## 这儿有20道大厂面试题等你查收

[前端技术优选](javascript:void(0);) 昨天

今年来，各大公司都缩减了HC，甚至是采取了“裁员”措施，在这样的大环境之下，想要获得一份更好的工作，必然需要付出更多的努力。本文挑选了20道大厂面试题，建议在阅读时，先思考一番，不要直接看解析。尽管，本文所有的答案，都是我在翻阅各种资料，思考并验证之后，才给出的。但因水平有限，本人的答案未必是最优的，如果您有更好的答案，欢迎给我留言。如果有错误，可以在评论区指出。本文篇幅较长，希望小伙伴们能够坚持读完，如果想加入交流群，可以通过公众号添加我为好友。

### 1. new的实现原理是什么？

new 的实现原理:

1. 创建一个空对象，构造函数中的this指向这个空对象
2. 这个新对象被执行 [[原型]] 连接
3. 执行构造函数方法，属性和方法被添加到this引用的对象中
4. 如果构造函数中没有返回其它对象，那么返回this，即创建的这个的新对象，否则，返回构造函数中返回的对象。



### 2. 如何正确判断this的指向？

如果用一句话说明 this 的指向，那么即是: 谁调用它，this 就指向谁。

但是仅通过这句话，我们很多时候并不能准确判断 this 的指向。因此我们需要借助一些规则去帮助自己：

this 的指向可以按照以下顺序判断:

#### 全局环境中的 this

浏览器环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this 都指向全局对象 window;

node 环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部），this 都是空对象 {};

#### 是否是 new 绑定

如果是 new 绑定，并且构造函数中没有返回 function 或者是 object，那么 this 指向这个新对象。如下:

构造函数返回值不是 function 或 object。 newSuper() 返回的是 this 对象。

构造函数返回值是 function 或 object， newSuper()是返回的是Super种返回的对象。

#### 函数是否通过 call,apply 调用，或者使用了 bind 绑定，如果是，那么this绑定的就是指定的对象【归结为显式绑定】。



这里同样需要注意一种**特殊**情况，如果 call,apply 或者 bind 传入的第一个参数值是 undefined或者 null，严格模式下 this 的值为传入的值 null /undefined。非严格模式下，实际应用的默认绑定规则，this 指向全局对象(node环境为global，浏览器环境为window)



#### 隐式绑定，函数的调用是在某个对象上触发的，即调用位置上存在上下文对象。典型的隐式调用为: xxx.fn()

#### 默认绑定，在不能应用其它绑定规则时使用的默认规则，通常是独立函数调用。

非严格模式：node环境，执行全局对象 global，浏览器环境，执行全局对象 window。

严格模式：执行 undefined

#### 箭头函数的情况：

箭头函数没有自己的this，继承外层上下文绑定的this。

### 3. 深拷贝和浅拷贝的区别是什么？实现一个深拷贝

深拷贝和浅拷贝是针对复杂数据类型来说的，浅拷贝只拷贝一层，而深拷贝是层层拷贝。

#### 深拷贝

深拷贝复制变量值，对于非基本类型的变量，则递归至基本类型变量后，再复制。深拷贝后的对象与原来的对象是完全隔离的，互不影响，对一个对象的修改并不会影响另一个对象。

#### 浅拷贝

浅拷贝是会将对象的每个属性进行依次复制，但是当对象的属性值是引用类型时，实质复制的是其引用，当引用指向的值改变时也会跟着变化。

可以使用 forin、 Object.assign、 扩展运算符 ... 、 Array.prototype.slice()、 Array.prototype.concat() 等，例如:

可以看出浅拷贝只最第一层属性进行了拷贝，当第一层的属性值是基本数据类型时，新的对象和原对象互不影响，但是如果第一层的属性值是复杂数据类型，那么新对象和原对象的属性值其指向的是同一块内存地址。

#### 深拷贝实现

1.深拷贝最简单的实现是: JSON.parse(JSON.stringify(obj))

JSON.parse(JSON.stringify(obj)) 是最简单的实现方式，但是有一些缺陷：

1. 对象的属性值是函数时，无法拷贝。
2. 原型链上的属性无法拷贝
3. 不能正确的处理 Date 类型的数据
4. 不能处理 RegExp
5. 会忽略 symbol
6. 会忽略 undefined

