看完这几道 JavaScript 面试题，让你与考官对答如流（下）

原创 前端小智 大迁世界 2020-01-12 18:30

> 作者：Mark A
> 译者：前端小智
> 来源：dev

由于篇幅过长，我将此系列分成上中下三篇，上、中篇：

看完这几道 JavaScript 面试题，让你与考官对答如流（中）

看完这几道 JavaScript 面试题，让你与考官对答如流（上）

- 51. 什么是 `async/await` 及其如何工作?
- 52. 展开运算符和Rest运算符有什么区别?
- 53. 什么是默认参数?
- 54. 什么是包装对象（wrapper object）?
- 55. 隐式和显式转换有什么区别?
- 56. 什么是NaN? 以及如何检查值是否为 NaN?
- 57. 如何判断值是否为数组?
- 58. 如何在不使用`%`模运算符的情况下检查一个数字是否是偶数?
- 59. 如何检查对象中是否存在某个属性?
- 60. AJAX 是什么?
- 61. 如何在JavaScript中创建对象?
- 62. Object.seal 和 Object.freeze 方法之间有什么区别?
- 63. 对象中的 in 运算符和 hasOwnProperty 方法有什么区别?
- 64. 有哪些方法可以处理javascript中的异步代码?
- 65. 函数表达式和函数声明之间有什么区别?
- 66. 调用函数，可以使用哪些方法?
- 67. 什么是缓存及它有什么作用?
- 68. 手动实现缓存方法
- 69. 为什么typeof null返回 object? 如何检查一个值是否为 null?
- 70. new 关键字有什么作用?
- 71. 什么时候不使用箭头函数? 说出三个或更多的例子?
- 72. Object.freeze() 和 const 的区别是什么?
- 73. 如何在 JS 中“深冻结”对象?
- 74. `Iterator`是什么，有什么作用?
- 75. `Generator` 函数是什么，有什么作用?

51. 什么是 `async/await` 及其如何工作?

`async/await` 是 JS 中编写异步或非阻塞代码的新方法。它建立在**Promises**之上，让异步代码的可读性和简洁度都更高。

`async/await` 是 JS 中编写异步或非阻塞代码的新方法。它建立在 `Promises` 之上，相对于 Promise 和回调，它的可读性和简洁度都更高。但是，在使用此功能之前，我们必须先学习 `Promises` 的基础知识，因为正如我之前所说，它是基于 `Promise` 构建的，这意味着幕后使用仍然是**Promise**。

使用 Promise

```
function callApi() {
  return fetch("url/to/api/endpoint")
    .then(resp => resp.json())
    .then(data => {
      //do something with "data"
    }).catch(err => {
      //do something with "err"
    });
}
```

使用async/await

在 `async/await` ，我们使用 tru/catch 语法来捕获异常。

```
async function callApi() {
  try {
    const resp = await fetch("url/to/api/endpoint");
    const data = await resp.json();
    //do something with "data"
  } catch (e) {
    //do something with "err"
  }
}
```

注意:使用 `async` 关键声明函数会隐式返回一个**Promise**。

```
const giveMeOne = async () => 1;

giveMeOne()
  .then((num) => {
    console.log(num); // logs 1
  });
```

注意: `await` 关键字只能在 `async function` 中使用。在任何非**async function**的函数中使用 `await` 关键字都会抛出错误。 `await` 关键字在执行下一行代码之前等待右侧表达式(可能是一个**Promise**)返回。

```
const giveMeOne = async () => 1;

function getOne() {
  try {
    const num = await giveMeOne();
    console.log(num);
  } catch (e) {
    console.log(e);
  }
}

// Uncaught SyntaxError: await is only valid in async function
```

```
async function getTwo() {
  try {
    const num1 = await giveMeOne(); // 这行会等待右侧表达式执行完成
    const num2 = await giveMeOne();
    return num1 + num2;
  } catch (e) {
    console.log(e);
  }
}

await getTwo(); // 2
```

52. 展开(spread)运算符和 剩余(Rest) 运算符有什么区别?

