JavaScript 面试题

JS 的数据类型有哪些？它们在内存里面的模型是怎样的？堆和栈有什么区别?

原始数据类型（存放在栈内存中）：

- number：任何类型的数字，整数或浮点数
- bigInt：任意长度的整数
- string：字符串
- boolean：布尔值
- null：未知的值
- undefined：未定义的值
- symbol：唯一的标识符

引用数据类型（存放在堆内存中）：

- object（细分为object、array、function）

栈内存:

- 存储原始数据类型
- 按值访问
- 存储的值大小固定
- 由系统自动分配内存空间
- 空间小，运行效率高
- 先进后出，后进先出

堆内存:

- 存储引用数据类型
- 按引用地址访问
- 存储的值大小不定，可动态调整
- 由代码进行指定分配
- 空间大，运行效率相对较低
- 无序存储，可根据引用直接获取

对象的键支持什么类型?

支持String、Symbol 类型

Symbol：表示一个独一无二的值。通过Symbol函数就可以创建一个Symbol类型的数据

null 和 undefined 有什么区别？什么是NaN

他们都属于 JavaScript 的7种基本类型之一，都属于虚值

- undefined 是未指定特定值的变量的默认值，或者没有显式返回值的函数，还包括对象中不存在的属性，这些 JS 引擎都会为其分配 undefined 值
- null 是“不代表任何值的值”。null 是已明确定义给变量的值

NaN 表示“非数字”，是JS中的一个值。该值是将数字转换会执行为非数字值的运算结果

说一下常见的检测数据类型的几种方法?

- typeof：其中数组、对象、null 都会被判断为 Object，其他判断都正确
- instanceof：只能判断引用数据类型，不能判断基本数据类型
- constructor：它有2个作用，一是判断数据的类型，二是对象实例通过 constructor 对象访问它的构造函数。需要注意的事情是如果创建了一个对象来改变它的原型，constructor·就不能来判断数据类型了
- Object.prototype.toString.call()

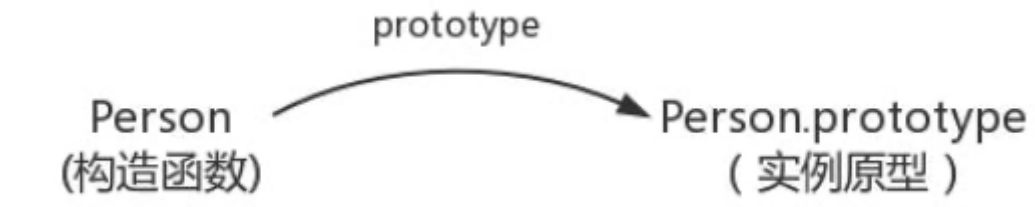
JS 原型链是什么？原型链的顶端是什么？Object 的原型是什么？Object的原型的原型是什么？在数组原型链上实现删除数组重复数据的方法?

首先明白原型是什么？在ES6之前，JS没有类和继承的概念，JS 是通过原型来实现继承，在 JS 中一个构造函数默认带有一个 prototype 属性，这个属性是一个对象，同时这个 prototype 对象自带一个 constructor 属性，这个属性指向我们这个构造函数，同时每个实例都会有一个 __proto__ 属性执行这个 prototype 对象，我们可以把这个叫做隐式原型，我们在使用一个实例的方法时，会先检查这个实例是否有这个方法，没有的话会检查这个 prototype 对象是否有这个方法

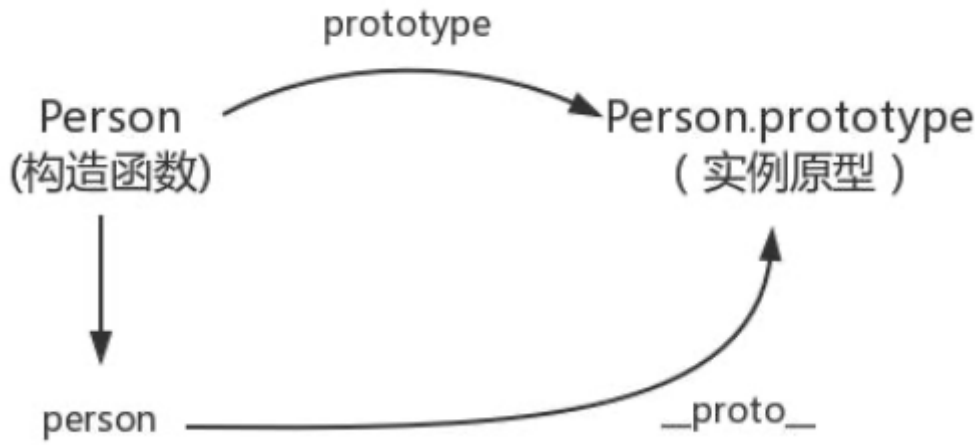
基于这个规则，如果让原型对象指向另外一个类型的实例，即 constructor.prototype = instance，这个时候如果试图引用 constructor 构造的实例 instance 的某个属性，首先会在 instance 内部属性找一遍，接着会在 instance.__proto__（constructor.prototype）中寻找，这样就形成了一条原型链，原型链的顶端是 Object.prototype

补充学习：

每个函数都会有个 prototype 属性，这个属性指向了一个对象，这个对象就是调用该函数而创建的实例的原型，那么什么是原型，可以这样理解，每一个 JavaScript 对象在创建的时候，就会关联另一个对象，这个对象就是我们所说的原型，每一个对象都会以原型链继承属性，如图：



