### ****1. 数组的 push 方法****

var a = [].push(...[1, 2, 3]);

答案：3。

#### ****? 解析****

... 是 ES6 中的扩展运算符，相当于：push(1, 2, 3); push 方法返回的是该数组的新长度。因此是 3。Array 中的各个方法的返回值都是什么，可以往下看第五个问题。

### ****2. Set 数据结构存数据****

var set = new Set([0, 2, 2, 0, 0, 5, 9, {}, {}, NaN, NaN]);

答案：7。分别是：[0, 2, 5, 9, {}, {}, NaN]。

#### ****? 解析****

之所以有两个对象是因为对象是引用类型。而 NaN === NaN 总是返回 false，不应该也是两个吗？事实上，NaN 和 undefined 都可以被存储在 Set 中， NaN 之间被视为相同的值（NaN 被认为是相同的，尽管 NaN !== NaN）。

✅ ****拓展****：考虑下面的代码 set 数据中有哪些值？

var a = new Set([undefined, undefined, null, null, -0, +0, NaN, "", ""]);

结果：Set(5) {undefined, null, 0, NaN, ""}

### ****3. Object.is() 函数****

考虑下面的代码输出结果为 true 还是 false？

console.log(Object.is(-0, +0));

console.log(Object.is(0, -0));

console.log(Object.is(0, +0));

console.log(Object.is(NaN, NaN));

console.log(Object.is(NaN, 0/0));

console.log(Object.is([],[]));

答案：false,false,true,true,true,false。

#### ****? 解析****

Object.is() 判断两个值是否相同。如果下列任何一项成立，则两个值相同：

* 两个值都是 undefined；
* 两个值都是 null
* 两个值都是 true 或者都是 false
* 两个值是由相同个数的字符按照相同的顺序组成的字符串
* 两个值指向同一个对象
* 两个值都是数字并且：
  + 都是正零 +0（没有符号的 0 相当于 +0）
  + 都是负零 -0
  + 都是 NaN
  + 都是除零和 NaN 外的其它同一个数字

可以看出，这种相等性判断逻辑和传统的 == 运算不同。== 运算符会对它两边的操作数做隐式类型转换（如果它们类型不同），然后才进行相等性比较，但 Object.is 不会做这种类型转换。

这与 === 运算符的判定方式也不一样。=== 运算符（和 == 运算符）将数字值 -0 和 +0 视为相等，并认为 Number.NaN 不等于 NaN。

上边例题中，Object.is(NaN, 0/0) 之所以为 true，是因为 0/0 等于 NaN。而 0/2 等于多少呢？2/0 又是等于多少呢？于是诞生出了下面几个问题：

console.log(Object.is(NaN, 0/2)); // false

console.log(Object.is(NaN, 2/0)); // false

console.log(Object.is(2/0, 0/2)); // false

console.log(Object.is(Infinity, 2/0)); // true

console.log(Object.is(Infinity, Infinity)); // true

console.log(Object.is(-Infinity, -Infinity)); // true

在 JavaScript 中，0 除以一个不为零的数时，如果这个数是正数，则结果为 +0；如果这个数是负数，则结果是 -0。当一个不为零的数除以 0 时，当这个数是负数时，则结果是 -Infinity，表示负无穷；当这个数是正数时，则结果是 Infinity，表示正无穷；

在 JavaScript 中，无论是 == 还是 ===，Infinity 都等于自身，-Infinity 也是都等于自身。因此上面代码中，后三个结果是 true。

✅ ****拓展****：如何实现这个方法？

Object.is = function(x, y){

// 如果两个值相等

if (x === y) {

// 让 -0 != +0 成立

return x !== 0 || 1 / x === 1 / y;

} else {

// 如果两个值不相等（如：NaN ！== NaN）

// 让 NaN == NaN 成立

return x !== x && y !== y;

}}

解释一下上面将 -0 != +0 成立的原理。假如参数 x 为 +0，y 为 -0，x !== 0 显然不成立（反过来也一样），然后看第二个条件，1/+0 等于 Infinity，1/-0 等于 -Infinity。因此也不相等，于是返回 false。

****特例：**** Object.is(0.2 + 0.1, 0.3); // false

### ****4. 字符串中的 replace 函数****

考虑下面的代码执行结果是什么？

'a.b.c'.replace(/(.)\.(.)\.(.)/, '$2.$1.$0');

