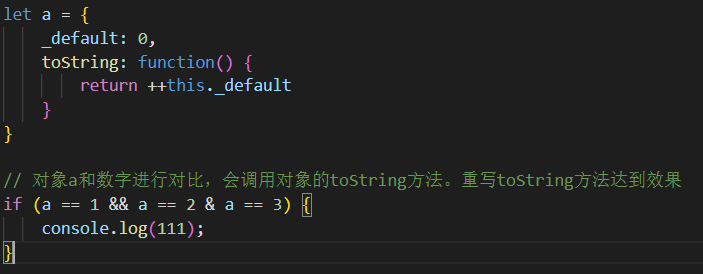
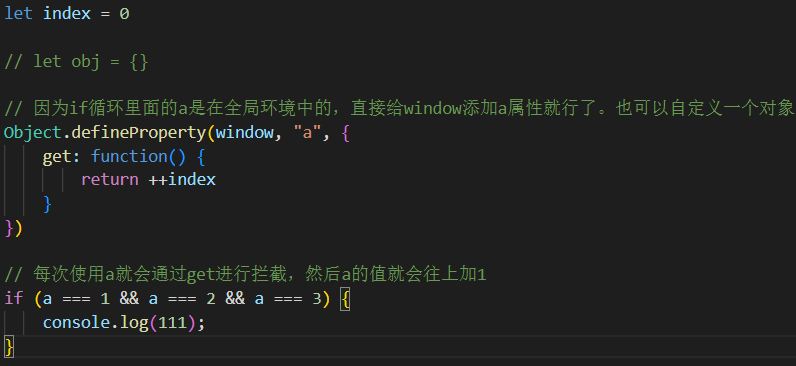
### 第一题 Object.defineProperty()

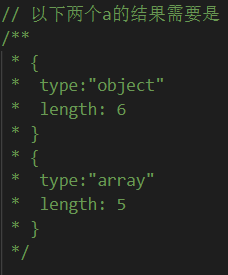
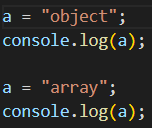
==



===



#### 思维扩散1



#### 思维扩散2



#### 思维扩散3



### 第二题



结果：30

解析：{}进行运算，会隐式转换为”[object Obejct]”



结果：0

解析：数组重写了toString方法，[1，2，3].toString() 🡺 “1,2,3”

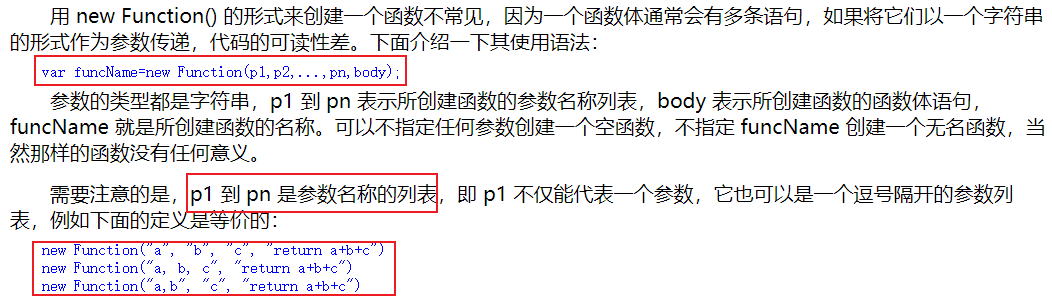
[] = “”



