# ES6

# let和const 命令

## let 命令

### 基本用法

ES5只有全局作用域和函数作用域，ES6引入块级作用域， let const声明的变量

|  |
| --- |
| for(var i=0; i<10; i++) {}  console.log(i); // 10，i是全局变量  ----------------------------------------------------  for (let i = 0; i < 10; i++) {}  console.log(i); // 报错i是块级作用域变量 |

|  |
| --- |
| var a = [];  for (var i = 0; i < 10; i++) {  a[i] = function () {  console.log(i);  };  }  a[6](); // 10, i是全局变量  ----------------------------------------------------  var a = [];  for (let i = 0; i < 10; i++) {  a[i] = function () {  console.log(i);  };  }  a[6](); // 6 |

另外，for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，

而循环体内部是一个单独的子作用域

|  |
| --- |
| for (let i = 0; i < 3; i++) {  let i = 'abc';  console.log(i);  }  输出了 3 次abc,  这表明函数内部的变量i与循环变量i不在同一个作用域，有各自单独的作用域  // abc  // abc  // abc |

总结: 块级作用域，其实就是 匿名函数立即调用

|  |
| --- |
| 封闭空间:(function(){ var a=12; })()  现在:{let a=12;} |

|  |
| --- |
| 案例：  window.onload=function(){  var aBtn=document.getElementsByTagName('input');  for(var i=0; i<aBtn.length; i++){  (function(i){  aBtn[i].onclick=function(){  console.log(i);  }  })(i);  }  };  ----------------------------------------------------  window.onload=function(){  var aBtn=document.getElementsByTagName('input');  for(let i=0; i<aBtn.length; i++){  aBtn[i].onclick=function(){  console.log(i);  }  }  }; |

### 不存在变量提升

|  |
| --- |
| 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  // var 的情况  console.log(foo); // 输出undefined  var foo = 2;  // let 的情况  console.log(bar); // 报错ReferenceError  let bar = 2; |

### 暂时性死区

|  |
| --- |
| 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  总之，在代码块内，使用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”（temporal dead zone，简称 TDZ）  var tmp = 123;  if (true) {  tmp = 'abc'; // ReferenceError  let tmp;  }  ------------------------------------------  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  }  --------------------------------------------  typeof x; // ReferenceError  let x;  作为比较，如果一个变量根本没有被声明，使用typeof反而不会报错  typeof undeclared\_variable // "undefined" |

有些“死区”比较隐蔽，不太容易发现。

|  |
| --- |
| 数x默认值等于另一个参数y，而此时y还没有声明，属于“死区”  function bar(x = y, y = 2) {  return [x, y];  }  bar(); // 报错  --------------------------------------------  另外，下面的代码也会报错，与var的行为不同  var x = x; // 不报错  let x = x; // 报错  // ReferenceError: x is not defined |

### 不允许重复声明

|  |
| --- |
| let不允许在相同作用域内，重复声明同一个变量,也不能在函数内部重新声明参数  // 报错  function func() {  let a = 10;  var a = 1;  }  // 报错  function func() {  let a = 10;  let a = 1;  }  ------------------------------------  function func(arg) {  let arg;  }  func() // 报错  function func(arg) {  {  let arg;  }  }  func() // 不报错 |

## 块级作用域

### ES5只有全局作用域和函数作用域，没有块级作用域

|  |
| --- |
| 第一种场景，内层变量可能会覆盖外层变量  var tmp = new Date();  function f() {  console.log(tmp); // undefined 变量提升  if (false) {  var tmp = 'hello world';  }  }  f(); // undefined  ---------------------------------------  第二种场景，用来计数的循环变量泄露为全局变量  var s = 'hello';  for (var i = 0; i < s.length; i++) {  console.log(s[i]);  }  console.log(i); // 5 泄露成了全局变量 |

### ES6 的块级作用域， let实际上为 JavaScript 新增了块级作用域

|  |
| --- |
| ES6 允许块级作用域的任意嵌套  外层作用域无法读取内层作用域的变量  内层作用域可以定义外层作用域的同名变量  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  // IIFE 写法  (function () {  var tmp = ...;  }());  // 块级作用域写法  {  let tmp = ...;  } |

### 块级作用域与函数声明

|  |
| --- |
| 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  // 情况一  if (true) {  function f() {}  }  // 情况二  try {  function f() {}  } catch(e) {  // ...  }  上面两种函数声明，根据 ES5 的规定都是非法的  但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，  因此上面两种情况实际都能运行，不会报错。 |

|  |
| --- |
| 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。  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(); // I am inside! ES5 中运行,在if内声明的函数f会被提升到函数头部  }()); |

## const 命令

### 基本用法

|  |
| --- |
| 1.const声明一个只读的常量。一旦声明，常量的值就不能改变  2.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.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  4.const声明的常量，也与let一样不可重复声明 |

### 本质

|  |
| --- |
| const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。  对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。  但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，  const只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心 |

### Object.freeze冻结对象

|  |
| --- |
| const foo = Object.freeze({});  // 常规模式时，下面一行不起作用；  // 严格模式时，该行会报错  foo.prop = 123;  -----------------------------------------------------  对象的属性也应该冻结  var constantize = (obj) => {  Object.freeze(obj);  Object.keys(obj).forEach( (key, i) => {  if ( typeof obj[key] === 'object' ) {  constantize( obj[key] );  }  });}; |

### ES6 声明变量的六种方法

|  |
| --- |
| ES5 只有两种声明变量的方法：var命令和function命令。  ES6 除了添加let和const命令，后面章节还会提到，  另外两种声明变量的方法：import命令和class命令。  所以，ES6 一共有 6 种声明变量的方法 |

## 顶层对象的属性

|  |
| --- |
|  |

## global 对象

|  |
| --- |
| 浏览器里面，顶层对象是window，但 Node 和 Web Worker 没有window。  浏览器和 Web Worker 里面，self也指向顶层对象，但是 Node 没有self。  Node 里面，顶层对象是global，但其他环境都不支持。 |

# 变量的解构赋值

## 数组的解构赋值

### 基本用法

|  |
| --- |
| ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构  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 |

### 等号的右边不是数组

|  |
| --- |
| 等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错  // 报错  let [foo] = 1;  let [foo] = false;  let [foo] = NaN;  let [foo] = undefined;  let [foo] = null;  let [foo] = {};  要么转为对象以后不具备 Iterator 接口（前五个表达式），要么本身就不具备 Iterator 接口（最后一个表达式） |

