**面试篇寒冬求职季之你必须要懂的原生JS(上)**

**1.原始类型有哪几种？null 是对象吗？原始数据类型和复杂数据类型存储有什么区别？**

* 原始类型有6种，分别是undefined,null,bool,string,number,symbol(ES6新增)。
* 虽然 typeof null 返回的值是 object,但是null不是对象，而是基本数据类型的一种。
* 原始数据类型存储在栈内存，存储的是值。
* 复杂数据类型存储在堆内存，存储的是地址。当我们把对象赋值给另外一个变量的时候，复制的是地址，指向同一块内存空间，当其中一个对象改变时，另一个对象也会变化

**2. typeof 是否正确判断类型? instanceof呢？ instanceof 的实现原理是什么？**

首先 typeof 能够正确的判断基本数据类型，但是除了 null, typeof null输出的是对象。

但是对象来说，typeof 不能正确的判断其类型， typeof 一个函数可以输出 'function',而除此之外，输出的全是 object,这种情况下，我们无法准确的知道对象的类型。

instanceof可以准确的判断复杂数据类型，但是不能正确判断基本数据类型。(正确判断数据类型请戳：[github.com/YvetteLau/B…](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FYvetteLau%2FBlog%2Fblob%2Fmaster%2FJS%2Fdata-type.js" \t "_blank))

instanceof 是通过原型链判断的，A instanceof B, 在A的原型链中层层查找，是否有原型等于B.prototype，如果一直找到A的原型链的顶端(null;即Object.prototype.\_\_proto\_\_),仍然不等于B.prototype，那么返回false，否则返回true.

instanceof的实现代码:

// L instanceof R

function instance\_of(L, R) {//L 表示左表达式，R 表示右表达式

var O = R.prototype;// 取 R 的显式原型

L = L.\_\_proto\_\_; // 取 L 的隐式原型

while (true) {

if (L === null) //已经找到顶层

return false;

if (O === L) //当 O 严格等于 L 时，返回 true

return true;

L = L.\_\_proto\_\_; //继续向上一层原型链查找

}

}

**3. for of , for in 和 forEach,map 的区别。**

* for...of循环：具有 iterator 接口，就可以用for...of循环遍历它的成员(属性值)。for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象、Generator 对象，以及字符串。for...of循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性。对于普通的对象，for...of结构不能直接使用，会报错，必须部署了 Iterator 接口后才能使用。可以中断循环。
* for...in循环：遍历对象自身的和继承的可枚举的属性, 不能直接获取属性值。可以中断循环。
* forEach: 只能遍历数组，不能中断，没有返回值(或认为返回值是undefined)。
* map: 只能遍历数组，不能中断，返回值是修改后的数组。

PS: Object.keys()：返回给定对象所有可枚举属性的字符串数组。

关于forEach是否会改变原数组的问题，有些小伙伴提出了异议，为此我写了代码测试了下(注意数组项是复杂数据类型的情况)。 除了forEach之外，map等API，也有同样的问题。

let arry = [1, 2, 3, 4];

arry.forEach((item) => {

item \*= 10;

});

console.log(arry); //[1, 2, 3, 4]

arry.forEach((item) => {

arry[1] = 10; //直接操作数组

});

console.log(arry); //[ 1, 10, 3, 4 ]

let arry2 = [

{ name: "Yve" },

{ age: 20 }

];

arry2.forEach((item) => {

item.name = 10;

});

console.log(arry2);//[ { name: 10 }, { age: 20, name: 10 } ]

**4. 如何判断一个变量是不是数组？**

1.使用 Array.isArray 判断，如果返回 true, 说明是数组

2.使用 instanceof Array 判断，如果返回true, 说明是数组

3.使用 Object.prototype.toString.call 判断，如果值是 [object Array], 说明是数组

4.通过 constructor 来判断，如果是数组，那么 arr.constructor === Array. (不准确，因为我们可以指定 obj.constructor = Array)

function fn() {

console.log(Array.isArray(arguments));

