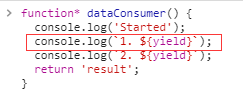
闭包

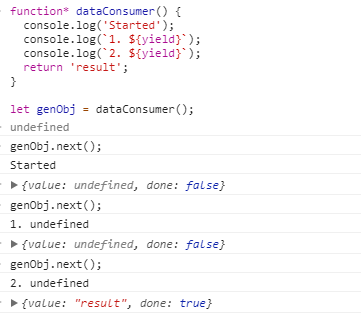
函数自执行

变量先声明再使用（未声明的变量）

apply/call/bind/prototype

var定义变量存在变量提升，脚本运行时，变量已存在，但是没有值

常量中混变量写法： 用 ${变量} ，最外围是 `



一、let命令、const命令

1、let声明的变量、const声明的常量只在块级作用域有效

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，使立即执行函数表达式（IIFE）不再必要

const声明一个只读的常量。一旦声明，常量的值就不能改变。

{

let a = 10;

var b = 1;

}

a // ReferenceError: a is not defined.

b // 1

// IIFE 写法

(function () {

var tmp = ...;

...

}());

// 块级作用域写法

{

let tmp = ...;

...

}

2、暂时性死区

用let、const声明的变量在其作用域内存在暂时性死区，在变量生命赋值语句前调用这个变量会报错

// 函数参数默认声明变量，y未先声明再使用报错

function bar(x = y, y = 2) {

return [x, y];

}

bar(); // 报错

// 函数参数默认声明变量，x先声明再使用不报错

function bar(x = 2, y = x) {

return [x, y];

}

bar(); // [2, 2]

// 不报错

var x = x;

// 报错

let x = x;

// ReferenceError: x is not defined

3、不存在变量提升

// 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;

4、不允许在相同作用域内，重复声明同一个变量。

// 报错

function () {

let a = 10;

var a = 1;

}

// 报错

function () {

let a = 10;

let a = 1;

}

5、块级作用域与函数声明

---只对 ES6 的浏览器实现有效---

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。

---其他环境的实现---

块级作用域的函数声明当作let处理

\*\*\*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\*\*\*

// 函数声明语句

{

let a = 'secret';

function f() {

return a;

}

}

// 函数表达式

{

let a = 'secret';

let f = function () {

return a;

};

}

ES6 的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。

// 不报错

'use strict';

if (true) {

function f() {}

}

// 报错

'use strict';

if (true)

function f() {}

6、do表达式

let x = do {

let t = f();

t \* t + 1;

};

//变量x会得到整个块级作用域的返回值

7、本质

常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

const foo = {};

// 为 foo 添加一个属性，可以成功

foo.prop = 123;

foo.prop // 123

// 将 foo 指向另一个对象，就会报错

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

常量a是一个数组，这个数组本身是可写的，但是如果将另一个数组赋值给a，就会报错。

const a = [];

a.push('Hello'); // 可执行

a.length = 0; // 可执行

a = ['Dave']; // 报错

8、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

9、顶层对象属性

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。

var a = 1;

// 如果在Node的REPL环境，可以写成global.a

// 或者采用通用方法，写成this.a

window.a // 1

let b = 1;

window.b // undefined

二、变量的解构赋值

1、数组的解构赋值

### 01 基本用法：模式匹配

//完全解构

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

foo // 1

bar // 2

baz // 3

//不完全解构

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]

//解构不成功, 变量的值就等于undefined

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

x // "a"

y // undefined

z // []

// 报错

let [foo] = 1;

let [foo] = false;

let [foo] = NaN;

let [foo] = undefined;

let [foo] = null;

let [foo] = {};

### 02 默认值

let [x = 1] = [undefined];

x // 1

let [x = 1] = [null];

x // null

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];

2、对象的解构赋值

### 01 基本用法

//变量有同名属性写法

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

foo // "aaa"

bar // "bbb"

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

baz // undefined

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

baz // "aaa"

//变量没有同名属性写法

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

// 正确的写法

let x;

({x} = {x: 1});

