

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证



目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

其他

- 源码
- 修订历史
- 反馈意见

读懂 ECMAScript 规格

- 1.概述
- 2.术语
- 3.抽象操作的标准流程
- 4.相等运算符
- 5.数组的空位
- 6.数组的 map 方法

1. 概述

规格文件是计算机语言的官方标准，详细描述语法规则和实现方法。

一般来说，没有必要阅读规格，除非你要写编译器。因为规格写得非常抽象和精炼，又缺乏实例，不容易理解，而且对于解决实际的应用问题，帮助不大。但是，如果你遇到疑难的语法问题，实在找不到答案，这时可以去查看规格文件，了解语言标准是怎么说的。规格是解决问题的“最后一招”。

这对 JavaScript 语言很有必要。因为它的使用场景复杂，语法规则不统一，例外很多，各种运行环境的行为不一致，导致奇怪的语法问题层出不穷，任何语法书都不可能囊括所有情况。查看规格，不失为一种解决语法问题的最可靠、最权威的终极方法。

本章介绍如何读懂 ECMAScript 6 的规格文件。

ECMAScript 6 的规格，可以在 ECMA 国际标准组织的官方网站（www.ecma-international.org/ecma-262/6.0/）免费下载和在线阅读。

这个规格文件相当庞大，一共有 26 章，A4 打印的话，足足有 545 页。它的特点就是规定得非常细致，每一个语法行为、每一个函数的实现都做了详尽的清晰的描述。基本上，编译器作者只要把每一步翻译成代码就可以了。这很大程度上，保证了所有 ES6 实现都有一致的行为。

ECMAScript 6 规格的 26 章之中，第 1 章到第 3 章是对文件本身的介绍，与语言关系不大。第 4 章是对这门语言总体设计的描述，有兴趣的读者可以读一下。第 5 章到第 8 章是语言宏观层面的描述。第 5 章是规格的名词解释和写法的介绍，第 6 章介绍数据类型，第 7 章介绍语言内部用到的抽象操作，第 8 章介绍代码如何运行。第 9 章到第 26 章介绍具体的语法。

对于一般用户来说，除了第 4 章，其他章节都涉及某一方面的细节，不用通读，只要在用到的时候，查阅相关章节即可。

2. 术语

ES6 规格使用了一些专门的术语，了解这些术语，可以帮助你读懂规格。本节介绍其中的几个。

抽象操作

所谓“抽象操作”（abstract operations）就是引擎的一些内部方法，外部不能调用。规格定义了一系列的抽象操作，规定了它们的行为，留给各种引擎自己去实现。

举例来说，`Boolean(value)` 的算法，第一步是这样的。

```
1.Let b be ToBoolean(value) .
```

这里的 `ToBoolean` 就是一个抽象操作，是引擎内部求出布尔值的算法。

许多函数的算法都会多次用到同样的步骤，所以 ES6 规格将它们抽出来，定义成“抽象操作”，方便描述。

Record 和 field

ES6 规格将键值对（key-value map）的数据结构称为 Record，其中的每一组键值对称为 field。这就是说，一个 Record 由多个 field 组成，而每个 field 都包含一个键名（key）和一个键值（value）。

[[Notation]]

ES6 规格大量使用 [[Notation]] 这种书写法，比如 [[Value]]、[[Writable]]、[[Get]]、[[Set]] 等等。它用来指代 field 的键名。

举例来说，obj 是一个 Record，它有一个 Prototype 属性。ES6 规格不会写 obj.Prototype，而是写 obj. [[Prototype]]。一般来说，使用 [[Notation]] 这种书写法的属性，都是对象的内部属性。

所有的 JavaScript 函数都有一个内部属性 [[Call]]，用来运行该函数。

```
F. [[Call]](V, argumentsList)
```

上面代码中，F 是一个函数对象，[[Call]] 是它的内部方法，F. [[call]]() 表示运行该函数，V 表示 [[Call]] 运行时 this 的值，argumentsList 则是调用时传入函数的参数。

Completion Record

每一个语句都会返回一个 Completion Record，表示运行结果。每个 Completion Record 有一个 [[Type]] 属性，表示运行结果的类型。

[[Type]] 属性有五种可能的值。

- normal
- return
- throw
- break
- continue

如果 [[Type]] 的值是 normal，就称为 normal completion，表示运行正常。其他的值，都称为 abrupt completion。其中，开发者只需要关注 [[Type]] 为 throw 的情况，即运行出错；break、continue、return 这三个值都只出现在特定场景，可以不用考虑。

3. 抽象操作的标准流程

抽象操作的运行流程，一般是下面这样。

1. Let resultCompletionRecord be AbstractOp().
2. If resultCompletionRecord is an abrupt completion, return resultCompletionRecord.
3. Let result be resultCompletionRecord. [[Value]].
4. return result.

