**JS常见手写题**

[一、防抖 2](#_Toc854181639)

[二、节流 4](#_Toc302626478)

[三、new的模拟实现 5](#_Toc1001939650)

[四、bind 模拟实现 7](#_Toc1180421423)

[五、call 模拟实现 10](#_Toc888925375)

[六、apply 模拟实现 12](#_Toc125045446)

[七、数组扁平化 13](#_Toc1399804156)

[八、对象扁平化 14](#_Toc825097007)

[九、手写Promise 15](#_Toc1103487970)

[十、手写发布/订阅 EventEmitter 18](#_Toc653536298)

[十一、instanceOf 实现 19](#_Toc1753189728)

[十二、深拷贝 20](#_Toc236638009)

[十三、JS继承 23](#_Toc35303019)

[十四、图片懒加载 26](#_Toc632353761)

[十五、函数柯里化 28](#_Toc73092124)

[十六、异步并发数限制 29](#_Toc98681984)

[十七、异步串行/异步并行 30](#_Toc690729604)

# 一、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

};

# 四、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;

}

# 五、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

**1、调用 ES6 中的 flat 方法**

ary = arr.flat(Infinity)

console.log([1, [2, 3, [4, 5, [6, 7]]]].flat(Infinity))

**2、普通递归**

let result = []

let flatten = function (arr) {

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

let item = arr[i]

if (Array.isArray(arr[i])) {

flatten(item)

} else {

result.push(item)

}

}

return result

}

let arr = [1, 2, [3, 4], [5, [6, 7]]]

console.log(flatten(arr))

**3、利用 reduce 函数迭代**

function flatten(arr) {

return arr.reduce((pre, cur) => {

return pre.concat(Array.isArray(cur) ? flatten(cur) : cur)

}, [])

}

let arr = [1, 2, [3, 4], [5, [6, 7]]]

console.log(flatten(arr))

**4、扩展运算符**

function flatten(arr) {

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

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

}

return arr

}

let arr = [1, 2, [3, 4], [5, [6, 7]]]

console.log(flatten(arr))

# 八、对象扁平化

/\* 题目\*/

var entryObj = {

a: {

b: {

c: {

dd: 'abcdd'

}

},

d: {

xx: 'adxx'

},

e: 'ae'

}

}

// 要求转换成如下对象

var outputObj = {

'a.b.c.dd': 'abcdd',

'a.d.xx': 'adxx',

'a.e': 'ae'

}

**// 手写实现**

function objectFlat(obj = {}) {

const res = {}

function flat(item, preKey = '') {

Object.entries(item).forEach(([key, val]) => {

const newKey = preKey ? `${preKey}.${key}` : key

if (val && typeof val === 'object') {

flat(val, newKey)

} else {

res[newKey] = val

}

})

}

flat(obj)

return res

}

// 测试

const source = { a: { b: { c: 1, d: 2 }, e: 3 }, f: { g: 2 } }

console.log(objectFlat(source));

# 九、手写Promise

class Mypromise {

constructor(fn) {

this.state = "pending";

this.successFun = [];

this.failFun = [];

let resolve = val => {

// 保持状态改变不可变（resolve和reject只准触发一种）

if (this.state !== "pending") return;

this.state = "success";

setTimeout(() => {

// 执行当前事件里面所有的注册函数

this.successFun.forEach(item => item.call(this, val));

});

};

let reject = err => {

if (this.state !== "pending") return;

this.state = "fail";

setTimeout(() => {

this.failFun.forEach(item => item.call(this, err));

});

};

try {

fn(resolve, reject);

} catch (error) {

reject(error);

}

}

then(resolveCallback, rejectCallback) {

// 判断回调是否是函数

resolveCallback = typeof resolveCallback !== "function" ? v => v : resolveCallback;

rejectCallback == typeof rejectCallback !== "function"

? err => {

throw err;

}

: rejectCallback;

// 为了保持链式调用 继续返回promise

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

// 将回调注册到successFun事件集合里面去

this.successFun.push(val => {

try {

let x = resolveCallback(val); // 执行回调函数

// 如果回调函数结果是普通值

// 如果回调函数结果是一个promise对象

x instanceof Mypromise ? x.then(resolve, reject) : resolve(x);

} catch (error) {

reject(error);

}

});

this.failFun.push(val => {

try {

let x = rejectCallback(val);

x instanceof Mypromise ? x.then(resolve, reject) : reject(x);

} catch (error) {

reject(error);

}

});

});

