75 道 JavaScript 面试题， 2.5 万字刷个够（含解析）

考题列表

考题列表

1. undefined 和 null 有什么区别？
2. && 运算符能做什么
3. || 运算符能做什么
4. 使用 + 或一元加运算符是将字符串转换为数字的最快方法吗？
5. DOM 是什么？
6. 什么是事件传播?
7. 什么是事件冒泡？
8. 什么是事件捕获？
9. event.preventDefault() 和 event.stopPropagation()方法之间有什么区别？
10. 如何知道是否在元素中使用了event.preventDefault()方法？
11. 为什么此代码obj.someprop.x会引发错误?
12. 什么是event.target？
13. 什么是event.currentTarget？
14. == 和 === 有什么区别？
15. 为什么在 JS 中比较两个相似的对象时返回 false？
16. !! 运算符能做什么？
17. 如何在一行中计算多个表达式的值？
18. 什么是提升？
19. 什么是作用域？
20. 什么是闭包？
21. JavaScript中的虚值是什么？
22. 如何检查值是否虚值？
23. 'use strict' 是干嘛用的？
24. JavaScript中 this 值是什么？
25. 对象的 prototype 是什么？
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
36. 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？
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？

1.undefined 和 null 有什么区别？

在理解`undefined`和`null`之间的差异之前，我们先来看看它们的相似类。

它们属于 JavaScript 的 7 种基本类型。

```
let primitiveTypes = ['string','number','null','undefined','boolean','symbol', 'bigint'];
```

它们是属于虚值，可以使用`Boolean(value)`或`!!value`将其转换为布尔值时，值为`false`。

```
console.log(!!null); // false
console.log(!!undefined); // false

console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
```

接着来看看它们的区别。

`undefined` 是未指定特定值的变量的默认值，或者没有显式返回值的函数，如：`console.log(1)`，还包括对象中不存在的属性，这些 JS 引擎都会为其分配 `undefined` 值。

```
let _thisIsUndefined;
const doNothing = () => {};
const someObj = {
  a : "ay",
  b : "bee",
  c : "si"
};

console.log(_thisIsUndefined); // undefined
console.log(doNothing()); // undefined
console.log(someObj["d"]); // undefined
```

`null` 是**“不代表任何值的值”**。`null` 是已明确定义给变量的值。在此示例中，当 `fs.readFile` 方法未引发错误时，我们将获得 `null` 值。

```
fs.readFile('path/to/file', (e,data) => {
   console.log(e); // 当没有错误发生时，打印 null
   if(e){
     console.log(e);
   }
   console.log(data);
 });
```

在比较 `null` 和 `undefined` 时，我们使用 `==` 时得到 `true`，使用 `===` 时得到 `false`：

```
console.log(null == undefined); // true
console.log(null === undefined); // false
```

2. && 运算符能做什么

`&&` 也可以叫**逻辑与**，在其操作数中找到第一个虚值表达式并返回它，如果没有找到任何虚值表达式，则返回最后一个真值表达式。它采用短路来防止不必要的工作。

```
console.log(false && 1 && []); // false
console.log(" " && true && 5); // 5
```

使用if语句

```
const router: Router = Router();
router.get('/endpoint', (req: Request, res: Response) => {
   let conMobile: PoolConnection;
   try {
      //do some db operations
   } catch (e) {
   if (conMobile) {
    conMobile.release();
   }
  }
});
```

使用&&操作符

```
const router: Router = Router();

router.get('/endpoint', (req: Request, res: Response) => {
  let conMobile: PoolConnection;
  try {
     //do some db operations
  } catch (e) {
    conMobile && conMobile.release()
  }
});
```

3. || 运算符能做什么

`||` 也叫或 逻辑或，在其操作数中找到第一个真值表达式并返回它。这也使用了短路来防止不必要的工作。在支持 ES6 默认函数参数之前，它用于初始化函数中的默认参数值。

```
console.log(null || 1 || undefined); // 1

function logName(name) {
  var n = name || "Mark";
  console.log(n);
}

logName(); // "Mark"
```

