**1. let 和 const**

let声明变量，const声明产量

特点：

(1)不具有变量提升，

(2)在声明之前存在暂时性死区，

(3)不允许重复声明，

(4)块级作用域

顶层对象：浏览器--> window， node-->global

ES5中，两者等价，导致顶层对象的属性是到处可以读写的，这非常不利于模块化编程。

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

**2. 变量解构赋值**

(1)解构不成功返回undefined

(2)允许指定默认值（使用严格相等运算符（===），严格等于undefined，默认值才会生效）

**3.字符串扩展**

**(1)模板字符串`` ，使用变量`My name is ${name}`**

(2)includes()包含某个字符

startsWith('a',n)从n位置以a开始

endsWith('a',n)从n位置以a结束

repeat(n)重复n次

padStart(n,'ab')/padEnd(n,'ab')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

trimStart()/trimEnd()去除空格

matchAll()正则匹配

**4.数值扩展**

Number.isFinite() 数值是否有限，参数不是数值，返回false

Number.isNaN() 是否NaN，参数类型不是NaN,返回false

Number.parseInt() 数值取整

Number.parseFloat() 浮点数

Number.isInteger() 是否整数，参数不是数值，返回false

**5.数组扩展**

**(1)...扩展运算符**

**(2)Array.from()将类数组（具有length属性）转为真正的数组**

( Array.prototype.slice.call(obj) 伪数组转数组 )

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。例：Array.from([1, 2, 3], (x) => x \* x) // [1, 4, 9]

Array.from()另一应用是将字符串转为数组，然后返回字符串的长度

(3)Array.of()用于将一组值，转换为数组

**6.对象扩展**

(1)Object.is() 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(+0, -0) // false

Object.is(NaN, NaN) // true

注：除了比较值，还会比较对象指向的地址。

(2)Object.assign(target, source1, source2) 对象的合并,第一个参数是目标对象，后面的参数都是源对象

注意:

a.如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

b.如果该参数不是对象，则会先转成对象，然后返回;

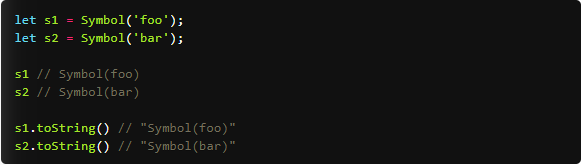
c.undefined和null无法转成对象，所以如果它们作为参数，就会报错。

**7.Symbol(新数据类型)**

为了解决ES5对象属性名的冲突，ES6引入了一种新的原始数据类型Symbol，表示独一无二的值。

Symbol值通过Symbol()生成，不能new，会报错。

Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。



注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的; Symbol 值不能与其他类型的值进行运算，会报错。

Symbol 值可以显式转为字符串，也可以转为布尔值，但是不能转为数值。

**Symbol.prototype.description创建Symbol的时候，可以添加一个描述**

const sym=Symbol(‘foo’)

sym.description // ’foo’

Symbol 值作为对象属性名时，不能用点运算符，因为点运算符后面总是字符串，不会读取Symbol作为标识符所指代的那个值，在对象内部，使用Symbol值定义属性时，必须放在方括号中。

**8.Set**

**Set本身是构造函数，可以生成Set数据结构，类似于数组，但是成员的值都是唯一的。**

可以用来数组去重，但是得到的是一个伪数组，需要再转成数组。

var arr = [1,2,3,6,4,2,5,7,3,5,2,9]

const newarr = new Set(arr) // 得到的newarr是伪数组

Array.from(newarr) // 返回真的数组

[…newarr] //返回真的数组

也可以用来去除字符串中重复的字符

[...new Set('ababbc')].join('')

注意：向Set加入值得时候，不会发生类型转换，也就是说数值5和字符‘5’是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是向 Set 加入值时认为NaN等于自身，而精确相等运算符认为NaN不等于自身。两个对象总是不相等的，两个空对象被视为两个值。

