JavaScript篇

一、数据类型

1. JavaScript有哪些数据类型，它们的区别?
2. 数据类型检测的方式有哪些
3. 基本数据类型和引用数据类型的区别
4. 判断数组的方式有哪些
5. null和undefined区别
6. 为什么typeof null得到object而不是null?
7. null == undefined 得到true 且 null===undefined得到false 为什么
8. intanceof 操作符的实现原理及实现
9. 为什么0.1+0.2 ! == 0.3，如何让其相等
10. 如何获取安全的 undefined 值?
11. typeof NaN 的结果是什么?
12. isNaN 和 Number.isNaN 函数的区别?
13. == 操作符的强制类型转换规则?
14. 其他值到字符串的转换规则?
15. 其他值到数字值的转换规则?
16. JS 字符串转数字的方法
17. 其他值到布尔类型的值的转换规则?
18. || 和 && 操作符的返回值?
19. Object.is() 与比较操作符 “===”、“==” 的区别?
20. 什么是 JavaScript 中的包装类型?
21. JavaScript 中如何进行隐式类型转换?
22. + 操作符什么时候用于字符串的拼接?
23. 为什么会有BigInt的提案?
24. object.assign和扩展运算法是深拷贝还是浅拷贝，两者区别

二、ES6

1. let、const、var的区别
2. 如果new一个箭头函数的会怎么样
3. 箭头函数与普通函数的区别
4. 箭头函数的this指向哪里?
5. 扩展运算符的作用及使用场景
6. Proxy 可以实现什么功能?
7. 对对象与数组的解构的理解
8. 如何提取高度嵌套的对象里的指定属性?
9. 对 rest 参数的理解
10. ES6中模板语法与字符串处理
11. 简单介绍一下 symbol
12. 简单讲一讲 ES6、ES7、ES8 的一些新特性

三、JavaScript基础

1. new操作符的实现原理
2. map和Object的区别
3. JavaScript有哪些内置对象
4. 常用的正则表达式有哪些?
5. 对JSON的理解
6. JavaScript脚本延迟加载的方式有哪些?
7. JavaScript 类数组对象的定义?
8. 数组有哪些原生方法?
9. Unicode、UTF-8、UTF-16、UTF-32的区别?
10. 常见的位运算符有哪些？其计算规则是什么?
11. 为什么函数的 arguments 参数是类数组而不是数组？如何遍历类数组?
12. 什么是 DOM 和 BOM?
13. 对类数组对象的理解，如何转化为数组
14. 数组去重
15. escape、encodeURI、encodeURIComponent 的区别
16. 对AJAX的理解，实现一个AJAX请求
17. JavaScript为什么要进行变量提升，它导致了什么问题?
18. 什么是尾调用，使用尾调用有什么好处?
19. ES6模块与CommonJS模块有什么异同?
20. 常见的DOM操作有哪些
21. 什么是事件模型，DOM0 级和 DOM2 级有什么区别，DOM 的分级是什么
22. use strict是什么意思？使用它区别是什么?
23. 如何判断一个对象是否属于某个类?
24. 强类型语言和弱类型语言的区别
25. 解释性语言和编译型语言的区别
26. for...in和for...of的区别
27. 如何使用for...of遍历对象
28. ajax、axios、fetch的区别
29. 数组的遍历方法有哪些
30. forEach和map方法有什么区别
31. 说说前端中的事件流
32. 如何让事件先冒泡后捕获
33. 事件委托以及冒泡原理
34. 什么是事件监听
35. JS 的各种位置，比如 clientHeight,scrollHeight,offsetHeight ,以及 scrollTop,，offsetTop,clientTop 的区别?
36. JS 拖拽功能的实现
37. JS 的节流和防抖
38. 如何理解前端模块化
39. 说一下 CommonJS、AMD 和 CMD
40. 对象深度克隆的简单实现（包装对象，Date 对象，正则对象）

41. JS 监听对象属性的改变
42. 实现一个两列等高布局，讲讲思路
43. JS 怎么控制一次加载一张图片，加载完后再加载下一张
44. 性能优化
45. 能来讲讲 JS 的语言特性吗
46. JS 的全排列
47. 动画游戏卡顿甚至崩溃的原因（3-5 个）以及解决办法（3-5 个）
48. 说一下什么是 virtual dom
49. 说说 C++,Java，JavaScript 这三种语言的区别
50. JS 加载过程阻塞，解决方法。

四、原型与原型链

1. 对原型、原型链的理解
2. 原型修改、重写
3. 原型链指向
4. JS 原型链，原型链的顶端是什么？Object 的原型是什么？Object 的原型的原型是什么?
5. 在数组原型链上实现删除数组重复数据的方法：

五、执行上下文/作用域链/闭包

1. 对闭包的理解
2. 循环中使用闭包解决 var 定义函数的问题
3. 对作用域、作用域链的理解
4. 对执行上下文的理解

六、this/call/apply/bind

1. 对 this 对象的理解
2. call() 和 apply() 的区别?
3. 实现 call、apply 及 bind 函数

七、异步编程

1. 异步编程的实现方式?
2. setTimeout、Promise、Async/Await 的区别
3. 对 Promise 的理解
4. Promise 的基本用法
5. Promise 解决了什么问题
6. Promise.all 和 Promise.race 的区别的使用场景
7. 对 async/await 的理解
8. await 到底在等啥?
9. async/await 的优势
10. async/await 对比 Promise 的优势

11. async/await 如何捕获异常
12. 并发与并行的区别?
13. 什么是回调函数？回调函数有什么缺点？如何解决回调地狱问题?
14. setTimeout、setInterval 和 requestAnimationFrame 之间的区别
15. 补充 get 和 post 请求在缓存方面的区别
16. Ajax 解决浏览器缓存问题
17. 将原生的 ajax 封装成 promise
18. JS 实现跨域
19. 跨域的原理
20. 同源策略

八、面向对象

1. 对象创建的方式有哪些?
2. JS 中继承实现的几种方式

九、垃圾回收与内存泄漏

1. 浏览器的垃圾回收机制
2. 哪些情况会导致内存泄漏

一、数据类型

1. JavaScript 有哪些数据类型，它们的区别?

基本数据类型：**number**、**string**、**boolean**、Undefined、**NaN(**特殊值**)**、BigInt、Symbol
引入数据类型：Object

NaN 是 JS 中的特殊值，表示非数字，NaN 不是数字，但是他的数据类型是数字，它不等于任何值，包括自身，在布尔运算时被当做 false，NaN 与任何数运算得到的结果都是 NaN，计算失败或者运算无法返回正确的数值的就会返回 NaN，一些数学函数的运算结果也会出现 NaN ,

其中 Symbol 和 BigInt 是 ES6 中新增的数据类型：

- Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。
- BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

这些数据可以分为原始数据类型和引用数据类型：

- 栈：原始数据类型（Undefined、Null、Boolean、Number、String）
- 堆：引用数据类型（对象、数组和函数）

2、基本数据类型和引用数据类型的区别

1. 基本数据类型是按值访问的，也就是说我们可以操作保存在变量中的实际的值，
2. 基本数据类型的值是不可变的，任何方法都无法改变一个基本类型的值，当这个变量重新赋值后看起来变量的值是改变了，但是这里变量名只是指向变量的一个指针，所以改变的是指针的指向改变，该变量是不变的，但是引用类型可以改变
3. 基本数据类型不可以添加属性和方法，但是引用类型可以
4. 基本数据类型的赋值是简单赋值，如果从一个变量向另一个变量赋值基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上，引用数据类型的赋值是对象引用，
5. 基本数据类型的比较是值的比较，引用类型的比较是引用的比较，比较对象的内存地址是否相同
6. 基本数据类型是存放在栈区的，引用数据类型同事保存在栈区和堆区

2. 数据类型检测的方式有哪些

（1）typeof

```
console.log(typeof 2);               // number
console.log(typeof true);            // boolean
console.log(typeof 'str');           // string
console.log(typeof []);              // object
console.log(typeof function(){});    // function
console.log(typeof {});              // object
console.log(typeof undefined);       // undefined
console.log(typeof null);            // object
```

其中数组、对象、null都会被判断为object，其他判断都正确。

（2）instanceof

instanceof可以正确判断对象的类型，其内部运行机制是判断在其原型链中能否找到该类型的原型。

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);               // false

console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

可以看到，instanceof只能正确判断引用数据类型，而不能判断基本数据类

型。instanceof 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

（3） constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

constructor有两个作用，一是判断数据的类型，二是对象实例通过 constrcutor 对象访问它的构造函数。需要注意，如果创建一个对象来改变它的原型，constructor就不能用来判断数据类型了：

```
function Fn(){};

Fn.prototype = new Array();

var f = new Fn();

console.log(f.constructor===Fn);    // false
console.log(f.constructor===Array); // true
```

（4）Object.prototype.toString.call()

Object.prototype.toString.call() 使用 Object 对象的原型方法 toString 来判断数据类型：

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

同样是检测对象obj调用toString方法，obj.toString()的结果和Object.prototype.toString.call(obj)的结果不一样，这是为什么？

这是因为toString是Object的原型方法，而Array、function等类型作为**Object**的实例，都重写了**toString**方法。不同的对象类型调用toString方法时，根据原型链的知识，调用的是对应的重写之后的toString方法（function类型返回内容为函数体的字符串，Array类型返回元素组成的字符串...），而不会去调用Object上原型toString方法（返回对象的具体类型），所以采用obj.toString()不能得到其对象类型，只能将obj转换为字符串类型；因此，在

想要得到对象的具体类型时，应该调用 Object 原型上的 toString 方法。

3. 判断数组的方式有哪些

- 通过 Object.prototype.toString.call() 做判断
  Object.prototype.toString.apply / call (变量) === '[object Array]';

- 通过原型链做判断

obj.__proto__ === Array.prototype;

- 通过 ES6 的 Array.isArray() 做判断
  Array.isArray(变量)　// 返回值为 true（是）和 false（不是）

- 通过 instanceof 做判断

obj **instanceof** Array　//返回值为 true（是）和 false（不是）

- 通过 Array.prototype.isPrototypeOf

Array.prototype.isPrototypeOf(obj)

4. null 和 undefined 区别

undefined：未定义的值
得到 undefined 的方式：

1. 声明了一个变量，但没有赋值
2. 访问对象上不存在的属性
3. 函数定义了形参，但没有传递实参
4. 使用函数的返回值，当没有 return 操作时，就默认返回一个原始的状态值，这个值就是 undefined，表明函数的返回值未被定义。
5. 使用 **void** 对表达式求值
    - ECMAScript 明确规定 void 操作符 对任何表达式求值都返回 undefined

null： 代表空值

区别：undefined 表示一个变量自然的、最原始的状态值，而 null 则表示一个变量被人为的设置为空对象，而不是原始状态。所以，在实际使用过程中，为了保证变量所代表的语义，不要对一个变量显式的赋值 undefined，当需要释放一个对象时，直接赋值为 null 即可。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回 “object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

5. 为什么 typeof null 得到 object 而不是 null?

typeof null 的结果是 Object。因为 javascript 中不同对象在底层都表示为二进制，而 javascript 中会把二进制前三位都为 0 的判断为 object 类型，而 null 的二进制表示全都是 0，自然前三位也是 0，所以执行 typeof

时会返回'object。
在 JavaScript 第一个版本中，所有值都存储在 32 位的单元中，每个单元包含一个小的 类型标签 **(1-3 bits)** 以及当前要存储值的真实数据。类型标签存储在每个单元的低位中，共有五种数据类型：
000: object - 当前存储的数据指向一个对象。
1: int - 当前存储的数据是一个 31 位的有符号整数。
010: double - 当前存储的数据指向一个双精度的浮点数。
100: string - 当前存储的数据指向一个字符串。
110: boolean - 当前存储的数据是布尔值。

如果最低位是 1，则类型标签标志位的长度只有一位；如果最低位是 0，则类型标签标志位的长度占三位，为存储其他四种数据类型提供了额外两个 bit 的长度。
有两种特殊数据类型：

- undefined的值是 (-2)30(一个超出整数范围的数字)；
- null 的值是机器码 NULL 指针 (null 指针的值全是 0)

那也就是说null的类型标签也是000，和Object的类型标签一样，所以会被判定为Object。

53、 null == undefined 得到 true 且 null===undefined得到 false 为什么

要比较相等性之前，不能将 null 和 undefined 转换成其他任何值，但 null == undefined会返回 true 。ECMAScript 规范中是这样定义的。
原因：null： Null类型，代表“空值”，代表一个空对象指针，使用typeof运算得到 “object”，所以你可以认为它是一个特殊的对象值。
undefined： Undefined类型，当一个声明了一个变量未初始化时，得到的就是undefined。
实际上，undefined值是派生自null值的，ECMAScript标准规定对二者进行相等性测试要返回true，

6. intanceof 操作符的实现原理及实现

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceof(left, right) {
  // 获取对象的原型
  let proto = Object.getPrototypeOf(left)
  // 获取构造函数的 prototype 对象
  let prototype = right.prototype;

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
```

```
    if (proto === prototype) return true;
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原型
    proto = Object.getPrototypeOf(proto);
  }
}
```

7. 为什么 0.1+0.2 ! == 0.3，如何让其相等

在开发过程中遇到类似这样的问题：

```
let n1 = 0.1, n2 = 0.2
console.log(n1 + n2)  // 0.30000000000000004
```

这里得到的不是想要的结果，要想等于0.3，就要把它进行转化：

```
(n1 + n2).toFixed(2) // 注意，toFixed为四舍五入
```

toFixed(num) 方法可把 Number 四舍五入为指定小数位数的数字。那为什么会出现这样的结果呢?

计算机是通过二进制的方式存储数据的，所以计算机计算0.1+0.2的时候，实际上是计算的两个数的二进制的和。0.1的二进制是0.0001100110011001100...（1100循环），0.2的二进制是：0.00110011001100...（1100循环），这两个数的二进制都是无限循环的数。那JavaScript是如何处理无限循环的二进制小数呢?

一般我们认为数字包括整数和小数，但是在 JavaScript 中只有一种数字类型：Number，它的实现遵循IEEE 754标准，使用64位固定长度来表示，也就是标准的double双精度浮点数。在二进制科学表示法中，双精度浮点数的小数部分最多只能保留52位，再加上前面的1，其实就是保留53位有效数字，剩余的需要舍去，遵从“0舍1入”的原则。

根据这个原则，0.1和0.2的二进制数相加，再转化为十进制数就是：0.30000000000000004。

下面看一下双精度数是如何保存的：

- 第一部分（蓝色）：用来存储符号位（sign），用来区分正负数，0表示正数，占用1位
- 第二部分（绿色）：用来存储指数（exponent），占用11位
- 第三部分（红色）：用来存储小数（fraction），占用52位

对于0.1，它的二进制为：

```
0.0001100110011001100110011001100110011001100110011001100110011001
10011...
```

转为科学计数法（科学计数法的结果就是浮点数）：
1.1001100110011001100110011001100110011001100110011001*2^-4

可以看出0.1的符号位为0，指数位为-4，小数位为：
10011001100110011001100110011001100110011001100110011001

那么问题又来了，指数位是负数，该如何保存呢?
IEEE标准规定了一个偏移量，对于指数部分，每次都加这个偏移量进行保存，这样即使指数是负数，那么加上这个偏移量也就是正数了。由于JavaScript的数字是双精度数，这里就以双精度数为例，它的指数部分为11位，能表示的范围就是0~2047，IEEE固定双精度数的偏移量为**1023**。

- 当指数位不全是0也不全是1时 (规格化的数值)，IEEE规定，阶码计算公式为 e-Bias。 此时e最小值是1，则1-1023= -1022，e最大值是2046，则2046-1023=1023，可以看到，这种情况下取值范围是-1022~1013。
- 当指数位全部是0的时候 (非规格化的数值)，IEEE规定，阶码的计算公式为1-Bias，即1-1023= -1022。
- 当指数位全部是1的时候 (特殊值)，IEEE规定这个浮点数可用来表示3个特殊值，分别是正无穷，负无穷，NaN。 具体的，小数位不为0的时候表示NaN；小数位为0时，当符号位s=0时表示正无穷，s=1时候表示负无穷。

对于上面的0.1的指数位为-4，-4+1023 = 1019 转化为二进制就是：
1111111011.
所以，0.1表示为：
0 1111111011 10011001100110011001100110011001100110011001100110011001

说了这么多，是时候该最开始的问题了，如何实现0.1+0.2=0.3呢?
对于这个问题，一个直接的解决方法就是设置一个误差范围，通常称为“机器精度”。对JavaScript来说，这个值通常为2-52，在ES6中，提供了Number.EPSILON属性，而它的值就是2-52，只要判断0.1+0.2-0.3是否小于Number.EPSILON，如果小于，就可以判断为0.1+0.2 ===0.3

```
function numberepsilon(arg1,arg2){
  return Math.abs(arg1 - arg2) < Number.EPSILON;
}

console.log(numberepsilon(0.1 + 0.2, 0.3)); // true
```

8. 如何获取安全的 undefined 值?

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

9. typeof NaN 的结果是什么?

NaN 指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel

value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。
typeof NaN; *// "number"*

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 x === x 不成立）的值。而 NaN !== NaN 为 true。

10. isNaN 和 Number.isNaN 函数的区别?

- 函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断。
- 函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN ，不会进行数据类型的转换，这种方法对于 NaN 的判断更为准确。

11. == 操作符的强制类型转换规则?

对于 == 来说，如果对比双方的类型不一样，就会进行类型转换。假如对比 x 和 y 是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否**相同，**相同的话就比较两者的大小；
2. 类型不相同的话，就会进行类型转换；
3. 会先判断是否在对比 null 和 undefined，是的话就会返回 true
4. 判断两者类型是否为 string 和 number，是的话就会将字符串转换为 number