4. 使用 + 或一元加运算符是将字符串转换为数字的最快方法吗？

根据[MDN文档][1]，`+`是将字符串转换为数字的最快方法，因为如果值已经是数字，它不会执行任何操作。

5. DOM 是什么？

DOM 代表**文档对象模型**，是 HTML 和 XML 文档的接口(API)。当浏览器第一次读取(解析)HTML文档时，它会创建一个大对象，一个基于 HTM L文档的非常大的对象，这就是**DOM**。它是一个从 HTML 文档中建模的树状结构。DOM 用于交互和修改DOM结构或特定元素或节点。

假设我们有这样的 HTML 结构：

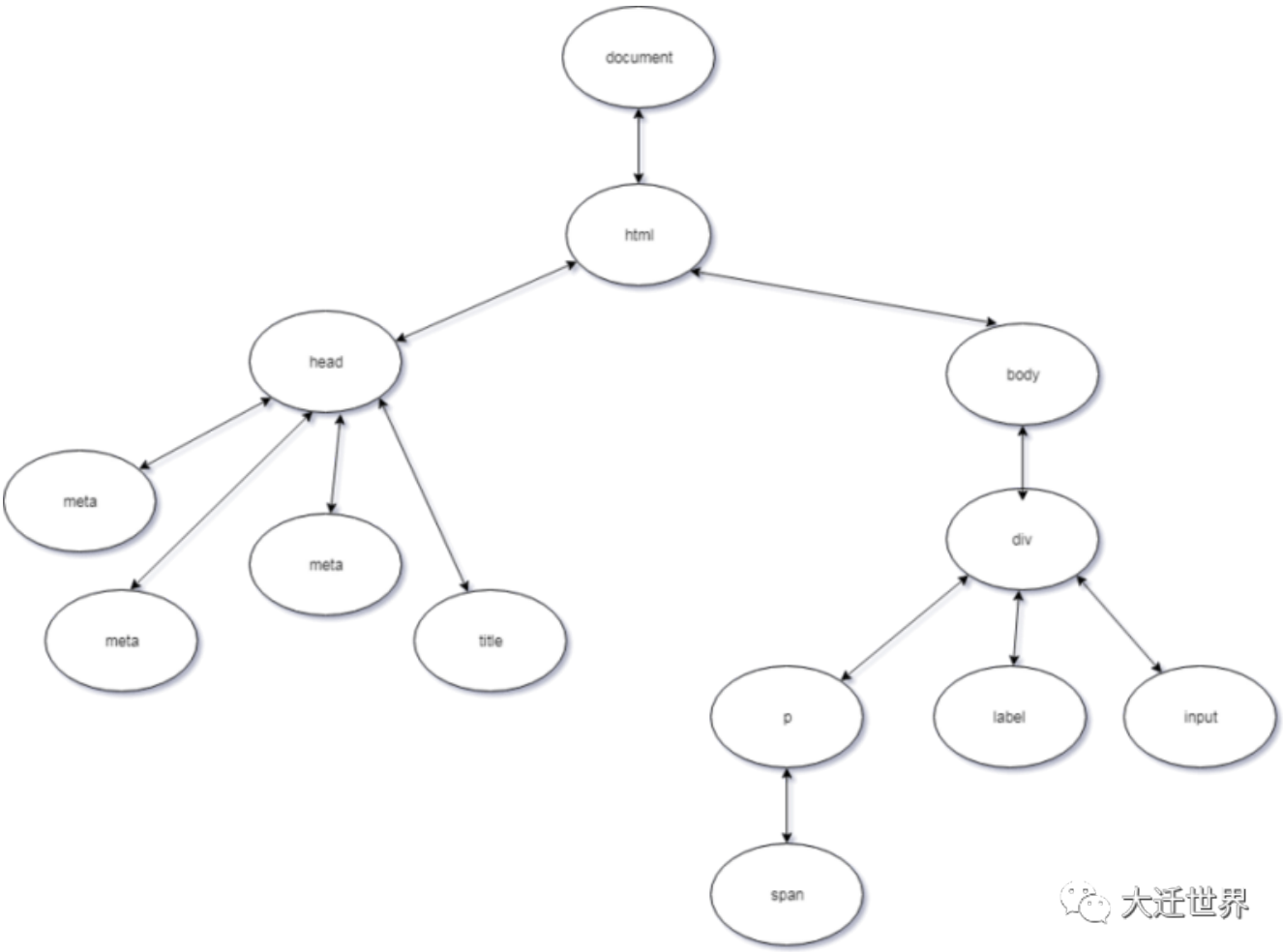
```
<!DOCTYPE html>
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Document Object Model</title>
</head>

<body>
   <div>
      <p>
         <span></span>
      </p>
      <label></label>
      <input>
   </div>
</body>

</html>
```