Set实例的属性和方法

属性：Set.prototype.constructor 构造函数，默认就是Set函数。

Set.prototype.size：返回Set实例的成员总数。

方法：（1）操作方法

Set.prototype.add(value) 添加某个值，返回set本身

Set.prototype.delete(value) 删除某个值，返回boolean，是否删除成功

Set.prototype.has(value) 是否有该值，返回boolean

Set.prototype.clear() 清除所有成员，没有返回值

（2）遍历方法

Set.prototype.keys() 返回键名的遍历器

Set.prototype.values() 返回键值的遍历器

Set.prototype.entries() 返回键值对的遍历器

Set.prototype.forEach() 使用回调函数遍历每个成员

**9.Map**

**Map数据结构，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。

构造函数，通过new生成Map数据结构。

Map实例的属性和方法

属性：Map.size：返回Map实例的成员总数。

方法：（1）操作方法

Map.prototype.set(key, value) 设置键名key对应的键值为value，返回整个 Map 结构

Map.prototype.get(key) 读取key对应的键值，如果找不到key，返回undefined。

Map.prototype.has(key) 是否有某个键，返回一个布尔值。

Map.prototype.delete(key) 删除某个键，返回boolean。

Map.prototype.clear() 清除所有成员，没有返回值。

（2）遍历方法

Map.prototype.keys() 返回键名的遍历器

Map.prototype.values() 返回键值的遍历器

Map.prototype.entries() 返回键值对的遍历器

Map.prototype.forEach() 使用回调函数遍历每个成员

需要特别注意的是，Map 的遍历顺序就是插入顺序。

**与其他数据结构的互相转换**

(1)Map转为数组，使用扩展运算符 […myMap]

(2) 数组转为Map，new Map([[true,7],[{foo:3},[‘abc’]]])

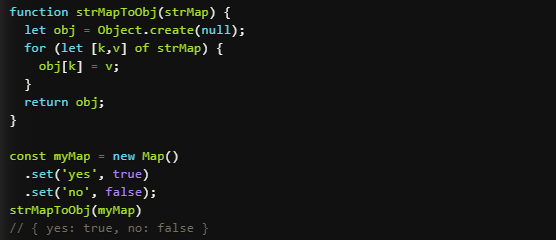
// Map:{

true=>7,

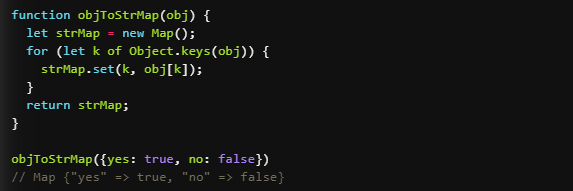
Object{foo:3}=>[’abc’]

}

(3)Map转为对象



(4)对象转为Map



**10.Proxy（代理器）**

Proxy用于修改某些操作的默认行为，可以理解成，在目标对象之前架设一层“拦截”，外界对该对象访问，都必须先通过这层拦截，可以对外界的访问进行过滤和改写。

Proxy构造函数，用来生成Proxy实例。

var proxy = new Proxy(traget, handler)

target表示所要拦截的目标对象，handler也是一个对象，用来定制拦截行为。如果handler没有设置任何拦截，等同于直接通向原对象。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}

});

proxy.time // 35

proxy.name // 35

proxy.title // 35

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

一个技巧是将 Proxy 对象，设置到object.proxy属性，从而可以在object对象上调用。

var object = { proxy: new Proxy(target, handler) };

Proxy 实例也可以作为其他对象的原型对象。

let obj = Object.create(proxy);

**11.Promise（解决回调地狱）**

Promise是异步编程的一种解决方案，Promise是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。

**1）Promise特点：**

A. 对象的状态不受外界影响。包含3种状态：pending进行中/fulfilled已成功/rejected已失败。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

B. 一旦状态改变，就不会再变，任何时候都可以得到这个结果，这时就称为 resolved（已定型）。

Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

Promise 的最大问题是代码冗余，原来的任务被 Promise 包装了一下，不管什么操作，一眼看去都是一堆then，原来的语义变得很不清楚。

**2）.基本用法**

Promise构造函数，生成Promise实例。

const promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。

resolve函数的作用是将Promise对象的状态从pending改为fulfilled（resolved），在异步操作成功时调用，并将异步操作的结果作为参数传递出去。

reject函数的作用是将Promise对象的状态从pending改为rejected，在异步操作失败时调用，并将异步操作报出的错误作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定resolved状态和reject状态的回调函数。

Promise 新建后就会立即执行。

let promise = new Promise(function(resolve, reject) {

console.log('Promise');

resolve();

});

promise.then(function() {

console.log('resolved.');

});

console.log('Hi!');

// Promise

// Hi!

