Js 常见手写题

主要的内容有：

防抖

节流

new

bind

call

apply

deepCopy

es5 实现继承

instanceof

promise

数组扁平化

对象扁平化

图片懒加载

EventEmitter(发布订阅模式)

函数柯里化

异步并发数限制

异步串行 | 异步并行

1. **防抖**

防抖：触发事件后在n秒后，函数只能执行一次；若在n秒内又触发了事件，则会计算函数执行时间。

频繁触发

let num = 1;

let content = document.getElementById('content');

function count() {

content.innerHTML = num++;

};

content.onmousemove = count

防抖函数分为‘非立即执行版’和‘立即执行版’。

1. 非立即执行版

触发事件后函数不会立即执行，而是在 n 秒后执行，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

function debounce(func, wait){

let timer

return function(){

let context = this

let args = arguments

if(timer) clearTimeout(timer)

timer = setTimeout(()=>{

func.apply(context, args)

}, wait)

}

}

content.onmousemove = debounce(count,1000);

2. 立即执行版

触发事件后函数会立即执行，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

function debounce(func, wait){

let timer

return function(){

let context = this

let args = arguments

if(timer) clearTimeout(timer)

let timer = setTimeout(()=>{

timer = null

}, wait)

if(!timer) func.apply(context, args)

}

}

3. 双剑合并版

/\*\*

\* @desc 函数防抖

\* @param func 函数

\* @param wait 延迟执行毫秒数

\* @param immediate true 表立即执行，false 表非立即执行

\*/

function debounce(func,wait,immediate) {

let timeout;

return function () {

let context = this;

let args = arguments;

if (timeout) clearTimeout(timeout);

if (immediate) {

timeout = setTimeout(() => {

timeout = null;

}, wait)

if (!timeout) func.apply(context, args)

}

else {

timeout = setTimeout(function(){

func.apply(context, args)

}, wait);

}

}

}

**二、节流**

节流：连续触发事件在n秒内只执行一次，即会稀释函数的执行频率。 关于节流的实现，有两种主流的实现方式，一种是使用时间戳，一种是设置定时器。

备注：可以先看看‘防抖‘的实现。

1. 时间戳

时间戳版的函数触发是在'时间段内开始'的时候;

function throttle(func, wait){

let previous = 0

return function(){

let timer = Date.now()

let context = this

let args = arguments

if(timer - previous > wait){

func.apply(context, args)

previous = Date.now()

}

}

}

2. 定时器

定时器版的函数触发是在'时间段内结束'的时候.

function throttle(func, wait){

let timer

return function(){

let context = this

let args = arguments

if(!timer){

timer = setTimeout(()=>{

timer = null;

func.apply(context, args)

}, wait)

}

}

}

**三、new的模拟实现**

new 创建了一个用户定义的对象类型的实例 或 具有构造函数的内置对象类型之一。

**new 实现的功能:**

（1）在堆内存中创建一个新的对象

（2）该新对象内部的[[Prototype]]指针被赋值为构造函数的 prototype 属性

（3）将构造函数内的this 被赋值为 这个新对象

（4）逐个执行函数中的代码（给新对象添加属性等）

（5）如果构造函数返回非空对象（是对象类型，且非空），则返回该对象；否则，将新建的对象作为返回值

**模拟实现 new 操作符**

因为 new 是关键字，所以无法像 bind 函数一样直接覆盖，所以我们写一个函数，命名为 objectFactory，来模拟 new 的效果。用的时候是这样的：

function Otaku () {

……

}

// 使用 new

var person = new Otaku(……);

// 使用 objectFactory，模拟new

var person = objectFactory(Otaku, ……)

**1. 初步实现**

分析：

new 的结果是一个新对象，因此在模拟实现时，需要建立一个新对象 obj；

obj 会具有 Otaku 构造函数里的属性，想想经典继承的例子，我们可以使用 Otaku.apply(obj, arguments)来给 obj 添加新的属性。

// 第一版代码

function objectFactory(Otaku, ...) {

var obj = new Object(), // 创建新对象

Constructor = [].shift.call(arguments); // 取出构造函数（第一个参数），另外获取构造函数传入的参数

obj.\_\_proto\_\_ = Constructor.prototype; // 将新对象内部的 \_proto\_ 指针指向构造函数的prototype

Constructor.apply(obj, arguments); // 构造函数绑定 新对象作为this，及参数

return obj; // 返回新对象

};

将上述代码复制到浏览器中运行，验证没问题，撒花！

**2. 返回值效果实现**

构造函数在返回时，会判断返回值是否是对象：

若返回值是非空对象（是对象类型，且非空），则将构造函数返回值 返回即可；

否则（没有返回值，或返回值是 原始值 或 空对象），将新建的对象作为返回值。

考虑返回值的情况：

// 第二版的代码

function objectFactory(Otaku, ...) {

var obj = new Object(),

Constructor = [].shift.call(arguments);

obj.\_\_proto\_\_ = Constructor.prototype;

var ret = Constructor.apply(obj, arguments); // 获取构造函数的返回值

return typeof ret === 'object' ? ret : obj; // 依据返回值类型，分别返回

};

1. **bind 模拟实现**

bind 函数的三个特点：

（1）返回一个函数

（2）可以传入参数

（3）bind 后返回的函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。bind传入的 this 值被忽略，同时调用时的参数被提供给模拟函数。

**bind功能举例：**

var foo = {

value: 1

};

function bar() {

console.log(this.value);

}

var bindFoo = bar.bind(foo); // 返回了一个函数

bindFoo(); // 1

**一、返回函数的模拟实现**

// 第一版

Function.prototype.bind2 = function (context) { // context 为函数要绑定的对象

var self = this; // this 为调用 bind 的函数

return function () {

return self.apply(context);

}

}

**二、传参的模拟实现**

疑问：我在 bind 的时候，是否可以传参呢？我在执行 bind 返回的函数的时候，可不可以传参呢？

var foo = {

value: 1

};

function bar(name, age) {

console.log('val: ',this.value);

console.log('name:', name);

console.log('age:', age);

}

var bindFoo = bar.bind(foo, 'daisy');

bindFoo('18');

// val: 1

// name: daisy

// age: 18

函数需要传 name 和 age 两个参数，竟然还可以在 bind 的时候，只传一个 name，在执行返回的函数的时候，再传另一个参数 age!