//false; 因为arguments是类数组，但不是数组

console.log(Array.isArray([1,2,3,4])); //true

console.log(arguments instanceof Array); //fasle

console.log([1,2,3,4] instanceof Array); //true

console.log(Object.prototype.toString.call(arguments)); //[object Arguments]

console.log(Object.prototype.toString.call([1,2,3,4])); //[object Array]

console.log(arguments.constructor === Array); //false

arguments.constructor = Array;

console.log(arguments.constructor === Array); //true

console.log(Array.isArray(arguments)); //false

}

fn(1,2,3,4);

**5.类数组和数组的区别是什么？**

类数组:

1）拥有length属性，其它属性（索引）为非负整数（对象中的索引会被当做字符串来处理）;

2）不具有数组所具有的方法；

类数组是一个普通对象，而真实的数组是Array类型。

常见的类数组有: 函数的参数 arugments, DOM 对象列表(比如通过 document.querySelectorAll 得到的列表), jQuery 对象 (比如 $("div")).

类数组可以转换为数组:

//第一种方法

Array.prototype.slice.call(arrayLike, start);

//第二种方法

[...arrayLike];

//第三种方法:

Array.from(arrayLike);

**PS:** 任何定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组。

Array.from方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象。

**6. == 和 === 有什么区别？**

=== 不需要进行类型转换，只有类型相同并且值相等时，才返回 true.

== 如果两者类型不同，首先需要进行类型转换。具体流程如下:

1. 首先判断两者类型是否相同，如果相等，判断值是否相等.
2. 如果类型不同，进行类型转换
3. 判断比较的是否是 null 或者是 undefined, 如果是, 返回 true .
4. 判断两者类型是否为 string 和 number, 如果是, 将字符串转换成 number
5. 判断其中一方是否为 boolean, 如果是, 将 boolean 转为 number 再进行判断
6. 判断其中一方是否为 object 且另一方为 string、number 或者 symbol , 如果是, 将 object 转为原始类型再进行判断

let person1 = {

age: 25

}

let person2 = person1;

person2.gae = 20;

console.log(person1 === person2);

//true,注意复杂数据类型，比较的是引用地址

### 思考: [] == ![]

我们来分析一下: [] == ![] 是true还是false？

1. 首先，我们需要知道 ! 优先级是高于 == (更多运算符优先级可查看: [运算符优先级](https://link.juejin.im?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FOperators%2FOperator_Precedence))
2. ![] 引用类型转换成布尔值都是true,因此![]的是false
3. 根据上面的比较步骤中的第五条，其中一方是 boolean，将 boolean 转为 number 再进行判断，false转换成 number，对应的值是 0.
4. 根据上面比较步骤中的第六条，有一方是 number，那么将object也转换成Number,空数组转换成数字，对应的值是0.(空数组转换成数字，对应的值是0，如果数组中只有一个数字，那么转成number就是这个数字，其它情况，均为NaN)
5. 0 == 0; 为true

**7. ES6中的class和ES5的类有什么区别？**

1. ES6 class 内部所有定义的方法都是不可枚举的;
2. ES6 class 必须使用 new 调用;
3. ES6 class 不存在变量提升;
4. ES6 class 默认即是严格模式;
5. ES6 class 子类必须在父类的构造函数中调用super()，这样才有this对象;ES5中类继承的关系是相反的，先有子类的this，然后用父类的方法应用在this上。

**8. 数组的哪些API会改变原数组？**

修改原数组的API有:

splice/reverse/fill/copyWithin/sort/push/pop/unshift/shift

不修改原数组的API有:

slice/map/forEach/every/filter/reduce/entry/entries/find

注: 数组的每一项是简单数据类型，且未直接操作数组的情况下(稍后会对此题重新作答)。

**9. let、const 以及 var 的区别是什么？**

* let 和 const 定义的变量不会出现变量提升，而 var 定义的变量会提升。
* let 和 const 是JS中的块级作用域
* let 和 const 不允许重复声明(会抛出错误)
* let 和 const 定义的变量在定义语句之前，如果使用会抛出错误(形成了暂时性死区)，而 var 不会。
* const 声明一个只读的常量。一旦声明，常量的值就不能改变(如果声明是一个对象，那么不能改变的是对象的引用地址)

