题目

1. 防抖节流

这也是一个经典题目了，首先要知道什么是防抖，什么是节流。

- 防抖： 在一段时间内，事件只会最后触发一次。
- 节流： 事件，按照一段时间的间隔来进行触发。
- 实在不懂的话，可以去这个大佬的Demo地址玩玩防抖节流DEMO

```
 // 防抖
function debounce(fn) {
  let timeout = null;
  return function () {
    // 如果事件再次触发就清除定时器，重新计时
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      fn.apply(this, arguments);
    }, 500);
  };
}

// 节流
function throttle(fn) {
  let flag = null; // 通过闭包保存一个标记
  return function () {
    if (flag) return; // 当定时器没有执行的时候标记永远是null
    flag = setTimeout(() => {
      fn.apply(this, arguments);
       // 最后在setTimeout执行完毕后再把标记设置为null(关键)
       // 表示可以执行下一次循环了。
      flag = null;
    }, 500);
  };
}

```

这道题主要还是考查对 防抖 节流 的理解吧，千万别记反了！

2. **2.一个正则题**

要求写出 区号+8位数字，或者区号+特殊号码: **10010/110**，中间用短横线隔开的正则验证。 区号就是三位数字开头。

例如 **010-12345678**

```
let reg = /^\d{3}-(\d{8}|10010|110)/g
```

这个比较简单，熟悉正则的基本用法就可以做出来了。

3. **不使用a标签，如何实现a标签的功能**

```
// 通过 window.open 和 location.href 方法其实就可以实现。
// 分别对应了a标签的 blank 和 self 属性
```

4. **不使用循环API 来删除数组中指定位置的元素（如：删除第三位） 写越多越好**

这个题的意思就是，不能循环的API（如 for filter之类的）。

```

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// 方法一 ： splice 操作数组 会改变原数组
arr.splice(2, 1)


// 方法二 ： slice 截取选中元素 返回新数组 不改变原数组
arr.slice(0, 2).concat(arr.slice(3,))

// 方法三 delete数组中的元素 再把这个元素给剔除掉
delete arr[2]
arr.join(" ").replaceAll(/\s{1,2}/g," ").split(" ")
```

5. **深拷贝**

深拷贝和浅拷贝的区别就在于

- 浅拷贝： 对于复杂数据类型，浅拷贝只是把引用地址赋值给了新的对象，改变这个新对象的值，**原对象的值也会一起改变**。
- 深拷贝： 对于复杂数据类型，拷贝后地址引用都是新的，改变拷贝后新对象的值，**不会影响原对象的值**。

所以关键点就在于对复杂数据类型的处理，这里我写了两种写法，第二中比第一种有部分性能提升

```
const isObj = (val) => typeof val === "object" && val !== null;

// 写法1
function deepClone(obj) {
    // 通过 instanceof 去判断你要拷贝的变量它是否是数组（如果不是数组则对象）。

    // 1. 准备你想返回的变量（新地址）。
    const newObj = obj instanceof Array ? [] : {}; // 核心代码。

    // 2. 做拷贝；简单数据类型只需要赋值，如果遇到复杂数据类型就再次进入进行深拷贝，直到所找到的数据为简单数据类型为止。
    for (const key in obj) {
        const item = obj[key];
        newObj[key] = isObj(item) ? deepClone(item) : item;
    }

    // 3. 返回拷贝的变量。
    return newObj;
}




// 写法2 利用es6新特性 WeakMap弱引用 性能更好 并且支持 Symbol
function deepClone2(obj, wMap = new WeakMap()) {
  if (isObj(obj)) {
    // 判断是对象还是数组
    let target = Array.isArray(obj) ? [] : {};

    // 如果存在这个就直接返回
    if (wMap.has(obj)) {
      return wMap.get(obj);
    }

    wMap.set(obj, target);

    // 遍历对象
    Reflect.ownKeys(obj).forEach((item) => {
      // 拿到数据后判断是复杂数据还是简单数据 如果是复杂数据类型就继续递归调用
      target[item] = isObj(obj[item]) ? deepClone2(obj[item], wMap) : obj[item];
    });

    return target;
```

```
  } else {
    return obj;
  }
}
```