上面的第一步是调用抽象操作 AbstractOp()，得到 resultCompletionRecord，这是一个 Completion Record。第二步，如果这个 Record 属于 abrupt completion，就将 resultCompletionRecord 返回给用户。如果此处没有返回，就表示运行结果正常，所得的值存放在 resultCompletionRecord. [[Value]] 属性。第三步，将这个值记为 result。第四步，将 result 返回给用户。

ES6 规格将这个标准流程，使用简写的方式表达。

1. Let result be AbstractOp().

```
2. ReturnIfAbrupt(result) .
3. return result .
```

这个简写方式里面的 `ReturnIfAbrupt(result)`，就代表了上面的第二步和第三步，即如果有报错，就返回错误，否则取出值。

甚至还有进一步的简写格式。

```
1. Let result be ? AbstractOp() .
2. return result .
```

上面流程的 `?`，就代表 `AbstractOp()` 可能会报错。一旦报错，就返回错误，否则取出值。

除了 `?`，ES 6 规格还使用另一个简写符号 `!`。

```
1. Let result be ! AbstractOp() .
2. return result .
```

上面流程的 `!`，代表 `AbstractOp()` 不会报错，返回的一定是 `normal completion`，总是可以取出值。

4. 相等运算符

下面通过一些例子，介绍如何使用这份规格。

相等运算符（`==`）是一个很让人头痛的运算符，它的语法行为多变，不符合直觉。这个小节就看看规格怎么规定它的行为。

请看下面这个表达式，请问它的值是多少。

```
0 == null
```

如果你不确定答案，或者想知道语言内部怎么处理，就可以去查看规格，[7.2.12](#) 小节是对相等运算符（`==`）的描述。

规格对每一种语法行为的描述，都分成两部分：先是总体的行为描述，然后是实现的算法细节。相等运算符的总体描述，只有一句话。

“The comparison `x == y`, where `x` and `y` are values, produces `true` or `false`.”

上面这句话的意思是，相等运算符用于比较两个值，返回 `true` 或 `false`。

下面是算法细节。

```
1. ReturnIfAbrupt(x).
2. ReturnIfAbrupt(y).
3. If Type(x) is the same as Type(y), then
    1. Return the result of performing Strict Equality Comparison x === y .
4. If x is null and y is undefined, return true .
5. If x is undefined and y is null, return true .
6. If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y) .
7. If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y .
8. If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y .
9. If Type(y) is Boolean, return the result of the comparison x == ToNumber(y) .
```

- 10.If `Type(x)` is either String, Number, or Symbol and `Type(y)` is Object, then return the result of the comparison `x == ToPrimitive(y)`.
- 11.If `Type(x)` is Object and `Type(y)` is either String, Number, or Symbol, then return the result of the comparison `ToPrimitive(x) == y`.
- 12.Return `false`.

上面这段算法，一共有 12 步，翻译如下。

- 1.如果 `x` 不是正常值（比如抛出一个错误），中断执行。
- 2.如果 `y` 不是正常值，中断执行。
- 3.如果 `Type(x)` 与 `Type(y)` 相同，执行严格相等运算 `x === y`。
- 4.如果 `x` 是 `null`，`y` 是 `undefined`，返回 `true`。
- 5.如果 `x` 是 `undefined`，`y` 是 `null`，返回 `true`。
- 6.如果 `Type(x)` 是数值，`Type(y)` 是字符串，返回 `x == ToNumber(y)` 的结果。
- 7.如果 `Type(x)` 是字符串，`Type(y)` 是数值，返回 `ToNumber(x) == y` 的结果。
- 8.如果 `Type(x)` 是布尔值，返回 `ToNumber(x) == y` 的结果。
- 9.如果 `Type(y)` 是布尔值，返回 `x == ToNumber(y)` 的结果。
- 10.如果 `Type(x)` 是字符串或数值或 Symbol 值，`Type(y)` 是对象，返回 `x == ToPrimitive(y)` 的结果。
- 11.如果 `Type(x)` 是对象，`Type(y)` 是字符串或数值或 Symbol 值，返回 `ToPrimitive(x) == y` 的结果。
- 12.返回 `false`。

