ECMAScript 6 入门

作者：阮一峰

目录

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。

标准的制定者有计划，以后每年发布一次标准，使用年份作为版本。因为ES6的第一个版本是在2015年发布的，所以又称ECMAScript 2015（简称ES2015）。

2016年6月，小幅修订的《ECMAScript 2016 标准》（简称 ES2016）如期发布。由于变动非常小（只新增了数组实例的`includes`方法和指数运算符），因此 ES2016 与 ES2015 基本上是同一个标准，都被看作是 ES6。根据计划，2017年6月将发布 ES2017。

---

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？

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提交给国际标准化组织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. 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。

---

3. 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/es5-compat-table/es6/。随着时间的推移，支持度已经越来越高了，ES6的大部分特性都实现了。

Node.js是JavaScript语言的服务器运行环境，对ES6的支持度比浏览器更高。通过Node，可以体验更多ES6的特性。建议使用版本管理工具nvm，来安装Node，因为可以自由切换版本。不过，`nvm`不支持Windows系统，如果你使用Windows系统，下面的操作可以改用nvmw或nvm-windows代替。

安装nvm需要打开命令行窗口，运行下面的命令。

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/<version nu
```

上面命令的`version number`处，需要用版本号替换。本节写作时的版本号是`v0.29.0`。该命令运行后，`nvm`会默认安装在用户主目录的`.nvm`子目录。

然后，激活`nvm`。

```
$ source ~/.nvm/nvm.sh
```

激活以后，安装Node的最新版。

```
$ nvm install node
```

安装完成后，切换到该版本。

```
$ nvm use node
```

使用下面的命令，可以查看Node所有已经实现的ES6特性。

```
$ node --v8-options | grep harmony

  --harmony_typeof
  --harmony_scoping
  --harmony_modules
  --harmony_symbols
  --harmony_proxies
  --harmony_collections
  --harmony_observation
  --harmony_generators
  --harmony_iteration
  --harmony_numeric_literals
  --harmony_strings
  --harmony_arrays
  --harmony_maths
  --harmony
```

上面命令的输出结果，会因为版本的不同而有所不同。

我写了一个ES-Checker模块，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 Dectations
Your runtime supports 57% of ECMAScript 6
========================================
```

---

4. 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，这个特性还没有得到广泛支持，Babel将其转为普通函数，就能在现有的JavaScript环境执行了。

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。

```
# ES2015转码规则
$ npm install --save-dev babel-preset-es2015

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

# ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个
$ npm install --save-dev babel-preset-stage-0
$ npm install --save-dev babel-preset-stage-1
$ npm install --save-dev babel-preset-stage-2
$ npm install --save-dev babel-preset-stage-3
```

然后，将这些规则加入`.babelrc`。

```
{
  "presets": [
    "es2015",
    "react",
    "stage-2"
  ],
  "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-cli`

Babel提供`babel-cli`工具，用于命令行转码。

它的安装命令如下。

```
$ npm install --global babel-cli
```

基本用法如下。

```
# 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
```

上面代码是在全局环境下，进行Babel转码。这意味着，如果项目要运行，全局环境必须有Babel，也就是说项目产生了对环境的依赖。另一方面，这样做也无法支持不同项目使用不同版本的Babel。

一个解决办法是将`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"
  },
}
```

转码的时候，就执行下面的命令。

```
$ npm run build
```

---

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REPL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。

---

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。

```
$ 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。

---

babel-core

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。

安装命令如下。

```
$ npm install babel-core --save
```

然后，在项目中就可以调用`babel-core`。

```
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: ['es2015']
  })
  .code;
// '"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、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。

安装命令如下。

```
$ npm install --save 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。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.
<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 ] ]
```

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"] }]]
  }
}
```

在线转换

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。

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`.eslintrc`，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。

---

5. 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 src="https://google.github.io/traceur-compiler/bin/BrowserSyste
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js
<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(e
    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模块。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 src="https://google.github.io/traceur-compiler/bin/BrowserSyste
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js
<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.js环境的用法

Traceur的Node.js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);
```

---

6. ECMAScript 7

2013年3月，ES6的草案封闭，不再接受新功能了。新的功能将被加入ES7。

任何人都可以向TC39提案，从提案到变成正式标准，需要经历五个阶段。每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，就差不多等于肯定会包括在ES7里面。

本书的写作目标之一，是跟踪ECMAScript语言的最新进展。对于那些明确的、或者很有希望列入ES7的功能，尤其是那些Babel已经支持的功能，都将予以介绍。

本书介绍的ES7功能清单如下。

Stage 0：

- Function Bind Syntax：函数的绑定运算符
- String.prototype.at：字符串的静态方法at

Stage 1：

- Class and Property Decorators：Class的修饰器
- Class Property Declarations：Class的属性声明
- Additional export-from Statements：export的写法改进
- String.prototype.{trimLeft,trimRight}：字符串删除头尾空格的方法

Stage 2：

- Rest/Spread Properties：对象的Rest参数和扩展运算符

Stage 3

- SIMD API：“单指令，多数据”命令集
- Async Functions：async函数
- Object.values/Object.entries：Object的静态方法values()和entries()
- String padding：字符串长度补全
- Trailing commas in function parameter lists and calls：函数参数的尾逗号
- Object.getOwnPropertyDescriptors：Object的静态方法getOwnPropertyDescriptors

Stage 4：

- Array.prototype.includes：数组实例的includes方法
- Exponentiation Operator：指数运算符

ECMAScript当前的所有提案，可以在TC39的官方网站Github.com/tc39/ecma262查看。

Babel转码器可以通过安装和使用插件来使用各个stage的语法。

留言