2.实现一个 deepClone 函数

1. 如果是基本数据类型，直接返回
2. 如果是 RegExp 或者 Date 类型，返回对应类型
3. 如果是复杂数据类型，递归。
4. 考虑循环引用的问题



### 4. call/apply 的实现原理是什么？

call 和 apply 的功能相同，都是改变 this 的执行，并立即执行函数。区别在于传参方式不同。

* func.call(thisArg,arg1,arg2,...)：第一个参数是 this 指向的对象，其它参数依次传入。
* func.apply(thisArg,[argsArray])：第一个参数是 this 指向的对象，第二个参数是数组或类数组。

一起思考一下，如何模拟实现 call ？

首先，我们知道，函数都可以调用 call，说明 call 是函数原型上的方法，所有的实例都可以调用。即: Function.prototype.call。

* 在 call 方法中获取调用 call()函数
* 如果第一个参数没有传入，那么默认指向 window/global(非严格模式)
* 传入 call 的第一个参数是 this 指向的对象，根据隐式绑定的规则，我们知道 obj.foo(), foo() 中的 this 指向 obj;因此我们可以这样调用函数 thisArgs.func(...args)
* 返回执行结果

apply 的实现思路和 call 一致，仅参数处理略有差别。如下：



### 5. 柯里化函数实现

在开始之前，我们首先需要搞清楚函数柯里化的概念。

函数柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

函数柯里化的主要作用：

* 参数复用
* 提前返回 – 返回接受余下的参数且返回结果的新函数
* 延迟执行 – 返回新函数，等待执行

### 6. 如何让 (a == 1 && a == 2 && a == 3) 的值为true？

1. 利用隐式类型转换

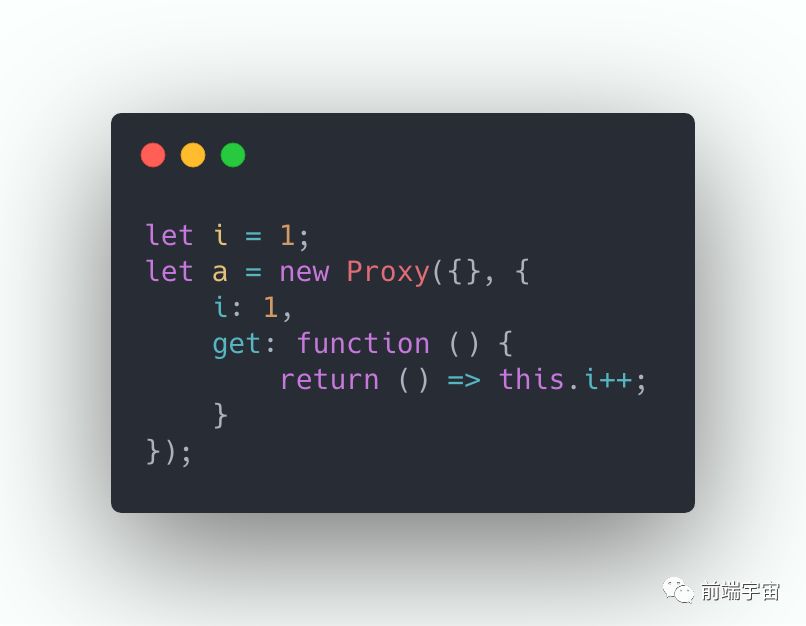
== 操作符在左右数据类型不一致时，会先进行隐式转换。

a==1&&a==2&&a==3 的值意味着其不可能是基本数据类型。因为如果 a 是 null 或者是 undefined bool类型，都不可能返回true。

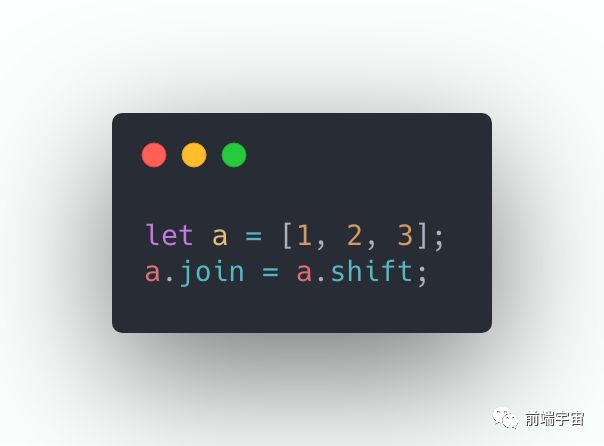
因此可以推测 a 是复杂数据类型，JS 中复杂数据类型只有 object，回忆一下，Object 转换为原始类型会调用什么方法？

