一.数据类型以及数据结构

1.分类

JavaScript共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是ES6 中新增的数据类型：

- 每个通过Symbol函数创建的Symbol值都是唯一的，即使创建时传入相同的参数，它们也不相等。这意味着Symbol类型可以用来创建一些不会被意外覆盖的属性名或者常量。

  Symbol类型的应用包括：

  - 创建对象的私有属性：可以使用Symbol类型作为属性名，这样可以避免属性被意外修改或者覆盖。
  - 定义常量：可以使用Symbol类型定义一些常量，确保其唯一性。
  - Symbol作为属性名时不会被遍历出来：通过Symbol类型作为对象的属性名，可以确保这些属性不会被for...in、Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()等方法遍历出来，从而保护这些属性的安全性。
  - 作为枚举类型：可以使用Symbol类型定义一组枚举值，例如表示一组操作或状态：

```
const Color = {
  RED: Symbol('RED'),
  GREEN: Symbol('GREEN'),
  BLUE: Symbol('BLUE')
};

// 使用枚举值
```

```
let selectedColor = Color.RED;

if (selectedColor === Color.RED) {
  console.log('Selected color is Red');
} else if (selectedColor === Color.GREEN) {
  console.log('Selected color is Green');
} else if (selectedColor === Color.BLUE) {
  console.log('Selected color is Blue');
}
```

- 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：

原始数据类型（Undefined、Null、Boolean、Number、String）

引用数据类型（对象）

2.数据存储位置：

原始数据类型直接存储在栈（stack）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；

引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

> tips：堆和栈的概念存在于数据结构和操作系统内存中，在数据结构中：

- 在数据结构中，栈中数据的存取方式为先进后出。
- 堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。

在操作系统中，内存被分为栈区和堆区：

- 栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 堆区内存一般由开发者分配释放，若开发者不释放，程序结束时可能由垃圾回收机制回收

3.检测数据类型

- typeof

用于基本数据类型检测，对象、null都会被判断为object

null会判断为object的原因是：历史遗留问题，在计算机中以二进制来保存数据，其中用三位二进制数来表示数据类型，由于null全为0，而000就是用来识别object类型的，所以null会被判断为object

- instanceOf

返回结果是布尔值，`instanceof`一般**用来正确判断引用数据类型**，而不能判断基本数据类型，但是如果我们的基本数据类型是用new xxx这样的形式创建的，那么就可以正确判断了，例如：

```
// 创建一个字符串对象
let str = new String('Hello');

// 使用instanceof运算符来判断对象是否是String构造函数的实例
console.log(str instanceof String); // 输出 true，因为str是String构造函数的实例

// 创建一个数字对象
let num = new Number(123);
```

```
// 使用instanceof运算符来判断对象是否是Number构造函数的实例
console.log(num instanceof Number); // 输出 true，因为num是Number构造函数的实例

// 创建一个布尔对象
let bool = new Boolean(true);

// 使用instanceof运算符来判断对象是否是Boolean构造函数的实例
console.log(bool instanceof Boolean); // 输出 true，因为bool是Boolean构造函数的实例

// 使用instanceof运算符来判断基本数据类型
let strPrimitive = 'Hello';
console.log(strPrimitive instanceof String); // 输出 false，因为strPrimitive是基本数据类型，不是String构造函数的实例
```

这是因为 `instanceof` 是基于原型链的查询，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。

- **constructor属性**

  既能判断基本数据类型又能判断引用数据类型

```
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
console.log(([]).constructor === Array); // true
console.log(([]).constructor === Object); // false
```

`constructor` 有两个作用，一是判断数据的类型，二是对象实例通过 `constrcutor` 属性访问它的构造函数。

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，例如：

```
let arr=[1,2,3];
Object.setPrototypeOf(arr, Object);
console.log(arr.constructor === Array); //false
```

- Object.prototype.toString.call()

  既能判断基本数据类型又能判断引用数据类型

```
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));
/*
[object Number]
[object Boolean]
[object String]
[object Array]
[object Function]
[object Object]
[object Undefined]
[object Null]
*/
```

4.手写instanceOf()

```
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);
  }
}
```

5.其他类型强制转化为String的规则

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，

例子：显示强制类型转换：

```
// 显示将 Symbol 类型转换为字符串
var symbol = Symbol("example symbol");
var symbolString = String(symbol);
console.log(symbolString); // 输出: 'Symbol(example symbol)'
```

隐式强制类型转换（错误示例）：

```
// 隐式将 Symbol 类型转换为字符串（会产生错误）
var symbol = Symbol("example symbol");
var symbolStringImplicit = symbol + ""; // 会产生错误
```

对引用类型来说，除非自行定义 toString() 方法，否则会调用Object.prototype.toString()方法，显示的结果是"[object Object]"。如果对象重写了 toString() 方法，字符串化时就会调用该方法并使用其返回值。

12.其他值强制转换为数字值的转换规则

`Number()` 函数：使用 `Number()` 函数将值显式转换为数字。

- 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 类型的值转换如果是纯数字组成的则转化为对应的数字，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。具体来说，就是先检查应用类型是否可以访问到valueOf()方法。如果有并且返回并且返回的数字就直接调用，如果返回的不是数字是其他基本类型的值那么再遵循以

上规则。如果返回的不是数字也不是其它基本类型的值，就继续调用 toString() 方法。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

*tips：**对于数组**，具体地讲：空数组转换为0，只有一个元素并且这个元素不是NaN则转换为这个数值，其余情况均转换为NaN*，这是因为数组重写了valueOf方法以及toString（）方法

对于最普通的对象，它的valueOf方法也就是Object.prototype.valueOf()返回的是对象本身，这也就是为什么我们在Number(obj)时，先调用Object.prototype.valueOf()返回的是**对象本身**，不是基础类型的值，那么就使用 toString() 方法，返回的是"[Objcet,Object]"，是基础类型（String）,那么按照基本类型强制转化的规则，不是数字，返回的是undefined。

13.其他值强制转换为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
- **""（注意空数组为真）**

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。

18.何时进行隐式类型转换?

- `+` **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。

- `-`、`*`、`\`操作符 两边都被转换为数字，不能转化为数字的返回NaN
- 对于`==`操作符 两边都被转换为数字
- 对于`<`和`>`比较符 如果两边都是字符串不进行类型转换，比较字母表顺序；其它情况，转变为数字比较

19.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声明的变量不允许改变指针的指向。

20.const对象的属性可以修改吗

基本数据类型（原始数据类型）直接存储在栈（stack）中，而引用数据类型的值存储在堆（heap）中，在栈中存储的是堆内存中的地址指针。使用 `const` 声明的变量要求我们不可以改变栈中的指针指向，因此对于基本数据类型，无法修改使用 `const` 声明的变量，这使其表现为常量。对于引用数据类型，栈中保存的仅是在堆中的地址，实际的属性值存储在堆中。因此，在使用 `const` 声明引用数据类型变量时，虽然不可重新赋值一个新的引用类型给该变量（因为这会改变栈中的指针指向），但并不限制修改内部属性值。这意味着可以更改引用数据类型变量指向的对象的属性值，但不能将其指向另一个对象。

总结：`const` 对于基本数据类型表现为常量，不可修改；对于引用数据类型，`const` 限制了重新赋值新对象，但允许修改对象的属性值。

综上，const对象的属性是可以修改的

21.遍历Object的属性的方法有哪些?