1. cba
2. c.b.a
3. ...
4. b.a.$0

答案：第四个选项正确。

#### ****? 解析****

要做出来这道题目，需要先了解 replace 这个函数。replace 函数非常强大，它是用来匹配特定的字符串或正则表达式，然后把匹配到的结果替换成新的字符串的函数。一般我们使用 replace 时第一个参数是一个字符串或者正则表达式，第二个参数是一个字符串：

// 当第一个参数是字符串时，仅第一个匹配项会被替换。"abcdcba".replace("a", "y"); // ybcdcba// 正则表达式中的 g 参数表示全局匹配，有了它，所有匹配项都会被代替"abcdcba".replace(/a/g, "y"); // ybcdcby

上面用法很常见。在正则表达式中你是否见过这样的表达式：

'abcdabc'.match(/^(\w+)\w\*\1$/);

上面会匹配到。而且匹配到了两个。一个是全局的匹配字符，另一个是捕获（括号里的内容 abc）。在这个正则表达式里 \1 是一个 反向引用，指向正则表达式中第 n 个括号（从左开始数）中匹配的子字符串。

在 replace 函数中，第二个参数可以插入下面的特殊变量名作为参数：

插入一个 "$"

replace 中的 $n 与正则表达式中的反向引用有些相似，而且都是 n 都是从 1 开始。因此上面的代码 '$2.$1.$0' 中的 $0 是没有意义的（一个普通的字符串）。$2 表示第左起二个括号，$1 表示左起第一个括号。于是被替换成了 b.a.$0。

✅ ****拓展****：replace 函数的第二个参数可以是一个函数。

当第二个参数是一个函数时，当匹配执行后，该函数就会执行。函数的返回值作为替换字符串。另外要注意的是，如果第一个参数是正则表达式，并且其为全局匹配模式，那么这个方法将被多次调用，每次匹配都会被调用。这个函数有多个参数：

匹配的子串（对应于上述的 $&）

下面的函数将所有出现的大写字母转换为小写，并且在匹配位置前加一个连字符（-）。

function styleHyphenFormat(propertyName){

function upperToHyphenLower(match) {

return '-' + match.toLowerCase();

}

return propertyName.replace(/[A-Z]/g, upperToHyphenLower);}

比如字符串是 borderTop，替换后将变成 border-top。

### ****5. Array 中的方法****

执行下列语句后，a.length 的值为：

var a = [];

a.push(1, 2);

a.shift(3, 4);

a.concat([5, 6]);

a.splice(0, 1, 2);

答案：1。

#### ****? 解析****

* push 函数可以一次 push 多个元素（[1,2]），并****返回更新后的数组的长度****；
* shift 没有参数，它表示删除数组的第一项元素，并****返回该元素的值****，会改变原来的数组（[1]）；
* concat 合并两个或多个数组，此方法不会更改现有数组，而是****返回一个新数组****（[1]）；
* splice 删除或替换现有元素或者原地添加新的元素来修改数组,并****以数组形式返回被修改的内容****。此方法会改变原数组（从下标 0 开始，删除 1 个元素，并从下标 1 开始插入一个元素 2，因此数组 a 变成了 [2]。a.length 等于 1）；

✅ ****拓展****：Array 数组方法中哪些方法会改变原数组？

除了上面提到的 push, shift, splice 之外还有这么几个会改变原数组：

1. sort 对数组的元素进行排序，并****返回数组****。默认排序顺序是在将元素转换为字符串，然后比较它们的UTF-16代码单元值序列时构建的。
2. pop 弹出数组的最后一个元素，并****返回该元素的值****；
3. unshift 将一个或多个元素添加到数组的开头，并****返回该数组的新长度****；
4. reverse 将数组中元素的位置颠倒，****并返回该数组****；
5. copyWithin(target, start?, end?) 浅复制数组的一部分到同一数组中的另一个位置，并****返回改变后的数组****，不会改变原数组的长度（例如：[1,2,3,4,5,6].copyWithin(4,0,2); 表示从下标 0 开始复制，一直复制到下标 2 结束，但不包括下标 2，把复制的内容转移到下标为 4 的位置及其以后的位置，就变成了 [1,2,3,4,1,2]）；
6. fill(value, start?, end?) 用一个固定值填充一个数组中从起始索引到终止索引内的全部元素，不包括终止索引，并****返回修改后的数组****；

后两个方法都是 ES6 或 ES7 新出的方法，因此兼容性可能不太好。

### ****6. 逻辑运算符****

考虑下面的代码，哪个是正确的？

var a = true;var b = 1;

