03-JS高级-常见面试题

1.this绑定

1.1 this的绑定规则有几种?

- 默认绑定：独立函数调用，函数没有被绑定到某个对象上进行调用
- 隐式绑定：通过某个对象发起的函数调用，在调用对象内部有一个对函数的引用。
- 显式绑定：明确this指向的对象，第一个参数相同并要求传入一个对象。
  - apply/call
  - bind
- new绑定：
  - 创建一个全新对象
  - 新对象被执行prototype链接
  - 新对象绑定到函数调用的this
  - 如果函数没有返回其他对象，表达式会返回这个对象

1.2 this的面试题解析

```
var name = "window";
function Person(name) {
  this.name = name;
  this.foo1 = function () {
    console.log(this.name);
  };
  this.foo2 = () => console.log(this.name);
  this.foo3 = function () {
    return function () {
      console.log(this.name);
    };
  };
  this.foo4 = function () {
    return () => {
      console.log(this.name);
    };
  };
}
var person1 = new Person("person1");
var person2 = new Person("person2");
// person1.foo1() // person1 隐式调用
// person1.foo1.call(person2) // person2 显示调用 this指向person2所在的对象
```

```
// person1.foo2(); // person1 箭头函数 向上层作用查找 上层作用域中的this为person1指向的对象
// person1.foo2.call(person2); // person1 箭头函数 显示绑定没用

// person1.foo3()() // window 相当于将返回的函数赋值给一个变量 指向该变量 是独立函数调用
// person1.foo3.call(person2)() // window 默认调用
// person1.foo3().call(person2) // person2 将函数的this显示绑定到person2所在的对象

// person1.foo4()() // person1 箭头函数 向上层作用域中查找this foo4中的this隐式绑定为person1
// person1.foo4.call(person2)() // person2  箭头函数 向上层作用域中查找this foo4中的this显示绑定为person2
// person1.foo4().call(person2) // person1 箭头函数显示绑定没用 和person1.foo4()()xiang'tong
```

作用域

什么是变量提升、函数提升?

变量提升：

- 简单说就是在js代码执行前引擎会先进行预编译，预编译期间会将变量声明与函数声明提升至其对应作用域的最顶端，函数内声明的变量只会提升至该函数作用域最顶层。
- 当函数内部定义的一个变量与外部相同时，那么函数体内的这个变量就会被上升到最顶端。

举个例子，如：

```
console.log(a); // undefined
var a = 3; // 会将var a 的声明提升至最顶端
```

函数提升：

- 函数提升只会提升函数声明式写法，函数表达式的写法不存在函数提升。
- 函数提升的优先级大于变量提升的优先级，即函数提升在变量提升之上。

说说你对GO/AO/VO的理解?

GO

- Global Object JS代码在执行前会现在堆内存中创建一个全局对象(GO)
- 用于存放一些定义好的变量方法等包含Date Array String Number setTimeout等
- 同时有一个window属性指向自己
- 同时在语法分析转成AST的过程中也会将一些变量 函数 存放在GO中 只是变量的初始值为undefined

AO

- 函数在执行前会先在堆内存中创建一个AO(Activation Object)对象 里面存放这arguments 对应函数的形参 以及在函数中定义的变量 初始值为undefined

VO

- Variable Object 在执行函数时 会在执行上下文栈(ECS)中进入一个函数执行上下文(FEC)其中有三个核心 核心之一是VO 指向的是该函数在内存中解析时创建的AO 而在全局执行上下文中指向的是GO

说说你对作用域和作用域链的理解?

作用域

- 在ES5中，全局是一个作用域，函数也会产生作用域。
- 在ES6中，代码块、let、const等都会有属于自己的作用域。

作用域链

- 当进入到一个执行上下文时，执行上下文会关联一个作用域链。
- 通常作用域链在解析时就被确定，作用域链与函数的定义位置有关，与它的调用位置无关

你是如何理解闭包的,闭包到底是什么?

- **什么是闭包?**

  一个普通的函数function，如果它可以访问外层作用域的自由变量，那么这个函数和周围环境就是一个闭包。

  从狭义的角度来说：JavaScript中一个函数，如果访问了外层作用域的变量，那么它是一个闭包

- **应用场景**

  防抖 、节流 、立即执行函数 、组合函数等等

数组

常用的数组操作方法有哪些?

Array.shift()

- 删除并返回第一个元素 作用：从数组中删除第一个元素（即下标为0的元素），并返回该元素。 注意：1）删除元素之后，数组的长度-1。

Array.pop()

- 删除并返回最后一个元素 作用：从数组中删除最后一个元素（即下标为length-1的元素），并返回该元素。 注意：1）删除元素之后，数组的长度-1。

Array.push(param1[,param2,...paramN])

- 尾部添加元素 作用：在数组的尾部添加一个元素，并返回新数组的长度。 注意：1）它是直接修改该数组，而不是重新创建一个数组。

Array.unshift(newElement1[,newElement2,...newElementN])

- 头部添加元素 作用：在数组的头部添加一个或多个元素，并返回新数组的长度。 注意：1）它是直接修改该数组，而不是重新创建一个数组。

Array.join([separator])

- 转换成字符串 作用：把数组的所有元素放入到一个字符串中。 注意：1）参数separator表示字符串中元素的分隔符，可以为空，默认为半角逗号。

Array.reverse()

- 反转数组 作用：把数组的所有元素顺序反转。 注意：1）该方法会直接修改数组，而不会创建新的数组。

数组如何进行降维（扁平化）

- 利用Array.some方法判断数组中是否还存在数组，es6展开运算符连接数组

```
let arr = [1,2,[3,4]]
while (arr.some(item => Array.isArray(item))) {
    arr = [].concat(...arr);
}
```

- 使用数组的concat方法

```
let arr = [1,2,[3,4]]
let result = []
result = Array.prototype.concat.apply([], arr)
```

- 使用数组的concat方法和扩展运算符

```
var arr = [1,2,[3,4]]
var result = []
result = [].concat(...arr)
```

- es6中的flat函数也可以实现数组的扁平化

```
let arr = [1,2,['a','b',['中','文',[1,2,3,[11,21,31]]]],3];
let result = arr.flat( Infinity )
```

数组去重，能用几种方法实现?