- 遍历对象**所有的可枚举属性（自有的+继承的属性）但是不包括Symbol属性**，使用 `for...in`
- 遍历对象**自有的所有可枚举属性（非继承属性）但是不包括Symbol属性**，使用 `Object.keys()` 或 `for...in + Object.hasOwnProperty.call()`（object.values()和object.entries()也是同样的道理）
- 遍历对象**自有的所有可枚举和不可枚举属性（非继承属性）但是不包括Symbol属性**，使用 `Object.getOwnPropertyNames()`
- 获取对象**所有的Symbol属性**，使用Object.getOwnPropertySymbols()
- 获取对象**所有的属性，无论是否枚举，是否继承，是否为Symbol**：Reflect.ownKeys()

  tips：遍历对象的顺序究竟是怎么样的?

1. 首先如果我们的 key 值存在像 `'1'`、`'200'` 这种正整数格式的字符串。先遍历这样的属性，顺序是按照 key 值的大小来排列的。
2. 接下来按照插入顺序遍历剩下的属性（不包括symbo属性）
3. 如果可以遍历Symbol类型的属性，最后再按照插入顺序遍历 `Symbol` 类型

尽管会遵循上面的规则，但是 `for...in` 还会遍历原型的属性。所以 `for...in` 的变量元素的规则是先按照我们上面讲的对象遍历规则去变量对象本身，接下来再按照此规则去遍历对象的原型，以此类推，直到遍历到顶部

22.map和weakMap的区别

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结构原生提供三个遍历器生成函数和两个遍历方法

- 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。

  注意返回的均是遍历器，如果要遍历还要用for of循环来遍历这个遍历器
- myMap.forEach()：遍历Map的所有成员。参数是一个回调函数，回调函数的三个参数分别是Map中的value,key以及map本身体
- for of 循环

（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。

WeakMap 中的类型为对象的键都是弱引用，即垃圾回收机制不考虑 WeakMap 对某个对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakMap 之中。

也正是因为这样的特性，WeakMap 内部有多少个成员，取决于垃圾回收机制有没有运行，运行前后很可能成员个数是不一样的，而垃圾回收机制何时运行是不可预测的，因此 **ES6 规定 WeakMap 不可遍历。**

> 补充：
> **Set和WeakSet的区别**：
>
> 1. 集合（Set）
>
> ES6 新增的一种新的数据结构，类似于数组，但成员是唯一且无序的，没有重复的值。
>
> - Set 实例属性
> - - size：元素数量
> - Set 实例方法

- 操作方法
  - add(value)：新增，相当于 array里的push
  - delete(value)：存在即删除集合中value
  - has(value)：判断集合中是否存在 value
  - clear()：清空集合
- 遍历方法（遍历顺序为插入顺序）
- keys()：返回一个包含集合中所有键的迭代器
  - values()：返回一个包含集合中所有值得迭代器
  - entries()：返回一个包含Set对象中所有元素得键值对迭代器
  - forEach(callbackFn)
  - forOf

WeakSet

WeakSet 对象允许你将**弱引用对象**储存在一个集合中

WeakSet 与 Set 的区别：

- WeakSet 只能储存对象引用，不能存放值，而 Set 对象都可以
- WeakSet 对象中储存的对象值都是被弱引用的，即垃圾回收机制不考虑 WeakSet 对该对象的应用，如果没有其他的变量或属性引用这个对象值，则这个对象将会被垃圾回收掉（不考虑该对象还存在于 WeakSet 中），所以，WeakSet 对象里有多少个成员元素，取决于垃圾回收机制有没有运行，运行前后成员个数可能不一致，遍历结束之后，有的成员可能取不到了（被垃圾回收了），WeakSet 对象是无法被遍历的（ES6 规定 WeakSet 不可遍历），也没有办法拿到它包含的所有元素

方法：

- add(value)：在WeakSet 对象中添加一个元素value
- has(value)：判断 WeakSet 对象中是否包含value
- delete(value)：删除元素 value
- clear()：清空所有元素，**注意该方法已废弃**

23.map和Object的区别

	Map	Object
默认的键	Map默认情况不包含任何键，只包含显式插入的键。	Object 有一个原型对象, 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。	对象有一套自己既定的规则，在此规则下，对象的遍历顺序会受插入元素顺序的影响，但是不完全受插入元素先后顺序的影响。
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
遍历	Map 具有 Iterator 接口，所以可以直接用for of遍历。	迭代Object的方法有多种，并且也可以设置属性是否为可遍历。
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。

24.对JSON的理解

即JavaScript Object Notation的缩写

JSON 是一种按照 JavaScript 对象语法的数据格式，虽然它是基于 JavaScript 语法，但它独立于 JavaScript，这也是为什么许多程序环境能够读取和生成 JSON。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端，通常会将数据对象转换为 JSON 格式，使用 `JSON.stringify()` 方法将 JavaScript 对象转换为 JSON 字符串，然后将它传递到后端，在后端，服务器接收到前端发送的 JSON 数据后，通常会将 JSON 字符串解析为相应的数据结构（如对象），以便后续处理。在大多数后端语言中，都提供了内置的 JSON 解析方法来实现这一步骤。

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：

1. **字符串表示**：
   - 在 JSON 中，属性名必须用双引号括起来，如 `"name": "John"`。
   - 在 JavaScript 对象中，属性名可以不使用引号，或者使用单引号，如 `name: 'John'` 或 `'name': 'John'`。
2. **值类型**：
   - JSON 中的值可以是字符串、数字、布尔值、数组、对象、null。
   - JavaScript 对象中的值还可以是函数、日期对象等，这些类型在 JSON 中无法表示。
3. **特殊值**：
   - JSON 不支持 JavaScript 中的特殊值，如 undefined。
   - JavaScript 对象可以包含 undefined 值。
4. **注释**：
   - JSON 不支持注释，而 JavaScript 对象可以包含注释。

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。

二.原型与原型链

1.对原型和原型链的理解：

在ES5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。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。

2.原型对象的修改、重写

当重写一个构造函数的原型时，新的原型对象将不再具有默认的 `constructor` 属性，默认的 `constructor` 属性指向构造函数，但是重写之后不再指向构造函数。同时，再次调用构造函数创建的实例的constructer属性也不再指向构造函数。

3.原型链的终点是什么？如何打印出原型链的终点？

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`

4.如何获得对象非原型链上的属性?

用for..in遍历对象的属性时使用`hasOwnProperty()`方法来判断属性是否属于原型链的属性

三.箭头函数以及this的指向

1.箭头函数和普通函数的区别

（1）箭头函数比普通函数更加简洁

- 如果只有一个参数，可以省去参数的括号
- 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。箭头函数中没有`this`绑定，必须通过查找作用域链来决定其值。 如果箭头函数被非箭头函数包含，则`this`绑定的是最近一层非箭头函数的`this`，否则`this`的值则被设置为全局对象。因此箭头函数的this在创建的时候已经确定好了。

（3）箭头函数继承来的this指向永远不会改变

即便这个箭头函数是作为某个对象的方法调用，this依旧指向定义时确定的this,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 

（6）箭头函数没有自己的arguments 

箭头函数没有自己的arguments对象。在箭头函数中访问arguments实际上获得的是它外层普通函数的arguments值。

（7）箭头函数没有prototype

（8）箭头函数不能用作Generator函数，不能使用yeild关键字

2.显示绑定----call()、apply()、bind()

apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数，所以我们还要再对这个返回的新函数进行调用才可以实现this的绑定。bind方法也可以传入其余参数，意义和用法与call()方法相同。

3.this的指向：

this指向的是一个对象，this的指向取决于两点：调用的方式和绑定的规则

调用的方式可以分为函数独立调用还是作为一个对象的方法调用，当作为函数独立调用时，this指向全局对象，而作为一个对象的方法的调用时this指向这个对象

绑定规则：

- 默认绑定：

当函数独立调用时，this默认绑定window或者undefined(严格模式下)