### Set 结构，也可以使用数组的解构赋值

|  |
| --- |
| let [x, y, z] = new Set(['a', 'b', 'c']);  x // "a" |

### 默认值

|  |
| --- |
| let [foo = true] = [];  foo // true  let [x, y = 'b'] = ['a']; // x='a', y='b'  let [x, y = 'b'] = ['a', undefined]; // x='a', y='b' |

|  |
| --- |
| 注意，ES6 内部使用严格相等运算符（===），判断一个位置是否有值。  所以，只有当一个数组成员严格等于undefined，默认值才会生效。  let [x = 1] = [undefined];  x // 1  let [x = 1] = [null];  x // null |

|  |
| --- |
| 如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。  function f() {  console.log('aaa');  }  let [x = f()] = [1];  上面代码中，因为x能取到值，所以函数f根本不会执行 |

|  |
| --- |
| 默认值可以引用解构赋值的其他变量，但该变量必须已经声明  let [x = 1, y = x] = []; // x=1; y=1  let [x = 1, y = x] = [2]; // x=2; y=2  let [x = 1, y = x] = [1, 2]; // x=1; y=2  let [x = y, y = 1] = []; // ReferenceError: y is not defined  x用y做默认值时，y还没有声明 |

## 对象的解构赋值

|  |
| --- |
| 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。 |

### 基本用法

|  |
| --- |
| let { bar, foo } = { foo: "aaa", bar: "bbb" };  foo // "aaa"  bar // "bbb"  let { baz } = { foo: "aaa", bar: "bbb" };  baz // undefined |

### 如果变量名与属性名不一致

|  |
| --- |
| let { foo: baz } = { foo: 'aaa', bar: 'bbb' };  baz // "aaa"  let obj = { first: 'hello', last: 'world' };  let { first: f, last: l } = obj;  f // 'hello'  l // 'world'  这实际上说明，对象的解构赋值是下面形式的简写  let { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };  也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。  真正被赋值的是后者，而不是前者 |

### 解构用于嵌套结构的对象

|  |
| --- |
| let obj = {  p: [  'Hello',  { y: 'World' }  ]};  let { p: [x, { y }] } = obj;  x // "Hello"  y // "World" |

### 默认值

|  |
| --- |
| var {x = 3} = {};  x // 3  var {x, y = 5} = {x: 1};  x // 1 y // 5  var {x: y = 3} = {};  y // 3  var {x: y = 3} = {x: 5};  y // 5  var { message: msg = 'Something went wrong' } = {};  msg // "Something went wrong" |

默认值生效的条件是，对象的属性值严格等于undefined。

|  |
| --- |
| var {x = 3} = {x: undefined};  x // 3  var {x = 3} = {x: null};  x // null |

如果解构失败，变量的值等于undefined。

|  |
| --- |
| let {foo} = {bar: 'baz'};  foo // undefined |

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

|  |
| --- |
| // 错误的写法  let x;  {x} = {x: 1};  // SyntaxError: syntax error  ---------------------------------------------  // 正确的写法  let x;  ({x} = {x: 1}); |

|  |
| --- |
| let { log, sin, cos } = Math;  由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。  let arr = [1, 2, 3];  let {0 : first, [arr.length - 1] : last} = arr;  first // 1  last // 3 |

解构赋值允许等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式

|  |
| --- |
|  |

## 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 |

## 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  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 |

## undefined和null

|  |
| --- |
| 解构赋值的规则是，只要等号右边的值不是对象或数组，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 |

## 函数参数的解构赋值

|  |
| --- |
| function add([x, y]){  return x + y;  }  add([1, 2]); // 3  ----------------------------------------------------  [[1, 2], [3, 4]].map(([a, b]) => a + b);  // [ 3, 7 ] |

### 默认值

|  |
| --- |
| function move({x = 0, y = 0} = {}) {  return [x, y];  }  move({x: 3, y: 8}); // [3, 8]  move({x: 3}); // [3, 0]  move({}); // [0, 0]  move(); // [0, 0]  -----------------------------------------------------  function move({x, y} = { x: 0, y: 0 }) {  return [x, y];  }  move({x: 3, y: 8}); // [3, 8]  move({x: 3}); // [3, undefined]  move({}); // [undefined, undefined]  move(); // [0, 0]  是为函数move的参数指定默认值，而不是为变量x和y指定默认值  ----------------------------------------------------------  undefined就会触发函数参数的默认值  [1, undefined, 3].map((x = 'yes') => x);  // [ 1, 'yes', 3 ] |

## 圆括号问题

|  |
| --- |
|  |

## 用途

### ****交换变量的值****

|  |
| --- |
| let x = 1;  let y = 2;  [x, y] = [y, x]; |

### ****从函数返回多个值****

|  |
| --- |
| // 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(); |

### ****函数参数的定义****

|  |
| --- |
| // 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}); |

### ****提取 JSON 数据****

|  |
| --- |
| let jsonData = {  id: 42,  status: "OK",  data: [867, 5309]};  let { id, status, data: number } = jsonData;  console.log(id, status, number);  // 42, "OK", [867, 5309] |

### ****函数参数的默认值****

|  |
| --- |
| jQuery.ajax = function (url, {  async = true,  beforeSend = function () {},  cache = true,  complete = function () {},  crossDomain = false,  global = true,  // ... more config  } = {}) {  // ... do stuff  }; |

### ****遍历 Map 结构****

任何部署了 Iterator 接口的对象，都可以用for...of循环遍历

|  |
| --- |
| const map = new Map();  map.set('first', 'hello');  map.set('second', 'world');  for (let [key, value] of map) {  console.log(key + " is " + value);  }  // first is hello// second is world |

如果只想获取键名，或者只想获取键值，可以写成下面这样

|  |
| --- |
| // 获取键名  for (let [key] of map) {  // ...  }  // 获取键值  for (let [,value] of map) {  // ...  } |

### ****输入模块的指定方法****

|  |
| --- |
| 加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰  const { SourceMapConsumer, SourceNode } = require("source-map"); |

# 字符串的扩展

|  |
| --- |
|  |

## 字符的 Unicode 表示法

|  |
| --- |
| JavaScript 允许采用\uxxxx形式表示一个字符，其中xxxx表示字符的 Unicode 码点  "\u0061"  // "a"  但是，这种表示法只限于码点在\u0000~\uFFFF之间的字符。  超出这个范围的字符，必须用两个双字节的形式表示。  "\uD842\uDFB7"  // ""  \u20BB7"  // " 7" |

## codePointAt()

|  |
| --- |
|  |

## String.fromCodePoint()

|  |
| --- |
| 用于从码点返回对应字符，但是这个方法不能识别 32 位的 UTF-16 字符（Unicode 编号大于0xFFFF）。 |

## 字符串的遍历器接口

|  |
| --- |
| ES6 为字符串添加了遍历器接口（详见《Iterator》一章），  使得字符串可以被for...of循环遍历  for (let codePoint of 'foo') {  console.log(codePoint)  }  // "f"  // "o"  // "o" |

## normalize()

|  |
| --- |
|  |

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

|  |
| --- |
| 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原字符串的头部。  endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。  let s = 'Hello world!';  s.startsWith('Hello') // true  s.endsWith('!') // true  s.includes('o') // true  ---------------------------------------------  第二个参数，表示开始搜索的位置  let s = 'Hello world!';  s.startsWith('world', 6) // true  s.endsWith('Hello', 5) // true  s.includes('Hello', 6) // false  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 |

