

ES 2015

前端 蔡超

概況

★ 2015年6月18日定稿！

★ ES6

Standard ECMA-262 *ECMAScript® 2015 Language Specification*

★ Node.js

6th edition (June 2015)

This Standard defines the ECMAScript 2015 general purpose programming language.

★ 知名项目

The following files can be freely downloaded:

File name	Size (Bytes)	Content
ECMA-262.pdf	12 502 495	Acrobat (r) PDF file
ECMA-262 edition 6		Browsable HTML

★ ES6

This 6th edition will be also approved as fourth edition of the international standard ISO/IEC 16262:2016.

The previous replaced "historical" editions of this Ecma Standard are available [here](#).

Arrows function

default + rest + spread

Classes

let + const

String

Promises

async await

Destructuring

binary and octal literals

Map + Set + Weakmap + Weakset

Proxies

Unicode

Modules + Module loader

Iterators

Generators

Enhanced Object Literals

Symbols reflect api

Arrows function

default + rest + spread

let + const

Classes

Map + Set + Weakmap + Weakset

Destructuring

String

Modules + Module loader

Iterators

Symbols

Enhanced Object Literals

let + const

let + const

- JS的问题：变量声明提升 & 函数声明提升

```
var n = 1;

(function () {
    console.log(n);
    var n = 2;
    console.log(n);
})();

console.log(n);

// undefined
// 2
// 1
```

let + const

- let 是可以认为是var的一个升级版，它把变量锁死在自己的作用域里

```
// ES5

var a = 1;

if (1 === a) {
  var b = 2;
}

console.log(a); // 1
console.log(b); // 2

// window
console.log(window.a); // 1
console.log(window.b); // 2
```

```
// ES6 – let

let a = 1;

if (1 === a) {
  let b = 2;
}

console.log(a); // 1
console.log(b); // ReferenceError: b
is not defined

// window
console.log(window.a); // 1
console.log(window.b); // undefined
```

let + const

- **const**与**let**很像，属于自己作用域内的常量，不可以更改的值，使用**const**来做声明

```
// ES6 - let

const PI = 3.1415926; // 全局的PI常量
{
  const PIPI = 3.14; // 本作用域的PI常量
}
console.log(PI); // 3.1415926
PI=3;
console.log(PI); // 3.1415926
console.log(PIPI); // 3.14
```

default + rest + spread

default + rest + spread

这三项放在一起，是ES6针对函数参数部分的优化改进

default + rest + spread

- **default**

为形参提供默认值， 在没有参数传入时使用默认值 其他语言很早就有的设定

```
// ES5
function inc(number, increment) {
  // 如果没有increment参数传入， 我们默认为1
  increment = increment || 1;
  return number + increment;
}
console.log(inc(2, 2)); // 4
console.log(inc(2));   // 3
```

```
// ES6, 使用了默认值
function inc(number, increment = 1) {
  return number + increment;
}
```

default + rest + spread

- default

甚至你还可以放在参数中间启用默认值

```
function sum(a, b = 2, c) {  
  return a + b + c;  
}  
  
console.log(sum(1, 5, 10));          // 16 -> b === 5  
console.log(sum(1, undefined, 10)); // 13 -> b as default
```

default + rest + spread

- **default**

默认值除了写常量， 变量， 还可以接收一个函数返回值

```
function getDefaultIncrement() {  
    return 1;  
}  
  
function inc(number, increment = getDefaultIncrement()) {  
    return number + increment;  
}  
  
console.log(inc(2, 2)); // 4  
console.log(inc(2));   // 3
```

default + rest + spread

- rest

下面让我们写个将所有参数累加的方法，如果使用ES5，需要调用 **arguments**，可能会这么写：

```
// ES5
function sum() {
  var numbers = Array.prototype.slice.call(arguments),
      result = 0;
  numbers.forEach(function (number) {
    result += number;
  });
  return result;
}

console.log(sum(1));          // 1
console.log(sum(1, 2, 3, 4, 5)); // 15
```

