1.let声明变量

(1).仅在块级作用域中有效，适合用于for循环中定义i值，每次循环都是一个不同的变量i；

(2).不存在变量提升

console.log(foo); // 输出undefined

console.log(bar); // 报错ReferenceError

var foo = 2;

let bar = 2;

(3).暂时性死区（TDZ）

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。ES6明确规定，如果区块中存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

if (true) {

// TDZ开始

tmp = 'abc'; // ReferenceError

console.log(tmp); // ReferenceError

let tmp; // TDZ结束

console.log(tmp); // undefined

tmp = 123;

console.log(tmp); // 123

}

这样的设计是为了让程序员养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。

(4).不允许重复声明

// 报错

function () {

let a = 10;

var a = 1;

}

// 报错

function () {

let a = 10;

let a = 1;

}

function func(arg) {

let arg; // 报错

}

function func(arg) {

{

let arg; // 不报错

}

}

2.块级作用域

(1).块级作用域的出现，实际上使得获得广泛应用的立即执行匿名函数（IIFE）不再必要了。

// IIFE写法

(function () {

var tmp = ...;

...

}());

// 块级作用域写法

{

let tmp = ...;

...

}

(2). ES5规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明，ES6引入了块级作用域，明确允许在块级作用域之中声明函数。

并且ES6规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

function f() { console.log('I am outside!'); }

(function () {

if (false) {

// 重复声明一次函数f

function f() { console.log('I am inside!'); }

}

f();

}());

上面代码在ES5中运行，会得到“I am inside!”，因为在if内声明的函数f会被提升到函数头部，实际运行的代码如下。

// ES5版本

function f() { console.log('I am outside!'); }

(function () {

function f() { console.log('I am inside!'); }

if (false) {

}

f();

}());

ES6的运行结果就完全不一样了，会得到“I am outside!”。因为块级作用域内声明的函数类似于let，对作用域之外没有影响，实际运行的代码如下。

// ES6版本

function f() { console.log('I am outside!'); }

(function () {

f();

}());

很显然，这种行为差异会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6在[附录B](http://www.ecma-international.org/ecma-262/6.0/index.html#sec-block-level-function-declarations-web-legacy-compatibility-semantics)里面规定，浏览器的实现可以不遵守上面的规定，有自己的[行为方式](http://stackoverflow.com/questions/31419897/what-are-the-precise-semantics-of-block-level-functions-in-es6)。

* 允许在块级作用域内声明函数。
* 函数声明类似于var，即会提升到全局作用域或函数作用域的头部。
* 同时，函数声明还会提升到所在的块级作用域的头部。

注意，上面三条规则只对ES6的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作let处理。

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

3.const命令

const声明一个只读的常量。一旦声明，常量的值就不能改变。

const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。

const的作用域与let命令相同：只在声明所在的块级作用域内有效。

const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

const声明的常量，也与let一样不可重复声明。

对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。const命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，所以将一个对象声明为常量必须非常小心。

const foo = {};

foo.prop = 123;

foo.prop

// 123

foo = {}; // TypeError: "foo" is read-only

4.全局对象的属性

ES5之中，全局对象的属性与全局变量是等价的。

var a = 1;

window.a // 1

ES6为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是全局对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于全局对象的属性。也就是说，从ES6开始，全局变量将逐步与全局对象的属性脱钩。

let b = 1;

window.b // undefined

5.解构赋值

(1)数组的解构赋值

let [foo, [[bar], baz]] = [1, [[2], 3]];

foo // 1

bar // 2

baz // 3

let [ , , third] = ["foo", "bar", "baz"];

third // "baz"

let [x, , y] = [1, 2, 3];

x // 1

y // 3

let [head, ...tail] = [1, 2, 3, 4];

head // 1

tail // [2, 3, 4]

let [x, y, ...z] = ['a'];

x // "a"

y // undefined

z // []