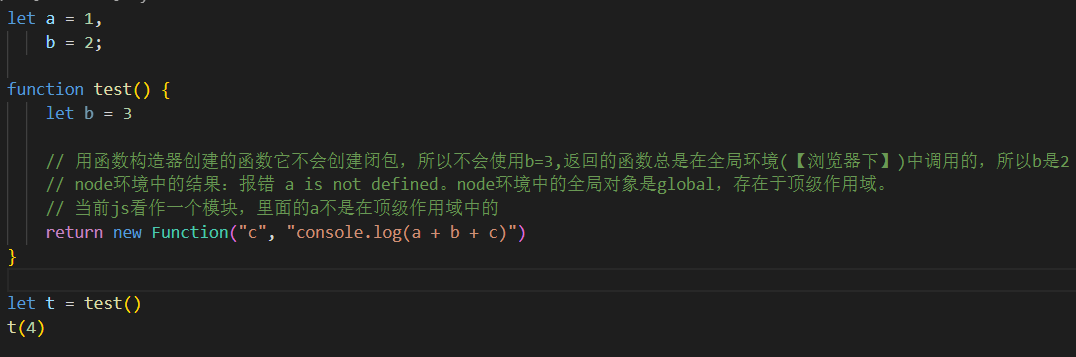
结果：0

解析：代码相当于

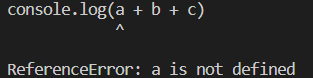
### 第三题：Function()相关



**通过new Function()声明的函数只能访问自己的作用域和全局的作用域**



node环境下的结果：



浏览器下的结果：

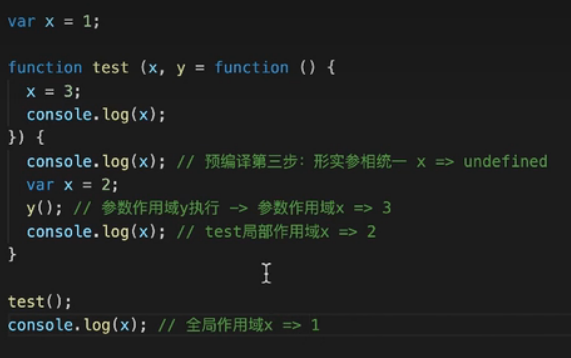


下面这两种方式调用Function是一样的



### 第四题(重点)：

[【全网首发:已完结】一道引发思考的『作用域间的纠葛』笔试题【知识储备】\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Wq4y167UZ?spm_id_from=333.999.0.0&vd_source=a67e813436aadc4608992466bdaea6bc)

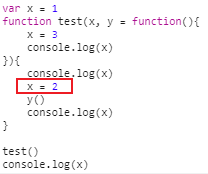


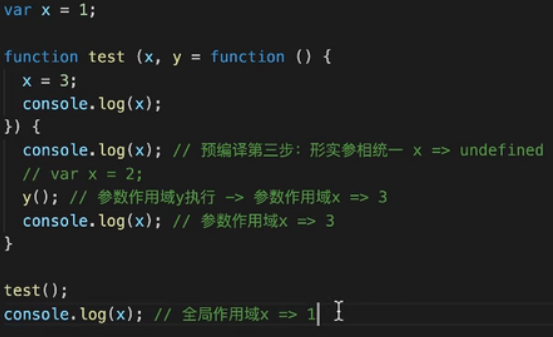
形参位置是有独立空间保存变量的.和函数体内部声明的变量，所在的作用域不同

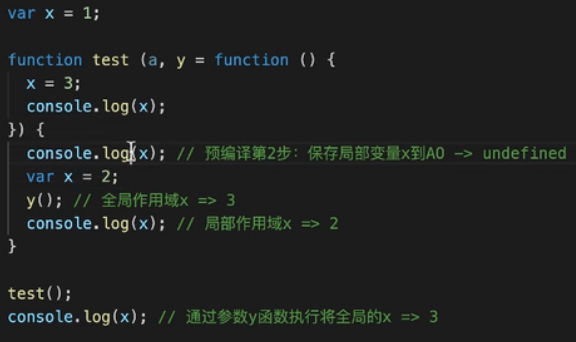
y的函数里面的x就是指向形参x

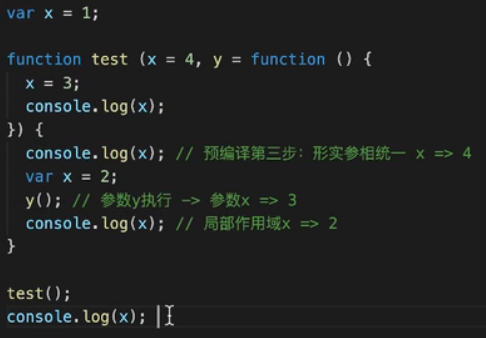
如果函数体里面重新let 了和形参名x一样的变量，那么这个变量x是当前函数体内的局部变量，和形参没有关系。函数体内部获取x变量，是从当前作用域里面获取。