default + rest + spread

- rest

ES6提供了一个叫rest的机制，我们可以理解为余下的所有（参数），上面累加的函数可以改写为

```
// ES5
function sum(...numbers) {
  var result = 0;
  numbers.forEach(function (number) {
    result += number;
  });
  return result;
}

console.log(sum(1)); // 1
console.log(sum(1, 2, 3, 4, 5)); // 15
```

default + rest + spread

- rest

ES6提供了一个叫rest的机制，我们可以理解为余下的所有（参

数），上面累加的函数可以改写为

注意：一旦使用了rest，形参位置就只能使用rest写法，不能再拆

分参数了，如下就是错误的：

```
function sum(...numbers, last) // 语法错误
```

default + rest + spread

- spread

spread是与上面rest很相关的一个特性，因为都使用了3个点....，

rest是行参定义时， spread是实参传入时

```
// ES6
function sum(a, b, c) {
  return a + b + c;
}
var args = [1, 2, 3];

// ES6
console.log(sum(...args)); // 6

// ES5我们要这么写
console.log(sum.apply(undefined, args)); // 6
```

default + rest + spread

- spread

spread是与上面rest很相关的一个特性，因为都使用了3个点....，

rest是行参定义时， spread是实参传入时

spread没有rest的行参数量限制问题，可以后面拆分实际参数传

入，如下：

```
function sum(a, b, c) {  
    return a + b + c;  
}  
var args = [1, 2];  
  
console.log(sum(...args, 3)); // 6
```

String

String

- 字符串插入（模板）

```
let x = 1;
let y = 2;
let sumTpl = `${x} + ${y} = ${x + y}`;
console.log(sumTpl); // 1 + 2 = 3
```

如上，可以使用 \${变量名} 的语法直接插入变量，有了它，我们可以扔掉JS模版系统了。

String

- 多行字符串

```
let types = `Number  
String  
Array  
Object`;  
console.log(types); // Number  
                  // String  
                  // Array  
                  // Object
```

```
var types = “Number\nString\nArray\nObject”;  
  
console.log(types); // Number  
                  // String  
                  // Array  
                  // Object
```

String

- 原生字符串 **String.raw**

```
// 输出多行文本
let interpreted = 'raw\nstring';

// 包含了`\\n`这个转义字符
```

```
// 使用了 String.raw 方法，把 \n 直接输出，没有被转义
let raw = String.raw`raw\nstring`;

console.log(interpreted);    // raw
                           // string
console.log(raw === esaped); // true
```

String

- 支持枚举

```
let str = 'abc';
console.log(str.length); // 3
for (let c of str) {
  console.log(c); // a
    // b
    // c
}
```

String

- 还可以用我们之前讲到的 **spread** 把字符串很方便的转为数组

```
let str = 'abcd';
let chars = [...str];
console.log(chars); // ['a', 'b', 'c', 'd']
```

String

- 新增的字符串方法

```
// repeat(n) -使字符串重复生成n次
console.log('abc'.repeat(3)); // 'abcabcabc'

// startsWith(str, starts = 0)-是否从某位置开始某个字串? 布尔返回值
console.log('ecmascript'.startsWith('ecma'));      // true
console.log('ecmascript'.startsWith('script', 4)); // true

// endsWith(str, ends = str.length) -是否从某位置以某字符串结束? 布尔返回值
console.log('ecmascript'.endsWith('script')); // true
console.log('ecmascript'.endsWith('ecma', 4)); // true

// includes(str, starts = 0) : boolean - 从某个位置开始是否包含某个字符串
console.log('ecmascript'.includes('ecma'));      // true
console.log('ecmascript'.includes('script', 4)); // true
```

String

- Unicode 扩展支持

支持双16位编码字符 two 16-bit code units

```
let str = '𠮷';  
console.log(str.length);          // 2  
console.log(str === '\uD842\uDFB7'); // true  
  
console.log(str.charCodeAt(0)); // 55362  
console.log(str.charCodeAt(1)); // 57271
```