等价的**DOM**是这样的：



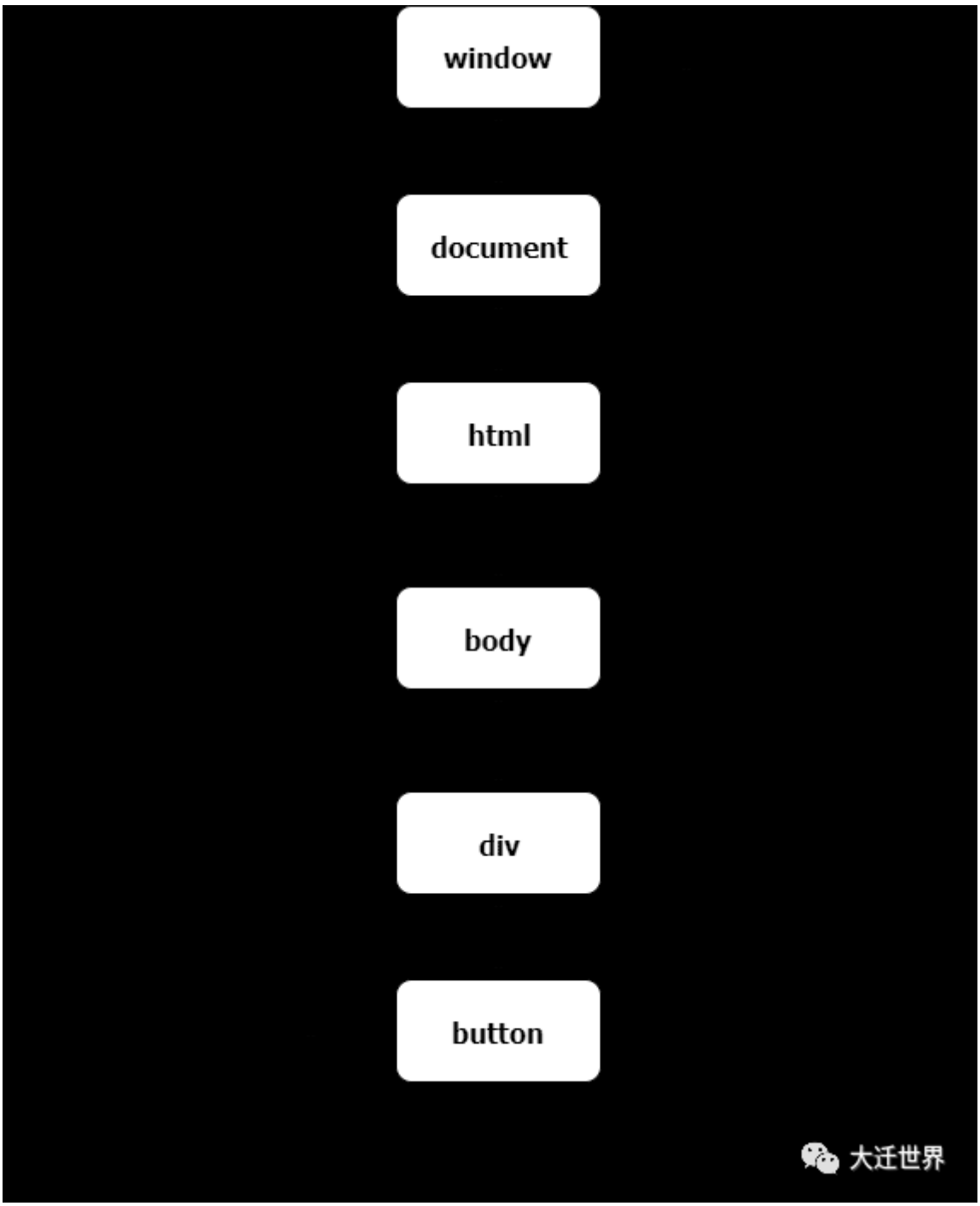
JS 中的`document`对象表示DOM。它为我们提供了许多方法，我们可以使用这些方法来选择元素来更新元素内容，等等。

6. 什么是事件传播?