利用ES6 Set去重（ES6中最常用）

```
function unique (arr) {
  return Array.from(new Set(arr))
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

利用for嵌套for，然后splice去重（ES5中最常用）

```
function unique(arr){
        for(var i=0; i<arr.length; i++){
            for(var j=i+1; j<arr.length; j++){
                if(arr[i]==arr[j]){         //第一个等同于第二个，splice方法删除第二
个
                    arr.splice(j,1);
                    j--;
                }
            }
        }
return arr;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}] //NaN和{}没
有去重，两个null直接消失了
```

利用indexOf去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array .indexOf(arr[i]) === -1) {
            array .push(arr[i])
        }
    }
    return array;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
```

```
// [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {…}, {…}]  //NaN、{}没有去重
```

利用sort去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return;
    }
    arr = arr.sort()
    var arrry= [arr[0]];
    for (var i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i-1]) {
            arrry.push(arr[i]);
        }
    }
    return arrry;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
// [0, 1, 15, "NaN", NaN, NaN, {…}, {…}, "a", false, null, true, "true", undefined] // NaN、{}没有去重
```

利用includes

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array =[];
    for(var i = 0; i < arr.length; i++) {
            if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值
                    array.push(arr[i]);
                }
    }
    return array
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]
// {} 没有去重
```

利用filter

```
function unique(arr) {
  return arr.filter(function(item, index, arr) {
    //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素
    return arr.indexOf(item, 0) === index;
  });
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {…}, {…}]
```

利用递归去重

```
function unique(arr){
    return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],
[]);
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr));
// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]
```

数组中的forEach和map的区别?

`forEach()`和`map()`两个方法都是ECMA5中Array引进的新方法，都是用来遍历数组中的每一项。

它们之间的区别：

- map速度比forEach快
- map会返回一个新数组，不对原数组产生影响,foreach不会产生新数组，forEach返回undefined
- map因为返回数组所以可以链式操作，forEach不能
- map里可以用return（return的是什么，相当于把数组中的这一项变为什么（并不影响原来的数组，只是相当于把原数组克隆一份，把克隆的这一份的数组中的对应项改变了）。
- forEach里用return不起作用，forEach不能用break，会直接报错。

for in和for of的区别?

在JavaScript中遍历数组通常是使用for...i循环，在ES5具有遍历数组功能的还有forEach、map、filter、some、every、reduce、reduceRight等。for...in和for...of是两种增强型循环，for...in是ES5标准，在ES6中新增了for...of的循环方式。

for...in：遍历以任意顺序迭代一个对象的除Symbol以外的可枚举属性，包括继承的可枚举属性。

for...of：遍历在可迭代对象，包括 `Array`，`Map`，`Set`，`String`，`TypedArray`，arguments 对象等等

它们的区别：

- for...in可以遍历对象和数组，for...of不能遍历对象
- for...in 循环不仅遍历对象的键名，还会遍历手动添加的其它键，甚至包括原型链上的键
- for...in遍历的索引为字符串类型
- for..of适用遍历数/数组对象/字符串/map/set等拥有**迭代器对象**的集合，但是不能遍历对象
- for...of与forEach()不同的是，它可以正确响应break、continue和return语句
- 具有迭代器对象才可以使用for...of

2 函数

2.1 说出apply、call、bind函数的用法和区别?

用法：

- apply

  第一个参数: 绑定this

  第二个参数: 传入额外的实参, 以数组的形式

- call

  第一个参数: 绑定this

  参数列表: 后续的参数以多参数的形式传递, 会作为实参

- bind(不希望obj对象身上有函数)

```
var bar = foo.bind(obj)
bar() // this -> obj
```

区别：

- call、apply和bind都可以改变函数的this指向
- call、apply和bind第一个参数的是this要指向的对象
- call、apply和bind都可以后续为函数传参，apply是将参数并成一个数组，call和bind是将参数依次列出
- call、apply都是直接调用，bind生成的this指向改变函数需要手动调用。

什么是纯函数？如何编写纯函数?

纯函数：纯函数一般具有以下的特点：

- 确定的输入一定会有确定的输出（外部环境的任何变化不会影响函数内部的操作产生的结果）
- 纯函数的执行不会产生副作用。（函数内部的操作也不会对函数外部产生任何影响）

纯函数在react和redux中应用比较多。

编写纯函数：

```
//一般的数学方法可以写成纯函数,例如相加
function sum(...args) {
  var result = args.reduce((perValue, item) => {
    return preValue + item
  }, 0)
  return result
}
```

什么是函数柯里化？柯里化有什么作用？

函数的柯里化：

- 将传入多个参数的函数转变成传入单个参数并且返回一个函数用于接收剩余的参数的函数。
- 每一层函数都接收一个参数并对参数进行处理。

柯里化的作用：

- 单一职责：每一个函数只用处理传入的单个参数，每个函数的职责单一而且确定
- 参数复用：可以拿到每一层函数执行的返回值作为一个新的函数，复用已经传入过的参数。

组合函数以及组合函数的作用？

组合函数：

- 组合函数是将多个函数组合到一起，进行依次调用的函数使用模式。

组合函数的作用：

- 减少重复代码的编写，提高代码的复用性，便于开发。
- 可以对任意个函数进行组合，返回新的具有多个被组合函数功能的新函数

说说你对严格模式的理解？

严格模式是一种JavaScript的限制模式，因为种种历史原因，JavaScript语言在非严格模式下是比较松散的。在JavaScript不断优化和加入新特性的过程中，为了兼容早期的JavaScript，一些错误和不规范的写法也被保留了下来。这些错误也不会被抛出。在开启了严格模式后，js引擎会以一种更严格的规范执行JavaScript代码，一些不规范的写法和错误也会直接抛出。

开启严格模式的方法：

- 对文件开启：在文件的开头写上"use strict"
- 对函数开启：在函数的开头写上"use strict"

严格模式下的语法限制：

- 不允许意外创建全局变量（不写var、let、const这种声明变量的关键字）
- 会对静默失败的赋值操作抛出异常
- 试图删除不可删除的属性
- 不允许函数参数有相同的名称
- 不允许只有0开头的八进制语法
- 不允许使用with
- 无法获取eval中定义的变量
- this绑定不会默认转成对象

浏览器和v8引擎

浏览器内核是什么？有哪些常见的浏览器内核?

浏览器内核又称浏览器渲染引擎，是浏览器的最核心部分。负责解析网页语法并渲染网页。

常见的浏览器内核有：

- trident（三叉戟）---- IE浏览器、360安全浏览器、UC浏览器、搜狗高速浏览器、百度浏览器
- gecko（壁虎） ---- Mozilla、Firefox
- pestro -> Blink ---- Opera
- Webkit ---- Safari、360极速浏览器、搜狗高速浏览器、移动端浏览器
- Webkit -> Blink ----Chrome、Edge

说出浏览器输入一个URL到页面显示的过程?

