看完这几道 JavaScript 面试题，让你与考官对答如流（中）

原创 前端小智 大迁世界 2020-01-09 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，上篇：

看完这几道 JavaScript 面试题，让你与考官对答如流（上）

- 26. 什么是IIFE，它的用途是什么？
- 27. Function.prototype.apply方法的用途是什么？
- 28. Function.prototype.call方法的用途是什么？
- 29. Function.prototype.apply 和 Function.prototype.call 之间有什么区别？
- 30. Function.prototype.bind的用途是什么？
- 31. 什么是函数式编程？JavaScript的哪些特性使其成为函数式语言的候选语言？
- 32. 什么是高阶函数？
- 33. 为什么函数被称为一等公民？
- 34. 手动实现`Array.prototype.map`方法
- 35. 手动实现`Array.prototype.filter`方法
- 35. 手动实现`Array.prototype.reduce`方法
- 37. arguments 的对象是什么？
- 38. 如何创建一个没有 prototype(原型) 的对象？
- 39. 为什么在调用这个函数时，代码中的`b`会变成一个全局变量？
- 40. ECMAScript是什么？
- 41. ES6或ECMAScript 2015有哪些新特性？
- 42. `var`,`let`和`const`的区别是什么
- 43. 什么是箭头函数？
- 44. 什么是类？
- 45. 什么是模板字符串？
- 46. 什么是对象解构？
- 47. 什么是 ES6 模块？
- 48. 什么是`Set`对象，它是如何工作的？
- 49. 什么是回调函数？
- 50. Promise 是什么？

26. 什么是 IIFE，它的用途是什么？

IIFE或立即调用的函数表达式是在创建或声明后将被调用或执行的函数。创建**IIFE的**语法是，将 `function (){}` 包裹在在括号 `()` 内，然后再用另一个括号 `()` 调用它，如：`(function(){})()`

```
(function(){
  ...
} ());

(function () {
  ...
})();

(function named(params) {
  ...
})();

(() => {

});

(function (global) {
  ...
})(window);

const utility = (function () {
  return {
    ...
  }
})
```

这些示例都是有效的**IIFE**。倒数第二个救命表明我们可以将参数传递给**IIFE**函数。最后一个示例表明，我们可以将 `IIFE` 的结果保存到变量中，以便稍后使用。

IIFE的一个主要作用是避免与全局作用域内的其他变量命名冲突或污染全局命名空间，来个例子。

```
<script src="https://cdnurl.com/somelibrary.js"></script>
```

假设我们引入了一个 `omelibr.js` 的链接，它提供了一些我们在代码中使用的全局函数，但是这个库有两个方法我们没有使用： `createGraph` 和 `drawGraph` ，因为这些方法都有 `bug` 。我们想实现自己的 `createGraph` 和 `drawGraph` 方法。

解决此问题的一种方法是直接覆盖：

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
   function createGraph() {
      // createGraph logic here
   }
   function drawGraph() {
      // drawGraph logic here
   }
</script>
```

当我们使用这个解决方案时，我们覆盖了库提供给我们的那两个方法。

另一种方式是我们自己改名称：

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
   function myCreateGraph() {
      // createGraph logic here
```

```
    }
    function myDrawGraph() {
        // drawGraph logic here
    }
</script>
```

当我们使用这个解决方案时，我们把那些函数调用更改为新的函数名。

还有一种方法就是使用**IIFE**：

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
    const graphUtility = (function () {
        function createGraph() {
            // createGraph logic here
        }
        function drawGraph() {
            // drawGraph logic here
        }
        return {
            createGraph,
            drawGraph
        }
    })
</script>
```

在此解决方案中，我们要声明了 `graphUtility` 变量，用来保存**IIFE**执行的结果，该函数返回一个包含两个方法 `createGraph` 和 `drawGraph` 的对象。

IIFE 还可以用来解决一个常见的面试题：

