**JS常见手写题**

目录

[一、防抖 2](#_Toc1532897693)

[二、节流 4](#_Toc50213192)

[三、new的模拟实现 5](#_Toc2119528320)

[四、bind 模拟实现 7](#_Toc453737804)

[五、call 模拟实现 11](#_Toc256841331)

[六、apply 模拟实现 13](#_Toc290119647)

[七、数组扁平化 14](#_Toc1253028439)

[八、对象扁平化 15](#_Toc1424331791)

[九、手写Promise 16](#_Toc744198228)

[十、手写发布/订阅 EventEmitter 18](#_Toc794857868)

[十一、instanceOf 实现 19](#_Toc1827903136)

[十二、深拷贝 20](#_Toc1814436417)

[十三、JS继承 22](#_Toc965073119)

[十四、图片懒加载 26](#_Toc39925242)

[十五、函数柯里化 29](#_Toc1008644430)

[十六、compose函数组合 31](#_Toc51025592)

[十七、字符串模版匹配 33](#_Toc741149591)

[十八、URL Params解析 34](#_Toc1096023337)

[十九、异步并发请求 35](#_Toc1896984640)

[二十、异步并发控制 38](#_Toc1078621118)

[异步串行、异步并行 40](#_Toc1475665899)

# 一、防抖

防抖：在一段连续的事件触发后，函数只能执行一次；若在n秒内又触发了事件，则会重新计算函数执行时间。防抖是从 “最后一次触发” 算等待时间的。

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) //定时器清除后，定时器中的回调函数就不会执行了

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); // 取消定时器，仅仅只是回调函数不执行了，并不影响timer值

}

var callNow = !timer; // 如果已经执行过，不再执行

timer = setTimeout(function(){

timer = null;

}, wait)

if (callNow) 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) {

// 判断是否可以立即执行（当前没有待执行的定时器）

const callNow = !timeout;

timeout = setTimeout(() => {

timeout = null;

}, wait);

// 如果可以立即执行，就执行函数

if (callNow) func.apply(context, args);

} else {

timeout = setTimeout(function () {

func.apply(context, args);

}, wait);

}

};

}

# 二、节流

节流：连续触发事件每隔 n 秒执行一次，会稀释函数的执行频率。从 “第一次触发” 开始算固定间隔（而“防抖”是从最后一次触发开始算等待时间）

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

第一次触发“立即执行”，之后按固定间隔执行。

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

第一次触发后“延迟 interval 执行”，之后按固定间隔执行。

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

// 取出构造函数（第一个参数），另外获取构造函数传入的参数

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

// 将新对象内部的 \_proto\_ 指针指向构造函数的prototype

obj.\_\_proto\_\_ = Constructor.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

// 依据返回值类型，分别返回

return (ret !== null && (typeof ret === 'object' || typeof ret === 'function')) ? ret : obj;

};

# 四、bind 模拟实现

bind 函数的三个特点：

（1）返回一个函数，这个函数内会调用原始函数 并绑定this

（2）可以预先传入部分参数，后续调用时只需补充剩余参数

（3）当 bind 返回的新函数被用作构造函数时，bind 绑定的 this 会被忽略，新函数的 this 指向构造函数自身的实例对象，但预设参数仍然有效。

**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));

}

// 正确的原型继承

var F = function () {};

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

fBound.prototype = new F();

~~// fBound.prototype = this.prototype; / /this为调用 bind 的函数, 这种写法有问题！！❌~~

return fBound;

}

详细代码解析：

1. 检测当前函数是否通过 new 调用（即作为构造函数使用）

```js

this instanceof fBound

```

1. 若被当做普通函数调用时，this 指向全局对象（非严格模式）或 undefined（严格模式），显然不在 fBound 的原型链上，`this instanceof fBound`结果为 false。
2. 若被当做构造函数使用 new 调用，this 指向新创建的实例，`this instanceof fBound`结果为 true。

2. 在处理构造函数的原型继承时：

（1）fBound.prototype = this.prototype; 中 fBound.prototype 和 Foo.prototype 指向同一个对象。如果后续修改 fBound.prototype（比如 fBound.prototype.bar = 1），会直接修改 Foo.prototype，污染原函数的原型链。