## repeat()

|  |
| --- |
| 1.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) // ""  ---------------------------------------  2.数如果是小数，会被取整  'na'.repeat(2.9) // "nana"  ------------------------------------  3.如果repeat的参数是负数或者Infinity，会报错  'na'.repeat(Infinity) // RangeError  'na'.repeat(-1) // RangeError  -------------------------------------  4.但是，如果参数是 0 到-1 之间的小数，则等同于 0，这是因为会先进行取整运算  0 到-1 之间的小数，取整以后等于-0，repeat视同为 0。  'na'.repeat(-0.9) // ""  ----------------------------------------------  5.参数NaN等同于 0  'na'.repeat(NaN) // ""  ---------------------------------------------------  6.如果repeat的参数是字符串，则会先转换成数字  'na'.repeat('na') // ""  'na'.repeat('3') // "nanana" |

## padStart()，padEnd()

|  |
| --- |
| padStart()用于头部补全，padEnd()用于尾部补全  第一个参数是字符串补全生效的最大长度，  第二个参数是用来补全的字符串  'x'.padStart(5, 'ab') // 'ababx''  x'.padStart(4, 'ab') // 'abax'  'x'.padEnd(5, 'ab') // 'xabab'  'x'.padEnd(4, 'ab') // 'xaba'  ------------------------------------------------  1.如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串  'xxx'.padStart(2, 'ab') // 'xxx'  'xxx'.padEnd(2, 'ab') // 'xxx'  ------------------------------------------------------  2.如果用来补全的字符串与原字符串，两者的长度之和超过了最大长度，则会截去超出位数的补全字符串  'abc'.padStart(10, '0123456789')  // '0123456abc'  ------------------------------------------------------  3.如果省略第二个参数，默认使用空格补全长度  'x'.padStart(4) // ' x'  'x'.padEnd(4) // 'x  ------------------------------------------------------  4.padStart()的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串  '1'.padStart(10, '0') // "0000000001"  '12'.padStart(10, '0') // "0000000012"  '123456'.padStart(10, '0') // "0000123456"  ------------------------------------------------------  5.另一个用途是提示字符串格式  '12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"  '09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12" |

## trimStart()，trimEnd()

|  |
| --- |
| trimStart()消除字符串头部的空格，trimEnd()消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。  const s = ' abc ';  s.trim() // "abc"  s.trimStart() // "abc "  s.trimEnd() // " abc"  浏览器还部署了额外的两个方法，  trimLeft()是trimStart()的别名，  trimRight()是trimEnd()的别名 |

## matchAll()

|  |
| --- |
| matchAll方法返回一个正则表达式在当前字符串的所有匹配，详见《正则的扩展》的一章 |

## 模板字符串

|  |
| --- |
| 如果在模板字符串中需要使用反引号，则前面要用反斜杠转义  let greeting = `\`Yo\` World!`; // `Yo` World!  模板字符串之中还能调用函数  `foo ${fn()} bar`  模板字符串甚至还能嵌套  const tmpl = addrs => `  <table>  ${addrs.map(addr => `  <tr><td>${addr.first}</td></tr>  <tr><td>${addr.last}</td></tr>  `).join('')}  </table>  `; |

## 实例：模板编译

|  |
| --- |
|  |

## 标签模板

|  |
| --- |
| 可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。  这被称为“标签模板”功能（tagged template）  alert`123`  // 等同于  alert(123)  ----------------------------------------------  标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数  let a = 5;  let b = 10;  tag`Hello ${ a + b } world ${ a \* b }`;  // 等同于  tag(['Hello ', ' world ', ''], 15, 50); |

## String.raw()

|  |
| --- |
| String.raw方法可以作为处理模板字符串的基本方法，它会将所有变量替换，而且对斜杠进行转义，方便下一步作为字符串来使用。  String.raw`Hi\n${2+3}!`;  // 返回 "Hi\\n5!"  String.raw`Hi\u000A!`;  // 返回 "Hi\\u000A!"  String.raw方法也可以作为正常的函数使用。这时，它的第一个参数，应该是一个具有raw属性的对象，且raw属性的值应该是一个数组  String.raw({ raw: 'test' }, 0, 1, 2);  // 't0e1s2t'  // 等同于String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2); |

## 模板字符串的限制

|  |
| --- |
| 前面提到标签模板里面，可以内嵌其他语言。  但是，模板字符串默认会将字符串转义，导致无法嵌入其他语言。  举例来说，标签模板里面可以嵌入 LaTEX 语言。 |

# ****正则的扩展****

|  |
| --- |
|  |

# 数值的扩展

|  |
| --- |
|  |

## 二进制和八进制表示法

|  |
| --- |
| 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  -----------------------------------------------------  从 ES5 开始，在严格模式之中，**八进制**就不再允许使用前缀0表示，  **ES6 进一步明确，要使用前缀0o表示**  // 非严格模式  (function(){  console.log(0o11 === 011);  })() // true  // 严格模式  (function(){  'use strict';  console.log(0o11 === 011);  })() // Uncaught SyntaxError: Octal literals are not allowed in strict mode.  -----------------------------------------------------  **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 |

## 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），即不是Infinity。  **注意，如果参数类型不是数值，Number.isFinite一律返回false**  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  如果参数类型不是NaN， Number.isNaN一律返回false  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  isFinite(25) // true  isFinite("25") // true  Number.isFinite(25) // true  Number.isFinite("25") // false  isNaN(NaN) // true  isNaN("NaN") // true  Number.isNaN(NaN) // true  Number.isNaN("NaN") // false  Number.isNaN(1) // false |

## 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  // 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 |

## Number.isInteger()

|  |
| --- |
| Number.isInteger()判断一个数值是否为整数  JavaScript 内部，整数和浮点数采用的是同样的储存方法，所以 25 和 25.0 被视为同一个值  Number.isInteger(25) // true  Number.isInteger(25.0) // true  Number.isInteger(25.1) // false  如果参数不是数值，Number.isInteger返回false  Number.isInteger() // false  Number.isInteger(null) // false  Number.isInteger('15') // false  Number.isInteger(true) // false |

## Number.EPSILON

|  |
| --- |
| 表示 1 与大于 1 的最小浮点数之间的差  Number.EPSILON === Math.pow(2, -52)  // true |

## 安全整数和 Number.isSafeInteger()

|  |
| --- |
| 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），  超过这个范围，无法精确表示这个值。  Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER这两个常量，  用来表示这个范围的上下  Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内  let a = 2\*\*53-1;  let b = -(2\*\*53-1);  console.log(2\*\*53-1);  console.log(Number.MAX\_SAFE\_INTEGER);  console.log(-(2\*\*53-1));  console.log(Number.MIN\_SAFE\_INTEGER); |