由于 `0` 的类型是数值，`null` 的类型是 `Null`（这是规格4.3.13 小节的规定，是内部 `Type` 运算的结果，跟 `typeof` 运算符无关）。因此上面的前 11 步都得不到结果，要到第 12 步才能得到 `false`。

```
0 == null // false
```

5. 数组的空位

下面再看另一个例子。

```
const a1 = [undefined, undefined, undefined];
const a2 = [, , ,];

a1.length // 3
a2.length // 3

a1[0] // undefined
a2[0] // undefined

a1[0] === a2[0] // true
```

上面代码中，数组 `a1` 的成员是三个 `undefined`，数组 `a2` 的成员是三个空位。这两个数组很相似，长度都是 3，每个位置的成员读取出来都是 `undefined`。

但是，它们实际上存在重大差异。

```
0 in a1 // true
0 in a2 // false

a1.hasOwnProperty(0) // true
a2.hasOwnProperty(0) // false
```

```
Object.keys(a1) // ["0", "1", "2"]
Object.keys(a2) // []

a1.map(n => 1) // [1, 1, 1]
a2.map(n => 1) // [, , ]
```

上面代码一共列出了四种运算，数组 **a1** 和 **a2** 的结果都不一样。前三种运算（**in** 运算符、数组的 **hasOwnProperty** 方法、**Object.keys** 方法）都说明，数组 **a2** 取不到属性名。最后一种运算（数组的 **map** 方法）说明，数组 **a2** 没有发生遍历。

为什么 **a1** 与 **a2** 成员的行为不一致？数组的成员是 **undefined** 或空位，到底有什么不同？

规格的12.2.5 小节《数组的初始化》给出了答案。

“Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an AssignmentExpression (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.”

翻译如下。

“数组成员可以省略。只要逗号前面没有任何表达式，数组的 **length** 属性就会加 1，并且相应增加其后成员的位置索引。被省略的成员不会被定义。如果被省略的成员是数组最后一个成员，则不会导致数组 **length** 属性增加。”

上面的规格说得很清楚，数组的空位会反映在 **length** 属性，也就是说空位有自己的位置，但是这个位置的值是未定义，即这个值是不存在的。如果一定要读取，结果就是 **undefined**（因为 **undefined** 在 JavaScript 语言中表示不存在）。

这就解释了为什么 **in** 运算符、数组的 **hasOwnProperty** 方法、**Object.keys** 方法，都取不到空位的属性名。因为这个属性名根本就不存在，规格里面没说要为空位分配属性名(位置索引)，只说要为下一个元素的位置索引加 1。

至于为什么数组的 **map** 方法会跳过空位，请看下一节。

6. 数组的 **map** 方法

规格的22.1.3.15 小节定义了数组的 **map** 方法。该小节先是总体描述 **map** 方法的行为，里面没有提到数组空位。

后面的算法描述是这样的。

```
1. Let O be ToObject(this value) .
2. ReturnIfAbrupt(O) .
3. Let len be ToLength(Get(O, "length")) .
4. ReturnIfAbrupt(len) .
5. If IsCallable(callbackfn) is false , throw a TypeError exception.
6. If thisArg was supplied, let T be thisArg ; else let T be undefined .
7. Let A be ArraySpeciesCreate(O, len) .
8. ReturnIfAbrupt(A) .
9. Let k be 0.
10. Repeat, while k < len

    1. Let Pk be ToString(k) .
    2. Let kPresent be HasProperty(O, Pk) .
    3. ReturnIfAbrupt(kPresent) .
```

4.If `kPresent` is `true`, then

- 1.Let `kValue` be `Get(0, Pk)` .
2. `ReturnIfAbrupt(kValue)` .
- 3.Let `mappedValue` be `Call(callbackfn, T, «kValue, k, 0»)` .
4. `ReturnIfAbrupt(mappedValue)` .
- 5.Let `status` be `CreateDataPropertyOrThrow (A, Pk, mappedValue)` .
6. `ReturnIfAbrupt(status)` .