支持字符以UTF-16编码，新的 codePointAt 方法

```
console.log(str.codePointAt(0)); // 134071  
console.log(str.codePointAt(1)); // 57271  
  
console.log(str.codePointAt(0) === 0x20BB7); // true
```

Classes

Classes

- ES6以前的（伪）类

```
function Animal(name){  
    this.name = name;  
}  
Animal.prototype.breathe = function () {  
    console.log(this.name + ' is breathing');  
}
```

Classes

- ES6原生支持写类了！

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  breathe() {  
    console.log(`${this.name} is breathing`);  
  }  
}
```

```
function Animal(name){  
  this.name = name;  
}  
Animal.prototype.breathe = function () {  
  console.log(this.name + ' is breathing');  
}
```

Classes

- 作为类当然要支持继承

```
class Dog extends Animal {  
  constructor(name) {  
    super(name);  
  }  
  bark() {  
    console.log(`Woof! ${this.name} is barking`);  
  }  
}
```

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  breathe() {  
    console.log(`${this.name} is breathing`);  
  }  
}
```

Classes

- 使用方法没有变化

```
let puppy = new Dog();
puppy.breathe();    // Animal 父类的方法
puppy.bark();       // Dog 子类的方法
```

```
class Dog extends Animal {
  constructor(name) {
    super(name);
  }
  bark() {
    console.log(`Woof! ${this.name} is barking`);
  }
}
```

Enhanced Object Literals

Enhanced Object Literals

- 增强的对象字面量，写法多样，更像一个类

key / value 同名时可以简写

```
var obj = {  
    // handler: handler的简写  
    handler,  
};
```

Enhanced Object Literals

- 增强的对象字面量，写法多样，更像一个类

支持直接定义原型了 __proto__

```
var obj = {  
    // handler: handler的简写  
    handler,  
  
    // __proto__属性  
    __proto__: theProtoObj,  
};
```

Enhanced Object Literals

- 增强的对象字面量，写法多样，更像一个类

方法声明简写

```
var obj = {
  // handler: handler的简写
  handler,

  // __proto__属性
  __proto__: theProtoObj,

  // 方法声明
  toString() {
    return "d " + super.toString();
  },
};
```

Enhanced Object Literals

- 增强的对象字面量，写法多样，更像一个类

动态属性名直接写表达式

```
var obj = {
  // handler: handler的简写
  handler,

  // __proto__ 属性
  __proto__: theProtoObj,

  // 方法声明
  toString() {
    return "d " + super.toString();
  },

  // 动态属性名
  [ "prop_" + ( () => 42)() ]: 42
}
```

Enhanced Object Literals

ES3-5的年代，我们还在分不清JSON和Object的区别

看了上面的高级Object，大家有些体会了不？

Arrow function

Arrow function

牢记：箭头函数总（就）是个匿名函数

Arrow function

- 语法规则

```
([param] [, param]) => {  
    statements  
}
```

```
// 无参数  
() => { ... }
```

```
// 单参数, 括号可以省略  
x => { ... }
```

```
// 多参数  
(x, y) => { ... }
```

```
// 块体  
x => { return x * x }
```

```
// 表达式体, 同上  
x => x * x
```

- 表达式体 / 块体

```
param => expression
```

```
(param1, param2) => { block }
```

Arrow function

- 举个栗子

Array的Map高阶函数

```
// ES5  
[3, 4, 5].map(function (n) {  
    return n * n;  
});
```

```
// ES6 箭头函数这么写  
[3, 4, 5].map(n => n * n);
```

Arrow function

- 箭头函数会自动帮我们处理this 😊

```
// ES5
function FancyObject() {
  var self = this;
  self.name = 'FancyObject';
  setTimeout(function () {
    self.name = 'Hello World!';
  }, 1000);
}
```

```
// ES6 Arrow Function
function FancyObject() {
  this.name = 'FancyObject';
  setTimeout(() => {
    this.name = 'Hello World!'; // this直接使用的是FancyObject
  }, 1000);
```