**10. 在JS中什么是变量提升？什么是暂时性死区？**

变量提升就是变量在声明之前就可以使用，值为undefined。

在代码块内，使用 let/const 命令声明变量之前，该变量都是不可用的(会抛出错误)。这在语法上，称为“暂时性死区”。暂时性死区也意味着 typeof 不再是一个百分百安全的操作。

typeof x; // ReferenceError(暂时性死区，抛错)

let x;

复制代码

typeof y; // 值是undefined,不会报错

复制代码

暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

**11. 如何正确的判断this? 箭头函数的this是什么？**

this的绑定规则有四种：默认绑定，隐式绑定，显式绑定，new绑定.

1. 函数是否在 new 中调用(new绑定)，如果是，那么 this 绑定的是新创建的对象。
2. 函数是否通过 call,apply 调用，或者使用了 bind (即硬绑定)，如果是，那么this绑定的就是指定的对象。
3. 函数是否在某个上下文对象中调用(隐式绑定)，如果是的话，this 绑定的是那个上下文对象。一般是 obj.foo()
4. 如果以上都不是，那么使用默认绑定。如果在严格模式下，则绑定到 undefined，否则绑定到全局对象。
5. 如果把 null 或者 undefined 作为 this 的绑定对象传入 call、apply 或者 bind, 这些值在调用时会被忽略，实际应用的是默认绑定规则。
6. 箭头函数没有自己的 this, 它的this继承于上一层代码块的this。

测试下是否已经成功Get了此知识点(浏览器执行环境):

var number = 5;

var obj = {

number: 3,

fn1: (function () {

var number;

this.number \*= 2;

number = number \* 2;

number = 3;

return function () {

var num = this.number;

this.number \*= 2;

console.log(num);

number \*= 3;

console.log(number);

}

})()

}

var fn1 = obj.fn1;

fn1.call(null);

obj.fn1();

console.log(window.number);

**12. 词法作用域和this的区别。**

* 词法作用域是由你在写代码时将变量和块作用域写在哪里来决定的
* this 是在调用时被绑定的，this 指向什么，完全取决于函数的调用位置(关于this的指向问题，本文已经有说明)

**13. 谈谈你对JS执行上下文栈和作用域链的理解。**

执行上下文就是当前 JavaScript 代码被解析和执行时所在环境, JS执行上下文栈可以认为是一个存储函数调用的栈结构，遵循先进后出的原则。

* JavaScript执行在单线程上，所有的代码都是排队执行。
* 一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。
* 每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行-完成后，当前函数的执行上下文出栈，并等待垃圾回收。
* 浏览器的JS执行引擎总是访问栈顶的执行上下文。
* 全局上下文只有唯一的一个，它在浏览器关闭时出栈。

作用域链: 无论是 LHS 还是 RHS 查询，都会在当前的作用域开始查找，如果没有找到，就会向上级作用域继续查找目标标识符，每次上升一个作用域，一直到全局作用域为止。

**题难不难？不难！继续挑战一下！难！知道难，就更要继续了！**

**14. 什么是闭包？闭包的作用是什么？闭包有哪些使用场景？**

闭包是指有权访问另一个函数作用域中的变量的函数，创建闭包最常用的方式就是在一个函数内部创建另一个函数。

闭包的作用有:

1. 封装私有变量
2. 模仿块级作用域(ES5中没有块级作用域)
3. 实现JS的模块

**15. call、apply有什么区别？call,aplly和bind的内部是如何实现的？**

call 和 apply 的功能相同，区别在于传参的方式不一样:

* fn.call(obj, arg1, arg2, ...),调用一个函数, 具有一个指定的this值和分别地提供的参数(参数的列表)。
* fn.apply(obj, [argsArray]),调用一个函数，具有一个指定的this值，以及作为一个数组（或类数组对象）提供的参数。

call核心:

* 将函数设为传入参数的属性
* 指定this到函数并传入给定参数执行函数
* 如果不传入参数或者参数为null，默认指向为 window / global
* 删除参数上的函数

