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. 数组的扩展
8. 函数的扩展
9. 对象的扩展
10. Symbol
11. Proxy和Reflect
12. 二进制数组
13. Set和Map数据结构
14. Iterator和for...of循环
15. Generator函数
16. Promise对象
17. 异步操作和Async函数
18. Class
19. Decorator
20. Module
21. 编程风格
22. 读懂规格

数组的扩展

1. Array.from()
2. Array.of()
3. 数组实例的copyWithin()
4. 数组实例的find()和findIndex()
5. 数组实例的fill()
6. 数组实例的entries()，keys()和values()
7. 数组实例的includes()
8. 数组的空位

---

1. Array.from()

`Array.from`方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括ES6新增的数据结构Set和Map）。

下面是一个类似数组的对象，`Array.from`将它转为真正的数组。

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

// ES5的写法
```

23. 参考链接

其他

- 源码
- 修订历史
- 反馈意见

```
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ES6的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

实际应用中，常见的类似数组的对象是DOM操作返回的NodeList集合，以及函数内部的`arguments`对象。`Array.from`都可以将它们转为真正的数组。

```
// NodeList对象
let ps = document.querySelectorAll('p');
Array.from(ps).forEach(function (p) {
  console.log(p);
});

// arguments对象
function foo() {
  var args = Array.from(arguments);
  // ...
}
```

上面代码中，`querySelectorAll`方法返回的是一个类似数组的对象，只有将这个对象转为真正的数组，才能使用`forEach`方法。

只要是部署了Iterator接口的数据结构，`Array.from`都能将其转为数组。

```
Array.from('hello')
// ['h', 'e', 'l', 'l', 'o']

let namesSet = new Set(['a', 'b'])
Array.from(namesSet) // ['a', 'b']
```

上面代码中，字符串和Set结构都具有Iterator接口，因此可以被`Array.from`转为真正

的数组。

如果参数是一个真正的数组，`Array.from`会返回一个一模一样的新数组。

```
Array.from([1, 2, 3])
// [1, 2, 3]
```

值得提醒的是，扩展运算符（`...`）也可以将某些数据结构转为数组。

```
// arguments对象
function foo() {
  var args = [...arguments];
}

// NodeList对象
[...document.querySelectorAll('div')]
```

扩展运算符背后调用的是遍历器接口（`Symbol.iterator`），如果一个对象没有部署这个接口，就无法转换。`Array.from`方法则是还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有`length`属性。因此，任何有`length`属性的对象，都可以通过`Array.from`方法转为数组，而此时扩展运算符就无法转换。

```
Array.from({ length: 3 });
// [ undefined, undefined, undefined ]
```

上面代码中，`Array.from`返回了一个具有三个成员的数组，每个位置的值都是`undefined`。扩展运算符转换不了这个对象。

对于还没有部署该方法的浏览器，可以用`Array.prototype.slice`方法替代。

```
const toArray = (() =>
  Array.from ? Array.from : obj => [].slice.call(obj)
)();
```

`Array.from`还可以接受第二个参数，作用类似于数组的`map`方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

```
Array.from(arrayLike, x => x * x);
// 等同于
Array.from(arrayLike).map(x => x * x);

Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

下面的例子是取出一组DOM节点的文本内容。

```
let spans = document.querySelectorAll('span.name');

// map()
let names1 = Array.prototype.map.call(spans, s => s.textContent);

// Array.from()
let names2 = Array.from(spans, s => s.textContent)
```

下面的例子将数组中布尔值为`false`的成员转为`0`。

```
Array.from([1, , 2, , 3], (n) => n || 0)
// [1, 0, 2, 0, 3]
```

另一个例子是返回各种数据的类型。

```
function typesOf () {
```

```
  return Array.from(arguments, value => typeof value)
}
typesOf(null, [], NaN)
// ['object', 'object', 'number']
```

如果`map`函数里面用到了`this`关键字，还可以传入`Array.from`的第三个参数，用来绑定`this`。

`Array.from()`可以将各种值转为真正的数组，并且还提供`map`功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法。

```
Array.from({ length: 2 }, () => 'jack')
// ['jack', 'jack']
```

上面代码中，`Array.from`的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。

`Array.from()`的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，可以避免JavaScript将大于`\uFFFF`的Unicode字符，算作两个字符的bug。