* 如果部署了 [Symbol.toPrimitive] 接口，那么调用此接口，若返回的不是基本数据类型，抛出错误。
* 如果没有部署 [Symbol.toPrimitive] 接口，那么根据要转换的类型，先调用 valueOf / toString
  1. 非Date类型对象， hint 是 default 时，调用顺序为： valueOf >>> toString，即 valueOf 返回的不是基本数据类型，才会继续调用 valueOf，如果 toString 返回的还不是基本数据类型，那么抛出错误。
  2. 如果 hint 是 string(Date对象的hint默认是string) ，调用顺序为： toString >>> valueOf，即 toString 返回的不是基本数据类型，才会继续调用 valueOf，如果 valueOf 返回的还不是基本数据类型，那么抛出错误。
  3. 如果 hint 是 number，调用顺序为： valueOf >>> toString

1. 利用数据劫持(Proxy/Object.definedProperty)



1. 数组的 toString 接口默认调用数组的 join 方法，重新 join 方法



### 7. 什么是BFC？BFC的布局规则是什么？如何创建BFC？

Box 是 CSS 布局的对象和基本单位，页面是由若干个Box组成的。

元素的类型 和 display 属性，决定了这个 Box 的类型。不同类型的 Box 会参与不同的 Formatting Context。

Formatting Context

Formatting Context 是页面的一块渲染区域，并且有一套渲染规则，决定了其子元素将如何定位，以及和其它元素的关系和相互作用。

Formatting Context 有 BFC (Block formatting context)，IFC (Inline formatting context)，FFC (Flex formatting context) 和 GFC (Grid formatting context)。FFC 和 GFC 为 CC3 中新增。

BFC布局规则

* BFC内，盒子依次垂直排列。
* BFC内，两个盒子的垂直距离由 margin 属性决定。属于同一个BFC的两个相邻Box的margin会发生重叠【符合合并原则的margin合并后是使用大的margin】
* BFC内，每个盒子的左外边缘接触内部盒子的左边缘（对于从右到左的格式，右边缘接触）。即使在存在浮动的情况下也是如此。除非创建新的BFC。
* BFC的区域不会与float box重叠。
* BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。
* 计算BFC的高度时，浮动元素也参与计算。

如何创建BFC

* 根元素
* 浮动元素（float 属性不为 none）
* position 为 absolute 或 fixed
* overflow 不为 visible 的块元素
* display 为 inline-block, table-cell, table-caption

BFC 的应用

1. 防止 margin 重叠 (同一个BFC内的两个两个相邻Box的 margin 会发生重叠，触发生成两个BFC，即不会重叠)
2. 清除内部浮动 (创建一个新的 BFC，因为根据 BFC 的规则，计算 BFC 的高度时，浮动元素也参与计算)
3. 自适应多栏布局 (BFC的区域不会与float box重叠。因此，可以触发生成一个新的BFC)

### 8. 异步加载JS脚本的方式有哪些？

<script> 标签中增加 async(html5) 或者 defer(html4) 属性,脚本就会异步加载。

<scriptsrc="../XXX.js"defer></script>

defer 和 async 的区别在于：

* defer 要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），在window.onload 之前执行；
* async 一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。
* 如果有多个 defer 脚本，会按照它们在页面出现的顺序加载
* 多个 async 脚本不能保证加载顺序

动态创建 script 标签

动态创建的 script ，设置 src 并不会开始下载，而是要添加到文档中，JS文件才会开始下载。



XHR 异步加载JS

### 9. ES5有几种方式可以实现继承？分别有哪些优缺点？

ES5 有 6 种方式可以实现继承，分别为：

##### 1. 原型链继承

原型链继承的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。



缺点：

1. 通过原型来实现继承时，原型会变成另一个类型的实例，原先的实例属性变成了现在的原型属性，该原型的引用类型属性会被所有的实例共享。
2. 在创建子类型的实例时，没有办法在不影响所有对象实例的情况下给超类型的构造函数中传递参数。

##### 2. 借用构造函数

**借用构造函数**的技术，其基本思想为:

在子类型的构造函数中调用超类型构造函数。



优点:

1. 可以向超类传递参数
2. 解决了原型中包含引用类型值被所有实例共享的问题

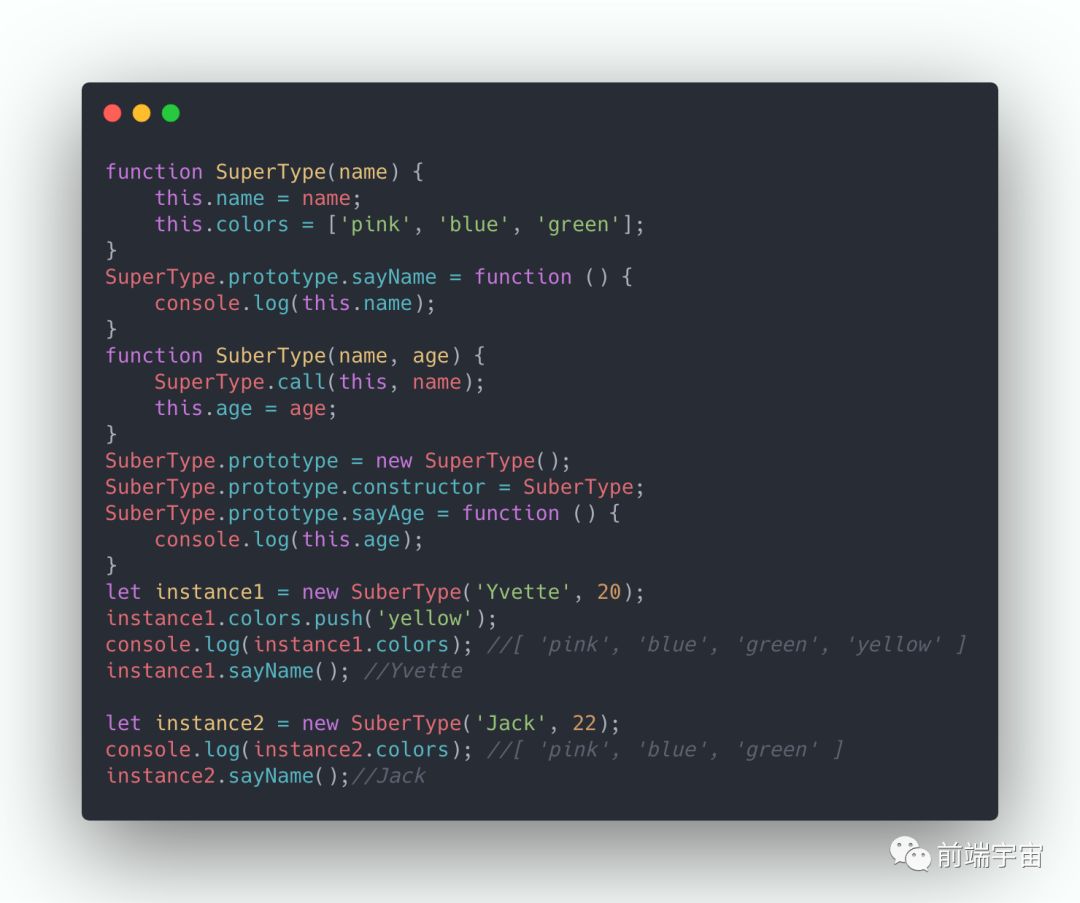
缺点:

1. 方法都在构造函数中定义，函数复用无从谈起，另外超类型原型中定义的方法对于子类型而言都是不可见的。

##### 3. 组合继承(原型链 + 借用构造函数)

组合继承指的是将原型链和借用构造函数技术组合到一块，从而发挥二者之长的一种继承模式。基本思路：

使用原型链实现对原型属性和方法的继承，通过借用构造函数来实现对实例属性的继承，既通过在原型上定义方法来实现了函数复用，又保证了每个实例都有自己的属性。



缺点:

* 无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。

优点:

* 可以向超类传递参数
* 每个实例都有自己的属性
* 实现了函数复用

##### 4. 原型式继承

原型继承的基本思想：

借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。

在 object() 函数内部，先穿甲一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例，从本质上讲， object() 对传入的对象执行了一次浅拷贝。

ECMAScript5通过新增 Object.create()方法规范了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象(可以覆盖原型对象上的同名属性)，在传入一个参数的情况下， Object.create() 和 object() 方法的行为相同。

在没有必要创建构造函数，仅让一个对象与另一个对象保持相似的情况下，原型式继承是可以胜任的。

缺点:

同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

##### 5. 寄生式继承

寄生式继承是与原型式继承紧密相关的一种思路。寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部已某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。

基于 person 返回了一个新对象 -—— person2，新对象不仅具有 person 的所有属性和方法，而且还有自己的 sayHi() 方法。在考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。

缺点：

* 使用寄生式继承来为对象添加函数，会由于不能做到函数复用而效率低下。
* 同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

##### 6. 寄生组合式继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法，基本思路：

不必为了指定子类型的原型而调用超类型的构造函数，我们需要的仅是超类型原型的一个副本，本质上就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。寄生组合式继承的基本模式如下所示：