Function.prototype.call = function (context) {

/\* 如果第一个参数传入的是 null 或者是 undefined, 那么指向this指向 window/global \*/

/\*如果第一个参数传入的不是null或者是undefined, 那么必须是一个对象 \*/

if (!context) {

//context为null或者是undefined

context = typeof window === 'undefined' ? global : window;

}

context.fn = this; //this指向的是当前的函数(Function的实例)

let rest = [...arguments].slice(1);

//获取除了this指向对象以外的参数, 空数组slice后返回的仍然是空数组

let result = context.fn(...rest); //隐式绑定,当前函数的this指向了context.

delete context.fn;

return result;

}

//测试代码

var foo = {

name: 'Selina'

}

var name = 'Chirs';

function bar(job, age) {

console.log(this.name);

console.log(job, age);

}

bar.call(foo, 'programmer', 20);

// Selina programmer 20

bar.call(null, 'teacher', 25);

// 浏览器环境: Chirs teacher 25; node 环境: undefined teacher 25

复制代码

**apply:**

apply的实现和call很类似，但是需要注意他们的参数是不一样的，apply的第二个参数是数组或类数组.

Function.prototype.apply = function (context, rest) {

if (!context) {

//context为null或者是undefined时,设置默认值

context = typeof window === 'undefined' ? global : window;

}

context.fn = this;

let result = context.fn(...rest);

delete context.fn;

return result;

}

var foo = {

name: 'Selina'

}

var name = 'Chirs';

function bar(job, age) {

console.log(this.name);

console.log(job, age);

}

bar.apply(foo, ['programmer', 20]);

// Selina programmer 20

bar.apply(null, ['teacher', 25]);

// 浏览器环境: Chirs programmer 20; node 环境: undefined teacher 25

**Bind:**

bind 和 call/apply 有一个很重要的区别，一个函数被 call/apply 的时候，会直接调用，但是 bind 会创建一个新函数。当这个新函数被调用时，bind() 的第一个参数将作为它运行时的 this，之后的一序列参数将会在传递的实参前传入作为它的参数。

Function.prototype.bind = function(context) {

if(typeof this !== "function"){

throw new TypeError("not a function");

}

let self = this;

let args = [...arguments].slice(1);

function Fn() {};

Fn.prototype = this.prototype;

let bound = function() {

let res = [...args, ...arguments]; //bind传递的参数和函数调用时传递的参数拼接

context = this instanceof Fn ? this : context || this;

return self.apply(context, res);

}

//原型链

bound.prototype = new Fn();

return bound;

}

var name = 'Jack';

function person(age, job, gender){

console.log(this.name , age, job, gender);

}

var Yve = {name : 'Yvette'};

let result = person.bind(Yve, 22, 'enginner')('female');

**16. new的原理是什么？通过new的方式创建对象和通过字面量创建有什么区别？**

new:

1. 创建一个新对象。
2. 这个新对象会被执行[[原型]]连接。
3. 将构造函数的作用域赋值给新对象，即this指向这个新对象.
4. 如果函数没有返回其他对象，那么new表达式中的函数调用会自动返回这个新对象。

function new(func) {

lat target = {};

target.\_\_proto\_\_ = func.prototype;

let res = func.call(target);

if (typeof(res) == "object" || typeof(res) == "function") {

return res;

}

return target;

}

字面量创建对象，不会调用 Object构造函数, 简洁且性能更好;

new Object() 方式创建对象本质上是方法调用，涉及到在proto链中遍历该方法，当找到该方法后，又会生产方法调用必须的 堆栈信息，方法调用结束后，还要释放该堆栈，性能不如字面量的方式。