```
function countSymbols(string) {
  return Array.from(string).length;
}
```

---

2. Array.of()

`Array.of`方法用于将一组值，转换为数组。

```
Array.of(3, 11, 8) // [3,11,8]
Array.of(3) // [3]
Array.of(3).length // 1
```

这个方法的主要目的，是弥补数组构造函数`Array()`的不足。因为参数个数的不同，会导致`Array()`的行为有差异。

```
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

上面代码中，`Array`方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于2个时，`Array()`才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。

`Array.of`基本上可以用来替代`Array()`或`new Array()`，并且不存在由于参数不同而导致的重载。它的行为非常统一。

```
Array.of() // []
Array.of(undefined) // [undefined]
Array.of(1) // [1]
Array.of(1, 2) // [1, 2]
```

`Array.of`总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

`Array.of`方法可以用下面的代码模拟实现。

```
function ArrayOf(){
```

```
  return [].slice.call(arguments);
}
```

---

3. 数组实例的copyWithin()

数组实例的`copyWithin`方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
```

它接受三个参数。

- target（必需）：从该位置开始替换数据。
- start（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。
- end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

这三个参数都应该是数值，如果不是，会自动转为数值。

```
[1, 2, 3, 4, 5].copyWithin(0, 3)
// [4, 5, 3, 4, 5]
```

上面代码表示将从3号位直到数组结束的成员（4和5），复制到从0号位开始的位置，结果覆盖了原来的1和2。

下面是更多例子。

```
// 将3号位复制到0号位
[1, 2, 3, 4, 5].copyWithin(0, 3, 4)
// [4, 2, 3, 4, 5]

// -2相当于3号位，-1相当于4号位
[1, 2, 3, 4, 5].copyWithin(0, -2, -1)
// [4, 2, 3, 4, 5]

// 将3号位复制到0号位
[].copyWithin.call({length: 5, 3: 1}, 0, 3)
// {0: 1, 3: 1, length: 5}

// 将2号位到数组结束，复制到0号位
var i32a = new Int32Array([1, 2, 3, 4, 5]);
i32a.copyWithin(0, 2);
// Int32Array [3, 4, 5, 4, 5]

// 对于没有部署TypedArray的copyWithin方法的平台
// 需要采用下面的写法
[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);
// Int32Array [4, 2, 3, 4, 5]
```

---

4. 数组实例的find()和findIndex()

数组实例的`find`方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为`true`的成员，然后返回该成员。如果没有符合条件的成员，则返回`undefined`。

```
[1, 4, -5, 10].find((n) => n < 0)
// -5
```

上面代码找出数组中第一个小于0的成员。

```
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

上面代码中，`find`方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的`findIndex`方法的用法与`find`方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回`-1`。

```
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

这两个方法都可以接受第二个参数，用来绑定回调函数的`this`对象。

另外，这两个方法都可以发现`NaN`，弥补了数组的`IndexOf`方法的不足。

```
[NaN].indexOf(NaN)
// -1

[NaN].findIndex(y => Object.is(NaN, y))
// 0
```

上面代码中，`indexOf`方法无法识别数组的`NaN`成员，但是`findIndex`方法可以借助`Object.is`方法做到。

5. 数组实例的fill()

`fill`方法使用给定值，填充一个数组。

```
['a', 'b', 'c'].fill(7)
// [7, 7, 7]

new Array(3).fill(7)
// [7, 7, 7]
```

上面代码表明，`fill`方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。

`fill`方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

```
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

上面代码表示，`fill`方法从1号位开始，向原数组填充7，到2号位之前结束。

---

6. 数组实例的entries()，keys()和values()

ES6提供三个新的方法——`entries()`，`keys()`和`values()`——用于遍历数组。它们都返回一个遍历器对象（详见《Iterator》一章），可以用`for...of`循环进行遍历，唯一的区别是`keys()`是对键名的遍历、`values()`是对键值的遍历，`entries()`是对键值对的遍历。

```
for (let index of ['a', 'b'].keys()) {
  console.log(index);
}
// 0
// 1

for (let elem of ['a', 'b'].values()) {
  console.log(elem);
}
// 'a'
// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {
  console.log(index, elem);
}
// 0 "a"
// 1 "b"
```