Destructuring

Destructuring

解构可以让我更灵活更高效的写代码，它允许按照模式匹配绑定代码，包括数组和对象。它很像一个简单对象查找，当没有找到值时，返回`undefined`。

Destructuring

- Array 数组部分

ES6提供了解构的方式，让我们更方便更灵活的将数组值拆给每个单独的变量

```
// ES5
var point = [1, 2];
var xVal = point[0],
    yVal = point[1];

console.log(xVal); // 1
console.log(yVal); // 2
```

```
// ES6
let point = [1, 2];
let [xVal, yVal] = point;

console.log(xVal); // 1
console.log(yVal); // 2
```

Destructuring

- Array 数组部分

ES6提供了解构的方式，让我们更方便更灵活的将数组值拆给每个单独的变量
对换位置也是可以的

```
// ES6
let point = [1, 2];
let [xVal, yVal] = point;
console.log(xVal); // 1
console.log(yVal); // 2
```

```
// ES6
let point = [1, 2];
let [xVal, yVal] = point;
[xVal, yVal] = [yVal, xVal];
console.log(xVal); // 2
console.log(yVal); // 1
```

Destructuring

- Array 数组部分

ES6提供了解构的方式，让我们更方便更灵活的将数组值拆给每个单独的变量
对换位置也是可以的
或者当我们忘掉了一个变量的时候，也是OK的

```
let threeD = [1, 2, 3];
let [a, , c] = threeD;

console.log(a); // 1
console.log(c); // 3
```

Destructuring

- Array 数组部分

ES6提供了解构的方式，让我们更方便更灵活的将数组值拆给每个

单独的变量

对换位置也是可以的

或者当我们忘掉了一个变量的时候，也是OK的

甚至是嵌套的二维数组

```
let nested = [1, [2, 3], 4];
let [a, [b], d] = nested;
console.log(a); // 1
console.log(b); // 2
console.log(d); // 4
```

Destructuring

- Object 对象部分

对象的解构，相对于数组就更好理解了，我们使用对象字面量中
赋值符号（冒号）左边的key直接用来赋值

```
let point = {  
    x: 1,  
    y: 2  
};  
let { x: a, y: b } = point;  
  
console.log(a); // 1  
console.log(b); // 2
```

Destructuring

- Object 对象部分

对象解构像数组一样，同样支持嵌套赋值

```
let point = {  
    x: 1,  
    y: 2,  
    z: {  
        one: 3,  
        two: 4  
    }  
};  
let { x: a, y: b, z: { one: c, two: d } } = point;  
  
console.log(a); // 1  
console.log(b); // 2  
console.log(c); // 3  
console.log(d); // 4
```

Destructuring

- 数组、对象混合结构

```
let mixed = {  
    one: 1,  
    two: 2,  
    values: [3, 4, 5]  
};  
let { one: a, two: b, values: [c, , e] } = mixed;  
  
console.log(a); // 1  
console.log(b); // 2  
console.log(c); // 3  
console.log(e); // 5
```

Destructuring

- 数组、对象混合结构

甚至，我们的function返回值也可以直接解构赋值

```
function mixed () {  
  return {  
    one: 1,  
    two: 2,  
    values: [3, 4, 5]  
  };  
}  
let { one: a, two: b, values: [c, , e] } = mixed();  
  
console.log(a); // 1  
console.log(b); // 2  
console.log(c); // 3  
console.log(e); // 5
```

Destructuring

- 特别说明

如果解构和需要赋值的变量数量不匹配，那么对应位置的变量将设置为`undefined`

```
let point = {  
  x: 1  
};  
let { x: a, y: b } = point;  
console.log(a); // 1  
console.log(b); // undefined
```

Destructuring

- 特别说明

如果我们忘了`let / const / var`的话，会抛出一个错误

```
let point = {  
  x: 1  
};  
{ x: a } = point; // 被解释为block, point中没有a这个label, 所以x没能被赋值
```

但如果我们在外面套个括号的话.....

```
let point = {  
  x: 1  
};  
({ x: a } = point);  
  
console.log(a); // 1
```