- 隐式绑定：
  调用的对象内部有对函数的引用时或者直接作为整个对象的方法时,this指向这个对象：

```
function foo() {
  console.log(this)
}

var obj1 = {
  name: 'obj1',
  foo: foo
}
obj1.foo()

var obj2 = {
  name: 'obj2',
  bar: function () {
    console.log(this)
  }
}
obj2.bar()

var obj3 = {
  name: 'obj3',
  baz: obj2.bar //只是引用了obj2的bar这个函数，而不是通过ob2调用，所以最后打印的是obj3而不是obj2
}
obj3.baz()
```

隐式绑定可能会出现隐式丢失的问题，丢失的是this的指向，主要发生在参数传递和变量赋值当中，本质是因为赋值和传递的都是函数的引用，而最后直接调用这个函数，因此相当于直接调用函数，此时this自然指向的是全局对象. 出现这种现象, 我们应该理解**this的指向是在普通函数调用**

时确定的,也就是函数调用时在作用域中查找而不是在函数定义时查找

```
//1.变量赋值,我们直接把一个对象的方法赋值给一个变量,这个变量成为了一个函数,然后调用这个变量,发现this指向全局或者undefined
var a = 1;
function foo() {
var a = 3;
console.log(this.a)
}
var obj = {
a:2,
foo:foo
};
var bar = obj.foo;
bar() // 1

//2.参数传递,直接把一个对象的方法传递给一个函数作为函数的参数
var a = 1;
function foo() {
var a = 3;
console.log(this.a)
}
function doFun(fn) {
fn()
}
var obj = {
a:2,
foo:foo
};
doFun(obj.foo;) // 1
```

如果不想隐式丢失，应该避免这样的赋值和传参，同时也可以使用显示绑定来绑定目标对象

- 显示绑定：
  不希望在对象内部包含这个函数的引用，但又希望通过对象强制调用，使用call/apply/bind进行显式绑定
- new绑定：通过new关键字来创建构造函数的实例，this绑定到创建的实例上

4.为什么箭头函数不能使用new 来创建一个对象实例?

new操作符的实现步骤如下：

1. **创建一个空对象**：首先，`new`操作符会创建一个空的JavaScript对象。
2. **设置原型链**：接着，这个新创建的空对象的`__proto__`属性会被赋值为构造函数的`prototype`属性。这一步是为了确保新创建的对象可以访问构造函数原型上定义的属性和方法。
3. **执行构造函数**：然后，构造函数会被调用，并且其`this`值会被绑定到新创建的对象上。这意味着在构造函数中，任何对`this`的引用都会指向新创建的对象，允许我们为该对象添加属性和方法。
4. **返回新对象**：如果构造函数返回一个对象，则该对象会成为`new`表达式的结果。如果构造函数没有显式返回一个对象，则会自动返回步骤1中创建的新对象。

所以，上面的第二、三步，箭头函数都是没有办法执行的。

四.闭包

1.对闭包的理解

闭包是闭包是将函数与其周围状态的引用捆绑在一起的组合体。换句话说，闭包允许你从内部函数访问外部函数的作用域。。这时因为，在JS中，变量的作用域属于函数作用域，在函数执行后作用域就会被清理、内存也随之被收回，但是由于闭包时建立在一个函数内部的子函数，由于其可访问上级作用域的原因，即使上级函数执行完，作用域也不会随之销毁，这时的子函数，便拥有了访问上级作用域中的变量的权限，即使上级函数执行完后，作用域内的值也不会被销毁。

```
var a = 10
function foo(){
    console.log(a)
}

function sum() {
    var a = 20
    foo()
}

sum()
/* 输出
    10而不是20
/
```

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

(3）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。

3. 对执行上下文的理解

执行上下文可以分为以下三种类型：

1. **全局执行上下文（Global Execution Context）**：是默认的、最顶层的执行上下文。它在整个页面的生命周期中只会被创建一次，通常用于全局变量的声明和函数的定义。
2. **函数执行上下文（Function Execution Context）**：**当一个函数被调用时，就会创建一个新的函数执行上下文**。每次函数调用都会创建一个新的执行上下文，函数的参数会作为变量被添加到函数执行上下文的变量对象中。
3. **块级作用域执行上下文（Block Scope Execution Context）**：在 ES6 引入了块级作用域（通过 `let` 和 `const` 声明变量），相应的执行上下文会在代码块内部创建。

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。

执行上下文的创建

创建执行上下文有**明确的几个步骤：**

1. 确定 `this`，即我们所熟知的 `this` **绑定**。
2. 创建 **词法环境（LexicalEnvironment）组件**。
3. 创建 **变量环境组件（VariableEnvironment）组件**。

tips:什么是词法环境组件和变量环境组件?

这两个组件统称为**词法环境**，所谓词法环境，就是一个环境记录器和一个 outer 对象，其中环境记录器记录变量，outer 指向父级作用域，而这两个组件的区别是：

变量环境组件（VariableEnvironment component） 用来登记 `var`、`function` 等变量声明

词法环境组件（LexicalEnvironment component） 用来登记 `let`、`const`、`class` 等变量声明

tips：JavaScript属于**解释型语言**，JavaScript的执行分为解释和执行两个阶段，这两个阶段所做的事并不一样：

解释阶段：

- **词法分析**
- **语法分析**
- **作用域规则确定**

执行阶段：

- **创建执行上下文**
- **执行函数代码**

- 垃圾回收

五.异步编程

1.promise

手写promise

```
class myPromise {
    state = "pending";
    value = undefined;
    message = undefined;
    constructor(fn){
        const resoloveHandler= (value) =>{
            this.state = "resoloved";
            this.value =value;
        }
        const rejectHanler = (message) => {
            this.state = "rejected";
            this.message = message;
        }
        try {
            fn(resoloveHandler,rejectHanler);
        }
        catch (error){
            rejectHanler(error)
        }
    }
    then(fn1,fn2) {
        if (this.state === "resoloved"){
            return new myPromise ((resolve,reject)=>{
                const v = fn1(this.value);
```

```
                resolve(v);
            })
        }
        if (this.state === "rejected"){
            return new myPromise ((resolve,reject)=>{
                const m = fn2(this.message);
                resolve(m);
            })

        }
    }
}
//一些静态方法: resolve(),all(),race()
//静态方法
myPromise.resolve = (value) =>{
    return new myPromise((resolve,reject) => {
        resolve(value);
    })
}
myPromise.reject = (message) => {
    return new myPromise((resolve,reject)=>{
        reject(message);
    })
}
MyPromise.all = function (promiseList = []) {
    const p1 = new MyPromise((resolve, reject) => {
        const result = [] // 存储 promiseList 所有的结果
        const length = promiseList.length
        let resolvedCount = 0

        promiseList.forEach(p => {
            p.then(data => {
                result.push(data)

                // resolvedCount 必须在 then 里面做 ++
                // 不能用 index
```

```
                resolvedCount++
                if (resolvedCount === length) {
                    // 已经遍历到了最后一个 promise
                    resolve(result)
                }
            }).catch(err => {
                reject(err)
            })
        })
    })
    return p1
}

MyPromise.race = function (promiseList = []) {
    let resolved = false // 标记
    const p1 = new Promise((resolve, reject) => {
        promiseList.forEach(p => {
            p.then(data => {
                if (!resolved) {
                    resolve(data)
                    resolved = true
                }
            }).catch((err) => {
                reject(err)
            })
        })
    })
    return p1
}
```