```
var li = document.querySelectorAll('.list-group > li');
for (var i = 0, len = li.length; i < len; i++) {
    li[i].addEventListener('click', function (e) {
        console.log(i);
    })
```

假设我们有一个带有 `list-group` 类的 `ul` 元素，它有 `5` 个 `li` 子元素。当我们单击单个 `li` 元素时，打印对应的下标值。但在此外上述代码不起作用，这里每次点击 `li` 打印 `i` 的值都是 `5` ，这是由于闭包的原因。

闭包只是函数记住其当前作用域，父函数作用域和全局作用域的变量引用的能力。当我们在全局作用域内使用 `var` 关键字声明变量时，就创建全局变量 `i` 。因此，当我们单击 `li` 元素时，它将打印 `5` ，因为这是稍后在回调函数中引用它时 `i` 的值。

使用 **IIFE** 可以解决此问题：

```
var li = document.querySelectorAll('.list-group > li');
for (var i = 0, len = li.length; i < len; i++) {
    (function (currentIndex) {
        li[currentIndex].addEventListener('click', function (e) {
            console.log(currentIndex);
        })
    })(i);
}
```

该解决方案之所以行的通，是因为**IIFE**会为每次迭代创建一个新的作用域，我们捕获 `i` 的值并将其传递给 `currentIndex` 参数，因此调用**IIFE**时，每次迭代的 `currentIndex` 值都是不同的。

27. Function.prototype.apply 方法的用途是什么?

`apply()` 方法调用一个具有给定this值的函数，以及作为一个数组（或类似数组对象）提供的参数。

```
const details = {
  message: 'Hello World!'
};

function getMessage(){
  return this.message;
}

getMessage.apply(details); // 'Hello World!'
```

> `call()` 方法的作用和 `apply()` 方法类似，区别就是 `call()` 方法接受的是参数列表，而 `apply()` 方法接受的是一个参数数组。

```
const person = {
  name: "Marko Polo"
};

function greeting(greetingMessage) {
  return `${greetingMessage} ${this.name}`;
}

greeting.apply(person, ['Hello']); // "Hello Marko Polo!"
```

28. `Function.prototype.call` 方法的用途是什么?

`call()` 方法使用一个指定的 `this` 值和单独给出的一个或多个参数来调用一个函数。

```
const details = {
  message: 'Hello World!'
};

function getMessage(){
  return this.message;
}

getMessage.call(details); // 'Hello World!'
```

注意：该方法的语法和作用与 `apply()` 方法类似，只有一个区别，就是 `call()` 方法接受的是一个参数列表，而 `apply()` 方法接受的是一个包含多个参数的数组。

```
const person = {
  name: "Marko Polo"
};

function greeting(greetingMessage) {
```

```
  return `${greetingMessage} ${this.name}`;
}

greeting.call(person, 'Hello'); // "Hello Marko Polo!"
```

29. Function.prototype.apply 和 Function.prototype.call 之间有什么区别？

`apply()` 方法可以在使用一个指定的 `this` 值和一个参数数组（或类数组对象）的前提下调用某个函数或方法。 `call()` 方法类似于 `apply()` ，不同之处仅仅是 `call()` 接受的参数是参数列表。

```
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]); // 15
reduceAdd.call(obj2, 1, 2, 3, 4, 5); // 15
```

30. Function.prototype.bind 的用途是什么？

`bind()` 方法创建一个新的函数，在 `bind()` 被调用时，这个新函数的 `this` 被指定为 `bind()` 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

```
import React from 'react';

class MyComponent extends React.Component {
     constructor(props){
          super(props);
          this.state = {
             value : ""
          }
          this.handleChange = this.handleChange.bind(this);
          // 将 “handleChange” 方法绑定到 “MyComponent” 组件
     }

     handleChange(e){
       //do something amazing here
     }

     render(){
        return (
              <>
                <input type={this.props.type}
                        value={this.state.value}
                     onChange={this.handleChange}
```