当**事件**发生在**DOM**元素上时，该**事件**并不完全发生在那个元素上。在**"冒泡阶段"**中，事件冒泡或向上传播至父级，祖父母，祖父母或父级，直到到达window为止；而在**"捕获阶段"**中，事件从`window`开始向下触发元素 事件或`event.target`。

事件传播有三个阶段：

1. **捕获阶段**–事件从`window`开始，然后向下到每个元素，直到到达目标元素。
2. **目标阶段**–事件已达到目标元素。
3. **冒泡阶段**–事件从目标元素冒泡，然后上升到每个元素，直到到达`window`。



7. 什么是事件冒泡？

当**事件**发生在**DOM**元素上时，该**事件**并不完全发生在那个元素上。在冒泡阶段，事件冒泡，或者事件发生在它的父代，祖父母，祖父母的父代，直到到达`window`为止。

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```

对应的 JS 代码:

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })
```

```
});
```

`addEventListener`方法具有第三个可选参数`useCapture`，其默认值为`false`，事件将在冒泡阶段中发生，如果为`true`，则事件将在捕获阶段中发生。如果单击`child`元素，它将分别在控制台上记录`child`，`parent`，`grandparent`，`html`，`document`和`window`，这就是事件冒泡。

8. 什么是事件捕获？

当事件发生在 **DOM** 元素上时，该事件并不完全发生在那个元素上。在捕获阶段，事件从`window`开始，一直到触发事件的元素。

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```

对应的 JS 代码:

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
```

```
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })

});
```

`addEventListener`方法具有第三个可选参数`useCapture`，其默认值为`false`，事件将在冒泡阶段中发生，如果为`true`，则事件将在捕获阶段中发生。如果单击`child`元素，它将分别在控制台上打印`window`，`document`，`html`，`grandparent`和`parent`，这就是**事件捕获**。

9. event.preventDefault() 和 event.stopPropagation()方法之间有什么区别？

`event.preventDefault()`方法可防止元素的默认行为。如果在表单元素中使用，它将阻止其提交。如果在锚元素中使用，它将阻止其导航。如果在上下文菜单中使用，它将阻止其显示或显示。`event.stopPropagation()`方法用于阻止捕获和冒泡阶段中当前事件的进一步传播。

10. 如何知道是否在元素中使用了`event.preventDefault()`方法？

我们可以在事件对象中使用`event.defaultPrevented`属性。它返回一个布尔值用来表明是否在特定元素中调用了`event.preventDefault()`。

11. 为什么此代码`obj.someprop.x`会引发错误?

```
const obj = {};
console.log(obj.someprop.x);
```

显然，由于我们尝试访问`someprop`属性中的`x`属性，而 someprop 并没有在对象中，所以值为`undefined`。记住对象本身不存在的属性，并且其原型的默认值为`undefined`。因为`undefined`没有属性`x`，所以试图访问将会报错。

12. 什么是 event.target ？

简单来说，`event.target`是发生事件的元素或触发事件的元素。

假设有如下的 HTML 结构：

```
<div onclick="clickFunc(event)" style="text-align: center;margin:15px;
border:1px solid red;border-radius:3px;">
    <div style="margin: 25px; border:1px solid royalblue;border-radius:3px;">
        <div style="margin:25px;border:1px solid skyblue;border-radius:3px;">
          <button style="margin:10px">
             Button
          </button>
        </div>
    </div>
 </div>
```

JS 代码如下：

```
function clickFunc(event) {
  console.log(event.target);
}
```

如果单击 `button`，即使我们将事件附加在最外面的`div`上，它也将打印 `button` 标签，因此我们可以得出结论 `event.target` 是触发事件的元素。

13. 什么是 event.currentTarget？？

`event.currentTarget` 是我们在其上显式附加事件处理程序的元素。

假设有如下的 HTML 结构：

```
<div onclick="clickFunc(event)" style="text-align: center;margin:15px;
border:1px solid red;border-radius:3px;">
  <div style="margin: 25px; border:1px solid royalblue;border-radius:3px;">
      <div style="margin:25px;border:1px solid skyblue;border-radius:3px;">
        <button style="margin:10px">
           Button
        </button>
      </div>
  </div>
</div>
```