```
1 == '1'
     ↓
1 ==  1
```

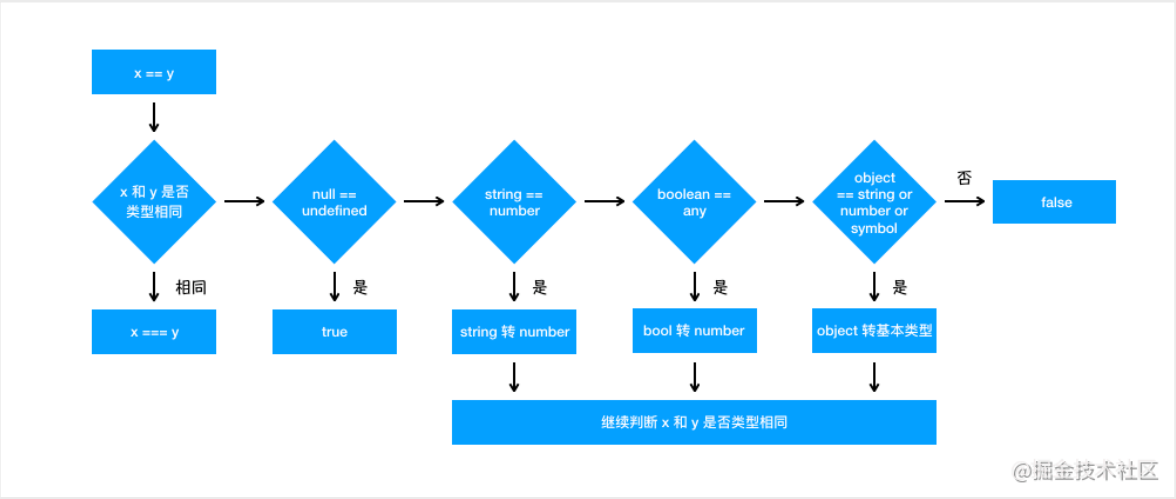
5. 判断其中一方是否为 boolean，是的话就会把 boolean 转为 number 再进行判断

```
'1' == true
       ↓
'1' ==  1
       ↓
 1  ==  1
```

6. 判断其中一方是否为 object 且另一方为 string、number 或者 symbol，是的话就会把 object 转为原始类型再进行判断

```
'1' == { name: 'js' }       ↓'1' == '[object Object]'
```

其流程图如下：



12. 其他值到字符串的转换规则?

- Null 和 Undefined 类型 ， null 转换为 "null"，undefined 转换为 "undefined"，
- Boolean 类型，true 转换为 "true"，false 转换为 "false"。
- Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
- Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
- 对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

13. 其他值到数字值的转换规则?

- Undefined 类型的值转换为 NaN。
- Null 类型的值转换为 0。
- Boolean 类型的值，true 转换为 1，false 转换为 0。
- String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。
- Symbol 类型的值不能转换为数字，会报错。
- 对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。
如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

76、JS 字符串转数字的方法

通过函数 parseIn（t ），可解析一个字符串，并返回一个整数，语法为 parseIn（tstring ,radix）
string：被解析的字符串

radix：表示要解析的数字的基数，默认是十进制，如果 radix<2 或>36, 则返回 NaN

14. 其他值到布尔类型的值的转换规则?

以下这些是假值：・undefined・null・false・+0、-0 和 NaN・""
假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

15. || 和 && 操作符的返回值?

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先强制转换为布尔类型，然后再执行条件判断。

- 对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。
- && 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果

16. Object.is() 与比较操作符 “===”、“==” 的区别?

“==”：不全相等，只比较数据，不比较类型，如果两边的类型不一致，则会进行强制类型转化后再进行比较
“===”：全等，既要比较数据，也比较数据类型，如果两边的类型不一致时，不会做强制类型准换，直接返回 false
(1) == 主要存在：强制转换成 number, null==undefined
" "==0　//true
"0"==0 //true
" " !="0" //true
123=="123" //true
null==undefined //true
(2)Object.js：一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。
主要的区别就是+0 != -0 而 NaN==NaN
(相对比===和==的改进)

17. 什么是 JavaScript 中的包装类型?

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象，如：

```
const a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

在访问'abc'.length时，JavaScript 将'abc'在后台转换成String('abc')，然后再访问其length属性。
JavaScript也可以使用Object函数显式地将基本类型转换为包装类型：
```
var a = 'abc'
Object(a) // String {"abc"}
```

也可以使用valueOf方法将包装类型倒转成基本类型：
```
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

看看如下代码会打印出什么：
```
var a = new Boolean( false );
if (!a) {
    console.log( "Oops" ); // never runs
}
```

答案是什么都不会打印，因为虽然包裹的基本类型是false，但是false被包裹成包装类型后就成了对象，所以其非值为false，所以循环体中的内容不会运行。

18. JavaScript 中如何进行隐式类型转换?

首先要介绍ToPrimitive方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值 （无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身；如果值为对象，其看起来大概是这样：
```
/**
* @obj 需要转换的对象
* @type 期望的结果类型
*/
ToPrimitive(obj,type)
```

type的值为number或者string。

（**1**）当type为number时规则如下：

- 调用obj的valueOf方法，如果为原始值，则返回，否则下一步；
- 调用obj的toString方法，后续同上；
- 抛出TypeError 异常。

（**2**）当type为string时规则如下：

- 调用obj的toString方法，如果为原始值，则返回，否则下一步；
- 调用obj的valueOf方法，后续同上；
- 抛出TypeError 异常。

可以看出两者的主要区别在于调用toString和valueOf的先后顺序。默认情况下：

- 如果对象为 Date 对象，则type默认为string；
- 其他情况下，type默认为number。

总结上面的规则，对于 Date 以外的对象，转换为基本类型的大概规则可以概括为一个函数：

```
var objToNumber = value => Number(value.valueOf().toString())
objToNumber([]) === 0
objToNumber({}) === NaN
```

而 JavaScript 中的隐式类型转换主要发生在+、-、*、/以及==、>、<这些运算符之间。而这些运算符只能操作基本类型值，所以在进行这些运算前的第一步就是将两边的值用 ToPrimitive 转换成基本类型，再进行操作。

以下是基本类型的值在不同操作符的情况下隐式转换的规则 （对于对象，其会被 ToPrimitive 转换成基本类型，所以最终还是要应用基本类型转换规则）：

1. +操作符

+操作符的两边有至少一个 string 类型变量时，两边的变量都会被隐式转换为字符串；其他情况下两边的变量都会被转换为数字。

```
1 + '23' // '123'
 1 + false // 1
 1 + Symbol() // Uncaught TypeError: Cannot convert a Symbol value to a number
 '1' + false // '1false'
 false + true // 1
```

2. -、*、\操作符

NaN 也是一个数字

```
1 * '23' // 23
 1 * false // 0
 1 / 'aa' // NaN
```

3. 对于==操作符

操作符两边的值都尽量转成 number：

```
3 == true // false, 3 转为 number 为 3，true 转为 number 为 1
'0' == false //true, '0'转为 number 为 0，false 转为 number 为 0
'0' == 0 // '0'转为 number 为 0
```

4. 对于<和>比较符

如果两边都是字符串，则比较字母表顺序：

```
'ca' < 'bd' // false
'a' < 'b' // true
```

其他情况下，转换为数字再比较：

```
'12' < 13 // true
false > -1 // true
```

以上说的是基本类型的隐式转换，而对象会被 ToPrimitive 转换为基本类型再进行转换：

```
var a = {}
a > 2 // false
```

其对比过程如下：
a.valueOf() // {}, 上面提到过，ToPrimitive 默认 type 为 number，所以先 valueOf，结果还是个对象，下一步
a.toString() // "[object Object]"，现在是一个字符串了
Number(a.toString()) // NaN，根据上面 < 和 > 操作符的规则，要转换成数字
NaN > 2 //false，得出比较结果

又比如：

```
var a = {name:'Jack'}
var b = {age: 18}
a + b // "[object Object][object Object]"
```

运算过程如下：
a.valueOf() // {}，上面提到过，ToPrimitive 默认 type 为 number，所以先 valueOf，结果还是个对象，下一步
a.toString() // "[object Object]"
b.valueOf() // 同理
b.toString() // "[object Object]"
a + b // "[object Object][object Object]"

19. + 操作符什么时候用于字符串的拼接?

根据 ES5 规范，如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话，+ 将进行拼接操作。如果其中一个操作数是对象（包括数组），则首先对其调用 ToPrimitive 抽象操作，该抽象操作再调用 [[DefaultValue]]，以数字作为上下文。如果不能转换为字符串，则会将其转换为数字类型来进行计算。
简单来说就是，如果 + 的其中一个操作数是字符串（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字加法。
那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字。

20. 为什么会有 BigInt 的提案?

JavaScript 中 Number.MAX_SAFE_INTEGER 表示最大安全数字，计算结果是 9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。但是一旦超过这个范围，js 就会出现计算不准确的情况，这在大数计算的时候不得不依靠一些第三方库进行解决，因此官方提出了 BigInt 来解决此问题。

21. object.assign 和扩展运算法是深拷贝还是浅拷贝，两者区别

扩展运算符：

```
let outObj = {
  inObj: {a: 1, b: 2}
}
```

```
let newObj = {...outObj}
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}

Object.assign():
let outObj = {
  inObj: {a: 1, b: 2}
}
let newObj = Object.assign({}, outObj)
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}
```

可以看到，两者都是浅拷贝。

- Object.assign() 方法接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。它会修改了一个对象，因此会触发 ES6 setter。
- 扩展操作符（...）使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象中。它不复制继承的属性或类的属性，但是它会复制ES6的 symbols 属性。

二、ES6

1. let、const、var的区别

（1）块级作用域： 块作用域由 { }包括，let和const具有块级作用域，var不存在块级作用域。块级作用域解决了ES5中的两个问题：

- 内层变量可能覆盖外层变量
- 用来计数的循环变量泄露为全局变量

（2）变量提升： var存在变量提升，let和const不存在变量提升，即在变量只能在声明之后使用，否在会报错。

（3）给全局添加属性： 浏览器的全局对象是window，Node的全局对象是global。var声明的变量为全局变量，并且会将该变量添加为全局对象的属性，但是let和const不会。

（4）重复声明： var声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的遍历。const和let不允许重复声明变量。

（5）暂时性死区： 在使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为暂时性死区。使用var声明的变量不存在暂时性死区。

（6）初始值设置： 在变量声明时，var 和 let 可以不用设置初始值。而const声明变量必须设置初始值。

（7）指针指向： let和const都是ES6新增的用于创建变量的语法。 let创建的变量是可以更改指针指向（可以重新赋值）。但const声明的变量是不允许改变指针的指向。

区别	var	let	const
是否有块级作用域	×	✓	✓

是否存在变量提升	✓	×	×
是否添加全局属性	✓	×	×
能否重复声明变量	✓	×	×
是否存在暂时性死区	×	✓	✓
是否必须设置初始值	×	×	✓
能否改变指针指向	✓	✓	×

2. const对象的属性可以修改吗

const保证的并不是变量的值不能改动，而是变量指向的那个内存地址不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。
但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

3. 如果new一个箭头函数的会怎么样

箭头函数是ES6中的提出来的，它没有prototype，也没有自己的this指向，更不可以使用arguments参数，所以不能New一个箭头函数。
new操作符的实现步骤如下：

1. 创建一个对象
2. 将构造函数的作用域赋给新对象（也就是将对象的__proto__属性指向构造函数的prototype属性）
3. 指向构造函数中的代码，构造函数中的this指向该对象（也就是为这个对象添加属性和方法）
4. 返回新的对象

所以，上面的第二、三步，箭头函数都是没有办法执行的。

4. 箭头函数与普通函数的区别

（1）箭头函数比普通函数更加简洁

- 如果没有参数，就直接写一个空括号即可
- 如果只有一个参数，可以省去参数的括号
- 如果有多个参数，用逗号分割
- 如果函数体的返回值只有一句，可以省略大括号
- 如果函数体不需要返回值，且只有一句话，可以给这个语句前面加一个void关键字。最常见的就是调用一个函数：

（2）箭头函数没有自己的**this**

箭头函数不会创建自己的this， 所以它没有自己的this，它只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它在定义时已经确定了，之

后不会改变。
（3）箭头函数继承来的**this**指向永远不会改变，
（4）**call()**、**apply()**、**bind()**等方法不能改变箭头函数中**this**的指向

（5）箭头函数不能作为构造函数使用
构造函数在new的步骤在上面已经说过了，实际上第二步就是将函数中的this指向该对象。 但是由于箭头函数时没有自己的this的，且this指向外层的执行环境，且不能改变指向，所以不能当做构造函数使用。
（6）箭头函数没有自己的**arguments,**但是可以访问外围函数的arguments 对象
（7）箭头函数没有**prototype,**不能通过 new 关键字调用，同样也没有new.target 值和原型
（8）箭头函数不能用作**Generator**函数，不能使用**yeild**关键字

5. 箭头函数的this指向哪里?
箭头函数不同于传统JavaScript中的函数，箭头函数并没有属于自己的this，它所谓的this是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的this，所以是不会被new调用的，这个所谓的this也不会被改变。
可以用Babel理解一下箭头函数:

```
// ES6
const obj = {
  getArrow() {
    return () => {
      console.log(this === obj);
    };
  }
}
```

转化后:

```
// ES5，由 Babel 转译
var obj = {
  getArrow: function getArrow() {
    var _this = this;
    return function () {
      console.log(_this === obj);
    };
  }
};
```

6. 扩展运算符的作用及使用场景
（1）对象扩展运算符
对象的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
```

上述方法实际上等价于:

```
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
```

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。Object.assign方法的第一个参数是目标对象，后面的参数都是源对象。(如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性)。
同样，如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let bar = {a: 1, b: 2};
let baz = {...bar, ...{a:2, b: 4}};  // {a: 2, b: 4}
```

利用上述特性就可以很方便的修改对象的部分属性。在redux中的reducer函数规定必须是一个纯函数，reducer中的state对象要求不能直接修改，可以通过扩展运算符把修改路径的对象都复制一遍，然后产生一个新的对象返回。
需要注意：扩展运算符对对象实例的拷贝属于浅拷贝。
（**2**）数组扩展运算符
数组的扩展运算符可以将一个数组转为用逗号分隔的参数序列，且每次只能展开一层数组。

```
console.log(...[1, 2, 3])
// 1 2 3
console.log(...[1, [2, 3, 4], 5])
// 1 [2, 3, 4] 5
```

下面是数组的扩展运算符的应用：

- 将数组转换为参数序列

```
function add(x, y) {
  return x + y;
}
const numbers = [1, 2];
add(...numbers) // 3
```

- 复制数组

```
const arr1 = [1, 2];
const arr2 = [...arr1];
```

要记住：扩展运算符**(...)**用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中，这里参数对象是个数组，数组里面的所有对象都是基础数据类型，将所有基础数据类型重新拷贝到新的数组中。

- 合并数组

如果想在数组内合并数组，可以这样：

```
const arr1 = ['two', 'three'];const arr2 = ['one', ...arr1, 'four', 'five'];//
```

["one", "two", "three", "four", "five"]

- 扩展运算符与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];first // 1rest  // [2, 3, 4, 5]
```

需要注意：如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...rest, last] = [1, 2, 3, 4, 5];        // 报错const [first, ...rest, last] = [1, 2, 3, 4, 5];  // 报错
```

- 将字符串转为真正的数组

```
[...'hello']    // [ "h", "e", "l", "l", "o" ]
```

- 任何 **Iterator** 接口的对象，都可以用扩展运算符转为真正的数组

比较常见的应用是可以将某些数据结构转为数组：

```
// arguments对象
function foo() {
  const args = [...arguments];
}
```

用于替换es5中的Array.prototype.slice.call(arguments)写法。

- 使用Math函数获取数组中特定的值

```
const numbers = [9, 4, 7, 1];
Math.min(...numbers); // 1
Math.max(...numbers); // 9
```

7. Proxy 可以实现什么功能?

在 Vue3.0 中通过 Proxy 来替换原本的 Object.defineProperty 来实现数据响应式。

Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

```
let p = new Proxy(target, handler)
```

target 代表需要添加代理的对象，handler 用来自定义对象中的操作，比如可以用来自定义 set 或者 get 函数。

下面来通过 Proxy 来实现一个数据响应式：

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
```

```
    }
  }
  return new Proxy(obj, handler)
}
let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
  }
)
p.a = 2 // 监听到属性a改变
p.a // 'a' = 2
```