## Math 对象的扩展

|  |
| --- |
|  |

|  |
| --- |
| 17 个,都是静态方法，只能在 Math 对象上调用。  Math.trunc方法用于去除一个数的小数部分，返回整数部分  对于非数值，Math.trunc内部使用Number方法将其先转为数值  对于空值和无法截取整数的值，返回NaN  --------------------------------------------------------------  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。  Math.cbrt()方法用于计算一个数的立方根。  Math.clz32()方法将参数转为 32 位无符号整数的形式，然后这个 32 位值里面有多少个前导 0  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.fround()方法返回一个数的32位单精度浮点数形式  Math.hypot()方法返回所有参数的平方和的平方根。  Math.expm1(x)返回 ex - 1，即Math.exp(x) - 1。  Math.log1p(x)方法返回1 + x的自然对数，即Math.log(1 + x)。如果x小于-1，返回NaN  Math.log10(x)返回以 10 为底的x的对数。如果x小于 0，则返回 NaN  Math.log2(x)返回以 2 为底的x的对数。如果x小于 0，则返回 NaN   * Math.sinh(x) 返回x的双曲正弦（hyperbolic sine） * Math.cosh(x) 返回x的双曲余弦（hyperbolic cosine） * Math.tanh(x) 返回x的双曲正切（hyperbolic tangent） * Math.asinh(x) 返回x的反双曲正弦（inverse hyperbolic sine） * Math.acosh(x) 返回x的反双曲余弦（inverse hyperbolic cosine） * Math.atanh(x) 返回x的反双曲正切（inverse hyperbolic tangent） |

## 指数运算符

|  |
| --- |
| ES2016 新增了一个指数运算符（\*\*）  2 \*\* 2 // 4  2 \*\* 3 // 8  -------------------------------------------------  这个运算符的一个特点是右结合，而不是常见的左结合。多个指数运算符连用时，  是从最右边开始计算的。  // 相当于 2 \*\* (3 \*\* 2)  2 \*\* 3 \*\* 2  // 512  上面代码中，首先计算的是第二个指数运算符，而不是第一个  -------------------------------------------------  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;  ----------------------------------------------------------------  注意，V8 引擎的指数运算符与Math.pow的实现不相同，对于特别大的运算结果，  两者会有细微的差异。  Math.pow(99, 99)  // 3.697296376497263e+197  99 \*\* 99  // 3.697296376497268e+197  上面代码中，两个运算结果的最后一位有效数字是有差异的 |

# 函数的扩展

|  |
| --- |
|  |

## 函数参数的默认值

### 基本用法

|  |
| --- |
| 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);  }  log('Hello') // Hello World  log('Hello', 'China') // Hello China  log('Hello', '') // Hello World  缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。  就像上面代码的最后一行，参数y等于空字符，结果被改为默认值  为了避免这个问题，通常需要先判断一下参数y是否被赋值，如果没有，再等于默认值  if (typeof y === 'undefined') {  y = 'World';  }  --------------------------------------------------  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);  }  log('Hello') // Hello World  log('Hello', 'China') // Hello China  log('Hello', '') // Hello  --------------------------------------------------------  参数变量是默认声明的，所以不能用let或const再次声明  function foo(x = 5) {  let x = 1; // error  const x = 2; // error  } |

### 与解构赋值默认值结合使用

|  |
| --- |
| // 写法一  function m1({x = 0, y = 0} = {}) {  return [x, y];  }  // 写法二  function m2({x, y} = { x: 0, y: 0 }) {  return [x, y];  }  上面两种写法都对函数的参数设定了默认值，  区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；  写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值  // 函数没有参数的情况  m1() // [0, 0]  m2() // [0, 0]  // x 和 y 都有值的情况  m1({x: 3, y: 8}) // [3, 8]  m2({x: 3, y: 8}) // [3, 8]  // x 有值，y 无值的情况  m1({x: 3}) // [3, 0]  m2({x: 3}) // [3, undefined]  // x 和 y 都无值的情况  m1({}) // [0, 0];  m2({}) // [undefined, undefined]  m1({z: 3}) // [0, 0]  m2({z: 3}) // [undefined, undefined] |

### 参数默认值的位置

|  |
| --- |
| 如果非尾部的参数设置默认值，实际上这个参数是没法省略的  除非显式输入undefined, null，也没有触发默认值 |

### 函数的 length 属性

|  |
| --- |
| 指定了默认值以后，函数的length属性，将返回没有指定默认值的参数个数。  也就是说，指定了默认值后，length属性将失真。  这是因为length属性的含义是，该函数预期传入的参数个数。  某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。  同理，后文的 rest 参数也不会计入length属性。  **如果设置了默认值的参数不是尾参数，那么length属性也不再计入后面的参数了**  (function (a = 0, b, c) {}).length // 0  (function (a, b = 1, c) {}).length // 1 |

### 作用域

|  |
| --- |
| 一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（context）。  等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的  var x = 1;  function f(x, y = x) {  console.log(y);  }  f(2) // 2  上面代码中，参数y的默认值等于变量x。调用函数f时，参数形成一个单独的作用域。  在这个作用域里面，默认值变量x指向第一个参数x，而不是全局变量x，所以输出是2  -------------------------------------------------------  var x = 1;  function foo(x = x) {  // ...  }  foo() // ReferenceError: x is not defined  参数x = x形成一个单独作用域。实际执行的是let x = x，由于暂时性死区的原因，  这行代码会报错”x 未定义“  -------------------------------------------------------  **如果参数的默认值是一个函数，该函数的作用域也遵守这个规则** |

### 应用

|  |
| --- |
| 利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误  function throwIfMissing() {  throw new Error('Missing parameter');}  function foo(mustBeProvided = throwIfMissing()) {  return mustBeProvided;  }  foo()// Error: Missing parameter  ---------------------------------------------------------  另外，可以将参数默认值设为undefined，表明这个参数是可以省略的。  function foo(optional = undefined) { ··· } |

## rest 参数

...扩展运算符、rest运算符

### 用于获取函数的多余参数

|  |
| --- |
| 用于获取函数的多余参数，这样就不需要使用arguments对象了  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();  -------------------------------------------------------------  下面是一个利用 rest 参数改写数组push方法的例子  function push(array, ...items) {  items.forEach(function(item) {  array.push(item);  console.log(item);  });}  var a = [];  push(a, 1, 2, 3) |

### 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

|  |
| --- |
| // 报错  function f(a, ...b, c) {// ...} |

### 函数的length属性，不包括 rest 参数

|  |
| --- |
| (function(a) {}).length // 1  (function(...a) {}).length // 0  (function(a, ...b) {}).length // 1 |