这道题主要是的方案就是，**递归加数据类型的判断**。

如是复杂数据类型，就递归的再次调用你这个拷贝方法 直到是简单数据类型后可以进行直接赋值

6. 手写call bind apply

call bind apply的作用都是可以进行修改this指向

- call 和 apply的区别在于参数传递的不同
- bind 区别在于最后会返回一个函数。

```

    // call
    Function.prototype.MyCall = function (context) {
      if (typeof this !== "function") {
        throw new Error('type error')
      }
      if (context === null || context === undefined) {
        // 指定为 null 和 undefined 的 this 值会自动指向全局对象(浏览器中为window)
        context = window
      } else {
        // 值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的实例对象
        context = Object(context)
      }

      // 使用Symbol 来确定唯一
      const fnSym = Symbol()

      //模拟对象的this指向
      context[fnSym] = this

      // 获取参数
      const args = [...arguments].slice(1)

      //绑定参数 并执行函数
      const result = context[fnSym](...args)

      //清除定义的this
      delete context[fnSym]

      // 返回结果
```

```
    return result
  }


  // call 如果能明白的话 apply其实就是改一下参数的问题
  // apply
  Function.prototype.MyApply = function (context) {
    if (typeof this !== "function") {
      throw new Error('type error')
    }

    if (context === null || context === undefined) {
      // 指定为 null 和 undefined 的 this 值会自动指向全局对象(浏览器中为window)
      context = window
    } else {
      // 值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的实例对象
      context = Object(context)
    }


    // 使用Symbol 来确定唯一
    const fnSym = Symbol()
    //模拟对象的this指向
    context[fnSym] = this

    // 获取参数
    const args = [...arguments][1]

    //绑定参数 并执行函数 由于apply 传入的是一个数组 所以需要解构
    const result = arguments.length > 1 ? context[fnSym](...args) :
context[fnSym]()

    //清除定义的this
    delete context[fnSym]

    // 返回结果   //清除定义的this
    return result
  }


  // bind
  Function.prototype.MyBind = function (context) {
    if (typeof this !== "function") {
      throw new Error('type error')
    }

```

```
    if (context === null || context === undefined) {
      // 指定为 null 和 undefined 的 this 值会自动指向全局对象(浏览器中为window)
      context = window
    } else {
      // 值为原始值（数字，字符串，布尔值）的 this 会指向该原始值的实例对象
      context = Object(context)
    }

    //模拟对象的this指向
    const self = this

    // 获取参数
    const args = [...arguments].slice(1)

    // 最后返回一个函数 并绑定 this 要考虑到使用new去调用，并且bind是可以传参的
    return function Fn(...newFnArgs) {
      if (this instanceof Fn) {
          return new self(...args, ...newFnArgs)
      }
          return self.apply(context, [...args, ...newFnArgs])
      }
  }
```

7. 手写实现继承

这里我就只实现两种方法了，ES6之前的寄生组合式继承 和 ES6之后的class继承方式。

```
  /**
  * es6之前  寄生组合继承
  */
  {
    function Parent(name) {
      this.name = name
      this.arr = [1, 2, 3]
    }

    Parent.prototype.say = () => {
      console.log('Hi');
    }

    function Child(name, age) {
      Parent.call(this, name)
      this.age = age
    }

    //  核心代码 通过Object.create创建新对象 子类 和 父类就会隔离
```

```
      // Object.create：创建一个新对象，使用现有的对象来提供新创建的对象的__proto__
      Child.prototype = Object.create(Parent.prototype)
      Child.prototype.constructor = Child
    }



    /**
    *   es6继承 使用关键字class
    */
     {
      class Parent {
        constructor(name) {
          this.name = name
          this.arr = [1, 2, 3]
        }
      }
      class Child extends Parent {
        constructor(name, age) {
          super(name)
          this.age = age
        }
      }
    }
```