其它api

Promise.allSettled

用来确定一组异步操作是否都结束了（不管成功或者失败）。

```
const p1 = ajax("https://xiongmaoyouxuan.com/api/tabs");
const p2 = new Promise((resolve, reject) => {
 setTimeout(() => {
     reject("超时! ");
 }, 2000)
})
Promise.allSettled([p1, p2]).then(res => {
    console.log(res) // 是一个返回的数组，包括状态和返回结果
}).catch(err => {
    console.error(err)
})
```

Promise.any

只要有一个实例变成fulfilled状态，实例就会变成fulfilled状态。如果所有参数实例都便成为rejected状态，包装实例就会变成rejected状态。

```
const p1 = new Promise((resolve, reject) => {
            setTimeout(_ => {
                reject("11111");
            }, 1000)
        })
        const p2 = new Promise((resolve, reject) => {
            setTimeout(_ => {
                reject("22222");
            }, 2000)
        })
```

```
const p3 = new Promise((resolve, reject) => {
    setTimeout(_ => {
        resolve("33333");
    }, 3000)
})
promise.any([p1, p2, p3]).then(res => {
    console.log(res);
}).catch(err => {
    console.error(err);
})
// 3秒后返回'33333'
```

finally()

不管任何状态改变都会触发finally()。

2.generator函数

1）语法

Generator 有两个区分于普通函数的部分：

- 一是在 function 后面，函数名之前有个 * ；
- 函数内部有 yield 表达式。

2）.执行机制

1. **定义 Generator 函数**：Generator 函数是通过在函数关键字前加上星号 `*` 来定义的。Generator 函数内部使用 `yield` 关键字来定义暂停点，每次调用 `next()` 方法会执行到下一个 `yield` 语句。

2. **调用 Generator 函数**：调用 Generator 函数时，不会立即执行函数内的代码，而是返回一个迭代器对象（Iterator）。要开始执行 Generator 函数内的代码，需要通过调用迭代器对象的 `next()` 方法，next方法返回一个包含**value和done属性的对象**，value属性值值是yield关键词后表达式的值，done属性表示 Generator 函数是否已经执行结束。

> **Generator 函数什么时候执行结束?**
> 执行到return语句，如果没有return语句，执行完函数体所有的语句结束。如果在执行结束后仍然使用 `next()` 方法, 返回的对象value属性值为undefined，**done属性值为true**

3. **迭代器对象的 `next()` 方法**：通过调用迭代器对象的 `next()` 方法来执行 Generator 函数内的代码，每次调用都会从上一次暂停的地方继续执行，直到遇到下一个 `yield` 关键字或函数结束。
4. **`return` 方法**：在 Generator 函数中使用 `return` 方法可以结束函数的执行，并指定返回的值。如果调用 `return` 方法时传入参数，则该参数会作为 `next()` 方法返回对象的value属性值；如果不传入参数，则返回的对象的value属性值值为 `undefined`。
5. **`yield\*` 表达式**：`yield*` 表达式用于在 Generator 函数内部调用另一个 Generator 函数，从而实现嵌套生成器调用。

3.使用场景

1. **迭代器（Iterator）**: Generator 函数可以用来创建自定义的迭代器，通过使用 `yield` 关键字来定义每个迭代步骤的返回值。这样可以简化迭代过程的实现，并且使得代码更加可读。例如，可以使用 Generator 函数来迭代一个数组或对象的元素。
2. **异步编程**: 由于 Generator 函数的特性，可以简化异步编程的复杂性。通过使用 `yield` 关键字暂停函数的执行，并通过 `next()` 方法恢复执行，可以实现更简洁、易读的异步代码。

```
function* asyncTaskGenerator() {
  console.log('Starting task 1...');
  yield new Promise(resolve => setTimeout(resolve,
1000));
  console.log('Task 1 complete.');

  console.log('Starting task 2...');
  yield new Promise(resolve => setTimeout(resolve,
1500));
  console.log('Task 2 complete.');

  console.log('All tasks complete!');
}

// 使用 Generator 函数来执行异步任务
const task = asyncTaskGenerator();
function runTask() {
  const { value, done } = task.next();
  if (done) {
    return;
  }
  value.then(() => {
    runTask();
  });
}

runTask();
```

3. **状态机（State Machine）**：Generator 函数的暂停和恢复特性使其非常适合用作状态机的实现。每个状态对应一个 Generator 函数，通过不同的 `yield` 表达式切换状态，从而实现复杂的状态流转和逻辑处理。

```
// 定义状态机的不同状态
```

```
function* stateOne() {
  console.log('In state one, performing some
actions...');
  yield 'stateTwo'; // 切换到 stateTwo
}

function* stateTwo() {
  console.log('In state two, performing some other
actions...');
  yield 'stateThree'; // 切换到 stateThree
}

function* stateThree() {
  console.log('In state three, performing final
actions...');
  yield 'end'; // 结束状态机
}

// 状态机执行函数
function stateMachine() {
  let currentState = stateOne(); // 初始状态为 stateOne

  while (true) {
    const { value, done } = currentState.next();

    if (done) {
      console.log('State machine ended.');
      break;
    }

    if (value === 'stateTwo') {
      currentState = stateTwo();
    } else if (value === 'stateThree') {
      currentState = stateThree();
    } else {
      console.log('Unknown state:', value);
```

```
        break;
      }
    }
  }

  // 执行状态机
  stateMachine();
```

4. **数据流的控制**：Generator 函数可以通过 yield 关键字来控制数据的流动。通过 yield 关键字可以将数据逐个地传递出来，实现了一种暂停和恢复执行的机制。这种控制数据流的方式可以帮助简化复杂的数据处理逻辑，同时也提高了代码的可读性和可维护性。

```
// 定义一个 Generator 函数来处理数据流
function* processData() {
  console.log('Processing data...');

  // 模拟一些数据处理过程
  for (let i = 0; i < 5; i++) {
    // 模拟异步操作，例如从网络请求数据
    yield fetchData(i);
  }

  console.log('Data processing complete.');
}

// 模拟异步获取数据的函数
function fetchData(index) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(`Data ${index}`);
    }, Math.random() * 1000); // 模拟不同的数据获取时间
```

```
  });
}

// 执行数据处理函数
async function processDataFlow() {
  const dataGenerator = processData();

  for await (const data of dataGenerator) {
    console.log('Received data:', data);
  }
}

// 执行数据流控制
processDataFlow();
```

3.async函数

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。

tips： `Promise.resolve()` ： `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。

await等待的是实际是一个返回值，await 不仅仅用于等 Promise 对象，它可以等待任意表达式的结果

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。然后后续的代码才会继续运行。

如果这个 Promise 被 reject 了（即异步操作失败），await 表达式会抛出一个 rejection，就像普通的 Promise 处理一样。我们可以在包裹 await 表达式的 try/catch 块中捕获这个 rejection，以便处理错误情况。

4.event loop

> 补充:
>
> ### 浏览器线程
>
> GUI渲染线程：负责渲染页面，解析html和CSS、构建DOM树、渲染树、和绘制页面，重绘重排也是在该线程执行
>
> JS引擎线程：一个tab中只有一个JS引擎线程(单线程)，负责解析和执行JS。**它和GUI渲染进程不能同时执行，只能一个一个来，如果JS执行过长就会导致阻塞掉帧**
>
> 计时器线程：指setInterval和setTimeout，因为JS引擎是单线程的，所以如果处于阻塞状态，那么计时器就会不准了，所以需要单独的线程来负责计时器工作
>
> 异步http请求线程：XMLHttpRequest连接后浏览器开的一个线程，比如请求有回调函数，异步线程就会将回调函数加入事件队列，等待JS引擎空闲执行
>
> 事件触发线程：主要用来控制事件循环，比如JS执行遇到计时器，AJAX异步请求等，就会将对应任务添加到事件触发线程中，在对应事件符合触发条件触发时，就把事件添加到待处理队列的队尾，等JS引擎处理。

js是一个单线程的语言，为了能够实现异步操作，引入了很多机制，event loop就是其中之一。在执行同步任务的相关代码的时候，我们按照代码从上到下的顺序放入一个叫做调用栈的数据结构中，符合先进后出的方式。如果代码是异步任务，则在当异步任务的**触发条件满足**时，将异步任务的回调函数压入`task Queue`中。先执行调用栈中的同步任务，当调用栈为空时，event loop会开始工作，即不断地，循环往复地轮询task queue，读取回调函数并执行。