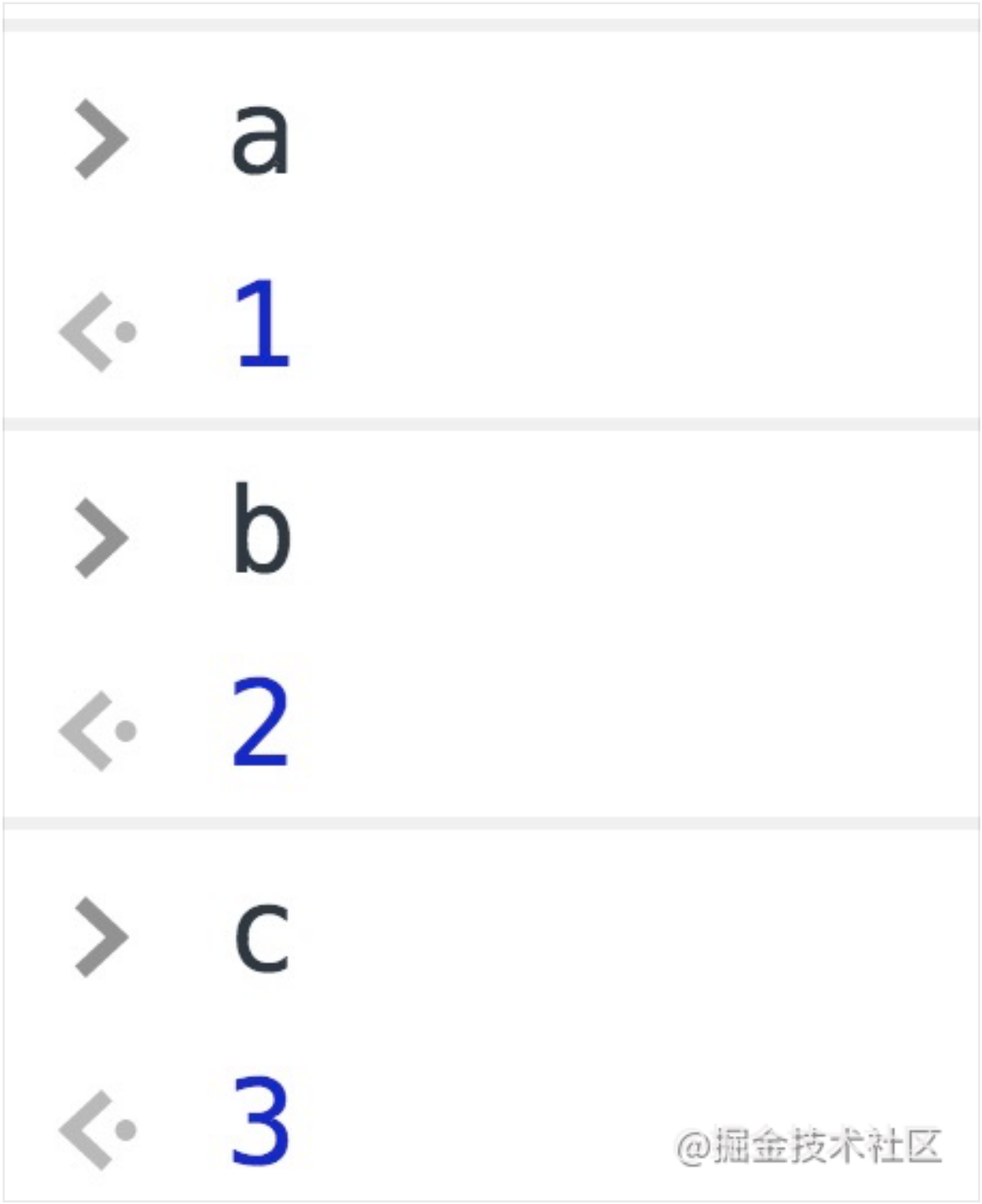
在上述代码中，通过自定义 set 和 get 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对对象任何属性进行读写时发出通知。
当然这是简单版的响应式实现，如果需要实现一个 Vue 中的响应式，需要在 get 中收集依赖，在 set 派发更新，之所以 Vue3.0 要使用 Proxy 替换原本的 API 原因在于 Proxy 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 Proxy 可以完美监听到任何方式的数据改变，唯一缺陷就是浏览器的兼容性不好。

8. 对对象与数组的解构的理解

解构是 ES6 提供的一种新的提取数据的模式，这种模式能够从对象或数组里有针对性地拿到想要的数值。 **1**）数组的解构 在解构数组时，以元素的位置为匹配条件来提取想要的数据的：

```
const [a, b, c] = [1, 2, 3]
```

最终，a、b、c分别被赋予了数组第0、1、2个索引位的值：



数组里的0、1、2索引位的元素值，精准地被映射到了左侧的第0、1、2个变量里去，这就是数组解构的工作模式。还可以通过给左侧变量数组设置空占位的方式，实现对数组中某几个元素的精准提取：

```
const [a,,c] = [1,2,3]
```

通过把中间位留空，可以顺利地把数组第一位和最后一位的值赋给 a、c 两个变量：

2）对象的解构 对象解构比数组结构稍微复杂一些，也更显强大。在解构对象时，是以属性的名称为匹配条件，来提取想要的数据的。现在定义一个对象：

```
const stu = {
  name: 'Bob',
  age: 24
}
```

假如想要解构它的两个自有属性，可以这样：

```
const { name, age } = stu
```

这样就得到了 name 和 age 两个和 stu 平级的变量：

注意，对象解构严格以属性名作为定位依据，所以就算调换了 name 和 age 的位置，结果也是一样的：

```
const { age, name } = stu
```

9. 如何提取高度嵌套的对象里的指定属性?

有时会遇到一些嵌套程度非常深的对象：

```
const school = {
  classes: {
    stu: {
      name: 'Bob',
      age: 24,
    }
  }
}
```

像此处的 name 这个变量，嵌套了四层，此时如果仍然尝试老方法来提取它：

```
const { name } = school
```

显然是不奏效的，因为 school 这个对象本身是没有 name 这个属性的，name 位于 school 对象的“儿子的儿子”对象里面。要想把 name 提取出来，一种比较笨的方法是逐层解构：

```
const { classes } = school
const { stu } = classes
```

```
const { name } = stu
name // 'Bob'
```

但是还有一种更标准的做法，可以用一行代码来解决这个问题：

```
const { classes: { stu: { name } }} = school

console.log(name)  // 'Bob'
```

可以在解构出来的变量名右侧，通过冒号 +{目标属性名}这种形式，进一步解构它，一直解构到拿到目标数据为止。

10. 对 rest 参数的理解

扩展运算符被用在函数形参上时，它还可以把一个分离的参数序列整合成一个数组：

```
function mutiple(...args) {
  let result = 1;
  for (var val of args) {
    result *= val;
  }
  return result;
}
mutiple(1, 2, 3, 4) // 24
```

这里，传入 mutiple 的是四个分离的参数，但是如果在 mutiple 函数里尝试输出 args 的值，会发现它是一个数组：

```
function mutiple(...args) {
  console.log(args)
}
mutiple(1, 2, 3, 4) //[1, 2, 3, 4]
```

这就是 ... rest运算符的又一层威力了，它可以把函数的多个入参收敛进一个数组里。这一点经常用于获取函数的多余参数，或者像上面这样处理函数参数个数不确定的情况。

11. ES6 中模板语法与字符串处理

ES6 提出了“模板语法”的概念。在 ES6 以前，拼接字符串是很麻烦的事情：

```
var name = 'css'
var career = 'coder'
var hobby = ['coding', 'writing']
var finalString = 'my name is ' + name + ', I work as a ' + career + ', I love ' + hobby[0] + ' and ' + hobby[1]
```

仅仅几个变量，写了这么多加号，还要时刻小心里面的空格和标点符号有没有跟错地方。但是有了模板字符串，拼接难度直线下降：

```
var name = 'css'
var career = 'coder'
var hobby = ['coding', 'writing']
```

```
var finalString = `my name is ${name}, I work as a ${career} I love ${hobby[0]} and ${hobby[1]}`
```

字符串不仅更容易拼了，也更易读了，代码整体的质量都变高了。这就是模板字符串的第一个优势——允许用${}的方式嵌入变量。但这还不是问题的关键，模板字符串的关键优势有两个：

- 在模板字符串中，空格、缩进、换行都会被保留
- 模板字符串完全支持“运算”式的表达式，可以在${}里完成一些计算

基于第一点，可以在模板字符串里无障碍地直接写 html 代码：

```
let list = `
      <ul>
            <li>列表项1</li>
            <li>列表项2</li>
      </ul>
`;
console.log(message); // 正确输出，不存在报错
```

基于第二点，可以把一些简单的计算和调用丢进 ${} 来做：

```
function add(a, b) {
  const finalString = `${a} + ${b} = ${a+b}`
  console.log(finalString)
}
add(1, 2) // 输出 '1 + 2 = 3'
```

除了模板语法外， ES6 中还新增了一系列的字符串方法用于提升开发效率：

（1）存在性判定：在过去，当判断一个字符/字符串是否在某字符串中时，只能用 indexOf > -1 来做。现在 ES6 提供了三个方法：includes、startsWith、endsWith，它们都会返回一个布尔值来告诉你是否存在。

- **includes**：判断字符串与子串的包含关系：

```
const son = 'haha'
const father = 'xixi haha hehe'
father.includes(son) // true
```

- **startsWith**：判断字符串是否以某个/某串字符开头：

```
const father = 'xixi haha hehe'
father.startsWith('haha') // false
father.startsWith('xixi') // true
```

- **endsWith**：判断字符串是否以某个/某串字符结尾：

```
const father = 'xixi haha hehe'
  father.endsWith('hehe') // true
```

（2）自动重复：可以使用 repeat 方法来使同一个字符串输出多次（被连续复制多次）：

```
const sourceCode = 'repeat for 3 times;'
const repeated = sourceCode.repeat(3)
console.log(repeated) // repeat for 3 times;repeat for 3 times;repeat for 3 times;
```

67、简单介绍一下 symbol

Symbol 是 ES6 的新增属性，表示独一无二的值，代表用给定名称作为唯一标识，这种类型的值可以这样创建，let id=symbol(“id”)，是一种类似于字符串的数据类型

Symbol 特点

- 1) Symbol 的值是唯一的，用来解决命名冲突的问题 ,比如向第三方对象中添加属性时。
- 2) Symbol 值不能与其他数据进行运算
- 3) Symbol 定义的对象属性不能使用 for...in 循环遍历，但是可以使用 Object.getOwnPropertySymbols 、Reflect.ownKeys 来获取对象的所有键名
- Symbol 不可以后期动态添加属性

87、简单讲一讲 ES6、ES7、ES8 的一些新特性

1️⃣、ES6：

1. ES6 在变量的声明和定义方面增加了 let、const 声明变量，有局部变量的概念，
2. 赋值中有解构赋值，
3. 同时 ES6 对字符串、 数组、正则、对象、函数等拓展了一些方法，如字符串方面的模板字符串、函数方面的默认参数、对象方面属性的简洁表达方式，ES6 也 引入了新的数据类型 symbol，新的数据结构 set 和 map,symbol 可以通过 typeof 检测出来，为解决异步回调问题，引入了 promise 和 generator，
4. 实现 Class 和模块，通过 Class 可以更好的面向对象编程，使用模块加载方便模块化编程，当然考虑到 浏览器兼容性，我们在实际开发中需要使用 babel 进行编译

重要的特性：

1. 块级作用域：ES5 只有全局作用域和函数作用域，块级作用域的好处是不再需要立即执行的函数表达式，循环体中的闭包不再有问题
2. rest 参数：用于获取函数的多余参数，这样就不需要使用 arguments 对象了，
3. promise:一种异步编程的解决方案，比传统的解决方案回调函数和事件更合理强大
4. 模块化：其模块功能主要有两个命令构成，export 和 import，export 命令用于规定模块的

5. 对外接口，import 命令用于输入其他模块提供的功能

2️⃣、ES7新特性：

1. 求幂运算符（**）；例：Math.pow(3, 2) === 3 ** 2　// 9
2. Array.prototype.includes() 方法
3. 函数作用域中严格模式的变更。

3️⃣、ES8新特性

1. **async**、**await**异步解决方案
2. **Object.entries()**：该方法会将某个对象的可枚举属性与值按照二维数组的方式返回。（如果目标对象是数组，则会将数组的下标作为键值返回）
3. **Object.values()**：它的工作原理和Object.entries()方法很像，但是它只返回键值对中的值，结果是一维数组
4. 字符串填充**padStart()**、**padEnd()**：

例：

```
'react'.padStart(10, 'm')      //'mmmmmreact'
'react'.padEnd(10, 'm')       //' reactmmmmm'
```

三、JavaScript基础

1. new操作符的实现原理

new操作符的执行过程：

1、在函数内部创建一个新的空对象，
2、将创建好的对象的__proto__指向该函数的prototype；
3、执行构造函数中的代码，将其中的this指向当前新创建的对象，
4、将该对象作为函数的返回值通过return将该对象返回

具体实现：

```
function objectFactory() {
  let newObject = null;
  let constructor = Array.prototype.shift.call(arguments);
  let result = null;
  // 判断参数是否是一个函数
  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }
  // 新建一个空对象，对象的原型为构造函数的 prototype 对象
  newObject = Object.create(constructor.prototype);
  // 将 this 指向新建对象，并执行函数
  result = constructor.apply(newObject, arguments);
  // 判断返回对象
  let flag = result && (typeof result === "object" || typeof result ===
```

```
"function");
  // 判断返回结果
  return flag ? result : newObject;
}
// 使用方法
objectFactory(构造函数, 初始化参数);
```

2. map和Object的区别

	Map	Object
意外的键	Map默认情况不包含任何键，只包含显式插入的键。	Object 有一个原型, 原型链上的键名有可能和自己在对象上的设置的键名产生冲突。
键的类型	Map的键可以是任意值，包括函数、对象或任意基本类型。	Object 的键必须是String 或是Symbol。
键的顺序	Map 中的 key 是有序的。因此，当迭代的时候， Map 对象以插入的顺序返回键值。	Object 的键是无序的
Size	Map 的键值对个数可以轻易地通过size 属性获取	Object 的键值对个数只能手动计算
迭代	Map 是 iterable 的，所以可以直接被迭代。	迭代Object需要以某种方式获取它的键然后才能迭代。
性能	在频繁增删键值对的场景下表现更好。	在频繁添加和删除键值对的场景下未作出优化。

3. map和weakMap的区别

（1）**Map** map本质上就是键值对的集合，但是普通的Object中的键值对中的键只能是字符串。而ES6提供的Map数据结构类似于对象，但是它的键不限制范围，可以是任意类型，是一种更加完善的Hash结构。如果Map的键是一个原始数据类型，只要两个键严格相同，就视为是同一个键。
实际上Map是一个数组，它的每一个数据也都是一个数组，其形式如下：

```
const map = [
    ["name","张三"],
    ["age",18],
]
```

Map数据结构有以下操作方法：

- **size**： map.size 返回Map结构的成员总数。
- **set(key,value)**：设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返

回的是当前Map对象，所以可以链式调用）
- **get(key)**：该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**：该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**：该方法删除某个键，返回true，如果删除失败，返回false。
- **clear()**：map.clear()清除所有成员，没有返回值。

Map结构原生提供是三个遍历器生成函数和一个遍历方法
- keys()：返回键名的遍历器。
- values()：返回键值的遍历器。
- entries()：返回所有成员的遍历器。
- forEach()：遍历Map的所有成员。

```
const map = new Map([
    ["foo",1],
    ["bar",2],
])
for(let key of map.keys()){
   console.log(key);  // foo bar
}
for(let value of map.values()){
    console.log(value); // 1 2
}
for(let items of map.entries()){
   console.log(items);  //["foo",1] ["bar",2]
}
map.forEach( (value,key,map) => {
    console.log(key,value); // foo 1    bar 2
})
```

（2）**WeakMap** WeakMap 对象也是一组键值对的集合，其中的键是弱引用的。其键必须是对象，原始数据类型不能作为key值，而值可以是任意的。
该对象也有以下几种方法：
- **set(key,value)**：设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前Map对象，所以可以链式调用）
- **get(key)**：该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**：该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**：该方法删除某个键，返回true，如果删除失败，返回false。

其clear()方法已经被弃用，所以可以通过创建一个空的WeakMap并替换原对象来实现清除。
WeakMap的设计目的在于，有时想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。一旦不再需要这两个对象，就必须手动删除这个引

用，否则垃圾回收机制就不会释放对象占用的内存。
而 WeakMap 的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。
总结：

- Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（ null 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

4. JavaScript 有哪些内置对象

全局的对象（ global objects ）或称标准内置对象，不要和 "全局对象（global object）" 混淆。这里说的全局的对象是说在 全局作用域里的对象。全局作用域中的其他对象可以由用户的脚本创建或由宿主程序提供。
标准内置对象的分类：
（1）值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。例如 Infinity、NaN、undefined、null 字面量
（2）函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。例如 eval()、parseFloat()、parseInt() 等
（3）基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。例如 Object、Function、Boolean、Symbol、Error 等
（4）数字和日期对象，用来表示数字、日期和执行数学计算的对象。例如 Number、Math、Date
（5）字符串，用来表示和操作字符串的对象。例如 String、RegExp
（6）可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 Array
（7）使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。 例如 Map、Set、WeakMap、WeakSet
（8）矢量集合，SIMD 矢量集合中的数据会被组织为一个数据序列。 例如 SIMD 等
（9）结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 JSON 编码的数据。例如 JSON 等
（10）控制抽象对象 例如 Promise、Generator 等
（11）反射。例如 Reflect、Proxy
（12）国际化，为了支持多语言处理而加入 ECMAScript 的对象。例如 Intl、Intl.Collator 等
（13）WebAssembly
（14）其他。例如 arguments
总结： js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。一般经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学

计算的单体内置对象如 Math 对象。

5. 常用的正则表达式有哪些?

```
// （1）匹配 16 进制颜色值
var regex = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;

// （2）匹配日期，如 yyyy-mm-dd 格式
var regex = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;

// （3）匹配 qq 号
var regex = /^[1-9][0-9]{4,10}$/g;

// （4）手机号码正则
var regex = /^1[34578]\d{9}$/g;

// （5）用户名正则
var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}$/;
```

6. 对 JSON 的理解

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为 JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

- JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。
- JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

7. JavaScript 脚本延迟加载的方式有哪些?

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- **defer** 属性： 给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。
- **async** 属性： 给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。
- 动态创建 **DOM** 方式： 动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。
- 使用 **setTimeout** 延迟方法： 设置一个定时器来延迟加载js脚本文件
- 让 **JS** 最后加载： 将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

8. JavaScript 类数组对象的定义?

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 arguments 和 DOM 方法的返回结果，还有一个函数也可以被看作是类数组对象，因为它含有 length 属性值，代表可接收的参数个数。
常见的类数组转换为数组的方法有这样几种：
（1）通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

（2）通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

（3）通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

（4）通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

9. 数组有哪些原生方法?