补充一个小知识， ES6的Class继承在通过 Babel 进行转换成ES5代码的时候 使用的就是 寄生组合式继承。

继承的方法有很多，记住上面这两种基本就可以了！

8. 手写 new 操作符

首先我们要知道 new一个对象的时候他发生了什么。

其实就是在内部生成了一个对象，然后把你的属性这些附加到这个对象上，最后再返回这个对象。

```
function myNew(fn, ...args) {
  // 基于原型链 创建一个新对象
  let newObj = Object.create(fn.prototype)

  // 添加属性到新对象上 并获取obj函数的结果
  let res = fn.call(newObj, ...args)

  // 如果执行结果有返回值并且是一个对象，返回执行的结果，否则，返回新创建的对象
  return res && typeof res === 'object' ? res : newObj;
}
```

9. **js执行机制 说出结果并说出why**

这道题考察的是，js的任务执行流程，对宏任务和微任务的理解

```
console.log("start");

setTimeout(() => {
  console.log("setTimeout1");
}, 0);

(async function foo() {
  console.log("async 1");

  await asyncFunction();

  console.log("async2");

})().then(console.log("foo.then"));

async function asyncFunction() {
  console.log("asyncFunction");

  setTimeout(() => {
    console.log("setTimeout2");
  }, 0);

  new Promise((res) => {
    console.log("promise1");

    res("promise2");
  }).then(console.log);
}

console.log("end");
```