// resolved

上面代码中，Promise 新建后立即执行，所以首先输出的是Promise。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以resolved最后输出。

Promise对象实现ajax操作例子：

const getJSON = function(url) {

const promise = new Promise(function(resolve, reject){

const handler = function() {

if (this.readyState !== 4) {

return;

}

if (this.status === 200) {

resolve(this.response);

} else {

reject(new Error(this.statusText));

}

};

const client = new XMLHttpRequest();

client.open("GET", url);

client.onreadystatechange = handler;

client.responseType = "json";

client.setRequestHeader("Accept", "application/json");

client.send();

});

return promise;

};

getJSON("/posts.json").then(function(json) {

console.log('Contents: ' + json);

}, function(error) {

console.error('出错了', error);

});

注意，调用resolve或reject并不会终结 Promise 的参数函数的执行。

new Promise((resolve, reject) => {

resolve(1);

console.log(2);

}).then(r => {

console.log(r);

});

// 2

// 1

上面代码中，调用resolve(1)以后，后面的console.log(2)还是会执行，并且会首先打印出来。这是因为立即 resolved 的 Promise 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务。

一般来说，调用resolve或reject以后，Promise 的使命就完成了，后继操作应该放到then方法里面，而不应该直接写在resolve或reject的后面。所以，最好在它们前面加上return语句，这样就不会有意外。

new Promise((resolve, reject) => {

return resolve(1);

// 后面的语句不会执行

console.log(2);

})

**3）.Promise.prototype.then()**

Promise实例具有then方法，then方法是定义在原型对象Promise.prototype上的。

作用：为Promise实例添加状态改变时的回调函数。then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

then方法返回的是一个新的promise实例，因此可以采用链式写法，即then方法后再调用另一个then方法。

**4）.Promise.prototype.catch()**

Promise.prototype.catch方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数。then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。如果 Promise 状态已经变成resolved，再抛出错误是无效的。

const promise = new Promise(function(resolve, reject) {

resolve('ok');

throw new Error('test');

});

promise

.then(function(value) { console.log(value) })

.catch(function(error) { console.log(error) });

// ok

上面代码中，Promise 在resolve语句后面，再抛出错误，不会被捕获，等于没有抛出。因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了。

catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。

**5）.Promise.all()**

Promise.all方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.all([p1, p2, p3]);

Promise.all方法接受一个数组作为参数,p1,p2,p3都是Promise实例。

**6）.Promise.race()**

Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。

const p = Promise.race([p1, p2, p3]);

只要p1/p2/p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

**12.Iterator和for…of**

(1). Iterator遍历器是一种接口，为各种不同的数据结构提供统一的访问机制。

Iterator 的作用有三个：

一是为各种数据结构，提供一个统一的、简便的访问接口；

二是使得数据结构的成员能够按某种次序排列；

三是 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费。

Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环。当使用for...of循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）。

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内。

原生具备Itarator接口的数据结构有：Array/Map/Set/String/TypedArray/arguments对象/NodeLost对象

(2).for…of循环

for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、Generator 对象，以及字符串。

**13.Generator**

Generator是一个状态机，封装了多个内部状态。还是一个遍历器对象生成函数，返回的是遍历器对象，可以依次遍历generator函数内部的每一个状态。

Generator函数的2个特征：

1. function关键字和函数名之间有一个星号\*；
2. （2）函数体内部使用yield表达式，定义不同的内部状态。

function\* helloWorldGenerator() {

yield 'hello';

yield 'world';

return 'ending';}

var hw = helloWorldGenerator();

Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象（Iterator Object）。必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。换言之，Generator 函数是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。

总结一下，调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

**Thunk 函数**

编译器的“传名调用”（即用到的时候才计算）实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做 Thunk 函数，它是“传名调用”的一种实现策略，用来替换某个表达式。

function f(m) {

return m \* 2;

}

f(x + 5);

// 等同于

var thunk = function () {

return x + 5;

};

function f(thunk) {

return thunk() \* 2;

}

**14.async await**

async await

**15.class**

class

**16.module(模块化)**

import，export，export default导出一个指定的模块

import和export命令只能在模块的顶层，不能在代码块之中。import命令会被JavaScript引擎静态分析，先于模块内的其他语句执行，导致无法动态加载，Node的require方法是运行时加载模块，import命令无法取代require的动态加载功能，因此引入了import()函数，完成动态加载。

import()返回一个 Promise 对象。下面是一个例子。

const main = document.querySelector('main');

import(`./section-modules/${someVariable}.js`)

.then(module => {

module.loadPageInto(main);

})

.catch(err => {

main.textContent = err.message;

});

import()函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，就会加载指定的模块。另外，import()函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

import()适用场景：按需加载，条件加载，动态的模块路径。

**注意：**import()加载模块成功以后，这个模块会作为一个对象，当作then方法的参数

**ES6模块与CommonJS模块的差异**

**(1)CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。**

CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的import有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

**(2)CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。**

因为 CommonJS 加载的是一个对象（即module.exports属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

**(3) ES6 模块之中，顶层的this指向undefined；CommonJS 模块的顶层this指向当前模块。**