展开运算符(spread)是三个点(...)，可以将一个数组转为用逗号分隔的参数序列。说的通俗易懂点，有点像化骨绵掌，把一个大元素给打散成一个个单独的小元素。

剩余运算符也是用三个点(...)表示，它的样子看起来和展开操作符一样，但是它是用于解构数组和对象。在某种程度上，剩余元素和展开元素相反，展开元素会“展开”数组变成多个元素，剩余元素会收集多个元素和“压缩”成一个单一的元素。

```
function add(a, b) {
  return a + b;
};

const nums = [5, 6];
const sum = add(...nums);
console.log(sum);
```

在本例中，我们在调用 `add` 函数时使用了展开操作符，对 `nums` 数组进行展开。所以参数 `a` 的值是 `5` ， `b` 的值是 `6` ，所以 `sum` 是 `11` 。

```
function add(...rest) {
  return rest.reduce((total,current) => total + current);
};

console.log(add(1, 2)); // 3
console.log(add(1, 2, 3, 4, 5)); // 15
```

在本例中，我们有一个 `add` 函数，它接受任意数量的参数，并将它们全部相加，然后返回总数。

```
const [first, ...others] = [1, 2, 3, 4, 5];
console.log(first); // 1
console.log(others); // [2,3,4,5]
```

这里，我们使用剩余操作符提取所有剩余的数组值，并将它们放入除第一项之外的其他数组中。

53. 什么是默认参数?

默认参数是在 JS 中定义默认变量的一种新方法，它在ES6或ECMAScript 2015版本中可用。

```
//ES5 Version
function add(a,b){
  a = a || 0;
  b = b || 0;
  return a + b;
}

//ES6 Version
function add(a = 0, b = 0){
  return a + b;
}
add(1); // returns 1
```

我们还可以在默认参数中使用解构。

```
function getFirst([first, ...rest] = [0, 1]) {
  return first;
}

getFirst();  // 0
getFirst([10,20,30]);  // 10

function getArr({ nums } = { nums: [1, 2, 3, 4] }){
    return nums;
}

getArr(); // [1, 2, 3, 4]
getArr({nums:[5,4,3,2,1]}); // [5,4,3,2,1]
```

我们还可以使用先定义的参数再定义它们之后的参数。

```
function doSomethingWithValue(value = "Hello World", callback = () => { console.log(val
  callback();
}
doSomethingWithValue(); //"Hello World"
```

54. 什么是包装对象（wrapper object）？

我们现在复习一下JS的数据类型，JS数据类型被分为两大类，**基本类型**和**引用类型**。

基本类型： `Undefined` , `Null` , `Boolean` , `Number` , `String` , `Symbol` , `BigInt`

引用类型： `Object` , `Array` , `Date` , `RegExp` 等，说白了就是对象。

其中引用类型有方法和属性，但是基本类型是没有的，但我们经常会看到下面的代码：

```
let name = "marko";

console.log(typeof name); // "string"
console.log(name.toUpperCase()); // "MARKO"
```

`name` 类型是 `string` ，属于基本类型，所以它没有属性和方法，但是在这个例子中，我们调用了一个 `toUpperCase()` 方法，它不会抛出错误，还返回了对象的变量值。

原因是基本类型的值被临时转换或强制转换为**对象**，因此 `name` 变量的行为类似于**对象**。除 `null` 和 `undefined` 之外的每个基本类型都有自己**包装对象**。也就是：`String`，`Number`，`Boolean`，`Symbol` 和 `BigInt`。在这种情况下，`name.toUpperCase()` 在幕后看起来如下：

```
console.log(new String(name).toUpperCase()); // "MARKO"
```

在完成访问属性或调用方法之后，新创建的对象将立即被丢弃。

55. 隐式和显式转换有什么区别）？

隐式强制转换是一种将值转换为另一种类型的方法，这个过程是自动完成的，无需我们手动操作。

假设我们下面有一个例子。

```
console.log(1 + '6'); // 16
console.log(false + true); // 1
console.log(6 * '2'); // 12
```

第一个 `console.log` 语句结果为 `16`。在其他语言中，这会抛出编译时错误，但在 JS 中，`1` 被转换成字符串，然后与 `+运` 算符连接。我们没有做任何事情，它是由 JS 自动完成。

第二个 `console.log` 语句结果为 `1`，JS 将 `false` 转换为 `boolean` 值为 `0`，，`true` 为 `1`，因此结果为 `1`。