（2）我们的目标是“想实现原型继承，但不共享原型对象”，这里这里可以参考：[JS继承](../对象\_类\_面向对象/3.0\_JS继承.md) 中的方案4 “原型式继承” 或者方案6 “寄生组合式继承”

```js

const F = function () {};

F.prototype = this.prototype; // F 的原型指向原函数原型

fBound.prototype = new F(); // fBound 原型是 F 的实例

```

若上面方案比较繁琐的话，可以直接用 Object.create：

```js

fBound.prototype = Object.create(this.prototype);

```

原理：Object.create(proto) 会创建一个新对象，其 \_\_proto\_\_ 指向 proto（即原函数的原型）。

# 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。

// this指向“call的调用者”，call的调用者是个函数！

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：

// 这里用call不合适，因为就是为了模拟实现call的！

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

// context.fn(...args)

// 法2：

var args = [];

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

args.push('arguments[' + i + ']'); // args为：['arguments[1]', 'arguments[2]']

}

eval('context.fn(' + args +')'); // // 数组前加''，相当于调用arr.json(',')

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;

// 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

apply 的实现跟 call 类似。

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);

}

);

}

});

# 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

/\*\*

函数特点：

1. 处理的数据类型：基本数据类型、引用类型

2. 引用类型有：Date、Regexp、Function、Set、Map、Array、Object

3. 循环引用处理，避免爆栈：Set、Map、Array、Object

4. 对象深拷贝：

（1）创建新对象时保留其原型链

（2）拷贝的属性是实例自有属性（可枚举 & 不可枚举、字符串键 & Symbol键）

\*/

function deepClone(target, weakMap = new WeakMap()) {

// 六大基本数据类型

if (target === null || typeof target !== 'object') {

// if (!(target instanceof Object)) { // 语义不清晰

return target

}

// 处理循环引用

if (weakMap.has(target)) {

return weakMap.get(target);

}

// 日期对象

if (target instanceof Date) {

return new Date(target.getTime());

}

// 正则对象

if (target instanceof RegExp) {

return new RegExp(target.source, target.flags);

}

// Map：键和值分别进行拷贝

if (target instanceof Map) {

const clonedMap = new Map();

weakMap.set(target, clonedMap); // 处理循环引用

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

clonedMap.set(deepClone(key, weakMap), deepClone(value, weakMap));

});

return clonedMap;

}

// Set：每个元素进行拷贝

if (target instanceof Set) {

const clonedSet = new Set();

weakMap.set(target, clonedSet); // 处理循环引用

target.forEach(value => {

clonedSet.add(deepClone(value, weakMap));

});

return clonedSet;

}

// 函数：直接返回原函数即可，函数的深拷贝很难完美实现！

if (target instanceof Function) {

return target;

}

// 数组

if (Array.isArray(target)) {

const clonedArr = [];

weakMap.set(target, clonedArr);

// 用 for 循环或 forEach 遍历，保证顺序 并且 只处理索引元素

target.forEach(item => {

clonedArr.push(deepClone(item, weakMap));

});

return clonedArr;

}

// 创建新对象并保留原型链

const clonedObj = new target.constructor();

// 遍历实例自有属性（可枚举 & 不可枚举、字符串键 & Symbol键）

const ownKeys = Reflect.ownKeys(target);

ownKeys.forEach(key => {

clonedObj[key] = deepClone(target[key], weakMap);

});

weakMap.set(target, clonedObj);

return clonedObj

}

# 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。

与原型式继承的区别：

（1）原型式继承：仅仅创建一个与原对象关联的新对象，新对象可以共享原对象的属性和方法，但不会主动添加新功能。

（2）寄生式继承：在原型式继承的基础上，主动为新对象添加新属性或方法（即"增强"对象）。

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

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

其中，src="default.png"：页面首次加载时显示的 “占位图”，data-src="https://xxxx/real.png"：存储真实图片的地址。通过JS将data-src的值赋给src，触发真实图片加载。

可以给img标签统一自定义属性data-src='default.png'，当检测到图片出现在可视窗口之后再补充src属性，此时才会进行图片资源加载。提供四种判断方法：