}

//静态方法

static all(promiseArr) {

let result = [];

let count = 0 //声明一个计数器 每一个promise返回就+1

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

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

promiseArr[i].then(

res => {

//这里不能直接push数组 因为要控制顺序一一对应(感谢评论区指正)

result[i] = res

count++

//只有全部的promise执行成功之后才resolve出去

if (count === promiseArr.length) {

resolve(result);

}

},

err => {

reject(err);

}

);

}

});

}

//静态方法

static race(promiseArr) {

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

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

promiseArr[i].then(

res => {

//promise数组只要有任何一个promise 状态变更 就可以返回

resolve(res);

},

err => {

reject(err);

}

);

}

});

}

}

// 使用

let promise1 = new Mypromise((resolve, reject) => {

setTimeout(() => {

resolve(123);

}, 2000);

});

promise1

.then(

res => {

console.log(res); //过两秒输出123

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

setTimeout(() => {

resolve("success");

}, 1000);

});

},

err => {

console.log(err);

}

)

.then(

res => {

console.log(res); //再过一秒输出success

},

err => {

console.log(err);

}

);

let promise2 = new Mypromise((resolve, reject) => {

setTimeout(() => {

resolve(1234);

}, 1000);

});

Mypromise.all([promise1,promise2]).then(res=>{

console.log(res);

})

Mypromise.race([promise1, promise2]).then(res => {

console.log(res);

});

# 十、手写发布/订阅 EventEmitter

**class EventEmitter** {

constructor() {

this.events = {};

}

**on(type, callBack)** { // 实现订阅

if (!this.events)

this.events = Object.create(null);

if (!this.events[type]) {

this.events[type] = [callBack];

} else {

this.events[type].push(callBack);

}

}

**off(type, callBack)** { // 删除订阅

if (!this.events[type]) return;

this.events[type] = this.events[type].filter(item => {

return item !== callBack;

});

}

**once(type, callBack)** { // 只执行一次订阅事件

function fn() {

callBack();

this.off(type, fn);

}

this.on(type, fn);

}

**emit(type, ...rest)** { // 触发事件

this.events[type] && this.events[type].forEach(fn => fn.apply(this, rest));

}

}

// 使用如下

const event = new EventEmitter();

const handle = (...rest) => {

console.log(rest);

};

event.on("click", handle);

event.emit("click", 1, 2, 3, 4);

event.off("click", handle);

event.emit("click", 1, 2);

event.once("dbClick", () => {

console.log(123456);

});

event.emit("dbClick");

event.emit("dbClick");

# 十一、instanceOf 实现

instanceof 主要的作用就是**判断一个实例是否属于某种类型**，也可以判断一个实例是否是其**父类型、祖先类型的实例**。

function new\_instance\_of(leftVaule, rightVaule) {

let rightProto = rightVaule.prototype; // 取右表达式的 prototype 值

leftVaule = leftVaule.\_\_proto\_\_; // 取左表达式的\_\_proto\_\_值

while (true) {

if (leftVaule === null) {

return false;

}

if (leftVaule === rightProto) {

return true;

}

leftVaule = leftVaule.\_\_proto\_\_

}

}

# 十二、深拷贝

const mapTag = '[object Map]';

const setTag = '[object Set]';

const arrayTag = '[object Array]';

const objectTag = '[object Object]';

const argsTag = '[object Arguments]';

const boolTag = '[object Boolean]';

const dateTag = '[object Date]';

const numberTag = '[object Number]';

const stringTag = '[object String]';

const symbolTag = '[object Symbol]';

const errorTag = '[object Error]';

const regexpTag = '[object RegExp]';

const funcTag = '[object Function]';

const deepTag = [mapTag, setTag, arrayTag, objectTag, argsTag];

function forEach(array, iteratee) {

let index = -1;

const length = array.length;

while (++index < length) {

iteratee(array[index], index);

}

return array;

}

function isObject(target) {

const type = typeof target;

return target !== null && (type === 'object' || type === 'function');

}

function getType(target) {

return Object.prototype.toString.call(target);

}

function getInit(target) {

const Ctor = target.constructor;

return new Ctor();

}

function cloneSymbol(targe) {

return Object(Symbol.prototype.valueOf.call(targe));

}

function cloneReg(targe) {

const reFlags = /\w\*$/;

const result = new targe.constructor(targe.source, reFlags.exec(targe));

result.lastIndex = targe.lastIndex;

return result;