### 02 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"

var {x = 3} = {x: undefined};

x // 3

var {x = 3} = {x: null};

x // null

3、数值和布尔值的解构赋值

let {toString: s} = 123;

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

let {toString: s} = true;

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

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

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

4、用途

### 01 交换变量的值

### 02 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();

### 03 函数参数的定义

### 04 提取json数据

### 05 函数参数默认值

### 06 遍历Map结构

### 07 输入模块的指定方法

const { SourceMapConsumer, SourceNode } = require("source-map");

三、字符串的扩展

1、字符的 Unicode 表示法

//ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符

"\u{20BB7}"

// "𠮷"

"\u{41}\u{42}\u{43}"

// "ABC"

let hello = 123;

hell\u{6F} // 123

'\u{1F680}' === '\uD83D\uDE80'

// true

2、codePointAt()

var s = "𠮷";

s.length // 2

s.charAt(0) // ''

s.charAt(1) // ''

s.charCodeAt(0) // 55362

s.charCodeAt(1) // 57271

var s = '𠮷a';

s.codePointAt(0) // 134071

s.codePointAt(1) // 57271

s.codePointAt(2) // 97

3、String.fromCodePoint()

String.fromCodePoint(0x20BB7)

// "𠮷"

String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'

// true

4、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"

var text = String.fromCodePoint(0x20BB7);

for (let i = 0; i < text.length; i++) {

console.log(text[i]);

}

// " "

// " "

for (let i of text) {

console.log(i);

}

// "𠮷"

5、at()

ES5对字符串对象提供charAt方法，返回字符串给定位置的字符。

ES6对字符串对象提供at方法，返回字符串给定位置的字符。可以识别Unicode编号大于0xFFFF的字符，返回正确的字符。

6、normalize()

7、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

这三个方法都支持第二个参数，表示开始搜索的位置。

var s = 'Hello world!';

//第二个参数n表示开始搜索的位置

s.startsWith('world', 6) // true 从第n个位置直到字符串结束

s.endsWith('Hello', 5) // true 前n个字符

s.includes('Hello', 6) // false 从第n个位置直到字符串结束

8、repeat(n)

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

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

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

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"

9、padStart()，padEnd()

padStart和padEnd一共接受两个参数，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。

padStart(n,str)用于头部补全, padEnd(n,str)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'

原字符串的长度，等于或大于指定的最小长度，则返回原字符串。

'xxx'.padStart(2, 'ab') // 'xxx'

'xxx'.padEnd(2, 'ab') // 'xxx'

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 '

10、模板字符串

$('#result').append(`

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

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

are on sale!

`);

01用反引号（’）标识

02在模板字符串中需要使用反引号，则前面要用反斜杠转义。

var greeting = `\`Yo\` World!`;

03 使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中, 可以使用trim方法消除空格和缩进。

04 模板字符串中嵌入变量，需要将变量名写在${}之中，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，调用函数。

11、实例：模板编译

var template = `

<ul>

<% for(var i=0; i < data.supplies.length; i++) { %>

<li><%= data.supplies[i] %></li>

<% } %>

</ul>

`;

四、正则的扩展

1、RegExp构造函数

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

// 等价于

var regex = /xyz/i;

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

// 等价于

var regex = /xyz/i;

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

// "i"

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()。

match(regexp) : 根据pattern进行正则匹配,如果匹配到,返回匹配结果,如匹配不到返回null

var str="1 plus 2 equal 3"

console.log(str.match(/\d+/g))

// 有/g全局匹配标志返回匹配全部数组

["1", "2", "3"]

console.log(str.match(/\d+/))

// 没有/g全局匹配标志返回首个匹配元素，和index属性（匹配元素在字符串中位置），input属性（StringObject对象的引用）

["1", index: 0, input: "1 plus 2 equal 3"]

search(regexp)：检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串。返回stringObject 中第一个与 regexp 相匹配的子串的起始位置。没有找到任何匹配的子串，则返回 -1。

var str = "This isn't me";

var reg = /is/i; // /i表示忽略大小写

var ret = str.search(reg);

console.log(ret);//2

replace(regexp，replacement)：返回一个新的字符串，是用 replacement 替换了 regexp 的第一次匹配或所有匹配之后得到的。





split(separator,howmany): 一个字符串数组。该数组是通过在 separator 指定的边界处将字符串 stringObject 分割成子串创建的。返回的数组中的字串不包括 separator 自身。

test(string):测试string是否包含有匹配结果，包含返回true，不包含返回false。

var str = "bbs.byr.cn";