法1：使用浏览器原生 loading="lazy" 属性 // 有兼容性问题

法2：“元素顶部到可视窗口顶部的距离”小于可视窗口的高度。

let {top, bottom} = element.getBoundingClientRect();

top < clientHeight && bottom > 0

法3：IntersectionObserver

法4：element.offsetTop <

document.documentElement.clientHeight + document.documentElement.scrollTop

元素距离"文档顶部"的距离 < 屏幕可视窗口高度 + 元素滚动的高度(即：浏览器窗口顶部 到 文档顶部之间的距离)。

注意：下面的这个方法 判断是否在视口内，不一定准的。因为element.offsetTop 是图片相对于其最近的已定位祖先元素的顶部距离（而非相对于页面顶部）。如果图片的父元素设置了 position: relative/absolute/fixed，offsetTop 的值就不是相对于页面顶部的，此时公式会完全失效。

**方法1：**

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、函数组合copmose 两个是函数式编程的基础，被广泛应用在 Redux、Lodash 等库中。

**柯里化**：将接受多个参数的函数 转成 一系列接受部分参数的函数链，参数收集够厚执行原函数。

**组合函数compose**：接收多个函数作为参数，并返回一个新的函数，新的函数在执行时从右到左依次调用这些函数。主要是为了将“函数嵌套”的写法转成 线性扁平化，提高可读性。

1. 什么是柯里化（ curry）？

定义：将接受 多个参数的一个函数 转换成 一系列 可分步接收部分参数 的函数链，当参数收集完整后，执行原函数。函数链中的函数每次接收一个或多个参数。

add (a, b, c) → curriedAdd(1)(2)(3)

add (a, b, c) → curriedAdd(1, 2)(3)

1. 柯里化用途

参数复用（固定部分参数）、延迟执行（分步接收参数）、函数组合的基础

**3. 如何实现 curry 函数？**

核心逻辑拆解：“收集参数 → 判断是否执行 → 递归收集”

那么我们如何确定何时达到足够的参数呢？有两种思路：

1. 通过函数的 length 属性，获取函数的形参个数，形参的个数就是函数所需的参数个数

(2) 在调用柯里化工具函数时，手动指定所需的参数个数

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) {

// 参数够了，直接执行原函数

return func.apply(this, args)

}

// 参数不够，返回新函数继续“收集后续参数”

return function (...args2) {

// 递归调用 curried，合并已有参数和新参数

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

}

}

}

// 测试

function add (a, b, c) {

return a + b + c

}

const curriedAdd = curry(add)

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

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

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

结合例子分析柯里化分步传参的过程

以 curriedAdd = curry(add)（add(a,b,c) 需 3 个参数）为例，看 curriedAdd(1)(2)(3) 的执行流程：

1. 第一次调用 curriedAdd(1)：

args = [1]，func.length = 3 → 1 >= 3 不成立。

返回新函数 f1（用于收集下一批参数）。

1. 第二次调用 f1(2)：

实际是调用 curried.apply(this, [1].concat([2])) → 即 curried(1, 2)。

args = [1, 2]，func.length = 3 → 2 >= 3 不成立。

返回新函数 f2（继续收集参数）。

1. 第三次调用 f2(3)：

实际是调用 curried.apply(this, [1,2].concat([3])) → 即 curried(1,2,3)。

args = [1,2,3]，func.length = 3 → 3 >= 3 成立。

执行 add.apply(this, [1,2,3]) → 返回 6。

# 十六、compose函数组合

接收多个函数作为参数，并返回一个新的函数。该函数在执行时，会从右到左依次调用这些函数，前一个函数的输出作为后一个函数的输入，最终返回最后一个函数的输出。函数组合 (compose) 是一种函数式编程模式。

没有 compose 时，多层函数调用可能写成嵌套形式：

// 嵌套调用：可读性差，嵌套层级深时易混乱

fn1(fn2(fn3(x)));