}

function cloneFunction(func) {

const bodyReg = /(?<={)(.|\n)+(?=})/m;

const paramReg = /(?<=\().+(?=\)\s+{)/;

const funcString = func.toString();

if (func.prototype) {

const param = paramReg.exec(funcString);

const body = bodyReg.exec(funcString);

if (body) {

if (param) {

const paramArr = param[0].split(',');

return new Function(...paramArr, body[0]);

} else {

return new Function(body[0]);

}

} else {

return null;

}

} else {

return eval(funcString);

}

}

function cloneOtherType(targe, type) {

const Ctor = targe.constructor;

switch (type) {

case boolTag:

case numberTag:

case stringTag:

case errorTag:

case dateTag:

return new Ctor(targe);

case regexpTag:

return cloneReg(targe);

case symbolTag:

return cloneSymbol(targe);

case funcTag:

return cloneFunction(targe);

default:

return null;

}

}

function clone(target, map = new WeakMap()) {

// 克隆原始类型

if (!isObject(target)) {

return target;

}

// 初始化

const type = getType(target);

let cloneTarget;

if (deepTag.includes(type)) {

cloneTarget = getInit(target, type);

} else {

return cloneOtherType(target, type);

}

// 防止循环引用

if (map.get(target)) {

return map.get(target);

}

map.set(target, cloneTarget);

// 克隆set

if (type === setTag) {

target.forEach(value => {

cloneTarget.add(clone(value, map));

});

return cloneTarget;

}

// 克隆map

if (type === mapTag) {

target.forEach((value, key) => {

cloneTarget.set(key, clone(value, map));

});

return cloneTarget;

}

// 克隆对象和数组

const keys = type === arrayTag ? undefined : Object.keys(target);

forEach(keys || target, (value, key) => {

if (keys) {

key = value;

}

cloneTarget[key] = clone(target[key], map);

});

return cloneTarget;

}

module.exports = {

clone

};

# 十三、JS继承

继承方法有：原型链、盗用构造函数、组合继承、原型式继承、寄生式继承、寄生组合继承。

**1. 原型链**

**目标：**基本思想就是通过原型继承多个引用类型的属性和方法。

**关键实现步骤：**把原型作为另一个类型的实例。

function SuperType() {

this.property = true;

}

SuperType.prototype.getSuperValue = function() {

return this.property;

};

function SubType() {

this.subproperty = false;

}

// 继承 SuperType

SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function () {

return this.subproperty;

};

let instance = new SubType();

console.log(instance.getSuperValue()); // true，调用父元素方法

**原型链的问题：**

（1）原型中包含的引用值会在所有实例间共享。

（2）子类型在实例化时 不能给父类型的构造函数传参。

**2. 盗用构造函数**

**目标：**解决原型包含引用值导致的继承问题，以及子类示例化时不能向父类构造函数传参问题。

**关键实现步骤：**在子类构造函数中调用父类构造函数。

function SuperType(name){

this.name = name;

}

function SubType() {

// 继承 SuperType 并传参

SuperType.call(this, "Nicholas");

// 实例属性

this.age = 29;

}

let instance = new SubType();

console.log(instance.name); // "Nicholas";

console.log(instance.age); // 29

**盗用构造函数问题：**

（1）必须在构造函数中定义方法，因此函数不能重用。（这也是使用构造函数模式自定义类型的问题）

（2）子类也不能访问父类原型上定义的方法，因此所有类型只能使用构造函数模式。

**3. 组合继承**

组合继承 也称 ‘伪经典继承’，综合了 原型链 和 盗用构造函数，将两者的优点集中了起来。是 JavaScript 中使用最多的继承模式。

**基本思路：**使用原型链 继承 原型上的属性和方法，而通过 盗用构造函数 继承 实例属性。 这样既可以把方 法定义在原型上以实现重用，又可以让每个实例都有自己的属性。

**缺点：**存在效率问题。最主要的效率问题就是父类构造函数始终会被调用两次。

function SuperType(name){

this.name = name;

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

}

SuperType.prototype.sayName = function() {

console.log(this.name);

};

function SubType(name, age){

// 继承属性

SuperType.call(this, name); // 第二次调用 SuperType()

this.age = age;

}

// 继承方法

SubType.prototype = new SuperType(); // 第一次调用 SuperType()

SubType.prototype.sayAge = function() {

console.log(this.age);