当然，js对异步任务又进行了分类，从而出现了不同的task queue，即微任务队列和宏任务队列。

> **宏任务:**
> setTimeout
> setInterval
> AJAX
> DOM事件
>
> **微任务:**
>
> Promise.then
> async 函数的await
>
> 注意:
> **当创建一个 Promise 对象时，传入的 executor 函数会立即执行，所以 executor 函数应该是同步执行的。**

event loop的过程也进一步得到了丰富。当同步任务的调用栈为空之后，先查看微任务队列:

- 若存在微任务，将微任务队列全部执行(包括执行微任务过程中产生的新微任务)
- 若无微任务，查看宏任务队列，执行队列首部的红任务，这个宏任务执行完毕后会再次查看微任务队列，并重复上述操作
- 最终event loop的结束的条件是宏任务队列为空

> 微任务队列会在DOM渲染前执行
>
> 宏任务会在 DOM 渲染后执行

六.垃圾回收机制

1. 浏览器的垃圾回收机制

垃圾回收的概念：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不再参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

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到页面卸载；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。

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。**

- **标记清除：**
  标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。
- **引用计数：**

  当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变为0时，说明这个变量已经没有价值，因此，在垃圾回收机制下次再运行时，这个变量所占有的内存空间就会被释放出来。

  缺点： 这种方法会引起**循环引用**的问题：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。

  (对象的引用计数在创建时为1)

```
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
```

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：** 清空一个数组时，不要直接给其赋值为空数组，因为这样会创建一个新的空对象，最好的做法是令数组的length属性为0
- **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。
- **避免闭包中的循环引用**：

```
// 不推荐的做法，会创建循环引用
function createClosure() {
  let obj = {};
  obj.func = function() {
    console.log(obj);
  };
  return obj;
}

let closure = createClosure();
closure.func();

// 推荐的做法，避免循环引用
function createClosure() {
  let obj = {};
  obj.func = function() {
    console.log(obj);
  };
  obj.clearFunc = function() {
    obj.func = null;
  };
  return obj;
```

```
}

let closure = createClosure();
closure.func(); // 输出obj
closure.clearFunc();
closure.func(); // 什么都不会输出，避免了循环引用
```

- 尽量使用局部变量而不是全局变量可以更快地释放内存。这是因为局部变量的生命周期通常比全局变量短，一旦离开了它的环境（例如：函数执行结束），局部变量就可以被标记为垃圾回收。

2. 哪些情况会导致内存泄漏

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。

七.DOM

DOM的本质是什么?

一种树型数据结构，用来表示网页文档的层次结构，包括了各种元素节点。通过操作 DOM，我们可以实现动态交互效果。

1.attribute和property

property相关api；

```
const div1 = document.getElementById('div1')
console.log('div1', div1)

const divList = document.getElementsByTagName('div') // 集合
console.log('divList.length', divList.length)
console.log('divList[1]', divList[1])

const containerList =
document.getElementsByClassName('container') // 集合
console.log('containerList.length', containerList.length)
console.log('containerList[1]', containerList[1])

const pList = document.querySelectorAll('p')
console.log('pList', pList)

const pList = document.querySelectorAll('p')
const p1 = pList[0]

// property 形式
p1.style.width = '100px'
console.log( p1.style.width )
p1.className = 'red'
console.log( p1.className )
console.log(p1.nodeName)
console.log(p1.nodeType) // 1
```

attribute相关api

```
// attribute
p1.setAttribute('data-name', 'imooc')
console.log( p1.getAttribute('data-name') )
p1.setAttribute('style', 'font-size: 50px;')
console.log( p1.getAttribute('style') )
```

二者的区别：

property可以修改的是dom对象上自带的属性，无法获取或修改自定义属性。而attribute也可以修改dom对象上自带的属性，并且是可以设置自定义属性的。

都有可能会引发DOM的重新渲染

2.DOM结点修改

```
//获取结点
const div1 = document.getElementById('div1')
const div2 = document.getElementById('div2')

// 新建节点
const newP = document.createElement('p')
newP.innerHTML = 'this is newP'
// 插入节点
div1.appendChild(newP)

// 移动节点
const p1 = document.getElementById('p1')
//把原来的p1移动到div2盒子中
div2.appendChild(p1)

// 获取父元素
```

```
console.log( p1.parentNode )

// 获取子元素列表
const div1ChildNodes = div1.childNodes
//返回的结果除了DOM以外还有文本内容
console.log( div1.childNodes )
const div1ChildNodesP =
Array.prototype.slice.call(div1.childNodes).filter(child =>
{
    if (child.nodeType === 1) {
        return true
    }
    return false
})
//对文本内容过滤，child.nodeType === 1则为DOM元素
console.log('div1ChildNodesP', div1ChildNodesP)
//移除元素
div1.removeChild( div1ChildNodesP[0] )
```

3.优化DOM操作的性能

DOM 查询做缓存

```
// 不缓存 DOM 查询结果
for (let = 0; i < document.getElementsByTagName('p').length; i++) {
    // 每次循环，都会计算 length ，频繁进行 DOM 查询
}

// 缓存 DOM 查询结果
const pList = document.getElementsByTagName('p')
const length = pList.length
for (let i = 0; i < length; i++) {
    // 缓存 length ，只进行一次 DOM 查询
}
```

将频繁操作改为一次性操作

```
const listNode = document.getElementById('list')

// 创建一个文档片段，此时还没有插入到 DOM 树中
const frag = document.createDocumentFragment()

// 执行插入
for(let x = 0; x < 10; x++) {
    const li = document.createElement("li")
    li.innerHTML = "List item " + x
    frag.appendChild(li)
}

// 都完成之后，再插入到 DOM 树中
listNode.appendChild(frag)
```

八. BOM操作

BOM 操作 (Browser Object Model)

navigator 和 screen

```
// navigator
const ua = navigator.userAgent
const isChrome = ua.indexOf('Chrome')
console.log(isChrome)

// screen
console.log(screen.width)
console.log(screen.height)
```

location 和 history

```
// location
console.log(location.href)
console.log(location.protocol) // 'http:' 'https:'
console.log(location.pathname) // '/learn/199'
console.log(location.search)
console.log(location.hash)

// history
history.back()
history.forward()
```

九.事件

1.事件的绑定和冒泡

事件冒泡（Event Bubbling）是指事件在DOM树中向上传播的过程。当一个事件触发在某个元素上时，该事件会首先在触发的元素上被处理，然后逐级向上传播，直到最顶层的祖先元素。如果父元素有对应事件的监听函数的话，就会触发监听函数。

在事件处理程序中，`event.target` 和 `event.currentTarget` 是两个常用的属性，它们表示事件当前发生的元素。

- `event.target` 表示触发事件的实际目标元素。它是指引起事件的元素，即事件最初发生在哪个元素上。
- `event.currentTarget` 表示当前绑定事件的元素。它是指当前正在执行事件处理程序的元素，不一定是事件最初发生的元素。