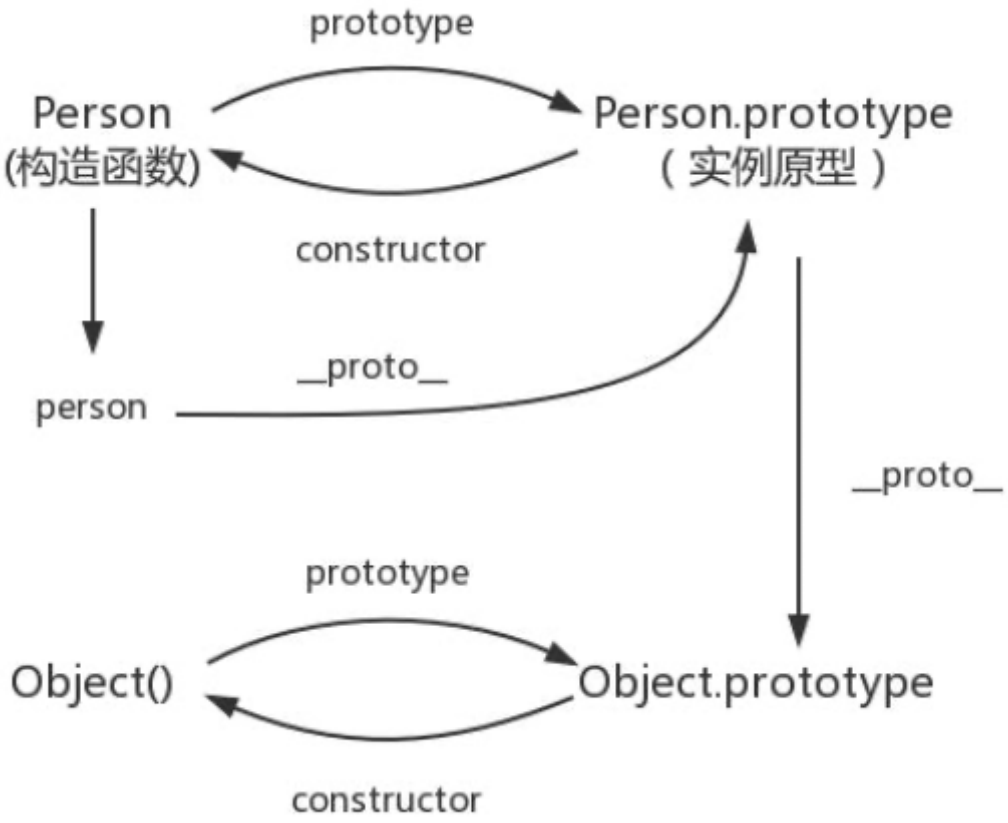
那么怎么表示实例与实例原型的关系呢？这时候就要用到第二个属性__proto__，这是每一个JS对象都会有的一个属性，指向这个对象的原型，如图：



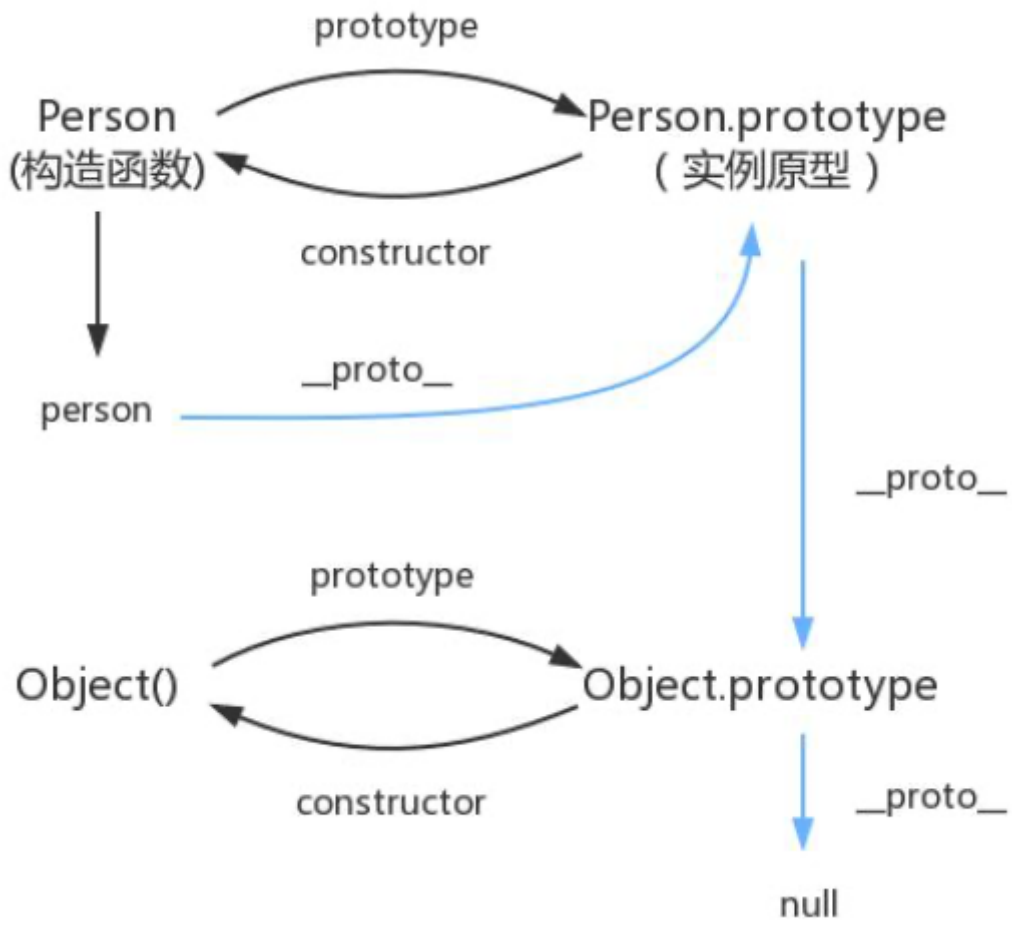
既然实例对象和构造函数都可以指向原型，那么原型是否有实例指向构造函数或者实例呢？指向实例是没有的，因为一个构造函数可以生成多个实例，但是原型有属性可以直接指向构造函数，通过 constructor 即可