- URL 输入
  - 检查输入的内容是否是一个合法的 URL 链接
  - 判断输入的 URL 是否完整, 如果不完整，浏览器可能会对域进行猜测，补全前缀或者后缀
  - 使用用户设置的默认搜索引擎来进行搜索
- DNS 解析
  - 浏览器不能直接通过域名找到对应的服务器 IP 地址
  - 所以需要进行 DNS 解析，查找到对应的 IP 地址进行访问。
- 建立 TCP 连接
- 发送 HTTP / HTTPS 请求（建立 TLS 连接）
  - 向服务器 发起 TCP 连接请求
  - 当这个请求到达服务端后，通过 TCP 三次握手，建立 TCP 的连接。
    - 1.客户端发送 SYN 包到服务器，并进入 SYN_SEND 状态，等待服务器确认
    - 2.服务器收到 SYN 包，必须确认客户的 SYN，同时自己也发送一个 SYN 包，此时服务器进入 SYN_RECV 状态。
    - 客户端收到服务器的 SYN包，向服务器发送确认包，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。
- 服务器响应请求
  - 当浏览器到 web 服务器的连接建立后，浏览器会发送一个初始的 HTTP GET 请求，请求目标

通常是一个 HTML 文件。服务器收到请求后，将发回一个 HTTP 响应报文，内容包括相关响应头和 HTML 正文。

- 浏览器解析渲染页面
    - 处理 HTML 标记并构建 DOM 树。
    - 处理 CSS 标记并构建 CSSOM 树。
    - 将 DOM 与 CSSOM 合并成一个渲染树
    - 根据渲染树来布局，以计算每个节点的几何信息。
    - 将各个节点绘制到屏幕上。
- HTTP 请求结束，断开 TCP 连接.

说说你对 JS 引擎的理解

- JavaSript 是一门解释型语言
- JS 引擎是 JavaScript 语言的运行解释器
    - 浏览器内核中有两种引擎，其中一种就是 JS 引擎
        - **排版引擎**
            - 负责 HTML 和 CSS 解析和排版
        - **JS 引擎**
            - 负责解析和运行 JavaScript 语句
- 常见 JS 引擎有
    - SpiderMonkey -> 第一款 JavaScript 引擎，Brendan Eich 开发
    - Chakra -> 微软开发
    - WebKit -> JavaScriptCore -> APPLE 开发
    - Chrome -> V8 -> GOOGLE 开发
    - 小程序 -> JSCore -> 腾讯开发

说说V8引擎的内存管理

- JavaScript的内存管理是自动的
- 关于原始数据类型 直接在栈内存中分配
- 关于复杂数据类型 在堆内存中分配

说说V8引擎的垃圾回收器

- 因为内存大小是有限的 所以在内存不需要的时候 需要进行释放 用于腾出空间
- GC对于内存管理有着对应的算法
- 常见的算法
    - 引用计数(Reference Count)
        - 当一个对象有引用指向它时 对应的引用计数+1
        - 当没有对象指向它时 则为0 此时进行回收
        - 但是有一个严重的问题 - 会产生循环引用
    - 标记清除(Mark-Sweep)
        - 核心思路: 可达性
        - 有一个根对象 从该对象出发 开始引用到所用到的对象 对于根对象没有引用到的对象 认

为是不可用的对象
        - 对于不可用的对象 则进行回收
        - 该算法有效的解决了循环引用的问题
        - 目前V8引擎采用的就是该算法
- V8引擎为了优化 在采用标记清除的过程中也引用了其他的算法
    - 标记整理
        - 和标记清除相似 不同的是回收时 会将保留下来的存储对象整合到连续的内存空间 避免内存碎片化
    - 分代收集(Generational Collection)
        - 将内存中的对象分为两组 新的空间 旧的空间
        - 对于长期存活的对象 会将该对象从新空间移到旧空间中 同时GC检查次数减少
        - 将新空间分为from和to 对象的GC查找之后从from移动到to空间中 然后to变为from from变为to 循环几次 对于依然存在的对象 移动到旧空间中
    - 增量收集(Increment Collection)
        - 如果存在许多对象 则GC试图一次性遍历所有的对象 可能会对性能造成一定的影响
        - 所以引擎试图将垃圾收集工作分成几部分 然后这几部分逐一处理 这样会造成微小的延迟而不是很大的延迟
    - 闲时收集(Idle-time Collection)
        - GC只会在CPU空闲的时候运行 减少可能对代码执行造成的影响

v8引擎执行代码的大致流程

- Parse模块：将JavaScript代码转成AST Tree
- Ignition :解释器 将ASTTree 转换为字节码(byte Code)
    - 同时收集TurboFan 优化需要的信息
- TurboFan :编译器 将字节码编译为CPU可以直接执行的机器码(machine code)
    - 如果某一个函数呗被多次调用 则会被标记为热点函数 会经过TurBoFan转换的优化的机器码让CPU执行 提高代码性能
    - 如果后续执行代码过程中 改函数调用时的参数类型发生了改变 则会逆向的转成字节码 让CPU执行

v8引擎执行流程:

- 首先会编译JavaScript 编译过程分为三步
- 1 词法分析(scanner)
    - 会将对应的每一行的代码的字节流分解成有意义的代码块 代码块被称为词法单元(token 进行记号化)
- 2 语法分析(parser)
    - 将对应的tokens分析成一个元素逐级嵌套的树 这个树称之为 抽象语法树(Abstract Syntax Tree AST)
    - 这里也有对应的 pre-parser
- 3 将AST 通过Ignition解释器转换成对应的字节码(ByteCode) 交给CPU执行 同时收集信息
    - 将可优化的信息 通过TurBoFan编译器 编译成更好使用的机器码交给CPU执行

- 如果后续代码的参数类型发生改变 则会逆优化(Deoptimization)为字节码

说说线程和进程的区别以及关系

- **进程**
  - 是`cpu`分配资源的最小单位；（是能拥有资源和独立运行的最小单位）
  - 计算机已经运行的程序，是操作系统管理程序的一种方式 (**官方说法**)
  - 可以认为启动一个应用程序，就会默认启动一个进程（也可能是多个进程）**(个人解释)**
  - 也可以说进程是线程的容器
- **线程**
  - 是`cpu`调度的最小单位；（线程是建立在进程的基础上的一次程序运行单位，一个进程中可以有多个线程）
  - 操作系统能够运行运算调度的最小单位，通常情况下它被包含在进程中 **(官方说法)**
  - 每一个进程中，都会启动至少一个线程用来执行程序中的代码，这个线程被称之为主线程
- **操作系统的工作方式**
  - 如何做到同时让多个进程同时工作?
    - 因为`CPU`的运算速度非常快, 可以快速的在多个进程之间迅速的切换
    - 当进程中的线程获取到世间片时, 就可以快速执行我们编写的代码
    - 由于`CPU`执行速度过于变态, 对于用户来说是感受不到这种快速切换的

`JavaScript`为什么是单线程?