例子：

```
<div id="outer">
  <div id="inner">Click me!</div>
</div>
```

```
<script>
  const outer = document.querySelector('#outer');
  const inner = document.querySelector('#inner');

  outer.addEventListener('click', function(event) {
    console.log('target:', event.target.id);
    console.log('currentTarget:', event.currentTarget.id);
  });

  inner.addEventListener('click', function(event) {
    console.log('target:', event.target.id);
    console.log('currentTarget:', event.currentTarget.id);
  });
</script>
```

```
控制台:
target: inner
currentTarget: outer

target: inner
currentTarget: inner
```

2.哪些事件支持冒泡

可以通过 **event.bubbles** 属性可以判断该事件是否可以冒泡

事件	是否冒泡
click	可以
dbclick	可以
keydown	可以
keyup	可以
mousedown	可以
mousemove	可以
mouseout	可以
mouseover	可以
mouseup	可以
scroll	可以

概括来说，鼠标事件，和键盘事件，以及点击事件是支持冒泡的

`mousemove` 事件是一种鼠标事件，它在用户移动鼠标指针时触发。具体来说，当用户将鼠标光标从一个元素移到另一个元素或在页面上的某个位置移动鼠标时，浏览器会生成 `mousemove` 事件。这个事件通常用于跟踪鼠标的位置和响应用户的鼠标操作。

两组对比：

（1）

`mouseover`：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，冒泡过程。对应的移除事件是mouseout

`mouseenter`：当鼠标移除元素本身（不包含元素的子元素）会触发事件，也就是不会冒泡，对应的移除事件是mouseleave

（2）

`mouseout`：只要鼠标指针移出事件所绑定的元素或其子元素，都会触发该事件

`mouseleave`：只有鼠标指针移出事件所绑定的元素时，才会触发该事件

3.阻止冒泡的方式有哪些

阻止冒泡行为：非IE 浏览器 stopPropagation()，IE 浏览器 window. event. cancelBubble = true

阻止默认行为：非 IE 浏览器 preventDefault()，IE 浏览器 window. event. returnValue = false

4.事件代理

利用事件冒泡的特性，在父元素上统一处理子元素的事件。

事件代理的优点包括：

1. 减少事件处理程序的数量：不需要为每个子元素都绑定事件处理程序。
2. 提高性能：减少了大量的事件处理程序，节约了内存和性能开销。
3. 简化代码：通过统一处理父元素上的事件，使代码更加简洁和易于维护。

5.手写一个通用事件绑定函数（包括事件绑定和事件代理）

```
// 通用的事件绑定函数
function bindEvent(elem, type, selector, fn) {
    //参数判断
    if (fn == null) {
        fn = selector
        selector = null
    }
    elem.addEventListener(type, function(event) {
        const target = event.target
```

```
        if (selector) {
            // 代理绑定
            if (target.matches(selector)) {
                fn.call(target, event)
            }
////matches 方法是用于检查一个元素是否与指定的选择器匹配的方法。它是 Element 对象的原生方法，通常用于在事件委托（event delegation）等场景中方便地判断事件的目标元素是否符合特定的选择器。
///matches 方法接受一个参数，即 CSS 选择器字符串，用于指定要检查的选择器。
        } else {
            // 普通绑定
            fn.call(target, event)
        }
    })
}

// 普通绑定
const btn1 = document.getElementById('btn1')
bindEvent(btn1, 'click', function (event) {
    // console.log(event.target) // 获取触发的元素
    event.preventDefault() // 阻止默认行为
    alert(this.innerHTML)
})

// 代理绑定
const div3 = document.getElementById('div3')
bindEvent(div3, 'click', 'a', function (event) {
    event.preventDefault()
    alert(this.innerHTML)
})
```

无限下拉图片列表，如何监听每个图片的点击

- 事件代理
- 用 e.target 获取触发元素
- 用 matches 来判断是否是触发元素

十.var let const

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声明的变量不允许改变指针的指向。

2.const对象的属性可以修改吗

基本数据类型（原始数据类型）直接存储在栈（stack）中，而引用数据类型的值存储在堆（heap）中，在栈中存储的是堆内存中的地址指针。使用 `const` 声明的变量要求我们不可以改变栈中的指针指向，因此对于基本数据类型，无法修改使用 `const` 声明的变量，这使其表现为常量。对于引用数据类型，栈中保存的仅是在堆中的地址，实际的属性值存储在堆中。因此，在使用 `const` 声明引用数据类型变量时，虽然不可重新赋值一个新的引用类型给该变量（因为这会改变栈中的指针指向），但并不限制修改内部属性值。这意味着可以更改引用数据类型变量指向的对象的属性值，但不能将其指向另一个对象。

总结： `const` 对于基本数据类型表现为常量，不可修改；对于引用数据类型， `const` 限制了重新赋值新对象，但允许修改对象的属性值。

综上，const对象的属性是可以修改的

十一.解构赋值

解构赋值

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）数组的解构赋值

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

（2）对象的解构赋值

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

```
const stu = {
  name: 'Bob',
  age: 24
}
const { name, age } = stu
console.log(name)//'Bob'
comsole.log(age)//24
```

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：

十二.rest参数

对 rest 参数的理解

rest参数（剩余参数）是一种在函数参数中使用扩展运算符（`...`）的语法，它允许我们将一个不定数量的参数表示为一个数组。这个特性在处理函数参数个数不确定的情况时非常有用，使得函数的编写更加灵活和强大。

重要的是，**Rest参数必须是函数参数列表中的最后一个参数，因为它会收集所有剩余的输入参数。**

使用场景和例子

1. 收集多余的参数

当你想在函数中处理那些超出明确声明参数数量的额外参数时，Rest参数非常有用。

```
function printNames(first, second, ...others) {
  console.log(`First: ${first}, Second: ${second}`);
  console.log(`Others: ${others.join(', ')}`);
}

printNames('Alice', 'Bob', 'Charlie', 'Dave', 'Eve');
// 输出:
// First: Alice, Second: Bob
// Others: Charlie, Dave, Eve
```

在这个例子中，`first`和`second`参数接收前两个参数值，而所有剩余的参数值都被收集到`others`数组中。

2. 函数参数个数不确定

Rest参数允许函数接受任意数目的参数，这对于创建某些类型的工具函数特别有用。

```
function sum(...numbers) {
  return numbers.reduce((acc, current) => acc + current,
0);
}

console.log(sum(1, 2, 3, 4, 5)); // 输出: 15
```

在这个例子中，`sum`函数可以接受任意数量的参数，并返回它们的总和。

优点

- **灵活性**：Rest参数提供了一种非常灵活的方式来处理函数参数，特别是当你不确定将会接收多少参数时。
- **可读性**：使用Rest参数可以使函数签名更简洁明了，增强代码的可读性。
- **替代`arguments`对象**：在ES6之前，JavaScript提供了一个`arguments`对象来访问函数的参数。然而，`arguments`是一个类数组对象，不是一个真正的数组，这限制了它的使用。Rest参数提供了一种更现代、更易用的方式来处理函数参数，且它是一个真正的数组实例，可以直接使用数组方法。

十三.数组的方法

数组的方法

（1）改变原数组的方法

1.push（） 末尾添加数据

2. pop（） 末尾删除数据

3.unshift（） 头部添加数据

4.shift（） 头部删除数据

5.reverse（） 翻转数组

语法: 数组名.reverse()

作用: 就是用来翻转数组的

返回值: 就是翻转好的数组

```
var arr = [10, 20, 30, 40]
res=arr.reverse()
console.log(arr);//[40,30,20,10]
console.log(res);//[40,30,20,10]
```