第三个 `console.log` 语句结果 `12`，它将 `'2'` 转换为一个数字，然后乘以 `6 * 2`，结果是12。

而显式强制是将值转换为另一种类型的方法，我们需要手动转换。

```
console.log(1 + parseInt('6'));
```

在本例中，我们使用 `parseInt` 函数将 `'6'` 转换为 `number`，然后使用 `+` 运算符将 `1` 和 `6` 相加。

56. 什么是NaN？以及如何检查值是否为NaN？

`NaN` 表示**“非数字”**是 JS 中的一个值，该值是将数字转换或执行为非数字值的运算结果，因此结果为 `NaN`。

```
let a;

console.log(parseInt('abc')); // NaN
console.log(parseInt(null)); // NaN
console.log(parseInt(undefined)); // NaN
console.log(parseInt(++a)); // NaN
```

```
console.log(parseInt({} * 10)); // NaN
console.log(parseInt('abc' - 2)); // NaN
console.log(parseInt(0 / 0)); // NaN
console.log(parseInt('10a' * 10)); // NaN
```

JS 有一个内置的 isNaN 方法，用于测试值是否为isNaN值，但是这个函数有一个奇怪的行为。

```
console.log(isNaN()); // true
console.log(isNaN(undefined)); // true
console.log(isNaN({})); // true
console.log(isNaN(String('a'))); // true
console.log(isNaN(() => { })); // true
```

所有这些 console.log 语句都返回 true ，即使我们传递的值不是 NaN 。

在 ES6 中，建议使用 Number.isNaN 方法，因为它确实会检查该值（如果确实是 NaN ），或者我们可以使自己的辅助函数检查此问题，因为在 JS 中，**NaN**是唯一的值，它不等于自己。

```
function checkIfNaN(value) {
  return value !== value;
}
```

57. 如何判断值是否为数组？

我们可以使用 Array.isArray 方法来检查值是否为**数组**。当传递给它的参数是数组时，它返回 true ，否则返回 false 。

```
console.log(Array.isArray(5));  // false
console.log(Array.isArray("")); // false
console.log(Array.isArray()); // false
console.log(Array.isArray(null)); // false
console.log(Array.isArray({ length: 5 })); // false

console.log(Array.isArray([])); // true
```

如果环境不支持此方法，则可以使用 polyfill 实现。

```
function isArray(value){
 return Object.prototype.toString.call(value) === "[object Array]"
}
```

当然还可以使用传统的方法：

```
let a = []
if (a instanceof Array) {
  console.log('是数组')
} else {
  console.log('非数组')
}
```

58. 如何在不使用`%`模运算符的情况下检查一个数字是否是偶数?

我们可以对这个问题使用按位 `&` 运算符， `&` 对其操作数进行运算，并将其视为二进制值，然后执行与运算。

```
function isEven(num) {
  if (num & 1) {
    return false
  } else {
    return true
  }
}
```

```
0  二进制数是  000
1  二进制数是  001
2  二进制数是  010
3  二进制数是  011
4  二进制数是  100
5  二进制数是  101
6  二进制数是  110
7  二进制数是  111
```

以此类推...

与运算的规则如下:

a	b	a & b
0	0	0
0	1	0
1	1	1

因此，当我们执行 `console.log(5&1)` 这个表达式时，结果为 `1` 。首先， `&` 运算符将两个数字都转换为二进制，因此 `5` 变为 `101` ， `1` 变为 `001` 。

然后，它使用按位怀运算符比较每个位（ `0` 和 `1` ）。 `101&001` ，从表中可以看出，如果 `a & b` 为 `1` ，所以 `5&1` 结果为 `1` 。

101 & 001
101
001
001

- 首先我们比较最左边的 `1&0` ，结果是 `0` 。

- 然后我们比较中间的 `0&amp;0` ，结果是 `0` 。
- 然后我们比较最后 `1&amp;1` ，结果是 `1` 。
- 最后，得到一个二进制数 `001` ，对应的十进制数，即 `1` 。

  由此我们也可以算出 `console.log(4 & 1)` 结果为 `0` 。知道 `4` 的最后一位是 `0` ，而 `0 & 1` 将是 `0` 。如果你很难理解这一点，我们可以使用递归函数来解决此问题。

-
```
function isEven(num) {
  if (num < 0 || num === 1) return false;
  if (num == 0) return true;
  return isEven(num - 2);
}
```

59. 如何检查对象中是否存在某个属性？

检查对象中是否存在属性有三种方法。

第一种使用 `in` 操作符号：

```
const o = {
  "prop" : "bwahahah",
  "prop2" : "hweasa"
};

console.log("prop" in o); // true
console.log("prop1" in o); // false
```

第二种使用 `hasOwnProperty` 方法， `hasOwnProperty()` 方法会返回一个布尔值，指示对象自身属性中是否具有指定的属性（也就是，是否有指定的键）。

```
console.log(o.hasOwnProperty("prop2")); // true
console.log(o.hasOwnProperty("prop1")); // false
```