接下来讲解实例和原型的关系：

当读取实例的属性时，如果找不到，就会查找与对象相关的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层，那么原型的原型是什么呢？首先，原型也是一个对象，既然是对象，我们就可以通过构造函数的方式创建它，所以原型对西那个就是通过 Object 构造函数生成的，如图：



Object.prototype 指向 null，null 表示没有对象，即该处不应有值，所以 Object.prototype 没有原型，如图：



图中蓝色的线就是原型链

最后补充三点：

```
function Person () {}
const person = new Person()
Person === person.constructor
```

原本 person 中没有 constructor 属性，当不能读取到 constructor 属性时，会从 person 的原型中读取，所以指向构造函数 Person

__proto__：

绝大部分浏览器支持这个

前面说到，每个对象都会从原型继承属性，但是引用《你不知道的JavaScript》中的话，继承意味着复制操作，然而 JS 默认不会复制对象的属性，相反，JS 只是在两个对象之间创建一个关联，这样子一个对象就可以通过委托访问另一个对象的属性和函数，所以与其叫继承，叫委托更合适

箭头函数与普通函数的区别?

- 箭头函数没有this，所以需要通过查找作用域链来确定this的值，这就意味着如果箭头函数被非箭头函数包含，this 绑定的就是最近一层非箭头函数的this

- 箭头函数没有自己的 arguments 对象，但是可以访问外围函数的 arguments 对象
- 不能通过 new 关键字调用，同样也没有 new.target 值和原型

说一下类的创建和继承

类的创建：

- **es5：**new 一个 function，在这个function的原型上添加属性和方法，就可以生成一个类
- **es6：**使用 class 创建一个类，非标准的方法访问原型，然而它并不存在与Person.prototype 中，实际上它来自 Object.prototyppe，当使用 obj.__proto__时，可以理解为返回 Object.getPrototype(obj)

继承：

- **原型链继承：**将父类的实例作为子类的原型

```
function Parent () {}
Parent.prototype.name = '父类'
Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child () {
    this.subname = '子类'
}
Child.prototype = new Parent()
const child = new Child()
child.getName()
```

特点：实例既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：要为子类新增属性和方法，不能放到构造器中，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

- **构造继承：**使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类

```
function Parent () {
    this.parentName = '父类'
}
Parent.prototype.name = '父类'
Parent.prototype.getName = function () {
    console.log(this.name)
}
```

```
8
9  function Child () {
10     Parent.call(this)
11     this.name = '子类'
12 }
13 const child = new Child()
14 console.log(child.parentName)
```

特点：构造函数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

缺点：只能继承父类实例的属性和方法，不能继承原型上的属性和方法，无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

- **实例继承：**为父类实例添加新特性，作为子类实例返回（用的不多）

特点：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：实例是父类的实例，不是子类的实例，不支持多继承

- **拷贝继承：**（用的不多）

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）

- **组合继承：**相当于构造继承和原型链继承的组合体，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

```
1  function Parent () {
2      this.parentName = '父类'
3  }
4  Parent.prototype.name = '父类'
5  Parent.prototype.getName = function () {
6      console.log(this.name)
7  }
8
9  function Child () {
10     Parent.call(this)
11     this.name = '子类'
12 }
13 Child.prototype = new Parent()
14 Child.prototype.constructor = Cat
15 const child = new Child()
16 console.log(child.parentName)
```

特色：可以继承实例属性/方法，也可以继承原型属性/方法

缺点：调用了两次父类构造函数，生产了两份实例

- **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时，就不会初始化两次实例方法/属性，避免了组合继承的缺点

```
function Parent () {
    this.parentName = '父类'
}
Parent.prototype.name = '父类'
Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child () {
    Parent.call(this)
    this.name = '子类'
}

(function () {
    // 创建一个没有实例方法的类
    const Super = function () {}
    Super.prototype = Parent.prototype
    // 将实例作为子类的原型
    Child.prototype = new Super()
})()

const child = new Child()
console.log(child.parentName)
```

- class 继承：主要通过 extends 和 super 这两个关键字

```
class Parent {
    constructor (name) {
        this.name = name
    }
    getName () {
        console.log(this.name)
    }
}
class Child extends Parent {
    constructor (name) {
        super(name)
```

```
        this.sex = 'boy'
    }
}
```

for in 和 for of 的区别?

第一种回答:

两者都可以用于遍历

- for in 遍历的是数组的索引（index），使用 for in 会遍历数组所有的可枚举属性，包括原型，总是得到对象的 key 或数组、字符串的下标
- for of 遍历的是数组元素值（value），使用 for of 遍历的只是数组内的元素，不包括原型属性和索引，总是得到对象的 value 或数组、字符串的值

第二种回答:

- 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

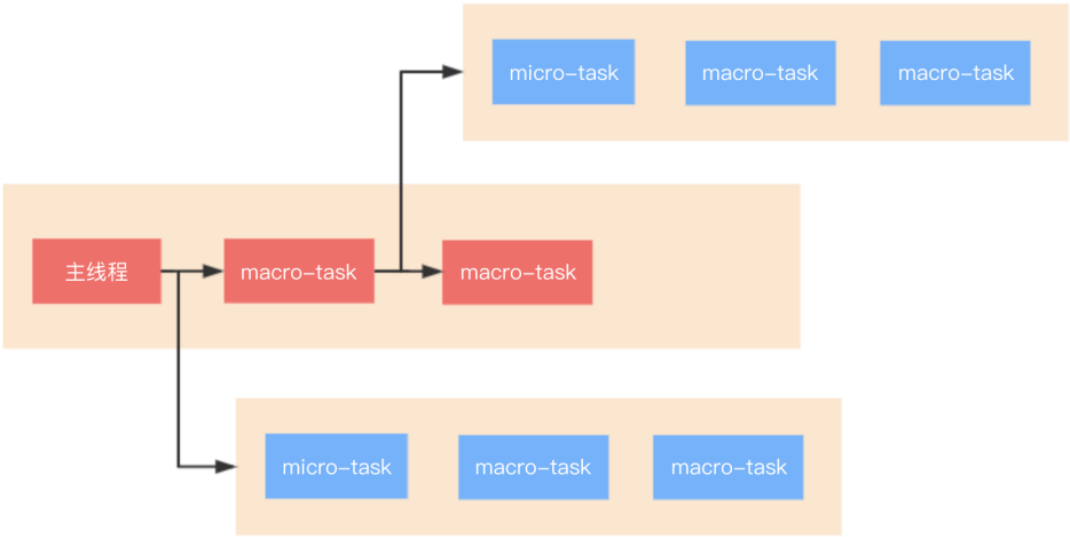
JS 事件循环机制（事件循环机制整体流程是什么）?