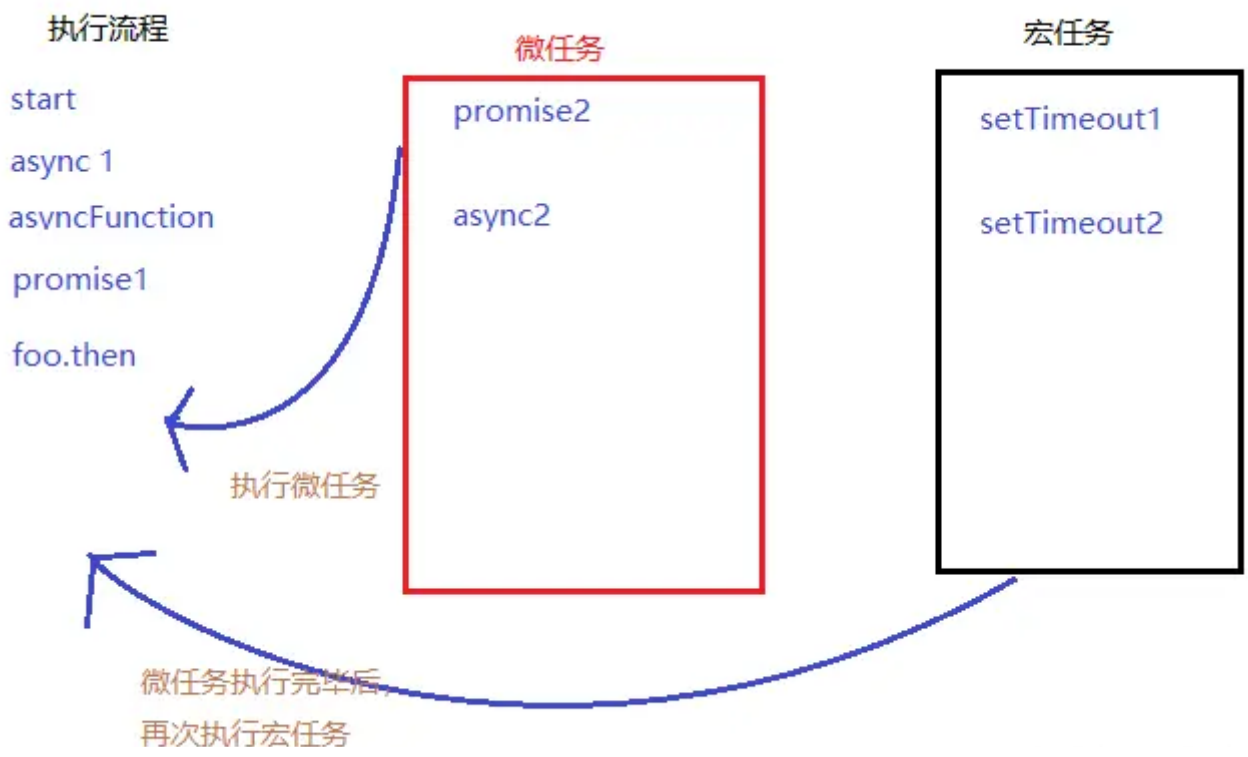
提示：

1. script标签算一个宏任务所以最开始就执行了
2. async await 在await之后的 Promise 都会被放到微任务队列中去

开始执行：

- 最开始碰到 console.log("start"); 直接执行并打印出 `start`
- 往下走，遇到一个 setTimeout1 就放到 宏任务队列
- 碰到立即执行函数 foo， 打印出 `async 1`

- 遇到 await 堵塞队列，先 `执行await的函数`
- 执行 asyncFunction 函数， 打印出 `asyncFunction`
- 遇到第二个 setTimeout2， `放到宏任务队列`
- new Promise 立即执行，打印出 `promise1`
- 执行到 res("promise2") 函数调用，就是Promise.then。 `放到微任务队列`
- asyncFunction函数就执行完毕， 把后面的打印 async2 会放到 `微任务队列`
- 然后打印出立即执行函数的then方法 `foo.then`
- 最后执行打印 `end`
- 开始执行微任务的队列 打印出第一个 `promise2`
- 然后打印第二个 `async2`
- 微任务执行完毕，执行宏任务 打印第一个 `setTimeout1`
- 执行第二个宏任务 打印 `setTimeout2` 、
- 就此，函数执行完毕



画工不好，能理解到意思就行😭。 看看你们的想法和答案是否和这个流程一致

10. **如何拦截全局Promise reject，但并没有设定 reject处理器 时候的错误**

这道题我是没写出来，最开始想着 trycatch 但这个并不是全局的。

后续查了资料才发现 是用一个window上面的方法

```
// 使用Try catch 只能拦截try语句块里面的
try {
  new Promise((resolve, reject) => {
    reject("WTF 123");
  });
} catch (e) {
  console.log("e", e);
  throw e;
}

// 使用 unhandledrejection 来拦截全局错误   (这个是对的)
window.addEventListener("unhandledrejection", (event) => {
  event && event.preventDefault();
  console.log("event", event);
});
```

11. 手写实现sleep

这个我只通过了一种方法实现，就是刚刚我们在上面js执行流程中我有提过。 **await** 会有异步堵塞的意思

还有一个方法是我在网上找到的方法，通过完全堵塞进程的方法来实现 `这个有点吊`

```
    // 使用 promise 配合await的异步方法来实现 sleep
    {
      (async () => {
        console.log('start');
        await sleep(3000)
        console.log('end');

        function sleep(timer) {
          return new Promise(res => {
            setTimeout(() => {
              res()
            }, timer);
          })
        }
      })();
    }

    // 方法二 这是完全堵塞进程来达到sleep
    {
      (async () => {
        console.log('start');
        await sleep(3000)
        console.log('end');
```

```
24
25          function sleep(delay) {
26            let t = Date.now();
27            while (Date.now() - t <= delay) {
28              continue;
29            }
30          };
31        })()
32      }
```