第三种使用括号符号 `obj["prop"]` 。如果属性存在，它将返回该属性的值，否则将返回 `undefined` 。

```
console.log(o["prop"]); // "bwahahah"
console.log(o["prop1"]); // undefined
```

60. AJAX 是什么？

即异步的 **JavaScript 和 XML**，是一种用于创建快速动态网页的技术，传统的网页（不使用 AJAX）如果需要更新内容，必需重载整个网页面。使用**AJAX**则不需要加载更新整个网页，实现部分内容更新

用到AJAX的技术：

- **HTML** - 网页结构
- **CSS** - 网页的样式
- **JavaScript** - 操作网页的行为和更新DOM
- **XMLHttpRequest API** - 用于从服务器发送和获取数据
- **PHP，Python，Nodejs** - 某些服务器端语言

61. 如何在 JS 中创建对象？

使用对象字面量：

```
const o = {
  name: "前端小智",
  greeting() {
    return `Hi，我是${this.name}`;
  }
};

o.greeting(); // "Hi，我是前端小智"
```

使用构造函数：

```
function Person(name) {
   this.name = name;
}

Person.prototype.greeting = function () {
   return `Hi，我是${this.name}`;
}

const mark = new Person("前端小智");

mark.greeting(); // "Hi，我是前端小智"
```

使用 Object.create 方法：

```
const n = {
   greeting() {
      return `Hi，我是${this.name}`;
   }
};

const o = Object.create(n);
o.name = "前端小智";
```

62. Object.seal 和 Object.freeze 方法之间有什么区别？

Object.freeze()

`Object.freeze()` 方法可以冻结一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改 。`freeze()` 返回和传入的参数相同的对象。

Object.seal()

```
Object.seal()方法封闭一个对象，阻止添加新属性并将所有现有属性标记为不可配置。当前属性的值只要可
```

方法的相同点：

1. ES5新增。
2. 对象不可能扩展，也就是不能再添加新的属性或者方法。
3. 对象已有属性不允许被删除。
4. 对象属性特性不可以重新配置。

方法不同点：

- `Object.seal` 方法生成的密封对象，如果属性是可写的，那么可以修改属性值。
  * `Object.freeze` 方法生成的冻结对象，属性都是不可写的，也就是属性值无法更改。

63. `in` 运算符和 `Object.hasOwnProperty` 方法有什么区别？

hasOwnPropert方法

`hasOwnPropert()` 方法返回值是一个布尔值，指示对象自身属性中是否具有指定的属性，因此这个方法会忽略掉那些从原型链上继承到的属性。

看下面的例子：

```
Object.prototype.phone= '15345025546';

let obj = {
    name: '前端小智',
    age: '28'
}
console.log(obj.hasOwnProperty('phone')) // false
console.log(obj.hasOwnProperty('name')) // true
```

可以看到，如果在函数原型上定义一个变量 `phone` ， `hasOwnProperty` 方法会直接忽略掉。

in 运算符

如果指定的属性在指定的对象或其原型链中，则 `in` 运算符返回 `true` 。

还是用上面的例子来演示：

```
console.log('phone' in obj) // true
```

可以看到 `in` 运算符会检查它或者其原型链是否包含具有指定名称的属性。

64. 有哪些方法可以处理 JS 中的异步代码？

- 回调
- Promise
- async/await
- 还有一些库：async.js，bluebird，q，co

65. 函数表达式和函数声明之间有什么区别？

看下面的例子：

```
hoistedFunc();
notHoistedFunc();

function hoistedFunc(){
  console.log("注意：我会被提升");
}

var notHoistedFunc = function(){
  console.log("注意：我没有被提升");
}
```

`notHoistedFunc` 调用抛出异常：`Uncaught TypeError: notHoistedFunc is not a function`，而 `hoistedFunc` 调用不会，因为 `hoistedFunc` 会被提升到作用域的顶部，而 `notHoistedFunc` 不会。

66. 调用函数，可以使用哪些方法？

在 JS 中有4种方法可以调用函数。

作为函数调用——如果一个函数没有作为方法、构造函数、`apply`、`call` 调用时，此时 `this` 指向的是 `window` 对象（非严格模式）

```
//Global Scope

function add(a,b){
  console.log(this);
  return a + b;
}

add(1,5); // 打印 "window" 对象和 6

const o = {
  method(callback){
    callback();
  }
}

o.method(function (){
```

```
    console.log(this); // 打印 "window" 对象
});
```