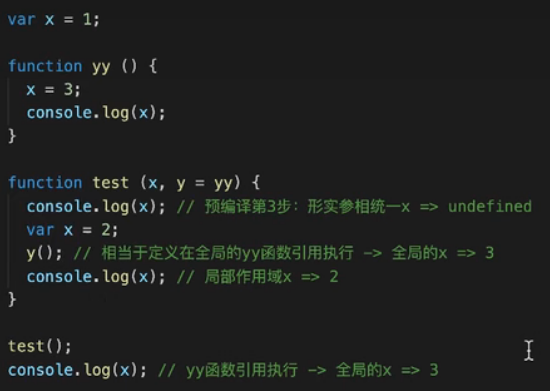
如果没有重新let 那么x就是形参x

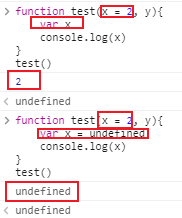
 





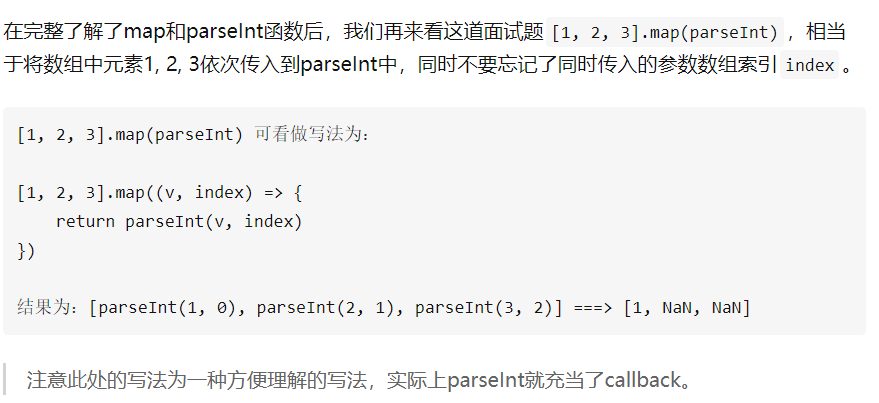




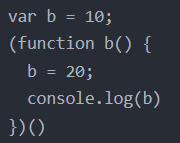


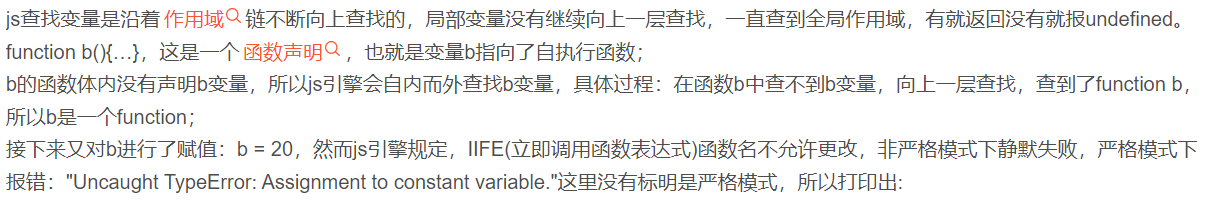
### 第五题

[1, 2, 3].map(parseInt)结果为？

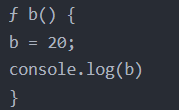


### 第六题

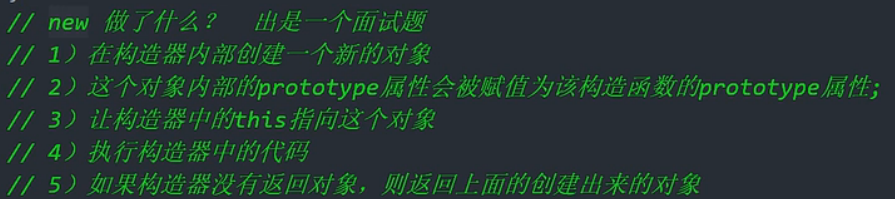


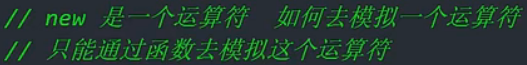


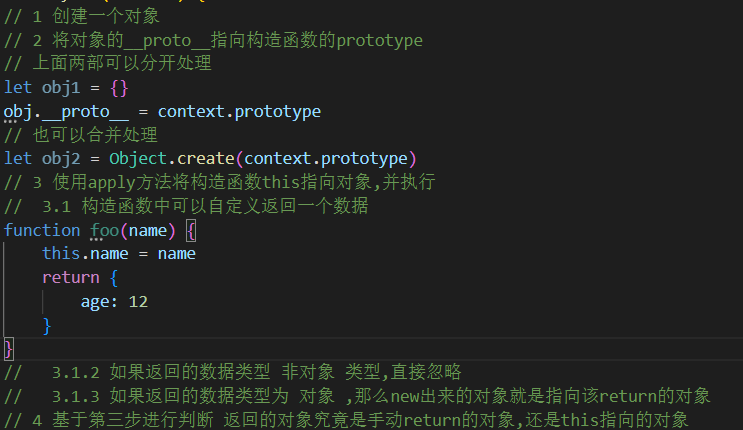
结果是立即执行函数



### new







#### 实现代码

function myNew(context) {

    // 1 创建一个对象

    // 2 将对象的\_\_proto\_\_指向构造函数的prototype

    // 3 使用apply方法将构造函数this指向对象,并执行

    //  3.1 构造函数中可以自定义返回一个数据

    //   3.1.2 如果返回的数据类型 非对象 类型,直接忽略

    //   3.1.3 如果返回的数据类型为 对象 ,那么new出来的对象就是指向该return的对象

    // 4 基于第三步进行判断 返回的对象究竟是手动return的对象,还是this指向的对象

    const obj = new Object

    obj.\_\_proto\_\_ = context.prototype

    let res = context.apply(obj, [...arguments].slice(1))