通过对象字面量定义对象时，不会调用Object构造函数。

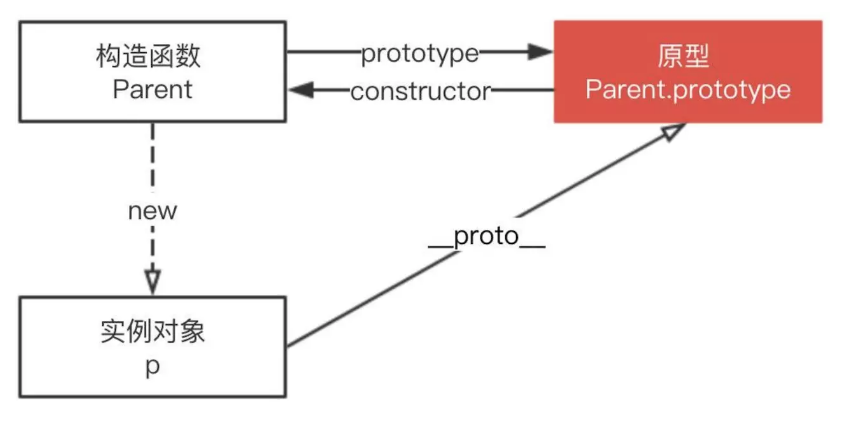
**17. 谈谈你对原型的理解？**

在 JavaScript 中，每当定义一个对象（函数也是对象）时候，对象中都会包含一些预定义的属性。其中每个函数对象都有一个prototype 属性，这个属性指向函数的原型对象。使用原型对象的好处是所有对象实例共享它所包含的属性和方法。

**18. 什么是原型链？【原型链解决的是什么问题？】**

原型链解决的主要是继承问题。

每个对象拥有一个原型对象，通过 **proto** (读音: dunder proto) 指针指向其原型对象，并从中继承方法和属性，同时原型对象也可能拥有原型，这样一层一层，最终指向 null(Object.proptotype.\_\_proto\_\_ 指向的是null)。这种关系被称为原型链 (prototype chain)，通过原型链一个对象可以拥有定义在其他对象中的属性和方法。

构造函数 Parent、Parent.prototype 和 实例 p 的关系如下:(p.\_\_proto\_\_ === Parent.prototype)

**19. prototype 和 \_\_proto\_\_ 区别是什么？**

prototype是构造函数的属性。

\_\_proto\_\_ 是每个实例都有的属性，可以访问 [[prototype]] 属性。

实例的\_\_proto\_\_ 与其构造函数的prototype指向的是同一个对象。

function Student(name) {

this.name = name;

}

Student.prototype.setAge = function(){

this.age=20;

}

let Jack = new Student('jack');

console.log(Jack.\_\_proto\_\_);

//console.log(Object.getPrototypeOf(Jack));;

console.log(Student.prototype);

console.log(Jack.\_\_proto\_\_ === Student.prototype);//true

**20. 使用ES5实现一个继承？**

组合继承(最常用的继承方式)

function SuperType() {

this.name = name;

this.colors = ['red', 'blue', 'green'];

}

SuperType.prototype.sayName = function() {

console.log(this.name);

}

function SubType(name, age) {

SuperType.call(this, name);

this.age = age;

}

SubType.prototype = new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function() {

console.log(this.age);

}

其它继承方式实现，可以参考《JavaScript高级程序设计》

**21. 什么是深拷贝？深拷贝和浅拷贝有什么区别？**

浅拷贝是指只复制第一层对象，但是当对象的属性是引用类型时，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。

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。

实现一个深拷贝:

function deepClone(obj) { //递归拷贝

if(obj === null) return null; //null 的情况

if(obj instanceof RegExp) return new RegExp(obj);

if(obj instanceof Date) return new Date(obj);

if(typeof obj !== 'object') {

//如果不是复杂数据类型，直接返回

return obj;

}

/\*\*

\* 如果obj是数组，那么 obj.constructor 是 [Function: Array]

\* 如果obj是对象，那么 obj.constructor 是 [Function: Object]

\*/

let t = new obj.constructor();

for(let key in obj) {

//如果 obj[key] 是复杂数据类型，递归

t[key] = deepClone(obj[key]);

}

return t;

}

**看不下去了？别人的送分题会成为你的送命题**

**22. 防抖和节流的区别是什么？防抖和节流的实现。**

防抖和节流的作用都是防止函数多次调用。区别在于，假设一个用户一直触发这个函数，且每次触发函数的间隔小于设置的时间，防抖的情况下只会调用一次，而节流的情况会每隔一定时间调用一次函数。