JS 代码如下：

```
function clickFunc(event) {
  console.log(event.currentTarget);
}
```

如果单击 `button`，即使我们单击该 `button`，它也会打印最外面的`div`标签。在此示例中，我们可以得出结论，`event.currentTarget` 是附加事件处理程序的元素。

14. == 和 === 有什么区别？

`==`用于一般比较，`===`用于严格比较，`==`在比较的时候可以转换数据类型，`===`严格比较，只要类型不匹配就返回`flase`。

先来看看 `==` 这兄弟：

强制是将值转换为另一种类型的过程。在这种情况下，`==`会执行隐式强制。在比较两个值之前，`==`需要执行一些规则。

假设我们要比较`x == y`的值。

1. 如果`x`和`y`的类型相同，则 JS 会换成`===`操作符进行比较。
2. 如果`x`为`null`，`y`为`undefined`，则返回`true`。
3. 如果`x`为`undefined`且`y`为`null`，则返回`true`。
4. 如果`x`的类型是`number`，`y`的类型是`string`，那么返回`x == toNumber(y)`。

5. 如果`x`的类型是`string`，`y`的类型是`number`，那么返回`toNumber(x) == y`。
6. 如果`x`为类型是`boolean`，则返回`toNumber(x)== y`。
7. 如果`y`为类型是`boolean`，则返回`x == toNumber(y)`。
8. 如果`x`是`string`、`symbol`或`number`，而`y`是`object`类型，则返回`x == toPrimitive(y)`。
9. 如果`x`是`object`，`y`是`string`，`symbol`则返回`toPrimitive(x) == y`。
10. 剩下的 返回`false`

注意：`toPrimitive`首先在对象中使用`valueOf`方法，然后使用`toString`方法来获取该对象的原始值。

举个例子。

x	y	x == y
5	5	true
1	'1'	true
null	undefined	true
0	false	true
'1,2'	[1,2]	true
'[object Object]'	{}	true

这些例子都返回`true`。

第一个示例符合`条件1`，因为`x`和`y`具有相同的类型和值。

第二个示例符合`条件4`，在比较之前将`y`转换为数字。

第三个例子符合`条件2`。

第四个例子符合`条件7`，因为`y`是`boolean`类型。

第五个示例符合`条件8`。使用`toString()`方法将数组转换为字符串，该方法返回`1,2`。

最后一个示例符合`条件8`。使用`toString()`方法将对象转换为字符串，该方法返回`[object Object]`。

x	y	x === y
5	5	true
1	'1'	false
null	undefined	false
0	false	false
'1,2'	[1,2]	false
'[object Object]'	{}	false

如果使用`===`运算符，则第一个示例以外的所有比较将返回`false`，因为它们的类型不同，而第一个示例将返回`true`，因为两者的类型和值相同。

具体更多规则可以对参考我之前的文章：

[我对 JS 中相等和全等操作符转化过程一直很迷惑，直到有了这份算法][2]

15. 为什么在 JS 中比较两个相似的对象时返回 false？

先看下面的例子：

```
let a = { a: 1 };
let b = { a: 1 };
let c = a;
console.log(a === b); // 打印 false，即使它们有相同的属性
console.log(a === c); // true
```

JS 以不同的方式比较对象和基本类型。在基本类型中，JS 通过值对它们进行比较，而在对象中，JS 通过引用或存储变量的内存中的地址对它们进行比较。这就是为什么第一个`console.log`语句返回`false`，而第二个`console.log`语句返回`true`。`a`和`c`有相同的引用地址，而`a`和`b`没有。

16. !! 运算符能做什么？

`!!`运算符可以将右侧的值强制转换为布尔值，这也是将值转换为布尔值的一种简单方法。

```
console.log(!!null); // false
console.log(!!undefined); // false
console.log(!!''); // false
console.log(!!0); // false
console.log(!!NaN); // false
console.log(!!' '); // true
console.log(!!{}); // true
console.log(!![]); // true
console.log(!!1); // true
console.log(!![].length); // false
```