1. (a || b) === true;
2. (b || a) === true;

答案：第一个正确。

#### ****? 解析****

| 运算符 | 语法 | 说明 |
| --- | --- | --- |
| 逻辑与，AND（&&） | expr1 && expr2 | 若 expr1 可转换为 true，则返回 expr2；否则，返回 expr1。 |
| 逻辑或，OR（||） | expr1 || expr2 | 若 expr1 可转换为 true，则返回 expr1；否则，返回 expr2。 |
| 逻辑非，NOT（!） | !expr | 若 expr 可转换为 true，则返回 false；否则，返回 true。 |

因此，上面代码中，(a || b) 表达式中的 a 是 true，则直接返回了 true，于是第一个正确。(b || a) 中的 b 是 1 可以转成 true，于是返回 b。但 1 === true 会返回 false，因此第二个选项不正确。

会被转换为 false 的表达式有：

* null；
* NaN；
* 0；
* 空字符串（"" or '' or ``）；
* undefined。

使用双重非运算符（!!）的一个场景，是显式地将任意值强制转换为其对应的布尔值，或者使用 Boolean 构造函数。

✅ ****拓展****：考虑下面的表达式，都会返回什么结果？

var a = 0 && "" && 1 && null;var b = 0 || "" || 1 || null;var c = false && (true || true);var d = false && true || true;var e = n5 = !!"";var f = ({} || ![]) && 1 || !!false;

需要说明的是，这三个运算符都是从左到右进行运算的。首先先看第一个，全都是 && 运算符，&& 运算符第一个为真值就返回第二个值，不是真值则返回第一个。因此判断步骤如下：

var a = 0 && "" && 1 && null;

a = 0 && 1 && null; // 0 不是真值，则返回它本身

a = 0 && null; // 0 不是真值，则返回它本身

a = 0; // 0 不是真值，则返回它本身

第二项全是 ||，|| 运算符前一个是真值就返回这个值，不是真值就返回另一个值，于是：

var b = 0 || "" || 1 || null;

b = "" || 1; // 0 不是真值，就返回另一个：""

b = 1 || null; // "" 不是真值，就返回另一个：1

b = 1; // 1 是真值，返回自身

通过上面也能发现一个规律，当一个表达式中的运算符全是 || 或者 && 时，如果是 || 运算符，它的第一个值是真值时，后面的表达式就不用再看了，必定返回第一个值。例如：1 || "" || 2 || 3 || null，返回结果是 1。而 && 运算符与 || 刚好相反，只要前面的值是假值，后面的就不用在看了，最终返回的必定是这个假值。例如：null && 0 && 1 && 2 && ""，返回的结果是 null。

c 中，先算括号里的内容，(true || true) 必定返回 true，false && true 的前一项是假值，直接返回这个值：false，即：c == false。

d 中，false && true 返回 false，变成 false || true，前一个值是假值，则返回第二个值，因此 d == true。

e 中，!!"" "" 是假值，!"" 变成真值（true），!!"" 又变成了假值（false）。因此，e == false。

f 的表达式比较复杂，首先计算括号里的内容。{} 是真值，![] 是假值（此时转变成了 false），括号里其实是 {} || false，前一个是真值，于是返回第一个，然后就变成了 {} && 1 || !!false。{} && 1 中 {} 是真值，于是返回第二个，然后就变成了 1 || !!false，因为 1 是正值，于是直接返回该值。所以 f == 1。

### ****7. JavaScript 数据类型****

* 问：JavaScript 中有几种****原始数据类型****（有时候也叫基本数据类型）？

答：六种，它们是一种既非对象也无方法的数据。分别是：string、number、boolean、null、undefined、symbol。

* 问：除了原始数据类型还有哪些类型？

答：除了原始数据类型就是引用类型，即：Object 对象类型。它是内存中的可以被标识符（指针）引用的一块区域。{} !== {}，比较的是地址。引用类型主要有：Object、Array、Date、Set、Map、类型数组（比如 Int8Array）、WeakMap、WeakSet、JSON等。

因此，Object 加上上面的六种原始数据类型，ECMAScript 标准定义了 7 种数据类型。

✅ ****拓展****：第七种原始数据类型：BigInt

BigInt 是一种内置对象，它提供了一种方法来表示大于 2^53 - 1 的整数。这原本是 Javascript 中可以用 Number 表示的最大数字。BigInt 可以表示任意大的整数。