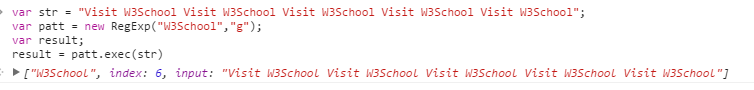
var reg = /b/;

var ret = reg.test(str);

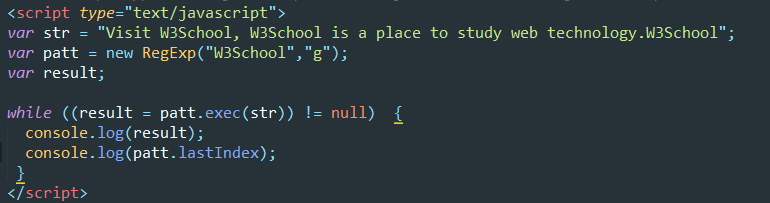
alert(ret); //true

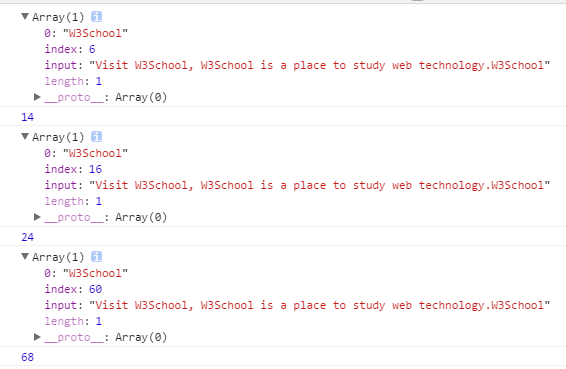
exec(string): 进行非全局匹配与match非全局匹配结果相同，进行全局匹配，在匹配后，它将把 RegExpObject 的 lastIndex 属性设置为匹配文本的最后一个字符的下一个位置。这就是说，您可以通过反复调用 exec() 方法来遍历字符串中的所有匹配文本。当 exec() 再也找不到匹配的文本时，它将返回 null，并把 lastIndex 属性重置为 0。

非全局匹配：



全局匹配：





五、数值的扩展

1、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

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

2、Number.isFinite(), Number.isNaN()

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

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。

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

3、Number.parseInt(), Number.parseFloat()

Number.parseInt === parseInt // true

Number.parseFloat === parseFloat // true

// 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

4、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

5、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'

6、安全整数和Number.isSafeInteger()

7、Math对象的扩展

### 01 Math.trunc()

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.trunc(NaN); // NaN

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

Math.trunc(); // NaN

### 02 Math.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。
* Math.sign(-5) // -1
* Math.sign(5) // +1
* Math.sign(0) // +0
* Math.sign(-0) // -0
* Math.sign(NaN) // NaN
* Math.sign('foo'); // NaN
* Math.sign(); // NaN