如果解构不成功，变量的值就等于undefined。

(2).对象的解构赋值

var { foo, bar } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

var { bar, foo } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

var { baz } = { foo: "aaa", bar: "bbb" };

baz // undefined

(3).字符串的解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

const [a, b, c, d, e] = 'hello';

a // "h"

b // "e"

c // "l"

d // "l"

e // "o"

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。

let {length : len} = 'hello';

len // 5

(4).数值和布尔值的解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

let {toString: s} = 123;

s === Number.prototype.toString // true

let {toString: s} = true;

s === Boolean.prototype.toString // true

上面代码中，数值和布尔值的包装对象都有toString属性，因此变量s都能取到值。

解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。

let { prop: x } = undefined; // TypeError

let { prop: y } = null; // TypeError

(5).函数参数的解构赋值

函数的参数也可以使用解构赋值。

function add([x, y]){

return x + y;

}

add([1, 2]); // 3

上面代码中，函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量x和y。对于函数内部的代码来说，它们能感受到的参数就是x和y。

下面是另一个例子。

[[1, 2], [3, 4]].map(([a, b]) => a + b);

// [ 3, 7 ]

(6).解构赋值的作用

①交换变量的值；

②从函数返回多个值

③函数参数的定义

④提取JSON数据

⑤函数参数的默认值

⑥遍历Map结构

⑦输入模块的指定方法

6.字符串的扩展

(1).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。

这三个方法都支持第二个参数，表示开始搜索的位置。使用第二个参数n时，endsWith的行为与其他两个方法有所不同。它针对前n个字符，而其他两个方法针对从第n个位置直到字符串结束。

(2).repeat()

repeat方法返回一个新字符串，表示将原字符串重复n次。

'x'.repeat(3) // "xxx"

'hello'.repeat(2) // "hellohello"

'na'.repeat(0) // ""

(3).模板字符串

传统的JavaScript语言，输出模板通常是这样写的。

$('#result').append(

'There are <b>' + basket.count + '</b> ' +

'items in your basket, ' +

'<em>' + basket.onSale +

'</em> are on sale!'

);

上面这种写法相当繁琐不方便，ES6引入了模板字符串解决这个问题。

$('#result').append(`

There are <b>${basket.count}</b> items

in your basket, <em>${basket.onSale}</em>

are on sale!

`);

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

大括号内部可以放入任意的JavaScript表达式，可以进行运算，以及引用对象属性。

var x = 1;

var y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

`${x} + ${y \* 2} = ${x + y \* 2}`

// "1 + 4 = 5"

var obj = {x: 1, y: 2};

`${obj.x + obj.y}`

// 3

模板字符串之中还能调用函数。

function fn() {

return "Hello World";

}

`foo ${fn()} bar`

// foo Hello World bar

7.正则的扩展

(1). 在ES5中，RegExp构造函数的参数有两种情况。

第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。

var regex = new RegExp('xyz', 'i');

// 等价于

var regex = /xyz/i;

第二种情况是，参数是一个正则表示式，这时会返回一个原有正则表达式的拷贝。

var regex = new RegExp(/xyz/i);

// 等价于

var regex = /xyz/i;

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。

(2).字符串的正则方法

字符串对象共有4个方法，可以使用正则表达式：match()、replace()、search()和split()。

ES6将这4个方法，在语言内部全部调用RegExp的实例方法，从而做到所有与正则相关的方法，全都定义在RegExp对象上。

* String.prototype.match 调用 RegExp.prototype[Symbol.match]
* String.prototype.replace 调用 RegExp.prototype[Symbol.replace]
* String.prototype.search 调用 RegExp.prototype[Symbol.search]
* String.prototype.split 调用 RegExp.prototype[Symbol.split]

8.数值的扩展

(1).新增Number.isFinite(),Number.isNaN()

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

Number.isNaN()用来检查一个值是否为NaN。

这两个方法只对数值有效，非数值一律返回false。

(2).Number.parseInt(),Numner.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。