    return typeof res === "object" ? res : obj

}

function Foo(name, age) {

    this.name = name

    this.age = age

    this.fn = function() {

        console.log(this.name);

    }

    // return {

    //     gender: "王五"

    // }

}

let s = myNew(Foo, "zs", 12)

console.log(myNew(Foo, "zs", 12));

s.fn()

### 额外附加题

#### JSON.stringify()深拷贝

##### 弊端

如果 obj 的属性有时间对象，则 JSON.stringify 后再 JSON.parse 的结果，时间将转为字符串的形式，而不是对象的形式

如果 obj 的属性有 RegExp (正则表达式的缩写)、Error 对象，则序列化的结果将只得到空对象；

如果 obj 的属性有函数（function），undefined，Symbol， 则序列化的结果会把 function 或 undefined 丢失；

如果obj 的属性有 NaN、Infinity 和 -Infinity，则序列化的结果会变成 null

JSON.stringify() 只能序列化对象的可枚举的自有属性，例如如果 obj 中的对象是有构造函数生成的， 则使用 JSON.parse(JSON.stringify(obj)) 深拷贝后，会丢弃对象的 constructor；

如果对象中存在循环引用的情况也无法正确实现深拷贝，并且会导致死循环，最后抛出异常

-----------------------------------

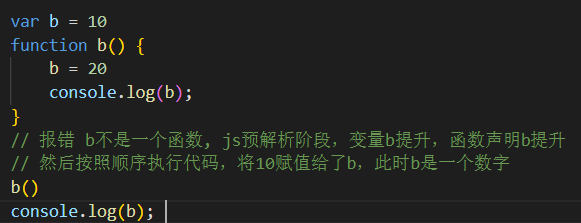
##### 其他常用场景

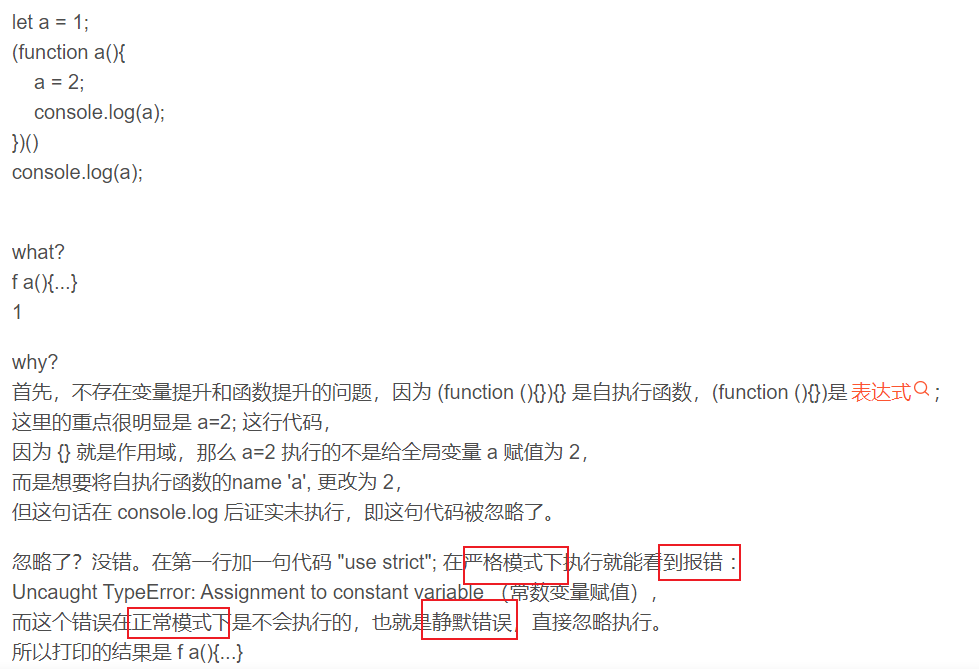
判断数组是否包含某对象： 将数组和要比对的对象转换成字符串后，比对字符串包含关系。（对象顺序很关键）