- 这主要和js的用途有关，js是作为浏览器的脚本语言，主要是实现用户与浏览器的交互，以及操作dom
- 这决定了它只能是单线程，否则会带来很复杂的同步问题。
- 比如js被设计了多线程，如果有一个线程要修改一个dom元素，另一个线程要删除这个dom元素，此时浏览器就会一脸茫然，不知所措。
- 所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变

浏览器是多进程的?

- 在浏览器中，每打开一个tab页面，其实就是新开了一个进程，在这个进程中，还有ui渲染线程，js引擎线程，http请求线程等。
- 因此浏览器是一个多进程的。为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

什么是重排重绘，如何减少重排重绘

重排(Reflow)：

- 元素的位置发生变动时发生重排，也叫回流。此时在关键渲染路径中的 Layout 阶段，计算每一个元素在设备视口内的确切位置和大小。当一个元素位置发生变化时，其父元素及其后边的元素位置都可能发生变化，代价极高。

重绘(Repaint):

- 元素的样式发生变动，但是位置没有改变。此时在关键渲染路径中的 Paint 阶段，将渲染树中的每个节点转换成屏幕上的实际像素，这一步通常称为绘制或栅格化

另外，重排必定会造成重绘。以下是避免过多重排重绘的方法：

1. 使用 `DocumentFragment` 进行 DOM 操作，不过现在原生操作很少也基本上用不到
2. CSS 样式尽量批量修改
3. 避免使用 table 布局
4. 为元素提前设置好高宽，不因多次渲染改变位置

面向对象

什么是原型、原型链?

原型：

在JavaScript中，每一个对象都会有一个属性[[prototype]]，这个属性就是对象的原型，这个属性的值也是一个对象，是原对象的原型对象。访问对象中属性时，会先在对象自身进行查找，如果没有找到，那么会去对象的原型对象上查找。

原型链：

每个对象都有自己的原型对象，原型对象也有自己的原型对象。在访问对象的属性时，会沿着对象自身=>自身的原型对象=>原型对象的原型对象.....这样的链条一路查找上去，这条链式结构就叫做原型链。原型链的尽头是Object的原型对象的[[prototype]]属性，值为null。

如何通过原型链实现继承?

原型链继承：重写子类的显式原型对象，让子类的显式原型对象的隐式原型指向父类的显式原型对象。

```
function createObject(o) {
  function F() {}
  F.prototype = o
  return new F()
}
function inherit(Subtype, Supertype) {
  Subtype.prototype = createObject(Supertype.prototype)
  Object.defineProperty(Subtype.prototype, "constructor", {
    enumerable: false,
    configurable: true,
    writable: true,
    value: Subtype
  })
```

```
}
function Person() {}
function Student() {
  Person.call(this)
}
inherit(Student, Person)
```

继承的各个方案以及优缺点?

方案一：直接将父类的prototype赋值给子类的prototype，父类和子类共享原型对象

缺点：在子类原型对象上添加方法和属性会影响到父类

```
function Person() {}
function Student() {}
Student.prototype = Pesrson.prototype
```

方案二：通过new操作符创建一个新的对象，将这个对象作为子类的原型对象(显式原型)

缺点：

- 子类的实例对象继承过来的属性是在原型上的，无法打印
- 没有完美的实现属性的继承（子类的实对象可以从父类继承属性，也可以拥有自己的属性）

```
function Person() {}
function Student() {}
var p = new Person()
Student.prototype = p
```

方案三：通过new操作符创建一个新的对象，将这个对象作为子类的原型对象(显式原型)，并且在子类的内部通过借用构造函数的方法实现属性的继承

缺点：父类构造函数会被调用两次，并且子类的实例对象总是有两份相同的属性，一份在自身，一份在其原型对象上

```
function Person(arg1, arg2) {}
function Student() {
  Person.call(this, arg1, arg2)
}
var p = new Person()
Student.prototype = p
```

方案四：让子类的原型对象(显式原型)的原型对象(隐式原型)指向父类的原型对象(显式原型)

缺点：存在兼容性问题,`__proto__`属性只有部分游览器支持

```
function Person() {}
function Student() {}
Student.prototype.__proto__ = Person.prototype
```

方案五：寄生组合式继承(ES5中实现继承的最终方案)

```
function createObject(o) {
  function F() {}
  F.prototype = o
  return new F()
}
function inherit(Subtype, Supertype) {
  Subtype.prototype = createObject(Supertype.prototype)
  Object.defineProperty(Subtype.prototype, "constructor", {
    enumerable: false,
    configurable: true,
    writable: true,
    value: Subtype
  })
}
function Person() {}
function Student() {
  Person.call(this)
}
inherit(Student, Person)
```

说说你对面向对象多态的理解

- 当对不同的数据类型执行同一个操作时, 如果表现出来的行为(形态)不一样, 那么就是多态的体现
- 继承也是多态的前提

ES6-13

说说let、const和var的区别?

- 作用域提升
  - var声明的变量是会进行作用域提升
  - let、const没有进行作用域提升，但是会在解析阶段被创建出来
  - let,const具有暂时性死区
- 块级作用域
  - var不存在块级作用域
  - let和const存在块级作用域
- 重复声明

    - var允许重复声明变量
    - let和const在同一作用域不允许重复声明变量
- 修改声明的变量
    - let,var 可以修改声明的变量
    - const它表示保存的数据一旦被赋值，就不能被修改,但是如果赋值的是引用类型，那么可以通过引用找到对应的对象，修改对象的内容

说说ES6~ES13新增了哪些知识点?

ES6 :

- 使用class用来定义类
    - constructor构造器
    - extends实现继承
    - super关键字代表继承的父类
- 对象字面量的增强
    - 属性的简写
    - 方法的简写
    - 计算属性名
- 解构
- let/const的使用
    - 不能重复声明变量
    - 不存在作用域提升
    - 存在暂时性死区
    - 不添加window
    - 存在块级作用域
- 字符串模板
    - 在模板字符串中，我们可以通过 ${expression} 来嵌入动态的内容
    - 标签模板字符串
- 函数的默认参数
- 函数的剩余参数
- 箭头函数
    - 没有显式原型prototype
    - 不绑定this、arguments、super参数
- 展开语法
    - 在函数调用时使用；
    - 在数组构造时使用；
    - 展开运算符其实是一种浅拷贝
    - 在构建对象字面量时，也可以使用展开运算符，这个是在ES2018（ES9）中添加的新特性；
- 规范了二进制和八进制的写法
- 新增Symbol
- Set、WeakSet、Map、WeakMap

ES7 :

- Array Includes
  - 通过includes来判断一个数组中是否包含一个指定的元素，根据情况，包含返回 true，否则返回false。
- 指数exponentiation运算符
  - **对数字来计算乘方。