可以用在一个整数字面量后面加 n 的方式定义一个 BigInt ，如：10n，或者调用函数 BigInt()。例如：

const theBiggestInt = 9007199254740991n;const alsoHuge = BigInt(9007199254740991);// ↪ 9007199254740991nconst hugeString = BigInt("9007199254740991");// ↪ 9007199254740991nconst hugeHex = BigInt("0x1fffffffffffff"); // 16进制// ↪ 9007199254740991nconst hugeBin = BigInt("0b11111111111111111111111111111111111111111111111111111"); // 二进制// ↪ 9007199254740991n

typeof hugeBin;// ↪ bigint

// 使用 Object 包装后， BigInt 被认为是一个普通 "object" ：typeof Object(1n) === 'object'; // true

它在某些方面类似于 Number ，但是也有几个关键的不同点：不能用于 Math 对象中的方法；不能和任何 Number 实例混合运算，两者必须转换成同一种类型。在两种类型来回转换时要小心，因为 BigInt 变量在转换成 Number 变量时可能会丢失精度。

可以和 BigInt 一起使用的操作符：+、\*、-、\*\*、% 。除 >>> （无符号右移）之外的位操作也可以支持。/ 操作符对于整数的运算也没问题。但该操作符结果会向零取整，也就是说不会返回小数部分。

const rounded = 5n / 2n; // 2nvar a = 1n + 3n; // 4n

#### ****比较****

BigInt 和 Number 不是严格相等的，但是宽松相等的。

1n === 1; // false

0n == 0; // true

0n === 0n; // true

1n < 2; // false

1n <= 2; // trueBoolean(0n); // falseBoolean(1n); // true!0n; // true

bigint 还可以存入数组中，并且能做排序操作：

const mixed = [4n, 6, -12n, 10, 4, 0, 0n];

mixed.sort();// ↪ [-12n, 0, 0n, 10, 4n, 4, 6]

有关更多关于 BigInt 的知识可以阅读 MDN 上的文档：****BigInt[1]****

### ****8. 字符转义****

下面的代码会返回什么结果？

'\\\\\\'.replace(new RegExp('\\\\\\\\', 'gi'), '/');

答案：/\

#### ****? 解析****

这道题看似是考察 replace 方法的用法，其实是考察正则表达式和字符串中的字符转义。new RegExp('\\\\\\\\') 返回值是这样的：/\\\\/，本来八个 \，变成了四个。原因是在 js 字符串中，\ 是特殊字符，它用于转义特殊字符，\\ 在字符串中相当于一个 \。

当你在控制台上输入一个 var str = '\\\\\\' 时，发现 str 的值实际是 \\\，六个变成了三个。

当定义下面的表达式将会报错，原因是 \ 会把它后面的 ' 给转义。

var str = '\'; // Invalid or unexpected token

要想让一个字符串等于 ' 或者 " 或者 \，可以这么做：

var str = '\''; // str == 'var str = "\""; // str == "var str = '""'; // str == ""var str = '\\'; // str = \var str = "'\""; // str = '"

var str = '/'; // str = /var str = '\/'; // str = /

除了上面的转义字符之外，还有下面这些：

* \0 空字符
* \n 换行符
* \t 水平制表符
* \f 换页符
* \r 回车符
* \b 退格符
* \v 垂直制表符
* \uXXXX unicode 码

最终这个代码变成了 '\\\'.replace(/\\\\/gi, '/')。字符转义转义完了，但是正则表达式中也需要字符转义，\ 同样也是用于转义特殊字符，\\ 在正则表达式中会匹配一个 \。因此 /\\\\/ 正则表达式其实匹配的是 \\，于是会把 \\\ 字符串中的 \\ 替换成 /，最终结果是 /\。

正则表达式中的字符转义：

* \t 匹配一个水平制表符（tab）
* \r 匹配一个回车符（carriage return）
* \n 匹配一个换行符（linefeed）
* \v 匹配一个垂直制表符（vertical tab）
* \f 匹配一个换页符（form-feed）
* [\b] 匹配一个退格符（backspace）（不要与 \b 混淆，\b 表示匹配一个单词边界）
* \0 匹配一个 NUL 字符。不要在此后面跟小数点；

正则表达式中的 \ 对于那些通常被认为字面意义的字符来说，表示下一个字符具有特殊用处，并且不会被按照字面意义解释。例如 /b/ 匹配字符 'b'。在 b 前面加上一个反斜杠，即使用 /\b/，则该字符变得特殊，以为这匹配一个单词边界。