## 严格模式

|  |
| --- |
| 从 ES5 开始，函数内部可以设定为严格模式  function doSomething(a, b) {  'use strict';  // code  }  ES2016 做了一点修改，规定只要函数参数使用了**默认值、解构赋值、或者扩展运算符**，  那么函数内部就不能显式设定为严格模式，否则会报错。  // 报错  function doSomething(a, b = a) {  'use strict';  // code  }  // 报错  const doSomething = function ({a, b}) {  'use strict';  // code  };  // 报错  const doSomething = (...a) => {  'use strict';  // code  };  const obj = {  // 报错  doSomething({a, b}) {  'use strict';  // code  }  };  -------------------------------------------------  两种方法可以规避这种限制。第一种是设定全局性的严格模式，这是合法的。  'use strict';  function doSomething(a, b = a) {  // code  }  第二种是把函数包在一个无参数的立即执行函数里面。  const doSomething = (function () {  'use strict';  return function(value = 42) {  return value;  };  }()); |

## name 属性

|  |
| --- |
| 函数的name属性，返回该函数的函数名  function foo() {}  foo.name // "foo"  --------------------------------------  如果将一个匿名函数赋值给一个变量，ES5 的name属性，会返回空字符串，  而 ES6 的name属性会返回实际的函数名  var f = function () {};  // ES5  f.name // ""  // ES6  f.name // "f"  -----------------------------------------------  如果将一个具名函数赋值给一个变量，  则 ES5 和 ES6 的name属性都返回这个具名函数原本的名字。  const bar = function baz() {};  // ES5  bar.name // "baz"  // ES6  bar.name // "baz"  -----------------------------------------------  Function构造函数返回的函数实例，name属性的值为anonymous。  (new Function).name // "anonymous"  -----------------------------------------------  bind返回的函数，name属性值会加上bound前缀。  function foo() {};  foo.bind({}).name // "bound foo"  (function(){}).bind({}).name // "bound " |

## 箭头函数

### 基本使用

|  |
| --- |
| 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，  否则会报错。  // 报错  let getTempItem = id => { id: id, name: "Temp" };  // 不报错  let getTempItem = id => ({ id: id, name: "Temp" });  下面是一种特殊情况，虽然可以运行，但会得到错误的结果。  let foo = () => { a: 1 };  foo() // undefined  ------------------------------------------------  箭头函数的一个用处是简化回调函数。  // 正常函数写法  [1,2,3].map(function (x) {  return x \* x;  });  // 箭头函数写法  [1,2,3].map(x => x \* x);  ------------------------------------------------  另一个例子是  // 正常函数写法  var result = values.sort(function (a, b) {  return a - b;  });  // 箭头函数写法  var result = values.sort((a, b) => a - b);  ------------------------------------------------  下面是 rest 参数与箭头函数结合的例子。  const numbers = (...nums) => nums;  numbers(1, 2, 3, 4, 5)  // [1,2,3,4,5]  const headAndTail = (head, ...tail) => [head, tail];  headAndTail(1, 2, 3, 4, 5)  // [1,[2,3,4,5]] |

### 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。  上面四点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。 |
| 箭头函数可以让setTimeout里面的this，绑定**定义时**所在的作用域，而不是指向**运行时**所在的作用域 |

### 不适用场合

|  |
| --- |
|  |

### 嵌套的箭头函数

|  |
| --- |
|  |

## 双冒号运算符

|  |
| --- |
| 箭头函数可以绑定this对象，大大减少了显式绑定this对象的写法（call、apply、bind）。  但是，箭头函数并不适用于所有场合，所以现在有一个[提案](https://github.com/zenparsing/es-function-bind)，提出了“函数绑定”（function bind）运算符，用来取代call、apply、bind调用。  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。 |

## 尾调用优化

|  |
| --- |
| 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。  尾调用不一定出现在函数尾部，只要是最后一步操作即可  function f(x) {  if (x > 0) {  return m(x)  }  return n(x);} |

## 函数参数的尾逗号

|  |
| --- |
|  |

## 尾递归

|  |
| --- |
| 函数调用自身，称为递归。如果尾调用自身，就称为尾递归  function factorial(n) {  if (n === 1) return 1;  return n \* factorial(n - 1);}  factorial(5) // 120  如果改写成尾递归，只保留一个调用记录，复杂度 O(1) 。  function factorial(n, total) {  if (n === 1) return total;  return factorial(n - 1, n \* total);}  factorial(5, 1) // 120 |

# 数组的扩展

|  |
| --- |
|  |

## 扩展运算符

### 含义

|  |
| --- |
| 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  [...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;  }  const numbers = [4, 38];  add(...numbers) // 42  -----------------------------------------------  注意，扩展运算符如果放在括号中，JavaScript 引擎就会认为这是函数调用。  如果这时不是函数调用，就会报错。  (...[1, 2])  // Uncaught SyntaxError: Unexpected number  console.log((...[1, 2]))  // Uncaught SyntaxError: Unexpected number  console.log(...[1, 2])  // 1 2  上面前两种情况都会报错，因为扩展运算符所在的括号不是函数调用，  而第三种情况console.log(...[1, 2])就不会报错，因为这时是函数调用 |

### 替代函数的 apply 方法

|  |
| --- |
|  |

### 扩展运算符的应用

### ****复制数组****

### ****合并数组****

### ****与解构赋值结合****

### ****字符串****

### ****实现了 Iterator 接口的对象****

### ****Map 和 Set 结构，Generator 函数****

## 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']  -------------------------------------------  // NodeList对象  let ps = document.querySelectorAll('p');  Array.from(ps).filter(p => {  return p.textContent.length > 100;  });  // arguments对象  function foo() {  var args = Array.from(arguments);  // ...  }  ---------------------------------------------  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'] |

## 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。 |

## 数组实例的 copyWithin()

|  |
| --- |
| 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。 |

## 数组实例的 find() 和 findIndex()

|  |
| --- |
| 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 |

## 数组实例的 fill()

|  |
| --- |
| fill方法使用给定值，填充一个数组。 |

## 数组实例的 entries()，keys() 和 values()

|  |
| --- |
|  |

## 数组实例的 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  [1, 2, 3].includes(2) // true  [1, 2, 3].includes(4) // false  [1, 2, NaN].includes(NaN) // true  ----------------------------------------  [1, 2, 3].includes(3, 3); // false  [1, 2, 3].includes(3, -1); // 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  [NaN].indexOf(NaN)  // -1  [NaN].includes(NaN)  // true |

## 数组实例的 flat()，flatMap()

|  |
| --- |
| Array.prototype.flat()用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数据没有影响。 |

## 数组的空位

|  |
| --- |
| Array构造函数返回的数组都是空位  Array(3) // [, , ,]  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  ----------------------------------------------------------------  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(), reduce(), 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。 * 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。 * 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。 * 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 |

