1、let命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。  
  
var命令会发生”变量提升“现象，即变量可以在声明之前使用，值为undefined。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

let不允许在相同作用域内，重复声明同一个变量。  
块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（IIFE）不再必要了。

let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”

// IIFE 写法

(function () {

var tmp = ...;

...

}());

// 块级作用域写法

{

let tmp = ...;

...

}

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。  
在块级作用域之前加上do，使它变为do表达式。

let x = do {

let t = f();

t \* t + 1;

};

上面代码中，变量x会得到整个块级作用域的返回值。

Let 不允许重复声明，es6的块级作用域

2. const声明一个只读的常量。一旦声明，常量的值就不能改变。对于const  
来说，只声明不赋值，就会报错。

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。

3. var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从ES6开始，全局变量将逐步与顶层对象的属性脱钩。

window.a = 1;

a // 1 顶层对象属性

a = 2; 全局变量

window.a // 2

4. 以前，为变量赋值，只能直接指定值。数组的解构赋值

let a = 1;

let b = 2;

let c = 3;

ES6 允许写成下面这样。

let [a, b, c] = [1, 2, 3];

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

head // 1

tail // [2, 3, 4]

5对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

let { 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。

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

foo // "aaa"

bar // "bbb"

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

baz // undefined

6. 变量的解构赋值用途

**（1）交换变量的值**

let x = 1;

let y = 2;

[x, y] = [y, x];

上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。

**（2）从函数返回多个值**

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

// 返回一个数组

function example() {

return [1, 2, 3];

}

let [a, b, c] = example();

// 返回一个对象

function example() {

return {

foo: 1,

bar: 2

};

}

let { foo, bar } = example();

**（3）函数参数的定义**

解构赋值可以方便地将一组参数与变量名对应起来。

// 参数是一组有次序的值

function f([x, y, z]) { ... }

f([1, 2, 3]);

// 参数是一组无次序的值

function f({x, y, z}) { ... }

f({z: 3, y: 2, x: 1});

**（4）提取JSON数据**

解构赋值对提取JSON对象中的数据，尤其有用。

let jsonData = {

id: 42,

status: "OK",

data: [867, 5309]

};

let { id, status, data: number } = jsonData;

console.log(id, status, number);

// 42, "OK", [867, 5309]

上面代码可以快速提取 JSON 数据的值。

7. codePointAt方法会正确返回32位的UTF-16字符的码点。对于那些两个字节储存的常规字符，它的返回结果与charCodeAt方法相同。

codePointAt方法返回的是码点的十进制值，如果想要十六进制的值，可以使用toString方法转换一下。

var s = '𠮷a';

s.codePointAt(0) // 134071

s.codePointAt(1) // 57271

s.codePointAt(2) // 97

s.codePointAt(0).toString(16) // "20bb7"

s.codePointAt(2).toString(16) // "61"

for (let ch of s) for...of循环

* var text = String.fromCodePoint(0x20BB7);  
  8. **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。

var s = 'Hello world!';

s.startsWith('Hello') // true

s.endsWith('!') // true

s.includes('o') // true

9. 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) // ""

10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'

11. 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。

12. 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"

13. ES6 提供了二进制和八进制数值的新的写法，分别用前缀0b（或0B）和0o（或0O）表示。

0b111110111 === 503 // true

0o767 === 503 // true

如果要将0b和0o前缀的字符串数值转为十进制，要使用Number方法。

Number('0b111') // 7

Number('0o10') // 8

14. 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

它们与传统的全局方法isFinite()和isNaN()的区别在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，Number.isFinite()对于非数值一律返回false, Number.isNaN()只有对于NaN才返回true，非NaN一律返回false。

