## ES6

#### 1.新的变量声明方式 let/const

总结：

1. 没有变量提升，必须先声明再使用，否则报错。
2. 块级作用域：let声明的变量只在它所在的代码块有效
3. 相同作用域内，不能重复声明同一个变量
4. “暂时性死区”: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。

{

let a = 10;

var b = 1;

}

a // ReferenceError: a is not defined.

b // 1

let:变量

const：常量（指向固定的一个地址）

一旦声明变量，就必须立即初始化，不能留到以后赋值。

一旦声明，常量的值就不能改变。但是如果是个对象可以改变对象的属性值。

声明变量的6种方法：var function let const import class

块级作用域与函数声明

#### 变量的解构赋值

1. “模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。（数组，对象，字符串，数值，布尔值，函数参数）
2. 默认值，只有当其中一个成员等于undefined，才会默认赋值。null与undefined不严格相等。
3. 用途：变量赋值；交换变量的值；从函数返回多个值；函数参数的定义；提取 JSON 数据；函数参数的默认值；遍历 Map 结构；输入模块的指定方法

let [x = 1] = [undefined];

x // 1

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

对于 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"

如果变量名与属性名不一致，必须写成下面这样。

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。

对象的解构赋值可以取到继承的属性

const obj1 = {};

const obj2 = { foo: 'bar' };

Object.setPrototypeOf(obj1, obj2);

const { foo } = obj1;

foo // "bar"

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。

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

不能使用圆括号的情况：变量声明语句；函数参数；赋值语句的模式

赋值语句的非模式部分，可以使用圆括号

#### 字符串扩展

模板字符串：用反引号（`），`${变量}`

标签模板

alert`123`

// 等同于

alert(123)

for......of 循环遍历

codePointAt 拆开

String.fromCharCode合并

normalize()方法，用来将字符的不同表示方法统一为同样的形式。

includes(字符串，开始位置)：返回布尔值，表示是否找到了参数字符串。

startsWith(字符串，开始位置)：返回布尔值，表示参数字符串是否在原字符串的头部。

endsWith(字符串，开始位置)：返回布尔值，表示参数字符串是否在原字符串的尾部。

repeat（n）方法返回一个新字符串，表示将原字符串重复n次

at() : 返回字符串给定位置的字符

padStart(补全后的长度，用于补全的字符串): 用于头部补全；

padEnd(补全后的长度，用于补全的字符串): 用于尾部补全

用<%= ... %>输出 JavaScript 表达式

String.raw`` 阻止斜杠转译比如\\n->\\\\n

函数和传参或变量

#### 正则的扩展

修饰符 u

修饰符 y

#### 数值的扩展

Number对象

Math 对象上新增了 17 个与数学相关的方法

Number.isFinite()否整数 true/false

isInteger 是否整数 true/false

isNAN 是否NAN

parseInt

parseFloat

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

// 检查是不是有限的 (-2^53,2^53)

console.log(Number.isFinite(1)) // true

console.log(Number.isFinite(NaN)) // false

console.log(Number.isFinite(1/0)) // false

// 检查是不是NaN

console.log(Number.isNaN(NaN)) // true

console.log(Number.isNaN(0)) // false

// 检查是不是整数

console.log(Number.isInteger(1)) //true

// 这要注意

console.log(Number.isInteger(1.0)) //true

console.log(Number.isInteger(1.1)) //false

console.log(Number.isInteger('1')) //false

// 最大和最小安全数值

console.log(Number.MAX\_SAFE\_INTEGER)

console.log(Number.MIN\_SAFE\_INTEGER)

console.log(Number.isSafeIntger(1)) //true

// 取整

console.log(Math.trunc(1.1)) //1

console.log(Math.trunc(1.9)) //1

//判断一个数到底是正数、负数、还是零

console.log(Math.sign(-2)) //-1

console.log(Math.sign(0)) // 0

console.log(Math.sign(2)) // 1

console.log(Math.sign('2')) // 1

console.log(Math.sign('aa')) // NaN

#### 函数的扩展

##### 函数默认参数

1. 参数默认值可以与解构赋值的默认值，结合起来使用。给参数设置默认值
2. 参数变量是默认声明的
3. 有默认值，不能有同名的参数

function foo({x, y = 5}) {

console.log(x, y);}

foo({}) // undefined 5

foo({x: 1}) // 1 5

foo({x: 1, y: 2}) // 1 2

foo() // TypeError: Cannot read property 'x' of undefined

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）。

let x = 1;

function f(y = x) {

let x = 2;

console.log(y);}

f() // 1

##### rest 参数

rest 参数（形式为...变量名），可以传入任意数目的参数，实际上就是个数组，rest 参数之后不能再有其他参数

function add(...values) {

let sum = 0;

for (var val of values) {

sum += val;