5.Increase `k` by 1.

11.Return `A` .

翻译如下。

- 1.得到当前数组的 `this` 对象
- 2.如果报错就返回
- 3.求出当前数组的 `length` 属性
- 4.如果报错就返回
- 5.如果 `map` 方法的参数 `callbackfn` 不可执行，就报错
- 6.如果 `map` 方法的参数之中，指定了 `this`，就让 `T` 等于该参数，否则 `T` 为 `undefined`
- 7.生成一个新的数组 `A`，跟当前数组的 `length` 属性保持一致
- 8.如果报错就返回
- 9.设定 `k` 等于 0
- 10.只要 `k` 小于当前数组的 `length` 属性，就重复下面步骤
 - 1.设定 `Pk` 等于 `ToString(k)`，即将 `K` 转为字符串
 - 2.设定 `kPresent` 等于 `HasProperty(0, Pk)`，即求当前数组有没有指定属性
 - 3.如果报错就返回
 - 4.如果 `kPresent` 等于 `true`，则进行下面步骤
 - 1.设定 `kValue` 等于 `Get(0, Pk)`，取出当前数组的指定属性
 - 2.如果报错就返回
 - 3.设定 `mappedValue` 等于 `Call(callbackfn, T, «kValue, k, 0»)`，即执行回调函数
 - 4.如果报错就返回
 - 5.设定 `status` 等于 `CreateDataPropertyOrThrow (A, Pk, mappedValue)`，即将回调函数的值放入 `A` 数组的指定位置
 - 6.如果报错就返回
 5. `k` 增加 1
- 11.返回 `A`

仔细查看上面的算法，可以发现，当处理一个全是空位的数组时，前面步骤都没有问题。进入第 10 步中第 2 步时，`kPresent` 会报错，因为空位对应的属性名，对于数组来说是不存在的，因此就会返回，不会进行后面的步骤。

```
const arr = [, , ,];
arr.map(n => {
  console.log(n);
  return 1;
}) // [, , ,]
```

上面代码中，`arr` 是一个全是空位的数组，`map` 方法遍历成员时，发现是空位，就直接跳过，不会进入回调函数。因此，回调函数里面的 `console.log` 语句根本不会执行，整个 `map` 方法返回一个全是空位的新数组。

V8 引擎对 `map` 方法的实现如下，可以看到跟规格的算法描述完全一致。

```
function ArrayMap(f, receiver) {
  CHECK_OBJECT_COERCIBLE(this, "Array.prototype.map");

  // Pull out the length so that modifications to the length in the
  // loop will not affect the looping and side effects are visible.
  var array = TO_OBJECT(this);
  var length = TO_LENGTH_OR_UINT32(array.length);
  return InnerArrayMap(f, receiver, array, length);
}

function InnerArrayMap(f, receiver, array, length) {
  if (!IS_CALLABLE(f)) throw MakeTypeError(kCalledNonCallable, f);

  var accumulator = new InternalArray(length);
  var is_array = IS_ARRAY(array);
  var stepping = DEBUG_IS_STEPPING(f);
  for (var i = 0; i < length; i++) {
    if (HAS_INDEX(array, i, is_array)) {
      var element = array[i];
      // Prepare break slots for debugger step in.
      if (stepping) %DebugPrepareStepInIfStepping(f);
      accumulator[i] = %Call(f, receiver, element, i, array);
    }
  }
  var result = new GlobalArray();
  %MoveArrayContents(accumulator, result);
  return result;
}
```

留言

14 Comments

ECMAScript 6 入门

1 Login ▾

Recommend

Tweet

Share


Sort by Best ▾



Join the discussion...

LOG IN WITH


OR SIGN UP WITH DISQUS ?

- 

reverland • 3 years ago

这个HAS_INDEX。。。

macro HAS_INDEX(array, index, is_array) = ((is_array && %_HasFastPackedElements(%IS_VAR(array))) ? (index < array.length) : (index in array));

1 ^ | ▾ • Reply • Share ›
- 

李红 • 5 months ago

虽然买了书但是还是会看看网页版，支持一波！！

^ | ▾ • Reply • Share ›



ToPrimitive是什么方法?