对于那些通常特殊对待的字符，表示下一个字符不具有特殊用途，会被按照字面意义解释。例如，\* 是一个特殊字符，表示匹配某个字符 0 或多次，如 /a\*/ 意味着 0 或多个 "a"。为了匹配字面意义上的 \_ ，在它前面加上一个反斜杠，例如，/a\\*/匹配 a\_。

在正则表达式中，如果要匹配 \*、[、]、{、}、(、)、-、^、$、\、|、?、+、/、\ 等一些在正则表达式中有特殊含义的字符时，应在前面加一个 \ 作转义。

### ****9. 变量提升****

下面程序打印的结果是？

function fn(a){

console.log(a);

var a = 123;

console.log(a);

function a(){}

console.log(a);

console.log(b);

var b = function(){}}

fn(2);

答案：function a(){}、123、123、undefined

#### ****? 解析****

在运行 fn 函数时，首先会扫描代码，把变量声明和函数声明提到函数顶部，因此上面代码就变成了：

function fn(a){

var a;

function a(){};

var b;

console.log(a); // function

a = 123;

console.log(a); // 123

console.log(a); // 123

console.log(b); // undefined

b = function(){}}

这里需要注意的是：var b = function(){} 是函数表达式，而非函数声明。因此提升的是变量 b，它会默认等于 undefined。function b(){} 才是函数声明。

再看下一个例子：

console.log(fn);function fn(fn){

console.log(fn);

var fn = 111;

function fn(){};}fn(222);var fn = 123;

可能会出乎意料，可能会认为：fn 函数先提升，后又声明了 fn，此时打印 fn 的值不应该是 undefined 吗？这里有一个误区，当一个变量有值但又声明了一次，这两个变量会合并成一个，值会保留，例如：

var aaa = 111;var aaa;

console.log(aaa);

打印结果是 111，当你明确给第二个 aaa 变量赋为 undefined 时打印结果才是 undefined。

上面的问题也是，var fn = 123; 中 fn 变量会提升，发现已经有一个 fn 变量了，而且是个函数，于是第一次打印 fn 就是一个函数。然后调用 fn 函数。

调用 fn 函数，fn 函数内部也需要变量提升，先提升 fn 变量，然后提升 fn 函数声明，于是打印出 fn 是一个函数。

function fn(fn){

function fn(){};

console.log(fn);

fn = 111;}

✅ ****拓展****：ES6 中的 let、const 关键字

ES6 允许我们可以使用 let、const 声明变量。使用这两个关键字将不会提升变量到代码块的顶部。因此，在变量声明之前引用这个变量，将抛出引用错误。被 let、const 声明的变量将从代码块一开始的时候就处在一个“暂时性死区”，直到这个变量被声明为止。而且使用 let 或者 const 不能重复声明变量。使用 let、const 必须先声明再赋值。

比如下面的例子：

function fn(n){

console.log("start i == ", i); // undefined

for(var i = 0;i < n;i ++){

setTimeout(function(){

console.log(i);

}, 100);

}

console.log("end i == ",i); // 3}fn(3);

定时器的打印结果是 3 3 3。而且循环两端也能打印出变量 i 的值，这是因为开始循环之前，i 被提升到了函数顶端。而 setTimout 执行完毕后会在最近的作用域中寻找变量 i，此时 i 已经变成了 3。而如果使用 let 声明变量 i，报错，表示 i 没有定义，说明变量不会提升，去掉两边的打印后，就不再报错，定时器就会输出 0 1 2。

let 不仅不会提升变量，声明的变量相当于声明了一个作用域，这个作用域被限制在块级中的变量、语句或者表达式中。

for(let i = 0;i < n;i ++>){}

for 循环就是一个语句，let 使得****每次程序进入花括号就产生了一个块级作用域****，相当于是每个 setTimeout 处在不同的作用域内，每个作用域的 i 值各不相同。然后打印出了 0 1 2。let 最好不要在 if 语句中使用，不然外部访问不到 if 语句内部声明的变量。在 switch-case 中声明变量时，别的 case 语句也访问不到：

let x = 1;switch(x) {

case 0:

let foo;

break;

case 1:

let foo; // SyntaxError for redeclaration.

break;}

### ****10. 闭包****

一个经典的例子：

function fn(n){

for(var i = 0;i < n;i ++){

setTimeout(function(){

console.log(i);