```
                />
            </>
        )
    }
}
```

31. 什么是函数式编程？JavaScript 的哪些特性使其成为函数式语言的候选语言？

函数式编程（通常缩写为FP）是通过编写纯函数，避免共享状态、可变数据、副作用 来构建软件的过程。数式编程是声明式 的而不是命令式 的，应用程序的状态是通过纯函数流动的。与面向对象编程形成对比，面向对象中应用程序的状态通常与对象中的方法共享和共处。

函数式编程是一种编程范式 ，这意味着它是一种基于一些基本的定义原则（如上所列）思考软件构建的方式。当然，编程范示的其他示例也包括面向对象编程和过程编程。

函数式的代码往往比命令式或面向对象的代码更简洁，更可预测，更容易测试 - 但如果不熟悉它以及与之相关的常见模式，函数式的代码也可能看起来更密集杂乱，并且 相关文献对新人来说是不好理解的。

JavaScript支持闭包和高阶函数是函数式编程语言的特点。

32. 什么是高阶函数?

高阶函数只是将函数作为参数或返回值的函数。

```
function higherOrderFunction(param,callback){
    return callback(param);
}
```

33. 为什么函数被称为一等公民?

在JavaScript中，函数不仅拥有一切传统函数的使用方式（声明和调用），而且可以做到像简单值一样赋值（`var func = function(){}`）、传参（`function func(x,callback){callback();}`）、返回（`function(){return function(){}}`），这样的函数也称之为**第一级函数（First-class Function）**。不仅如此，JavaScript中的函数还充当了类的构造函数的作用，同时又是一个 `Function` 类的实例(instance)。这样的多重身份让JavaScript的函数变得非常重要。

34. 手动实现 `Array.prototype.map 方法`

`map()` 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果。

```
function map(arr, mapCallback) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof mapCallback !== 'function') {
    return [];
  } else {
    let result = [];
    // 每次调用此函数时，我们都会创建一个 result 数组
    // 因为我们不想改变原始数组。
    for (let i = 0, len = arr.length; i < len; i++) {
      result.push(mapCallback(arr[i], i, arr));
      // 将 mapCallback 返回的结果 push 到 result 数组中
    }
    return result;
  }
}
```

35. 手动实现`Array.prototype.filter`方法

`filter()` 方法创建一个新数组，其包含通过所提供函数实现的测试的所有元素。

```
function filter(arr, filterCallback) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof filterCallback !== 'function')
  {
    return [];
  } else {
    let result = [];
     // 每次调用此函数时，我们都会创建一个 result 数组
     // 因为我们不想改变原始数组。
    for (let i = 0, len = arr.length; i < len; i++) {
      // 检查 filterCallback 的返回值是否是真值
      if (filterCallback(arr[i], i, arr)) {
      // 如果条件为真，则将数组元素 push 到 result 中
        result.push(arr[i]);
      }
    }
    return result; // return the result array
  }
}
```

36. 手动实现`Array.prototype.reduce`方法

`reduce()` 方法对数组中的每个元素执行一个由您提供的 `reducer` 函数(升序执行)，将其结果汇总为单个返回值。

```
function reduce(arr, reduceCallback, initialValue) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof reduceCallback !== 'function')
  {
    return [];
  } else {
    // 如果没有将initialValue传递给该函数，我们将使用第一个数组项作为initialValue
    let hasInitialValue = initialValue !== undefined;
    let value = hasInitialValue ? initialValue : arr[0];
   `

    // 如果有传递 initialValue，则索引从 1 开始，否则从 0 开始
    for (let i = hasInitialValue ? 0 : 1, len = arr.length; i < len; i++) {
      value = reduceCallback(value, arr[i], i, arr);
    }
    return value;
```