^ | v • Reply • Share ›



Jason Lee → ghr • 4 months ago

第10章讲Symbol.toPrimitive的时候提了一下。这个大概就是：

1. 如果定义了Symbol.toPrimitive就按这个来
2. 否则试一下valueOf()，如果返回值是primitive就用这个
3. 否则返回toString()

以上是凭记忆写的，不保证细节完全准确，如果有问题请查一下规格

^ | v • Reply • Share ›



卫思伍 • a year ago

个人觉得：7.2.12 并非是 == 的算法，12.10.3 才是说 == (Equality Comparison Operator)的算法的。原因如下：

1) 在ES5 文档的11.9.1节中描述的就是==的执行步骤,其描述的步骤和12.10.3中是一致的。ES5文档11.9.3节描述则和ES6 7.2.12 内容一致。

2) 在ES6 文档 第7章中一开始就说了：

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification。

^ | v • Reply • Share ›



ruanyf Mod → 卫思伍 • a year ago

我觉得，12.10.3 只是描述语义，说“Return the result of performing Abstract Equality Comparison *rval* == *lval*”，具体的算法还是在 7.2.12。

^ | v • Reply • Share ›



卫思伍 → ruanyf • a year ago

举一个例子，如果照着 7.2.12的方法去比较以下代码的话，好像会出问题：

```
var x = new Date('2017/01/01');
```

```
var y = new Date('2017/01/01');
```

判断 `x == y`

那么根据



可以知道实际上会走向用严格相等来判断。于是再按照严格相等写的算法去推断



see more

上一章

下一章

^ | v • Reply • Share ›



ruanyf Mod → 卫思伍 • a year ago

我查了一下，`toPrimitive` 的定义在 7.1.1。

^ | v • Reply • Share ›



卫思伍 → ruanyf • a year ago

推荐一篇文章供参考：

<https://developer.mozilla.o...>

从这篇文章也可以看出来，`==` 运算符确实用的是这种算法，我的之前的表述是不正确的。

我来重新描述一下我的看法：

在es6文档中 表达式相关的章节在12章，其中有很多涉及操作符的小节，每一小节一般都含有 **static semantics** 和 **runtime semantics** 两种内容，关于这两种内容的不同可以参考这个连接：<https://www.quora.com/Whats...>

也就是说，12章中关于操作符的描述主要集中在操作符如何正确使用和正确使用的操作符和变量连成的表达式的语义是什么。例如在 `==` 操作符的 **runtime semantics** 中就指出最后要使用 **Abstract Equality Comparison** 算法去得出最终值。

我感觉我在上面列出的推导过程，如果没有逻辑错误的话，那么很有可能是一个英语问题，理由如下：

ES6 里面的 **Strict Equality Comparison** 算法描述的是：

If x and y are the same Object value, return true

而在ES5里面，**Strict Equality Comparison** 算法则描述为：

Return true if x and y refer to the same object. Otherwise, return false。

4 ^ | v • Reply • Share ›



卫思伍 → ruanyf • a year ago

恩，实际上我是猜测它用`toPrimitive` 去获取 `object` 对象的值的。并且我也做了例子

```
var x = new Date('2017/01/01');
```

```
var flag_x = x[Symbol.toPrimitive].call(x, 'default');
```

```
console.log(x == flag_x)
```

结果是真的。

^ | v • Reply • Share ›



Tom Tang • 2 years ago

我曹，厉害了

^ | v • Reply • Share ›



Bf Dong • 2 years ago

今天来查东西的时候发现了左上角 的搜索的 `input outline` 去掉体验会好一点儿😁；一切为了体验。。

^ | v • Reply • Share ›



ruanyf Mod → Bf Dong • 2 years ago

谢谢指出，已经取消。

^ | v • Reply • Share ›



Chenghao Lan • 3 years ago

对解答面试题特别有用-_-#

^ | v • Reply • Share ›

Proxy和Reflect

65 comments • 3 years ago

liudeyin — https://uploads.disquscdn.c... ...

对象和函数的扩展

90 comments • 5 years ago

yunnysunny — 可惜Null传导运算符现在还不支持

数组的扩展

71 comments • 5 years ago

liyun — 不好意思，之前是我理解错了浅拷贝和深拷贝的概念

Promise对象

106 comments • 5 years ago

冯飞林 — 谢谢