// ES5的写法

parseInt('12.34') // 12

parseFloat('123.45#') // 123.45

// ES6的写法

Number.parseInt('12.34') // 12

Number.parseFloat('123.45#') // 123.45

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

Number.parseInt === parseInt // true

Number.parseFloat === parseFloat // true

(3).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

(4).Math方法的扩展

①Math.trunc()

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

Math.trunc(4.1) // 4

Math.trunc(4.9) // 4

Math.trunc(-4.1) // -4

Math.trunc(-4.9) // -4

Math.trunc(-0.1234) // -0

对于非数值，Math.trunc内部使用Number方法将其先转为数值。

Math.trunc('123.456')

// 123

对于空值和无法截取整数的值，返回NaN。

Math.trunc(NaN); // NaN

Math.trunc('foo'); // NaN

Math.trunc(); // NaN

②Math.sign()

Math.sign方法用来判断一个数到底是正数、负数、还是零。

它会返回五种值。

* 参数为正数，返回+1；
* 参数为负数，返回-1；
* 参数为0，返回0；
* 参数为-0，返回-0;
* 其他值，返回NaN。

9.数组的扩展

(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

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);

// ...

}

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]

(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。

(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

(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(),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

10.函数

(1).参数默认值

function log(x, y = 'World') {

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello

(2).rest参数

ES6引入rest参数（形式为“...变量名”），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

function add(...values) {

let sum = 0;

for (var val of values) {

sum += val;

}

return sum;

}

add(2, 5, 3) // 10

上面代码的add函数是一个求和函数，利用rest参数，可以向该函数传入任意数目的参数。

下面是一个rest参数代替arguments变量的例子。

// arguments变量的写法

function sortNumbers() {

return Array.prototype.slice.call(arguments).sort();

}

// rest参数的写法

const sortNumbers = (...numbers) => numbers.sort();

注意，rest参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

(3).扩展运算符

扩展运算符（spread）是三个点（...）。它好比rest参数的逆运算，将一个数组转为用逗号分隔的参数序列。

console.log(...[1, 2, 3])

// 1 2 3

console.log(1, ...[2, 3, 4], 5)

// 1 2 3 4 5

[...document.querySelectorAll('div')]

// [<div>, <div>, <div>]

该运算符主要用于函数调用。

function push(array, ...items) {

array.push(...items);

}

function add(x, y) {

return x + y;

}

var numbers = [4, 38];

add(...numbers) // 42

上面代码中，array.push(...items)和add(...numbers)这两行，都是函数的调用，它们的都使用了扩展运算符。该运算符将一个数组，变为参数序列。

扩展运算符与正常的函数参数可以结合使用，非常灵活。

function f(v, w, x, y, z) { }

var args = [0, 1];

f(-1, ...args, 2, ...[3]);

// ES5的写法

function f(x, y, z) {

// ...

}

var args = [0, 1, 2];

f.apply(null, args);

// ES6的写法

function f(x, y, z) {

// ...

}

var args = [0, 1, 2];

f(...args);

(4).name属性

函数的name属性，返回该函数的函数名

function foo() {}

foo.name // "foo"

var func1 = function () {};

// ES5

func1.name // ""

// ES6

func1.name // "func1"

(5).箭头函数

ES6允许使用“箭头”（=>）定义函数。

var f = v => v;

上面的箭头函数等同于：

var f = function(v) {

return v;

};

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

var f = () => 5;

// 等同于

var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;

// 等同于

var sum = function(num1, num2) {

return num1 + num2;

};

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。

var sum = (num1, num2) => { return num1 + num2; }

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号。

var getTempItem = id => ({ id: id, name: "Temp" });

箭头函数有几个使用注意点。

（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用Rest参数代替。

（4）不可以使用yield命令，因此箭头函数不能用作Generator函数。