参考 [说说JS的事件循环机制 (含满分答题技巧)]

浏览器的事件循环机制:

JavaScript 代码执行的过程中，除了依靠函数调用栈来处理函数的执行顺序外，还依靠任务队列（task queue）来处理另外一些代码的执行。整个执行过程，我们称之为事件循环过程。一个线程中，事件循环是唯一的，但是任务队列可以拥有多个。任务队列又分成 macro-task（宏任务）和 micro-task （微任务），在最新的标准中，它们被分别成为 task 与 jobs

具体流程如图所示:



综上所述，先执行宏任务，然后执行该宏任务产生的微任务，若微任务在执行过程中产生了新的微任务，，则继续执行微任务，微任务执行完毕后，再回到宏任务中进行下一轮循环。直到任务队列中的所有任务执行完毕

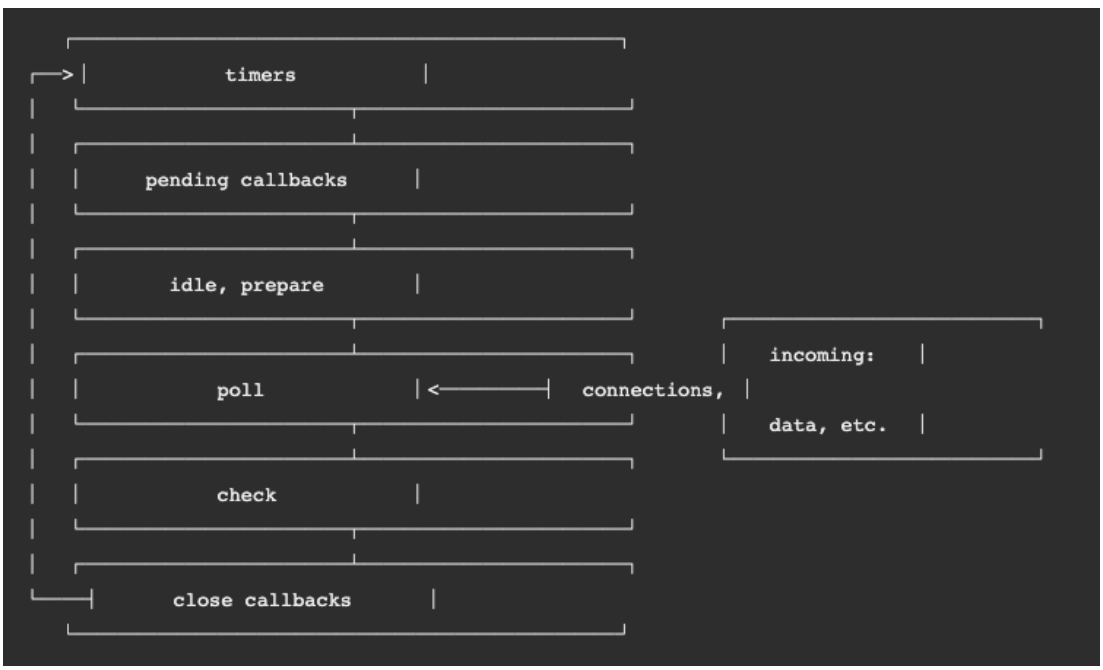
注意点：async/await 执行顺序

我们知道 async 隐式返回 Promise 作为结果的函数，那么可以简单理解为，await 后面的函数执行完毕时，await 会产生一下微任务（Promise.then 是微任务），但是我们要注意这个微任务产生的时机，它是执行 await 之后，直接跳出 async 函数，执行其他代码（此处就是协程的运作，A暂停执行，控制权交给B）。其他代码执行完毕后，再回到 async 函数去执行剩下的代码，然后把 await 后面的代码注册到微任务中

node的事件循环机制：

事件循环就是 node 处理非阻塞 I/O 操作的机制，node 中事件循环的实现是依靠 libuv 引擎

官网的 node 事件循环简化图：



输入数据阶段（incoming data）=> 轮询阶段（poll） => 检查阶段（check） => 关闭事件回调函数（close callback） => 定时器检测阶段（timers）=> I/O 事件回调阶段（I/O callbacks） => 闲置阶段（idle，prepare）=> 轮询阶段

阶段概述：

- 轮询阶段(poll)：检索新的 I/O 事件；执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，那些由计时器和setImmediate() 调度之外，其余情况 node 将在适当的时候在此阻塞）
- 检查阶段(check)：setImmediate() 回调函数在这里执行
- 关闭事件回调函数(close callback)：一些关闭的回调函数，如：socket.on('close')
- 定时器检测阶段(timers)：本阶段执行 timer 的回调，即 setTImeout、setInterval 里面的回调
- I/O 事件回调阶段(I/O callbacks)：执行延迟到下一循环迭代的 I/O 回调，即上一轮循环中未被执行的一些 I/O 回调
- 闲置阶段(idle，prepare)：仅内部系统使用

macro-task（宏任务）大概包括：

- script（整体代码）
- setTimeout
- setInterval
- setImmediate
- I/O
- UI render

micro-task（微任务）大概包括:

- Promise
- Async/Await（实际上就是Promise）
- MutationObserver（html5新特性）
- process.nextTick（与普通微任务有区别，在微任务队列执行之前执行）

浏览器和node的事件循环最主要的区别是什么?