- 数组和字符串的转换方法：toString()、toLocalString()、join() 其中 join() 方法可以指定转换为字符串时的分隔符。
- 数组尾部操作的方法 pop() 和 push()，push 方法可以传入多个参数。
- 数组首部操作的方法 shift() 和 unshift() 重排序的方法 reverse() 和 sort()，sort() 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。
- 数组连接的方法 concat() ，返回的是拼接好的数组，不影响原数组。
- 数组截取办法 slice()，用于截取数组中的一部分返回，不影响原数组。
- 数组插入方法 splice()，影响原数组查找特定项的索引的方法，indexOf() 和 lastIndexOf() 迭代方法 every()、some()、filter()、map() 和 forEach() 方法
- 数组归并方法 reduce() 和 reduceRight() 方法

10. Unicode、UTF-8、UTF-16、UTF-32的区别?

（1）Unicode

在说Unicode之前需要先了解一下ASCII码：ASCII 码（American Standard Code for Information Interchange）称为美国标准信息交换码。

- 它是基于拉丁字母的一套电脑编码系统。
- 它定义了一个用于代表常见字符的字典。
- 它包含了"A-Z"(包含大小写)，数据"0-9" 以及一些常见的符号。
- 它是专门为英语而设计的，有128个编码，对其他语言无能为力

ASCII码可以表示的编码有限，要想表示其他语言的编码，还是要使用Unicode来表示，可以说Unicode是ASCII 的超集。

Unicode全称 Unicode Translation Format，又叫做统一码、万国码、单一码。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

Unicode的实现方式（也就是编码方式）有很多种，常见的是**UTF-8**、**UTF-16**、**UTF-32**和**USC-2**。

（2）UTF-8

UTF-8是使用最广泛的Unicode编码方式，它是一种可变长的编码方式，可以是1—4个字节不等，它可以完全兼容ASCII码的128个字符。

注意： UTF-8 是一种编码方式，Unicode是一个字符集合。

UTF-8的编码规则：

- 对于单字节的符号，字节的第一位为0，后面的7位为这个字符的Unicode编码，因此对于英文字母，它的Unicode编码和ACSII编码一样。
- 对于**n**字节的符号，第一个字节的前n位都是1，第n+1位设为0，后面字节的前两位一律设为10，剩下的没有提及的二进制位，全部为这个符号的Unicode码 。

来看一下具体的Unicode编号范围与对应的UTF-8二进制格式 ：

编码范围（编号对应的十进制数）	二进制格式
0x00—0x7F （0-127）	0xxxxxxx
0x80—0x7FF （128-2047）	110xxxxx 10xxxxxx
0x800—0xFFFF （2048-65535）	1110xxxx 10xxxxxx 10xxxxxx
0x10000—0x10FFFF （65536以上）	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

那该如何通过具体的Unicode编码，进行具体的UTF-8编码呢？步骤如下：

- 找到该Unicode编码的所在的编号范围，进而找到与之对应的二进制格式
- 将Unicode编码转换为二进制数（去掉最高位的0）
- 将二进制数从右往左一次填入二进制格式的X中，如果有X未填，就设为0

来看一个实际的例子：“马”字的Unicode编码是：0x9A6C，整数编号是39532 （1）首选确定了该字符在第三个范围内，它的格式是 1110xxxx 10xxxxxx 10xxxxxx （2）39532对应的二进制数为1001 1010 0110 1100 （3）将二进制数填入X中，结果是：11101001 10101001 10101100

（3）UTF-16

1. 平面的概念
在了解UTF-16之前，先看一下平面的概念： Unicode编码中有很多很多的字符，它并不是一次性定义的，而是分区进行定义的，每个区存放 **65536**（216）个字符，这称为一个平面，目前总共有17 个平面。
最前面的一个平面称为基本平面，它的码点从 **0 — 216-1**，写成16进制就是U+0000 — U+FFFF，那剩下的16个平面就是辅助平面，码点范围是U+10000—U+10FFFF。
2. UTF-16 概念：
UTF-16也是Unicode编码集的一种编码形式，把Unicode字符集的抽象码位映射为16位长的整数（即码元）的序列，用于数据存储或传递。Unicode字符的码位需要1个或者2个16位长的码元来表示，因此UTF-16也是用变长字节表示的。
3. UTF-16 编码规则：
- 编号在 U+0000—U+FFFF 的字符（常用字符集），直接用两个字节表示。
- 编号在 U+10000—U+10FFFF 之间的字符，需要用四个字节表示。

4. 编码识别
那么问题来了，当遇到两个字节时，怎么知道是把它当做一个字符还是和后面的两个字节一起当做一个字符呢?
UTF-16 编码肯定也考虑到了这个问题，在基本平面内，从 U+D800 — U+DFFF 是一个空段，也就是说这个区间的码点不对应任何的字符，因此这些空段就可以用来映射辅助平面的字符。
辅助平面共有 **220** 个字符位，因此表示这些字符至少需要 20 个二进制位。
UTF-16 将这 20 个二进制位分成两半，前 10 位映射在 U+D800 — U+DBFF，称为高位（H），后 10 位映射在 U+DC00 — U+DFFF，称为低位（L）。这就相当于，将一个辅助平面的字符拆成了两个基本平面的字符来表示。
因此，当遇到两个字节时，发现它的码点在 U+D800 —U+DBFF之间，就可以知道，它后面的两个字节的码点应该在 U+DC00 — U+DFFF 之间，这四个字节必须放在一起进行解读。
5. 举例说明
以 "[?]" 字为例，它的 Unicode 码点为 0x21800，该码点超出了基本平面的范围，因此需要用四个字节来表示，步骤如下：
- 首先计算超出部分的结果：0x21800 - 0x10000
- 将上面的计算结果转为20位的二进制数，不足20位就在前面补0，结果为：0001000110 0000000000
- 将得到的两个10位二进制数分别对应到两个区间中
- U+D800 对应的二进制数为 1101100000000000， 将0001000110填充在它的后10 个二进制位，得到 1101100001000110，转成 16 进制数为 0xD846。同理，低位为 0xDC00，所以这个字的UTF-16 编码为 0xD846 0xDC00

（4） UTF-32
UTF-32 就是字符所对应编号的整数二进制形式，每个字符占四个字节，这个是直接进行转换的。该编码方式占用的储存空间较多，所以使用较少。
比如“马”字的Unicode编号是：U+9A6C，整数编号是39532，直接转化为

二进制：1001 1010 0110 1100，这就是它的UTF-32编码。

（**5**）总结

Unicode、**UTF-8**、**UTF-16**、**UTF-32**有什么区别?

- Unicode 是编码字符集（字符集），而UTF-8、UTF-16、UTF-32是字符集编码（编码规则）；
- UTF-16 使用变长码元序列的编码方式，相较于定长码元序列的UTF-32算法更复杂，甚至比同样是变长码元序列的UTF-8也更为复杂，因为其引入了独特的代理对这样的代理机制；
- UTF-8需要判断每个字节中的开头标志信息，所以如果某个字节在传送过程中出错了，就会导致后面的字节也会解析出错；而UTF-16不会判断开头标志，即使错也只会错一个字符，所以容错能力教强；
- 如果字符内容全部英文或英文与其他文字混合，但英文占绝大部分，那么用UTF-8就比UTF-16节省了很多空间；而如果字符内容全部是中文这样类似的字符或者混合字符中中文占绝大多数，那么UTF-16就占优势了，可以节省很多空间；

11. 常见的位运算符有哪些？其计算规则是什么?

现代计算机中数据都是以二进制的形式存储的，即0、1两种状态，计算机对二进制数据进行的运算加减乘除等都是叫位运算，即将符号位共同参与运算的运算。

常见的位运算有以下几种：

运算符	描述	运算规则
&	与	两个位都为1时，结果才为1
`	`	或
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进制位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃

1. 按位与运算符（**&**）

定义： 参加运算的两个数据按二进制位进行“与”运算。 运算规则：

0 & 0 = 0

0 & 1 = 0

1 & 0 = 0

1 & 1 = 1

总结：两位同时为1，结果才为1，否则结果为0。 例如：3&5 即：

0000 0011

  0000 0101

 = 0000 0001

因此 3&5 的值为1。 注意：负数按补码形式参加按位与运算。
用途：
（**1**）判断奇偶
只要根据最未位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用if ((i & 1) == 0)代替if (i % 2 == 0)来判断a是不是偶数。
（**2**）清零
如果想将一个单元清零，即使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。
2. 按位或运算符（**|**）
定义： 参加运算的两个对象按二进制位进行“或”运算。
运算规则：
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

总结：参加运算的两个对象只要有一个为1，其值为1。 例如：3|5即：
0000 0011
 0000 0101
= 0000 0111

因此，3|5的值为7。 注意：负数按补码形式参加按位或运算。
3. 异或运算符（**^**）
定义： 参加运算的两个数据按二进制位进行“异或”运算。
运算规则：
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

总结：参加运算的两个对象，如果两个相应位相同为0，相异为1。 例如：3|5即：
0000 0011
 0000 0101
= 0000 0110

因此，3^5的值为6。 异或运算的性质：

- 交换律：(a^b)^c == a^(b^c)
- 结合律：(a + b)^c == a^b + b^c
- 对于任何数x，都有 x^x=0，x^0=x
- 自反性: a^b^b=a^0=a;

4. 取反运算符 **(~)**
定义： 参加运算的一个数据按二进制进行“取反”运算。

运算规则：
~ 1 = 0~ 0 = 1

总结：对一个二进制数按位取反，即将0变1，1变0。 例如：~6 即：
0000 0110= 1111 1001

在计算机中，正数用原码表示，负数使用补码存储，首先看最高位，最高位1表示负数，0表示正数。此计算机二进制码为负数，最高位为符号位。 当发现按位取反为负数时，就直接取其补码，变为十进制：
0000 0110 = 1111 1001反码：1000 0110补码：1000 0111

因此，~6的值为-7。
5. 左移运算符（**<<**）
定义： 将一个运算对象的各二进制位全部左移若干位，左边的二进制位丢弃，右边补0。 设 a=1010 1110，a = a<< 2 将a的二进制位左移2位、右补0，即得a=1011 1000。 若左移时舍弃的高位不包含1，则每左移一位，相当于该数乘以2。
6. 右移运算符（**>>**）
定义： 将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。 例如：a=a>>2 将a的二进制位右移2位，左补0 或者 左补1得看被移数是正还是负。 操作数每右移一位，相当于该数除以2。
7. 原码、补码、反码
上面提到了补码、反码等知识，这里就补充一下。 计算机中的有符号数有三种表示方法，即原码、反码和补码。三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，而数值位，三种表示方法各不相同。
（**1**）原码
原码就是一个数的二进制数。例如：10的原码为0000 1010
（**2**）反码
- 正数的反码与原码相同，如：10 反码为 0000 1010
- 负数的反码为除符号位，按位取反，即0变1，1变0。

例如：-10
原码：1000 1010
反码：1111 0101

（**3**）补码
- 正数的补码与原码相同，如：10 补码为 0000 1010
- 负数的补码是原码除符号位外的所有位取反即0变1，1变0，然后加1，也就是反码加1。

例如：-10
原码：1000 1010
反码：1111 0101
补码：1111 0110

12. 为什么函数的 arguments 参数是类数组而不是数组？如何遍历类数组?

arguments是一个对象，它的属性是从 0 开始依次递增的数字，还有callee和length等属性，与数组相似；但是它却没有数组常见的方法属性，如forEach, reduce等，所以叫它们类数组。
要遍历类数组，有三个方法：
（1）将数组的方法应用到类数组上，这时候就可以使用call和apply方法，如：

```
function foo(){
  Array.prototype.forEach.call(arguments, a => console.log(a))
}
```

（2）使用Array.from方法将类数组转化成数组：

```
function foo(){
  const arrArgs = Array.from(arguments)
  arrArgs.forEach(a => console.log(a))
}
```

（3）使用展开运算符将类数组转化成数组

```
function foo(){
    const arrArgs = [...arguments]
    arrArgs.forEach(a => console.log(a))
}
```

13. 什么是 DOM 和 BOM?

- DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。
- BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。BOM的核心是window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

14. 对类数组对象的理解，如何转化为数组

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有arguments 和 DOM 方法的返回结果，函数参数也可以被看作是类数组对象，因为它含有 length属性值，代表可接收的参数个数。
常见的类数组转换为数组的方法有这样几种：

- 通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

- 通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

- 通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

- 通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

39、数组去重

法一：indexOf 循环去重
法二：ES6 Set 去重；Array.from(new Set(array))
法三：Object 键值对去重；把数组的值存成 Object 的 key 值，比如 Object[value1] = true，在判断另一个值的时候，如果 Object[value2]存在的话，就说明该值是重复的。

15. escape、encodeURI、encodeURIComponent 的区别

- encodeURI 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。
- encodeURIComponent 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。
- escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，escape 是直接在字符的 unicode 编码前加上 %u，而 encodeURI 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %。

16. 对 AJAX 的理解，实现一个 AJAX 请求

AJAX 是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。
创建 AJAX 请求的步骤：

- 创建一个 **XMLHttpRequest** 对象。
- 在这个对象上使用 **open** 方法创建一个 **HTTP** 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。
- 在发起请求前，可以为这个对象添加一些信息和监听函数。比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。
- 当对象的属性和监听函数设置完成后，最后调用 **sent** 方法来向服务器发起请求，可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", url, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};
// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
```

使用 Promise 封装 AJAX：

```
// promise 封装实现：
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
```

```
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
    xhr.responseType = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
  return promise;
}
```

17. JavaScript为什么要进行变量提升，它导致了什么问题?

变量提升的表现是，无论在函数中何处位置声明的变量，好像都被提升到了函数的首部，可以在变量声明前访问到而不会报错。

造成变量声明提升的本质原因是 js 引擎在代码执行前有一个解析的过程，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当访问一个变量时，会到当前执行上下文中的作用域链中去查找，而作用域链的首端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。

首先要知道，JS在拿到一个变量或者一个函数的时候，会有两步操作，即解析和执行。

- 在解析阶段，JS会检查语法，并对函数进行预编译。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出this、arguments和函数的参数。
  - 全局上下文：变量定义，函数声明
  - 函数上下文：变量定义，函数声明，this，arguments
- 在执行阶段，就是按照代码的顺序依次执行。

那为什么会进行变量提升呢？主要有以下两个原因：

- 提高性能
- 容错性更好

（**1**）提高性能 在JS代码执行之前，会进行语法检查和预编译，并且这一操作只进行一次。这么做就是为了提高性能，如果没有这一步，那么每次执行代码前都必须重新解析一遍该变量（函数），而这是没有必要的，因为变量（函数）的代码并不会改变，解析一遍就够了。

在解析的过程中，还会为函数生成预编译代码。在预编译时，会统计声明了哪些变量、创建了哪些函数，并对函数的代码进行压缩，去除注释、不必要的空白等。这样做的好处就是每次执行函数时都可以直接为该函数分配栈空间（不需要再解析一遍去获取代码中声明了哪些变量，创建了哪些函数），并且因为代码压缩的原因，代码执行也更快了。

（**2**）容错性更好

变量提升可以在一定程度上提高JS的容错性，看下面的代码：

```
a = 1;var a;console.log(a);
```

如果没有变量提升，这两行代码就会报错，但是因为有了变量提升，这段代码

就可以正常执行。
虽然，在可以开发过程中，可以完全避免这样写，但是有时代码很复杂的时候。可能因为疏忽而先使用后定义了，这样也不会影响正常使用。由于变量提升的存在，而会正常运行。
总结：

- 解析和预编译过程中的声明提升可以提高性能，让函数可以在执行时预先为变量分配栈空间
- 声明提升还可以提高JS代码的容错性，使一些不规范的代码也可以正常执行

变量提升虽然有一些优点，但是他也会造成一定的问题，在ES6中提出了let、const来定义变量，它们就没有变量提升的机制。下面看一下变量提升可能会导致的问题：

```
var tmp = new Date();

function fn(){
    console.log(tmp);
    if(false){
        var tmp = 'hello world';
    }
}


fn();  // undefined
```

在这个函数中，原本是要打印出外层的tmp变量，但是因为变量提升的问题，内层定义的tmp被提到函数内部的最顶部，相当于覆盖了外层的tmp，所以打印结果为undefined。