有 compose 时，代码更扁平、直观：

// 扁平化地组合成一个函数：执行顺序一目了然（fn3 → fn2 → fn1）

const composed = compose(fn1, fn2, fn3);

composed(x); // 等价于 fn1(fn2(fn3(x)))

compose 的价值，在于更高效地组合单一职责的函数，同时解决“手动组合时的代码冗余、可读性差” 问题。

compose 与 函数嵌套 的关系：compose 并没有消除函数嵌套的执行逻辑（最终还是要嵌套执行），但它将嵌套的代码写法从 “手动多层嵌套” 变成了 “线性扁平组合”。

**如何实现 compose？**

// 基础函数组合（从右到左执行）

function compose(...fns) {

// 如果没有传入函数，返回一个返回自身的函数

if (fns.length === 0) {

return (arg) => arg;

}

// 如果只有一个函数，直接返回该函数

if (fns.length === 1) {

return fns[0];

}

// 组合多个函数，从右到左执行

return fns.reduce(

(pre, cur) => (...args) => pre(cur(...args))

// 第一个args是剩余参数rest，会将参数转为数组args；第二个args是扩展参数，调用的时候将数组中的各项分别展开来调用

);

}

使用示例：

function toUpper(str) {

return str.toUpperCase();

}

function reverse(str) {

return str.split('').reverse().join('');

}

function addPrefix(str) {

return `Prefix: ${str}`;

}

// 组合函数：先反转，再转大写，最后添加前缀

const processStr = compose(addPrefix, toUpper, reverse);

console.log(processStr('hello')); // "Prefix: OLLEH"

// 执行顺序：reverse("hello") → toUpper("olleh") → addPrefix("OLLEH")

# 十七、字符串模版匹配

let template = '我是{{name}}，年龄{{age}}，性别{{sex}}';

let person = {

name: '布兰',

age: 12

}

用真实数据替换后，转成“我是布兰，年龄12，性别undefined”

**如何实现：**

function render2(template, data) {

const reg = /\{\{(\w+)\}\}/g;

let regResult

let finalStr = template

while ((regResult = reg.exec(template)) !== null) {

// regResult[0] // 完整匹配的字符串

// regResult[1] // 捕获组捕获到的内容

// 替换当前匹配项

finalStr = finalStr.replace(regResult[0], data[regResult[1]])

}

return finalStr

}

render(template, person);

# 十八、URL Params解析

function parseParam(url) {

// const arr = url.match(/\?([^#.]+)(\#.\*)?/) //两者是等价的

const arr = /\?([^#.]+)(\#.\*)?/.exec(url)

if (arr === null) {

return ''

}

const paramsStr = arr[1];

const paramsArr = paramsStr.split('&');

let paramsObj = {};

paramsArr.forEach(param => {

if (/=/.test(param)) { // 处理有 value 的参数

let [key, val] = param.split('=');

val = decodeURIComponent(val);

if (paramsObj.hasOwnProperty(key)) {

paramsObj[key] = [].concat(paramsObj[key], val);

} else {

paramsObj[key] = val;

}

} else {

paramsObj[param] = true; // 处理没有 value 的参数，默认设为true

}

})

return paramsObj;

}

使用：

let url = 'http://www.domain.com/?user=anonymous&id=123&id=456&city=%E5%8C%97%E4%BA%AC&enabled';

parseParam(url)

/\* 结果

{ user: 'anonymous',

id: [ 123, 456 ], // 重复出现的 key 要组装成数组，能被转成数字的就转成数字类型

city: '北京', // 中文需解码

enabled: true, // 未指定值得 key 约定为 true

}

\*/

# 十九、异步并发请求

大文件分片上传：

所有的分片都成功，相当于大文件才上传成功了；

但凡其中有一个分片失败了，这个大文件就上传失败了，后面的分片也没必要上传了！

**sizeMap[分片id]**：请求进度获取，axios中onUploadProgress实现。大文件的上传进度需将所有上传成功+正在上传的分片数据量累加

**req\_result**：每个分片请求的结果，成功/失败 {status:true/fasle}

count：执行过的请求数，finally中统计，请求是不管成功/失败

j：当前正在执行的请求的索引，（请求前记录，和count略有不同）

async function concurrentRequest(onUploadProgress: Function) {

return new Promise((resolve) => {

let count = 0;

let j = -1;

let req\_result: Array<reqResult> = [];

let sizeMap: any = {};

const request = async () => {

let idx = ++j;