作为方法调用——如果一个对象的属性有一个函数的值，我们就称它为**方法**。调用该方法时，该方法的 `this` 值指向该对象。

```
const details = {
  name : "Marko",
  getName(){
    return this.name;
  }
}

details.getName(); // Marko
```

作为构造函数的调用-如果在函数之前使用 `new` 关键字调用了函数，则该函数称为 `构造函数` 。构造函数里面会默认创建一个空对象，并将 `this` 指向该对象。

```
function Employee(name, position, yearHired) {
  // 创建一个空对象 {}
  // 然后将空对象分配给“this”关键字
  // this = {};
  this.name = name;
  this.position = position;
  this.yearHired = yearHired;
  // 如果没有指定 return ,这里会默认返回 this
};

const emp = new Employee("Marko Polo", "Software Developer", 2017);
```

使用 apply 和 call 方法调用——如果我们想显式地指定一个函数的 `this` 值，我们可以使用这些方法，这些方法对所有函数都可用。

```
const obj1 = {
 result:0
};

const obj2 = {
 result:0
};


function reduceAdd(){
   let result = 0;
   for(let i = 0, len = arguments.length; i < len; i++){
     result += arguments[i];
   }
   this.result = result;
}


reduceAdd.apply(obj1, [1, 2, 3, 4, 5]);  // reduceAdd 函数中的 this 对象将是 obj1
reduceAdd.call(obj2, 1, 2, 3, 4, 5); // reduceAdd 函数中的 this 对象将是 obj2
```

67. 什么是缓存及它有什么作用?

缓存是建立一个函数的过程，这个函数能够记住之前计算的结果或值。使用缓存函数是为了避免在最后一次使用相同参数的计算中已经执行的函数的计算。这节省了时间，但也有

不利的一面，即我们将消耗更多的内存来保存以前的结果。

68. 手动实现缓存方法]

```
function memoize(fn) {
  const cache = {};
  return function (param) {
    if (cache[param]) {
      console.log('cached');
      return cache[param];
    } else {
      let result = fn(param);
      cache[param] = result;
      console.log(`not cached`);
      return result;
    }
  }
}

const toUpper = (str ="")=> str.toUpperCase();

const toUpperMemoized = memoize(toUpper);

toUpperMemoized("abcdef");
toUpperMemoized("abcdef");
```

这个缓存函数适用于接受一个参数。我们需要改变下，让它接受多个参数。

```
const slice = Array.prototype.slice;
function memoize(fn) {
  const cache = {};
  return (...args) => {
    const params = slice.call(args);
    console.log(params);
    if (cache[params]) {
      console.log('cached');
      return cache[params];
    } else {
      let result = fn(...args);
      cache[params] = result;
      console.log(`not cached`);
      return result;
    }
  }
}
const makeFullName = (fName, lName) => `${fName} ${lName}`;
const reduceAdd = (numbers, startingValue = 0) => numbers.reduce((total, cur) => total

const memoizedMakeFullName = memoize(makeFullName);
const memoizedReduceAdd = memoize(reduceAdd);

memoizedMakeFullName("Marko", "Polo");
memoizedMakeFullName("Marko", "Polo");

memoizedReduceAdd([1, 2, 3, 4, 5], 5);
memoizedReduceAdd([1, 2, 3, 4, 5], 5);
```

69. 为什么typeof null 返回 object? 如何检查一个值是否为 null?

`typeof null == 'object'` 总是返回 `true`，因为这是自 JS 诞生以来 `null` 的实现。曾经有人提出将 `typeof null == 'object'` 修改为 `typeof null == 'null'`，但是被拒绝了，因为这将导致更多的**bug**。

我们可以使用严格相等运算符 `===` 来检查值是否为 `null`。

```
function isNull(value){
  return value === null;
}
```

70. new 关键字有什么作用？

`new` 关键字与构造函数一起使用以创建对象：

```
function Employee(name, position, yearHired) {
  this.name = name;
  this.position = position;
  this.yearHired = yearHired;
};

const emp = new Employee("Marko Polo", "Software Developer", 2017);
```

`new` 关键字做了 4 件事：

- 创建空对象 `{}`
- 将空对象分配给 `this` 值
- 将空对象的 **`proto`** 指向构造函数的 `prototype`
- 如果没有使用显式 `return` 语句，则返回 `this`

看下面事例：

```
function Person() {
 this.name = '前端小智'
}
```

根据上面描述的，`new Person()` 做了：

- 创建一个空对象：`var obj = {}`
- 将空对象分配给 `this` 值：this = obj
- 将空对象的 **`proto__` 指向构造函数的 `prototype`:`this.__proto__ = Person().prototype**
- 返回 `this`：`return this`