```
  }
}
```

37. arguments 的对象是什么?

arguments 对象是函数中传递的参数值的集合。它是一个类似数组的对象，因为它有一个length属性，我们可以使用数组索引表示法 arguments[1] 来访问单个值，但它没有数组中的内置方法，如： forEach 、 reduce 、 filter 和 map 。

我们可以使用 Array.prototype.slice 将 arguments 对象转换成一个数组。

```
function one() {
  return Array.prototype.slice.call(arguments);
}
```

注意:箭头函数中没有 arguments 对象。

```
function one() {
  return arguments;
}
const two = function () {
  return arguments;
}
const three = function three() {
  return arguments;
}

const four = () => arguments;

four(); // Throws an error  - arguments is not defined
```

当我们调用函数 four 时，它会抛出一个 ReferenceError: arguments is not defined error 。使用 rest 语法，可以解决这个问题。

```
const four = (...args) => args;
```

这会自动将所有参数值放入数组中。

38. 如何创建一个没有 prototype(原型)的对象?

我们可以使用 Object.create 方法创建没有原型的对象。

```
const o1 = {};
console.log(o1.toString()); // [object Object]

const o2 = Object.create(null);
console.log(o2.toString());
// throws an error o2.toString is not a function
```

39. 为什么在调用这个函数时，代码中的`b`会变成一个全局变量?

```
function myFunc() {
  let a = b = 0;
}

myFunc();
```

原因是赋值运算符是从右到左的求值的。这意味着当多个赋值运算符出现在一个表达式中时，它们是从右向左求值的。所以上面代码变成了这样：

```
function myFunc() {
  let a = (b = 0);
}

myFunc();
```

首先，表达式 `b = 0` 求值，在本例中 `b` 没有声明。因此，JS引擎在这个函数外创建了一个全局变量 `b` ，之后表达式 `b = 0` 的返回值为 `0` ，并赋给新的局部变量 `a` 。

我们可以通过在赋值之前先声明变量来解决这个问题。

```
function myFunc() {
  let a,b;
  a = b = 0;
}
myFunc();
```

40. ECMAScript 是什么?

ECMAScript 是编写脚本语言的标准，这意味着JavaScript遵循ECMAScript标准中的规范变化，因为它是JavaScript的蓝图。

ECMAScript 和 Javascript，本质上都跟一门语言有关，一个是语言本身的名字，一个是语言的约束条件
只不过发明JavaScript的那个人（Netscape公司），把东西交给了ECMA（European Computer Manufacturers Association），这个人规定一下他的标准，因为当时有java语言了，又想强调这个东西是让ECMA这个人定的规则，所以就这样一个神奇的东西诞生了，这个东西的名称就叫做ECMAScript。

javaScript = ECMAScript + DOM + BOM（自认为是一种广义的JavaScript）

ECMAScript说什么JavaScript就得做什么！

JavaScript（狭义的JavaScript）做什么都要问问ECMAScript我能不能这样干！如果不能我就错了！能我就是对的！

——突然感觉JavaScript好没有尊严，为啥要搞个人出来约束自己，

那个人被创造出来也好委屈，自己被创造出来完全是因为要约束JavaScript。

41. ES6或ECMAScript 2015有哪些新特性?

- 箭头函数
- 类
- 模板字符串
- 加强的对象字面量
- 对象解构
- Promise
- 生成器
- 模块
- Symbol
- 代理
- Set
- 函数默认参数
- rest 和展开
- 块作用域

42. `var`,`let`和`const`的区别是什么？

var 声明的变量会挂载在 window 上，而 let 和 const 声明的变量不会：

```
var a = 100;
console.log(a,window.a);    // 100 100

let b = 10;
console.log(b,window.b);    // 10 undefined

const c = 1;
console.log(c,window.c);    // 1 undefined
```

var 声明变量存在变量提升， let 和 const 不存在变量提升：