如果不使用`for...of`循环，可以手动调用遍历器对象的`next`方法，进行遍历。

```
let letter = ['a', 'b', 'c'];
let entries = letter.entries();
console.log(entries.next().value); // [0, 'a']
console.log(entries.next().value); // [1, 'b']
console.log(entries.next().value); // [2, 'c']
```

---

7. 数组实例的includes()

`Array.prototype.includes`方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的`includes`方法类似。该方法属于ES7，但Babel转码器已经支持。

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, NaN].includes(NaN); // true
```

该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

```
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
```

没有该方法之前，我们通常使用数组的`indexOf`方法，检查是否包含某个值。

```
if (arr.indexOf(el) !== -1) {
  // ...
}
```

`indexOf`方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。二是，它内部使用严格相当运算符（===）进行判断，这会导致对`NaN`的误判。

```
[NaN].indexOf(NaN)
// -1
```

`includes`使用的是不一样的判断算法，就没有这个问题。

```
[NaN].includes(NaN)
// true
```

下面代码用来检查当前环境是否支持该方法，如果不支持，部署一个简易的替代版本。

```
const contains = (() =>
  Array.prototype.includes
    ? (arr, value) => arr.includes(value)
    : (arr, value) => arr.some(el => el === value)
)();
contains(["foo", "bar"], "baz"); // => false
```

另外，Map和Set数据结构有一个`has`方法，需要注意与`includes`区分。

- Map结构的`has`方法，是用来查找键名的，比如`Map.prototype.has(key)`、`WeakMap.prototype.has(key)`、`Reflect.has(target, propertyKey)`。
- Set结构的`has`方法，是用来查找值的，比如`Set.prototype.has(value)`、`WeakSet.prototype.has(value)`。

---

8. 数组的空位

数组的空位指，数组的某一个位置没有任何值。比如，`Array`构造函数返回的数组都是空位。

```
Array(3) // [, , ,]
```

上面代码中，`Array(3)`返回一个具有3个空位的数组。

注意，空位不是`undefined`，一个位置的值等于`undefined`，依然是有值的。空位是没有任何值，`in`运算符可以说明这一点。

```
0 in [undefined, undefined, undefined] // true
0 in [, , ,] // false
```

上面代码说明，第一个数组的0号位置是有值的，第二个数组的0号位置没有值。

ES5对空位的处理，已经很不一致了，大多数情况下会忽略空位。

- `forEach()`，`filter()`，`every()` 和`some()`都会跳过空位。
- `map()`会跳过空位，但会保留这个值
- `join()`和`toString()`会将空位视为`undefined`，而`undefined`和`null`会被处理成空字符串。

```
// forEach方法
[,'a'].forEach((x,i) => console.log(i)); // 1

// filter方法
['a',,'b'].filter(x => true) // ['a','b']

// every方法
[,'a'].every(x => x==='a') // true

// some方法
[,'a'].some(x => x !== 'a') // false

// map方法
[,'a'].map(x => 1) // [,1]

// join方法
[,'a',undefined,null].join('#') // "#a##"

// toString方法
[,'a',undefined,null].toString() // ",a,,"
```

ES6则是明确将空位转为`undefined`。

`Array.from`方法会将数组的空位，转为`undefined`，也就是说，这个方法不会忽略空位。

```
Array.from(['a',,'b'])
// [ "a", undefined, "b" ]
```

扩展运算符（`...`）也会将空位转为`undefined`。

```
[...['a',,'b']]
// [ "a", undefined, "b" ]
```

`copyWithin()`会连空位一起拷贝。

```
[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]
```

`fill()`会将空位视为正常的数组位置。

```
new Array(3).fill('a') // ["a","a","a"]
```

`for...of`循环也会遍历空位。

```
let arr = [, ,];
for (let i of arr) {
  console.log(1);
}
// 1
// 1
```

上面代码中，数组`arr`有两个空位，`for...of`并没有忽略它们。如果改成`map`方法遍历，空位是会跳过的。

`entries()`、`keys()`、`values()`、`find()`和`findIndex()`会将空位处理成`undefined`。

```
// entries()
[...[,'a'].entries()] // [[0,undefined], [1,"a"]]

// keys()
[...[,'a'].keys()] // [0,1]

// values()
[...[,'a'].values()] // [undefined,"a"]

// find()
[,'a'].find(x => true) // undefined

// findIndex()
[,'a'].findIndex(x => true) // 0
```

由于空位的处理规则非常不统一，所以建议避免出现空位。

留言