Iterators

Iterators

- **for of** 新的 for 循环

```
const arr = [1, 2, 3, 4, 5];
for (let item of arr) {
  console.log(item); // 1
                // 2
                // 3
                // 4
                // 5
}
```

Iterators

- **for of** 新的 for 循环

当然，**for of**循环里可以使用循环控制语句

```
const arr = [1, 2, 3, 4, 5];
for (let item of arr) {
  if (item > 4) {
    break;
  }
  if (0 !== item % 2) {
    continue;
  }
  console.log(item); // 2
                      // 4
}
```

Iterators

for of的内部实现，其实是使用一个枚举器实现的。

像Java和C#，ES6也为我们提供了可以自定义的枚举器，还内置了一些可以枚举的对象，如 **String, Array, TypedArray, Map and Set.**

Symbol

说到枚举器，我们先来看下这个：

Iterators

- Symbol

ES6中完全的新内容Symbol符号是一个唯一的、不可改变的数据类型，可以用来做对象的属性，来标示唯一对象，相当于数据库表中的ID字段，unique。

```
// Symbol
let s1 = Symbol('abc');
let s2 = Symbol('abc');

console.log(s1 !== s2); // true
console.log(typeof s1); // 'symbol'

let obj = {};
obj[s1] = 'abc';
console.log(obj); // Object { Symbol(abc): 'abc' }
```

Iterators

- **Symbol**

我们可以使用

Symbol.iterator

，配合**next()**方法

实现我们自定义

的枚举器，如：

```
let random1_10 = function (items = 1) {
  return {
    [Symbol.iterator]() {
      let cur = 0;
      return {
        next() {
          if (cur === items) {
            return {
              done: true
            }
          }
          ++cur;
          return {
            done: false,
            value: Math.floor(Math.random() * 10) + 1
          }
        }
      }
    };
  };
};

for (let n of random1_10(5)) {
  console.log(n); // prints 5 random numbers
}
```

Map + Set + WeakMap + WeakSet

Map + Set + WeakMap + WeakSet

ES6中新添加的四种数据集合类型，提供了更加方便的获取属性值的方法，不用像以前一样用`hasOwnProperty`来检查某个属性是属于原型链上的呢还是当前对象的。

Map + Set + WeakMap + WeakSet

- Set 是 ES6 新增的有序列表集合，它不会包含重复项

```
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size; // 2, 重复的不会再次保存;
s.has("hello"); // true
```

- Map 是 ES6 新增的有序键值对集合，key 和 value 都可以是任意类型，可以被 for ... of 遍历

```
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s); // 34
```

Map + Set + WeakMap + WeakSet

- WeakMap / WeakSet 最大的好处是可以避免内存泄漏，仅保持弱引用，也就是说它不阻止垃圾回收。垃圾回收会回收掉仅在Weak类型引用的对象。目前使用场景还不多，将来应该会很有用。

```
var ws = new WeakSet();
ws.add({ data: 42 });
// 被add的对象没有其他引用，所以不会存入ws中
```

```
var wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.get(s); // undefined
```

Module

Module

- NodeJS (CommonJS) 中的模块

```
// baz.js
let baz = 'baz';
export baz;
```

// 上面两行可以合并写成
export let baz = 'baz';

```
// app.js
import {baz} from "baz.js";
```

Module

- NodeJS (CommonJS) 中的模块

高级混合用法，使用了 * / as / 解构

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js 主App文件，import时使用了*和as，下面代码就可以使用.符号了
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js 其他文件，解构 (Destructuring) 方式直接赋值，上面介绍过了
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```

Module

- NodeJS (CommonJS) 中的模块

default 和 * 关键字

```
// lib/mathplusplus.js
export * from "lib/math";
export var e = 2.71828182846;
export default function(x) {
    return Math.exp(x);
}

// app.js
import exp, {pi, e} from "lib/mathplusplus";
alert("2π = " + exp(pi, e));
```

Promise

Generators
Koa

Babel 使用

<https://babeljs.io/repl/>

Q&A