```
var tmp = 'hello world';

for (var i = 0; i < tmp.length; i++) {
    console.log(tmp[i]);
}

console.log(i); // 11
```

由于遍历时定义的i会变量提升成为一个全局变量，在函数结束之后不会被销毁，所以打印出来11。

18. 什么是尾调用，使用尾调用有什么好处?

尾调用指的是函数的最后一步调用另一个函数。代码执行是基于执行栈的，所以当在一个函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这时可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

19. ES6模块与CommonJS模块有什么异同?

ES6 Module和CommonJS模块的区别：

- CommonJS是对模块的浅拷贝，ES6 Module是对模块的引用，即ES6 Module只存只读，不能改变其值，也就是指针指向不能变，类似const；
- import的接口是read-only（只读状态），不能修改其变量值。 即不能修改其变量的指针指向，但可以改变变量内部指针指向，可以对commonJS对重新赋值（改变指针指向），但是对ES6 Module赋值会编译报错。

ES6 Module和CommonJS模块的共同点：

- CommonJS和ES6 Module都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

20. 常见的DOM操作有哪些

1）DOM 节点的获取

DOM 节点的获取的API及使用：

```
getElementById // 按照 id 查询
getElementsByTagName // 按照标签名查询
getElementsByClassName // 按照类名查询
querySelectorAll // 按照 css 选择器查询

// 按照 id 查询
var imooc = document.getElementById('imooc') // 查询到 id 为 imooc 的元素
// 按照标签名查询
var pList = document.getElementsByTagName('p')  // 查询到标签为 p 的集合
console.log(divList.length)
console.log(divList[0])
// 按照类名查询
var moocList = document.getElementsByClassName('mooc') // 查询到类名为 mooc 的集合
// 按照 css 选择器查询
var pList = document.querySelectorAll('.mooc') // 查询到类名为 mooc 的集合
```

2）DOM 节点的创建

创建一个新节点，并把它添加到指定节点的后面。 已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

要求添加一个有内容的 span 节点到 id 为 title 的节点后面，做法就是：

```
// 首先获取父节点
var container = document.getElementById('container')
// 创建新节点
var targetSpan = document.createElement('span')
// 设置 span 节点的内容
targetSpan.innerHTML = 'hello world'
// 把新创建的元素塞进父节点里去
container.appendChild(targetSpan)
```

3）DOM 节点的删除

删除指定的 **DOM** 节点， 已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

需要删除 id 为 title 的元素，做法是：

```
// 获取目标元素的父元素
var container = document.getElementById('container')
// 获取目标元素
var targetNode = document.getElementById('title')
// 删除目标元素
container.removeChild(targetNode)
```

或者通过子节点数组来完成删除：

// 获取目标元素的父元素 var container = document.getElementById('container')// 获取目标元素 var targetNode = container.childNodes[1]// 删除目标元素 container.removeChild(targetNode)

4）修改 **DOM** 元素

修改 DOM 元素这个动作可以分很多维度，比如说移动 DOM 元素的位置，修改 DOM 元素的属性等。

将指定的两个 **DOM** 元素交换位置， 已知的 HTML 结构如下：

```
<html>
  <head>
```

```
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
      <p id="content">我是内容</p>
    </div>
  </body>
</html>
```

现在需要调换 title 和 content 的位置，可以考虑 insertBefore 或者 appendChild：

```
// 获取父元素
var container = document.getElementById('container')

// 获取两个需要被交换的元素
var title = document.getElementById('title')
var content = document.getElementById('content')
// 交换两个元素，把 content 置于 title 前面
container.insertBefore(content, title)
```

80、什么是事件模型，DOM0 级和 DOM2 级有什么区别，DOM 的分级是什么

事件模型：从事件发生开始，到所有处理函数执行完毕所经历的过程
包括如下三个阶段：

- 事件捕获阶段
- 处于目标阶段
- 事件冒泡阶段

JSDOM 标准事件流的触发的先后顺序为：先捕获再冒泡，点击 DOM 节点时，事件传播

顺序：事件捕获阶段，从上往下传播，然后到达事件目标节点，最后是冒泡阶段，从下往上传播

DOM0 级和 DOM2 级

共同点：能添加多个事件处理程序，按顺序执行，HTML 事件处理程序无法做到
区别：

DOM0 级事件处理：同时绑定几个不同的事件，例如在绑定 onclick 的基础上再绑定一个 onmouseover 为按钮 2 设置背景颜色（这里注意不能 onclick、onmouseover 事件都设为 alert 弹出哦，可能有冲突，dom0 和 dom2 都不能成功）；但是不能同时绑定多个相同的事件，比如 onclick；会覆盖，只会执行最后一个的函数，DOM0 级事件处理程序**只支持冒泡。

DOM2级事件处理：优点：同时绑定几个事件（相同或不同），然后顺序执行，不会覆盖。缺点：不具有跨浏览器优势，可以指定捕获阶段处理还是冒泡阶段处理。IE9能兼容dom2

21. use strict是什么意思？使用它区别是什么?

use strict 是一种 ECMAscript5 添加的（严格模式）运行模式，这种模式使得 Javascript 在更严格的条件下运行。设立严格模式的目的如下：

- 消除 Javascript 语法的不合理、不严谨之处，减少怪异行为;
- 消除代码运行的不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 Javascript 做好铺垫。

区别：

- 禁止使用 with 语句。
- 禁止 this 关键字指向全局对象。
- 对象不能有重名的属性。

22. 如何判断一个对象是否属于某个类?

- 第一种方式，使用 instanceof 运算符来判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。
- 第二种方式，通过对象的 constructor 属性来判断，对象的 constructor 属性指向该对象的构造函数，但是这种方式不是很安全，因为 constructor 属性可以被改写。
- 第三种方式，如果需要判断的是某个内置的引用类型的话，可以使用 Object.prototype.toString() 方法来打印对象的[[Class]] 属性来进行判断。

23. 强类型语言和弱类型语言的区别

- 强类型语言：强类型语言也称为强类型定义语言，是一种总是强制类型定义的语言，要求变量的使用要严格符合定义，所有变量都必须先定义后使用。Java和C++等语言都是强制类型定义的，也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。例如你有一个整数，如果不显式地进行转换，你不能将其视为一个字符串。
- 弱类型语言：弱类型语言也称为弱类型定义语言，与强类型定义相反。JavaScript语言就属于弱类型语言。简单理解就是一种变量类型可以被忽略的语言。比如JavaScript是弱类型定义的，在JavaScript中就可以将字符串'12'和整数3进行连接得到字符串'123'，在相加的时候会进行强制类型转换。

两者对比：强类型语言在速度上可能略逊色于弱类型语言，但是强类型语言带来的严谨性可以有效地帮助避免许多错误。

24. 解释性语言和编译型语言的区别

（1）解释型语言 使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行。是代码在执行时才被解释器一行行动态翻译和执行，而不是在执行之前就完成翻译。解释型语言不需要事先编译，其直接将源代码解释成机器码并立即执行，所以只要某一平台提供了相应的解释器即可运行该程序。其特点总结如下

- 解释型语言每次运行都需要将源代码解释称机器码并执行，效率较低；

- 只要平台提供相应的解释器，就可以运行源代码，所以可以方便源程序移植；
- JavaScript、Python等属于解释型语言。

（2）编译型语言 使用专门的编译器，针对特定的平台，将高级语言源代码一次性的编译成可被该平台硬件执行的机器码，并包装成该平台所能识别的可执行性程序的格式。在编译型语言写的程序执行之前，需要一个专门的编译过程，把源代码编译成机器语言的文件，如exe格式的文件，以后要再运行时，直接使用编译结果即可，如直接运行exe文件。因为只需编译一次，以后运行时不需要编译，所以编译型语言执行效率高。其特点总结如下：

- 一次性的编译成平台相关的机器语言文件，运行时脱离开发环境，运行效率高；
- 与特定平台相关，一般无法移植到其他平台；
- C、C++等属于编译型语言。

两者主要区别在于： 前者源程序编译后即可在该平台运行，后者是在运行期间才编译。所以前者运行速度快，后者跨平台性好。

25. for...in和for...of的区别

for...of 是ES6新增的遍历方式，允许遍历一个含有iterator接口的数据结构（数组、对象等）并且返回各项的值，和ES3中的for...in的区别如下

- for...of 遍历获取的是对象的键值，for...in 获取的是对象的键名；
- for... in 会遍历对象的整个原型链，性能非常差不推荐使用，而 for ... of 只遍历当前对象不会遍历原型链；
- 对于数组的遍历，for...in 会返回数组中所有可枚举的属性 (包括原型链上可枚举的属性)，for...of 只返回数组的下标对应的属性值；

总结： for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

26. 如何使用for...of遍历对象

for...of是作为ES6新增的遍历方式，允许遍历一个含有iterator接口的数据结构（数组、对象等）并且返回各项的值，普通的对象用for..of遍历是会报错的。

如果需要遍历的对象是类数组对象，用Array.from转成数组即可。

```
var obj = {
    0:'one',
    1:'two',
    length: 2
};
obj = Array.from(obj);
for(var k of obj){
    console.log(k)
}
```

如果不是类数组对象，就给对象添加一个[Symbol.iterator]属性，并指向一个迭代器即可。

```
//方法一：
var obj = {
    a:1,
```

```
    b:2,
    c:3
};

obj[Symbol.iterator] = function(){
        var keys = Object.keys(this);
        var count = 0;
        return {
                next(){
                        if(count<keys.length){
                                return {value: obj[keys[count++]],done:false};
                        }else{
                                return {value:undefined,done:true};
                        }
                }
        }
};

for(var k of obj){
        console.log(k);
}


// 方法二
var obj = {
    a:1,
    b:2,
    c:3
};
obj[Symbol.iterator] = function*(){
    var keys = Object.keys(obj);
    for(var k of keys){
        yield [k,obj[k]]
    }
};

for(var [k,v] of obj){
    console.log(k,v);
}
```

27. ajax、axios、fetch的区别

（1）**AJAX** Ajax 即“AsynchronousJavascriptAndXML”（异步 JavaScript

和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

- 本身是针对MVC编程，不符合前端MVVM的浪潮
- 基于原生XHR开发，XHR本身的架构不清晰
- 不符合关注分离（Separation of Concerns）的原则
- 配置和调用方式非常混乱，而且基于事件的异步模型不友好。

（2）**Fetch** fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多。**fetch**不是**ajax**的进一步封装，而是原生**js**，没有使用**XMLHttpRequest**对象。

fetch的优点：

- 语法简洁，更加语义化
- 基于标准 Promise 实现，支持 async/await
- 更加底层，提供的API丰富（request, response）
- 脱离了XHR，是ES规范里新的实现方式

fetch的缺点：

- fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
- fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: 'include'})
- fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
- fetch没有办法原生监测请求的进度，而XHR可以

（3）**Axios** Axios 是一种基于Promise封装的HTTP客户端，其特点如下：

- 浏览器端发起XMLHttpRequests请求
- node端发起http请求
- 支持Promise API
- 监听请求和返回
- 对请求和返回进行转化
- 取消请求
- 自动转换json数据
- 客户端支持抵御XSRF攻击

28. 数组的遍历方法有哪些

方法	是否改变原数组	特点
forEach()	否	数组方法，不改变原数组，没有返回值
map()	否	数组方法，不改变原数组，有返回值，可链式调用

filter()	否	数组方法，过滤数组，返回包含符合条件的元素的数组，可链式调用
for...of	否	for...of 遍历具有 Iterator 迭代器的对象的属性，返回的是数组的元素、对象的属性值，不能遍历普通的 obj 对象，将异步循环变成同步循环
every() 和 some()	否	数组方法，some() 只要有一个是 true，便返回 true；而 every() 只要有一个是 false，便返回 false.
find() 和 findIndex()	否	数组方法，find() 返回的是第一个符合条件的值；findIndex() 返回的是第一个返回条件的值的索引值
reduce() 和 reduceRight()	否	数组方法，reduce() 对数组正序操作；reduceRight() 对数组逆序操作

遍历方法的详细解释：《细数 JavaScript 中那些遍历和循环》

29. forEach 和 map 方法有什么区别

这方法都是用来遍历数组的，两者区别如下：

- forEach() 方法会针对每一个元素执行提供的函数，对数据的操作会改变原数组，该方法没有返回值；
- map() 方法不会改变原数组的值，返回一个新数组，新数组中的值为原数组调用函数处理之后的值；

6、说说前端中的事件流

HTML 中与 javascript 交互是通过事件驱动来实现的，例如鼠标点击事件 onclick、页面的滚动事件 onscroll 等等，可以向文档或者文档中的元素添加事件侦听器来预订事件。
想要知道这些事件是在什么时候进行调用的，就需要了解一下“事件流”的概念。

事件流：事件流描述的是从页面中接收事件的顺序,DOM2 级事件流包括下面几个阶段。

- 事件捕获阶段
- 处于目标阶段
- 事件冒泡阶段

7、如何让事件先冒泡后捕获

在 DOM 标准事件模型中，是先捕获后冒泡。但是如果要实现先冒泡后捕获的效果，对于同一个事件，监听捕获和冒泡，分别对应相应的处理函数，监听到捕获事件，先暂缓执行，直到冒泡事件被捕获后再执行捕获之间。

8、 事件委托以及冒泡原理。

事件委托是利用冒泡阶段的运行机制来实现的，就是把一个元素响应事件的函数委托到另一个元素，一般是把一组元素的事件委托到他的父元素上，

委托的优点是减少内存消耗，节约效率

动态绑定事件

事件冒泡，就是元素自身的事件被触发后，如果父元素有相同的事件，如 onclick 事件，那么元素本身的触发状态就会传递，也就是冒到父元素，父元素的相同事件也会一级一级根据嵌套关系向外触发，直到 document/window，冒泡过程结束。

68、什么是事件监听

addEventListener() 方法，用于向指定元素添加事件句柄，它可以更简单的控制事件，语法为

- element.addEventListener(event, function, useCapture);
  - 第一个参数是事件的类型 (如 "click" 或 "mousedown").
  - 第二个参数是事件触发后调用的函数。
  - 第三个参数是个布尔值用于描述事件是冒泡还是捕获。该参数是可选的。

事件传递有两种方式，冒泡和捕获事件传递定义了元素事件触发的顺序，如果你将 P 元素插入到 div 元素中，用户点击 P元素，在冒泡中，内部元素先被触发，然后再触发外部元素，捕获中，外部元素先被触发，在触发内部元素。

12、JS 的各种位置，比如 clientHeight,scrollHeight,offsetHeight ,以及 scrollTop,, offsetTop,clientTop 的区别?

- clientHeight：表示的是可视区域的高度，不包含 border 和滚动条
- offsetHeight：表示可视区域的高度，包含了 border 和滚动条
- scrollHeight：表示了所有区域的高度，包含了因为滚动被隐藏的部分。
- clientTop：表示边框 border 的厚度，在未指定的情况下一般为 0

- scrollTop：滚动后被隐藏的高度，获取对象相对于由 offsetParent 属性指定的父坐标(css定位的元素或 body 元素)距离顶端的高度。

13、 JS 拖拽功能的实现

首先是三个事件，分别是 mousedown，mousemove，mouseup

当鼠标点击按下的时候，需要一个 tag 标识此时已经按下，可以执行 mousemove 里面的具体方法。

clientX，clientY 标识的是鼠标的坐标，分别标识横坐标和纵坐标，并且我们用 offsetX 和 offsetY 来表示元素的元素的初始坐标，

移动的举例应该是：

- 鼠标移动时候的坐标-鼠标按下去时候的坐标。

也就是说定位信息为：

- 鼠标移动时候的坐标-鼠标按下去时候的坐标+元素初始情况下的 offetLeft. 还有一点也是原理性的东西，也就是拖拽的同时是绝对定位，我们改变的是绝对定位条件下的 left 以及 top 等等值。

补充：也可以通过 html5 的拖放（Drag 和 drop）来实现

16、JS 的节流和防抖

节流：指连续触发事件多次，但是在 n 秒内只执行一次函数，

函数节流的应用场景（间隔一段时间执行一次回调）：

- 滚动加载，加载更多或滚到底部监听
- 谷歌搜索框，搜索联想功能
- 高频点击提交，表单重复提交

防抖：就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

- 简单的说，当一个动作连续触发，则只执行最后一次。

常见实用场景，有滚动加载、搜索框输入、窗口大小拖拽 **Resize**。

如何理解前端模块化

前端模块化就是复杂的文件编程一个一个独立的模块，比如 JS 文件等等，分成独立的模块有利于重用（复用性）和维护（版本迭代），这样会引来模块之间相互依赖的问题，所以有了 commonJS 规范，AMD，CMD 规范等等，以及用于 JS 打包（编译等处理）的工具 webpack

20、说一下 CommonJS、AMD 和 CMD

一个模块是能实现特定功能的文件，有了模块就可以方便的使用别人的代码，想要什么功能就能加载什么模块。

CommonJS：开始于服务器端的模块化，同步定义的模块化，每个模块都是一个单独的作用域，模块输出，modules.exports，模块加载

require()引入模块。

AMD：中文名异步模块定义的意思。

requireJS 实现了 AMD 规范，主要用于解决下述两个问题。

1.多个文件有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器

2.加载的时候浏览器会停止页面渲染，加载文件越多，页面失去响应的时间越长。

语法：requireJS 定义了一个函数 define，它是全局变量，用来定义模块。

requireJS 的例子：