### 03 Math.cbrt()

Math.cbrt方法用于计算一个数的立方根。

### 04 Math.clz32()

JavaScript的整数使用32位二进制形式表示，Math.clz32方法返回一个数的32位无符号整数形式有多少个前导0。

Math.clz32(0) // 32

Math.clz32(1) // 31

Math.clz32(1000) // 22

Math.clz32(0b01000000000000000000000000000000) // 1

Math.clz32(0b00100000000000000000000000000000) // 2

### 05 Math.imul()

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。

### 06 Math.fround()

Math.fround方法返回一个数的单精度浮点数形式。

### 07 Math.hypot()

Math.hypot方法返回所有参数的平方和的平方根。

### 08 对数方法

#### **（1） Math.expm1()**

Math.expm1(x)返回ex - 1，即Math.exp(x) - 1。

#### **（2）Math.log1p()**

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。

#### **（3）Math.log10()**

Math.log10(x)返回以10为底的x的对数。如果x小于0，则返回NaN。

#### **（4）Math.log2()**

Math.log2(x)返回以2为底的x的对数。如果x小于0，则返回NaN。

### 09 三件函数方法

ES6新增了6个三角函数方法。

* 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）

### 10 Math.signbit()

* 如果参数是NaN，返回false
* 如果参数是-0，返回true
* 如果参数是负值，返回true
* 其他情况返回false

### 11 指数运算符

ES2016 新增了一个指数运算符（\*\*）。

let a = 1.5;

a \*\*= 2;

// 等同于 a = a \* a;

let b = 4;

b \*\*= 3;

// 等同于 b = b \* b \* b;

Math.pow(99, 99)

// 3.697296376497263e+197

99 \*\* 99

// 3.697296376497268e+197

六、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）。

类似数组的对象：

let arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c',

length: 3

};

// DOM操作返回的NodeList集合、函数内部的arguments对象

可遍历（iterable）的对象：

Array.from('hello')

// ['h', 'e', 'l', 'l', 'o']

let namesSet = new Set(['a', 'b'])

Array.from(namesSet) // ['a', 'b']

//字符串和Set结构都具有Iterator接口

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')]

2、Array.of()

Array.of方法用于将一组值，转换为数组。

3、数组实例的copyWithin(target, start, end)

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。

4、数组实例的find()和findIndex()

find(function(value, index, arr) {},obj)

//返回第一个找到的值，没有返回undefined

findIndex(function(value, index, arr) {},obj)

//返回第一个找到的值的位置的索引，没有返回-1

//obj为回调函数内部使用this绑定的对象

5、数组实例的entries()，keys()和values()

使用for...of循环，或者手动调用遍历器对象的next方法，进行遍历。

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 "

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']

6、数组实例的includes()

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，第二个参数表示从哪个位置开始检索

7、数组的空位

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

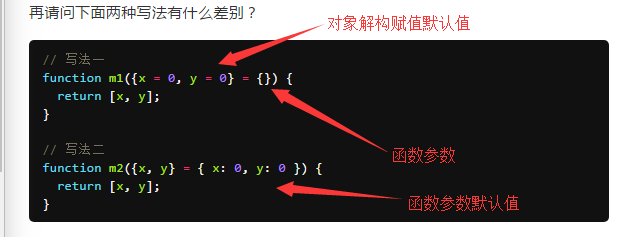
* forEach(), filter(), every() 和some()都会跳过空位。
* map()会跳过空位，但会保留这个值
* join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。

七、函数的扩展

1、函数参数的默认值

2、与解构赋值默认值结合使用

无参时调用参数默认值，参数是对象且为空时，调用对象解构的默认值



3、参数默认值的位置

非尾参数设置默认值，实参不可省略，需传入undefined默认参数才可生效

4、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，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

5、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。

var x = 1;

function foo(x, y = function() { x = 2; }) {

var x = 3;

y();

console.log(x);

}

foo() // 3

x // 1

var x = 1;

function foo(x, y = function() { x = 2; }) {

x = 3;

y();

console.log(x);

}

foo() // 2

x // 1

6、rest参数

rest参数必须是最后一个参数，rest参数不计入length属性

function push(array, ...items) {

items.forEach(function(item) {

array.push(item);

console.log(item);

});

}

var a = [];

push(a, 1, 2, 3)

push.length // 1

7、扩展运算符

将一个数组转为用逗号分隔的参数序列, 主要用于函数调用, 替代数组的apply方法

// ES5的写法

Math.max.apply(null, [14, 3, 77])

// ES6的写法

Math.max(...[14, 3, 77])

// 等同于

Math.max(14, 3, 77);

8、扩展运算符的应用

### 01合并数组

// ES5

[1, 2].concat(more)

// ES6

[1, 2, ...more]

var arr1 = ['a', 'b'];

var arr2 = ['c'];

var arr3 = ['d', 'e'];

