ECMAScript 2015 新特性

箭头函数和词法this

箭头函数是使用=>语法对函数定义的简写。它们在语法上类似于 C#，Java 8 和 CoffeeScript 中的相关特性。它们支持表达式（Expression bodies）和函数体（Statement bodies）。与函数不同，箭头函数与其上下文代码共享相同的词法this（愚人码头注：箭头函数并没有自己的this，它的this是派生而来的，根据“词法作用域”派生而来）。如果箭头函数在另一个函数体内，它共享其父函数的 arguments 变量。

ES2015 代码:

1. // 使用表达式（Expression bodies）
2. var odds = evens.map(v => v + 1);
3. var nums = evens.map((v, i) => v + i);
5. // 使用函数体（Statement bodies）
6. nums.forEach(v => {
7. if (v % 5 === 0)
8. fives.push(v);
9. });
11. // 词法`this`
12. var bob = {
13. \_name: "Bob",
14. \_friends: [],
15. printFriends() {
16. this.\_friends.forEach(f =>
17. console.log(this.\_name + " knows " + f));
18. }
19. };
21. // 词法 arguments
22. function square() {
23. let example = () => {
24. let numbers = [];
25. for (let number of arguments) {
26. numbers.push(number \* number);
27. }
29. return numbers;
30. };
32. return example();
33. }
35. square(2, 4, 7.5, 8, 11.5, 21); // returns: [4, 16, 56.25, 64, 132.25, 441]

更多信息:  
– [ES2015 中的箭头函数和词法 this](http://www.css88.com/archives/6959)

类

ES2015的类是一个基于原型的OO模式的简单语法糖。有一个方便的声明形式，使类模式更容易使用，并更具互操作性。类支持基于原型的继承，调用父类的构造函数，生成实例，静态方法和构造函数。

ES2015 代码:

1. class SkinnedMesh extends THREE.Mesh {
2. constructor(geometry, materials) {
3. super(geometry, materials);
5. this.idMatrix = SkinnedMesh.defaultMatrix();
6. this.bones = [];
7. this.boneMatrices = [];
8. //...
9. }
10. update(camera) {
11. //...
12. super.update();
13. }
14. static defaultMatrix() {
15. return new THREE.Matrix4();
16. }
17. }

增强对象字面量

扩展对象字面量支持在构造时设置原型，简化了foo: foo这样的赋值，定义方法和父级调用。这些也使得对象字面量和类声明更接近，并使得基于对象的设计更加方便。

ES2015 代码:

1. var obj = {
2. // 设置原型 "\_\_proto\_\_" 或 '\_\_proto\_\_' 同样可以正常使用
3. \_\_proto\_\_: theProtoObj,
4. // 计算属性名称不设置原型或
5. // 为重复的\_\_proto\_\_属性触发早期错误。
6. ['\_\_proto\_\_']: somethingElse,
7. // ‘handler: handler’的简写
8. handler,
9. // 方法
10. toString() {
11. // 调用父级
12. return "d " + super.toString();
13. },
14. // 计算（动态）属性名
15. [ "prop\_" + (() => 42)() ]: 42
16. };

\_\_proto\_\_属性需要原生支持，并且在以前的ECMAScript版本中已被弃用。现在大多数引擎支持该属性，但[有些仍不支持](https://kangax.github.io/compat-table/es6/#__proto___in_object_literals)。此外，请注意，只需要Web浏览器来实现它，如[附件B](http://www.ecma-international.org/ecma-262/6.0/index.html#sec-object.prototype.__proto__)所示。该属性在Node中可用。

模板字符串

模板字符串提供了用于构造字符串的语法糖。这类似于Perl，Python等中的字符串插值功能。你也可以构建一个自定义标签，避免注入攻击或用字符串内容构建更高层次的数据结构。

ES2015 代码:

1. // 基本字符串创建
2. `This is a pretty little template string.`
4. // 多行字符串
5. `In ES5 this is
6. not legal.`
8. // 插值变量绑定
9. var name = "Bob", time = "today";
10. `Hello ${name}, how are you ${time}?`
12. // 未转义的模板字符串
13. String.raw`In ES5 "\n" is a line-feed.`
15. // 创建一个HTTP请求头的模板字符串，通过内容替换来构建请求
16. GET`http://foo.org/bar?a=${a}&b=${b}
17. Content-Type: application/json
18. X-Credentials: ${credentials}
19. { "foo": ${foo},
20. "bar": ${bar}}`(myOnReadyStateChangeHandler);