- Node端，microtask（微任务） 在事件循环的各个阶段之间执行
- 浏览器端，microtask（微任务） 在事件循环的 macrotask（宏任务） 执行完之后执行

什么情况下 setTimeout(() => {}, 1000) 执行的时间会大于1s?

- 当前的任务执行时间过久的话，会导致setTimeout的任务被延后执行了
- 如果 setTimeout 存在嵌套调用，那么系统会设置最短时间间隔为4毫秒

说说前端中的事件流（事件模型）

定义：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
- 目标阶段
- 事件冒泡阶段

备注：IE只支持事件冒泡

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，分别对应响应的处理函数，监听到捕获事件，先暂缓执行，直到冒泡事件被执行后再执行捕获事件

说一下事件委托（事件代理）

简介：事件委托指的是，不在事件的触发元素上设置监听函数，而是在其父元素上设置，通过事件冒泡，父元素可以监听到触发元素上事件的触发，通过判断事件发生元素DOM的类型，来做出不同的响应

好处：比较适合动态元素的绑定，新添加的触发元素也会有监听函数，也可以有事件触发机制

优点：减少内存消耗和事件绑定，提高效率

实际应用：可以在父元素层面阻止事件向子元素传播，也可以替代子元素执行某些操作

说一下闭包

（1）什么是闭包?

闭包是指有权访问另外一个函数作用域中的变量的函数

闭包就是函数的局部变量的集合，只是这些局部变量在函数返回后会继续存在。闭包就是函数的“堆栈”，在函数返回后并不释放，我们也可以理解为这些函数堆栈并不在栈上分配而是在堆上分配。当在一个函数内定义一个函数就会产生闭包

（2）优缺点：

- **优点：**可以避免全局变量的污染
- **缺点：**闭包会常驻内存，会增加内存使用量，使用不当很容易造成内存泄漏

（3）特性：

- 函数嵌套函数
- 函数内部可以引用外部的参数和变量
- 参数和变量不会被垃圾回收机制回收

(4) 为什么要使用闭包?

- **匿名自执行函数**：我们知道所有的变量，如果不加上 var 关键字，则默认地会添加到全局对象的属性上去，这样的临时变量加到全局对象会有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度（因为变量的取值是需要从原型链上遍历的）。除了每次使用变量都是用 var 关键字外，我们在实际情况下经常遇到这样一种结果，即有的函数只需要执行一次，其内部无需维护，可以用闭包
- **结果缓存**：我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留
- **封装**：实现类和继承

(5) 实际开发中的应用：

- 模拟私有变量的实现
- 柯里化：把“接受n个参数的1个函数”改造为“只接受一个参数的n个互相嵌套的函数”的过程
- 偏函数：固定你函数的某一个或几个参数，然后返回一个新的函数（这个函数用于接收剩下的参数）。
- 函数的防抖、节流

说一下函数柯里化?

柯里化（currying）是把接收多个参数的原函数变换成接受一个单一参数（原来函数的第一个参数的函数）并返回一个新的函数，新的函数能够接受余下的参数，并返回和原函数相同的结果

- 参数对复用
- 提高实用性
- 延迟执行

只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。柯里化的函数可以延迟接收参数，就是比如一个函数需要接收的参数是两个，执行的时候必须接收两个参数，否则没法执行。但是柯里化后的函数，可以先接收一个参数

```
// 普通的add函数
function add (x, y) {
    return x + y
}
// Currying 后
function curryingAdd (x) {
    return function (y) {
        return x + y
```

```
    }
}
add(1, 2)           // 3
curryingAdd(1)(2)   // 3
```

深浅拷贝对象的区别和实现

如果是数组，我们可以利用数组的一些方法，比如 slice、concat 方法返回一个新数组的特性来实现拷贝，但假如数组嵌套来对象或者数组的话，使用 concat 方法克隆并不完整，如果数组元素是基本类型，就会拷贝一份，互不影响，而如果是对象或数组，就会只拷贝对象和数组的引用，这样我们无论在新旧数组进行了修改，两者都会发生变化，我们把这种复制引用的拷贝方法称为浅拷贝

深拷贝就是指完全的拷贝一个对象，即使嵌套了对象，两者也互相分离，修改一个对象的属性，不会影响到另外一个

浅拷贝的实现：

（1）Object.assign()：可以把任意多个的源对象自身的可枚举属性拷贝给目标对象，然后返回目标对象

（2）展开运算符

（3）Array.prototype.concat()

（4）Array.prototype.slice()

```
const shallowCopy = function (obj) {
    // 只拷贝对象
    if (typeof obj !== 'object') return
    // 根据 obj 的类型判断是新建一个数组还是对象
    const newObj = obj instanceof Array ? [] : {}
    // 遍历 obj，并且判断是 obj 的属性才拷贝
    for (const key in obj) {
        if (obj.hasOwnProperty(key)) {
            newObj[key] = obj[key]
        }
    }
    return newObj
}
```

深拷贝的实现

（1）使用JSON.parse(JSON.stringify(obj)) 可实现对数组和对象的深拷贝，但是不能拷贝函数和正则

（2）使用 Object.create() 能进行深拷贝

（3）深拷贝的简单实现：

```
// ES5
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;
}

function deepClone (target) {
    const map = new Map()
    function clone (target) {
        if (isObject(target)) {
            let cloneTarget = isArray(target) ? [] : {}
            if (map.get(target)) {
                return map.get(target)
            }
            map.set(target, cloneTarget)
            for (const key in target) {
                cloneTarget[key] = clone(target[key])
            }
            return cloneTarget
        } else {
            return target
        }
    }
    return clone(target)
}
```

说一下前端模块化

参考 [前端模块化：CommonJS,AMD,CMD,ES6]

前端模块化就是将复杂的文件拆分成一个个独立的模块，比如 JS 文件等，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，所以就有了 commonJS、AMD、CMD等规范，以及用于 JS 打包（编译等处理）的工具 webpack