// ES5的合并数组

arr1.concat(arr2, arr3);

// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组

[...arr1, ...arr2, ...arr3]

// [ 'a', 'b', 'c', 'd', 'e' ]

### 02与解构赋值结合

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

rest // [2, 3, 4, 5]

const [first, ...rest] = [];

first // undefined

rest // []

const [first, ...rest] = ["foo"];

first // "foo"

rest // []

### 03函数的返回值

JavaScript的函数只能返回一个值，如果需要返回多个值，只能返回数组或对象。扩展运算符提供了解决这个问题的一种变通方法。

var dateFields = readDateFields(database);

var d = new Date(...dateFields);

上面代码从数据库取出一行数据，通过扩展运算符，直接将其传入构造函数Date。

### 04 字符串

扩展运算符还可以将字符串转为真正的数组。

[...'hello']

// [ "h", "e", "l", "l", "o" ]

### 05实现了Iterator接口的对象

任何Iterator接口的对象，都可以用扩展运算符转为真正的数组。

var nodeList = document.querySelectorAll('div');

var array = [...nodeList];

那些没有部署Iterator接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。可以改为使用Array.from方法将类数组对象转为真正的数组

### 06 Map和Set结构，Generator函数

只要具有Iterator接口的对象，都可以使用扩展运算符

9、严格模式

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。

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;

};

}());

10、name 属性

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 //

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 // "bou

11、箭头函数

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。

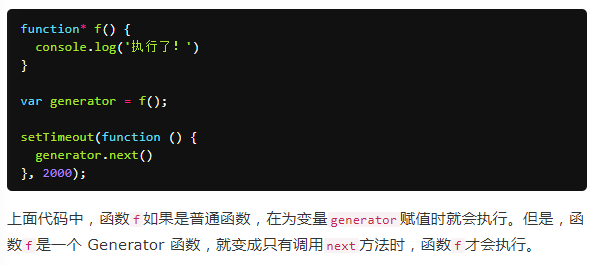
12、绑定 this

13、尾调用优化

14、尾递归优化

15、函数参数的尾逗号

## 9、函数立即执行



八、对象的扩展

1、属性的简洁表示法

2、属性名表达式

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"}

3、方法的 name 属性

有两种特殊情况：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。

4、Object.is()

5、Object.assign

九、Symbol

async 函数有多种使用形式：



# 十、Class 的基本语法

## 1、简介

* JS中生成实例对象是通过构造函数
* function Point(x, y) {
* this.x = x;
* this.y = y;
* }
* Point.prototype.toString = function () {
* return '(' + this.x + ', ' + this.y + ')';
* };
* var p = new Point(1, 2);
* ES6的class改写
* //定义类
* class Point {
* constructor(x, y) {
* this.x = x;
* this.y = y;
* }
* toString() {
* return '(' + this.x + ', ' + this.y + ')';
* }
* }

类的所有方法都定义在类的prototype属性上面。

class Point {

constructor() {

// ...

}

toString() {

// ...

}

toValue() {

// ...

}

}

// 等同于

Point.prototype = {

constructor() {},

toString() {},

toValue() {},

};

在类的实例上面调用方法，其实就是调用原型上的方法。

class B {}

let b = new B();

b.constructor === B.prototype.constructor // true

由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。

class Point {

constructor(){

// ...

}

}

Object.assign(Point.prototype, {

toString(){},

toValue(){}

});

prototype对象的constructor属性，直接指向“类”的本身，这与 ES5 的行为是一致的。

Point.prototype.constructor === Point // true

类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

class Point {

constructor(x, y) {

// ...

}

toString() {

// ...

}

}

Object.keys(Point.prototype)

// []

Object.getOwnPropertyNames(Point.prototype)

// ["constructor","toString"]

采用 ES5 的写法，toString方法就是可枚举的。

var Point = function (x, y) {

// ...

};

Point.prototype.toString = function() {

// ...

};

Object.keys(Point.prototype)

// ["toString"]

Object.getOwnPropertyNames(Point.prototype)

// ["constructor","toString"]

类的属性名，可以采用表达式。

let methodName = 'getArea';

class Square {

constructor(length) {

// ...