ES8 :

- Object values
  - 通过Object.values 来获取所有的value值
- Object entries
  - 通过 Object.entries 可以获取到一个数组，数组中会存放可枚举属性的键值对数组
- String Padding
  - padStart 和 padEnd 方法，分别对字符串的首尾进行填充的。
- Trailing Commas
  - 允许在函数定义和调用时多加一个逗号：
- Object.getOwnPropertyDescriptors

ES9 :

- 构建对象字面量时，可以使用展开运算符

ES10 :

- flat
  - flat() 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回。
- flatMap
  - flatMap是先进行map操作，再做flat的操作
  - flatMap中的flat相当于深度为1
- Object fromEntries
  - Object.formEntries将entries转换成一个对象
- trimStart trimEnd
  - 去除字符串前面或者后面的空格

ES11 :

- BigInt
  - BigInt，用于表示大的整数(超过最大安全整数)
  - BitInt的表示方法是在数值的后面加上n
- 空值合并操作符
  - ??当前面的值为null或者undefined是,显式??后面的值
- Optional Chaining
  - 可选链?.
  - 当?.前面的值为空时返回undefined
- Global This

    - JavaScript环境的全局对象
- for..in标准化
    - for...in遍历对象时遍历的是key

说说Set、WeakSet、Map、WeakMap的特点?

Set:

- 用来存储数据,类似于数组,
- 与数组的区别是元素不能重复,
- 可以使用forEach方法和使用for...of...遍历
- 常见属性和方法
    - size：返回Set中元素的个数
    - add(value)：添加某个元素，返回Set对象本身
    - delete(value)：从set中删除和这个值相等的元素，返回boolean类型
    - has(value)：判断set中是否存在某个元素，返回boolean类型
    - clear()：清空set中所有的元素
    - forEach(callback, [, thisArg])：通过forEach遍历set

WeakSet:

- 只能存储对象类型,不能存放基本数据类型,
- 对对象的引用是一个弱引用,如果没有其他对对象的引用,那么相应对象会被GC进行清除,
- 不能遍历
- 常见的方法
    - add(value)：添加某个元素，返回WeakSet对象本身
    - delete(value)：从WeakSet中删除和这个值相等的元素，返回boolean类型
    - has(value)：判断WeakSet中是否存在某个元素，返回boolean类型

Map:

- 用于存储映射关系,存储的为键值对,
- 每个键值对为一个数组,
- 与对象的区别是存储的key可以为一个对象
- 可以使用forEach方法和使用for...of...遍历
- 常见属性和方法
    - size：返回Set中元素的个数
    - set(key, value)：在Map中添加key、value，并且返回整个Map对象
    - get(key)：根据key获取Map中的value
    - has(key)：判断是否包括某一个key，返回Boolean类型
    - delete(key)：根据key删除一个键值对，返回Boolean类型
    - clear()：清空所有的元素
    - forEach(callback, [, thisArg])：通过forEach遍历Map

WeakMap:

- 存储的key只能为对象,不允许是其他类型
- 对对象的引用是一个弱引用,如果没有其他对对象的引用,那么相应对象会被GC进行清除,
- 不能进行遍历
- 常见的方法
    - set(key, value)：在Map中添加key、value，并且返回整个Map对象
    - get(key)：根据key获取Map中的value
    - has(key)：判断是否包括某一个key，返回Boolean类型
    - delete(key)：根据key删除一个键值对，返回Boolean类型

异步

说说Promise的作用和使用方法（各个回调的作用）

- **Promise的作用**
    - 是异步编程的一种解决方案，比传统回调函数解决方案更加合理和更强大。
    - **Promise** 对象用于表示一个异步操作的最终完成（或失败）及其结果值。
    - 能够把异步操作最终的成功返回值或者失败原因和相应的处理程序关联起来，这样使得异步方法可以像同步方法那样返回值。
    - **Promise**对象的状态不受外界影响，一旦状态改变,就不会再变
- **Promise的使用方法**

```
const promise = new Promise((resolve,reject) => {
  resolve(value) //该函数执行时会回调onFulfilled
  // reject(reason) //该函数执行时会回调onRejected
  console.log("这个回调函数会被立即执行~")
})

// 监听promise对象的状态 方式一
promise.then(onFulfilled).catch(onRejected)
// 监听promise对象的状态 方式二
promise.then(onFulfilled,onRejected)
```

Promise的实例方法和类方法

- Promise的实例方法:
    - then(onFulfilled,onRejected)
        - onFulfilled ---->成功时的回调
        - onRejected ----> 失败时的回调
        - 返回值是一个新的promise对象 所以promise支持链式调用的原因
    - catch(onRejected)
        - onRejected ---->失败时的回调

    - finally(callback)
        - callback ---->不管promise最后的状态,在执行完then或catch指定的回调函数后,都会执行的回调
- Promise的类方法
    - all()
        - 接受一个数组作为参数,数组元素是promise对象,返回一个新的promise对象
        - 可以不是数组,但必须是可迭代对象,且返回的每一个成员都是Promise实例
        - 只有数组里所以的promise对象都是fulfilled状态时,返回的promsie的状态是fulfilled
        - 当数组中的promise对象有一个的rejected状态时,返回的promise的状态时rejected
    - race()
        - 接受一个数组作为参数,数组元素是promise对象,返回一个新的promise对象
        - 只要数组中的实例有一个率先改变,返回的promise对象就跟着改变
    - allSettled()
        - 接受一个数组作为参数,数组元素是promise对象,返回一个新的promise对象
        - 只有等数组中所有的promise对象都发生状态改变后,返回的promsie对象状态才会改变
        - 返回的promsie对象,一旦状态发生改变,状态总是fulfilled
    - any()
        - 接受一个数组作为参数,数组元素是promise对象,返回一个新的promise对象
        - 只要数组实例中有一个变成fulfilled状态,返回的promise对象就会变成fulfilled状态
        - 只有当数组中所有的promise实例都变成rejected状态,返回的promise 对象才变成rejected状态
    - resolve()
        - 将现有对象转为promsie实例
    - rejected()
        - 返回一个新的promsie实例,该实例的状态未为rejected

说说什么是异步函数？与普通函数有什么区别?

- 使用`async`关键字声明的函数是异步函数

```
// async function foo() {}

// const bar = async function() {}

// const baz = async () => {}

// class Person {
//   async running() {}
// }
```

- 异步函数的执行流程
    - 异步函数的内部代码执行过程和普通的函数是一致的，默认情况下也是会被同步执行

- 返回值和普通函数的区别
  - 情况一：异步函数也可以有返回值，但是异步函数的返回值相当于被包裹到`Promise.resolve`中
  - 情况二：如果我们的异步函数的返回值是`Promise`，状态由会由`Promise`决定；
  - 情况三：如果我们的异步函数的返回值是一个对象并且实现了`thenable`，那么会由对象的方`then`法来决定
- 如果在`async`函数中抛出异常
  - 并不会报错, 而是作为`Promise`的`reject`来传递

Proxy

说出Proxy和Object.defineProperty的区别?