## some封装在一个数组里查找符合条件

|  |
| --- |
| let arr =['apple','banana','orange'];  function findInArray(arr, item){  return arr.some((val, index, arr)=>{  return val==item;  });  }  console.log(findInArray(arr, 'orange2')); |

|  |
| --- |
| // 求数组的和、阶乘  let arr = [1,2,3,4,5,6,7,8,9,10];  let res = arr.reduce((prev, cur, index, arr) =>{  return prev+cur;  });  console.log(res); |

|  |
| --- |
| let arr = [2,2,3];  let res = arr.reduce((prev, cur, index, arr) =>{  return prev \*\* cur;  return Math.pow(prev,cur);  });  console.log(res); |

## 类数组转数组

|  |
| --- |
| let aLi = document.querySelectorAll('ul li');  let arrLi = Array.from(aLi); //ES6方法  let arrLi = [...aLi]; //ES6方法  let arrLi = [].slice.call(aLi); //ES5之前方法 |

# 对象的扩展

|  |
| --- |
|  |

## 属性的简洁表示法

|  |
| --- |
| 对象的**变量和方法**都可以简写，  不要再对象简写里使用箭头函数，否则this指向为该对象定义时的上下文  const foo = 'bar';  const baz = {foo};  baz // {foo: "bar"}  // 等同于  const baz = {foo: foo};  ------------------------------------------------------  const o = {  method() {  return "Hello!";  }};  // 等同于  const o = {  method: function() {  return "Hello!";  }  };  ----------------------------------------------  这种写法用于函数的返回值，将会非常方便  function getPoint() {  const x = 1;  const y = 10;  return {x, y};  }  getPoint()  // {x:1, y:10}  ----------------------------------------------------------  CommonJS 模块输出一组变量，就非常合适使用简洁写法  module.exports = { getItem, setItem, clear };  // 等同于  module.exports = {  getItem: getItem,  setItem: setItem,  clear: clear  };  ------------------------------------------------------------------  属性的赋值器（setter）和取值器（getter），事实上也是采用这种写法。  const cart = {  \_wheels: 4,  get wheels () {  return this.\_wheels;  },  set wheels (value) {  if (value < this.\_wheels) {  throw new Error('数值太小了！');  }  this.\_wheels = value;  }  } |

## 应用

|  |
| --- |
| let x = 10;  let y =20;  function show({x,y}){  console.log(x, y);  }  show({x,y}) |

## 属性名表达式

|  |
| --- |
| JavaScript 定义对象的属性，有两种方法。  // 方法一  obj.foo = true;  // 方法二  obj['a' + 'bc'] = 123;  方法一是直接用标识符作为属性名，  方法二是用**表达式作**为属性名，这时要将表达式放在方括号之内  ----------------------------------------------------------  但是，如果使用**字面量方式定义对象（使用大括号**），  在 ES5 中只能使用方法一（标识符）定义属性。  var obj = {  foo: true,  abc: 123  };  ES6 允许**字面量定义对象时**，  用方法二（表达式）作为对象的属性名，即把表达式放在方括号内  let propKey = 'foo';  let obj = {  [propKey]: true,  ['a' + 'bc']: 123  }  ------------------------------------------------------  表达式还可以用于定义方法名。  let obj = {  ['h' + 'ello']() {  return 'hi';  }  };  obj.hello() // hi  ----------------------------------------------------------  注意，属性名表达式与简洁表示法，不能同时使用，会报错。  // 报错  const foo = 'bar';  const bar = 'abc';  const baz = { [foo] };  // 正确  const foo = 'bar';  const baz = { [foo]: 'abc'};  ----------------------------------------------------------  注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]，  这一点要特别小心。  const keyA = {a: 1};  const keyB = {b: 2};  const myObject = {  [keyA]: 'valueA',  [keyB]: 'valueB'  };  myObject // Object {[object Object]: "valueB"}  上面代码中，[keyA]和[keyB]得到的都是[object Object]，  所以[keyB]会把[keyA]覆盖掉，而myObject最后只有一个[object Object]属性。 |

## 方法的 name 属性

|  |
| --- |
| 函数的name属性，返回函数名。对象方法也是函数，因此也有name属性  如果对象的方法使用了取值函数（getter）和存值函数（setter），  则name属性不是在该方法上面，而是该方法的属性的描述对象的get和set属性上面，  返回值是方法名前加上get和set。  两种特殊情况：bind方法创造的函数，name属性返回bound加上原函数的名字；  Function构造函数创造的函数，name属性返回anonymous  如果对象的方法是一个 Symbol 值，那么name属性返回的是这个 Symbol 值的描述。  const key1 = Symbol('description');  const key2 = Symbol();  let obj = {  [key1]() {},  [key2]() {},  };  obj[key1].name // "[description]"  obj[key2].name // "" |

## 属性的可枚举性和遍历

### 可枚举性

|  |
| --- |
| 对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。Object.getOwnPropertyDescriptor方法可以获取该属性的描述对象。  let obj = { foo: 123 };  Object.getOwnPropertyDescriptor(obj, 'foo')  // {  // value: 123,  // writable: true,  // enumerable: true,  // configurable: true  // }  描述对象的enumerable属性，称为“可枚举性”，如果该属性为false，就表示某些操作会忽略当前属性。  目前，有四个操作会忽略enumerable为false的属性。   * for...in循环：只遍历对象自身的和继承的可枚举的属性。 * Object.keys()：返回对象自身的所有可枚举的属性的键名。 * JSON.stringify()：只串行化对象自身的可枚举的属性。 * Object.assign()： 忽略enumerable为false的属性，只拷贝对象自身的可枚举的属性。 |

## super 关键字

|  |
| --- |
| super，指向当前对象的原型对象  const proto = {  foo: 'hello'  };  const obj = {  foo: 'world',  find() {  return super.foo;  }  };  Object.setPrototypeOf(obj, proto);  obj.find() // "hello" |

## 对象的扩展运算符

### 解构赋值

|  |
| --- |
| let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };  x // 1  y // 2  z // { a: 3, b: 4 }  由于解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象。  let { x, y, ...z } = null; // 运行时错误let { x, y, ...z } = undefined; // 运行时错误  解构赋值必须是最后一个参数，否则会报错。  let { ...x, y, z } = someObject; // 句法错误let { x, ...y, ...z } = someObject; // 句法错误  解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本 |

### 扩展运算符

|  |
| --- |
| 对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。  let z = { a: 3, b: 4 };  let n = { ...z };  n // { a: 3, b: 4 }  由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。  let foo = { ...['a', 'b', 'c'] };  foo  // {0: "a", 1: "b", 2: "c"}  如果扩展运算符后面不是对象，则会自动将其转为对象。  // 等同于 {...Object(1)}  {...1} // {}  上面代码中，扩展运算符后面是整数1，会自动转为数值的包装对象Number{1}。由于该对象没有自身属性，所以返回一个空对象  // 等同于 {...Object(true)}  {...true} // {}  // 等同于 {...Object(undefined)}  {...undefined} // {}  // 等同于 {...Object(null)}  {...null} // {}  但是，如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。  {...'hello'}  // {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}  对象的扩展运算符等同于使用Object.assign()方法。  let aClone = { ...a };  // 等同于let aClone = Object.assign({}, a); |