}

[methodName]() {

// ...

}

}

## 2、严格模式

类和模块的内部，默认就是严格模式，所以不需要使用use strict指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。

## 3、constructor

定义了一个空的类Point，JavaScript 引擎会自动为它添加一个空的constructor方法。

class Point {

}

// 等同于

class Point {

constructor() {}

}

constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。

class Foo {

constructor() {

return Object.create(null);

}

}

new Foo() instanceof Foo

// false

## 4、类的实例对象

与 ES5 一样，实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

//定义类

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

toString() {

return '(' + this.x + ', ' + this.y + ')';

}

}

var point = new Point(2, 3);

point.toString() // (2, 3)

point.hasOwnProperty('x') // true

point.hasOwnProperty('y') // true

point.hasOwnProperty('toString') // false

point.\_\_proto\_\_.hasOwnProperty('toString') // true

上面代码中，x和y都是实例对象point自身的属性（因为定义在this变量上），所以hasOwnProperty方法返回true，而toString是原型对象的属性（因为定义在Point类上），所以hasOwnProperty方法返回false。这些都与 ES5 的行为保持一致。

与 ES5 一样，类的所有实例共享一个原型对象。

var p1 = new Point(2,3);

var p2 = new Point(3,2);

p1.\_\_proto\_\_ === p2.\_\_proto\_\_

//true

\_\_proto\_\_ 并不是语言本身的特性，这是各大厂商具体实现时添加的私有属性，虽然目前很多现代浏览器的JS引擎中都提供了这个私有属性，但依旧不建议在生产中使用该属性，避免对环境产生依赖。生产环境中，我们可以使用 Object.getPrototypeOf 方法来获取实例对象的原型，然后再来为原型添加方法/属性。

var p1 = new Point(2,3);

var p2 = new Point(3,2);

p1.\_\_proto\_\_.printName = function () { return 'Oops' };

p1.printName() // "Oops"

p2.printName() // "Oops"

var p3 = new Point(4,2);

p3.printName() // "Oops"

上面代码在p1的原型上添加了一个printName方法，由于p1的原型就是p2的原型，因此p2也可以调用这个方法。而且，此后新建的实例p3也可以调用这个方法。这意味着，使用实例的\_\_proto\_\_属性改写原型，必须相当谨慎，不推荐使用，因为这会改变“类”的原始定义，影响到所有实例。

## 5、class表达式

与函数一样，类也可以使用表达式的形式定义。

const MyClass = class Me {

getClassName() {

return Me.name;

}

};

上面代码使用表达式定义了一个类。需要注意的是，这个类的名字是MyClass而不是Me，Me只在 Class 的内部代码可用，指代当前类。

let inst = new MyClass();

inst.getClassName() // Me

Me.name // ReferenceError: Me is not defined

上面代码表示，Me只在 Class 内部有定义。

如果类的内部没用到的话，可以省略Me，也就是可以写成下面的形式。

const MyClass = class { /\* ... \*/ };

采用 Class 表达式，可以写出立即执行的 Class。

let person = new class {

constructor(name) {

this.name = name;

}

sayName() {

console.log(this.name);

}

}('张三');

person.sayName(); // "张三"

上面代码中，person是一个立即执行的类的实例。

## 6、不存在变量提升

类不存在变量提升（hoist），这一点与 ES5 完全不同。

## 7、私有方法

在命名上加以区别，在类的外部，还是可以调用到这个方法。

class Widget {

// 公有方法

foo (baz) {

this.\_bar(baz);

}

// 私有方法

\_bar(baz) {

return this.snaf = baz;

}

// ...

}

将私有方法移出模块，因为模块内部的所有方法都是对外可见的。使得bar实际上成为了当前模块的私有方法。

class Widget {

foo (baz) {

bar.call(this, baz);

}

// ...

}

function bar(baz) {

return this.snaf = baz;

}

利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值。

const bar = Symbol('bar');

const snaf = Symbol('snaf');

export default class myClass{

// 公有方法

foo(baz) {

this[bar](baz);

}

// 私有方法

[bar](baz) {

return this[snaf] = baz;

}

// ...