判断对象是否相等（对象顺序很关键）

让 localStorage / sessionStorage 可以存储对象

### 其他题目1





## 自定义的forEach解决异步问题

let arr = [

  () => console.log("start"),

  () => sleep(1000),

  () => console.log(1),

  () => sleep(2000),

  () => console.log("end"),

];

function sleep(delay) {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve();

    }, delay);

  });

}

Array.prototype.myForEach = async function (fn) {

  let arr = this,

    arrLen = arr.length,

    \_self = arguments[1];

  for (let i = 0; i < arrLen; i++) {

    await fn.apply(\_self, [arr[i], i, arr]);

  }

};

async function fn(arr) {

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

  //   await arr[i]();

  // }

  arr.myForEach(async (item) => {

    await item();

  });

}

fn(arr);

# js中继承的情况有哪几种

## 第一种：类继承

class Father {

  constructor(name) {

    this.name = name;

  }

}

class Child extends Father {

  constructor(age) {

    super("张三");

    this.age = age;

  }

}

const c = new Child(12);

console.log(c);



## 第二种：原型链继承

特点：1、实例可继承的属性有：

实例的构造函数的属性，父类构造函数属性，父类原型的属性。（新实例不会继承父类实例的属性！

缺点：1、新实例无法向父类构造函数传参。

2、继承单一。

3、所有新实例都会共享父类实例的属性。（原型上的属性是共享的，一个实例修改了原型属 性，另一个实例的原 型属性也会被修改！）

function Father() {

  this.name = "zhangsan";

}

function Child() {

  this.age = 12;

}

Child.prototype = new Father();

Child.prototype.constructor = Child;

const c = new Child();

console.log(c.constructor);

## 第三种：借用构造函数继承

特点：1、只继承了父类构造函数的属性，没有继承父类原型的属性。

2、解决了原型链继承缺点1、2、3。

3、可以继承多个构造函数属性（call多个）。

4、在子实例中可向父实例传参。

缺点：1、只能继承父类构造函数的属性。

2、无法实现构造函数的复用。（每次用每次都要重新调用）

3、每个新实例都有父类构造函数的副本，臃肿。

function Father() {

  this.name = "zhangsan";

}

function Child() {

  Father.call(this);

  this.age = 12;

}

const c = new Child();

console.log(c.name);

## 第四种：组合式继承(原型链继承+借用构造函数继承)

缺点：父构造函数new了两次，性能不是很好

优点：1、可以继承父类原型上的属性，可以传参，可复用。

2、每个新实例引入的构造函数属性是私有的

function Father() {

  this.name = "zhangsan";

  this.eat = function () {

    console.log("吃东西");

  };

}

// 父构造函数的原型方法

Father.prototype.run = function () {

  console.log("跑步");

};

function Child() {

  // 通过借用构造函数继承的方式，继承父构造函数的实例属性

  Father.call(this);

  this.age = 12;

}

Child.prototype = new Father();

Child.prototype.constructor = Child;

const c = new Child();

console.log(c.name); // 父构造函数的name值

c.eat(); // 父构造函数实例方法

c.run(); // 父构造函数原型上的方法

## 第五种：寄生组合继承

重点：

就是给原型式继承外面套了个壳子。

优点：

没有创建自定义类型，因为只是套了个壳子返回对象（这个），这个函数顺理成章就成 了创建的新对象。

缺点：

没用到原型，无法复用。

function Father() {

  this.name = "zhangsan";

  this.eat = function () {

    console.log("吃东西");