}, 100);

}}fn(3);

答案：3 3 3

#### ****? 解析****

setTimeout 是一个异步函数，异步函数意味着什么？同步代码执行完毕后他才会执行，上面代码中循环了三次，创建了三个定时器，定时器会等待执行（等待同步代码执行完，开始下一轮的宏任务）当同步代码执行完后，变量 i 变成了 3，这时开始调用定时器，于是全部打印出了 3。

要想打印出我们预期的 0 1 2，则可以利用闭包实现（新建了一个函数作用域）：

(function(i){

setTimeout(function(){

console.log(i);

}, 100);})(i)

这里利用了闭包，通过立即执行函数传参的形式把每次循环的 i 值保存起来，当调用定时器时，打印的 i 是立即执行函数保存的 i，于是打印出了 0 1 2。当然你也可以使用 let 声明变量。

闭包可以让你从内部函数访问外部函数作用域。在 JavaScript 中，每当函数被创建，就会在函数生成时生成闭包。例如下面的例子：

function func(){

var x = 0;

return function(){

console.log(x ++);

}}var f1 = func();var f2 = func();f1();f1();f2();

打印的结果将是：0 1 0。这里有一个小小的知识点，console.log(x ++); 会先打印 x，然后自加一。而 ++x 会先自加。func 内部的返回的函数就是一个闭包，它内部没有 x 变量，但可以往上层访问到外层函数的变量，当调用 f1 时，x 会加一，再次调用后又会加一。需要注意的是 f2 也是一个调用 func 后返回的函数，但 f1 与 f2 并不是一个函数，从本质上讲，func 是一个函数工厂，使用函数工厂创建了两个新函数。这两个函数可以说是独立的，互不影响的。它们共享相同的函数定义，但是保存了不同的词法环境（该函数和对其周围状态的引用）。

#### ****与 this 结合****

考虑下面的问题，打印结果会是什么？

var Test= {

foo: "test",

func: function() {

var self = this;

console.log(this.foo);

console.log(self.foo);

(function() {

console.log(this.foo);

console.log(self.foo);

}());

}};

Test.func();

答案：test test undefined test

#### ****? 解析****

前两个很容易理解，func 是对象里的一个方法，this 指向，func 内部有一个立即执行函数，它就是一个闭包。该函数要访问外部的 this.foo 和 self.foo。很显然 self.foo 能访问到，因为在 func 函数中 self 变量引用了 this。而 this.foo 是 undefined，说明访问不到 this。原因是：函数在调用时，this 默认指向 window 对象。这就好比下面的例子（立即执行函数是 f2）：

function f1(){

function f2(){

console.log(this); // window

}

f2();}f1();

✅ ****拓展****：严格模式下的 this 指向和箭头函数中的 this 指向

严格模式下，如果 this 没有被执行环境定义，那它将保持为 undefined。如果要想把 this 的值从一个环境传到另一个，就要用 call 或者 apply 方法。

在箭头函数中，this 与封闭词法环境的 this 保持一致。在全局代码中，它将被设置为全局对象。如果将 this 传递给 call、bind、或者 apply，它将被忽略。不过你仍然可以为调用添加参数，不过第一个参数（thisArg）应该设置为 null。

例如下面的例子，虽然有些不恰当，但足以说明问题。最好不要让对象的方法是一个箭头函数，箭头函数中的 this 并不指向 obj。箭头函数 this 的指向（不仅仅是 this，其实 super,new.target 等）由外围最近一层非箭头函数决定。

var obj = {

name: 'Ming',}const getName = (age) => {

return this.name + ' ' + age;}// call 的第一个参数将被忽略var res = getName.call(obj, 18);

console.log(res); // ' 18'

考虑下面的代码：

var obj = {

bar: function() {

var x = (() => this);

return x;

}};

var fn = obj.bar();

console.log(fn() === obj);

var fn2 = obj.bar;

console.log(fn2()() == window);

答案是 true 和 true。作为 obj 对象的一个方法来调用 bar，会把它的 this 绑定到 obj。但是注意，如果你只是引用 obj 的方法，而没有调用它（var fn2 = obj.bar），那么调用箭头函数后，this 指向 window，因为 fn2 函数相当于一个全局的函数，this 指向 window。fn2 是箭头函数的外层函数，会继承外层函数的 this。

### ****参考资料****

BigInt: *https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/BigInt*