## Es6 知识点

#### 字符串的扩展

##### includes(), startsWith(), endsWith()

传统上，JavaScript只有indexOf方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6又提供了三种新方法。

includes()：返回布尔值，表示是否找到了参数字符串。

startsWith()：返回布尔值，表示参数字符串是否在源字符串的头部。

endsWith()：返回布尔值，表示参数字符串是否在源字符串的尾部。

##### repeat()

repeat方法返回一个新字符串，表示将原字符串重复n次。

##### padStart()，padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。padStart()用于头部补全，padEnd()用于尾部补全。

'x'.padStart(5, 'ab') *// 'ababx'*'x'.padStart(4, 'ab') *// 'abax'*

'x'.padEnd(5, 'ab') *// 'xabab'*'x'.padEnd(4, 'ab') *// 'xaba'*

##### String.raw()

ES6还为原生的String对象，提供了一个raw方法。

String.raw方法，往往用来充当模板字符串的处理函数，返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，对应于替换变量后的模板字符串。

String.raw`Hi\n${2+3}!`;*// "Hi\\n5!"*

String.raw`Hi\u000A!`;*// 'Hi\\u000A!'*

如果原字符串的斜杠已经转义，那么String.raw不会做任何处理。

String.raw`Hi\\n`*// "Hi\\n"*

# **正则的扩展**

## **RegExp构造函数**

但是，ES5不允许此时使用第二个参数，添加修饰符，否则会报错。

**var** regex = **new** RegExp(/xyz/, 'i');*// Uncaught TypeError: Cannot supply flags when constructing one RegExp from another*

ES6改变了这种行为。如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

**new** RegExp(/abc/ig, 'i').flags*// "i"*

上面代码中，原有正则对象的修饰符是ig，它会被第二个参数i覆盖。

## **u修饰符**

ES6对正则表达式添加了u修饰符，含义为“Unicode模式”，用来正确处理大于\uFFFF的Unicode字符。也就是说，会正确处理四个字节的UTF-16编码。

/^\uD83D/u.test('\uD83D\uDC2A')*// false*

/^\uD83D/.test('\uD83D\uDC2A')*// true*

****（1）点字符****

点（.）字符在正则表达式中，含义是除了换行符以外的任意单个字符。对于码点大于0xFFFF的Unicode字符，点字符不能识别，必须加上u修饰符。

**var** s = '';

/^.$/.test(s) *// false*

/^.$/u.test(s) *// true*

上面代码表示，如果不添加u修饰符，正则表达式就会认为字符串为两个字符，从而匹配失败。

## **y 修饰符**

除了u修饰符，ES6还为正则表达式添加了y修饰符，叫做“粘连”（sticky）修饰符。

y修饰符的作用与g修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始。不同之处在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

**var** s = 'aaa\_aa\_a';**var** r1 = /a+/g;**var** r2 = /a+/y;

r1.exec(s) *// ["aaa"]*

r2.exec(s) *// ["aaa"]*

r1.exec(s) *// ["aa"]*

r2.exec(s) *// null*

上面代码有两个正则表达式，一个使用g修饰符，另一个使用y修饰符。这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是\_aa\_a。由于g修饰没有位置要求，所以第二次执行会返回结果，而y修饰符要求匹配必须从头部开始，所以返回null。

## **sticky属性**

与y修饰符相匹配，ES6的正则对象多了sticky属性，表示是否设置了y修饰符。

**var** r = /hello\d/y;

r.sticky *// true*

## **flags属性**

ES6为正则表达式新增了flags属性，会返回正则表达式的修饰符。

*// ES5的source属性// 返回正则表达式的正文*

/abc/ig.source*// "abc"*

*// ES6的flags属性// 返回正则表达式的修饰符*

/abc/ig.flags*// 'gi'*

# **数值的扩展**

## **Number.isFinite(), Number.isNaN()**

ES6在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。

Number.isFinite()用来检查一个数值是否为有限的（finite）。

Number.isFinite(15); *// true*

Number.isFinite(0.8); *// true*

Number.isFinite(NaN); *// false*

Number.isFinite(Infinity); *// false*

Number.isFinite(-Infinity); *// false*

Number.isFinite('foo'); *// false*

Number.isFinite('15'); *// false*

Number.isFinite(true); *// false*

Number.isNaN()用来检查一个值是否为NaN。

Number.isNaN(NaN) *// true*

Number.isNaN(15) *// false*

Number.isNaN('15') *// false*

Number.isNaN(true) *// false*

Number.isNaN(9/NaN) *// true*

Number.isNaN('true'/0) *// true*

Number.isNaN('true'/'true') *// true*

## **Number.parseInt(), Number.parseFloat()**

ES6将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。

## **Number.isInteger()**

Number.isInteger()用来判断一个值是否为整数。需要注意的是，在JavaScript内部，整数和浮点数是同样的储存方法，所以3和3.0被视为同一个值。

Number.isInteger(25) *// true*

Number.isInteger(25.0) *// true*

Number.isInteger(25.1) *// false*

Number.isInteger("15") *// false*

Number.isInteger(true) *// false*

## **Number.EPSILON**

ES6在Number对象上面，新增一个极小的常量Number.EPSILON。

Number.EPSILON*// 2.220446049250313e-16*Number.EPSILON.toFixed(20)*// '0.00000000000000022204'*

引入一个这么小的量的目的，在于为浮点数计算，设置一个误差范围。我们知道浮点数计算是不精确的。

0.1 + 0.2*// 0.30000000000000004*

0.1 + 0.2 - 0.3*// 5.551115123125783e-17*

5.551115123125783e-17.toFixed(20)*// '0.00000000000000005551'*

但是如果这个误差能够小于Number.EPSILON，我们就可以认为得到了正确结果。

5.551115123125783e-17 < Number.EPSILON*// true*

因此，Number.EPSILON的实质是一个可以接受的误差范围。

**function** **withinErrorMargin** (left, right) {

**return** Math.abs(left - right) < Number.EPSILON;

}

withinErrorMargin(0.1 + 0.2, 0.3)*// true*

withinErrorMargin(0.2 + 0.2, 0.3)*// false*

上面的代码为浮点数运算，部署了一个误差检查函数。

# **数组的扩展**

## **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）。

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 ]*

## **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*

## **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)

## **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。

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。

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*

## **数组实例的fill()**

fill方法使用给定值，填充一个数组。

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']*

## **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']*

## **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*

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

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)。

## **数组的空位**

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。

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

扩展运算符（...）也会将空位转为undefined。

copyWithin()会连空位一起拷贝。

fill()会将空位视为正常的数组位置。

for...of循环也会遍历空位。

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。

# **函数的扩展**

## **函数参数的默认值**

在ES6之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

**function** **log**(x, y) { y = y || 'World'; console.log(x, y);}

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

**function** **log**(x, y = 'World') { console.log(x, y); }

另外，一个容易忽略的地方是，如果参数默认值是变量，那么参数就不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

**let** x = 99;

**function** **foo**(p = x + 1) {console.log(p);}

foo() *// 100*

x = 200; foo() *// 201*

### **与解构赋值默认值结合使用**

参数默认值可以与解构赋值的默认值，结合起来使用。

**function** **foo**({x, y = 5}) { console.log(x, y); }