防抖(debounce): n秒内函数只会执行一次，如果n秒内高频事件再次被触发，则重新计算时间

function debounce(func, wait, immediate=true) {

let timeout, context, args;

// 延迟执行函数

const later = () => setTimeout(() => {

// 延迟函数执行完毕，清空定时器

timeout = null

// 延迟执行的情况下，函数会在延迟函数中执行

// 使用到之前缓存的参数和上下文

if (!immediate) {

func.apply(context, args);

context = args = null;

}

}, wait);

let debounced = function (...params) {

if (!timeout) {

timeout = later();

if (immediate) {

//立即执行

func.apply(this, params);

} else {

//闭包

context = this;

args = params;

}

} else {

clearTimeout(timeout);

timeout = later();

}

}

debounced.cancel = function () {

clearTimeout(timeout);

timeout = null;

};

return debounced;

};

**防抖的应用场景:**

* 每次 resize/scroll 触发统计事件
* 文本输入的验证（连续输入文字后发送 AJAX 请求进行验证，验证一次就好）

节流(throttle): 高频事件在规定时间内只会执行一次，执行一次后，只有大于设定的执行周期后才会执行第二次。

//underscore.js

function throttle(func, wait, options) {

var timeout, context, args, result;

var previous = 0;

if (!options) options = {};

var later = function () {

previous = options.leading === false ? 0 : Date.now() || new Date().getTime();

timeout = null;

result = func.apply(context, args);

if (!timeout) context = args = null;

};

var throttled = function () {

var now = Date.now() || new Date().getTime();

if (!previous && options.leading === false) previous = now;

var remaining = wait - (now - previous);

context = this;

args = arguments;

if (remaining <= 0 || remaining > wait) {

if (timeout) {

clearTimeout(timeout);

timeout = null;

}

previous = now;

result = func.apply(context, args);

if (!timeout) context = args = null;

} else if (!timeout && options.trailing !== false) {

// 判断是否设置了定时器和 trailing

timeout = setTimeout(later, remaining);

}

return result;

};

throttled.cancel = function () {

clearTimeout(timeout);

previous = 0;

timeout = context = args = null;

};

return throttled;

};

**函数节流的应用场景有:**

* DOM 元素的拖拽功能实现（mousemove）
* 射击游戏的 mousedown/keydown 事件（单位时间只能发射一颗子弹）
* 计算鼠标移动的距离（mousemove）
* Canvas 模拟画板功能（mousemove）
* 搜索联想（keyup）
* 监听滚动事件判断是否到页面底部自动加载更多：给 scroll 加了 debounce 后，只有用户停止滚动后，才会判断是否到了页面底部；如果是 throttle 的话，只要页面滚动就会间隔一段时间判断一次

**23. 取数组的最大值（ES5、ES6）**

// ES5 的写法

Math.max.apply(null, [14, 3, 77, 30]);

// ES6 的写法

Math.max(...[14, 3, 77, 30]);

// reduce

[14,3,77,30].reduce((accumulator, currentValue)=>{

return accumulator = accumulator > currentValue ? accumulator : currentValue

});

**24. ES6新的特性有哪些？**

1. 新增了块级作用域(let,const)
2. 提供了定义类的语法糖(class)
3. 新增了一种基本数据类型(Symbol)
4. 新增了变量的解构赋值
5. 函数参数允许设置默认值，引入了rest参数，新增了箭头函数
6. 数组新增了一些API，如 isArray / from / of 方法;数组实例新增了 entries()，keys() 和 values() 等方法
7. 对象和数组新增了扩展运算符
8. ES6 新增了模块化(import/export)
9. ES6 新增了 Set 和 Map 数据结构
10. ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例
11. ES6 新增了生成器(Generator)和遍历器(Iterator)

**25. setTimeout倒计时为什么会出现误差？**

setTimeout() 只是将事件插入了“任务队列”，必须等当前代码（执行栈）执行完，主线程才会去执行它指定的回调函数。要是当前代码消耗时间很长，也有可能要等很久，所以并没办法保证回调函数一定会在 setTimeout() 指定的时间执行。所以， setTimeout() 的第二个参数表示的是最少时间，并非是确切时间。