* 第一步：创建超类型原型的一个副本
* 第二步：为创建的副本添加 constructor 属性
* 第三步：将新创建的对象赋值给子类型的原型

至此，我们就可以通过调用 inheritPrototype 来替换为子类型原型赋值的语句：

优点:

只调用了一次超类构造函数，效率更高。避免在 SuberType.prototype上面创建不必要的、多余的属性，与其同时，原型链还能保持不变。

因此寄生组合继承是引用类型最理性的继承范式。

### 10. 隐藏页面中的某个元素的方法有哪些？

隐藏类型

屏幕并不是唯一的输出机制，比如说屏幕上看不见的元素（隐藏的元素），其中一些依然能够被读屏软件阅读出来（因为读屏软件依赖于可访问性树来阐述）。为了消除它们之间的歧义，我们将其归为三大类：

* 完全隐藏：元素从渲染树中消失，不占据空间。
* 视觉上的隐藏：屏幕中不可见，占据空间。
* 语义上的隐藏：读屏软件不可读，但正常占据空。

完全隐藏

##### 1. display 属性

1. display: none;

##### 2.hidden 属性

HTML5 新增属性，相当于 display:none

1. <div hidden>
2. </div>

视觉上的隐藏

##### 1.利用 position 和 盒模型 将元素移出可视区范围

1. 设置 posoition 为 absolute 或 fixed，通过设置 top、 left 等值，将其移出可视区域。
2. position:absolute;
3. left: -99999px;
4. 设置 position 为 relative，通过设置 top、 left 等值，将其移出可视区域。
5. position: relative;
6. left: -99999px;
7. height: 0
8. 设置 margin 值，将其移出可视区域范围（可视区域占位）。
9. margin-left: -99999px;
10. height: 0;

##### 2.利用 transfrom

1. 缩放
2. transform: scale(0);
3. height: 0;
4. 移动 translateX, translateY
5. transform: translateX(-99999px);
6. height: 0
7. 旋转 rotate
8. transform: rotateY(90deg);

##### 3.设置其大小为0

1. 宽高为0，字体大小为0：
2. height: 0;
3. width: 0;
4. font-size: 0;
5. 宽高为0，超出隐藏:
6. height: 0;
7. width: 0;
8. overflow: hidden;

##### 4.设置透明度为0

1. opacity: 0;

##### 5. visibility属性

1. visibility: hidden;

##### 6.层级覆盖， z-index 属性

1. position: relative;
2. z-index: -999;

再设置一个层级较高的元素覆盖在此元素上。

##### 7.clip-path 裁剪

1. clip-path: polygon(0 0, 0 0, 0 0, 0 0);

语义上的隐藏

##### aria-hidden 属性

读屏软件不可读，占据空间，可见。

1. <div aria-hidden="true">
2. </div>

### 11. let、const、var 的区别有哪些？

1.let/const 定义的变量不会出现变量提升，而 var 定义的变量会提升。

2.相同作用域中，let 和 const 不允许重复声明，var 允许重复声明。

3.const 声明变量时必须设置初始值

4.const 声明一个只读的常量，这个常量不可改变。

这里有一个非常重要的点即是：在JS中，复杂数据类型，存储在栈中的是堆内存的地址，存在栈中的这个地址是不变的，但是存在堆中的值是可以变得。有没有相当常量指针/指针常量~

一图胜万言，如下图所示，不变的是栈内存中 a 存储的 20，和 b 中存储的 0x0012ff21（瞎编的一个数字）。而 {age: 18, star: 200} 是可变的。

### 12. 说一说你对JS执行上下文栈和作用域链的理解？

在开始说明JS上下文栈和作用域之前，我们先说明下JS上下文以及作用域的概念。

#### JS执行上下文

执行上下文就是当前 JavaScript 代码被解析和执行时所在环境的抽象概念， JavaScript 中运行任何的代码都是在执行上下文中运行。

执行上下文类型分为：

* 全局执行上下文
* 函数执行上下文

执行上下文创建过程中，需要做以下几件事:

1. 创建变量对象：首先初始化函数的参数arguments，提升函数声明和变量声明。
2. 创建作用域链（Scope Chain）：在执行期上下文的创建阶段，作用域链是在变量对象之后创建的。
3. 确定this的值，即 ResolveThisBinding

#### 作用域

**作用域**负责收集和维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。—— 摘录自《你不知道的JavaScript》(上卷)

作用域有两种工作模型：词法作用域和动态作用域，JS采用的是**词法作用域**工作模型，词法作用域意味着作用域是由书写代码时变量和函数声明的位置决定的。( with 和 eval 能够修改词法作用域，但是不推荐使用，对此不做特别说明)