解构

解构允许使用模式匹配进行绑定，它支持匹配数组和对象。解构是故障弱化的，类似于标准对象以foo[‘foo’]方式查找属性，当没有找到时返回undefined。

ES2015 代码:

1. // 列表匹配
2. var [a, ,b] = [1,2,3];
3. a === 1;
4. b === 3;
6. // 对象匹配
7. var { op: a, lhs: { op: b }, rhs: c }
8. = getASTNode()
10. // 对象匹配的简写
11. // 绑定当前作用域的 `op`, `lhs` 和 `rhs`
12. var {op, lhs, rhs} = getASTNode()
14. // 可以用于参数
15. function g({name: x}) {
16. console.log(x);
17. }
18. g({name: 5})
20. // 弱化解构
21. var [a] = [];
22. a === undefined;
24. // 带默认值的弱化解构
25. var [a = 1] = [];
26. a === 1;
28. // 非结构化 + 默认参数
29. function r({x, y, w = 10, h = 10}) {
30. return x + y + w + h;
31. }
32. r({x:1, y:2}) === 23

默认参数（Default） + 不定参数（Rest） + 扩展运算符（Spread）

调用具有默认参数的函数，将数组转换为连续的函数参数，将连续的函数参数转换为数组。Rest让我们不再需要`arguments，并且更直接地解决了一些常见的问题。

ES2015 代码:

1. function f(x, y=12) {
2. // 如果不传递（或传递 undefined ）y 为 12
3. return x + y;
4. }
5. f(3) == 15

ES2015 代码:

1. function f(x, ...y) {
2. // y 是一个 Array（数组）；愚人码头注：...运算符将x之后的参数转换为名为y的数组
3. return x \* y.length;
4. }
5. f(3, "hello", true) == 6

ES2015 代码:

1. function f(x, y, z) {
2. return x + y + z;
3. }
4. // 数组中的每个元素作为参数传递；愚人码头注：...运算符将数组转换为连续的参数
5. f(...[1,2,3]) == 6

Let（局部变量） + Const（常量）

新增块级作用域。let是新的var。const是单赋值（仅允许被赋值一次）。静态限制（Static restrictions ）阻止变量在赋值前被使用。

ES2015 代码:

1. function f() {
2. {
3. let x;
4. {
5. // 可以执行，因为是在块级作用域内
6. const x = "sneaky";
7. // 错误，常量不允许被赋值
8. x = "foo";
9. }
10. // 可以执行，因为是使用let声明
11. x = "bar";
12. // 错误，在当前块级作用域内已经被声明了一次
13. let x = "inner";
14. }
15. }

迭代器（Iterators） + For..Of

Iterator对象像 CLR IEnumerable 或者 Java Iterable 一样启用自定义迭代。将 for..in 转换为自定义的基于迭代器的形如 for..of 的迭代，不需要实现一个数组，实现像LINQ这样的惰性设计模式。

ES2015 代码:

1. let fibonacci = {
2. [Symbol.iterator]() {
3. let pre = 0, cur = 1;
4. return {
5. next() {
6. [pre, cur] = [cur, pre + cur];
7. return { done: false, value: cur }
8. }
9. }
10. }
11. }
13. for (var n of fibonacci) {
14. // 循环将在n > 1000 时结束
15. if (n > 1000)
16. break;
17. console.log(n);
18. }

迭代器基于如下的鸭子类型的接口（使用[TypeScript](http://typescriptlang.org/)类型的语法来解析）：

TypeScript 代码:

1. interface IteratorResult {
2. done: boolean;
3. value: any;
4. }
5. interface Iterator {
6. next(): IteratorResult;
7. }
8. interface Iterable {
9. [Symbol.iterator](): Iterator
10. }

通过polyfill支持

要使用迭代器，你需要在项目中包含Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)。

生成器（Generators）

生成器使用function\*和yield简化迭代器的编写。一个通过function\*声明的函数会返回一个生成器实例。生成器是迭代器的子类型，包括额外的next和throw方法。这些使得值能够流回生成器，所以yield是一个返回值（或throw）的表达式。