};

let instance1 = new SubType("Nicholas", 29);

instance1.colors.push("black");

console.log(instance1.colors); // "red,blue,green,black"

instance1.sayName(); // "Nicholas";

instance1.sayAge(); // 29

let instance2 = new SubType("Greg", 27);

console.log(instance2.colors); // "red,blue,green"

instance2.sayName(); // "Greg";

instance2.sayAge(); // 27

**4. 原型式继承**

**目标：**即使不自定义类型 也可以通过原型实现对象之间的信息共享。

原型式继承非常适合不需要单独创建构造函数，但仍然需要在对象间共享信息的场合。但要记住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的。

function object(o) {

function F() {}

F.prototype = o;

return new F();

}

let person = { name: "Nicholas",

friends: ["Shelby", "Court", "Van"] };

let anotherPerson = object(person); // 调用上面封装的 object 函数

anotherPerson.name = "Greg";

anotherPerson.friends.push("Rob");

let yetAnotherPerson = object(person);

yetAnotherPerson.name = "Linda";

yetAnotherPerson.friends.push("Barbie");

console.log(person.friends); // "Shelby,Court,Van,Rob,Barbie"

**5. 寄生式继承**

寄生式继承（parasitic inheritance）是一种与原型式继承比较接近的一种继承方式。

寄生式继承 **存在的问题：**通过寄生式继承 给对象添加函数会导致函数难以重用，与构造函数模式类似。

function createAnother(original){

// object()函数不是寄生式继承所必需的，任何返回新对象的函数都可以在这里使用。

let clone = object(original); // 通过调用函数创建一个新对象

clone.sayHi = function() { // 以某种方式增强这个对象

console.log("hi");

};

return clone; // 返回这个对象

}

let person = { name: "Nicholas",

friends: ["Shelby", "Court", "Van"] };

let anotherPerson = createAnother(person);

anotherPerson.sayHi(); // "hi"

**6. 寄生组合式继承**

组合式继承存在的主要问题有：父类构造函数始终会执行 2 次。这样的话，就会有两组 name 和 colors 属性：一组在实例上，另一组在 SubType 的原型上。

寄生组合式继承 **关键实现步骤**：

（1）使用 寄生式继承 来 继承父类原型；（不是通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。）

（2）将返回的新对象 赋值给 子类原型。

寄生式组合继承基本是引用类型继承的最佳模式。

function inheritPrototype(subType, superType) {

let prototype = object(superType.prototype); // 是创建父类原型的一个副本

prototype.constructor = subType; // 给返回的prototype 对象设置 constructor 属性，解决由于重写原型导致默认 constructor 丢失的问题

subType.prototype = prototype; // 将新创建的对象 赋值 给子类型的原型

}

function SuperType(name) {

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;

}

inheritPrototype(SubType, SuperType); // 给子类型原型赋值

SubType.prototype.sayAge = function() {

console.log(this.age);

};

# 十四、图片懒加载

**方法1：**

// <img src="default.png" data-src="https://xxxx/real.png">

function isVisible(el) {

const position = el.getBoundingClientRect()

const windowHeight = document.documentElement.clientHeight

// 顶部边缘可见

const topVisible = position.top > 0 && position.top < windowHeight;

// 底部边缘可见

const bottomVisible = position.bottom < windowHeight && position.bottom > 0;

return topVisible || bottomVisible;

}

function imageLazyLoad() {

const images = document.querySelectorAll('img')

for (let img of images) {

const realSrc = img.dataset.src

if (!realSrc) continue

if (isVisible(img)) {

img.src = realSrc

img.dataset.src = ''

}

}

}

// 测试

window.addEventListener('load', imageLazyLoad)

window.addEventListener('scroll', imageLazyLoad)

// or

window.addEventListener('scroll', throttle(imageLazyLoad, 1000))

**方法2：**

export default class LazyLoad {

options: any;

elements: any;

constructor(options?: any) {

let \_this = this;

\_this.options = Object.assign({

srckey: 'data-src',

offset: 200

}, options);

\_this.bindEvent();

\_this.scrollHandler();

$(window).scroll($.proxy(\_this.scrollHandler, \_this));

}

bindEvent() {

let \_this = this;

if (window.addEventListener) {

window.addEventListener("scroll", \_this.scrollHandler.bind(\_this), false);

} else {

window.attachEvent("onscroll", \_this.scrollHandler.bind(\_this), false);

}

}

scrollHandler() {

let \_this = this;

\_this.elements = document.querySelectorAll(`img[${\_this.options.srckey}]`);

for (let i = 0; i < \_this.elements.length; i++) {

let element = \_this.elements[i];

let src = element.attributes[\_this.options.srckey].value;

let loaded = element.attributes['data-loaded'];

if (!loaded && src && \_this.inVisibleArea(element)) {

\_this.loadImage(element, src);