```
console.log(a); // undefined  ===>  a已声明还没赋值，默认得到undefined值
var a = 100;

console.log(b); // 报错：b is not defined  ===> 找不到b这个变量
let b = 10;

console.log(c); // 报错：c is not defined  ===> 找不到c这个变量
const c = 10;
```

let 和 const 声明形成块作用域

```
if(1){
  var a = 100;
  let b = 10;
}

console.log(a); // 100
console.log(b)  // 报错：b is not defined  ===> 找不到b这个变量

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

if(1){
  var a = 100;
  const c = 1;
}
```

```
console.log(a); // 100
console.log(c)  // 报错：c is not defined  ===> 找不到c这个变量
```

同一作用域下 let 和 const 不能声明同名变量，而 var 可以

```
var a = 100;
console.log(a); // 100

var a = 10;
console.log(a); // 10
-------------------------------------
let a = 100;
let a = 10;

//  控制台报错：Identifier 'a' has already been declared  ===> 标识符a已经被声明了。
```

暂存死区

```
var a = 100;

if(1){
    a = 10;
    //在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，
    // 而这时，还未到声明时候，所以控制台Error:a is not defined
    let a = 1;
}
```

const

```
/*
*     1、一旦声明必须赋值,不能使用null占位。
*
*     2、声明后不能再修改
*
*     3、如果声明的是复合类型数据，可以修改其属性
*
* */

const a = 100;

const list = [];
list[0] = 10;
console.log(list);    // [10]

const obj = {a:100};
obj.name = 'apple';
obj.a = 10000;
console.log(obj);    // {a:10000,name:'apple'}
```

43. 什么是箭头函数？

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的 `this` ， `arguments` ， `super` 或 `new.target` 。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

```
//ES5 Version
var getCurrentDate = function (){
  return new Date();
}
```

```
//ES6 Version
const getCurrentDate = () => new Date();
```

在本例中，ES5 版本中有 `function(){}` 声明和 `return` 关键字，这两个关键字分别是创建函数和返回值所需要的。在箭头函数版本中，我们只需要 `()` 括号，不需要 `return` 语句，因为如果我们只有一个表达式或值需要返回，箭头函数就会有一个隐式的返回。

```
//ES5 Version
function greet(name) {
  return 'Hello ' + name + '!';
}

//ES6 Version
const greet = (name) => `Hello ${name}`;
const greet2 = name => `Hello ${name}`;
```

我们还可以在箭头函数中使用与函数表达式和函数声明相同的参数。如果我们在一个箭头函数中有一个参数，则可以省略括号。

```
const getArgs = () => arguments

const getArgs2 = (...rest) => rest
```

箭头函数不能访问 `arguments` 对象。所以调用第一个 `getArgs` 函数会抛出一个错误。相反，我们可以使用**rest**参数来获得在箭头函数中传递的所有参数。

```
const data = {
  result: 0,
  nums: [1, 2, 3, 4, 5],
  computeResult() {
    // 这里的“this”指的是“data”对象
    const addAll = () => {
      return this.nums.reduce((total, cur) => total + cur, 0)
    };
    this.result = addAll();
  }
};
```

箭头函数没有自己的 `this` 值。它捕获词法作用域函数的 `this` 值，在此示例中，`addAll` 函数将复制 `computeResult` 方法中的 `this` 值，如果我们在全局作用域声明箭头函数，则 `this` 值为 `window` 对象。

44. 什么是类？

`类(class)` 是在 JS 中编写构造函数的新方法。它是使用构造函数的语法糖，在底层中使用仍然是原型和基于原型的继承。

```
//ES5 Version
function Person(firstName, lastName, age, address){
   this.firstName = firstName;
   this.lastName = lastName;
   this.age = age;
   this.address = address;
}
```

```
Person.self = function(){
  return this;
}

Person.prototype.toString = function(){
  return "[object Person]";
}

Person.prototype.getFullName = function (){
  return this.firstName + " " + this.lastName;
}