注意：也可以用于使用‘await’这样的异步编程，另见 ES7 await [提案](https://github.com/lukehoban/ecmascript-asyncawait)。

ES2015 代码:

1. var fibonacci = {
2. [Symbol.iterator]: function\*() {
3. var pre = 0, cur = 1;
4. for (;;) {
5. var temp = pre;
6. pre = cur;
7. cur += temp;
8. yield cur;
9. }
10. }
11. }
13. for (var n of fibonacci) {
14. // 循环将在n > 1000 时结束
15. if (n > 1000)
16. break;
17. console.log(n);
18. }

生成器接口（仅使用[TypeScript](http://typescriptlang.org/)类型语法阐述）：

TypeScript 代码:

1. interface Generator extends Iterator {
2. next(value?: any): IteratorResult;
3. throw(exception: any);
4. }

通过polyfill支持

要使用生成器，你需要在项目中包含Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)。

Comprehensions

在 Babel 6.0 中已经移除

Unicode 编码

新增了一系列的扩展来支持完整的unicode编码，其中包括字符串中新的unicode语法格式，正则表达式的u模式来处理代码点，新的API也让字符串可以处理21位的代码点（code points）。这些新特性允许我们使用JavaScript构建国际化的应用。

ES2015 代码:

1. // 和ES5.1相同
2. "".length == 2
4. // 正则表达式新的u模式
5. "".match(/./u)[0].length == 2
7. // 新形式
8. "\u{20BB7}" == "" == "\uD842\uDFB7"
10. // 新的字符串方法
11. "".codePointAt(0) == 0x20BB7
13. // for of迭代代码点
14. for(var c of "") {
15. console.log(c);
16. }

模块（Modules）

为了定义组件，从语言层面对模块进行了支持。编写方式借鉴了流行的JavaScript模块加载器（AMD, CommonJS）。由主机定义的默认加载器定义运行时的行为。使用隐式异步模式——在模块可以被获取和加载前不会有代码执行。

ES2015 代码:

1. // lib/math.js
2. export function sum(x, y) {
3. return x + y;
4. }
5. export var pi = 3.141593;

ES2015 代码:

1. // app.js
2. import \* as math from "lib/math";
3. console.log("2π = " + math.sum(math.pi, math.pi));

ES2015 代码:

1. // otherApp.js
2. import {sum, pi} from "lib/math";
3. console.log("2π = " + sum(pi, pi));

以及一些额外的功能包括 export default 和 export \*:

ES2015 代码:

1. // lib/mathplusplus.js
2. export \* from "lib/math";
3. export var e = 2.71828182846;
4. export default function(x) {
5. return Math.exp(x);
6. }

ES2015 代码:

1. // app.js
2. import exp, {pi, e} from "lib/mathplusplus";
3. console.log("e^π = " + exp(pi));

模块格式

Babel可以将ES2015的模块转换为一下几种格式：Common.js，AMD，System，以及UMD。你甚至可以创建你自己的方式。详见[模块文档](http://babeljs.io/docs/usage/modules/)。

模块加载器（Module Loaders）

不是ES2015的一部分

这部分留给ECMAScript 2015规范中的实现定义（implementation-defined）。最终的标准将在WHATWG的[Loader（加载器）规范](https://whatwg.github.io/loader/)中确定，但是目前正在修订中。下面的内容来自于之前的ES2015草稿。

模块加载器支持以下功能：

* 动态加载（Dynamic loading）
* 状态一致性（State isolation）
* 全局空间一致性（Global namespace isolation）
* 编译钩子（Compilation hooks）
* 嵌套虚拟化（Nested virtualization）

默认的加载器可以进行配置，新的加载器能构建评估并在独立或受限的上下文中加载代码。

ES2015 代码:

1. // 动态加载 – ‘System’ 是默认的加载器
2. System.import("lib/math").then(function(m) {
3. alert("2π = " + m.sum(m.pi, m.pi));
4. });
6. // 创建执行沙箱 – new Loaders
7. var loader = new Loader({
8. global: fixup(window) // replace ‘console.log’
9. });
10. loader.eval("console.log(\"hello world!\");");
12. // 直接操作模块的缓存
13. System.get("jquery");
14. System.set("jquery", Module({$: $})); // WARNING: not yet finalized

需要额外的polyfill

由于Babel默认使用common.js的模块，你需要一个polyfill来使用加载器API。[这里获取](https://github.com/ModuleLoader/es6-module-loader)。