12. 实现add(1)(2) =3

光这个的话，可以通过闭包的方式实现了

我给这个加了一个难度，如何才能实现一直调用

```
1     // 题意的答案
2   const add = (num1) => (num2)=> num2 + num1;
3
4
5   // 我自己整了一个加强版 可以无限链式调用 add(1)(2)(3)(4)(5)....
6   function add(x) {
7       // 存储和
8       let sum = x;
9
10      // 函数调用会相加，然后每次都会返回这个函数本身
11      let tmp = function (y) {
12        sum = sum + y;
13        return tmp;
14      };
15
16      // 对象的toString必须是一个方法 在方法中返回了这个和
17      tmp.toString = () => sum
18      return tmp;
19  }
20
21  alert(add(1)(2)(3)(4)(5))
```

无限链式调用实现的关键在于 **对象的 toString 方法:** `每个对象都有一个 toString() 方法，当该对象被表示为一个文本值时，或者一个对象以预期的字符串方式引用时自动调用`。

也就是我在调用很多次后，他们的结果会`存在add函数中的sum变量上`，当我alert的时候 add会**自动调用 toString**方法 `打印出 sum，也就是最终的结果`

13. 两个数组中完全独立的数据

就是找到仅在两个数组中出现过一次的数据

```
vvar a = [1, 2, 4], b = [1, 3, 8, 4]
const newArr = a.concat(b).filter((item, _, arr) => {
  return arr.indexOf(item) === arr.lastIndexOf(item)
})
```

最终出来的结果是 `[2,3,8]`， 原理其实很简单： 合并两个数组，然后查找数组的**第一个出现的索引**和**最后一个出现的索引**是否一致就可以判断是否是独立的数据了。

14. 判断完全平方数

就是判断一个数字能不能被开平方， 比如9的开平方是3 是对的。 5没法开平方就是错的。

```
var fn = function (num) {
  return num ** 0.5 % 1 == 0
};
```

原理就是，开平方后判断是否是正整数就行了

15. 函数执行 说出结果并说出why

```
function Foo() {
  getName = function () {
    console.log(1);
  };
  return this;
}

Foo.getName = function () {
  console.log(2);
}

Foo.prototype.getName = function () {
  console.log(3);
}

var getName = function () {
  console.log(4);
}

function getName() {
  console.log(5)
```

```
}

Foo.getName();

getName();

Foo().getName()

getName();

new Foo.getName();

new Foo().getName()

new new Foo().getName()
```

这道题其实就是看你对作用域的关系的理解吧

执行结果：

- **执行 Foo.getName()**, 执行 `Foo函数对象上的的静态方法。` 打印出 `2`
- **执行 getName()**， 就是执行的getName变量的函数。打印 `4`
  - 为什么这里是 执行的 变量getName，而不是函数getName呢。这得归功于 `js的预编译`
  - js在执行之前进行预编译，会进行 `函数提升` 和 `变量提升`
  - 所以函数和变量都进行提升了，但是 `函数声明的优先级最高` ，会被提升至 `当前作用域最顶端`
  - 当在执行到后面的时候会导致getName被重新赋值，就会把执行结果为 `4` 的这个函数赋值给变量
- **执行 Foo().getName()**， `调用Foo执行后返回值上的getName方法。` Foo函数执行了，里面会给 `外面的getName函数重新赋值` ，并返回了this。 也就是执行了this.getName。所以打印出了 `1`
- **执行 getName()**， 由于上一步，函数被重新赋值。所以这次的结果和上次的结果是一样的，还是为 `1`
- **执行 new Foo.getName()**， 这个 new 其实就是new了Foo上面的 `静态方法getName` 所以是 `2` 。 当然如果你们在这个函数里面打印this的话，会发现指向的是一个新对象 也就是new出来的一个新对象
  - 可以把 Foo.getName()看成一个整体，因为 `这里 . 的优先级比 new 高`
- **执行 new Foo().getName()**，这里函数执行 new Foo() 会返回一个对象，然后调用这个 `对象原型上的getName方法` ， 所以结果是 `3`
- **执行 new new Foo().getName()**, 这个和上一次的结果是一样，上一个函数调用后并咩有返回值，所以在进行new的时候也没有意义了。 最终结果也是 `3`