作用域分为：

* 全局作用域
* 函数作用域
* 块级作用域

#### JS执行上下文栈(后面简称执行栈)

执行栈，也叫做调用栈，具有 **LIFO** (后进先出) 结构，用于存储在代码执行期间创建的所有执行上下文。

规则如下：

* 首次运行JavaScript代码的时候,会创建一个全局执行的上下文并Push到当前的执行栈中，每当发生函数调用，引擎都会为该函数创建一个新的函数执行上下文并Push当前执行栈的栈顶。
* 当栈顶的函数运行完成后，其对应的函数执行上下文将会从执行栈中Pop出，上下文的控制权将移动到当前执行栈的下一个执行上下文。

以一段代码具体说明：

GlobalExecutionContext (即全局执行上下文)首先入栈，过程如下：

伪代码:

1. //全局执行上下文首先入栈
2. ECStack.push(globalContext);
3. //执行fun1();
4. ECStack.push(<fun1> functionContext);
5. //fun1中又调用了fun2;
6. ECStack.push(<fun2> functionContext);
7. //fun2中又调用了fun3;
8. ECStack.push(<fun3> functionContext);
9. //fun3执行完毕
10. ECStack.pop();
11. //fun2执行完毕
12. ECStack.pop();
13. //fun1执行完毕
14. ECStack.pop();
15. //javascript继续顺序执行下面的代码，但ECStack底部始终有一个 全局上下文（globalContext）;

#### 作用域链

作用域链就是从当前作用域开始一层一层向上寻找某个变量，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。

如：

fn2作用域链 = [fn2作用域, fn1作用域，全局作用域]

### 13. 防抖函数的作用是什么？请实现一个防抖函数

防抖函数的作用

防抖函数的作用就是控制函数在一定时间内的执行次数。防抖意味着N秒内函数只会被执行一次，如果N秒内再次被触发，则**重新**计算延迟时间。

**举例说明：** 小思最近在减肥，但是她非常吃吃零食。为此，与其男朋友约定好，如果10天不吃零食，就可以购买一个包(不要问为什么是包，因为**包治百病**)。但是如果中间吃了一次零食，那么就要重新计算时间，直到小思坚持10天没有吃零食，才能购买一个包。所以，管不住嘴的小思，没有机会买包(悲伤的故事)... 这就是 **防抖**。

防抖函数实现

1. 事件第一次触发时， timer 是 null，调用 later()，若 immediate 为 true，那么立即调用 func.apply(this,params)；如果 immediate 为 false，那么过 wait 之后，调用 func.apply(this,params)
2. 事件第二次触发时，如果 timer 已经重置为 null(即 setTimeout 的倒计时结束)，那么流程与第一次触发时一样，若 timer 不为 null(即 setTimeout 的倒计时未结束)，那么清空定时器，重新开始计时。

immediate 为 true 时，表示函数在每个等待时延的开始被调用。 immediate 为 false 时，表示函数在每个等待时延的结束被调用。

防抖的应用场景

1. 搜索框输入查询，如果用户一直在输入中，没有必要不停地调用去请求服务端接口，等用户停止输入的时候，再调用，设置一个合适的时间间隔，有效减轻服务端压力。
2. 表单验证
3. 按钮提交事件。
4. 浏览器窗口缩放，resize事件(如窗口停止改变大小之后重新计算布局)等。

### 14. 节流函数的作用是什么？有哪些应用场景，请实现一个节流函数

节流函数的作用

节流函数的作用是规定一个单位时间，在这个单位时间内最多只能触发一次函数执行，如果这个单位时间内多次触发函数，只能有一次生效。

节流函数实现

禁用第一次首先执行，传递 {leading:false} ；想禁用最后一次执行，传递 {trailing:false}

节流的应用场景

1. 按钮点击事件
2. 拖拽事件
3. onScoll
4. 计算鼠标移动的距离(mousemove)

### 15. 什么是闭包？闭包的作用是什么？

##### 闭包的定义

《JavaScript高级程序设计》:

闭包是指有权访问另一个函数作用域中的变量的函数

《JavaScript权威指南》：

从技术的角度讲，所有的JavaScript函数都是闭包：它们都是对象，它们都关联到作用域链。

《你不知道的JavaScript》

当函数可以记住并访问所在的词法作用域时，就产生了闭包，即使函数是在当前词法作用域之外执行。

##### 创建一个闭包

闭包使得函数可以继续访问定义时的词法作用域。拜 fn 所赐，在 foo() 执行后，foo 内部作用域不会被销毁。

##### 闭包的作用

* 能够访问函数定义时所在的词法作用域(阻止其被回收)。
* 私有化变量
* 模拟块级作用域
* 创建模块

模块模式具有两个必备的条件(来自《你不知道的JavaScript》)