  };

}

// 父构造函数的原型方法

Father.prototype.run = function () {

  console.log("跑步");

};

function Child() {

  // 通过借用构造函数继承的方式，继承父构造函数的实例属性

  Father.call(this);

  this.age = 12;

}

// 改动点，就是在组合式继承基础上进行了一些改动,看下方红字部分

// Child.prototype = new Father();

Child.prototype = Object.create(Father.prototype);

Child.prototype.constructor = Child;

const c = new Child();

console.log(c.name); // 父构造函数的name值

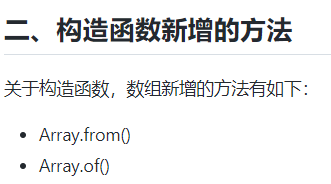
c.eat(); // 父构造函数实例方法

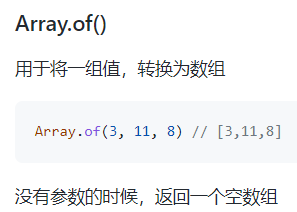
c.run(); // 父构造函数原型上的方法

console.log(c.constructor);

# ES6中数组新增了哪些扩展

1. 扩展元素符...
   1. 定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组
   2. 可以将字符串转为真正的数组
   3. 扩展运算符可以与解构赋值结合起来，用于生成数组







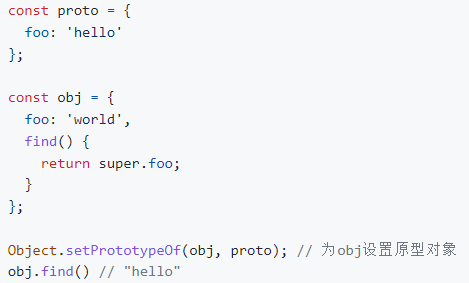
# ES6中对象新增了哪些扩展

## 一、属性的简写

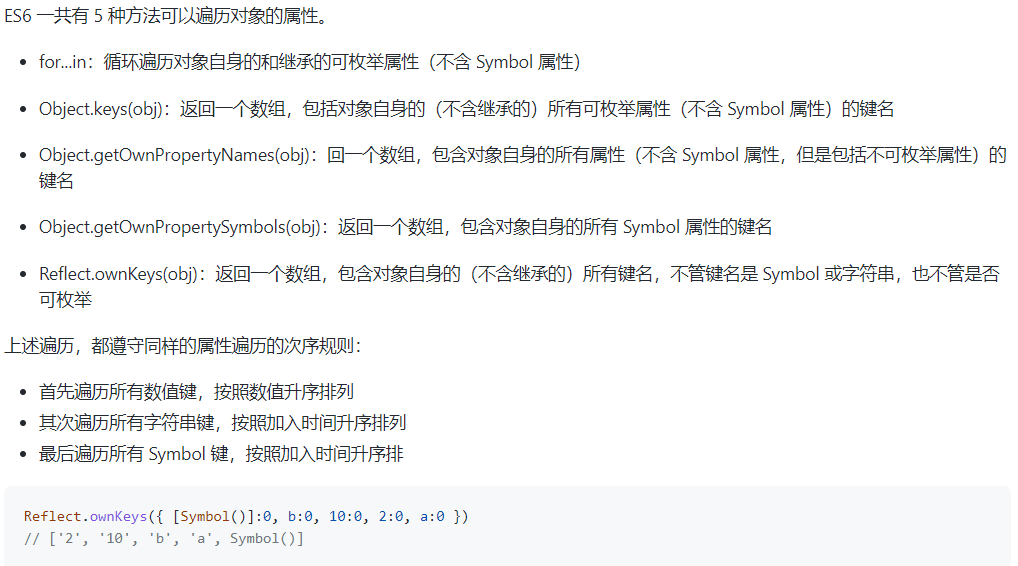
ES6中，当对象键名与对应值名相等的时候，可以进行简写

## super关键字

this关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字super，指向当前对象的原型对象



## 三、属性的遍历



。。。。。。。。。。

[面试官：ES6中对象新增了哪些扩展? · Issue #36 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/36)





# 整数反转