使用模块加载器

为了使用此功能，你需要告诉Babel使用system模块格式化工具。在此查看[System.js](https://github.com/systemjs/systemjs)。

Map + Set + WeakMap + WeakSet

为常见算法的实现提供了更有效的数据结构。WeakMaps提供了对对象的弱引用（不会被垃圾回收计数）。

ES2015 代码:

1. // Sets
2. var s = new Set();
3. s.add("hello").add("goodbye").add("hello");
4. s.size === 2;
5. s.has("hello") === true;
7. // Maps
8. var m = new Map();
9. m.set("hello", 42);
10. m.set(s, 34);
11. m.get(s) == 34;
13. // Weak Maps
14. var wm = new WeakMap();
15. wm.set(s, { extra: 42 });
16. wm.size === undefined
18. // Weak Sets
19. var ws = new WeakSet();
20. ws.add({ data: 42 });
21. // 由于传入的对象没有其他引用，故将不会被set保存。

需要polyfill支持

为了在所有环境下使用Maps，Sets，WeakMaps和WeakSets，你需要在项目中包含Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)。

代理对象 (Proxies)

代理对象可以创建一个具有目标对象全部行为的对象。可用于拦截，对象的虚拟化，记录/分析等。

ES2015 代码:

1. // 代理普通对象
2. var target = {};
3. var handler = {
4. get: function (receiver, name) {
5. return `Hello, ${name}!`;
6. }
7. };
9. var p = new Proxy(target, handler);
10. p.world === "Hello, world!";

ES2015 代码:

1. // 代理函数对象
2. var target = function () { return "I am the target"; };
3. var handler = {
4. apply: function (receiver, ...args) {
5. return "I am the proxy";
6. }
7. };
9. var p = new Proxy(target, handler);
10. p() === "I am the proxy";

下面是所有运行级别元操作（meta-operations）中可能出现的traps：

ES2015 代码:

1. var handler =
2. {
3. // target.prop
4. get: ...,
5. // target.prop = value
6. set: ...,
7. // 'prop' in target
8. has: ...,
9. // delete target.prop
10. deleteProperty: ...,
11. // target(...args)
12. apply: ...,
13. // new target(...args)
14. construct: ...,
15. // Object.getOwnPropertyDescriptor(target, 'prop')
16. getOwnPropertyDescriptor: ...,
17. // Object.defineProperty(target, 'prop', descriptor)
18. defineProperty: ...,
19. // Object.getPrototypeOf(target), Reflect.getPrototypeOf(target),
20. // target.\_\_proto\_\_, object.isPrototypeOf(target), object instanceof target
21. getPrototypeOf: ...,
22. // Object.setPrototypeOf(target), Reflect.setPrototypeOf(target)
23. setPrototypeOf: ...,
24. // for (let i in target) {}
25. enumerate: ...,
26. // Object.keys(target)
27. ownKeys: ...,
28. // Object.preventExtensions(target)
29. preventExtensions: ...,
30. // Object.isExtensible(target)
31. isExtensible :...
32. }

不支持的特性

由于ES5的局限性，Proxies无法被转换或者通过polyfill兼容，查看[不同JavaScript引擎](https://kangax.github.io/compat-table/es6/#Proxy)对该功能的支持。

Symbols

Symbol对对象的状态进行访问控制。Symbol允许对象的属性可以通过string（ES5）或者symbol键入。symbol是一种基本数据类型。可选的 name 参数用于调试——但并不是他本身的一部分。Symbol是唯一的，但不是私有的，因为他们通过诸如 Object.getOwnPropertySymbols 这样的反射特性来暴露。

ES2015 代码:

1. (function() {
3. // 模块内的symbol
4. var key = Symbol("key");
6. function MyClass(privateData) {
7. this[key] = privateData;
8. }
10. MyClass.prototype = {
11. doStuff: function() {
12. ... this[key] ...
13. }
14. };
16. // 被Babel部分支持，原生环境可以完全实现。
17. typeof key === "symbol"
18. })();
20. var c = new MyClass("hello")
21. c["key"] === undefined

通过polyfill部分实现

通过Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)部分实现。由于语言的限制，部分功能不能转换或通过polyfill兼容。您可以查看code.js的[注意事项](https://github.com/zloirock/core-js#caveats-when-using-symbol-polyfill)获取更多信息。