//ES6 Version
class Person {
     constructor(firstName, lastName, age, address){
         this.lastName = lastName;
         this.firstName = firstName;
         this.age = age;
         this.address = address;
     }

     static self() {
        return this;
     }

     toString(){
        return "[object Person]";
     }

     getFullName(){
        return `${this.firstName} ${this.lastName}`;
     }
}
```

重写方法并从另一个类继承。

```
//ES5 Version
Employee.prototype = Object.create(Person.prototype);

function Employee(firstName, lastName, age, address, jobTitle, yearStarted) {
  Person.call(this, firstName, lastName, age, address);
  this.jobTitle = jobTitle;
  this.yearStarted = yearStarted;
}

Employee.prototype.describe = function () {
  return `I am ${this.getFullName()} and I have a position of ${this.jobTitle} and I st
}

Employee.prototype.toString = function () {
  return "[object Employee]";
}

//ES6 Version
class Employee extends Person { //Inherits from "Person" class
  constructor(firstName, lastName, age, address, jobTitle, yearStarted) {
    super(firstName, lastName, age, address);
    this.jobTitle = jobTitle;
    this.yearStarted = yearStarted;
  }

  describe() {
    return `I am ${this.getFullName()} and I have a position of ${this.jobTitle} and I
  }

  toString() { // Overriding the "toString" method of "Person"
    return "[object Employee]";
  }
}
```

所以我们要怎么知道它在内部使用原型？

```
class Something {

}

function AnotherSomething(){

}
const as = new AnotherSomething();
const s = new Something();

console.log(typeof Something); // "function"
console.log(typeof AnotherSomething); // "function"
console.log(as.toString()); // "[object Object]"
console.log(as.toString()); // "[object Object]"
console.log(as.toString === Object.prototype.toString); // true
console.log(s.toString === Object.prototype.toString); // true
```

45. 什么是模板字符串？

模板字符串是在 JS 中创建字符串的一种新方法。我们可以通过使用反引号使模板字符串化。

```
//ES5 Version
var greet = 'Hi I\'m Mark';
```

```
//ES6 Version
let greet = `Hi I'm Mark`;
```

在 ES5 中我们需要使用一些转义字符来达到多行的效果，在模板字符串不需要这么麻烦：

```
//ES5 Version
var lastWords = '\n'
  + '   I  \n'
  + '   Am  \n'
  + 'Iron Man \n';


//ES6 Version
let lastWords = `
    I
    Am
  Iron Man
`;
```

在ES5版本中，我们需要添加 `\n` 以在字符串中添加新行。在模板字符串中，我们不需要这样做。

```
//ES5 Version
function greet(name) {
  return 'Hello ' + name + '!';
}


//ES6 Version
function greet(name) {
  return `Hello ${name} !`;
}
```

在 ES5 版本中，如果需要在字符串中添加表达式或值，则需要使用 `+` 运算符。在模板字符串s中，我们可以使用 `${expr}` 嵌入一个表达式，这使其比 ES5 版本更整洁。

46. 什么是对象解构？

对象析构是从对象或数组中获取或提取值的一种新的、更简洁的方法。假设有如下的对象：

```
const employee = {
  firstName: "Marko",
  lastName: "Polo",
  position: "Software Developer",
  yearHired: 2017
};
```

从对象获取属性，早期方法是创建一个与对象属性同名的变量。这种方法很麻烦，因为我们要为每个属性创建一个新变量。假设我们有一个大对象，它有很多属性和方法，用这种方法提取属性会很麻烦。

```
var firstName = employee.firstName;
var lastName = employee.lastName;
```

```
var position = employee.position;
var yearHired = employee.yearHired;
```

使用解构方式语法就变得简洁多了：

```
{ firstName, lastName, position, yearHired } = employee;
```

我们还可以为属性取别名：

```
let { firstName: fName, lastName: lName, position, yearHired } = employee;
```

当然如果属性值为 `undefined` 时，我们还可以指定默认值：