6.sort（） 排序

语法一: 数组名.sort()　　默认排序是将元素转换为字符串，然后按照它们的 UTF-16 码元值升序排序

语法二: 数组名.sort(function (a,b) {return a-b}) 会正序排列

语法三: 数组名.sort(function (a,b) {return b-a}) 会倒序排列

```
var arr = [2, 63, 48, 5, 4, 75, 69, 11, 23]
arr.sort()
console.log(arr);
arr.sort(function(a,b){return(a-b)})
console.log(arr);
arr.sort(function(a,b){return(b-a)})
console.log(arr);
```

7.splice（） 截取数组

语法一: 数组名.splice(开始索引,多少个)

作用: 就是用来截取数组的

返回值: 是一个新数组 里面就是你截取出来的数据

语法二: 数组名.splice(开始索引,多少个,你要插入的数据)

作用: 删除并插入数据

注意: 从你的开始索引起

返回值: 是一个新数组 里面就是你截取出来的数据

```
var arr = [2, 63, 48, 5, 4, 75]
res = arr.splice(1,2)
console.log(arr);
console.log(res);
//*****************************
//splice() 语法二
var arr = [2, 63, 48, 5, 4, 75]
res = arr.splice(1,1,99999,88888)
console.log(arr);
console.log(res);
```

8.copyWithin()方法

```
copyWithin(target, start, end)
```

`copyWithin()` 方法浅复制数组的一部分到同一数组中的另一个位置，并返回它，不会改变原数组的长度。

(2)不改变原数组的方法

1.concat（）合并数组

语法: 数组名.concat(数据)

作用: 合并数组的

返回值: 一个新的数组

```
//concat复制代码var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.concat(20,"小敏",50)
console.log(arr)
console.log(res);
```

2.join（） 数组转字符串

语法: 数组名.join(' 连接符')

作用: 就是把一个数组转成字符串

返回值: 就是转好的一个字符串

```
var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.join("+")
console.log(arr)
console.log(res);
```

3.slice（）截取数组的一部分数据

语法: 数组名.slice(开始索引, 结束索引)

作用: 就是截取数组中的一部分数据

返回值: 就是截取出来的数据 放到一个新的数组中

注意: 包前不好后 包含开始索引不包含结束索引

```
var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.slice(1,4)
console.log(arr)
console.log(res);
```

4.indexOf 从左检查数组中有没有这个数值

语法一: 数组名.indexOf(要查询的数据)

作用: 就是检查这个数组中有没有该数据

如果有就返回该数据**第一次**出现的索引

如果没有返回 -1

语法二: 数组名.indexOf(要查询的数据, 开始索引)

```
var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.indexOf(10)
console.log(arr)
console.log(res);
//************************************
语法二
var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.indexOf(10,1)
console.log(arr)
console.log(res);
```

5.lastIndexOf 从右检查数组中有没有这个数值

语法一: 数组名.indexOf(要查询的数据)

作用: 就是检查这个数组中有没有该数据

如果有就返回该数据**第一次**出现的索引

如果没有返回 -1

语法二: 数组名.lastIndexOf(要查询的数据, 开始索引)

```
//lastIndexOf复制代码var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.lastIndexOf(50)
console.log(arr)
console.log(res);
//************************************
//lastIndexOf 语法二
var arr = [10, 20, 10, 30, 40, 50, 60]
res = arr.lastIndexOf(50,4)
console.log(arr)
console.log(res);
```

6.forEach()

语法: 数组名.forEach(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

作用: 就是用来循环遍历数组的 代替了我们的for

7.map()

语法: 数组名.map(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

8.fillter()

语法: 数组名.filter(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

  要自己手动返回一个布尔值

9.every 判断数组是不是满足所有条件

语法: 数组名.every(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

  要自己手动返回一个布尔值

10.some（） 数组中有没有满足条件的

语法: 数组名.some(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组
  要自己手动返回一个布尔值
- 返回值是布尔值

11.find（）用来获取数组中满足条件的第一个数据

语法: 数组名.find(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

作用: 用来获取数组中满足条件的数据

返回值: 如果有 就是满足条件的第一个数据；如果没有就是undefined

注意: 要以return的形式执行返回条件

12.findIndex() 用来获取数组中满足条件的第一个数据的下标

语法: 数组名.findIndex(function (item,index,arr) {})

- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

作用: 用来获取数组中满足条件的数据

返回值: 如果有 就是满足条件的第一个数据；如果没有就是undefined

注意: 要以return的形式执行返回条件

13.reduce（）叠加后的效果

语法: 数组名.reduce(function (prev,item,index,arr) {},初始值)

- prev :一开始就是初始值 当第一次有了结果以后；这个值就是第一次的结果
- item : 这个表示的是数组中的每一项
- index : 这个表示的是每一项对应的索引
- arr : 这个表示的是原数组

作用: 就是用来叠加的

返回值: 就是叠加后的结果

注意: 以return的形式书写返回条件

```
var arr = [1, 2, 3, 4, 5]
var res = arr.reduce(function (prev, item) {
    return prev *= item
}, 1)
console.log(res);//120
```

十四.手写系列

1.object.assign和扩展运算符是深拷贝还是浅拷贝?

什么是深浅拷贝?

在 JavaScript 中，深拷贝（Deep Copy）和浅拷贝（Shallow Copy）是用来描述复制对象时的两种不同方式。

浅拷贝：浅拷贝只复制对象的第一层属性，而不复制嵌套对象的属性。这意味着如果原始对象的属性值是引用类型，浅拷贝后的对象仍然会共享这些引用类型的数据。

示例代码：

```
var obj1 = { a: 1, b: { c: 2 } };
var obj2 = Object.assign({}, obj1);

obj2.a = 3;
obj2.b.c = 4;

console.log(obj1); // { a: 1, b: { c: 4 } }
```

深拷贝：深拷贝是指创建一个新的对象，并且递归地复制原始对象所有层级的属性，包括嵌套的对象和数组。这样，新对象与原始对象完全独立，它们不共享任何引用类型的数据。

示例代码：

```
function deepCopy(obj) {
  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }
  var newObj = Array.isArray(obj) ? [] : {};
  for (var key in obj) {
    if (obj.hasOwnProperty(key)){
        newObj[key] = deepCopy(obj[key]);
    }
  }
  return newObj;
```

```
}

var obj1 = { a: 1, b: { c: 2 } };
var obj2 = deepCopy(obj1);

obj2.a = 3;
obj2.b.c = 4;

console.log(obj1); // { a: 1, b: { c: 2 } }
```

总结：

- 浅拷贝只复制对象的第一层属性，而深拷贝会递归复制所有层级的属性。
- 深拷贝生成的对象与原始对象完全独立，互不影响，而浅拷贝生成的对象可能会共享某些引用类型的数据。

回答原来的问题：

都是浅拷贝，因此只复制对象的顶层属性，而不会递归复制嵌套的对象或数组。如果原对象包含引用类型的属性值（例如对象或数组），则复制后的对象将共享这些引用类型属性，而不是创建它们的独立副本。

tips：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。最终返回目标对象

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。

> 补充：

在 JavaScript 中，实现对象的深拷贝（deep copy）通常可以通过以下几种方法：

1. **手动实现深拷贝函数**：手动编写函数来实现深拷贝，可以根据对象的类型进行相应的处理，对于特殊对象类型进行特殊处理。
2. **JSON 序列化和反序列化**：利用 `JSON.stringify` 方法将对象序列化为 JSON 字符串，再利用 `JSON.parse` 方法将 JSON 字符串反序列化为新的对象。这种方法能够实现简单的深拷贝，但对于包含函数、正则表达式等特殊对象的深拷贝不适用。
3. **使用第三方库**：许多第三方库（如 lodash、underscore 等）提供了现成的深拷贝方法，可以简化深拷贝的实现过程。