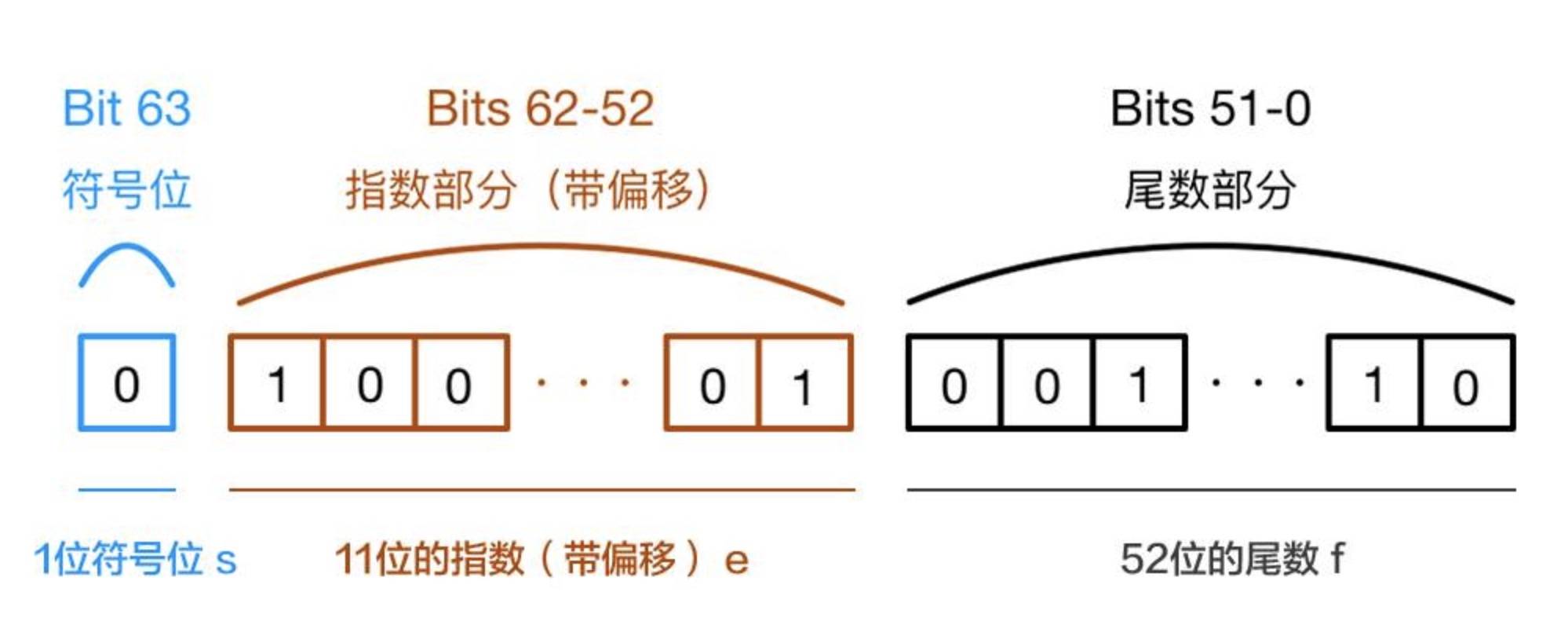
HTML5标准规定了 setTimeout() 的第二个参数的最小值不得小于4毫秒，如果低于这个值，则默认是4毫秒。在此之前。老版本的浏览器都将最短时间设为10毫秒。另外，对于那些DOM的变动（尤其是涉及页面重新渲染的部分），通常是间隔16毫秒执行。这时使用 requestAnimationFrame() 的效果要好于 setTimeout();

**26. 为什么 0.1 + 0.2 != 0.3 ?**

0.1 + 0.2 != 0.3 是因为在进制转换和进阶运算的过程中出现精度损失。

下面是详细解释:

JavaScript使用 Number 类型表示数字(整数和浮点数)，使用64位表示一个数字。



图片说明:

* 第0位：符号位，0表示正数，1表示负数(s)
* 第1位到第11位：储存指数部分（e）
* 第12位到第63位：储存小数部分（即有效数字）f

计算机无法直接对十进制的数字进行运算, 需要先对照 IEEE 754 规范转换成二进制，然后对阶运算。

1.进制转换

0.1和0.2转换成二进制后会无限循环

0.1 -> 0.0001100110011001...(无限循环)

0.2 -> 0.0011001100110011...(无限循环)

复制代码

但是由于IEEE 754尾数位数限制，需要将后面多余的位截掉，这样在进制之间的转换中精度已经损失。

2.对阶运算

由于指数位数不相同，运算时需要对阶运算 这部分也可能产生精度损失。

按照上面两步运算（包括两步的精度损失），最后的结果是

0.0100110011001100110011001100110011001100110011001100

结果转换成十进制之后就是 0.30000000000000004。

**27. promise 有几种状态, Promise 有什么优缺点 ?**

promise有三种状态: fulfilled, rejected, pending.

Promise 的优点:

1. 一旦状态改变，就不会再变，任何时候都可以得到这个结果
2. 可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数

Promise 的缺点:

1. 无法取消 Promise
2. 当处于pending状态时，无法得知目前进展到哪一个阶段

**28. Promise构造函数是同步还是异步执行，then呢 ?promise如何实现then处理 ?**

Promise的构造函数是同步执行的。then 是异步执行的。

promise的then实现，详见: [Promise源码实现](https://juejin.im/post/5c88e427f265da2d8d6a1c84)

**29. Promise和setTimeout的区别 ?**

Promise 是微任务，setTimeout 是宏任务，同一个事件循环中，promise.then总是先于 setTimeout 执行。

**30. 如何实现 Promise.all ?**

要实现 Promise.all,首先我们需要知道 Promise.all 的功能：

1. 如果传入的参数是一个空的可迭代对象，那么此promise对象回调完成(resolve),只有此情况，是同步执行的，其它都是异步返回的。
2. 如果传入的参数不包含任何 promise，则返回一个异步完成. promises 中所有的promise都“完成”时或参数中不包含 promise 时回调完成。
3. 如果参数中有一个promise失败，那么Promise.all返回的promise对象失败
4. 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 = function (promises) {

return new Promise((resolve, reject) => {

let index = 0;

let result = [];

if (promises.length === 0) {

resolve(result);

} else {

setTimeout(() => {

function processValue(i, data) {

result[i] = data;

if (++index === promises.length) {

resolve(result);

}

}

for (let i = 0; i < promises.length; i++) {

//promises[i] 可能是普通值

Promise.resolve(promises[i]).then((data) => {

processValue(i, data);

}, (err) => {

reject(err);

return;

});

}

})

}

});

}

**31.如何实现 Promise.finally ?**

不管成功还是失败，都会走到finally中,并且finally之后，还可以继续then。并且会将值原封不动的传递给后面的then.

Promise.prototype.finally = function (callback) {

return this.then((value) => {

return Promise.resolve(callback()).then(() => {

return value;

});

}, (err) => {

return Promise.resolve(callback()).then(() => {

throw err;

});

});

}

**32. 什么是函数柯里化？实现 sum(1)(2)(3) 返回结果是1,2,3之和**

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。

function sum(a) {

return function(b) {

return function(c) {

return a+b+c;

}

}

}

console.log(sum(1)(2)(3)); // 6

复制代码

引申：实现一个curry函数，将普通函数进行柯里化:

function curry(fn, args = []) {

return function(){

let rest = [...args, ...arguments];

if (rest.length < fn.length) {

return curry.call(this,fn,rest);

}else{

return fn.apply(this,rest);

}

}

}

//test

function sum(a,b,c) {

return a+b+c;

}

let sumFn = curry(sum);

console.log(sumFn(1)(2)(3)); //6

console.log(sumFn(1)(2, 3)); //6

如果您在面试中遇到了更多的原生JS问题，或者有一些本文未涉及到且有一定难度的JS知识，请给我留言。您的问题将会出现在后续文章中~

作者：前端小姐姐  
链接：https://juejin.im/post/5cab0c45f265da2513734390  
来源：掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。