```
let { firstName = "Mark", lastName: lName, position, yearHired } = employee;
```

47. 什么是 ES6 模块？

模块使我们能够将代码基础分割成多个文件，以获得更高的可维护性，并且避免将所有代码放在一个大文件中。在 ES6 支持模块之前，有两个流行的模块。

- **CommonJS-Node.js**
- AMD（异步模块定义）-**浏览器**

基本上，使用模块的方式很简单，`import` 用于从另一个文件中获取功能或几个功能或值，同时 `export` 用于从文件中公开功能或几个功能或值。

导出

使用 ES5（CommonJS）

```
// 使用 ES5 CommonJS - helpers.js
exports.isNull = function (val) {
  return val === null;
}

exports.isUndefined = function (val) {
  return val === undefined;
}

exports.isNullOrUndefined = function (val) {
  return exports.isNull(val) || exports.isUndefined(val);
}
```

使用 ES6 模块

```
// 使用 ES6 Modules - helpers.js
export function isNull(val){
  return val === null;
}

export function isUndefined(val) {
  return val === undefined;
```

```
}

export function isNullOrUndefined(val) {
  return isNull(val) || isUndefined(val);
}
```

在另一个文件中导入函数

```
// 使用 ES5 (CommonJS) - index.js
const helpers = require('./helpers.js'); // helpers is an object
const isNull = helpers.isNull;
const isUndefined = helpers.isUndefined;
const isNullOrUndefined = helpers.isNullOrUndefined;

// or if your environment supports Destructuring
const { isNull, isUndefined, isNullOrUndefined } = require('./helpers.js');
-------------------------------------------------------

// ES6 Modules - index.js
import * as helpers from './helpers.js'; // helpers is an object

// or

import { isNull, isUndefined, isNullOrUndefined as isValid } from './helpers.js';

// using "as" for renaming named exports
```

在文件中导出单个功能或默认导出

使用 ES5 (CommonJS)

```
// 使用 ES5 (CommonJS) - index.js
class Helpers {
  static isNull(val) {
    return val === null;
  }

  static isUndefined(val) {
    return val === undefined;
  }

  static isNullOrUndefined(val) {
    return this.isNull(val) || this.isUndefined(val);
  }
}


module.exports = Helpers;
```

使用ES6 Modules

```
// 使用 ES6 Modules - helpers.js
class Helpers {
  static isNull(val) {
    return val === null;
  }

  static isUndefined(val) {
    return val === undefined;
  }

  static isNullOrUndefined(val) {
```

```
    return this.isNull(val) || this.isUndefined(val);
  }
}

export default Helpers
```

从另一个文件导入单个功能

使用ES5（CommonJS）

```
// 使用 ES5 (CommonJS) - index.js
const Helpers = require('./helpers.js');
console.log(Helpers.isNull(null));
```

使用 ES6 Modules

```
import Helpers from '.helpers.js'
console.log(Helpers.isNull(null));
```

48. 什么是`Set`对象，它是如何工作的？

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

我们可以使用 `Set` 构造函数创建 `Set` 实例。

```
const set1 = new Set();
const set2 = new Set(["a","b","c","d","d","e"]);
```

我们可以使用 `add` 方法向 `Set` 实例中添加一个新值，因为 `add` 方法返回 `Set` 对象，所以我们可以以链式的方式再次使用 `add` 。如果一个值已经存在于 `Set` 对象中，那么它将不再被添加。

```
set2.add("f");
set2.add("g").add("h").add("i").add("j").add("k").add("k");
// 后一个“k”不会被添加到set对象中，因为它已经存在了
```

我们可以使用 `has` 方法检查 `Set` 实例中是否存在特定的值。

```
set2.has("a") // true
set2.has("z") // true
```

我们可以使用 `size` 属性获得 `Set` 实例的长度。