在 JavaScript 中，实现对象的浅拷贝（shallow copy）通常可以通过以下几种方法：

1. **扩展运算符（Spread Operator）**：使用扩展运算符 `...` 来创建一个新对象，并将原始对象的属性浅拷贝到新对象中。
2. **Object.assign() 方法**：使用 `Object.assign()` 方法将一个或多个源对象的所有可枚举属性复制到目标对象，并返回目标对象。这也是一种浅拷贝的方式。
3. **Array.prototype.concat() 或 Array.prototype.slice()**：对于数组对象，可以使用 `concat()` 方法或 `slice()` 方法进行浅拷贝。这两个方法都会返回一个新的数组，其中包含原始数组的浅拷贝。

2.浅比较 深比较 浅拷贝 深拷贝

```
const shallowEqual = (obj1, obj2) => {
  if (Object.is(obj1, obj2)) {
    return true;///不直接用等号相比，是为了防止NaN，它不满足自反性，但是Object.is修复了这个问题
  }
```

```
  if (
    typeof obj1 !== "object" ||
    typeof obj2 !== "object" ||
    obj1 === null ||
    obj2 === null
  ) {
    return false;
  }
  const [keys1,keys2] =
[Object.keys(obj1),Object.keys(obj2)];
  if (keys1.length!==keys2.length) return false;
  const len = keys1.length;
  for (let key of keys1){
    if (!obj1.hasOwnProperty(key)){
      return false;
    }
    if (obj1[key]!==obj2[key]){
      return false;//直接比较即可
    }
  }
  return true;
};
const deepEqual = (obj1,obj2) => {
  if (Object.is(obj1,obj2)){
    return true;
  }
  if (
    typeof obj1 !== "object" ||
    typeof obj2 !== "object" ||
    obj1 === null ||
    obj2 === null
  ) {
    return false;
  }
  const [keys1,keys2] =
[Object.keys(obj1),Object.keys(obj2)];
```

```
  if (keys1.length!==keys2.length) return false;
  for (let key of keys1){
    if (!obj1.hasOwnProperty(key)){
      return false;
    }
    else {
      const ans = deepEqual(obj1[key],obj2[key]);//递归比较
      if (!ans){
        return false;
      }
    }
  }
  return true;
};
const shallowCopy = (obj) => {
  if (typeof obj !== 'object'){
    return obj;
  }
  if (Array.isArray(obj)){
    return [...obj];
  }
  const newObj = {};
  const keys = Object.keys(obj);
  for (let key of keys){
    newObj[key] = obj[key];//直接赋值，也就是拷贝即可
  }
  return newObj;
}
const deepCopy = (obj) =>{
  if (typeof obj !== 'object'){
    return obj;
  }
  let newObj = {};
  if (Array.isArray(obj)){
    return obj.map((item)=>deepCopy(item));//注意数组的这种写
法
```

```
    }
  const keys = Object.keys(obj);
  for (let key of keys){
    newObj[key] = deepCopy(obj[key]); //递归深拷贝
  }
  return newObj;
}
```

注意深浅拷贝，newObj = deepCopu(obj)， 如果obj = [1,2,[3,4]], 想要看是不是深拷贝，要用 `newObj[2][3]`

这样去修改值，而不是仅使用newObj[2] 去修改值，这样相当于给你的新的对象重新分了一个内存空间，而不是改变原来共享的内存空间

3.防抖和节流

防抖

在前端开发中，防抖（Debouncing）是一种常用的技术，用于**限制在一定时间内频繁触发的事件的执行次数**。通常情况下，**防抖会延迟事件的触发，只有在一定时间内没有再次触发事件时，才会执行相应的操作。**

举个例子，当用户在输入框中连续输入文字时，如果每次输入都立即触发搜索请求，会造成不必要的资源浪费。使用防抖技术可以在用户停止输入一定时间后再触发搜索请求，从而减少请求次数。

```
function debounce(func, delay) {
  let timerId;

  return function() {
    const context = this;
    const args = arguments;

    clearTimeout(timerId);
```

```
    timerId = setTimeout(function() {
      func.apply(context, args);
    }, delay);
  };
}


//怎么使用这个防抖
// 原始事件处理函数
function handleInput() {
  console.log("Input event handled");
}

// 使用防抖包装事件处理函数
const debouncedHandleInput = debounce(handleInput, 300); //
延迟300毫秒

// 绑定事件监听器
inputElement.addEventListener("input",
debouncedHandleInput);
```

清除计时器：

- 防抖函数的目的是延迟执行某个函数，以避免频繁调用该函数。在这个过程中，如果前一个延迟尚未执行完成，但又有新的调用触发了，那么就需要取消前一个延迟，重新设定一个新的延迟时间。这就是为什么需要在每次函数调用时都清除之前设定的计时器，以确保只有最后一次调用生效。如果不清除计时器，就无法保证只有最后一次延迟执行生效，可能会导致函数被多次调用。

绑定this：

> 如果回调函数是一个箭头函数，apply的this是不会起作用的。
> 希望起作用的场景一般是回调函数内部使用了this.xxx，且是一个function()写法，那么apply绑定的this会起效。要注意的是，在function()中，严格模式下没有调用者时，this的值为undefined，因此可以说此时如果希望使用this必须要有apply。
> 节流与防抖的返回值：返回值必须是function命名的函数而不是箭头函数，否则这个函数的this就等于debounce的this等于undefined。

节流

节流（Throttling）是一种控制函数调用频率的技术，它可以限制函数在一定时间间隔内被调用的次数。这种技术通常用于处理频繁触发的事件，例如浏览器的滚动事件或者输入框的输入事件，以减少不必要的函数调用，提高性能和资源利用率。

手写一个节流函数的基本思路是，在函数被触发时记录当前时间戳，然后在函数执行时判断距离上次执行的时间是否已经超过指定的时间间隔，如果超过了，则执行函数并更新时间戳，否则不执行函数。以下是一个简单的 JavaScript 实现：

```
function throttle(func, delay) {
  let lastExecutedTime = 0;

  return function(...args) {
    const currentTime = Date.now();
    if (currentTime - lastExecutedTime > delay) {
      func.apply(this, args);
      lastExecutedTime = currentTime;
    }
  };
}
```

请实现一个节流函数throttle，将传入函数A进行节流处理，返回节流化后的函数B,节流后的函数B在指定时间内多次调用只会执行一次传入函数A
请考虑以下场景：
1.首次立即调用的场景
2.保证最后一次调用的场景

```
function throttle(func, delay) {
  let timerId; // 定时器ID
  let lastExecTime = 0; // 上次执行时间

  return function() {
    const context = this; // 保存函数执行上下文
    const args = arguments; // 保存传入的参数
    const currentTime = new Date().getTime(); // 获取当前时间

    if (currentTime - lastExecTime ⩾ delay) { // 如果距离上次执行超过指定延迟时间
      func.apply(context, args); // 执行传入的函数
      lastExecTime = currentTime; // 更新上次执行时间
    } else {
```

```
        clearTimeout(timerId); // 清除之前的定时器
        timerId = setTimeout(function() {
          func.apply(context, args); // 延迟执行传入的函数
          lastExecTime = currentTime; // 更新上次执行时间
        }, delay - (currentTime - lastExecTime)); // 计算延迟时
间
      }
    };
}
```