这可咋办？不急，我们用 arguments 进行处理：

// 第二版

Function.prototype.bind2 = function(context){

let self = this;

// 获取 bind2 函数从第二个参数到最后一个参数

let args = Array.prototype.slice.call(arguments, 1)

return function(){

// 这个时候的arguments是指 bind 后返回函数传入的参数

let afterBindArgs = Array.prototype.slice.call(arguments)

return self.apply(context, args.concat(afterBindArgs))

}

}

**三、调用 bind 后创建的新函数作为构造函数**

bind 后返回的函数也能使用new操作符创建对象： 即 bind 返回的函数作为构造函数的时候，bind 指定的 this 值会失效，即该this相关的属性都获取不到，但传入的参数依然生效。

举个例子：

var value = 2;

var foo = {

value: 1

};

function bar(name, age) {

this.habit = 'shopping';

console.log('val: ', this.value);

console.log('name: ', name);

console.log('age:', age);

}

bar.prototype.friend = 'kevin';

var bindFoo = bar.bind(foo, 'daisy');

var obj = new bindFoo('18');

// val: undefined

// name: daisy

// age: 18

console.log('obj.habit: ', obj.habit);

console.log('obj.friend: ', obj.friend);

// obj.habit: shopping

// obj.friend: kevin

注意：尽管在全局和 foo 中都声明了 value 值，最后依然返回了 undefind，说明绑定的 this 失效了，如果大家了解 new 的模拟实现，就会知道这个时候的 this 已经指向了 obj。具体可看：new 的模拟实现。

因此可通过修改返回函数的原型来实现：

// 第三版

Function.prototype.bind2 = function (context) {

// 若调用 bind 的不是函数，则直接报错

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

throw new Error("Function.prototype.bind - what is trying to be bound is not callable");

}

var self = this;

var args = Array.prototype.slice.call(arguments, 1);

var fBound = function () {

var bindArgs = Array.prototype.slice.call(arguments);

// 调用 bind 后创建的新函数绑定this：

// 1. 若不用做构造函数：直接绑定 context

// 2. 若用做构造函数时：this 指向实例，不应该绑定在 context上，直接绑定在实例上

return self.apply(this instanceof fBound ? this : context, args.concat(bindArgs));

}

fBound.prototype = this.prototype; //this为调用 bind 的函数

return fBound;

}

1. **call 模拟实现**

首先，看下call, apply 实现了哪些功能：

var foo = {

value: 1

};

function bar() {

console.log(this.value);

}

bar.call(foo); // 1

注意两点：

（1）call 改变了 this 的指向，指向到 foo

（2）bar 函数执行了

**1. 模拟实现第一步**

试想当调用 call 的时候，把 foo 对象改造成如下：

var foo = {

value: 1,

bar: function() {

console.log(this.value)

}

};

foo.bar(); // 1

这个时候 this 就指向了 foo，是不是很简单呢？但是这样却给 foo 对象本身添加了一个属性，这可不行呐！不过也不用担心，我们用 delete 再删除它不就好了~

所以我们模拟的步骤可以分为：

（1）将函数设为对象的属性

（2）执行该函数

（3）删除该函数

依据这个思路，我们初步实现第一版：

// 第一版

Function.prototype.call2 = function(context) {

// 首先要获取调用call的函数，用this可以获取

context.fn = this;

context.fn();

delete context.fn;

}

**2. 模拟实现第二步（考虑参数）**

// 第二版

Function.prototype.call2 = function(context) {

context.fn = this;

// 获取参数的两个方法

// 法1：

// let args = Array.prototype.slice.call(arguments, 1)

// context.fn(...args) // 没问题

// 法2：

var args = [];

for(var i = 1, len = arguments.length; i < len; i++) {

args.push('arguments[' + i + ']');

}

eval('context.fn(' + args +')');

delete context.fn;

}

**3. 模拟实现第三步（细节完善）**

（1）this 参数可以传 null、undefined，此时，this指向 window;

（2）函数是可以有返回值的

// 第三版

Function.prototype.call2 = function (context) {

var context = context || window; // this 为null、undefined

context.fn = this;

// 获取参数的两个方法

// 法1：

// let args = Array.prototype.slice.call(arguments, 1)

// context.fn(...args) // 没问题

// 法2：

var args = [];

for(var i = 1, len = arguments.length; i < len; i++) {

args.push('arguments[' + i + ']');

}

var result = eval('context.fn(' + args +')');

delete context.fn

return result; // 返回值

}

**六、apply 模拟实现**

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

var context = Object(context) || window;

context.fn = this;

var result;

if (!arr) {

result = context.fn();

}

else {

var args = [];

for (var i = 0, len = arr.length; i < len; i++) {

args.push('arr[' + i + ']');

}

result = eval('context.fn(' + args + ')')

}

delete context.fn

return result;

}

**七、数组扁平化**

function flatter(arr){

if(!arr.length) return;

while(arr.some((item) => Array.isArray(item))){

arr = [].concat(...arr)

}

return arr

}

// let b = [1,[2],3]

// console.log('...', ...b)

flatter([1, 2, [1, [2, 3, [4, 5, [6]]]]])

八、对象扁平化