}

return sum;}

add(2, 5, 3) // 10

##### 箭头函数的使用

var f = v => v;

// 等同于

var f = function (v) {return v;};

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。箭头函数根本没有自己的this，如果你在箭头函数中使用了this，那么该this一定就是外层的this，也因为这样不能用作构造函数；不可以使用arguments对象，用rest；不能用做Generator函数

function foo() {

setTimeout(() => {

console.log('id:', this.id);

}, 100);}

var id = 21;

foo.call({ id: 42 });

// id: 42

setTimeout里的this在普通函数应指向window，但是，箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42

箭头函数可以让this指向固定化，这种特性很有利于封装回调函数。

##### 尾调用优化

函数调用会在内存形成一个“调用记录”，又称“调用帧”（call frame），保存调用位置和内部变量等信息，尾调用优化可使调用帧只有一项。所有调用帧就是“调用栈”

#### 数组的扩展

##### 扩展运算符...

1.将一个数组转为用逗号分隔的参数序列。

[...[1,2,3]] //[1, 2, 3]

1. 代替apply方法，将数组转为函数的参数。
2. 复制数组（改变克隆的数组不会影响原数组）
3. 合并数组
4. 与解构赋值结合

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

first // 1

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

1. 字符串扩展成数组
2. Map 和 Set 结构，Generator 函数，遍历器（interator）可以使用
3. 扩展运算符（...）也可以将某些数据结构转为数组，比如NodeList ，arguments对象

// arguments对象

function foo() {

const args = [...arguments];

}

// NodeList对象

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

Array.from 将类似数组的结构转换成真正的数组

Array.of 方法用于将一组值，转换为数组。

copyWithin(target, start, end)

find：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。find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

findIndex：与find类似，符合条件返回成员位置，不符合条件返回-1

这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

function f(v){

return v > this.age;

}

let person = {name: 'John', age: 20};

[10, 12, 26, 15].find(f, person); // 26

数组map find findIndex 用法差不多返回的不一样

fill方法使用给定值，填充一个数组。

['a','b','c'].fill(2,0,2)//[2, 2, "c"]

keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

includes方法返回一个布尔值，表示某个数组是否包含给定的值

将空位转为undefined

for in

for of

#### 对象扩展

属性简写，方法也可以简写

const a={"b":"1","c":"2"};

a["b"];//1

a.b;//1

对象方法

is 比较两个值是否相等

assign方法用于对象的合并，后面属性覆盖前面的属性。浅拷贝，对象的引用。也可用于数组，字符串

\_\_proto\_\_属性（前后各两个下划线），用来读取或设置当前对象的prototype对象。

1. Object.setPrototypeOf()：写操作   
   2. Object.getPrototypeOf()：读操作   
   3. Object.create()：生成操作

in 方法判断属性是否在对象上

for in

for of

#### Symbol

独一无二的值

const sym = Symbol('foo');

sym.description // "foo"

在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在方括号之中。

#### Set和Map

唯一性数据不重复，可用于去重。返回的类似于数组，只有键值

给数组去重

[...new Set([2, 3, 5, 4, 5, 2, 2])]

add添加

has是否存在 返回一个布尔值

size长度

keys values

delete删除的值

clear

entries 返回键值对，变成数组

map ：返回的是键值对 ，new Map()和object配合

const map = new Map([

['name', '张三'],

['title', 'Author']]);

key不允许重复 ，重复会覆盖

set(key,value)

get(key)

has(key)delete/clear

{

const newClone = new Map().set("a","1").set( "b":"2" );

const newClone1 = newClone;

const obj = {}

for (let [k,v] of newClone) {

obj[k] = v;