15. 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

16. 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

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。

17. Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。

JavaScript能够准确表示的整数范围在-2^53到2^53之间（不含两个端点）

18. 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.sign()**

Math.sign方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

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。

19. Math.imul方法返回两个数以32位带符号整数形式相乘的结果，返回的也是一个32位的带符号整数。

Math.imul(2, 4) // 8

Math.imul(-1, 8) // -8

Math.imul(-2, -2) // 4

20Math.hypot方法返回所有参数的平方和的平方根。

Math.hypot(3, 4); // 5

Math.hypot(3, 4, 5); // 7.0710678118654755

Math.hypot(); // 0

21. ES2016 新增了一个指数运算符（\*\*）。

2 \*\* 2 // 4

2 \*\* 3 // 8

指数运算符可以与等号结合，形成一个新的赋值运算符（\*\*=）。

let a = 1.5;

a \*\*= 2;

// 等同于 a = a \* a;

let b = 4;

b \*\*= 3;

// 等同于 b = b \* b \* b;

22ES6 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);

}

log('Hello') // Hello World

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

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

23. 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。

…values就是一个数组，在函数中遍历即可。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。

在函数内部加入'use strict';表明使用严格模式，但如果参数中已经定义了数值，则默认函数内部不启用严格模式

函数的name属性 var f=function(){} f.name//---‘f’

24箭头函数

箭头函数导致this总是指向函数定义生效时所在的对象，本身小作用域

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" });

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。

另外，由于箭头函数没有自己的this，所以当然也就不能用call()、apply()、bind()这些方法去改变this的指向。

25.函数绑定运算符是并排的两个冒号（::），双冒号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即this对象），绑定到右边的函数上面。

foo::bar;

// 等同于

bar.bind(foo);

foo::bar(...arguments);

// 等同于

bar.apply(foo, arguments);

const hasOwnProperty = Object.prototype.hasOwnProperty;

function hasOwn(obj, key) {

return obj::hasOwnProperty(key);

}

26. 尾调用（Tail Call）是函数式编程的一个重要概念，本身非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。

27.正则表达式

* **^** 为匹配输入字符串的开始位置。
* **[0-9]+**匹配多个数字， **[0-9]** 匹配单个数字，**+** 匹配一个或者多个。
* **abc$**匹配字母 **abc** 并以 **abc** 结尾，**$** 为匹配输入字符串的结束位置。

/^[0-9]+abc$/

28. module exports require

通过 module 对象可以访问到当前模块的一些相关信息，但最多的用途是替换当前模块的导出对象。例如模块导出对象默认是一个普通对象，如果想改成一个函数的话，可以使用以下方式。

exports

exports 对象是当前模块的导出对象，用于导出模块公有方法和属性。别的模块通过 require 函数使用当前模块时得到的就是当前模块的 exports 对象。以下例子中导出了一个公有方法。

require

require 函数用于在当前模块中加载和使用别的模块，传入一个模块名，返回一个模块导出对象。模块名可使用相对路径（以./开头），或者是绝对路径（以/或C:之类的盘符开头）。另外，模块名中的.js扩展名可以省略。以下是一个例子。

**prototype 属性使您有能力向对象添加属性和方法。**

**reduce()方法**

reduce()方法接收一个函数callbackfn作为累加器（accumulator），数组中的每个值（从左到右）开始合并，最终为一个值。

var arr = [0,1,2,3,4]; arr.reduce(function (preValue,curValue,index,array) { return preValue + curValue; }); // 10 执行4次

var arr = [0,1,2,3,4]; arr.reduce(function (preValue,curValue,index,array) { return preValue + curValue; }, 5); // 15 执行5次，加上定义的5

prototype内属性、方法是能够共享，能够在不同的实例共同修改和获取

function Person(name){

this.name=name;

}

Person.prototype.share=[];

Person.prototype.printName=function(){

alert(this.name);

}

var person1=new Person('Byron');

var person2=new Person('Frank');

person1.share.push(1);

person2.share.push(2);

console.log(person2.share); //[1,2]

果不其然！实际上当代码读取某个对象的某个属性的时候，都会执行一遍搜索，目标是具有给定名字的属性，搜索首先从对象实例开始，如果在实例中找到该属性则返回，如果没有则查找prototype，如果还是没有找到则继续递归prototype的prototype对象，直到找到为止，如果递归到object仍然没有则返回错误。同样道理如果在实例中定义如prototype同名的属性或函数，则会覆盖prototype的属性或函数。