- Proxy的设计初衷就是监听对象的改变，并且提供了13中方法监听对象的操作，大大方便和丰富了对对象的监听操作
  - 拦截和监视外部对对象的访问
  - 可以直接监听数组的变化
- Object.defineProperty
  - 该属性设计初衷是定义对象的属性,所以有些监听操作是监听不到的
  - 对于复杂的对象,层级很深的话,需要深度监听
  - 删除属性,添加属性是不能被监听的
  - 不能监听数组的变化
    - 本质上是数组的length属性的数据属性描述符:
    - configurable: false 意味着length属性不能被修改,不能将length属性修改为存取属性描述符
    - 所以数组长度的变化的不能被监听的

```
const num = [1,2,3]
console.log(Object.getOwnPropertyDescriptors(num))
{
  '0': { value: 1, writable: true, enumerable: true, configurable: true
},
  '1': { value: 2, writable: true, enumerable: true, configurable: true
},
  '2': { value: 3, writable: true, enumerable: true, configurable: true
},
  length: { value: 3, writable: true, enumerable: false, configurable:
false }
}
```

说说什么是Reflect和为什么需要使用它

什么是Reflect:

- Reflect是一个对象，提供了多种方法方便我们统一管理对象。
- **Reflect** 是一个内置的对象，它提供拦截 JavaScript 操作的方法。这些方法与proxy handlers (en-US)的方法相同。
- 与大多数全局对象不同`Reflect`并非一个构造函数，所以不能通过new 运算符对其进行调用，或者将`Reflect`对象作为一个函数来调用。
- `Reflect`的所有属性和方法都是静态的（就像`Math`对象）。

为什么需要使用Reflect:

- 在对对象进行操作时有些方法会有返回值，操作对象变的更加规范。
- Object作为构造函数，操作对象的方法放在它身上不是很合适,早期的设计不规范导致的。
- 在使用Proxy监听对象时，用Reflect来操作对象避免了对原对象的直接操作。

迭代器和生成器

什么是迭代器?

- 迭代器是帮助我们对某个数据结构进行遍历的对象。
- 迭代器也是一个具体的对象，这个对象需要符合迭代器协议（iterable protocol）。
  - 迭代器协议定义了产生一系列值（无论是有限还是无限个）的标准方式
  - 在`JavaScript`中这个标准就是一个特定的`next`方法
- `next`方法的要求
  - 一个无参数或者一个参数的函数，返回一个应当拥有以下两个属性的对象：
  - `done`（`boolean`）
    - 如果迭代器可以产生序列中的下一个值，则为`false`。（这等价于没有指定`done`这个属性。）
    - 如果迭代器已将序列迭代完毕，则为`true`。这种情况下，`value`是可选的，如果它依然存在，即为迭代结束之后的默认返回值。
  - `value`
    - 迭代器返回的任何`JavaScript`值。`done`为`true`时可省略

```
// 封装一个为数组创建迭代器的函数
function createArrayIterator(arr) {
    let index = 0
    return {
        next: function() {
            if (index < arr.length) {
                return { done: false, value: arr[index++] }
            } else {
                return { done: true }
            }
        }
    }
}
```

什么是可迭代对象?

- 和迭代器不是一个概念
  - 当一个对象实现了`iterable protocol`协议时，它就是一个可迭代对象；
  - 这个对象的要求是必须实现`@@iterator`方法，在代码中我们使用`Symbol.iterator`访问该属性
- 可迭代对象的好处
  - 当一个对象变成一个可迭代对象的时候，就可以进行某些迭代操作
  - 比如`for...of`操作时，其实就会调用它的`@@iterator`方法
- 实现可迭代协议的原生对象
  - `String`、`Array`、`Map`、`Set`、`arguments`对象、`NodeList`集合...
- 可迭代对象的应用
  - JavaScript中语法：`for ...of`、展开语法（`spread syntax`）、`yield*`、解构赋值（`Destructuring_assignment`）
  - 创建一些对象时：`new Map([Iterable])`、`new WeakMap([iterable])`、`new Set([iterable])`、`new WeakSet([iterable])`
  - 一些方法的调用：`Promise.all(iterable)`、`Promise.race(iterable)`、`Array.from(iterable)`
- 迭代器的中断
  - 比如遍历的过程中通过`break`、`return`、`throw`中断了循环操作
  - 比如在解构的时候，没有解构所有的值
- 自定义类的迭代实现

```
class Person {
      constructor(name, age, height, friends) {
        this.name = name
                ...
      }
      // 实例方法
```

```
      running() {}
/      [Symbol.iterator]() {
        let index = 0
        const iterator = {
          next: () => {
            if (index < this.friends.length) {
              return { done: false, value: this.friends[index++] }
            } else {
              return { done: true }
            }
          }
        }
        return iterator
      }
    }
```

什么是生成器？生成器和迭代器有什么关系？

生成器是ES6中新增的一种函数控制、使用的方案，它可以让我们更加灵活的控制函数什么时候继续执行、暂停执行等。

- 生成器函数也是一个函数，但是和普通的函数有一些区别
  - 首先，生成器函数需要在function的后面加一个符号：*
  - 其次，生成器函数可以通过yield关键字来控制函数的执行流程：
  - 最后，生成器函数的返回值是一个（生成器）
- 生成器函数

```
生成器函数:
1.function后面会跟上符号: *
2.代码的执行可以被yield控制
3.生成器函数默认在执行时，返回一个生成器对象
    * 要想执行函数内部的代码，需要生成器对象，调用它的next操作
    * 当遇到yield时，就会中断执行
```

- 生成器事实上是一种特殊的迭代器

事件循环

说说你对事件队列、微任务、宏任务的理解

- **事件队列**
  - 事件队列是一种数据结构，可以存放要执行的任务。它符合**队列**“**先进先出**”的特点
- **宏/微任务**
  - 首先它们都是异步任务
  - 宏任务列队

        - 用来保存待执行的宏任务（回调）
        - 如: `pajax、setTimeout、setInterval、DOM`监听、`UI Rendering`等 会被加入到宏列队
    - 微任务列队
        - 用来保存待执行的微任务（回调）
        - 如: `Promise`的`then`回调、 `Mutation Observer API、queueMicrotask()`等 会被加入到微列队
    - `JS`执行时会区别这两个队列
        - `JS`引擎首先必须先执行所有的初始化同步任务代码
        - 每次准备取出第一个宏任务执行前，都要将所有的微任务一个一个取出来执行，
        - 也就是优先级比宏任务高，且与微任务所处的代码位置无关
        - 也就是宏任务执行之前，必须保证微任务队列是空的；
        - 如果不为空，那么就优先执行微任务队列中的任务（回调）