```
set2.size // returns 10
```

可以使用 `clear` 方法删除 `Set` 中的数据。

```
set2.clear();
```

我们可以使用 `Set` 对象来删除数组中重复的元素。

```
const numbers = [1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 5];
const uniqueNums = [...new Set(numbers)]; // [1,2,3,4,5,6,7,8]
```

49. 什么是回调函数?

回调函数是一段可执行的代码段，它作为一个参数传递给其他的代码，其作用是在需要的时候方便调用这段（回调函数）代码。

在JavaScript中函数也是对象的一种，同样对象可以作为参数传递给函数，因此函数也可以作为参数传递给另外一个函数，这个作为参数的函数就是回调函数。

```
const btnAdd = document.getElementById('btnAdd');

btnAdd.addEventListener('click', function clickCallback(e) {
    // do something useless
});
```

在本例中，我们等待 `id` 为 `btnAdd` 的元素中的 `click` 事件，如果它被单击，则执行 `clickCallback` 函数。回调函数向某些数据或事件添加一些功能。

数组中的 `reduce` 、 `filter` 和 `map` 方法需要一个回调作为参数。回调的一个很好的类比是，当你打电话给某人，如果他们不接，你留下一条消息，你期待他们回调。调用某人或留下消息的行为是事件或数据，回调是你希望稍后发生的操作。

50. Promise 是什么?

Promise 是异步编程的一种解决方案：从语法上讲， `promise` 是一个对象，从它可以获取异步操作的消息；从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。 `promise` 有三种状态： `pending(等待态)` ， `fulfiled(成功态)` ， `rejected(失败态)` ；状态一旦改变，就不会再变。创造 `promise` 实例后，它会立即执行。

```
fs.readFile('somefile.txt', function (e, data) {
  if (e) {
    console.log(e);
  }
  console.log(data);
});
```

如果我们在回调内部有另一个异步操作，则此方法存在问题。我们将有一个混乱且不可读的代码。此代码称为**“回调地狱”**。

```
// 回调地狱
fs.readFile('somefile.txt', function (e, data) {
  //your code here
  fs.readdir('directory', function (e, files) {
    //your code here
    fs.mkdir('directory', function (e) {
      //your code here
    })
```

```
  })
})
```

如果我们在这段代码中使用 promise ，它将更易于阅读、理解和维护。

```
promReadFile('file/path')
  .then(data => {
    return promReaddir('directory');
  })
  .then(data => {
    return promMkdir('directory');
  })
  .catch(e => {
    console.log(e);
  })
```

promise 有三种不同的状态：

- pending：初始状态，完成或失败状态的前一个状态
- fulfilled：操作成功完成
- rejected：操作失败

pending 状态的 Promise 对象会触发 fulfilled/rejected 状态，在其状态处理方法中可以传入参数/失败信息。当操作成功完成时，**Promise** 对象的 then 方法就会被调用；否则就会触发 catch 。如：

```
const myFirstPromise = new Promise((resolve, reject) => {
    setTimeout(function(){
        resolve("成功!");
    }, 250);
});

myFirstPromise.then((data) => {
    console.log("Yay! " + data);
}).catch((e) => {...});
```

由于篇幅过长，我将此系列分成上中下三篇，下篇我们在见。

原文：

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

---

交流

小智也在看 | 有哪些令人浑身发抖的故事?

如何使用SASS编写可重用的CSS

【TS 演化史 -- 12】ES5/ES3 的生成器和迭代支持及 --checkJS选项下 .js 文件中的错误

【动画演示】JavaScript 引擎运行原理

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（上）

下一篇
看完这几道 JavaScript 面试题，让你与考官对答如流（下）

喜欢此内容的人还喜欢

90后程序员辞职搞灰产，一年获利超700万，结局很刑！
大迁世界

100 个鲜为人知的 CSS 技巧汇总整理合集
大迁世界

最近前端爆了？？？
大迁世界