```
//定义模块
define(['dependency'], function(){
    var name = 'Byron';
    function printName(){
        console.log(name);
    }
    return {
        printName: printName
    };
});
//加载模块
require(['myModule'], function (my){
    my.printName();
}
```

RequireJS 定义了一个函数 define,它是全局变量，用来定义模块：

```
define(id?dependencies?,factory)
```

在页面上使用模块加载函数：

```
require([dependencies],factory)；
```

总结 AMD 规范：require（）函数在加载依赖函数的时候是异步加载的，这样浏览器不会失去响应，它指定的回调函数，只有前面的模块加载成功，才会去执行。因为网页在加载 JS 的时候会停止渲染，因此我们可以通过异步的方式去加载 JS,而如果需要依赖某些，也是异步去依赖，依赖后再执行某些方法。

21、对象深度克隆的简单实现（包装对象，Date 对象，正则对象）

1️⃣、JSON.parse(JSON.stringify())

原理： 用 JSON.stringify 将对象转成 JSON 字符串，再用 JSON.parse() 把字符串解析成对象，一去一来，新的对象产生了，而且对象会开辟新的栈，实现深拷贝。

这种方法虽然可以实现数组或对象深拷贝,但不能处理函数，这是因为 JSON.stringify() 方法是将一个 JavaScript 值 (对象或者数组) 转换为一个

JSON字符串，不能接受函数
2️⃣、递归方法：遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝

```
function deepClone(obj){
    var newObj= obj instanceof Array?[]:{};
    for(var i in obj){
        newObj[i]=typeof obj[i]=='object'?
        deepClone(obj[i]):obj[i];
    }
    return newObj;
}
```

缺陷：不能实现例如包装对象 Number,String,Boolean,以及正则对象 RegExp 和 Date 对象的克隆
3️⃣、函数库 lodash　该函数库也有提供_.cloneDeep 用来做 Deep Copy
4️⃣、Object.assign。 深读拷贝

```
function deepClone(obj){
    var newObj= obj instanceof Array ? []:{};
    for(var item in obj){
        var temple= typeof obj[item] == 'object' ?
deepClone(obj[item]):obj[item];
        newObj[item] = temple;
    }
    return newObj;
}
```

5️⃣、valueof() 函数
所有对象都有 valueOf 方法，valueOf 方法对于：如果存在任意原始值，它就默认将对象转换为表示它的原始值。对象是复合值，而且大多数对象无法真正表示为一个原始值，因此默认的 valueOf() 方法简单地返回对象本身，而不是返回一个原始值。数组、函数和正则表达式简单地继承了这个默认方法，调用这些类型的实例的 valueOf() 方法只是简单返回这个对象本身。

- 对于原始值或者包装类：

```
function baseClone(base){
    return base.valueOf();
}
//Number
var num=new Number(1);
var newNum=baseClone(num);    //newNum->1
//String
```

```
var str=new String('hello');
var newStr=baseClone(str);    // newStr->"hello"
//Boolean
var bol=new Boolean(true);
var newBol=baseClone(bol);    //newBol-> true
```
其实对于包装类，完全可以用=号来进行克隆，其实没有深度克隆一说，这里用 valueOf 实现，语法上比较符合规范。

- 对于 Date 类型：
  因为 valueOf 方法，日期类定义的 valueOf() 方法会返回它的一个内部表示：1970 年 1 月1 日以来的毫秒数.因此我们可以在 Date 的原型上定义克隆的方法：

```
Date.prototype.clone=function(){
    return new Date(this.valueOf());
}
var date=new Date('2010');
var newDate=date.clone();     // newDate-> Fri Jan 01 2010 08:00:00 GMT+0800
```

- 对于正则对象 RegExp：

```
RegExp.prototype.clone = function() {
    var pattern = this.valueOf();
    var flags = '';
    flags += pattern.global ? 'g' : '';
    flags += pattern.ignoreCase ? 'i' : '';
    flags += pattern.multiline ? 'm' : '';
    return new RegExp(pattern.source, flags);
};
var reg=new RegExp('/111/');
var newReg=reg.clone();    //newReg-> /\/111\//
```

24、JS 监听对象属性的改变

我们假设这里有一个 user 对象,
(1) 在 ES5 中可以通过 **Object.defineProperty** 来实现已有属性的监听

```
Object.defineProperty(user,'name',{
    set：function(key,value){
    }
})
```
缺点：如果 id 不在 user 对象中，则不能监听 id 的变化

(2) 在 ES6 中可以通过 Proxy 来实现
```
var user = new Proxy({}，{
    set：function(target,key,value,receiver){
    }
})
```
这样即使有属性在 user 中不存在，通过 user.id 来定义也同样可以这样监听这个属性的变化哦。

28、实现一个两列等高布局，讲讲思路

为了实现两列等高，可以给每列加上 padding-bottom:9999px;margin-bottom:-9999px;同时父元素设置 overflow:hidden;

31、JS 怎么控制一次加载一张图片，加载完后再加载下一张

(1) 方法 1
```
<div id="mypic">onloading.....</div>

var obj=new Image();
obj.src="http://www.phpernote.com/uploadfiles/editor/
20110724050220179.jpg";
obj.onload=function(){
    alert('图片的宽度为：'+obj.width+'；图片的高度为：'+obj.height);
    document.getElementById("mypic").innnerHTML="<img
src='"+this.src+"' />";
}
```

(2) 方法 2
```
<div id="mypic">onloading.....</div>

var obj=new Image();
obj.src="http://www.phpernote.com/uploadfiles/editor/
20110724050220179.jpg";
obj.onreadystatechange=function(){
    if(this.readyState=="complete"){
        alert('图片的宽度为：'+obj.width+'；图片的高度
为：'+obj.height);
        document.getElementById("mypic").innnerHTML="<img
src='"+this.src+"' />";
    }
}
```

43、性能优化

1. 减少 HTTP 请求
2. 使用内容发布网络（CDN）
3. 添加本地缓存
4. 压缩资源文件
5. 将 CSS 样式表放在顶部，把 javascript 放在底部（浏览器的运行机制决定）
6. 避免使用 CSS 表达式
7. 减少 DNS 查询
8. 使用外部 javascript 和 CSS
9. 避免重定向
10. 图片 lazyLoad

44、能来讲讲 JS 的语言特性吗

运行在客户端浏览器上；

1. 不用预编译，直接解析执行代码；
2. 是弱类型语言，较为灵活；
3. 与操作系统无关，跨平台的语言；
4. 脚本语言、解释性语言

49、JS 的全排列

1、冒泡排序

概念：将数组中的相邻两个元素进行比较，将比较大的数通过两两比较移动到数组末尾，执行一遍内层循环确定一个最大的数，外层循环是从数组开始遍历到末尾。

特点：每一轮循环后，最大的一个数被交换到末尾，因此，下一轮循环就可以“刨除”最后的数，每一轮循环都比上一轮循环的结束位置靠前一位。

2、选择排序

概念：让数组中的每一个数，依次与**后面的数**进行比较，如果前面的数大于后面的数，就进行位置的交换。

3、插入排序

1. 取出后一个元素（默认从第二个元素开始），在已经排序的元素序列中从后向前扫描；
2. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
3. 重复步骤 2，直到找到已排序的元素小于或者等于新元素的位置；
4. 将新元素插入到该位置后；
5. 重复步骤 1~4。

4、快速排序

（1）在数据集之中，选择一个元素作为"基准"（pivot）。

（2）所有小于"基准"的元素，都移到"基准"的左边；所有大于"基准"的元素，都移到"基准"的右边。

（3）对"基准"左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。

5、希尔排序：自组采用直接插入排序 针对有序序列在插入时采用交换法

动画游戏卡顿甚至崩溃的原因（3-5 个）以及解决办法（3-5 个）

原因可能是：

1.内存溢出问题。

2.资源过大问题。

3.资源加载问题。

4.canvas 绘制频率问题

解决办法：

1.针对内存溢出问题，我们应该在钢管离开可视区域后，销毁钢管，让垃圾收集器回收钢管，因为不断生成的钢管不及时清理容易导致内存溢出游戏崩溃。

2.针对资源过大问题，我们应该选择图片文件大小更小的图片格式，比如使用 webp、png 格式的图片，因为绘制图片需要较大计算量。

3.针对资源加载问题，我们应该在可视区域之前就预加载好资源，如果在可视区域生成钢管的话，用户的体验就认为钢管是卡顿后才生成的，不流畅。

4.针对 canvas 绘制频率问题，我们应该需要知道大部分显示器刷新频率为 60 次/s,因此游戏的每一帧绘制间隔时间需要小于 1000/60=16.7ms，才能让用户觉得不卡顿。（注意因为这是单机游戏，所以回答与网络无关）

62、说一下什么是 virtual dom

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异 把所记录的差异应用到所构建的真正的 DOM 树上，视图就更新了。Virtual DOM 本质上就是在 JS 和 DOM 之间做了一个缓存。

70、说说 C++,Java，JavaScript 这三种语言的区别

1️⃣、从静态类型还是动态类型来看

静态类型，编译的时候就能够知道每个变量的类型，编程的时候也需要给定类型，如 Java 中的整型 int，浮点型 float 等。C、C++、Java 都属于静态类型语言。

动态类型，运行的时候才知道每个变量的类型，编程的时候无需显示指定类型，如 JavaScript 中的 var、PHP 中的$。JavaScript、Ruby、Python 都属于动态类型语言。

对于静态类型，在编译后会大量利用已知类型的优势，如 int 类型，占用 4 个字节，编译后的代码就可以用内存地址加偏移量的方法存取变量，而地址加偏移量的算法汇编很容易实现。

对于动态类型，会当做字符串通通存下来，之后存取就用字符串匹配。

2️⃣、从编译型还是解释型来看

编译型语言，像 C、C++，需要编译器编译成本地可执行程序后才能运行，由开发人员在编写完成后手动实施。用户只使用这些编译好的本地代码，这些本地代码由系统加载器执行，由操作系统的 CPU 直接执行，无需其他额外的虚拟机等。

源代码=》抽象语法树=》中间表示=》本地代码

解释性语言，像 JavaScript、Python，开发语言写好后直接将代码交给用户，用户使用脚本解释器将脚本文件解释执行。对于脚本语言，没有开发人员的编译过程，当然，也不绝对。

源代码=》抽象语法树=》解释器解释执行。

对于 JavaScript，随着 Java 虚拟机 JIT 技术的引入，工作方式也发生了改变。可以将抽象语法树转成中间表示（字节码），再转成本地代码，如 JavaScriptCore，这样可以大大提高执行效率。也可以从抽象语法树直接转成本地代码，如 V8Java 语言，分为两个阶段。首先像 C++语言一样，经过编译器编译。和 C++的不同，C++编译生成本地代码，Java 编译后，生成字节码，字节码与平台无关。第二阶段，由 Java的运行环境也就是 Java 虚拟机运行字节码，使用解释器执行这些代码。一般情况下，Java虚拟机都引入了 JIT 技术，将字节码转换成本地代码来提高执行效率。注意，在上述情况中，编译器的编译过程没有时间要求，所以编译器可以做大量的代码优化措施。

对于 JavaScript 与 Java 它们还有的不同：

对于 Java，Java 语言将源代码编译成字节码，这个同执行阶段是分开的。也就是从源代码到抽象语法树到字节码这段时间的长短是无所谓的。

对于 JavaScript，这些都是在网页和 JavaScript 文件下载后同执行阶段一起在网页的加载和渲染过程中实施的，所以对于它们的处理时间有严格要求。

98、JS 加载过程阻塞，解决方法。

1️⃣、什么是加载过程阻塞：

在页面中我们通常会引用外部文件，而浏览器在解析HTML页面是从上到下依次解析、渲染，如果<head>中引用了一个a.js文件，而这个文件很大或者有问题，需要2秒加载，那么浏览器会停止渲染页面（此时是白屏显示，就是页面啥都没有），2秒后加载完成才会继续渲染，这个就是阻塞。

2️⃣、解决阻塞方法：

1、推迟加载（延迟加载）

如果页面初始的渲染并不依赖于js或者CSS可以用推迟加载，就是最后在加载js和css，把引用外部文件的代码写在最后。比如一些按钮的点击事件，比如轮播图动画的脚本也可以放在最后。

2、**defer**延迟加载

在文档解析完成开始执行，并且在DOMContentLoaded事件之前执行完成，会按照他们在文档出现的顺序去下载解析。效果和把script放在文档最后</body>之前是一样的。

注：defer最好用在引用外部文件中使用，用了defer不要使用document.write()方法;使用defer时最好不要请求样式信息，因为样式表可能尚未加载，浏览器会禁止该脚本等待样式表加载完成，相当于样式表阻塞脚本执行。

3、异步加载

- **async**异步加载：指定 script 标签的 async 属性，就是告诉浏览器不必等到加载完外部文件，可以边渲染边下载，什么时候下载完成什么时候执行。

```
<script type="text/javascript" src="a.js" async></script>
```

- **script dom element**法：这个方法是用js动态创建一个script元素添加在document中。
- defer="defer"：脚本将在页面完成解析时执行

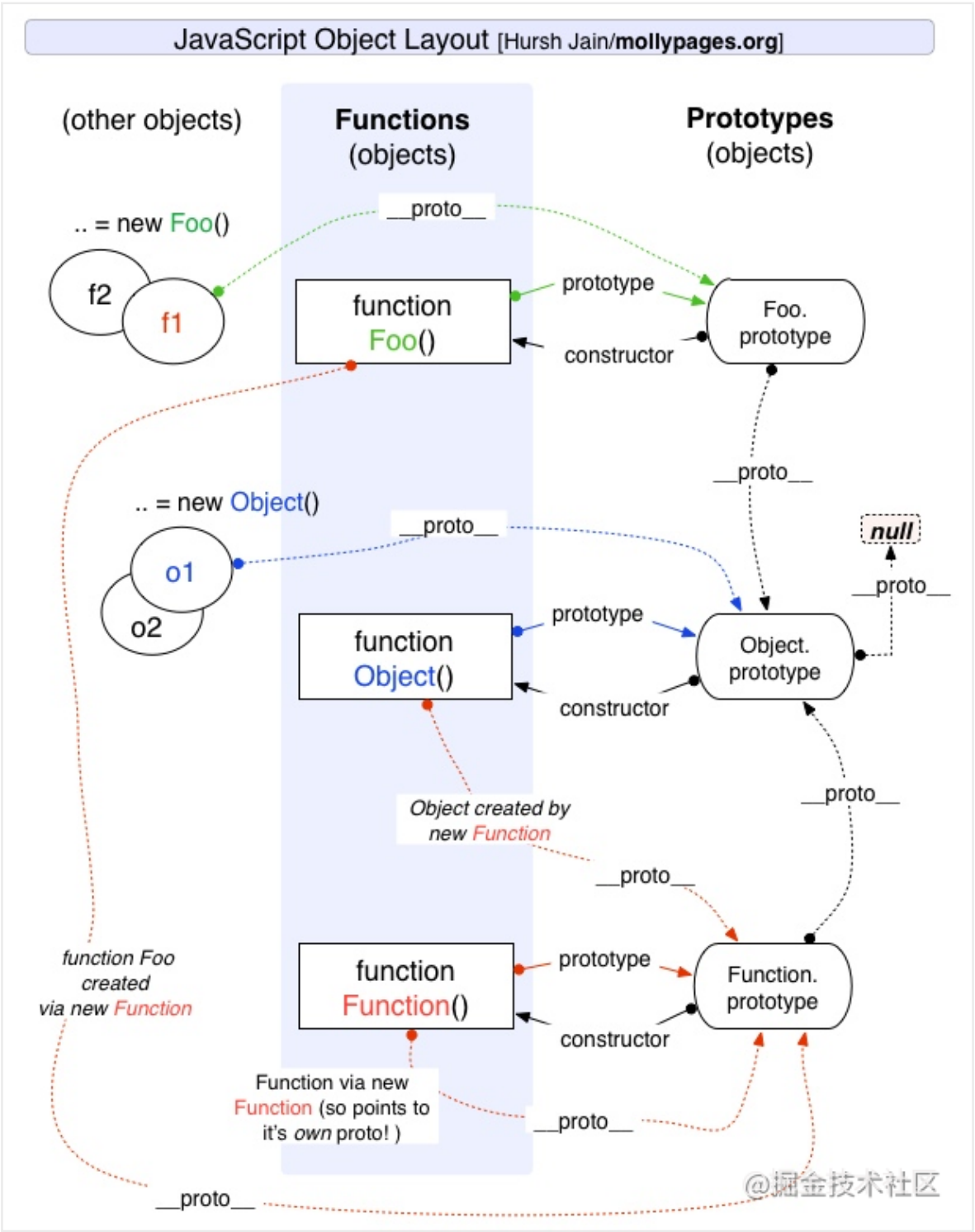
四、原型与原型链

1. 对原型、原型链的理解

在JavaScript中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

特点： JavaScript 对象是通过引用来传递的，创建的每个新对象实体中并没有一份属于自己的原型副本。当修改原型时，与之相关的对象也会继承这一改变。



2. 原型修改、重写

```
function Person(name) {
    this.name = name
}
// 修改原型
Person.prototype.getName = function() {}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype) // true
console.log(p.__proto__ === p.constructor.prototype) // true
```

```
// 重写原型
Person.prototype = {
    getName: function() {}
}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype)       // true
console.log(p.__proto__ === p.constructor.prototype) // false
```

可以看到修改原型的时候p的构造函数不是指向Person了，因为直接给Person的原型对象直接用对象赋值时，它的构造函数指向的了根构造函数Object，所以这时候p.constructor === Object ，而不是p.constructor === Person。要想成立，就要用constructor指回来：

