env']
  })
  .code;

console.log(es5Code);
// '"use strict";\n\nvar x = function x(n) {\n  return n + 1;\n};'
```

上面代码中，`transform` 方法的第一个参数是一个字符串，表示需要被转换的 ES6 代码，第二个参数是转换的配置对象。

@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`、`Map`、`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。

安装命令如下。

```
$ npm install --save-dev @babel/polyfill
```

然后，在脚本头部，加入如下一行代码。

```
import '@babel/polyfill';
// 或者
require('@babel/polyfill');
```

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文件。

浏览器环境

Babel 也可以用于浏览器环境，使用@babel/standalone模块提供的浏览器版本，将其插入网页。

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
<script type="text/babel">
// Your ES6 code
</script>
```

注意，网页实时将 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。

Babel 提供一个REPL 在线编译器，可以在线将 ES6 代码转为 ES5 代码。转换后的代码，可以直接作为 ES5 代码插入网页运行。

---

7. Traceur 转码器

Google 公司的Traceur转码器，也可以将 ES6 代码转为 ES5 代码。

直接插入网页

Traceur 允许将 ES6 代码直接插入网页。首先，必须在网页头部加载 Traceur 库文件。

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>
<script type="module">
  import './Greeter.js';
</script>
```

上面代码中，一共有 4 个 `script` 标签。第一个是加载 Traceur 的库文件，第二个和第三个是将这个库文件用于浏览器环境，第四个则是加载用户脚本，这个脚本里面可以使用 ES6 代码。

注意，第四个 `script` 标签的 `type` 属性的值是 `module` ，而不是 `text/javascript` 。这是 Traceur 编译器识别 ES6 代码的标志，编译器会自动将所有 `type=module` 的代码编译为 ES5，然后再交给浏览器执行。

除了引用外部 ES6 脚本，也可以直接在网页中放置 ES6 代码。

```
<script type="module">
  class Calc {
    constructor() {
      console.log('Calc constructor');
    }
    add(a, b) {
      return a + b;
    }
  }

  var c = new Calc();
  console.log(c.add(4,5));
</script>
```

正常情况下，上面代码会在控制台打印出 `9` 。

如果想对 Traceur 的行为有精确控制，可以采用下面参数配置的写法。

```
<script>
  // Create the System object
  window.System = new traceur.runtime.BrowserTraceurLoader();
  // Set some experimental options
  var metadata = {
    traceurOptions: {
      experimental: true,
      properTailCalls: true,
      symbols: true,
      arrayComprehension: true,
      asyncFunctions: true,
```

```
      asyncGenerators: exponentiation,
      forOn: true,
      generatorComprehension: true
    }
  };
  // Load your module
  System.import('./myModule.js', {metadata: metadata}).catch(function(ex) {
    console.error('Import failed', ex.stack || ex);
  });
</script>
```

上面代码中，首先生成 Traceur 的全局对象 `window.System`，然后 `System.import` 方法可以用来加载 ES6。加载的时候，需要传入一个配置对象 `metadata`，该对象的 `traceurOptions` 属性可以配置支持 ES6 功能。如果设为 `experimental: true`，就表示除了 ES6 以外，还支持一些实验性的新功能。

在线转换

Traceur 也提供一个在线编译器，可以在线将 ES6 代码转为 ES5 代码。转换后的代码，可以直接作为 ES5 代码插入网页运行。

上面的例子转为 ES5 代码运行，就是下面这个样子。

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>
<script>
$traceurRuntime.ModuleStore.getAnonymousModule(function() {
  "use strict";

  var Calc = function Calc() {
    console.log('Calc constructor');
  };

  ($traceurRuntime.createClass)(Calc, {add: function(a, b) {
    return a + b;
  }}, {});

  var c = new Calc();
  console.log(c.add(4, 5));
  return {};
});
</script>
```

命令行转换

作为命令行工具使用时，Traceur 是一个 Node 的模块，首先需要用 npm 安装。

```
$ npm install -g traceur
```

安装成功后，就可以在命令行下使用 Traceur 了。

Traceur 直接运行 ES6 脚本文件，会在标准输出显示运行结果，以前面的 `calc.js` 为例。

```
$ traceur calc.js
Calc constructor
9
```

如果要将 ES6 脚本转为 ES5 保存，要采用下面的写法。

```
$ traceur --script calc.es6.js --out calc.es5.js
```

上面代码的 `--script` 选项表示指定输入文件， `--out` 选项表示指定输出文件。

为了防止有些特性编译不成功，最好加上 `--experimental` 选项。

```
$ traceur --script calc.es6.js --out calc.es5.js --experimental
```

命令行下转换生成的文件，就可以直接放到浏览器中运行。

Node 环境的用法

Traceur 的 Node 用法如下（假定已安装 `traceur` 模块）。

```
var traceur = require('traceur');
var fs = require('fs');

// 将 ES6 脚本转为字符串
var contents = fs.readFileSync('es6-file.js').toString();

var result = traceur.compile(contents, {
  filename: 'es6-file.js',
  sourceMap: true,
  // 其他设置
  modules: 'commonjs'
});

if (result.error)
  throw result.error;

// result 对象的 js 属性就是转换后的 ES5 代码
fs.writeFileSync('out.js', result.js);
// sourceMap 属性对应 map 文件
fs.writeFileSync('out.js.map', result.sourceMap);
```

留言