# 对象的新增方法

## Object.is()

|  |
| --- |
| ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。、  它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0。  JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。  ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。  Object.is就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致  Object.is(NaN, NaN); // true  NaN == NaN // false  NaN === NaN // false  Object.is(+0, -0) // false  +0 == -0 // true  +0 === -0 // true |

## Object.assign()

用途:

1. 复制一个对象

2. 合并参数

|  |
| --- |
| Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。  如果只有一个参数，Object.assign会直接返回该参数。  const obj = {a: 1};  Object.assign(obj) === obj // true  如果该参数不是对象，则会先转成对象，然后返回。  typeof Object.assign(2) // "object"  由于undefined和null无法转成对象，所以如果它们作为参数，就会报错。  Object.assign(undefined) // 报错Object.assign(null) // 报错  Object.assign方法实行的是浅拷贝，而不是深拷贝。  let json = {a:1};  let json2 = {b:2, a:2};  let json3 = {c:3};  let obj = Object.assign({}, json, json2,json3);  console.log(obj);  let arr = ['apple','banana','orange'];  let arr2 = Object.assign([], arr);  arr2.push('tomato');  console.log(arr); // ["apple", "banana", "orange"]  console.log(arr2); // ["apple", "banana", "orange", "tomato"] |

## Object.getOwnPropertyDescriptors()

|  |
| --- |
| ES5 的Object.getOwnPropertyDescriptor()方法会返回某个对象属性的描述对象（descriptor）。ES2017 引入了Object.getOwnPropertyDescriptors()方法，返回指定对象所有自身属性（非继承属性）的描述对象。 |

## \_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()

|  |
| --- |
| Object.setPrototypeOf方法的作用与\_\_proto\_\_相同，用来设置一个对象的prototype对象，返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。  该方法与Object.setPrototypeOf方法配套，用于读取一个对象的原型对象。 |

## Object.keys()，Object.values()，Object.entries()

|  |
| --- |
|  |

## Object.fromEntries()

|  |
| --- |
| Object.fromEntries()方法是Object.entries()的逆操作，用于将一个键值对数组转为对象。  Object.fromEntries([  ['foo', 'bar'],  ['baz', 42]])  // { foo: "bar", baz: 42 }  该方法的主要目的，是将键值对的数据结构还原为对象，因此特别适合将 Map 结构转为对象。  // 例一const entries = new Map([  ['foo', 'bar'],  ['baz', 42]]);  Object.fromEntries(entries)  // { foo: "bar", baz: 42 }  // 例二const map = new Map().set('foo', true).set('bar', false);  Object.fromEntries(map)  // { foo: true, bar: false }  该方法的一个用处是配合URLSearchParams对象，将查询字符串转为对象。  Object.fromEntries(new URLSearchParams('foo=bar&baz=qux'))  // { foo: "bar", baz: "qux" } |

# Symbol

## 概述

## 作用

## 作为属性名的 Symbol

## 实例：消除魔术字符串

## 属性名的遍历

## Symbol.for()，Symbol.keyFor()

## 实例：模块的 Singleton 模式

## 内置的 Symbol 值

# Set 和 Map 数据结构

## Set

## WeakSet

## Map

## WeakMap

## symbol对象作用

|  |
| --- |
|  |

## Promise的使用场景，'Promise'它所解决的问题以及现在对于异步操作的解决方案

|  |
| --- |
| Promise的使用场景：ajax请求，回调函数，复杂操作判断。  Promise是ES6为了解决异步编程所诞生的。  异步操作解决方案：Promise、Generator、定时器（不知道算不算）、还有ES7的async |

## es6 promise ajax

|  |
| --- |
| 定义  const myHttpClient = url => {  return new Promise((resolve, reject) => {  let client = new XMLHttpRequest();  client.open("GET", url);  client.onreadystatechange = handler;  client.responseType = "json";  client.setRequestHeader("Accept", "application/json");  client.send();  function handler() {  if (this.readyState !== 4) {  return;  }  if (this.status === 200) {  resolve(this.response);  } else {  reject(new Error(this.statusText));  }  }  });  };  使用  myHttpClient('https://www.baidu.com').then(res => {  console.log(res);  }).catch(error => {  console.log(error);  }); |

## ES6中的Promise，怎样实现依次异步回调

|  |
| --- |
|  |

## Promise对象，嵌套try catch，内层嵌套错误捕获

|  |
| --- |
|  |

## async/await用途

|  |
| --- |
| 用途：让 promise 的异步变成同步运行成了可能,await 可以等到 promise 执行完毕。  作用：异步代码的新方式  async *function* timeout(*ms*) {  await new Promise((*resolve*) *=>* {  setTimeout(resolve, ms);  });  }  async *function* asyncPrint(*value*, *ms*) {  await timeout(ms);  *console*.log(value);  }  asyncPrint('hello world', 50); // 50毫秒后，才输出hello world  async函数完全可以看作多个异步操作，包装成的一个 *Promise* 对象，而await命令就是内部then命令的语法糖  ==========================================================================  promise示例：  *const* makeRequest = () *=>* {  return getJSON().then(*data* *=>* {  if (data.needsAnotherRequest) {  return makeAnotherRequest(data).then(*moreData* *=>* {  *console*.log(moreData)  return moreData  })  }else {  *console*.log(data)  return data  }  })  }  ==========================================================================  async/await示例：  *const* makeRequest = async () *=>* {  *const* data = await getJSON()  if (data.needsAnotherRequest) {  *const* moreData = await makeAnotherRequest(data);  *console*.log(moreData)  return moreData  } else {  *console*.log(data)  return data  }  }  函数前多了aync关键字，await关键字只能用在aync定义的函数内。  async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。  (示例中reosolve值就是字符串"done") |

## generator（异步编程、yield、next()、await 、async）

|  |
| --- |
|  |

## ES6中class 的new实例和es5的new实例区别

|  |
| --- |
|  |

## ES5的继承和ES6的继承有什么区别

|  |
| --- |
| ES5的继承时通过prototype或构造函数机制来实现。ES5的继承实质上是先创建子类的实例对象，然后再将父类的方法添加到this上（Parent.apply(this)）。  ES6的继承机制完全不同，实质上是先创建父类的实例对象this（所以必须先调用父类的super()方法），然后再用子类的构造函数修改this。  具体的：ES6通过class关键字定义类，里面有构造方法，类之间通过extends关键字实现继承。子类必须在constructor方法中调用super方法，否则新建实例报错。因为子类没有自己的this对象，而是继承了父类的this对象，然后对其进行加工。如果不调用super方法，子类得不到this对象。  ps：super关键字指代父类的实例，即父类的this对象。在子类构造函数中，调用super后，才可使用this关键字，否则报错。 |