CommonJS：开始于服务器端的模块化，同步定义的模块化，每个模块都是单独的作用域。Node.js是commonJS 规范的主要实践者，它有四个重要的环境变量为模块化的实现提供支持：module、exports、require、global。实际使用时，用 module.exports 定义当前模块对外输出的接口（不推荐直接用 exports），用 require 加载模块

```
// 定义模块
modules.exports = {
    // 需要向外暴露的函数、变量
}
// 引用模块
const module = require('模块路径')
```

commonJS 用同步的方式加载模块。在服务端，模块文件都存在本地磁盘，读取非常快，所以这样做不会有问题。但是在浏览器端，限于网络原因，更合理的方案是使用异步加载

AMD：AMD规范采用异步方式加载模块，模块的加载不会影响他后面语句的运行，所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。requireJS 实现了AMD规范，主要用于解决下述两个问题

- 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
- 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：用 require.config() 指定引用路径，用 define(id?, dependencies?, factory) 定义模块；require(([dependencies], factory) 加载模块

```
// requireJS例子
// 定义模块
define(['dependency'], function () {
})
// 加载模块
require(['myModule'], function (ctx) {})
```

总结：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

CMD：CMD 是另一种js模块化方案，它与AMD很类似，不同点在于：**AMD 推崇依赖前置、提前执行；CMD 推崇依赖就近、延迟执行。**此规范其实是在 sea.js 推广过程中产生的

ES6 Module：其模块功能主要由两个命令构成：export 和 import。export 命令用于规定模块的对外接口，import 命令用于输入其他模块提供的功能。ES6的模块不是对象，import 命令会被 JavaScript 引擎静态分析，在编译时就引入模板代码，而不是在代码运行时加载，所以无法实现条件加载。也正因为这个，使得静态分析成为可能

ES6 模块与 CommonJS 模块的差异：

- CommonJS模块输出的是一个值的拷贝，ES6 模块输出的是值的引用
  - CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值
  - ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的 import 有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块
- CommonJS模块是运行时加载，ES6 模块是编译时输出接口
  - 运行时加载：CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”
  - 编译时加载：ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import 时采用静态命令的形式。即在 import 时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”。
- CommonJS 模块的 require() 是同步加载模块，ES6 模块的 import 命令是异步加载，有一个独立的模块依赖的解析阶段

异步加载JS方式

- defer：只支持 IE，如果你的脚本不会改变文档的内容，可将 defer 属性加入到 <script> 标签中，以便加快文档的速度，因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显示给用户为止。defer 是在 HTML 解析完之后才会执行， 如果是多个，按照加载的顺序依次

```
// defer属性规定是否对脚本执行进行延迟，直到页面加载为止
<script type="text/javascript" src="xxx.js" defer="defer"></script>
```

- async：HTML5 属性仅适用于外部脚本，并且如果在 IE 中，同时存在defer 和 async，那么 defer 的优先级比较高，脚本将在页面完成时执行。async 是在加载完之后立即执行，如果是多个，执行顺序和加载顺序无关

```
// async属性规定一旦加载脚本可用，则会异步执行
```

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

- 创建 script 标签，插入到 DOM 中

```
(function () {
    var scriptELe = document.createElement('script')
    scriptEle.type = 'text/javascript'
    scriptEle.async = true
    scriptEle.src = 'http://cdn.bootcss.com/jquery/3.0.0-beta1/jquery.min.js'
    var x = document.getElementByTagName('head')[0]
    x.insertBefore(scriptEle, x.firstChild)
})()
```

说一下Set、Map、WeakSet、WeakMap

- Set 对象可以存储任何类型的数据。值是唯一的，没有重复的值
- WeakSet 结构与 Set 类似，也是不重复的值的集合，但是成员只能是对象，而不能是其他类型的值。WeakSet 不可遍历
- Map对象保存键值对，任意值都可以成为它的键或值
- WeakMap 对象与 Map 类似，但是只能接受对象作为键名（null除外），不接受其他类型的值作为键名。WeakMap 的键名所指向的对象，不计入垃圾回收机制

map 和 foreach 有什么区别?

- foreach() 会针对每一个元素执行提供的函数，该方法没有返回值，是否会改变原数组取决于数组元素的类型是基本类型还是引用类型
- 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

JS的节流和防抖是什么？怎么实现？有哪些应用场景?

- 节流：指连续触发事件但是在n秒钟只执行一次函数

- 防抖：指触发事件后在 n 秒内函数只能执行一次

```
<html>
    <head></head>
    <body>
        <input id="input"></input>
        <script>
```

```
            // 防抖函数
            function debounce (func, time) {
                let timer = null
                return function () {
                    clearTimeout(timer)
                    timer = setTimeout(() => {
                        func.apply(this, arguments)

                    }, time)
                }
            }

            // 测试用例
            function ajax (content) {
                console.log('ajax request ' + content + ' time: ' + new Date().getSeconds())
            }
                        let input = document.getElementById('input')

            let debounceAjax = debounce(ajax, 500)

            input.addEventListener('keyup', function (e) {
                debounceAjax(e.target.value)
            })
        </script>
    </body>
</html>
```

JS 的 new 操作符做了哪些事情?

new 操作符新建了一个空对象，这个对象原型指向构造函数的 prototype，执行构造函数后返回这个对象

- 创建一个类的实例：创建一个空对象 obj，然后把这个空对象的 __proto__ 设置为构造函数的 prototype
- 初始化实例：构造函数被传入参数并调用，关键字 this 被设定指向该实例 obj
- 返回实例 obj

```
function _new (constructor, ...args) {
    // 创建一个空对象
```

```
    const obj = {}
    // 空对象的`__proto__`指向构造函数的`prototype`, 为这个新对象添加属性
    obj.__proto__ = constructor.prototype
    // 构造函数的作用域赋给新对象
    const res =  constructor.apply(obj, args)
    // 返回新对象.如果没有显式return语句，则返回this
    return Object.prototype.toString.call(res) === '[objcet Object]' ? res : obj
}
```