子类内置

在ES2015中，像Array，Date和DOM Element这样的内置函数可以被子类化。

ES2015 代码:

1. // Array 子类的用户代码
2. class MyArray extends Array {
3. constructor(...args) { super(...args); }
4. }
6. var arr = new MyArray();
7. arr[1] = 12;
8. arr.length == 2

部分支持

内置子类别应该根据具体情况进行评估，因为诸如HTMLElement类可以被子类化。由于ES5引擎的限制，诸如Date，Array和Error等对象不能被子类化。

Math + Number + String + Object APIs

新增很多功能，如核心的Math库，数组转换和用于对象复制的Object.assign。

ES2015 代码:

1. Number.EPSILON
2. Number.isInteger(Infinity) // false
3. Number.isNaN("NaN") // false
5. Math.acosh(3) // 1.762747174039086
6. Math.hypot(3, 4) // 5
7. Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2
9. "abcde".includes("cd") // true
10. "abc".repeat(3) // "abcabcabc"
12. Array.from(document.querySelectorAll("\*")) // 将类似数组的对象转换为真正的数组
13. Array.of(1, 2, 3) // 类似与new Array(...)，但是当仅有一个参数时，两者表现不同
14. [0, 0, 0].fill(7, 1) // [0,7,7]
15. [1,2,3].findIndex(x => x == 2) // 1
16. ["a", "b", "c"].entries() // 迭代结果 [0, "a"], [1,"b"], [2,"c"]
17. ["a", "b", "c"].keys() // 迭代结果 0, 1, 2
18. ["a", "b", "c"].values() // 迭代结果 "a", "b", "c"
20. Object.assign(Point, { origin: new Point(0,0) })

通过polyfill有限的支持

上述许多API都通过[polyfill](http://babeljs.io/docs/usage/polyfill/)进行了支持，但是部分特性由于多种原因没有被实现（如，String.prototype.normalize需要编写大量额外的代码来实现），你可以在[这里](https://github.com/addyosmani/es6-tools#polyfills)找到更多的polyfill。

二进制和八进制字面量

新增两种数字字面量：二进制(b)和八进制(o)。

ES2015 代码:

1. 0b111110111 === 503 // true
2. 0o767 === 503 // true

仅支持字面模式

Babel仅可以转换0o767，并不能转换Number("0o767")。

Promises

Promises是一种异步编程的方式。Promises在将来可能会得到支持。目前很多的JavaScript库都使用了Promises。

ES2015 代码:

1. function timeout(duration = 0) {
2. return new Promise((resolve, reject) => {
3. setTimeout(resolve, duration);
4. })
5. }
7. var p = timeout(1000).then(() => {
8. return timeout(2000);
9. }).then(() => {
10. throw new Error("hmm");
11. }).catch(err => {
12. return Promise.all([timeout(100), timeout(200)]);
13. })

通过polyfill支持

要使用Promises，你需要引入Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)。

Reflect API

完整的Reflect API使得可以在运行级别对对象进行元操作。它相当与是Proxy API的逆，并允许调用对应的元操作，如proxy traps。这使得它在实现Proxy时非常有用。

ES2015 代码:

1. var O = {a: 1};
2. Object.defineProperty(O, 'b', {value: 2});
3. O[Symbol('c')] = 3;
5. Reflect.ownKeys(O); // ['a', 'b', Symbol(c)]
7. function C(a, b){
8. this.c = a + b;
9. }
10. var instance = Reflect.construct(C, [20, 22]);
11. instance.c; // 42

通过polyfill支持

要使用Reflect API，你需要引入Babel的[polyfill](http://babeljs.io/docs/usage/polyfill/)。

Tail Calls

现在递归调用函数不用担心栈无限增长，使得递归算法在面对无限的输入时更加安全。

ES2015 代码:

1. function factorial(n, acc = 1) {
2. "use strict";
3. if (n < = 1) return acc;
4. return factorial(n - 1, n \* acc);
5. }
7. // 如今运行这段代码会导致栈溢出
8. // 但是在ES2015中，即便输入很随意也可以安全运行
9. factorial(100000)

在 Babel 6 中暂时删除

由于支持全局尾调用的复杂性和性能影响，仅支持显式自引用尾递归。删除由于其他bug，将重新实现。