}

newClone.set("c","3");

console.log(newClone);// {"a" => "1", "b" => "2", "c" => "3"}

console.log(newClone1);// {"a" => "1", "b" => "2", "c" => "3"}

console.log(obj);//{a: "1", b: "2"}

}

#### Proxy

代理器 拦截

#### Iterator

遍历器，是一种接口=>for..of

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。再次掉用next方法可以继续指向下一个成员，直到结束位置。

调用next方法返回value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

调用Symbol.iterator生成一个返回遍历对象

Generator函数实现Iterator接口

let myIterable = {

[Symbol.iterator]: function\* () {

yield 1;

yield 2;

yield 3;

}}[...myIterable] // [1, 2, 3]

// 或者采用下面的简洁写法

let obj = {

\* [Symbol.iterator]() {

yield 'hello';

yield 'world';

}};

for (let x of obj) {

console.log(x);}

// "hello"

// "world"

默认调用 Iterator 接口

1. 解构赋值：对数组和 Set 结构进行解构赋值时，会默认调用Symbol.iterator方法
2. 扩展运算符（...）
3. yield\* 后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。
4. 接受数组作为参数的场合，都调用了遍历器接口。

for...of

Array.from()

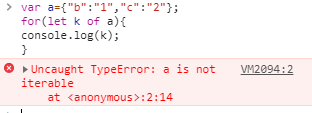
Map(), Set(), WeakMap(), WeakSet()（比如new Map([['a',1],['b',2]])）

Promise.all()

Promise.race()

1. DOM NodeList 对象、arguments对象的例子。

这样普通的对象不能用for...of遍历



#### Generator

Generator 函数是 ES6 提供的一种异步编程解决方案

function\* func(){

yield "a";

yield "b";

return "c"

}

var aFunc = func();

aFunc.next();

yield :yield表达式，定义Generator 函数内不同的内部状态。可以在等号右侧，否则要加圆括号

return结束执行

throw

next()

应用：

达到异步操作的效果

在任意对象上部署 Iterator 接口

#### async

对异步操作更好的控制，让异步操作按顺序执行

实际上await是个同步的命令 await 返回一个promise对象等待上一个执行完了才执行下一个。

async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await。

但是不需要依靠next()，内置的执行器可以自动执行。返回值是Promise对象，也就是说可以使用then来进行下一步

spawn()包裹的Generator函数会自动执行。

async比起Generator和Promise 实现起来更简洁

### 类

class 类名称{

constructer(){

}

}

constructer相当于立即执行

new 类名()

this是指向当前类的

新增属性通过.调用

### 继承

extends

super

### promise 对象

是异步编程的一种解决方案。Promise函数是同步执行的，但是.then是异步执行

三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）

触发其中一个，成功或失败

new Promise(function(Resolved,Reject){ Resolved() })

.then(

function(){ //成功 },

function(){ //失败 }) //成功

new Promise(function(Resolved,Reject){ Reject() })

.then(function(){ //成功 })

.cath(function(){ //失败 })//失败

能无限回调

.then().then()

throw 抛出异常 后面的代码不能执行

console.warn() 警告

race竞速方法 速度最快执行完成功就成功失败就失败，其他的不在执行

all规整方法 会同时开始执行 都是成功就都会成功，中间遇到失败就失败，不继续

### module

模块：将大的程序拆分成相互依赖的小文件，再用简单的方法拼装起来。

模块加载方案：commonJS，AMD

运行时加载，ES6编译时就加载完成

import 输入

export 输出

ES6模块自动采用严格模式

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。

export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

输出的必须是一个接口，函数和类同理

// 报错

export 1;

// 报错

var m = 1;

export m;

// 写法一

export var m = 1;

// 写法二

var m = 1;

export {m};

// 写法三

var n = 1;

export {n as m};

import载入的变量是只读的不能更改，如果是对象可以改写其属性

import具有提升效果，会提升到整个模块的头部，因为是编译时就加载执行了

不能用表达式和变量

载入相同的模块多次，只执行一次

export default 匿名函数默认输出，其实就是输出一个叫做default的变量

import的name可以是任意的

模块继承