};

## 8、this的指向

在构造方法中绑定this，这样就不会找不到print方法了。

class Logger {

constructor() {

this.printName = this.printName.bind(this);

}

// ...

}

使用箭头函数

class Logger {

constructor() {

this.printName = (name = 'there') => {

this.print(`Hello ${name}`);

};

}

// ...

}

使用Proxy，获取方法的时候，自动绑定this。

function selfish (target) {

const cache = new WeakMap();

const handler = {

get (target, key) {

const value = Reflect.get(target, key);

if (typeof value !== 'function') {

return value;

}

if (!cache.has(value)) {

cache.set(value, value.bind(target));

}

return cache.get(value);

}

};

const proxy = new Proxy(target, handler);

return proxy;

}

const logger = selfish(new Logger());

## 9、name属性

由于本质上，ES6 的类只是 ES5 的构造函数的一层包装，所以函数的许多特性都被Class继承，包括name属性。

class Point {}

Point.name // "Point"

name属性总是返回紧跟在class关键字后面的类名。

## 10、Class 的取值函数（getter）和存值函数（setter）

## 11、Class 的取值函数（getter）和存值函数（setter）

## 12、Class 的静态方法

在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

class Foo {

static classMethod() {

return 'hello';

}

}

Foo.classMethod() // 'hello'

var foo = new Foo();

foo.classMethod()

// TypeError: foo.classMethod is not a function

父类的静态方法，可以被子类继承。

class Foo {

static classMethod() {

return 'hello';

}

}

class Bar extends Foo {

}

Bar.classMethod() // 'hello'

子类调用父类中的静态方法，静态方法也是可以从super对象上调用的。

class Foo {

static classMethod() {

return 'hello';

}

}

class Bar extends Foo {

static classMethod() {

return super.classMethod() + ', too';

}

}

Bar.classMethod() // "hello, too"

## 13、class 的静态属性和实例属性

### 1、类的实例属性

第一种写法：

class MyClass {

myProp = 42;

constructor() {

console.log(this.myProp); // 42

}

}

第二种写法：

class ReactCounter extends React.Component {

constructor(props) {

super(props);

this.state = {

count: 0

};

}

}

第三种写法：

class ReactCounter extends React.Component {

state = {

count: 0

};

}

### 2、类的静态属性

类的静态属性只要在上面的实例属性写法前面，加上static关键字就可以了。

class MyClass {

static myStaticProp = 42;

constructor() {

console.log(MyClass.myStaticProp); // 42

}

}

// 老写法

class Foo {

// ...

}

Foo.prop = 1;

// 新写法

class Foo {

static prop = 1;

}

## 14、new.target属性

ES6 为new命令引入了一个new.target属性，该属性一般用在在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

function Person(name) {

if (new.target !== undefined) {

this.name = name;

} else {

throw new Error('必须使用new生成实例');

}

}

// 另一种写法

function Person(name) {

if (new.target === Person) {

this.name = name;

} else {

throw new Error('必须使用 new 生成实例');

}

}

var person = new Person('张三'); // 正确

var notAPerson = Person.call(person, '张三'); // 报错

Class 内部调用new.target，返回当前 Class。

class Rectangle {

constructor(length, width) {

console.log(new.target === Rectangle);

this.length = length;

this.width = width;

}

}

var obj = new Rectangle(3, 4); // 输出 true

子类继承父类时，new.target会返回子类。

class Rectangle {

constructor(length, width) {

console.log(new.target === Rectangle);

// ...

}

}

class Square extends Rectangle {

constructor(length) {

super(length, length);

}

}

var obj = new Square(3); // 输出 false

利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

class Shape {

constructor() {

if (new.target === Shape) {

throw new Error('本类不能实例化');

}

}

}

class Rectangle extends Shape {

constructor(length, width) {

super();

// ...

}

}

var x = new Shape(); // 报错

var y = new Rectangle(3, 4); // 正确

# 十一、class的继承