```
Person.prototype = {
    getName: function() {}
}
var p = new Person('hello')
p.constructor = Person
console.log(p.__proto__ === Person.prototype)       // true
console.log(p.__proto__ === p.constructor.prototype) // true
```

3. 原型链指向

```
p.__proto__  // Person.prototype
Person.prototype.__proto__  // Object.prototype
p.__proto__.__proto__ //Object.prototype
p.__proto__.constructor.prototype.__proto__ // Object.prototype
Person.prototype.constructor.prototype.__proto__ // Object.prototype
p1.__proto__.constructor // Person
Person.prototype.constructor  // Person
```

5. 如何获得对象非原型链上的属性?

使用后hasOwnProperty()方法来判断属性是否属于原型链的属性：

```
function iterate(obj){
  var res=[];
  for(var key in obj){
      if(obj.hasOwnProperty(key))
        res.push(key+': '+obj[key]);
  }
  return res;
}
```

71、JS 原型链，原型链的顶端是什么？Object 的原型是什么？Object 的原型的原型是什么?

原型对象：每个对象都有一个原型**prototype**对象，通过函数创建的对象也将拥有这个原型对象。

原型：是一个指向对象的指针，通常指的是**prototype**和**__proto__**这两个原型对象

- 每一个函数身上都有一个属性：**prototype**，称为“原型”；
  - 注意：对象是没有**prototype**属性，只有方法才有**prototype**属性
- 每一个对象（函数）身上都有一个属性： **__proto__** ，称为“隐式原型”；
- 每一个对象的 **__proto__** ，都指向构造该对象的函数的原型对象（**prototype**），

由于Object是构造函数，原型链终点是**Object.prototype. __proto__**，而Object.prototype.__proto__=== null // true，所以，原型链的终点是null。原型链上的所有原型都是对象，所有的对象最终都是由Object构造的，而Object.prototype的下一级是Object.prototype.__proto__。

在数组原型链上实现删除数组重复数据的方法：

```
Array.prototype.remove = function(val) {
    var index = this.indexOf(val);
    if (index > -1) {
        this.splice(index, 1);
    }
};
arr.remove(arr2)
```

五、执行上下文/作用域链/闭包

1. 对闭包的理解

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途；

- 闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。
- 闭包的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

闭包的应用：

1、模仿块级作用域。2、保存外部函数的变量。3、封装私有变量

比如，函数 A 内部有一个函数 B，函数 B 可以访问到函数 A 中的变量，那么

函数 B 就是闭包。

经典面试题：循环中使用闭包解决 var 定义函数的问题

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

首先因为 setTimeout 是个异步函数，所以会先把循环全部执行完毕，这时候 i 就是 6 了，所以会输出一堆 6。解决办法有三种：

- 第一种是使用闭包的方式

```
for (var i = 1; i <= 5; i++) {  ;(function(j) {    setTimeout(function timer() {      console.log(j)    }, j * 1000)  })(i)}
```

在上述代码中，首先使用了立即执行函数将 i 传入函数内部，这个时候值就被固定在了参数 j 上面不会改变，当下次执行 timer 这个闭包的时候，就可以使用外部函数的变量 j，从而达到目的。

- 第二种就是使用 setTimeout 的第三个参数，这个参数会被当成 timer 函数的参数传入。

```
for (var i = 1; i <= 5; i++) {
  setTimeout(
    function timer(j) {
      console.log(j)
    },
    i * 1000,
    i
  )
}
```

- 第三种就是使用 let 定义 i 了来解决问题了，这个也是最为推荐的方式

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

2. 对作用域、作用域链的理解

1）全局作用域和函数作用域

（1）全局作用域

- 最外层函数和最外层函数外面定义的变量拥有全局作用域
- 所有未定义直接赋值的变量自动声明为全局作用域
- 所有 window 对象的属性拥有全局作用域
- 全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，

容易引起命名冲突。

（2）函数作用域

- 函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到
- 作用域是分层的，内层作用域可以访问外层作用域，反之不行

2）块级作用域

- 使用ES6中新增的let和const指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{}包裹的代码片段）
- let和const声明的变量不会有变量提升，也不可以重复声明
- 在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

作用域链： 在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到window对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

3. 对执行上下文的理解

1. 执行上下文类型

（**1**）全局执行上下文

任何不在函数内部的都是全局执行上下文，它首先会创建一个全局的window对象，并且设置this的值等于这个全局对象，一个程序中只有一个全局执行上下文。

（**2**）函数执行上下文

当一个函数被调用时，就会为该函数创建一个新的执行上下文，函数的上下文可以有任意多个。

（**3**）eval函数执行上下文

执行在eval函数中的代码会有属于他自己的执行上下文，不过eval函数不常使用，不做介绍。

eval 函数：它的功能是将对应的字符串解析成 JS 并执行，应该避免使用 JS，因为非常消耗性能（2次，一次解析成 JS，一次执行）

2. 执行上下文栈

- JavaScript引擎使用执行上下文栈来管理执行上下文
- 当JavaScript执行代码时，首先遇到全局代码，会创建一个全局执行上下文并且压入执行栈中，每当遇到一个函数调用，就会为该函数创建一个新的执行上下文并压入栈顶，引擎会执行位于执行上下文栈顶的函数，当函数执行完成之后，执行上下文从栈中弹出，继续执行下一个上下文。当所有的代码都执行完毕之后，从栈中弹出全局执行上下文。

```
let a = 'Hello World!';
function first() {
  console.log('Inside first function');
```

```
  second();
  console.log('Again inside first function');
}
function second() {
  console.log('Inside second function');
}
first();
//执行顺序
//先执行second(),在执行first()
```

3. 创建执行上下文
创建执行上下文有两个阶段：创建阶段和执行阶段
1）创建阶段
（1）this绑定
- 在全局执行上下文中，this指向全局对象（window对象）
- 在函数执行上下文中，this指向取决于函数如何调用。如果它被一个引用对象调用，那么 this 会被设置成那个对象，否则 this 的值被

- 设置为全局对象或者 undefined

（2）创建词法环境组件
- 词法环境是一种有标识符——变量映射的数据结构，标识符是指变量/函数名，变量是对实际对象或原始数据的引用。
- 词法环境的内部有两个组件：加粗样式：环境记录器:用来储存变量个函数声明的实际位置外部环境的引用：可以访问父级作用域

（3）创建变量环境组件
- 变量环境也是一个词法环境，其环境记录器持有变量声明语句在执行上下文中创建的绑定关系。

2）执行阶段 此阶段会完成对变量的分配，最后执行完代码。
简单来说执行上下文就是指：
在执行一点JS代码之前，需要先解析代码。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。这一步执行完了，才开始正式的执行程序。
在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出this、arguments和函数的参数。
- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，this，arguments

六、this/call/apply/bind

1、this 的指向 哪几种

1. 全局中的this：this始终指向全局对象（window / Global）
2. 普通函数中的this：始终指向全局对象；
3. 对象方法中的this：始终是指向调用该方法的对象
4. 事件方法中的this：始终指向绑定该事件的元素节点（指向事件源）
5. 构造函数（class）中的this：始终指向new出来的实例对象

6. 箭头函数中的this：箭头函数中是没有this的，因此当我们在箭头函数中使用this时，实际上使用的是箭头函数父级作用域的this，
7. 回调函数中的*this*：始终指向*window*或*global*

2改变this指向

2.1、call();

语法：call(新的this指向，参数1，参数2，参数3.。。)

- 在改变函数this指向的同时，立即执行函数；
- 在需要传递参数时，call方法将参数一一传递

2.2、apply();

语法：apply(新的this指向，【参数1，参数2，参数3.。。】)

- 在改变函数this指向的同时，立即执行函数
- 在需要传递参数时，apply方法将参数放入数组中一一传递

2.3、bind()();

语法：bind(新的this指向)(参数1，参数2，参数3.。。)

- 在改变函数this指向的同时，会返回一个与原函数具有相同函数体和作用域的新函数，不会立即执行，需要手动重新调用后在执行；
- 当需要传递参数时，bind方法直接从返回的新函数中传递即可
- 注意：`this`将永久地被绑定到了`bind`的第一个参数(也就是我们想要绑定的this指向)，无论这个函数是如何被调用的。也就是说，bind只生效一次。

3. 实现call、apply 及 bind 函数

（1）**call** 函数的实现步骤：

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 处理传入的参数，截取第一个参数后的所有参数。
- 将函数作为上下文对象的一个属性。
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性。
- 返回结果。

```
Function.prototype.myCall = function (context) {
    // 判断调用对象
    if (typeof this !== "function") {
        console.error("type error");
    }
    // 获取参数
    let args = [...arguments].slice(1),
        result = null;
    // 判断 context 是否传入，如果未传入则设置为 window
    context = context || window;
    // 将调用函数设为对象的方法
    context.fn = this;
```

```
    // 调用函数
    result = context.fn(...args);
    // 将属性删除
    delete context.fn;
    return result;
};
```

（2）**apply** 函数的实现步骤：

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 将函数作为上下文对象的一个属性。
- 判断参数值是否传入
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性
- 返回结果

```
Function.prototype.myApply = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== "function") {
        throw new TypeError("Error");
    }
    let result = null;
    // 判断 context 是否存在，如果未传入则为 window
    context = context || window;
    // 将函数设为对象的方法
    context.fn = this;
    // 调用方法
    if (arguments[1]) {
        result = context.fn(...arguments[1]);
    } else {
        result = context.fn();
    }
    // 将属性删除
    delete context.fn;
    return result;
};
```

（3）**bind** 函数的实现步骤：

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 保存当前函数的引用，获取其余传入参数值。
- 创建一个函数返回
- 函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指

```
    定的上下文对象。
Function.prototype.myBind = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== "function") {
        throw new TypeError("Error");
    }
    // 获取参数
    var args = [...arguments].slice(1),
        fn = this;
    return function Fn() {
        // 根据调用方式，传入不同绑定值
        return fn.apply(
            this instanceof Fn ? this : context,
            args.concat(...arguments)
        );
    };
};
```

七、异步编程

1. 异步编程的实现方式?

JavaScript 中的异步机制可以分为以下几种：

- 回调函数 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。
- **Promise** 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。
- **generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。
- **async** 函数 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

2. setTimeout、Promise、Async/Await 的区别

（1）setTimeout

```
console.log('script start')   //1. 打印 script start
setTimeout(function(){
    console.log('settimeout') // 4. 打印 settimeout
})    // 2. 调用 setTimeout 函数，并定义其完成后执行的回调函数
console.log('script end')     //3. 打印 script start
// 输出顺序：script start->script end->settimeout
```

（2）Promise

Promise本身是同步的立即执行函数， 当在executor中执行resolve或者reject的时候, 此时是异步操作， 会先执行then/catch等，当主栈完成后，才会去调用resolve/reject中存放的方法执行，打印p的时候，是打印的返回结果，一个Promise实例。

```
console.log('script start')
let promise1 = new Promise(function (resolve) {
    console.log('promise1')
    resolve()
    console.log('promise1 end')
}).then(function () {
    console.log('promise2')
})
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
// 输出顺序: script start->promise1->promise1 end->script end->promise2->settimeout
```

当JS主线程执行到Promise对象时：

- promise1.then() 的回调就是一个 task
- promise1 是 resolved或rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue
- promise1 是 pending: 这个 task 就会放入 事件循环的未来的某个(可能下一个)回合的 microtask queue 中
- setTimeout 的回调也是个 task ， 它会被放入 macrotask queue 即使是 0ms 的情况

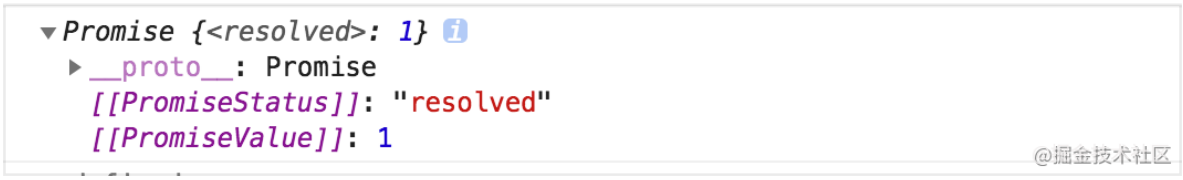
（3）async/await

```
async function async1(){
   console.log('async1 start');
    await async2();
    console.log('async1 end')
}
```

```
async function async2(){
    console.log('async2')
}
console.log('script start');
async1();
console.log('script end')
// 输出顺序：script start->async1 start->async2->script end->async1 end
```

async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。
例如：

```
async function func1() {
    return 1
}
console.log(func1())
```



func1的运行结果其实就是一个Promise对象。因此也可以使用then来处理后续逻辑。

```
func1().then(res => {
    console.log(res); // 30
})
```

await的含义为等待，也就是 async 函数需要等待await后的函数执行完成并且有了返回结果（Promise对象）之后，才能继续执行下面的代码。await通过返回一个Promise对象来实现同步的效果。

3.介绍一下 promise，及其底层如何实现

Promise 是一个对象，保存着未来将要结束的事件，她有两个特征:

1、对象的状态不受外部影响，Promise 对象代表一个异步操作，有三种状态，pending进行中，fulfilled 已成功，rejected 已失败，只有异步操作的结果，才可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也就是 promise 名字的由来

2、一旦状态改变，就不会再变，promise 对象状态改变只有两种可能，从 pending 改到fulfilled 或者从 pending 改到 rejected，只要这两种情况发生，状态就凝固了，不会再改变，这个时候就称为定型
resolved, Promise 的基本用法，

4. Promise的基本用法

Promise有五个常用的方法：then()、catch()、all()、race()、finally。下面就来看一下这些方法。

1. **then()**

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为resolved时调用，第二个回调函数是Promise对象的状态变为rejected时调用。其中第二个参数可以省略。 then方法返回的是一个新的Promise实例（不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

2. catch()

该方法相当于then方法的第二个参数，指向reject的回调函数。不过catch方法还有一个作用，就是在执行resolve回调函数时，如果出现错误，抛出异常，不会停止运行，而是进入catch方法中。

3. all()

all方法可以完成并行任务， 它接收一个数组，数组的每一项都是一个promise对象。当数组中所有的promise的状态都达到resolved的时候，all方法的状态就会变成resolved，如果有一个状态变成了rejected，那么all方法的状态就会变成rejected。

（4）race()

race方法和all一样，接受的参数是一个每项都是promise的数组，但是与all不同的是，当最先执行完的事件执行完之后，就直接返回该promise对象的值。如果第一个promise对象状态变成resolved，那自身的状态变成了resolved；反之第一个promise变成rejected，那自身状态就会变成rejected。

使用场景：当要做一件事，超过多长时间就不做了，可以用这个方法来解决：Promise.race([promise1, timeOutPromise(5000)]).then(res=>{})

5. finally()

finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

```
let promise1 = new Promise(function(resolve, reject){
      if (/* 异步操作成功 */){resolve(value);
      } else { reject(error);}
})
let promise2 = new Promise(function(resolve, reject){})

Promise.all([promise1, promise2]).then(res=>{}, error=>{}).catch((err) =>
```

{}).finally(() => {···});
Promise.race([promise1, promise2]).then(res=>{},err=>{}).catch((err) =>
{}).finally(() => {···});
上面代码中，不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。

5. Promise解决了什么问题

解决了地狱回调的问题。

6. Promise.all和Promise.race的区别的使用场景

（1）**Promise.all** Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被**reject**失败状态的值。
Promise.all中传入的是数组，返回的也是是数组，并且会将进行映射，传入的promise对象返回的值是按照顺序在数组中排列的，但是注意的是他们执行的顺序并不是按照顺序的，除非可迭代对象为空。
需要注意，Promise.all获得的成功结果的数组里面的数据顺序和Promise.all接收到的数组顺序是一致的，这样当遇到发送多个请求并根据请求顺序获取和使用数据的场景，就可以使用Promise.all来解决。
（2）**Promise.race**
顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。当要做一件事，超过多长时间就不做了，可以用这个方法来解决：
Promise.race([promise1,timeOutPromise(5000)]).then(res=>{})

7. 对async/await的理解

async/await其实是Generator的语法糖，它能实现的效果都能用then链来实现，它是为优化then链而开发出来的。从字面上来看，async是“异步”的简写，await则为等待，所以很好理解async用于申明一个function是异步的，而await用于等待一个异步方法执行完成。当然语法上强制规定await只能出现在asnyc函数中，先来看看async函数返回了什么：

```
async function testAsy(){
  return 'hello world';
}
let result = testAsy();
console.log(result)
```

所以，async 函数返回的是一个 Promise 对象。async 函数（包含函数语句、函数表达式、Lambda表达式）会返回一个 Promise 对象，如果在函数中 return 一个直接量，async 会把这个直接量通过 Promise.resolve() 封装成 Promise 对象。

async 函数返回的是一个 Promise 对象，所以在最外层不能用 await 获取其返回值的情况下，当然应该用原来的方式：then() 链来处理这个 Promise 对象，就像这样：