17. 如何在一行中计算多个表达式的值？

可以使用`逗号`运算符在一行中计算多个表达式。它从左到右求值，并返回右边最后一个项目或最后一个操作数的值。

```
let x = 5;

x = (x++ , x = addFive(x), x *= 2, x -= 5, x += 10);

function addFive(num) {
  return num + 5;
}
```

上面的结果最后得到`x`的值为`27`。首先，我们将`x`的值增加到`6`，然后调用函数`addFive(6)`并将`6`作为参数传递并将结果重新分配给`x`，此时`x`的值为`11`。之后，将`x`的当前值乘以`2`并将其分配给`x`，`x`的更新值为`22`。然后，将`x`的当前值减去`5`并将结果分配给`x` `x`更新后的值为`17`。最后，我们将`x`的值增加`10`，然后将更新的值分配给`x`，最终`x`的值为`27`。

18. 什么是提升？

提升是用来描述变量和函数移动到其(全局或函数)作用域顶部的术语。

为了理解提升，需要来了解一下**执行上下文。执行上下文**是当前正在执行的**“代码环境”。**执行上下文有两个阶段:`编译`和`执行`。

编译-在此阶段，JS 引荐获取所有**函数声明**并将其**提升**到其作用域的顶部，以便我们稍后可以引用它们并获取所有变量声明（使用`var`关键字进行声明），还会为它们提供默认值：`undefined`。

执行——在这个阶段中，它将值赋给之前提升的变量，并执行或调用函数(对象中的方法)。

注意:只有使用`var`声明的变量，或者函数声明才会被提升，相反，函数表达式或箭头函数，`let`和`const`声明的变量，这些都不会被提升。

假设在全局使用域，有如下的代码：

```
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));

function greet(name){
  return 'Hello ' + name + '!';
}

var y;
```

上面分别打印：`undefined`,`1`, `Hello Mark!`。

上面代码在编译阶段其实是这样的：

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

var y; // 默认值 undefined

// 等待“编译”阶段完成，然后开始“执行”阶段

/*
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
*/
```

编译阶段完成后，它将启动执行阶段调用方法，并将值分配给变量。

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

var y;

//start "execution" phase

console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
```

19. 什么是作用域？

JavaScript 中的作用域是我们可以有效访问变量或函数的区域。JS 有三种类型的作用域：**全局作用域**、**函数作用域**和**块作用域(ES6)**。

- **全局作用域**——在全局命名空间中声明的变量或函数位于全局作用域中，因此在代码中的任何地方都可以访问它们。

```
//global namespace
var g = "global";

function globalFunc(){
  function innerFunc(){
    console.log(g); // can access "g" because "g" is a global variable
  }
 innerFunc();
}
```

- **函数作用域**——在函数中声明的变量、函数和参数可以在函数内部访问，但不能在函数外部访问。

```
function myFavoriteFunc(a) {
  if (true) {
    var b = "Hello " + a;
  }
  return b;
}

myFavoriteFunc("World");

console.log(a); // Throws a ReferenceError "a" is not defined
console.log(b); // does not continue here
```

- **块作用域**-在块`{}`中声明的变量（`let`，`const`）只能在其中访问。

```
function testBlock(){
  if(true){
    let z = 5;
  }
  return z;
}

testBlock(); // Throws a ReferenceError "z" is not defined
```

作用域也是一组用于查找变量的规则。如果变量在当前作用域中不存在，它将向外部作用域中查找并搜索，如果该变量不存在，它将再次查找直到到达全局作用域，如果找到，则可以使用它，否则引发错误，这种查找过程也称为**作用域链**。

```
/* 作用域链

 内部作用域->外部作用域-> 全局作用域
*/

// 全局作用域
var variable1 = "Comrades";
var variable2 = "Sayonara";

function outer(){
// 外部作用域
var variable1 = "World";
function inner(){
// 内部作用域
  var variable2 = "Hello";
  console.log(variable2 + " " + variable1);
}
inner();
}
outer(); // Hello World
```

20. 什么是闭包？

这可能是所有问题中最难的一个问题，因为闭包是一个有争议的话题，这里从个人角度来谈谈，如果不妥，多多海涵。

闭包就是一个函数在声明时能够记住当前作用域、父函数作用域、及父函数作用域上的变量和参数的引用，直至通过作用域链上全局作用域，基本上闭包是在声明函数时创建的作用域。