* 必须有外部的封闭函数，该函数必须至少被调用一次(每次调用都会创建一个新的模块实例)
* 封闭函数必须返回至少**一个**内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态。

### 16. 实现 Promise.all 方法

在实现 Promise.all 方法之前，我们首先要知道 Promise.all 的功能和特点，因为在清楚了 Promise.all 功能和特点的情况下，我们才能进一步去写实现。

Promise.all 功能

Promise.all(iterable) 返回一个新的 Promise 实例。此实例在 iterable 参数内所有的 promise 都 fulfilled 或者参数中不包含 promise 时，状态变成 fulfilled；如果参数中 promise 有一个失败 rejected，此实例回调失败，失败原因的是第一个失败 promise 的返回结果。

1. let p = Promise.all([p1, p2, p3]);

p的状态由 p1,p2,p3决定，分成以下；两种情况：

（1）只有p1、p2、p3的状态都变成 fulfilled，p的状态才会变成 fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被 rejected，p的状态就变成 rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

Promise.all 的特点

Promise.all 的返回值是一个 promise 实例

* 如果传入的参数为空的可迭代对象， Promise.all 会 **同步** 返回一个已完成状态的 promise
* 如果传入的参数中不包含任何 promise, Promise.all 会 **异步** 返回一个已完成状态的 promise
* 其它情况下， Promise.all 返回一个 **处理中（pending）** 状态的 promise.

Promise.all 返回的 promise 的状态

* 如果传入的参数中的 promise 都变成完成状态， Promise.all 返回的 promise 异步地变为完成。
* 如果传入的参数中，有一个 promise 失败， Promise.all 异步地将失败的那个结果给失败状态的回调函数，而不管其它 promise 是否完成
* 在任何情况下， Promise.all 返回的 promise 的完成状态的结果都是一个数组

Promise.all 实现

### 17. 请实现一个 flattenDeep 函数，把嵌套的数组扁平化

例如:

1. flattenDeep([1, [2, [3, [4]], 5]]); //[1, 2, 3, 4, 5]

利用 Array.prototype.flat

ES6 为数组实例新增了 flat 方法，用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数组没有影响。

flat 默认只会 “拉平” 一层，如果想要 “拉平” 多层的嵌套数组，需要给 flat 传递一个整数，表示想要拉平的层数。

当传递的整数大于数组嵌套的层数时，会将数组拉平为一维数组，JS能表示的最大数字为 Math.pow(2,53)-1，因此我们可以这样定义 flattenDeep 函数

利用 reduce 和 concat

使用 stack 无限反嵌套多层嵌套数组

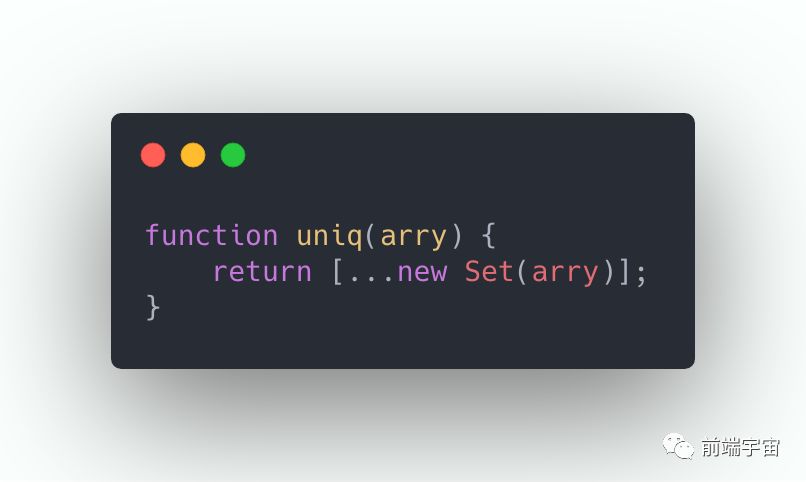
### 18. 请实现一个 uniq 函数，实现数组去重

例如:

1. uniq([1, 2, 3, 5, 3, 2]);//[1, 2, 3, 5]