```
async function testAsy(){
  return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
   console.log(v)   // hello world
})
```

那如果 async 函数没有返回值，又该如何？很容易想到，它会返回 Promise.resolve(undefined)。

联想一下 Promise 的特点——无等待，所以在没有 await 的情况下执行 async 函数，它会立即执行，返回一个 Promise 对象，并且，绝不会阻塞后面的语句。这和普通返回 Promise 对象的函数并无二致。

注意：Promise.resolve(x) 可以看作是 new Promise(resolve => resolve(x)) 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 Promise 实例。

8. await 到底在等啥?

await 在等待什么呢？ 一般来说，都认为 await 是在等待一个 async 函数完成。不过按语法说明，await 等待的是一个表达式，这个表达式的计算结果是 Promise 对象或者其它值（换句话说，就是没有特殊限定）。

因为 async 函数返回一个 Promise 对象，所以 await 可以用于等待一个 async 函数的返回值——这也可以说是 await 在等 async 函数，但要清楚，它等的实际是一个返回值。注意到 await 不仅仅用于等 Promise 对象，它可以等任意表达式的结果，所以，await 后面实际是可以接普通函数调用或者直接量的。所以下面这个示例完全可以正确运行：

```
function getSomething() {
```

```
    return "something";
}
async function testAsync() {
    return Promise.resolve("hello async");
}
async function test() {
    const v1 = await getSomething();
    const v2 = await testAsync();
    console.log(v1, v2);
}
test();
```

await 表达式的运算结果取决于它等的是什么。

- 如果它等到的不是一个 Promise 对象，那 await 表达式的运算结果就是它等到的东西。
- 如果它等到的是一个 Promise 对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

来看一个例子：

```
function testAsy(x){
   return new Promise(resolve=>{setTimeout(() => {
       resolve(x);
     }, 3000)
    }
   )
}
async function testAwt(){
  let result =  await testAsy('hello world');
  console.log(result);    // 3秒钟之后出现hello world
  console.log('cuger')   // 3秒钟之后出现cug
}
testAwt();
console.log('cug')  //立即输出cug
```

这就是 await 必须用在 async 函数中的原因。async 函数调用不会造成阻塞，它内部所有的阻塞都被封装在一个 Promise 对象中异步执行。await暂停当前async的执行，所以'cug''最先输出，hello world'和‘cuger’是3秒钟后同时出现的。

9. async/await的优势

单一的 Promise 链并不能发现 async/await 的优势，但是，如果需要处理由多个 Promise 组成的 then 链的时候，优势就能体现出来了（很有意思，Promise 通过 then 链来解决多层回调的问题，现在又用 async/await 来进一步优化它）。

假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步

骤的结果。仍然用 setTimeout 来模拟异步操作：

```
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
function takeLongTime(n) {
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}
function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}
function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}
function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}
```

现在用 Promise 方式来实现这三个步骤的处理：

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}
doIt();
// c:|var|test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
// doIt: 1507.251ms
```

输出结果 result 是 step3() 的参数 700 + 200 = 900。doIt() 顺序执行了三个

步骤，一共用了 300 + 500 + 700 = 1500 毫秒，和 console.time()/console.timeEnd() 计算的结果一致。
如果用 async/await 来实现呢，会是这样：

```
async function doIt() {
  console.time("doIt");
  const time1 = 300;
  const time2 = await step1(time1);
  const time3 = await step2(time2);
  const result = await step3(time3);
  console.log(`result is ${result}`);
  console.timeEnd("doIt");
}
doIt();
```

结果和之前的 Promise 实现是一样的，但是这个代码看起来是不是清晰得多，几乎跟同步代码一样

10. async/await 对比 Promise 的优势

- 代码读起来更加同步，Promise 虽然摆脱了回调地狱，但是 then 的链式调用也会带来额外的阅读负担
- Promise 传递中间值非常麻烦，而async/await几乎是同步的写法，非常优雅
- 错误处理友好，async/await 可以用成熟的 try/catch，Promise 的错误捕获非常冗余
- 调试友好，Promise 的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then代码块中使用调试器的步进 (step-over) 功能，调试器并不会进入后续的.then代码块，因为调试器只能跟踪同步代码的每一步。

11. async/await 如何捕获异常

```
async function fn(){
  try{
    let a = await Promise.reject('error')
  }catch(error){
    console.log(error)
  }
}
```

12. 并发与并行的区别?

- 并发是宏观概念，我分别有任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况就可以称之为并发。
- 并行是微观概念，假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B。同时完成多个任务的情况就可以称之为并行。

13. 什么是回调函数？回调函数有什么缺点？如何解决回调地狱问题?

以下代码就是一个回调函数的例子：

```
ajax(url, () => {
   // 处理逻辑
})
```

回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）。假设多个请求存在依赖性，可能会有如下代码：

```
ajax(url, () => {
   // 处理逻辑
   ajax(url1, () => {
      // 处理逻辑
      ajax(url2, () => {
         // 处理逻辑
      })
   })
})
```

以上代码看起来不利于阅读和维护，当然，也可以把函数分开来写：

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}
function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
  })
}
ajax(url, () => {
  // 处理逻辑
  firstAjax()
})
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。回调地狱的根本问题就是：

1. 嵌套函数存在耦合性，一旦有所改动，就会牵一发而动全身
2. 嵌套函数一多，就很难处理错误

当然，回调函数还存在着别的几个缺点，比如不能使用 try catch 捕获错误，不能直接 return。

14、setTimeout、setInterval 和 requestAnimationFrame 之间的区别

参考：http://www.cnblogs.com/xiaohuochai/p/5777186.html

setTimeout：延时定时器，该定时器在定时器到期后执行一个函数或指定的

一段代码
    - **let timeoutID = setTimeout(Function, delay);**
    - **setTimeout**() 是一个**异步函数**
    - **setTimeout** 函数的返回值 timeoutID 表示定时器的编号，这个值可以传递给`**clearTimeout**`来取消该定时器。
    - 需要设置默认时间，不设置默认为 0

setInterval：是重复调用一个函数或执行一个代码段，在每次调用之间具有固定的时间延迟
    - let intervalID = setInterval(Function, delay);
    - setInterval() 也是一个异步函数。
    - 需要设置间隔时间，如果这个参数值小于 10，则默认使用值为 10（这个参数每个浏览器都有默认的最小值
    - 缺陷：如果定时器代码在代码再次添加到队列之前还没完成执行，结果就会导致定时器代码连续运行好几次。而之间没有间隔，不过幸运的是：javascript 引擎足够聪明，能够避免这个问题。当且仅当没有该定时器的如何代码实例时，才会将定时器代码添加到队列中。这确保了定时器代码加入队列中最小的时间间隔为指定时间。
        - 这种重复定时器的规则有两个问题：1.某些间隔会被跳过 2. 多个定时器的代码执行时间可能会比预期小。

requestAnimationFrame：

- 概念：requestAnimationFrame 是浏览器用于定时循环操作的一个接口，类似于 setTimeout，主要用途是按帧对网页进行重绘，requestAnimationFrame() 方法告诉浏览器您希望执行动画并请求浏览器在下一次重绘之前调用指定的函数来更新动画。该方法使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用。
- 不需要设置时间间隔，采用的是系统时间间隔，不会因为前面的任务，不会影响 RAF，但是如果前面的任务多的话，会响应 setTimeout 和 setInterval 真正运行时的时间间隔。

特点：
（1）requestAnimationFrame 会把每一帧中的所有 DOM 操作集中起来，在一次重绘或回
流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率。
（2）在隐藏或不可见的元素中，requestAnimationFrame 将不会进行重绘或回流，这当然
就意味着更少的 CPU、GPU 和内存使用量
（3）requestAnimationFrame 是由浏览器专门为动画提供的 API，在运行时浏览器会自动
优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停，有效节省了
CPU 开销。

2、补充 get 和 post 请求在缓存方面的区别

get 请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。

post 不同，post 做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此 get 请求适合于请求缓存。

15、Ajax 解决浏览器缓存问题

在 ajax 发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since","0")。
在 ajax 发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control","no-cache")。
在 URL 后面加上一个随机数： "fresh=" + Math.random()。
在 URL 后面加上时间搓："nowtime=" + new Date().getTime()。
如果是使用 jQuery，直接这样就可以了 $.ajaxSetup({cache:false})。这样页面的所有 ajax都会执行这条语句就是不需要保存缓存记录。

23、将原生的 ajax 封装成 promise

```
var myNewAjax=function(url){
    return new Promise(function(resolve,reject){
        var xhr = new XMLHttpRequest();
        xhr.open('get',url);
        xhr.send(data);
        xhr.onreadystatechange=function(){
            if(xhr.status==200&&readyState==4){
                var json=JSON.parse(xhr.responseText);
                resolve(json)
            }else if(xhr.readyState==4&&xhr.status!=200){
                reject('error');
            }
        }
    })
}
```

46、JS 实现跨域

1️⃣：**JSONP**

原理：借助script标签的src属性在引入外部资源的时候不受同源策略限制这个特点；只能解决get请求类型

在jquery的ajax中添加以下属性：

```
$.ajax({
```

```
        dataType:'json',
})
```
在express的后端中，需要将res.send()替换成res.jsonp

2️⃣：**CORS**

```
// CORS 处理跨域
var allowCrossDomain = function (req, res, next) {
  // 设置允许跨域访问的请求源（* 表示接受任意域名的请求）
  res.header("Access-Control-Allow-Origin", "*");
  // 设置允许跨域访问的请求头
  res.header("Access-Control-Allow-Headers", "X-Requested-
With,Origin,Content-Type,Accept,Authorization");
  // 设置允许跨域访问的请求类型
  res.header("Access-Control-Allow-Methods",
"PUT,POST,GET,DELETE,OPTIONS");
  // 同意 cookie 发送到服务器（如果要发送cookie，Access-Control-Allow-
Origin 不能设置为星号）
  res.header('Access-Control-Allow-Credentials', 'true');
  next();
};
app.use(allowCrossDomain);
```

3️⃣：代理服务器 **nginx**（较复杂）或 **vue-cli**

将前端代码保存在代理服务器中，前端发送请求都发送到代理服务器，在由代理服务器去请求真正的目标服务器，服务器与服务器之间是可以实现跨域请求

50、跨域的原理

概念：在一个域中，去向其他的域发送请求。如果两个域之间，协议、IP、端口号任意一个不一样，则形成跨域，

注：浏览器在默认情况下是不支持跨域的，一旦跨域，就会报错

51、同源策略

概念：浏览器端的一个安全策略，要求在浏览器中只能进行同源（域）之间的访问，

八、面向对象

1. 对象创建的方式有哪些?

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在

ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种：

（1）第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

（2）第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

（3）第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

（4）第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

（6）第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

2. JS 中继承实现的几种方式，

1、原型链继承，将父类的实例作为子类的原型，他的特点是实例是子类的实例也是父类的实例，父类新增的原型方法/属性，子类都能够访问，并且原型链继承简单易于实现，缺点是来自原型对象的所有属性被所有实例共享，无法实现多继承，无法向父类构造函数传参。

2、构造继承，使用父类的构造函数来增强子类实例，即复制父类的实例属性给子类，构造继承可以向父类传递参数，可以实现多继承，通过 call 多个父类对象。但是构造继承只能继承父类的实例属性和方法，不能继承原型属性和方法，无法实现函数服用，每个子类都有父类实例函数的副本，影响性能

3、实例继承，为父类实例添加新特性，作为子类实例返回，实例继承的特点是不限制调用方法，不管是 new 子类（）还是子类（）返回的对象具有相同的效果，缺点是实例是父类的实例，不是子类的实例，不支持多继承

4、拷贝继承：特点：支持多继承，缺点：效率较低，内存占用高（因为要拷贝父类的属性）无法获取父类不可枚举的方法（不可枚举方法，不能使用 for in 访问到）

5、组合继承：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

6、寄生组合继承：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

九、垃圾回收与内存泄漏

1. 浏览器的垃圾回收机制 GC（garbage collection）

JavaScript 中内存管理的主要概念是可达性。就是那些以某种方式可访问或可用的值，它们被保证存储在内存中

（1）垃圾回收的概念

垃圾回收：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。
注：如果几个对象引用形成一个环，互相引用，但根访问不到它们，这几个对象也是垃圾，也要被清除。
回收机制：

- Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
- JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。
- 不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

（2）垃圾回收的方式

浏览器通常使用的垃圾回收方法有两种：标记清除，引用计数、增量标记算法。

一、标记-清除算法 **Mark-Sweep GC**

基本的垃圾回收算法称为“标记-清除”，定期执行以下“垃圾回收”步骤
流程阶段：

1. 标记阶段：从根集合出发，将所有活动对象及其子对象打上标记，分为进入环境和离开环境
2. 清除阶段：遍历堆，将非活动对象（即打上离开环境标记）的连接到空闲链表上，垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

优点：实现简单， 容易和其他算法组合
缺点
- 碎片化， 会导致无数小分块散落在堆的各处
- 分配速度不理想，每次分配都需要遍历空闲列表找到足够大的分块
- 与写时复制技术不兼容，因为每次都会在活动对象上打上标记

二、标记-压缩 **Mark-Compact**

流程：和“标记－清除”相似，不过在标记阶段后它将所有活动对象紧密的排在堆的一侧（压缩），消除了内存碎片， 不过压缩是需要花费计算成本的。标记后需要定位各个活动对象的新内存地址，然后再移动对象，总共搜索了3次堆。

优点：有效利用了堆，不会出现内存碎片 也不会像复制算法那样只能利用堆的一部分

缺点：压缩过程的开销，需要多次搜索堆

三、引用计数 **Reference Counting**

流程：引用计数，就是记录每个对象被引用的次数，每次新建对象、赋值引用和删除引用的同时更新计数器，如果计数器值为0则直接回收内存。很明显，引用计数最大的优势是暂停时间短
优点
- 可即刻回收垃圾
- 最大暂停时间短
- 没有必要沿指针查找， 不要和标记-清除算法一样沿着根集合开始查找

缺点
- 计数器的增减处理繁重
- 计数器需要占用很多位
- 实现繁琐复杂， 每个赋值操作都得替换成引用更新操作
- 循环引用无法回收
    - 例如： obj1和obj2通过属性进行相互引用，两个对象的引用次数都是2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，obj1和obj2还将会继续存在，因此它们的引用次数永远不会是0，就会引起循环引用。

```
function fun() {
   let obj1 = {};
   let obj2 = {};
   obj1.a = obj2; // obj1 引用 obj2
   obj2.a = obj1; // obj2 引用 obj1
}
```

这种情况下，就要手动释放变量占用的内存：
obj1.a = null
obj2.a = null

四、 **GC** 复制算法

流程：将堆分为两个大小相同的空间 From 和 To， 利用 From 空间进行

分配，当 From 空间满的时候，GC 将其中的活动对象复制到 To 空间，之后将两个空间互换即完成 GC
优点
- 优秀的吞吐量， 只需要关心活动对象
- 可实现高速分配； 因为分块是连续的，不需要使用空闲链表
- 不会发生碎片化
- 与缓存兼容

缺点
- 堆使用率低
- 与保守式 GC 不兼容
- 递归调用函数， 复制子对象需要递归调用复制函数 消耗栈

五、 保守式 **GC**

六、分代回收
出发点：大部分对象生成后马上就变成垃圾，很少有对象能活的很久
- 新生代 = 生成空间 + 2 * 幸存区 复制算法
- 老年代 标记-清除算法

对象在生成空间创建，当生成空间满之后进行 minor gc，将活动对象复制到第一个幸存区，并增加其“年龄” age，当这个幸存区满之后再将此次生成空间和这个幸存区的活动对象复制到另一个幸存区，如此反复，当活动对象的 age 达到一定次数后将其移动到老年代； 当老年代满的时候就用标记-清除或标记-压缩算法进行 major gc

吞吐量得到改善， 分代垃圾回收花费的时间是 GC 复制算法的四分之一；但是如果部分程序新生成对象存活很久的话分代回收会适得其反

七、增量式 GC

概念：将要回收的过程拆分成一个一个的小任务，每个小任务独立执行，给内存里面的增量开始标记和增量标记，避免页面等待时间太长，页面卡顿

本来 gc 只是默默的在幕后回收资源的，但是如果 gc 任务繁重则会长时间暂停应用程序的执行， 增量式 gc 就是一种逐渐推进垃圾回收来控制 mutator 最大暂停时间的方法
三色标记算法
- 白色： 还未搜索过的对象
- 灰色： 正在搜索的对象
- 黑色： 搜索完成的对象

根查找阶段： 对能直接从根引用的对象打上标记，堆放到标记栈里（白色 涂成 灰色）
标记阶段： 从标记栈中取出对象，将其子对象涂成灰色；这个阶段不是一下子处理所有的灰色对象，而只是处理一定个数，然后暂停 gc
清除阶段： 将没被标记的白色对象连接到空闲链表，并重置已标记的对象标记位
优点： 缩短最大暂停时间
缺点： 降低了吞吐量

八、增量标记法：老年代空间里面一定垃圾被回收，整理磁盘空间，需要用到

标记-整理法

（3）减少垃圾回收
虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

- 对数组进行优化： 在清空一个数组时，最简单的方法就是给其赋值为[ ]，但是与此同时会创建一个新的空对象，可以将数组的长度设置为 0，以此来达到清空数组的目的。
- 对 object 进行优化： 对象尽量复用，对于不再使用的对象，就将其设置为 null，尽快被回收。
- 对函数进行优化： 在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

2. 哪些情况会导致内存泄漏
以下四种情况会造成内存的泄漏：

- 意外的全局变量： 由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- 被遗忘的计时器或回调函数： 设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- 脱离 **DOM** 的引用： 获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。
- 闭包： 不合理的使用闭包，从而导致某些变量一直被留在内存当中。