看看小例子：

```
// 全局作用域
var globalVar = "abc";

function a(){
  console.log(globalVar);
}

a(); // "abc"
```

在此示例中，当我们声明 `a` 函数时，全局作用域是 `a` 闭包的一部分。

变量 `globalVar` 在图中没有值的原因是该变量的值可以根据调用函数 `a` 的位置和时间而改变。但是在上面的示例中，`globalVar` 变量的值为 `abc`。

来看一个更复杂的例子：

```
var globalVar = "global";
var outerVar = "outer"

function outerFunc(outerParam) {
  function innerFunc(innerParam) {
    console.log(globalVar, outerParam, innerParam);
  }
  return innerFunc;
}

const x = outerFunc(outerVar);
outerVar = "outer-2";
globalVar = "guess"
x("inner");
```

上面打印结果是 `guess outer inner`。

当我们调用 `outerFunc` 函数并将返回值 `innerFunc` 函数分配给变量 `x` 时，即使我们为 `outerVar` 变量分配了新值 `outer-2`，`outerParam` 也继续保留 `outer` 值，因为重新分配是在调用 `outerFunc` 之后发生的，并且当我们调用 `outerFunc` 函数时，它会在作用域链中查找 `outerVar` 的值，此时的 `outerVar` 的值将为 `"outer"`。

现在，当我们调用引用了 `innerFunc` 的 `x` 变量时，`innerParam` 将具有一个 `inner` 值，因为这是我们在调用中传递的值，而 `globalVar` 变量值为 `guess`，因为在调用 `x` 变量之前，我们将一个新值分配给 `globalVar`。

下面这个示例演示没有理解好闭包所犯的错误：

```
const arrFuncs = [];
for(var i = 0; i < 5; i++){
  arrFuncs.push(function (){
    return i;
  });
}
console.log(i); // i is 5

for (let i = 0; i < arrFuncs.length; i++) {
  console.log(arrFuncs[i]()); // 都打印 5
}
```

由于闭包，此代码无法正常运行。`var`关键字创建一个全局变量，当我们 push 一个函数时，这里返回的全局变量`i`。因此，当我们在循环后在该数组中调用其中一个函数时，它会打印`5`，因为我们得到`i`的当前值为`5`，我们可以访问它，因为它是全局变量。

因为闭包在创建变量时会保留该变量的引用而不是其值。我们可以使用**IIFES**或使用`let`来代替`var`的声明。

21. JavaScript 中的虚值是什么？

```
const falsyValues = ['', 0, null, undefined, NaN, false];
```

简单的来说虚值就是是在转换为布尔值时变为`false`的值。

22. 如何检查值是否虚值？

使用`Boolean`函数或者`!!`运算符。

23. 'use strict' 是干嘛用的？

`"use strict"`是 **ES5** 特性，它使我们的代码在函数或整个脚本中处于**严格模式**。**严格模式**帮助我们在代码的早期避免 bug，并为其添加限制。

严格模式的一些限制：

1. 变量必须声明后再使用
2. 函数的参数不能有同名属性，否则报错
3. 不能使用`with`语句
4. 不能对只读属性赋值，否则报错
5. 不能使用前缀 0 表示八进制数，否则报错
6. 不能删除不可删除的属性，否则报错
7. 不能删除变量`delete prop`，会报错，只能删除属性`delete global[prop]`
8. `eval`不能在它的外层作用域引入变量
9. `eval`和`arguments`不能被重新赋值
10. `arguments`不会自动反映函数参数的变化
11. 不能使用`arguments.callee`
12. 不能使用`arguments.caller`
13. 禁止`this`指向全局对象
14. 不能使用`fn.caller`和`fn.arguments`获取函数调用的堆栈
15. 增加了保留字（比如`protected`、`static`和`interface`）

设立"严格模式"的目的，主要有以下几个：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
2. 消除代码运行的一些不安全之处，保证代码运行的安全；
3. 提高编译器效率，增加运行速度；
4. 为未来新版本的Javascript做好铺垫。