}

}

}

inVisibleArea(element) {

let \_this = this;

let viewHeight = \_this.getViewHeight();

let scrollTop = \_this.getScrollTop();

let elementOffsetTop = \_this.getOffsetTop(element);

let visible = elementOffsetTop < viewHeight + scrollTop + this.options.offset;

return visible;

}

loadImage(image, src) {

image.src = src;

const fnType = image.addEventListener ? 'addEventListener' : 'attachEvent';

image[fnType]("load", () => { image.setAttribute('data-loaded', true); });

image[fnType]("error", () => { image.setAttribute('data-loaded', true); image.src = '//pic6.58cdn.com.cn/nowater/fangfe/n\_v2d028f091415c4ba6baa828e2612d714b.png'; });

}

//屏幕可视高度

getViewHeight() {

// 标准浏览器及IE9+ || 标准浏览器及低版本IE标准模式 || 低版本混杂模式

return window.innerHeight || document.documentElement.clientHeight || document.body.clientHeight;

}

//滚动高度

getScrollTop() {

// 标准浏览器及IE9+ || 标准浏览器及低版本IE标准模式 || 低版本混杂模式

return window.pageYOffset || document.documentElement.scrollTop || document.body.scrollTop;

}

//元素位置

getOffsetTop(el) {

return el.offsetParent

? el.offsetTop + this.getOffsetTop(el.offsetParent)

: el.offsetTop

}

};

# 十五、函数柯里化

柯里化：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

function curry(func) {

return function curried(...args) {

// 关键知识点：function.length 用来获取函数的形参个数

// 补充：arguments.length 获取的是实参个数

if (args.length >= func.length) {

return func.apply(this, args)

}

return function (...args2) {

return curried.apply(this, args.concat(args2))

}

}

}

// 测试

function sum (a, b, c) {

return a + b + c

}

const curriedSum = curry(sum)

console.log(curriedSum(1, 2, 3))

console.log(curriedSum(1)(2,3))

console.log(curriedSum(1)(2)(3))

# 十六、异步并发数限制

/\*\*

\* 关键点

\* 1. new promise 一经创建，立即执行

\* 2. 使用 Promise.resolve().then 可以把任务加到微任务队列，防止立即执行迭代方法

\* 3. 微任务处理过程中，产生的新的微任务，会在同一事件循环内，追加到微任务队列里

\* 4. 使用 race 在某个任务完成时，继续添加任务，保持任务按照最大并发数进行执行

\* 5. 任务完成后，需要从 doingTasks 中移出

\*/

function limit(count, array, iterateFunc) {

const tasks = []

const doingTasks = []

let i = 0

const enqueue = () => {

if (i === array.length) {

return Promise.resolve()

}

const task = Promise.resolve().then(() => iterateFunc(array[i++]))

tasks.push(task)

const doing = task.then(() => doingTasks.splice(doingTasks.indexOf(doing), 1))

doingTasks.push(doing)

const res = doingTasks.length >= count ? Promise.race(doingTasks) : Promise.resolve()

return res.then(enqueue)

};

return enqueue().then(() => Promise.all(tasks))

}

// test

const timeout = i => new Promise(resolve => setTimeout(() => resolve(i), i))

limit(2, [1000, 1000, 1000, 1000], timeout).then((res) => {

console.log(res)

})

# 十七、异步串行/异步并行

// 字节面试题，实现一个异步加法

function asyncAdd(a, b, callback) {

setTimeout(function () {

callback(null, a + b);

}, 500);

}

// 解决方案

// 1. promisify

const promiseAdd = (a, b) => new Promise((resolve, reject) => {

asyncAdd(a, b, (err, res) => {

if (err) {

reject(err)

} else {

resolve(res)

}

})

})

// 2. 串行处理

async function serialSum(...args) {

return args.reduce((task, now) => task.then(res => promiseAdd(res, now)), Promise.resolve(0))

}

// 3. 并行处理

async function parallelSum(...args) {

if (args.length === 1) return args[0]

const tasks = []

for (let i = 0; i < args.length; i += 2) {

tasks.push(promiseAdd(args[i], args[i + 1] || 0))

}

const results = await Promise.all(tasks)

return parallelSum(...results)

}

// 测试

(async () => {

console.log('Running...');

const res1 = await serialSum(1, 2, 3, 4, 5, 8, 9, 10, 11, 12)

console.log(res1)

const res2 = await parallelSum(1, 2, 3, 4, 5, 8, 9, 10, 11, 12)

console.log(res2)

console.log('Done');

})()