不应该使用箭头函数的一些情况：

- 当想要函数被提升时（箭头函数式匿名的）
- 想要在函数中使用 this/arguments 时，由于箭头函数本省不具有 this/arguments，因此它们取决于外部上下文
- 使用命名函数（箭头函数是匿名的）
- 使用函数作为构造函数时（箭头函数没有构造函数）
- 当想在对象字面是以将函数作为属性添加并在其中使用对象时，因为无法访问 this 即对象本身

JS中的垃圾回收机制

由于字符串、对象和数组没有固定大小，所以当他们的大小可知时，才能对他们进行动态的存储分配。JavaScript 程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体，只要像这样动态地分配内存，最终都要释放这些内存以便他们能够再被利用，否则，JavaScript 的解释器将会消耗完系统中所有可用的内存，造成系统崩溃

这段话解释了为什么需要系统来进行垃圾回收，JS 不像 C/C++，他有自己的一套垃圾回收机制（Garbage Collection）。JavaScript 的解释器可以检测到何时程序不再使用一个对象了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如：

```
var a = "hello world"
var b = "world"
var a = b
```

这时，JS会释放掉“hello world”，释放内存以便再引用

在V8引擎优化垃圾回收机制的两种方法：1、分代回收；2、增量GC。目的是通过对象的使用频率、存在市场来区分新生代和老生代对象，多回收新生代区，少回收老生代去，减少每次遍历的时间，从而减少GC的耗时

垃圾回收的方法：标记清除、计数引用

标记清除：

这是最常见的垃圾回收方式，当变量进入环境是，就标记这个变量为“进入环境”，从逻辑上讲，永远不会释放进入环境的变量所占的内存。只要执行流程进入相应的环境，就可能用到他们。当离开环境时，就标记为离开环境

垃圾回收器在运行的时候会给存储在内存中的变量都加上标记（所有都加），然后去掉环境变量中的变量，以及被环境变量中的变量所引用的变量（条件性去除标记）。删除所有被标记的变量后，删除的变量无法在环境变量中被访问。最后垃圾回收器会完成内存的清楚工作并回收他们所占用的内存

引用计数法：

另一种不太常见的方法就是引用计数法，引用计数法的意思就是每个值没引用的次数，当声明一个变量，并用一个引用类型的值赋值给改变量，则这个值的引用次数为0的时候，说明没有办法再访问这个值了，因此就把所占的内存给回收进来，这样垃圾收集器再次运行的时候，就会释放引用次数为0的这些值

用引用计数法会存在内存泄漏，下面来看原因：

```
function problem () {
    const objA = new Object()
    const objB = new Object()
    objA.someOtherObject = objB
    objB.anotherObject = objA
}
```

在这个例子里面，objA 和 objB 通过各自的属性相互引用，这样的话，两个对象的引用次数都为 2，在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用域，函数执行完成之后，因为计数不为 0，这样的相互引用如果大量存在就会导致内存泄露。特别是在 DOM 对象中，也容易存在这种问题:

```
var element = document.getElementById(’‘);
var myObj = new Object();
myObj.element = element;
element.someObject = myObj;
```

这样就不会有垃圾回收的过程

哪些情况会导致内存泄漏?

- 意外的全局变量：由于使用未声明的变量，而意外地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
- 被遗忘的计数器或回调函数：设置了 setInerval 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

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，他有两个特征：

- 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，这就是 promise 名字的由来
- 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 改到了 fulfilled 或者从 pending 改到了 rejected，只要两种情况发生，状态就凝固了，不会再改变，这个时候就成为定型 resolved

```
// ES5
function Promise (fn) {
    let state = 'pending', value = null, callbacks = []

    this.then = function (onFulfilled, onRejected) {
        return new Promise(function (resolve, reject) {
            handle({
                onFulfilled: onFulfilled || null,
                onRejected: onRejected || null,
                resolve: resolve,
                reject: reject
            })
        })
    }

    function handle (callback) {
        if (state === 'pending') {
            callbacks.push(callback)
            return
        }
        const cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected, ret
        if (cb === null) {
            cb = state === 'fulfilled' ? callback.resolve : callback.reject
            cb(value)
            return
```

```
        }
        ret = cb(value)
        callback.resolve(ret)
    }
    function resolve (newValue) {
        if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {
            const then = newValue.then
            if (typeof then === 'function') {
                then.call(newValue, resolve, reject)
                return
            }
        }
        state = 'fulfilled'
        value = newValue
        execute()
    }
    function reject (reason) {
        state = 'rejected'
        value = reason
        execute()
    }
    function execute () {
        setTimeout(function () {
            callbacks.forEach(function (callback) {
                handle(callback)
            })
        }, 0)
    }
    fn (resolve, reject)
}

// ES6
class PromiseM {
    constructor (process) {
        this.status = 'pending'
        this.msg = ''
        this.fulfilled = () => {}
        this.rejected = () => {}
        process(this.resolve.bind(this), this.reject.bind(this))
```

```
        return this
    }
    resolve (val) {
        this.status = 'fulfilled'
        this.msg = val
        this.then.call(this)
    }
    reject (err) {
        this.status = 'rejected'
        this.msg = err
        this.catch.call(this)
    }
    then (fulfilled) {
        if (fulfilled) this.fulfilled = fulfilled
        if (this.status === 'fulfilled') {
            this.fulfilled(this.msg)
        }
        return this
    }
    catch (rejected) {
        if (rejected) this.reject = rejected
        if (this.status === 'rejected') {
            this.reject(this.msg)
        }
        return this
    }
}

// 测试
const mm = () => new PromiseM((resolve, reject) => {
    setTimeout(() => {
        console.log('延迟5秒执行函数')
        reject('执行 reject')
    }, 5000)
})
mm().then(res => {
    console.log('res', res)
}).catch(err => {
    console.log('err', err)
})
```

说一下 slice、splice、split 的区别?