24. JavaScript 中 `this` 值是什么？

基本上，`this` 指的是当前正在执行或调用该函数的对象的值。`this` 值的变化取决于我们使用它的上下文和我们在哪里使用它。

```
const carDetails = {
  name: "Ford Mustang",
  yearBought: 2005,
  getName(){
    return this.name;
  },
  isRegistered: true
};

console.log(carDetails.getName()); // Ford Mustang
```

这通常是我们期望结果的，因为在 `getName` 方法中我们返回 `this.name` ，在此上下文中，`this` 指向的是 `carDetails` 对象，该对象当前是执行函数的"所有者"对象。

接下我们做些奇怪的事情：

```
var name = "Ford Ranger";
var getCarName = carDetails.getName;

console.log(getCarName()); // Ford Ranger
```

上面打印 `Ford Ranger` ，这很奇怪，因为在第一个 `console.log` 语句中打印的是 `Ford Mustang` 。这样做的原因是 `getCarName` 方法有一个不同的"所有者"对象，即 `window` 对象。在全局作用域中使用 `var` 关键字声明变量会在 `window` 对象中附加与变量名称相同的属性。请记住，当没有使用 `"use strict"` 时，在全局作用域中 `this` 指的是 `window` 对象。

```
console.log(getCarName === window.getCarName); // true
console.log(getCarName === this.getCarName); // true
```

本例中的 `this` 和 `window` 引用同一个对象。

解决这个问题的一种方法是在函数中使用 `apply` 和 `call` 方法。

```
console.log(getCarName.apply(carDetails)); // Ford Mustang
console.log(getCarName.call(carDetails));  // Ford Mustang
```

`apply` 和 `call` 方法期望第一个参数是一个对象，该对象是函数内部 `this` 的值。

`IIFE`或**立即执行的函数表达式**，在全局作用域内声明的函数，对象内部方法中的匿名函数和内部函数的`this`具有默认值，该值指向`window`对象。

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

function iHateThis(){
  console.log(this);
}

iHateThis(); // 打印 "window" 对象

const myFavoriteObj = {
 guessThis(){
    function getName(){
      console.log(this.name);
    }
    getName();
 },
 name: 'Marko Polo',
 thisIsAnnoying(callback){
   callback();
 }
};


myFavoriteObj.guessThis(); // 打印 "window" 对象
myFavoriteObj.thisIsAnnoying(function (){
 console.log(this); // 打印 "window" 对象
});
```

如果我们要获取`myFavoriteObj`对象中的`name`属性（即**Marko Polo**）的值，则有两种方法可以解决此问题。

一种是将`this`值保存在变量中。

```
const myFavoriteObj = {
 guessThis(){
  const self = this; // 把 this 值保存在 self 变量中
  function getName(){
    console.log(self.name);
  }
  getName();
 },
 name: 'Marko Polo',
 thisIsAnnoying(callback){
   callback();
  }
};
```

第二种方式是使用箭头函数

```
const myFavoriteObj = {
  guessThis(){
     const getName = () => {
       console.log(this.name);
     }
     getName();
  },
  name: 'Marko Polo',
  thisIsAnnoying(callback){
   callback();
  }
};
```

箭头函数没有自己的`this`。它复制了这个封闭的词法作用域中`this`值，在这个例子中，`this`值在`getName`内部函数之外，也就是`myFavoriteObj`对象。

25. 对象的 prototype(原型) 是什么？

简单地说，原型就是对象的蓝图。如果它存在当前对象中，则将其用作属性和方法的回退。它是在对象之间共享属性和功能的方法，这也是JavaScript实现继承的核心。

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

即使`o`对象中不存在`o.toString`方法，它也不会引发错误，而是返回字符串`[object Object]`。当对象中不存在属性时，它将查看其原型，如果仍然不存在，则将其查找到原型的原型，依此类推，直到在原型链中找到具有相同属性的属性为止。原型链的末尾是`Object.prototype`。

```
console.log(o.toString === Object.prototype.toString); // logs true
```

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
started at ${this.yearStarted}`;
}

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
started at ${this.yearStarted}`;
  }

  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) => {...});
```

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(value)
}) {
  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&0`，结果是`0`。
- 然后我们比较最后`1&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写就可以改变。

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 +
cur, startingValue);
```

```
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 'name'
of object
```

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的Iterator接口规范。

//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。