## ES6中的class和ES5的类有什么区别

|  |
| --- |
| 1.ES6 class 内部所有定义的方法都是不可枚举的;  2.ES6 class 必须使用 new 调用;  3.ES6 class 不存在变量提升;  4.ES6 class 默认即是严格模式;  5.ES6 class 子类必须在父类的构造函数中调用super()，这样才有this对象;ES5中类继承的关系是相反的，先有子类的this，然后用父类的方法应用在this上。 |

## ES6 class与ES5 function区别及联系

|  |
| --- |
|  |

## ES6有哪些新的东西

|  |
| --- |
|  |

## ES8 新特性

|  |
| --- |
| 字符串填充  str.padStart(targetLength[, padString])  str.padEnd(targetLength[, padString])复制代码  values和entries函数  *Object*.values(obj)  *Object*.entries(obj)复制代码  getOwnPropertyDescriptors函数  *Object*.getOwnPropertyDescriptors(obj)复制代码  函数参数逗号结尾  *function* es8(*var1*, *var2*, *var3*, ) { } 复制代码  异步函数  由async关键词定义的函数声明了一个可以异步执行的函数，返回一个AsyncFunction类型的对象。  fucntion fetchTextByPromise() {  return new Promise(*resolve* *=>* {  setTimeout(() *=>* {  resolve('es8');  }, 2000);  });  }  async *function* sayHello() {  *const* externalFetchedText = await fetchTextByPromise();  *console*.log(`Hello, ${externalFetchedText}`);  }  sayHello(); |

## 箭头函数

|  |
| --- |
| 语法比函数表达式更短，并且不绑定自己的this，arguments，super或 new.target。这些函数表达式最适合用于非方法函数，并且它们不能用作构造函数 |

## 箭头函数的this指向

|  |
| --- |
| 箭头函数所改变的并非把`this`局部化，而是完全不把`this`绑定到里面去;  就是`this`是取自外部的上下级作用域(但是又不是常规 function的语法糖)  因为箭头函数里并不支持`var self = this`或者`.bind(this)`这样的写法。  ===================  箭头函数的this指向谁?  肯定很多小伙伴会说指向局部方法内!!答案是错误的...  箭头函数所改变的并非把 this 局部化，而是完全不把 this 绑定到里面去;  就是 this 是取自外部的上下级作用域(但是又不是常规 function的语法糖)..  因为箭头函数里并不支持 var self = this 或者 .bind(this) 这样的写法。 |
|  |

## Set 和 Map 数据结构

|  |
| --- |
| 1.ES6 提供了新的数据结构 Set 它类似于数组，但是成员的值都是唯一的，没有重复的值。  2.ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。  也就是说，Object 结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。 |

## WeakMap 和 Map 的区别

|  |
| --- |
| 1.WeakMap 结构与 Map 结构基本类似，唯一的区别是它只接受对象作为键名（ null 除外），不接受其他类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。  2. WeakMap 最大的好处是可以避免内存泄漏。一个仅被 WeakMap 作为 key 而引用的对象，会被垃圾回收器回收掉。  3.WeakMap 拥有和 Map 类似的 set(key, value) 、get(key)、has(key)、delete(key) 没有任何与迭代有关的属性和方法。 |

## ES6模块机制

|  |
| --- |
|  |

## 装饰器原理

|  |
| --- |
| 语法糖，实则调用Object.defineProperty，可以添加、修改对象属性 |

## map和set优缺点

|  |
| --- |
| *1.Set* 数据结构类似数组，但所有成员的值唯一  *let* a = new Set();  [1,2,2,1,3,4,5,4,5].forEach(*x=>*a.add(x));  for(*let* k of a){  *console*.log(k)  };  // 1 2 3 4 5  ==================================================================  方法  add(value)：添加某个值，返回 *Set* 结构本身。  delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。  has(value)：返回一个布尔值，表示该值是否为Set的成员。  clear()：清除所有成员，没有返回值。  *let* a = new Set();  a.add(1).add(2); // a => Set(2) {1, 2}  a.has(2); // true  a.has(3); // false  a.delete(2); // true a => Set(1) {1}  a.clear(); // a => Set(0) {}  ==================================================================  基本使用  *let* a = new Set([1,2,3,3,4]);  [...a]; // [1,2,3,4]  a.size; // 4  // 数组去重  [...new Set([1,2,3,4,4,4])];// [1,2,3,4]  ==================================================================  2. *Map* 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现  *let* a = new Map();  *let* b = {name: 'leo' };  a.set(b,'my name'); // 添加值  a.get(b); // 获取值  a.size; // 获取总数  a.has(b); // 查询是否存在  a.delete(b); // 删除一个值  a.clear(); // 清空所有成员 无返回  ==================================================================  方法  keys()：返回键名的遍历器。  values()：返回键值的遍历器。  entries()：返回所有成员的遍历器。  forEach()：遍历 *Map* 的所有成员。  *let* a = new Map([ ['name', 'leo'], ['age', 18] ])  for (*let* i of a.keys()) {  *console*.log(i)  };  //name  //age  for (*let* i of a.values()) {  *console*.log(i)  };  //leo  //18  for (*let* i of a.entries()) {  *console*.log(i)  };  //["name", "leo"]  a.forEach((*v*, *k*, *m*) *=>* {  *console*.log(`key:${k},value:${v},map:${m}`)  })  //["age", 18]  ==================================================================  基本使用  1.传入数组作为参数，指定键值对的数组。  *let* a = new Map([ ['name','wzx'], ['age',23]])  2.如果对同一个键多次赋值，后面的值将覆盖前面的值。  *let* a = new Map();  a.set(1,'aaa').set(1,'bbb');  a.get(1); // 'bbb'  3.如果读取一个未知的键，则返回undefine  new Map().get('asdsad'); // undefined  4.同样的值的两个实例，在 *Map* 结构中被视为两个键  *let* a = new Map();  *let* a1 = ['aaa'];  *let* a2 = ['aaa'];  a.set(a1,111).set(a2,222);  a.get(a1); // 111  a.get(a2); // 222 |

## 箭头函数this指向问题、拓展运算符

|  |
| --- |
| 箭头函数this默认指向在定义它时,它所处的对象,而不是执行时的对象,定义它的时候,可能环境是window（即继承父级的this）。  ==========================  this的指向:  1、当函数作为对象的方法被调用时，this就会指向该对象。  2、作为普通函数，this指向window。  3、构造器调用，this指向返回的这个对象。  4、箭头函数 箭头函数的this绑定看的是this所在函数定义在哪个对象下，就绑定哪个对象。如果有嵌套的情况，则this绑定到最近的一层对象上  ============================  箭头函数this的原理：  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  ====================  怎么改变this的指向呢?  1.使用es6的箭头函数；  2.在函数内部使用that = this；  3.使用apply，call，bind；  4.*n*ew实例化一个对象  ====================  *var* a=11;  *function* test2(){  this.a=22;  *let* b=()*=>*{*console*.log(this.a)}  b();  }  *var* x=new test2();  //输出22  定义时绑定 |