16. 原型调用面试题 说出结果并说出 why

```
function Foo() {
  Foo.a = function () {
    console.log(1);
  };
  this.a = function () {
    console.log(2);
  };
}

Foo.prototype.a = function () {
  console.log(4);
};

Function.prototype.a = function () {
  console.log(3);
};


Foo.a();

let obj = new Foo();
obj.a();
Foo.a();
```

执行结果：

- **执行Foo.a()，Foo本身目前并没有a这个值**，就会通过 `proto` 进行查找，但是

```
Foo.__proto__ === Function.prototype
true
```

- ， 所以输出是 `3`
- **new 实例化了 Foo 生成对象 obj**，然后调用 **obj.a()**，但是在Foo函数内部给这个obj对象附上了a函数。 所以结果是 `2` 。 如果在内部没有给这个对象赋值a的话，就会去到原型链查找a函数，就会打印4.
- **执行Foo.a()**， 在上一步中Foo函数执行，**内部给Foo本身赋值函数a**，所以这次就打印 `1`

17. 数组分组改成减法运算

这个题的意思就是 `[5, [[4, 3], 2, 1]]` 变成 `(5 - ((4 - 3) - 2 - 1))` 并执行。 且 `不能使用eval()`

方法一： 既然不能用 eval， 那我们就用**new Function**吧🤭

方法二： 当然方法一有点违背了题意，所以还有第二种方法

```
var newArr = [5, [[4, 3], 2, 1]]

    // 1. 取巧
    // 转为字符串
    let newStringArr = `${JSON.stringify(newArr)}`
    // 循环修改括号和减号
    let fn = newStringArr.split("").map((el) => {
      switch (el) {
        case "[":
          return '('
        case "]":
          return ')'
        case ",":
          return '-'
        default:
          return el
      }
    }).join("")
    // 最终通过new Function 调用可以了!
    new Function("return " + fn)()


    // 2. 方法二
    function run(arr) {
      return arr.reduce((pre, cur) => {
        let first = Array.isArray(pre) ? run(pre) : pre
        let last = Array.isArray(cur) ? run(cur) : cur
        return first - last
      })
    }
    run(newArr)

```

- 方法一的原理就很简单，转成字符串循环修改括号和减号在进行拼接。最终 `通过 new Function 调用` 就可以了
- 方法二的意思就是通过 `reduce 进行一个递归调用` 的意思。 如果左边 `不是数组` 就可以减去右边的，但如果 `右边是数组的话，就要把右边的数组先进行减法运算` 。也是就减法括号运算的的优先级.

18. 手写数组的 flat

```
const flat = function (arr, deep = 1) {
  // 声明一个新数组
  let result = []

  arr.forEach(item => {
    if (Array.isArray(item) && deep > 0) {
      // 层级递减
      // deep--  来自评论区的大佬指正：deep - 1
      // 使用concat链接数组
      result = result.concat(flat(item, deep - 1))
    } else {
      result.push(item)
    }
  })
  return result
}
```

- 原理就是，先在内部生成一个新数组，遍历原来的数组
- 当原数组内 存在数组 `并且层级deep大于等于1时` 进行递归, 如果 `不满足这个条件就可以直接push数据到新数组` 去
- 递归同时要先把层级减少， 然后通过 `concat 链接递归出来的数组`
- 最终返回这个数组就可以了