// 所有请求均已发送

if (idx >= this.fileChunkList.length) return;

axios({

method: "put",

url: this.multipartUploadUrls[idx].signedUrl,

data: this.fileChunkList[idx].file,

signal: this.control.signal,

onUploadProgress: (progress) => {

let uploadSize = 0;

sizeMap[idx] = progress.loaded;

for (let key in sizeMap) {

uploadSize += sizeMap[key];

}

if (!this.isCancel)

onUploadProgress(uploadSize + this.uploadedSize);

},

})

.then(() => {

console.log("单个分片success! ", this.fileChunkList[idx].idx);

req\_result[idx] = { status: true };

})

.catch((e) => {

console.error("单个分片失败!", this.fileChunkList[idx].idx);

req\_result[idx] = { status: false };

})

.finally(() => {

count++;

let flag = false;

for (let i = 0; i < req\_result.length; i++) {

// 只要有失败的, 还在进行的请求取消掉 返回error

const status = req\_result[i]?.status;

if (typeof status == "boolean" && !status) {

flag = true;

return resolve(false); // concurrentRequest 异步返回结果

}

}

// 所有分片请求已发送完毕

if (count == this.fileChunkList.length) {

const isError = req\_result.find((req) => !req.status);

return resolve(!isError);

}

if (!flag) {

request();

}

});

};

const times = Math.min(this.fileChunkList.length, MAXNUM);

for (let i = 0; i < times; i++) {

request();

}

});

}

this.concurrentRequest(onUploadProgress).then(async (res) => {

console.warn("多路并发是否已全部上传：", res);

if (!res) {

onUploadFinish && (await onUploadFinish(false));

} else {

const res = await this.completeMultipartUpload(args);

console.warn("completeMultipartUpload: ", res);

if (res) {

onUploadFinish && (await onUploadFinish(true));

}

else {

onUploadError && onUploadError();

if (this.uploadId) {

await this.abortMultipartUpload({

uploadId: this.uploadId,

storageType: this.storageType,

path: this.path,

});

}

}

}

});

# 二十、异步并发控制

asyncPool 函数的作用是：限制同时执行的 timeout 任务数不超过 limit（这里是 2）。当任务数达到上限时，用 await Promise.race(executing) 等待 “任意一个任务先完成”，再补充新任务，最终用 Promise.all(results) 等待所有任务完成。

const beforTime = Date.now();

const timeout = delay =>

new Promise(resolve => {

setTimeout(() => {

console.log(delay, ' ', Date.now() - beforTime);

resolve(delay);

}, delay);

});

async function asyncPool(limit, arr) {

const results = [];

const executing = [];

for (const delay of arr) {

const p = timeout(delay);

results.push(p);

const e = p.then(() => executing.splice(executing.indexOf(e), 1));

executing.push(e);

if (executing.length >= limit) {

await Promise.race(executing);

}

}

return Promise.all(results);

}

asyncPool(2, [1000, 3000, 2000]).then(results => {

console.log('并发控制后，所有任务完成：', results);

});

asyncPool内部循环过程：

for循环：

1. 第一次循环，处理任务 1000：同步执行，创建p1（延迟 1000ms 的 Promise）、e1（获取p1的完成状态）

2. 第二次循环，处理任务 3000：

同步执行，创建p2（延迟 3000ms 的 Promise）、e2（获取p2的完成状态）

触发 await Promise.race([e1, e2])：暂停 asyncPool 函数执行，等待 e1 或 e2 先完成。

3. 同步任务卡柱了，微任务执行

p1（延迟 1000ms）先完成，它会导致两处的变化：e1.then() 与 Promise.race([e1, e2])

4. 第三次循环，处理任务 2000：

同步执行，创建p3（延迟 2000ms 的 Promise）、e3（获取p3的完成状态）

触发 await Promise.race([e2, e3])：暂停 asyncPool 执行，等待 e2 或 e3 先完成。

5. 假设 p3（延迟 2000ms）比 p2（延迟 3000ms）先完成：....参考上面被卡住的流程

6. for循环结束后，执行 return Promise.all(results)

当 Promise.all(results) 完成后，外层 then 回调执行，打印console

# 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');

})()