- slice：该方法是对数组进行部分截取，该方法返回一个新数组
- splice：从数组中移除一个或多个元素，并用新的元素替换它们，返回值为被删除的字符串
- split：把字符串分割成片段来创建一个字符串数组，与 Array.join 执行的操作是相反的

说一下怎么把类数组转换成数组?

```
//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)
```

说一下JSON.stringify有什么缺点?

- 如果 obj 里面有时间对象，则 JSON.stringify 后再 JSON.parse 的结果，时间将只是字符串的形式，而不是对象的形式
- 如果 obj 里有 RegExp（正则表达式的缩写）、Error 对象，则序列化的结果将只得到空对象
- 如果 obj 里有函数、undefined，则序列化的结果会把函数或undefined丢失
- 如果 obj 里有 NaN、Infinity 和 -Infinity，则序列化的结果会变成 null
- JSON.stringify() 只能序列化对象的可枚举的自有属性，例如，如果obj中的对象是有构造函数生成的。则使用 JSON.parse(JSON.stringify(obj)) 深拷贝后，会丢弃对象的constructor
- 如果对象中存在循环引用的情况也无法正确实现深拷贝

‘use strict’ 是干嘛用的?

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

严格模式的一些限制：

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

设立“严格模式”的目的，主要有以下几个：

- 消除JavaScript语法的一些不合理、不严谨之处，减少一些怪异行为；
- 消除代码运行的一些不安全之处，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

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

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

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

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

JavaScript 支持闭包和高阶函数是函数式编程语言的特点

什么是高阶函数?

高阶函数只是将函数作为参数或返回值的函数

Object.seal 和 Object.freeze 方法之间有什么区别?

- Object.freeze() 方法可以冻结一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改。freeze() 返回和传入的参数相同的对象
- Object.seal() 方法封闭一个对象，阻止添加新属性并将所有现有属性标记为不可配置。当前属性的值只要可写就可以改变

相同点:

- ES5新增
- 对象不可能扩展，也就是不能再添加新的属性或者方法
- 对象已有属性不允许被删除
- 对象属性特性不可以重新配置

不同点:

- Object.seal 方法生成的密封对象，如果属性是可写的，那么可以修改属性值
- Object.freeze 方法生成的冻结对象，属性都是不可写的，也就是属性值无法更改

'in' 运算符和 'Object.hasOwnProperty' 方法有什么区别?

- hasOwnProperty() 方法返回值是一个布尔值，指示对象自身属性中是否具有指定的属性，因此这个方法会忽略掉那些从原型链上继承到的属性
- 如果指定的属性在指定的对象或其原型链中，则 in 运算符返回 true

模拟实现JS的apply和call方法?

参考 面试官问：能否模拟实现JS的call和apply方法

```
// 浏览器环境 非严格模式
function getGlobalObject(){
    return this;
}
Function.prototype.applyFn = function apply(thisArg, argsArray){ // `apply` 方法的 `length` 属性是 `2`。
    // 1.如果 `IsCallable(func)` 是 `false`, 则抛出一个 `TypeError` 异常。
    if (typeof this !== 'function') {
```

```
        throw new TypeError(this + ' is not a function!')
    }
    // 2.如果 argArray 是 null 或 undefined，则返回提供 thisArg 作为 this 值并以空参数列表调用 func 的 [[Call]] 内部方法的结果。
    if (typeof thisArg === 'undefined' || thisArg === null) {
        argsArray = []
        // 在外面传入的 thisArg 值会修改并成为 this 值。
        // ES3: thisArg 是 undefined 或 null 时它会被替换成全局对象 浏览器里是window
        thisArg = getGlobalObject()
    }
    // 3.如果 Type(argArray) 不是 Object，则抛出一个 TypeError 异常 .
    if (argsArray !== new Object(argsArray)) {
        throw new TypeError('CreateListFromArrayLike called on non-object');
    }
    // ES3: 所有其他值会被应用 ToObject 并将结果作为 this 值，这是第三版引入的更改。
    thisArg = new Object(thisArg)
    var __fn = '__fn'
    thisArg[__fn] = this
    // 9.提供 thisArg 作为 this 值并以 argList 作为参数列表，调用 func 的 [[Call]] 内部方法，返回结果
    var result = thisArg[__fn](...argsArray)
    delete thisArg[__fn]
    return result
}
```

问题一：__fn 同名覆盖问题，thisArg对象上有 __fn，那就被覆盖了然后被删除了。

- 解决方案一：采用 ES6 Sybmol()
- 解决方案二：使用 Math.random() 模拟实现独一无二的 key

ES6

ES6 或 ECMAScript 2015 有哪些新特性?

- 箭头函数
- 类
- 模板字符串
- 加强的对象字面量
- 对象解构：从对象或数组中获取或提取值的一种新的、更简洁的方法

- Promise
- 生成器
- 模块
- Symbol
- 代理
- Set、Map
- 函数默认参数
- rest 和展开
- 块作用域

举例说明 ES6 对 String 字符串、Array 数组、Number数字、Object对象、Function函数类型做的常用升级优化?

参考 ES6面试、复习干货知识点汇总（全）

String:

- 字符串模板
- 增加了includes()、startsWith()、endsWith()、padStart()、padEnd()、repeat()等方法

Array:

- 数组解构赋值
- 扩展运算符
- 增加了 find()、copyWithin()、includes()、fill()、flat() 等方法

Number:

- 新增了 isFinite()、isNaN() 方法
- 在 Math 对象上新增了 Math.cbrt()、trunc()、hypot() 等较多的科学计数法运算方法

Object:

- 可以直接以变量形式声明对象属性或者方法
- 对象的解构赋值
- 对象的扩展运算符
- ES6 新增了 is() 方法，做两个目标对象的相等比较，还新增了 assign()、getOwnPropertyDescriptors()、getPrototypeOf()、setPrototypeOf()、Object.keys()、Object.values()、Object.entries() 方法

Function:

- 箭头函数
- 双冒号运算符，用来取代以往的bind、call和apply