法1: 利用ES6新增数据类型 Set

Set类似于数组，但是成员的值都是唯一的，没有重复的值。



法2: 利用 indexOf

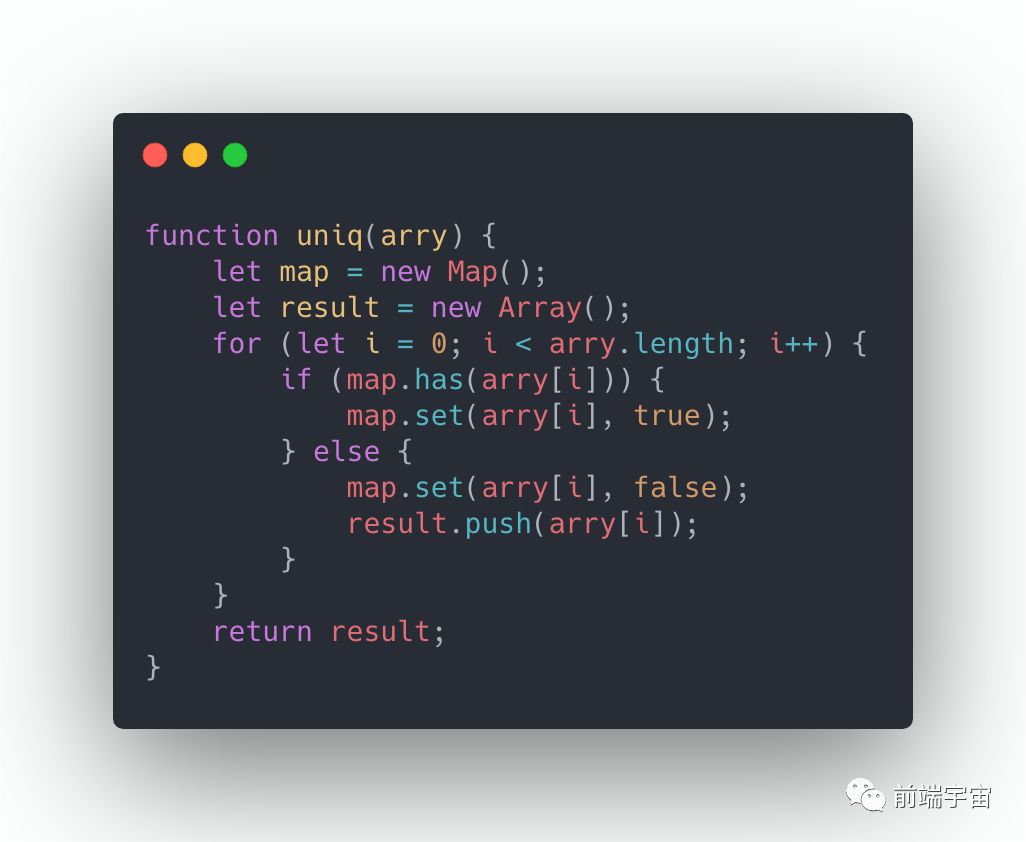
法3: 利用 includes



法4：利用 reduce



法5：利用 Map



### 19. 可迭代对象有哪些特点

ES6 规定，默认的 Iterator 接口部署在数据结构的 Symbol.iterator 属性，换个角度，也可以认为，一个数据结构只要具有 Symbol.iterator 属性( Symbol.iterator 方法对应的是遍历器生成函数，返回的是一个遍历器对象)，那么就可以其认为是可迭代的。

#### 可迭代对象的特点

* 具有 Symbol.iterator 属性， Symbol.iterator() 返回的是一个遍历器对象
* 可以使用 for...of 进行循环
* 通过被 Array.from 转换为数组

#### 原生具有 Iterator 接口的数据结构：

* Array
* Map
* Set
* String
* TypedArray
* 函数的 arguments 对象
* NodeList 对象

### 20. JSONP 的原理是什么？

尽管浏览器有同源策略，但是 <script> 标签的 src 属性不会被同源策略所约束，可以获取任意服务器上的脚本并执行。 jsonp 通过插入 script 标签的方式来实现跨域，参数只能通过 url 传入，仅能支持 get 请求。

实现原理:

* Step1: 创建 callback 方法
* Step2: 插入 script 标签
* Step3: 后台接受到请求，解析前端传过去的 callback 方法，返回该方法的调用，并且数据作为参数传入该方法
* Step4: 前端执行服务端返回的方法调用

jsonp源码实现

使用:



服务端代码(node):



### 参考文章：

[1] 珠峰架构课(墙裂推荐)

[2] [JavaScript高级程序设计第六章]

[3] Step-By-Step】高频面试题深入解析 / 周刊01

[4] Step-By-Step】高频面试题深入解析 / 周刊02

[5] Step-By-Step】高频面试题深入解析 / 周刊03

[6] Step-By-Step】高频面试题深入解析 / 周刊04