19. 数组转为tree

最顶层的**parent 为 -1** ，其余的 **parent都是为 上一层节点的id**

```
let arr = [
  { id: 0, name: '1', parent: -1, childNode: [] },
  { id: 1, name: '1', parent: 0, childNode: [] },
  { id: 99, name: '1-1', parent: 1, childNode: [] },
  { id: 111, name: '1-1-1', parent: 99, childNode: [] },
  { id: 66, name: '1-1-2', parent: 99, childNode: [] },
  { id: 1121, name: '1-1-2-1', parent: 112, childNode: [] },
  { id: 12, name: '1-2', parent: 1, childNode: [] },
  { id: 2, name: '2', parent: 0, childNode: [] },
  { id: 21, name: '2-1', parent: 2, childNode: [] },
  { id: 22, name: '2-2', parent: 2, childNode: [] },
  { id: 221, name: '2-2-1', parent: 22, childNode: [] },
  { id: 3, name: '3', parent: 0, childNode: [] },
  { id: 31, name: '3-1', parent: 3, childNode: [] },
  { id: 32, name: '3-2', parent: 3, childNode: [] }
]

```

```
    function arrToTree(arr, parentId) {
      // 判断是否是顶层节点，如果是就返回。不是的话就判断是不是自己要找的子节点
      const filterArr = arr.filter(item => {
        return parentId === undefined ? item.parent === -1 : item.parent === parentId
      })

      // 进行递归调用把子节点加到父节点的 childNode里面去
      filterArr.map(item => {
        item.childNode = arrToTree(arr, item.id)
        return item
      })

      return filterArr
    }

    arrToTree(arr)
```

- 这道题也是利用递归来进行的，在最开始会进行`是否是顶层节点的判断`
- 如果是就直接返回，如果不是则`判断是不是自己要添加到父节点的子节点`
- 然后再一层一层把节点加入进去
- 最后返回这个对象

20. 合并数组并排序去重

题意就是， 我有两个数组，把他们**两个合并**。然后并**去重**， `去重的逻辑是哪儿边的重复次数更多`，我就留下哪儿边的。

比如下面的数组中，`一边有两个数字5`， `另一半有三个数字5` 。则我需要`留下三个数字5`， `去掉两个数字5`。 循环往复，最后得到的结果在进行排序。

- 数组一： [1, 100, 0, 5, 1, 5]
- 数组二： [2, 5, 5, 5, 1, 3]
- 最终的结果： **[0, 1, 1, 2, 3, 5, 5, 5, 100]**

```
  // 判断出现次数最多的次数
    function maxNum(item, arr) {
      let num = 0;
      arr.forEach(val => {
        item === val && num++
      })

      return num
    }
```

```
10
11      function fn(arr1, arr2) {
12        // 使用Map数据类型来记录次数
13        let obj = new Map();
14
15        // 合并数组并找出最多的次数，并以键值对存放到Map数据类型
16        [...arr1, ...arr2].forEach(item => {
17          let hasNum = obj.get(item)
18          let num = 1
19          if (hasNum) {
20            num = hasNum + 1
21          }
22          obj.set(item, num)
23        })
24
25        // 存放合并并去重之后的数组
26        let arr = []
27        // 遍历Map数据类型 然后把次数最多的直接push到新数组
28        for (const key of obj.keys()) {
29          if (obj.get(key) > 1) {
30            for (let index = 0; index < Math.max(maxNum(key, arr1), maxNum(key,
    arr2)); index++) {
31              arr.push(key)
32            }
33          } else {
34            arr.push(key)
35          }
36        }
37
38      // 最后进行排序
39        return arr.sort((a, b) => a - b)
40      }
```

- 这个题的思路其实就是，我先把 `两个数组合并起来`
- 并以 `键值对的方式存放到Map数据类型` ，`键就是数据，而值就是这个数据出现的次数`
- 生成一个新数组，用来 `存放合并之后的数组`
- `遍历这个Map数据类型` ,如果这个数据 `出现的次数大于一` ，那么就去 `寻找两个数组中谁出现的次数更多` ，把 `出现次数更多的这个数据，循环push到新数组中` 。 如果 `出现次数等于一` ，那就直接push到新数组中即可。
- 最后再把 `数组进行排序` ，然后返回新数组就可。