71. 什么时候不使用箭头函数？说出三个或更多的例子？

不应该使用箭头函数一些情况：

- 当想要函数被提升时(箭头函数是匿名的)

- 要在函数中使用 `this/arguments` 时，由于箭头函数本身不具有 `this/arguments`，因此它们取决于外部上下文
- 使用命名函数(箭头函数是匿名的)
- 使用函数作为构造函数时(箭头函数没有构造函数)
- 当想在对象字面是以将函数作为属性添加并在其中使用对象时，因为咱们无法访问 `this` 即对象本身。

72. Object.freeze() 和 const 的区别是什么？]

`const` 和 `Object.freeze` 是两个完全不同的概念。

`const` 声明一个只读的变量，一旦声明，常量的值就不可改变：

```
const person = {
    name: "Leonardo"
};
let animal = {
    species: "snake"
};
person = animal; // ERROR "person" is read-only
```

`Object.freeze` 适用于值，更具体地说，适用于对象值，它使对象不可变，即不能更改其属性。

```
let person = {
    name: "Leonardo"
};
let animal = {
    species: "snake"
};
Object.freeze(person);
person.name = "Lima"; //TypeError: Cannot assign to read only property 'name' of object
console.log(person);
```

73. 如何在 JS 中“深冻结”对象？

如果咱们想要确保对象被深冻结，就必须创建一个递归函数来冻结对象类型的每个属性：

没有深冻结

```
let person = {
    name: "Leonardo",
    profession: {
        name: "developer"
    }
};
Object.freeze(person);
person.profession.name = "doctor";
console.log(person); //output { name: 'Leonardo', profession: { name: 'doctor' } }
```

深冻结

```
function deepFreeze(object) {
    let propNames = Object.getOwnPropertyNames(object);
    for (let name of propNames) {
        let value = object[name];
        object[name] = value && typeof value === "object" ?
            deepFreeze(value) : value;
    }
    return Object.freeze(object);
}
let person = {
    name: "Leonardo",
    profession: {
        name: "developer"
    }
};
deepFreeze(person);
person.profession.name = "doctor"; // TypeError: Cannot assign to read only property 'r
```

74. `Iterator`是什么，有什么作用？

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

`Iterator` 的作用有三个：

1. 为各种数据结构，提供一个统一的、简便的访问接口；
2. 使得数据结构的成员能够按某种次序排列；
3. 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。

遍历过程：

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。
3. 第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。
4. 不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用 `next` 方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含 `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。

```
//obj就是可遍历的，因为它遵循了Iterator标准，且包含[Symbol.iterator]方法，方法函数也符合标准
//obj.[Symbol.iterator]() 就是Iterator遍历器
let obj = {
  data: [ 'hello', 'world' ],
  [Symbol.iterator]() {
    const self = this;
    let index = 0;
    return {
      next() {
        if (index < self.data.length) {
          return {
            value: self.data[index++],
            done: false
          };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};
```

75. `Generator` 函数是什么，有什么作用？

如果说 JavaScrip 是 ECMAScript 标准的一种具体实现、 `Iterator` 遍历器是 `Iterator` 的具体实现，那么 `Generator` 函数可以说是 `Iterator` 接口的具体实现方式。

执行 `Generator` 函数会返回一个遍历器对象，每一次 `Generator` 函数里面的yield都相当一次遍历器对象的 `next()` 方法，并且可以通过 `next(value)` 方法传入自定义的 `value` ,来改变 `Generator` 函数的行为。

`Generator` 函数可以通过配合Thunk 函数更轻松更优雅的实现异步编程和控制流管理。

原文：

https://dev.to/macmacky/70-javascript-interview-questions-5gfi#1-whats-the-difference-between-undefined-and-null

---

交流

看完这几道 JavaScript 面试题，让你与考官对答如流（中）

看完这几道 JavaScript 面试题，让你与考官对答如流（上）

如何使用SASS编写可重用的CSS

小智也在看 | 有哪些令人浑身发抖的故事？

面试 10

面试 · 目录

上一篇

看完这几道 JavaScript 面试题，让你与考官对答如流（中）

下一篇

50 个JS 必须懂的面试题为你助力金九银十

喜欢此内容的人还喜欢

他因提及其他编程语言而被禁止

大迁世界

告别轮询，手把手教你封装WebSocket消息推送！

大迁世界