网络HTTP

说说你对HTTP协议的理解

http

- HyperText Transfer Protocol 超文本传输协议
- 超文本传输协议是一种用于分布式协作式的应用层协议
- 定义了客户端和服务器之间交换报文的格式和方式 默认为80端口
- 使用tcp作为传输层协议 保证了数据的可靠性

组成:`一个HTTP请求主要包括: 请求和响应`

- 请求
    - 主要包含: 请求行 请求头 请求体
    - 请求行:
        - 请求方法字段
        - URL字段
        - HTTP协议版本字段
        - ```
          GET/index.html HTTP/1.1
          ```
    - 请求头:
        - 键值对组成
        - User-Agent: 对应展示的浏览器的类型
        - Content-type: 对应的请求内容的数据类型
            - application/x-www-form-urlencoded 数据以&分割 的键值对 键值对用=分割
          ```

        - application/json json类型
        - application/xml xml类型
        - text/plain 文本类型
        - multipart/form-data 表示上传文件
      - keep-alive
    - 请求体: get/post所带的内容
- 响应
  - 响应行
    - 由协议版本 状态码 状态码的原因短语组成
    - `HTTP/1.1 200 OK`
  - 响应头
  - 响应体

请求方法

- get 向服务器获取数据
- post 将响应实体交给指定的资源
- head 请求一个与get请求响应相同的响应 没有实体
- put 上传文件 用于替换目标资源的所有
- patch 用于对资源的部分修改
- delete 删除指定的资源
- connect: 建立一个到目标资源标识的服务器的隧道 通常用于代理服务器
- track: 回显服务器收到的请求 主要用于测试和诊断

响应状态码

- 200 表示请求被服务器端正常处理
- 201 post请求 创建新的资源
- 301 永久重定向 表示资源被分配了新的URI 并返回该URI
- 4xx 表示客户端发生错误
  - 400 请求报文存在语法错误
  - 401 未授权的错误 必须携带身份信息
  - 403 没有权限访问
  - 404 服务器找不到请求资源
- 5xx 服务器错误
  - 500
  - 503 服务器不可用 处于维护或重载状态

说说XMLHttpRequest和Fetch请求的异同

Fecth提供了一种更加现代的处理方案

- 比如返回一个值是`Promise`

    - 在请求成功时调用resolve回调
- 与XMLHttpRequest不同 不用把所有操作放在同一个对象上
- 语法简单 更加语义化
- 基于标准的promise实现 支持async/await
- 更加底层

Fetch缺点

- 不支持abort(超时取消请求) 不支持超时控制
- 没有办法检测请求进度 XHR可以
- 默认不会携带cookie

ajax缺点

- 使用起来比较繁琐

如何取消请求的发送

- xhr
    - 如果使用`XMLHttpRequest`发送请求可以使用`XMLHttpRequest.abort()`
- fetch
    - 如果使用`fetch`发送请求可以使用`AbortController`

```
const controller = new AbortController();
const signal = controller.signal;
fetch('https://somewhere', { signal })
controller.abort()
```

- axios
    - 如果使用`axios`，取消原理同 fetch

```
var CancelToken = axios.CancelToken;
var source = CancelToken.source();

axios.get('/https://somewhere', {
  cancelToken: source.token
}
source.cancel()
```

什么是跨域？为什么有跨域问题？

什么是跨域：

> 跨域是指跨域名的访问，以下情况都属于跨域：

跨域原因说明	示例
域名不同	www.jd.com 与 www.taobao.com
域名相同，端口不同	www.jd.com:8080 与 www.jd.com:8081
二级域名不同	item.jd.com 与 miaosha.jd.com

> 如果**域名和端口都相同，但是请求路径不同**，不属于跨域。

为什么有跨域问题：

跨域问题是浏览器对于ajax请求的一种安全限制：**一个页面发起的ajax请求，只能是于当前页同域名的路径**，这能有效的阻止跨站攻击。

因此：**跨域问题 是针对ajax的一种限制**。

但是这却给我们的开发带来了不变，而且在实际生成环境中，肯定会有很多台服务器之间交互，地址和端口都可能不同，怎么办？这时我们就要解决跨域问题了

解决跨域问题

目前线上常用的跨域解决方案有2种：

- Nginx反向代理
  - 利用nginx反向代理把跨域为不跨域，支持各种请求方式
  - 缺点：需要在nginx进行额外配置
- CORS规范化的跨域请求解决方案，安全可靠。
  - 优势： 在服务端进行控制是否允许跨域，可自定义规则，支持各种请求方式
  - 缺点：CORS需要IE10+

什么是REST，用起来有什么好处

REST是一种软件架构模式（即API接口的设计模式），最常用的数据格式是JSON。

由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

通过REST模式设计的API可以把webapp全部功能进行封装，可以很容易的实现前后端分离，使的前端代码易编写，后端代码易测试。

其它

说说async和defer的使用以及区别?

浏览器在解析构建DOM树的过程中 如果遇到script元素会停止构建DOM树 先下载JavaScript代码 执行对应的脚本

但是某些JavaScript代码中可能存在对某个节点的操作 如果等待DOM树构建完成 之后在进行对应的操作 则会造成大量的回流和重绘

同时在如果JavaScript 代码过多 则浏览器处理的时间会过长 则会造成页面的阻塞

为了解决这个问题 出现了两个属性 async defer

defer

- 脚本的下载会与DOM树的构建同时进行
- 如果脚本提前下载好了 则会等到DOM树构建完成之后 在DOMContentLoaded事件之前执行defer中的代码
- 同时多个defer属性的script标签 则会按照顺序执行
- 推荐放到head标签中 可以早解析
- 对于script默认的内容 会忽略

async

- 脚本的下载会与DOM树的构建同时进行
- 让一个脚本完全独立 脚本的解析 运行于DOM的构建无关
- 多个async属性的脚本不保证运行顺序
- 通常用于独立的脚本 对于其他脚本 DOM没有依赖

LocalStorage和SessionStorage的区别

- LocalStorage提供一种永久性存储的方法 在网页关闭打开时 依然保留
- SessionStorage: 会话存储 再关闭该网页时 存储的内容被清除
- 区别:
    - localStorage永久性存储 SessionStorage在关闭当前页面时存储的内容就会失效
    - SessionStorage只能被同一个窗口的同源页面共享 localStorage除非手动删除 否则一直存在

说说new操作背后的原理?

- 在内存中创建一个空对象 。比如 var moni={}
- 将构造函数的显示原型赋值给这个对象的隐式原型 。moni.**proto**=Person.prototype
- this指向创建出来的新对象。this=moni
- 执行函数体代码
- 如果构造函数没有返回非空对象，那自动返回创建出来的新对象 return moni

说说你对防抖、节流的理解，以及它们的区别和应用场景?

防抖: 将多次执行函数变成最后一次执行 等待固定时间还没有事件触发时执行的函数

- 应用场景

    - 按钮的点击
    - 屏幕滚动时的复杂计算
    - 输入框输入时进行搜索
    - 用户缩放浏览器的resize事件
- 简单的防抖函数实现

```
function myDebounce(execFn, delay) {
     let timer = 0

     function _debounce(...args) {
       if (timer) clearTimeout(timer)
       timer = setTimeout(() => {
         execFn.apply(this, args)
         timer = null
       }, delay)
     }

     return _debounce
}
```

节流: 按照固定的时间频率(间隔)来执行对应的函数

- 应用场景:
    - 监听页面的滚动事件 通过节流来降低事件调用的频率
    - 鼠标移动
    - 用户频繁点击按钮的操作
- 简单实现

```
function myThrottle(execFn, interval) {
    let initTime = 0

    function throttle(...args) {
      let nowTime = Date.now()
      const waitTime = interval - (nowTime - initTime)
      if (waitTime <= 0) {
        execFn.apply(this, args)
        initTime = nowTime
      }
    }
    return throttle
  }
```

什么是值传递，什么引用传递?

- 值类型(原始类型):

    - 在变量中保存的是值本身,占据的空间是在栈内存中分配的
- 引用类型(对象类型):
    - 在变量中保存的是对象的“引用”,占据的空间是在堆内存中分配的
- 值传递: 是将值类型传递给函数参数，函数内部对参数的改变不会影响函数外部的变量， 例如:

```
function foo(a) {
 a = 200
}
var num = 100
foo(num)
console.log(num) // 100
```

- 引用传递: 是将引用类型传递给函数参数,函数参数保存的是对象的"引用"，在函数内部修改对象的属性，函数外部对象的属性也会跟着修改，例如:

```
function foo(a) {
 a.name = "HTML"
}
var obj = {
 name: "JavaScript"
}
foo(obj)
console.log(obj.name) // HTML
```

说说对象的引用赋值、浅拷贝、深拷贝的区别

对象的引用赋值

- 把源对象指向自身所在堆内存空间的指针给了新对象，两个对象所指向的内存空间是一样的，修改其中一个的值，另一个也会发生改变

对象的浅拷贝

- 可以通过{...obj}的方式进行对象的浅拷贝 (Object.assign({},obj))
- 对于obj中的值是原始数据类型的 将对应的值赋值给了newObj中对应的属性
- 对于obj中是复杂数据类型的值，把对应在内存中的指针赋值给了newObj中对应的key 对于复杂数据类型的value修改其中一个另一个也发生改变

对象的深拷贝

- newObj与obj中的属性值一样，但是是一个全新的对象，与原对象没有任何关系
- 默认情况下 js没有提供对应的深拷贝的方式 因为深拷贝是非常消耗内存的。
- 有对应的库实现了深拷贝 （lodash...） 。
- 实现深拷贝
    - JSON.parse(JSON.stringfy(obj))
        - 缺点: 对于某些属性如 undefined,Symbol,function,Symbol 会自动忽略; 对于set map

会转成对象
- 自己实现或使用第三方库

如何实现深拷贝和浅拷贝?

浅拷贝:

```
const info = {
  name: "why",
  age: 18,
  friend: {
    name: "kobe"
  },
  running: function() {},
  [Symbol()]: "abc"
}

// 浅拷贝-方式一
const obj2 = { ...info }

// 浅拷贝-方式二
const obj3 = Object.assign({}, info)
```

深拷贝:

```
function isObject(obj) {
    return obj !== null && (typeof obj === "object" || typeof obj ===
"function")
}

function deepClone(originValue) {
    // Symbol类型
    if (typeof originValue === "symbol") {
      return Symbol(originValue.description)
    }

    // 判断是否是对象
    if (!isObject(originValue)) return originValue;

    // set类型
    if (originValue instanceof Set) {
      const newSet = new Set()
      for (const setItem of originValue) {
        newSet.add(deepClone(setItem))
      }
```

```
      return newSet
    }

    // 判断是函数
    if (typeof originValue === "function") {
      return originValue
    }

    // 判断返回值是数组还是对象
    const newObj = Array.isArray(originValue) ? [] : {}
    if (Reflect) {
      for (let key of Reflect.ownKeys(originValue)) {
        {
          let value = originValue[key]
          // 让 SymbolKey的值不同
          if (typeof key === "symbol") {
            const newSymbolKey = Symbol(key.description)
            // 将原来的值赋值给新生成的Symbol key
            value = originValue[key]
            key = newSymbolKey
          }
          newObj[key] = deepClone(value)
        }
      }
    } else {
      for (const key in originValue) {
        const value = originValue[key]
        newObj[key] = deepClone(value)
      }

      // 对于Symbol类型的key forin 无法便利出来
      const symbolKeys = Object.getOwnPropertySymbols(originValue)
       for (const symbolKey in symbolKeys) {
       const originSymbolValue = symbolKeys[symbolKey]
       const newSymbol = Symbol(originSymbolValue.description)
       newObj[newSymbol] = deepClone(originValue[originSymbolValue])
      }
    }
    return newObj
}


const info = {
  name: "why",
  age: 18,
  friend: {
    name: "kobe"
  },
  running: function() {},
```

```
  [Symbol()]: "abc"
}
const newObj = deepCopy(info)
```

说说服务端渲染和前后端分离的区别

SSR(server side rendering) 服务端渲染

- 优点:
  - 更快的响应时间 不用等待所有的js加载完成 也能显示比较完整的页面
  - 更好的SEO 可以将SEO的关键信息直接在后台渲染成html 保证了搜索引擎能爬取到关键数据
  - 无需占用客户端资源 解析模板交给后端工作 对于客户端的资源占用更少
- 缺点
  - 占用服务器资源 一个小小的页面的改动 都需要请求一次完整的html页面 有悖于程序员的 =="DRY(Don`t repeat yourself)"==原则 如果短时间访问过多 对服务器造成一定的访问压力
  -
  - 一些常见的api需要先对运行环境判断再使用

前后端分离

- 优点:
  - 前端专注于ui界面的开发 后端专注于api的开发 单一
  - 体验更好
- 缺点:
  - 第一次响应内容较慢 不如服务端渲染快
  - 不利于SEO优化 只是记录一个页面 对于SEO较差